

Contents

[Windows Mixed Reality](#)

[Install the tools](#)

[Fundamentals](#)

[What is mixed reality?](#)

[What is a hologram?](#)

[Hardware](#)

[HoloLens hardware details](#)

[Immersive headset hardware details](#)

[Hardware accessories](#)

[Core building blocks](#)

[Gaze](#)

[Gestures](#)

[Motion controllers](#)

[Voice input](#)

[Spatial mapping](#)

[Spatial sound](#)

[Coordinate systems](#)

[Spatial anchors](#)

[Apps in mixed reality](#)

[Types of mixed reality apps](#)

[App model](#)

[App views](#)

[Academy](#)

[MR Basics 100: Getting started with Unity](#)

[MR Basics 101: Complete project with device](#)

[MR Basics 101E: Complete project with emulator](#)

[MR Input 210: Gaze](#)

[MR Input 211: Gesture](#)

[MR Input 212: Voice](#)

MR Input 213: Motion controllers
MR Spatial 220: Spatial sound
MR Spatial 230: Spatial mapping
MR Sharing 240: Multiple HoloLens devices
MR Sharing 250: HoloLens and immersive headsets
MR and Azure 301: Language translation
MR and Azure 302: Computer vision
MR and Azure 302b: Custom vision
MR and Azure 303: Natural language understanding
MR and Azure 304: Face recognition
MR and Azure 305: Functions and storage
MR and Azure 306: Streaming video
MR and Azure 307: Machine learning
MR and Azure 308: Cross-device notifications
MR and Azure 309: Application insights
MR and Azure 310: Object detection
MR and Azure 311: Microsoft Graph
MR and Azure 312: Bot integration
MR and Azure 313: IoT Hub Service

Design

About this design guidance
Get started with design
What is mixed reality?
My first year on the design team
Expanding the design process for mixed reality
The pursuit of more personal computing
AfterNow's process - envisioning, prototyping, building

Interaction design

Interaction fundamentals
Comfort
Gaze targeting
Gestures

Voice design

What is a hologram?

Holographic frame

Spatial mapping design

Spatial sound design

Motion controllers

Style

Color, light and materials

Spatial sound design

Typography

Scale

Case study - Spatial sound design for HoloTour

App patterns

Types of mixed reality apps

Room scan visualization

Cursors

Billboarding and tag-along

Updating 2D UWP apps for mixed reality

Controls

Text in Unity

Interactable object

Object collection

Displaying progress

App bar and bounding box

Designing for the mixed reality home

3D app launcher design guidance

Create 3D models for use in the home

Add custom home environments

Sample apps

Periodic Table of the Elements

Lunar Module

Galaxy Explorer

Design tools and resources

HoloSketch

Asset creation process

Development

Development overview

Unity

Unity development overview

Getting started

Recommended settings for Unity

Unity Play Mode

Exporting and building a Unity Visual Studio solution

Best practices for working with Unity and Visual Studio

Core building blocks

Camera in Unity

Coordinate systems in Unity

Persistence in Unity

Gaze in Unity

Gestures and motion controllers in Unity

Voice input in Unity

Spatial mapping in Unity

Spatial sound in Unity

Other key features

Shared experiences in Unity

Locatable camera in Unity

Focus point in Unity

Tracking loss in Unity

Keyboard input in Unity

Advanced topics

Using the Windows namespace with Unity apps for HoloLens

Using Vuforia with Unity

DirectX

DirectX development overview

Getting started

[Creating a holographic DirectX project](#)

[Getting a HolographicSpace](#)

[Rendering in DirectX](#)

Core building blocks

[Coordinate systems in DirectX](#)

[Gaze, gestures, and motion controllers in DirectX](#)

[Voice input in DirectX](#)

[Spatial sound in DirectX](#)

[Spatial mapping in DirectX](#)

Other key features

[Shared spatial anchors in DirectX](#)

[Locatable camera in DirectX](#)

[Keyboard, mouse, and controller input in DirectX](#)

Advanced topics

[Using XAML with holographic DirectX apps](#)

[Add holographic remoting](#)

WebVR

Using WebVR in Edge with Windows Mixed Reality

Porting apps

Porting guides

[Input porting guide for Unity](#)

[Updating your SteamVR application for Windows Mixed Reality](#)

[Updating 2D UWP apps for mixed reality](#)

Rendering and performance

[Holographic rendering](#)

[Volume rendering](#)

[Hologram stability](#)

[Performance recommendations for HoloLens apps](#)

[Performance recommendations for immersive headset apps](#)

[Performance recommendations for Unity](#)

Testing your app

- [Using Visual Studio to deploy and debug](#)
- [Testing your app on HoloLens](#)
- [Using the HoloLens emulator](#)
- [Using the Windows Mixed Reality simulator](#)
- [Advanced HoloLens Emulator and Mixed Reality Simulator input](#)
- [HoloLens emulator archive](#)
- [Perception simulation](#)
- [Finishing your app](#)
 - [Implement 3D app launchers \(UWP apps\)](#)
 - [Implement 3D app launchers \(Win32 apps\)](#)
 - [Enable placement of 3D models in the home](#)
 - [In-app purchases](#)
 - [Submitting an app to the Microsoft Store](#)
- [Remote tools](#)
 - [Using the Windows Device Portal](#)
 - [Device portal API reference](#)
 - [Holographic Remoting Player](#)
- [Other](#)
 - [QR code tracking](#)
 - [Shared experiences in mixed reality](#)
 - [Locatable camera](#)
 - [Mixed reality capture for developers](#)
 - [HoloLens Research mode](#)
- [Open source projects](#)
 - [Galaxy Explorer](#)
 - [Periodic Table of the Elements](#)
 - [Lunar Module](#)
- [Case studies](#)
 - [AfterNow's process - envisioning, prototyping, building](#)
 - [Building HoloSketch, a spatial layout and UX sketching app for HoloLens](#)
 - [Capturing and creating content for HoloTour](#)
 - [Creating a galaxy in mixed reality](#)

[Creating an immersive experience in Fragments](#)
[Creating impossible perspectives for HoloTour](#)
[Expanding the design process for mixed reality](#)
[Expanding the spatial mapping capabilities of HoloLens](#)
[HoloStudio UI and interaction design learnings](#)
[Lessons from the Lowe's kitchen](#)
[Looking through holes in your reality](#)
[My first year on the HoloLens design team](#)
[Representing humans in mixed reality](#)
[Scaling Datascape across devices with different performance](#)
[Spatial sound design for HoloTour](#)
[The pursuit of more personal computing](#)
[Using spatial sound in RoboRaid](#)
[Using the stabilization plane to reduce holographic turbulence](#)

Resources

[San Francisco Reactor Academy events calendar](#)
[CVPR 2018 HoloLens Research mode session](#)
[Mixed Reality Partner Program](#)
[App quality criteria](#)
[Spectator view](#)
[How it works - Mixed Reality Capture Studios](#)
[Holographic remoting software license terms](#)
[Mixed reality release notes](#)
[Current release notes](#)
[Release notes - April 2018](#)
[Release notes - October 2017](#)
[Release notes - August 2016](#)
[Release notes - May 2016](#)
[Release notes - March 2016](#)
[HoloLens how-to](#)
[Accounts on HoloLens](#)
[Calibration](#)

[Commercial features](#)

[Connecting to Wi-Fi on HoloLens](#)

[Environment considerations for HoloLens](#)

[Get apps for HoloLens](#)

[Give us feedback](#)

[HoloLens known issues](#)

[HoloLens troubleshooting](#)

[Mixed reality capture](#)

[Navigating the Windows Mixed Reality home](#)

[Reset or recover your HoloLens](#)

[Saving and finding your files](#)

[See your photos](#)

[Sensor tuning](#)

[Updating HoloLens](#)

[Contributing to this documentation](#)

Mixed reality blends real-world and virtual content into hybrid environments where physical and digital objects coexist and interact. Learn to build mixed reality experiences for Microsoft HoloLens and Windows Mixed Reality immersive headsets (VR).



Fundamentals

Get started with mixed reality key concepts, core building blocks, and app paradigms.



Academy

See code examples, do a coding tutorial and watch guest lectures.



Design

Get design guidance, build user interface and learn interaction and input.



Development

Get development guides, learn the technology and understand the science.

Resources



Open source projects

These open source projects, sample apps, and toolkits should help you accelerate development of applications targeting Microsoft HoloLens and Windows Mixed Reality immersive headsets. Leverage anything you find here and please contribute back as you learn - our whole community will benefit!



Immersive headset Enthusiast's Guide

We know you might be looking to dive deeper on Windows Mixed Reality and learn how to get the most of your new headset and motion controllers, so we created the Enthusiast's Guide to provide you with exclusive information and answer the top questions people have about Windows Mixed Reality before and after they buy.

IMPORTANT

All Windows Mixed Reality development materials are provided on this site for your reference only. Your app, its usage, and its effect on end users is your sole responsibility as the app developer, including ensuring that your app does not cause discomfort, injury or any other harm to an end user, and including appropriate warnings and disclaimers. You need to at all times take the appropriate steps in the development and publishing of your app to ensure that your app is safe and you meet all obligations in your [App Developer Agreement with Microsoft](#).

Install the tools

11/6/2018 • 5 minutes to read • [Edit Online](#)

Get the tools you need to build apps for Microsoft HoloLens and Windows Mixed Reality immersive (VR) headsets. There is no separate SDK for Windows Mixed Reality development; you'll use Visual Studio with the Windows 10 SDK.

Don't have a mixed reality device? You can install the [HoloLens emulator](#) to test some functionality of mixed reality apps without a HoloLens. You can also use the [Windows Mixed Reality simulator](#) to test your mixed reality apps for immersive headsets.

We recommend installing the Unity game engine as the easiest way to get started creating mixed reality apps, however, you can also build against DirectX if you'd like to use a custom engine.

TIP

Bookmark this page and check it regularly to keep up-to-date on the most recent version of each tool recommended for mixed reality development.

Installation checklist

TOOL	DESCRIPTION	NOTES
 Windows 10 (Manual install link)	Install the most recent version of Windows 10 so your PC's operating system matches the platform for which you're building mixed reality apps.	Installing Windows 10 <ul style="list-style-type: none">You can install the most recent version of Windows 10 via Windows Update in Settings or by creating installation media (using the link in the left column).See current release notes for information about the newest mixed reality features available with each release of Windows 10. Enable developer mode on your PC at Settings > Update & Security > For developers. Note for enterprise and corporate-managed PCs: if your PC is managed by an organization's IT department, you may need to contact them in order to update. 'N' versions of Windows: Windows Mixed Reality immersive (VR) headsets are not supported on 'N' versions of Windows.

Tool	Description	Notes
 Visual Studio 2017 (Install link)	<p>Fully-featured integrated development environment (IDE) for Windows and more. You'll use Visual Studio to write code, debug, test, and deploy.</p>	<p>Additional workloads to install:</p> <ul style="list-style-type: none"> Universal Windows Platform development <p>Note about Unity: unless you're intentionally trying to install a newer (non-LTS) version of Unity for a specific purpose, we recommend <i>not</i> installing the Unity workload as part of Visual Studio installation, and instead installing the LTS stream of Unity as noted below.</p>
 Windows 10 SDK (Manual install link)	<p>Provides the latest headers, libraries, metadata, and tools for building Windows 10 apps.</p>	<p>The Windows 10 SDK is included when you install Visual Studio. You can also download and install the latest version of the SDK using the link in the left column.</p>
 Unity long term support (LTS) version (Install link)	<p>The Unity game engine is the easiest way to create mixed reality experiences, with built-in support for Windows Mixed Reality features.</p>	<p>We recommend the Unity LTS stream as the best version with which to start new projects, and migrate forward to, in order to pick up the latest stability fixes. It is also the version the current Mixed Reality Toolkit (MRTK) supports.</p> <p>Some developers may want to use a different version of Unity for specific reasons (like using a preview version of MRTK). For those cases, Unity supports side-by-side installs of different versions.</p>
 Mixed Reality Toolkit (MRTK) for Unity (GitHub repo link)	<p>The MRTK is a collection of scripts and components intended to accelerate development of applications targeting Microsoft HoloLens and Windows Mixed Reality immersive (VR) headsets. The project is aimed at reducing barriers to entry to create mixed reality applications and contribute back to the community as we all grow.</p>	<p>Visit the MRTK GitHub repo (link in the left column) to learn more.</p>

Mixed Reality Toolkit

The Mixed Reality Toolkit is a collection of scripts and components intended to accelerate development of applications targeting Microsoft HoloLens and Windows Mixed Reality headsets. The project is aimed at reducing barriers to entry to create mixed reality applications and contribute back to the community as we all grow.

- [MixedRealityToolkit](#)
- [MixedRealityToolkit-Unity](#) - uses code from the base toolkit and makes it easier to consume in Unity.
- [MixedRealityCompanionKit](#) - code bits and components that may not run directly on HoloLens or

immersive (VR) headsets, but instead pair with them to build experiences targeting Windows Mixed Reality.

Setting up your PC for mixed reality development

The Windows 10 SDK works best on the Windows 10 operating system. This SDK is also supported on Windows 8.1, Windows 8, Windows 7, Windows Server 2012, Windows Server 2008 R2. Note that not all tools are supported on older operating systems.

For HoloLens development

When setting up your development PC for HoloLens development, please make sure it meets the system requirements for both [Unity](#) and [Visual Studio](#). If you plan to use the HoloLens emulator, you'll want to make sure your PC meets the [HoloLens emulator system requirements](#) as well.

If you plan to develop for both HoloLens and Windows Mixed Reality immersive (VR) headsets, please use the system recommendations and requirements in the section below.

For immersive (VR) headset development

NOTE

The following guidelines are the current minimum and recommended specs for your immersive (VR) headset *development PC*, and may be updated regularly.

WARNING

Do not confuse this with the [minimum PC hardware compatibility guidelines](#), which outlines the *consumer PC specs* to which you should target your immersive (VR) headset app or game.

If your immersive headset development PC does not have full-sized HDMI and/or USB 3.0 ports, you'll need [adapters](#) to connect your headset.

There are currently [known issues](#) with some hardware configurations, particularly with notebooks that have hybrid graphics.

	MINIMUM	RECOMMENDED
Processor	Notebook: Intel Mobile Core i5 7th generation CPU, Dual-Core with Hyper Threading Desktop: Intel Desktop i5 6th generation CPU, Dual-Core with Hyper Threading OR AMD FX4350 4.2Ghz Quad-Core equivalent	Desktop: Intel Desktop i7 6th generation (6 Core) OR AMD Ryzen 5 1600 (6 Core, 12 threads)
GPU	Notebook: NVIDIA GTX 965M, AMD RX 460M (2GB) equivalent or greater DX12 capable GPU Desktop: NVIDIA GTX 960/1050, AMD Radeon RX 460 (2GB) equivalent or greater DX12 capable GPU	Desktop: NVIDIA GTX 980/1060, AMD Radeon RX 480 (2GB) equivalent or greater DX12 capable GPU
GPU driver WDDM version	WDDM 2.2 driver	
Thermal Design Power	15W or greater	

Graphics display ports	1x available graphics display port for headset (HDMI 1.4 or DisplayPort 1.2 for 60Hz headsets, HDMI 2.0 or DisplayPort 1.2 for 90Hz headsets)	
Display resolution	Resolution: SVGA (800x600) or greater Bit depth: 32 bits of color per pixel	
Memory	8 GB of RAM or greater	16 GB of RAM or greater
Storage	>10 GB additional free space	
USB Ports	1x available USB port for headset (USB 3.0 Type-A) Note: USB must supply a minimum of 900mA	
Bluetooth	Bluetooth 4.0 (for accessory connectivity)	

See also

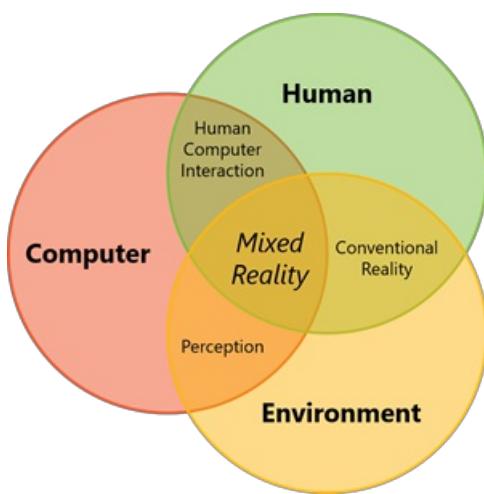
- [Development overview](#)
- [Using the HoloLens emulator](#)
- [Using the Windows Mixed Reality simulator](#)
- [Unity development overview](#)
- [DirectX development overview](#)
- [HoloLens emulator archive](#)

What is mixed reality?

11/6/2018 • 6 minutes to read • [Edit Online](#)

Mixed reality is the result of blending the physical world with the digital world. Mixed reality is the next evolution in human, computer, and environment interaction and unlocks possibilities that before now were restricted to our imaginations. It is made possible by advancements in computer vision, graphical processing power, display technology, and input systems. The term *mixed reality* was originally introduced in a 1994 paper by Paul Milgram and Fumio Kishino, "[A Taxonomy of Mixed Reality Visual Displays](#)." Their paper introduced the concept of the *virtuality continuum* and focused on how the categorization of taxonomy applied to displays. Since then, the application of mixed reality goes beyond displays but also includes environmental input, spatial sound, and location.

Environmental input and perception



Over the past several decades, the relationship between human input and computer input has been well explored. It even has a widely studied discipline known as *human computer interaction* or HCI. Human input happens through a variety of means including keyboards, mice, touch, ink, voice, and even Kinect skeletal tracking.

Advancements in sensors and processing are giving rise to a new area of computer input from environments. The interaction between computers and environments is effectively environmental understanding, or *perception*. Hence the API names in Windows that reveal environmental information are called the [perception APIs](#). Environmental input captures things like a person's position in the world (e.g. [head tracking](#)), surfaces and boundaries (e.g. [spatial mapping](#) and [spatial understanding](#)), ambient lighting, environmental sound, object recognition, and location.

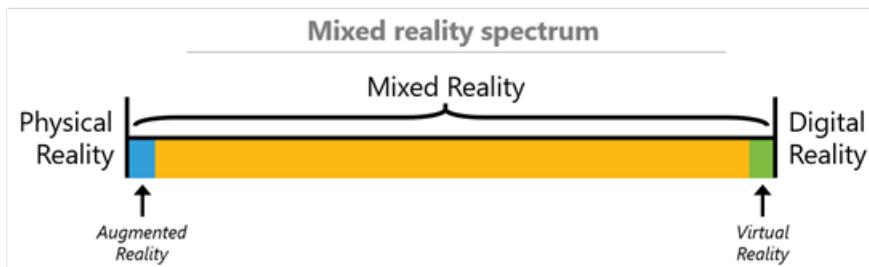
Now, the combination of all three – computer processing, human input, and environmental input – sets the opportunity to create true mixed reality experiences. Movement through the physical world can translate to movement in the digital world. Boundaries in the physical world can influence application experiences, such as game play, in the digital world. Without environmental input, experiences cannot blend between the physical and digital realities.

The mixed reality spectrum

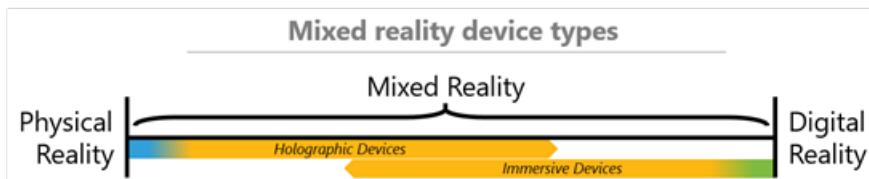
Since mixed reality is the blending of the physical world and digital world, these two realities define the polar ends of a spectrum known as the *virtuality continuum*. For simplicity, we refer to this as the *mixed reality spectrum*. On the left-hand side we have physical reality in which we, humans, exist. Then on the right-hand side we have the corresponding digital reality.

Most mobile phones on the market today have little to no environmental understanding capabilities. Thus the experiences they offer cannot mix between physical and digital realities. The experiences that overlay graphics on video streams of the physical world are *augmented reality*, and the experiences that occlude your view to present a digital experience are *virtual reality*. As you can see, the experiences enabled between these two extremes is *mixed reality*:

- Starting with the physical world, placing a digital object, such as a hologram, as if it was really there.
- Starting with the physical world, a digital representation of another person – an avatar – shows the location where they were standing when leaving notes. In other words, experiences that represent asynchronous collaboration at different points in time.
- Starting with a digital world, physical boundaries from the physical world, such as walls and furniture, appear digitally within the experience to help users avoid physical objects.



Most augmented reality and virtual reality offerings available today represent a very small part of this spectrum. They are, however, subsets of the larger mixed reality spectrum. Windows 10 is built with the entire spectrum in mind, and allows blending digital representations of people, places and things with the real world.



There are two main types of devices that deliver Windows Mixed Reality experiences:

1. **Holographic devices.** These are characterized by the device's ability to place digital content in the real world as if it were really there.
2. **Immersive devices.** These are characterized by the device's ability to create a sense of "presence" – hiding the physical world and replacing it with a digital experience.

CHARACTERISTIC	HOLOGRAPHIC DEVICES	IMMERSIVE DEVICES
Example Device	Microsoft HoloLens 	Acer Windows Mixed Reality Development Edition
Display	<i>See-through display.</i> Allows user to see physical environment while wearing the headset.	<i>Opaque display.</i> Blocks out the physical environment while wearing the headset.

Movement	Full six-degrees-of-freedom movement, both rotation and translation.	Full six-degrees-of-freedom movement, both rotation and translation.
----------	--	--

Note, whether a device is connected to or tethered to a separate PC (via USB cable or Wi-Fi) or self-contained (untethered) does not reflect whether a device is holographic or immersive. Certainly, features that improve mobility lead to better experiences and both holographic and immersive devices could be tethered or untethered.

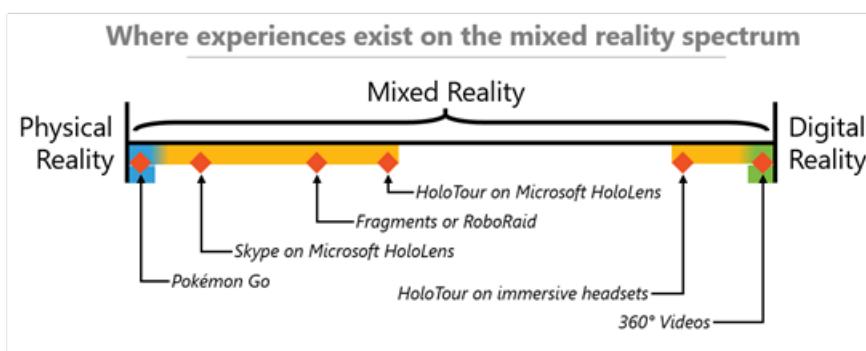
Devices and experiences

Technological advancement is what has enabled mixed reality experiences. There are no devices today that can run experiences across the entire spectrum; however, Windows 10 provides a common mixed reality platform for both device manufacturers and developers. Devices today can support a specific range within the mixed reality spectrum, and over time new devices should expand that range. In the future, holographic devices will become more immersive, and immersive devices will become more holographic.



Often, it is best to think what type of experience an app or game developer wants to create. The experiences will typically target a specific point or part on the spectrum. Then, developers should consider the capabilities of devices they want to target. For example, experiences that rely on the physical world will run best on HoloLens.

- **Towards the left (near physical reality).** Users remain present in their physical environment and are never made to believe they have left that environment.
- **In the middle (fully mixed reality).** These experiences perfectly blend the real world and the digital world. Viewers who have seen the movie *Jumanji* can reconcile how the physical structure of the house where the story took place was blended with a jungle environment.
- **Towards the right (near digital reality).** Users experience a completely digital environment and are oblivious to what occurs in the physical environment around them.



Here's how different experiences take advantage of their position on the mixed reality spectrum:

- **Skype on Microsoft HoloLens.** This experience allows collaboration through drawing in someone's physical environment. As an experience, it is currently further left on the spectrum because the physical environment remains the location of activity.
- **Fragments and RoboRaid.** Both of these take advantage of the layout of the user's physical environment, walls, floors, furniture to place digital content in the world. The experience moves further to the right on the spectrum, but the user always believes they are in their physical world.
- **HoloTour on Microsoft HoloLens.** HoloTour is designed with an immersive experience in mind. Users are meant to walk around tourist locations. On HoloLens, HoloTour pushes the limits of the device's immersive capabilities.

- **HoloTour on immersive devices.** Meanwhile when HoloTour runs on an immersive device, it showcases the environmental input by allowing users to walk around the tourist location. The boundaries that help users avoid walking into walls represent further capabilities that pull the experience towards the middle.
- **360° videos.** Since environmental input like translational movement does not affect the video playback experience, these experiences fall to the far right towards digital reality, effectively fitting into the narrow part of the spectrum that is virtual reality.

Skype for HoloLens, Fragments and RoboRaid are best experienced with HoloLens. Likewise, 360° video is best experienced on immersive devices. HoloTour showcases the best of what both types of devices can do today when trying to push towards the center experience of mixed reality.

See also

- [API Reference: Windows.Perception](#)
- [API Reference: Windows.Perception.Spatial](#)
- [API Reference: Windows.Perception.Spatial.Surfaces](#)

What is a hologram?

11/6/2018 • 4 minutes to read • [Edit Online](#)

HoloLens lets you create **holograms**, objects made of light and sound that appear in the world around you, just as if they were real objects. Holograms respond to your [gaze](#), [gestures](#) and [voice commands](#), and can interact with [real-world surfaces](#) around you. With holograms, you can create digital objects that are part of your world.

Device support

FEATURE	HOLOLENS	IMMERSIVE HEADSETS
Holograms	✓ <input type="checkbox"/>	

A hologram is made of light and sound

The holograms that HoloLens [renders](#) appear in the holographic frame directly in front of the user's eyes. Holograms add light to your world, which means that you see both the light from the display and the light from your surroundings. HoloLens doesn't remove light from your eyes, so holograms can't be rendered with the color black. Instead, black content appears as transparent.

Holograms can have many different appearances and behaviors. Some are realistic and solid, and others are cartoonish and ethereal. Holograms can highlight features in your surroundings, and they can be elements in your app's user interface.



Holograms can also make [sounds](#), which will appear to come from a specific place in your surroundings. On HoloLens, sound comes from two speakers that are located directly above your ears, without covering them. Similar to the displays, the speakers are additive, introducing new sounds without blocking the sounds from your environment.

A hologram can be placed in the world or tag along with you

When you have a particular location where you want a hologram, you can [place](#) it precisely there in the world. As you walk around that hologram, it will appear stable relative to the world around you. If you use a [spatial anchor](#) to pin that object firmly to the world, the system can even remember where you left it when you come back later.



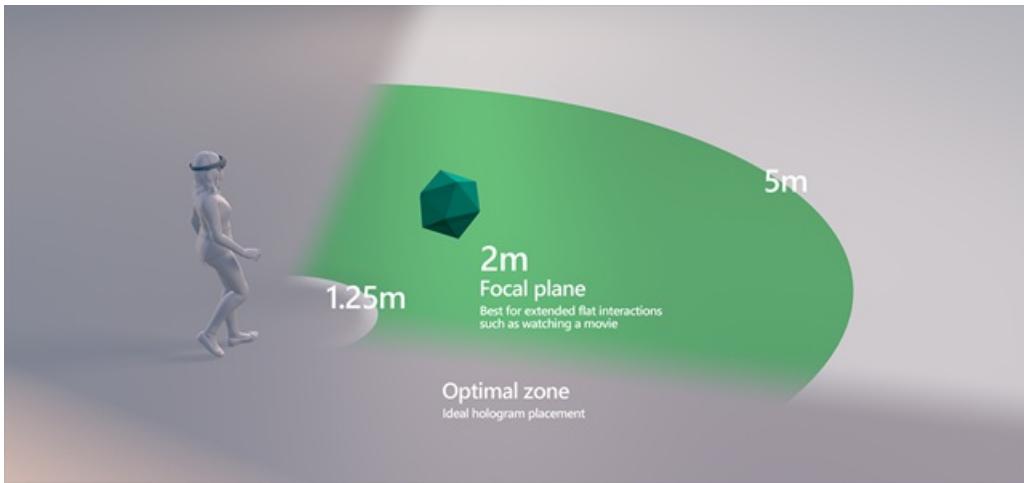
Some holograms follow the user instead. These tag-along holograms position themselves relative to the user, no matter where they walk. You may even choose to bring a hologram with you for a while and then place it on the wall once you get to another room.

Best practices

- Some scenarios may demand that holograms remain easily discoverable or visible throughout the experience. There are two high-level approaches to this kind of positioning. Let's call them "**display-locked**" and "**body-locked**".
 - Display-locked content is positionally "locked" to the device display. This is tricky for a number of reasons, including an unnatural feeling of "clingyness" that makes many users frustrated and wanting to "shake it off." In general, many designers have found it better to avoid display-locking content.
 - The body-locked approach is far more forgivable. Body-locking is when a hologram is tethered to the user's body or gaze vector, but is positioned in 3d space around the user. Many experiences have adopted a body-locking behavior where the hologram "follows" the user's gaze, which allows the user to rotate their body and move through space without losing the hologram. Incorporating a delay helps the hologram movement feel more natural. For example, some core UI of the Windows Holographic OS uses a variation on body-locking that follows the user's gaze with a gentle, elastic-like delay while the user turns their head.
- Place the hologram at a comfortable viewing distance typically about 1-2 meters away from the head.
- Provide an amount of drift for elements that must be continually in the holographic frame, or consider animating your content to one side of the display when the user changes their point of view.

Place holograms in the optimal zone - between 1.25m and 5m

Two meters is the most optimal, and the experience will degrade the closer you get from one meter. At distances nearer than one meter, holograms that regularly move in depth are more likely to be problematic than stationary holograms. Consider gracefully clipping or fading out your content when it gets too close so as not to jar the user into an unexpected experience.



A hologram interacts with you and your world

Holograms aren't only about light and sound; they're also an active part of your world. Gaze at a hologram and gesture with your hand, and a hologram can start to follow you. Give a voice command to a hologram, and it can reply.



Holograms enable personal interactions that aren't possible elsewhere. Because the HoloLens knows where it is in the world, a holographic character can look you directly in the eyes as you walk around the room.

A hologram can also interact with your surroundings. For example, you can place a holographic bouncing ball above a table. Then, with an [air tap](#), watch the ball bounce and make sound when it hits the table.

Holograms can also be occluded by real-world objects. For example, a holographic character might walk through a door and behind a wall, out of your sight.

Tips for integrating holograms and the real world

- Aligning to gravitational rules makes holograms easier to relate to and more believable. eg: Place a holographic dog on the ground & a vase on the table rather than have them floating in space.
- Many designers have found that they can even more believably integrate holograms by creating a "negative shadow" on the surface that the hologram is sitting on. They do this by creating a soft glow on the ground around the hologram and then subtracting the "shadow" from the glow. The soft glow integrates with the light from the real world and the shadow grounds the hologram in the environment.

A hologram is whatever you dream up

As a holographic developer, you have the power to break your creativity out of 2D screens and into the world around you. What will *you* build?



See also

- [Spatial sound](#)
- [Color, light and materials](#)

HoloLens hardware details

11/6/2018 • 2 minutes to read • [Edit Online](#)



Microsoft HoloLens is the world's first fully untethered holographic computer. HoloLens redefines personal computing through holographic experiences to empower you in new ways. HoloLens blends cutting-edge optics and sensors to deliver 3D holograms pinned to the real world around you.

How to get Microsoft HoloLens

Microsoft HoloLens is available to purchase as both a Development Edition and a Commercial Suite configuration (which includes a warranty and enterprise features for added security and device management). If you're in North America, you can also rent HoloLens through a partner.

[Click here for more details on purchasing or renting Microsoft HoloLens.](#)

Device Specifications

Optics



- See-through holographic lenses (waveguides)
- 2 HD 16:9 light engines
- Automatic pupillary distance calibration
- Holographic Resolution: 2.3M total light points
- Holographic Density: >2.5k radians (light points per radian)

Sensors



- 1 IMU
- 4 environment understanding cameras
- 1 depth camera
- 1 2MP photo / HD video camera
- Mixed reality capture
- 4 microphones
- 1 ambient light sensor

Human Understanding

- Spatial sound
- Gaze tracking
- Gesture input
- Voice support

Input / Output / Connectivity

- Built-in speakers
- Audio 3.5mm jack
- Volume up/down
- Brightness up/down
- Power button
- Battery status LEDs
- Wi-Fi 802.11ac
- Micro USB 2.0
- Bluetooth 4.1 LE

Power

- Battery Life
- 2-3 hours of active use
- Up to 2 weeks of standby time
- Fully functional when charging
- Passively cooled (no fans)

Processors



- Intel 32 bit architecture with TPM 2.0 support
- Custom-built Microsoft Holographic Processing Unit (HPU 1.0)

Weight

- 579g

Memory

- 64GB Flash
- 2GB RAM

What's in the box

- HoloLens Development Edition
- Clicker
- Carrying case
- Charger and cable
- Microfiber cloth
- Nose pads
- Overhead strap

OS and Apps

- Windows 10
- Windows Store
- Holograms
- Microsoft Edge
- Photos
- Settings
- Windows Feedback
- Calibration
- Learn Gestures

What you need to develop

- [Windows 10 PC able to run the latest compatible versions of Visual Studio and Unity](#)

Safety Eyewear

HoloLens has been tested and found to conform to the basic impact protection requirements of ANSI Z87.1, CSA Z94.3 and EN 166

Immersive headset hardware details

11/6/2018 • 2 minutes to read • [Edit Online](#)



Developers can purchase any of the Windows Mixed Reality immersive headsets available publicly at a number of retailers globally, as well as the [online Microsoft Store](#), and use them for mixed reality development. These immersive headsets deliver built-in inside-out tracking, meaning there is no need to purchase or install external trackers or place sensors on the wall. There's no complicated setup, just plug and play.

Device specifications

- Two high-resolution liquid crystal displays at 1440 x 1440 (the Samsung HMD Odyssey features an AMOLED display with 1440 x 1600 resolution)
- Display refresh rate up to 90 Hz (native)
- Built-in audio out and microphone support through 3.5mm jack (the Samsung HMD Odyssey includes built-in headphones and microphone)
- Single cable with HDMI 2.0 (display) and USB 3.0 (data) for connectivity
- Inside-out tracking

Input support

- [Motion controllers](#)
- [Gamepads supported by UWP](#)
- [Mouse and keyboard](#)
- [Voice](#) (via connected headset or microphone)

See also

- [Install the tools](#)

Hardware accessories

11/6/2018 • 4 minutes to read • [Edit Online](#)

Windows Mixed Reality devices support accessories. You'll pair supported accessories to HoloLens using Bluetooth, while you can use Bluetooth or USB to pair supported accessories to an immersive headset via the PC to which it is connected.

Two common scenarios for using accessories with HoloLens are as substitutes for the air tap gesture and the virtual keyboard. For this, the two most common accessories are the **HoloLens Clicker** and **Bluetooth keyboards**. Microsoft HoloLens includes a Bluetooth 4.1 radio and supports [Bluetooth HID](#) and [Bluetooth GATT](#) profiles.

Windows Mixed Reality immersive headsets require accessories for input beyond [gaze](#) and [voice](#). Supported accessories include **keyboard and mouse**, **gamepad**, and **motion controllers**.

Pairing Bluetooth accessories

Pairing a Bluetooth peripheral with Microsoft HoloLens is similar to pairing a Bluetooth peripheral with a Windows 10 desktop or mobile device:

1. From the Start Menu, open the **Settings** app
2. Go to **Devices**
3. Turn on the Bluetooth radio if it is off using the slider switch
4. Place your Bluetooth device in pairing mode. This varies from device to device. On most Bluetooth devices this is done by pressing and holding one or more buttons.
5. Wait for the name of the device to show up in the list of Bluetooth devices. When it does, select the device then select the **Pair** button. If you have many Bluetooth devices nearby you may need to scroll to the bottom of the Bluetooth device list to see the device you are trying to pair.
6. When pairing Bluetooth peripherals with input capability (e.g.: Bluetooth keyboards), a 6-digit or an 8-digit pin may be displayed. Be sure to type that pin on the peripheral and then press enter to complete pairing with Microsoft HoloLens.

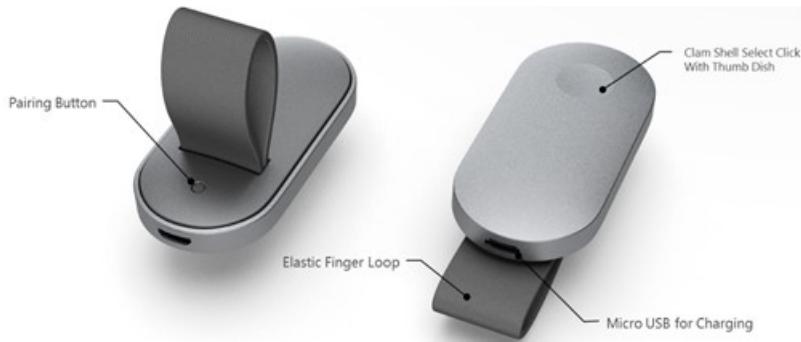
Motion controllers

Windows Mixed Reality [motion controllers](#) are supported by immersive headsets, but not HoloLens. These controllers offer precise and responsive tracking of movement in your field of view using the sensors in the immersive headset, meaning there is no need to install hardware on the walls in your space. Each controller features several methods of input.



HoloLens Clicker

The HoloLens Clicker is the first peripheral device built specifically for HoloLens and is included with the HoloLens Development Edition. The HoloLens Clicker allows a user to click and scroll with minimal hand motion as a replacement for the air-tap gesture. It is not a replacement for all [gestures](#). For instance, [bloom](#) and [resize or move](#) gestures use hand motions. The HoloLens clicker is an orientation sensor device with a simple button. It connects to the HoloLens using Bluetooth Low Energy (BTLE).



To select a [hologram](#), gaze at it, and then click. Orientation of the clicker does not matter for this operation. To scroll or pan, click and hold, then rotate the Clicker up/down or left/right. While scrolling, you'll reach the fastest speed with as little as +/- 15° of wrist rotation. Moving more will not scroll any faster.

There are two LEDs inside the Clicker:

- The white LED indicates whether the device is pairing (blinking) or charging (solid)
- The amber LED indicates that the device has low battery (blinking) or has suffered a failure (solid)

You can expect 2 weeks or more of regular use on a full charge (i.e., 2-3 hours on a wall charger). When the battery is low, the amber LED will blink 10 times over a 5-second period if you press the button or wake it from sleep. The amber LED will blink more rapidly over a 5-second period if your Clicker is in critically low battery mode.

Bluetooth keyboards

English language Qwerty Bluetooth keyboards can be paired and used anywhere you can use the holographic keyboard. Getting a quality keyboard makes a difference, so we recommend the [Microsoft Universal Foldable Keyboard](#) or the [Microsoft Designer Bluetooth Desktop](#).

Bluetooth gamepads

You can use a controller with apps and games that specifically enable gamepad support. Gamepads cannot be used to control the HoloLens user interface.

Xbox Wireless Controllers that come with the Xbox One S or sold as accessories for the Xbox One S feature Bluetooth connectivity that enable them to be used with HoloLens and immersive headsets. The Xbox Wireless Controller [must be updated](#) before it can be used with HoloLens.

Other brands of Bluetooth gamepads may work with Windows Mixed Reality devices, but support will vary by application.

Other Bluetooth accessories

As long as the peripheral supports either the Bluetooth HID or GATT profiles, it will be able to pair with HoloLens. Other Bluetooth HID and GATT devices besides keyboard, mice, and the HoloLens Clicker may require a companion application on Microsoft HoloLens to be fully functional.

Unsupported peripherals include:

- Peripherals in the Bluetooth audio profiles are not supported.
- Bluetooth audio devices such as speakers and headsets may appear as available in the Settings app, but are not supported to be used with Microsoft HoloLens as an audio endpoint.
- Bluetooth-enabled phones and PCs are not supported to be paired and used for file transfer.

Unpairing a Bluetooth peripheral

1. From the Start Menu, open the **Settings** app
2. Go to **Devices**
3. Turn on the Bluetooth radio if it is off
4. Find your device in the list of available Bluetooth devices
5. Select your device from the list and then select the **Remove** button

Disabling Bluetooth on Microsoft HoloLens

This will turn off the RF components of the Bluetooth radio and disable all Bluetooth functionality on Microsoft HoloLens.

1. From the Start Menu, open the **Settings** app
2. Go to **Devices**
3. Move the slider switch for Bluetooth to the OFF position

Gaze

11/6/2018 • 2 minutes to read • [Edit Online](#)

Gaze is the first form of input and is a primary form of targeting within mixed reality. Gaze tells you where the user is looking in the world and lets you determine their intent. In the real world, you'll typically look at an object that you intend to interact with. This is the same with gaze.

Mixed reality headsets use the position and orientation of your user's head, not their eyes, to determine their gaze vector. You can think of this vector as a laser pointer straight ahead from directly between the user's eyes. As the user looks around the room, your application can intersect this ray, both with its own holograms and with the [spatial mapping](#) mesh to determine what virtual or real-world object your user may be looking at.

On HoloLens, interactions should generally derive their targeting from the user's gaze, rather than trying to render or interact at the hand's location directly. Once an interaction has started, relative motions of the hand may be used to control the [gesture](#), as with the [manipulation or navigation](#) gesture. With immersive headsets, you can target using either gaze or pointing-capable [motion controllers](#).

Device support

FEATURE	HOLOLENS	IMMERSIVE HEADSETS
Gaze	✓ <input type="checkbox"/>	✓ <input type="checkbox"/>

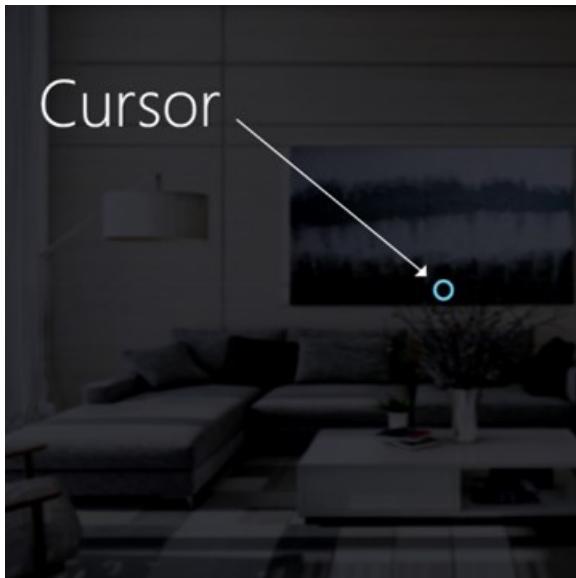
Uses of gaze

As a mixed reality developer, you can do a lot with gaze:

- Your app can intersect gaze with the holograms in your scene to determine where the user's attention is.
- Your app can target gestures and controller presses based on the user's gaze, letting the user select, activate, grab, scroll, or otherwise interact with their holograms.
- Your app can let the user place holograms on real-world surfaces, by intersecting their gaze ray with the spatial mapping mesh.
- Your app can know when the user is *not* looking in the direction of an important object, which can lead your app to give visual and audio cues to turn towards that object.

Cursor

Most apps should use a [cursor](#) (or other auditory/visual indication) to give the user confidence in what they're about to interact with. You typically position this cursor in the world where their gaze ray first interacts an object, which may be a hologram or a real-world surface.



An example visual cursor to show gaze

Giving action to the user's gaze

Once the user has targeted a hologram or real-world object using their gaze, their next step is to take action on that object. The primary ways for a user to take action are through [gestures](#), [motion controllers](#) and [voice](#).

See also

- [MR Input 210: Gaze](#)
- [Gaze, gestures, and motion controllers in DirectX](#)
- [Gaze in Unity](#)

Gestures

11/6/2018 • 7 minutes to read • [Edit Online](#)

Hand gestures allow users take action in mixed reality. Interaction is built on **gaze** to target and **gesture** or **voice** to act upon whatever element has been targeted. Hand gestures do not provide a precise location in space, but the simplicity of putting on a HoloLens and immediately interacting with content allows users to get to work without any other accessories.

Device support

FEATURE	HOLOLENS	IMMERSIVE HEADSETS
Gestures	✓	

Gaze-and-commit

To take actions, hand gestures use **Gaze** as the targeting mechanism. The combination of **Gaze** and the **Air tap** gesture results in a **gaze-and-commit** interaction. An alternative to gaze-and-commit is **point-and-commit**, enabled by **motion controllers**. Apps that run on HoloLens only need to support gaze-and-commit since HoloLens does not support motion controllers. Apps that run on both HoloLens and immersive headsets should support both gaze-driven and pointing-driven interactions, to give users choice in what input device they use.

The two core gestures of HoloLens

HoloLens currently recognizes two core component gestures - **Air tap** and **Bloom**. These two core interactions are the lowest level of spatial input data that a developer can access. They form the foundation for a variety of possible user actions.

Air tap

Air tap is a tapping gesture with the hand held upright, similar to a mouse click or select. This is used in most HoloLens experiences for the equivalent of a "click" on a UI element after targeting it with **Gaze**. It is a universal action that you learn once and then apply across all your apps. Other ways to perform select are by pressing the single button on a **HoloLens Clicker** or by speaking the voice command "Select".



Ready state for Air tap on HoloLens.

Air tap is a discrete gesture. A selection is either completed or it is not, an action is or is not taken within an

experience. It is possible, though not recommended without some specific purpose, to create other discrete gestures from combinations of the main components (e.g. a double-tap gesture to mean something different than a single-tap).



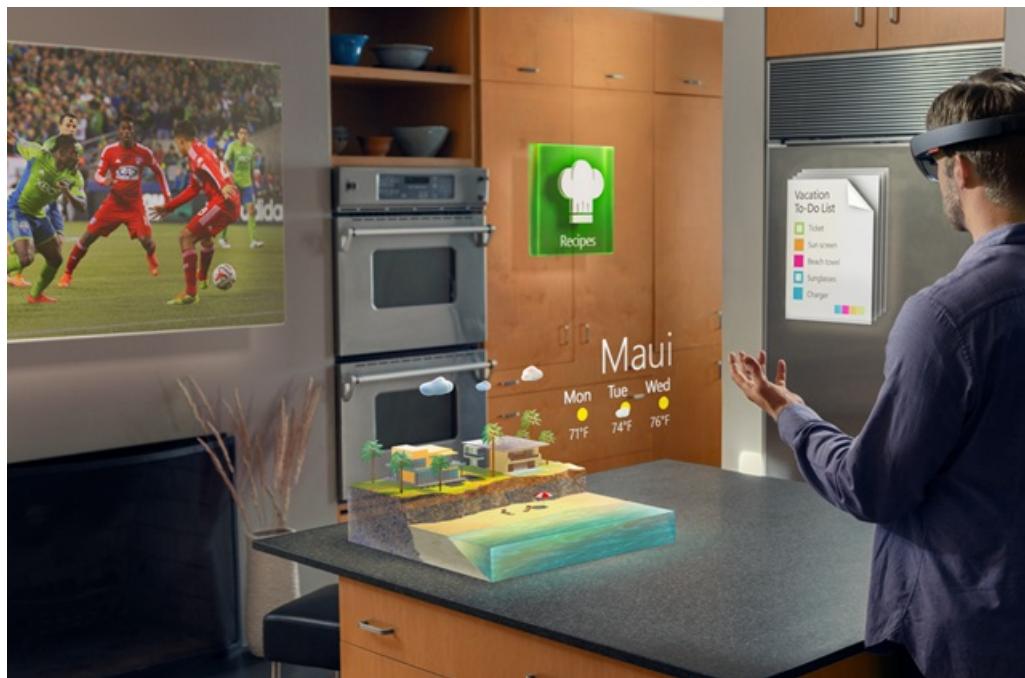
1. Finger in the ready position

2. Press finger down to tap or click

How to perform an Air tap: Raise your index finger to the ready position, press your finger down to tap or select and then back up to release.

Bloom

Bloom is the "home" gesture and is reserved for that alone. It is a special system action that is used to go back to the Start Menu. It is equivalent to pressing the Windows key on a keyboard or the Xbox button on an Xbox controller. The user can use either hand.



To do the bloom gesture on HoloLens, hold out your hand, palm up, with your fingertips together. Then open your hand. Note, you can also always return to Start by saying "Hey Cortana, Go Home". Apps cannot react specifically to a home action, as these are handled by the system.



How to perform the bloom gesture on HoloLens.

Composite gestures

Apps can recognize more than just individual taps. By combining tap, hold and release with the movement of the hand, more complex composite gestures can be performed. These composite or high-level gestures build on the low-level spatial input data (from Air tap and Bloom) that developers have access to.

COMPOSITE GESTURE	HOW TO APPLY
Air tap	The Air tap gesture (as well as the other gestures below) reacts only to a specific tap. To detect other taps, such as Menu or Grasp, your app must directly use the lower-level interactions described in two key component gestures section above.
Tap and hold	Hold is simply maintaining the downward finger position of the air tap. The combination of air tap and hold allows for a variety of more complex "click and drag" interactions when combined with arm movement such as picking up an object instead of activating it or "mousedown" secondary interactions such as showing a context menu. Caution should be used when designing for this gesture however, as users can be prone to relaxing their hand postures during the course of any extended gesture.
Manipulation	Manipulation gestures can be used to move, resize or rotate a hologram when you want the hologram to react 1:1 to the user's hand movements. One use for such 1:1 movements is to let the user draw or paint in the world. The initial targeting for a manipulation gesture should be done by gaze or pointing. Once the tap and hold starts, any manipulation of the object is then handled by hand movements, freeing the user to look around while they manipulate.

Navigation

Navigation gestures operate like a virtual joystick, and can be used to navigate UI widgets, such as radial menus. You tap and hold to start the gesture and then move your hand within a normalized 3D cube, centered around the initial press. You can move your hand along the X, Y or Z axis from a value of -1 to 1, with 0 being the starting point.

Navigation can be used to build velocity-based continuous scrolling or zooming gestures, similar to scrolling a 2D UI by clicking the middle mouse button and then moving the mouse up and down.

Navigation with rails refers to the ability of recognizing movements in certain axis until certain threshold is reached on that axis. This is only useful, when movement in more than one axis is enabled in an application by the developer, e.g. if an application is configured to recognize navigation gestures across X, Y axis but also specified X axis with rails. In this case system will recognize hand movements across X axis as long as they remain within an imaginary rails (guide) on X axis, if hand movement also occurs Y axis.

Within 2D apps, users can use vertical navigation gestures to scroll, zoom, or drag inside the app. This injects virtual finger touches to the app to simulate touch gestures of the same type. Users can select which of these actions take place by toggling between the tools on the bar above the app, either by selecting the button or saying '<Scroll/Drag/Zoom> Tool'.

Gesture recognizers

One benefit of using gesture recognition is that you can configure a gesture recognizer just for the gestures the currently targeted hologram can accept. The platform will do only the disambiguation necessary to distinguish those particular supported gestures. That way, a hologram that just supports air tap can accept any length of time between press and release, while a hologram that supports both tap and hold can promote the tap to a hold after the hold time threshold.

Hand recognition

HoloLens recognizes hand gestures by tracking the position of either or both hands that are visible to the device. HoloLens sees hands when they are in either the **ready state** (back of the hand facing you with index finger up) or the **pressed state** (back of the hand facing you with the index finger down). When hands are in other poses, the HoloLens will ignore them.

For each hand that HoloLens detects, you can access its position (without orientation) and its pressed state. As the hand nears the edge of the gesture frame, you're also provided with a direction vector, which you can show to the user so they know how to move their hand to get it back where HoloLens can see it.

Gesture frame

For gestures on HoloLens, the hand must be within a "gesture frame", in a range that the gesture-sensing cameras can see appropriately (very roughly from nose to waist, and between the shoulders). Users need to be trained on this area of recognition both for success of action and for their own comfort (many users will initially assume that the gesture frame must be within their view through HoloLens, and hold their arms up uncomfortably in order to interact). When using the HoloLens Clicker, your hands do not need to be within the gesture frame.

In the case of continuous gestures in particular, there is some risk of users moving their hands outside of the gesture frame while in mid-gesture (while moving some holographic object, for example), and losing their intended outcome.

There are three things that you should consider:

- User education on the gesture frame's existence and approximate boundaries (this is taught during HoloLens setup).
- Notifying users when their gestures are nearing/breaking the gesture frame boundaries within an application, to the degree that a lost gesture will lead to undesired outcomes. Research has shown the key qualities of such a notification system, and the HoloLens shell provides a good example of this type of notification (visual, on the central cursor, indicating the direction in which boundary crossing is taking place).
- Consequences of breaking the gesture frame boundaries should be minimized. In general, this means that the outcome of a gesture should be stopped at the boundary, but not reversed. For example, if a user is moving some holographic object across a room, movement should stop when the gesture frame is breached, but **not** be returned to the starting point. The user may experience some frustration then, but may more quickly understand the boundaries, and not have to restart their full intended actions each time.

See also

- [Gaze targeting](#)
- [Voice design](#)
- [MR Input 211: Gesture](#)
- [Gestures and motion controllers in Unity](#)
- [Gaze, gestures, and motion controllers in DirectX](#)
- [Motion controllers](#)

Motion controllers

11/6/2018 • 13 minutes to read • [Edit Online](#)

Motion controllers are [hardware accessories](#) that allow users to take action in mixed reality. An advantage of motion controllers over [gestures](#) is that the controllers have a precise position in space, allowing for fine grained interaction with digital objects. For Windows Mixed Reality immersive headsets, motion controllers are the primary way that users will take action in their world.

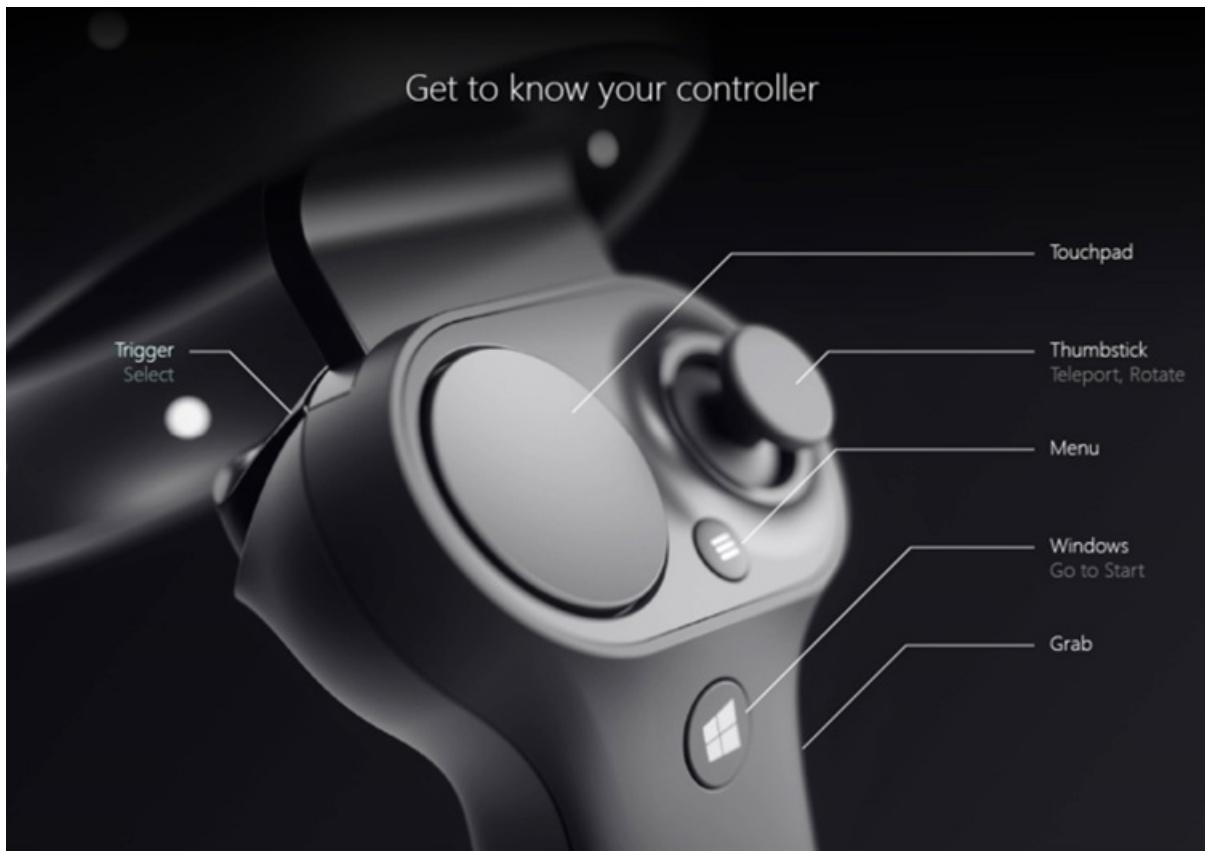


Device support

FEATURE	HOLOLENS	IMMERSIVE HEADSETS
Motion controllers		✓ <input type="checkbox"/>

Hardware details

Windows Mixed Reality motion controllers offer precise and responsive tracking of movement in your field of view using the sensors in the immersive headset, meaning there is no need to install hardware on the walls in your space. These motion controllers will offer the same ease of setup and portability as Windows Mixed Reality immersive headsets. Our device partners plan to market and sell these controllers on retail shelves this holiday.



Get to know your controller

Features:

- Optical tracking
- Trigger
- Grab button
- Thumbstick
- Touchpad

Setup

Before you begin

You will need:

- A set of two motion controllers.
- Four AA batteries.
- A PC capable of Bluetooth 4.0.

Check for Windows, Unity, and driver updates

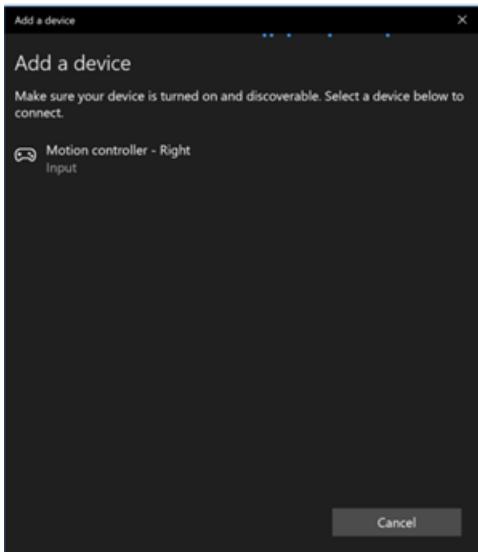
- Visit [Install the tools](#) for the preferred versions of Windows, Unity, etc. for mixed reality development.
- Make sure you have the most up-to-date [headset and motion controller drivers](#).

Pairing controllers

Motion controllers can be bonded with host PC using Windows settings like any other Bluetooth device.

1. Insert 2 AA batteries into the back of the controller. Leave the battery cover off for now.
2. If you're using an external USB Bluetooth Adapter instead of a built-in Bluetooth radio, please review the [Bluetooth best practices](#) before proceeding. For desktop configuration with built-in radio, please ensure antenna is connected.

3. Open **Windows Settings** -> **Devices** -> **Add Bluetooth or other device** -> **Bluetooth** and remove any earlier instances of "Motion controller – Right" and "Motion controller – Left". Check also Other devices category at the bottom of the list.
4. Select **Add Bluetooth or other device** and see it starting to discover Bluetooth devices.
5. Press and hold the controller's Windows button to turn on the controller, release once it buzzes.
6. Press and hold the pairing button (tab in the battery compartment) until the LEDs begin pulsing.
7. Wait "Motion controller - Left" or "Motion controller - Right" to appear to the bottom of the list. Select to pair. Controller will vibrate once when connected.



Select "Motion controller" to pair; if there are multiple instances, select one from the bottom of the list

8. You will see the controller appear in the Bluetooth settings under "**Mouse, keyboard, & pen**" category as **Connected**. At this point, you may get a firmware update – see [next section](#).
9. Reattach battery cover.
10. Repeat steps 1-9 for the second controller.

After successfully pairing both controllers, your settings should look like this under "**Mouse, keyboard, & pen**" category

Mouse, keyboard, & pen

Motion controller - Left
Connected

Motion controller - Right
Connected

Motion controllers connected

If the controllers are turned off after pairing, their status will show up as Paired. If controllers stay permanently under "Other devices" category pairing may have been only partially completed and need to be performed again to get controller functional.

Updating controller firmware

- If an immersive headset is connected to your PC, and new controller firmware is available, the firmware will be pushed to your motion controllers automatically the next time they're turned on. Controller firmware updates are indicated by a pattern of illuminating LED quadrants in a circular motion, and take 1-2 minutes.
- After the firmware update completes, the controllers will reboot and reconnect. Both controllers should be

connected now.

 Motion controller - Left
Connected

 Motion controller - Right
Connected

Controllers connected in Bluetooth settings

- Verify your controllers work properly:

1. Launch **Mixed Reality Portal** and enter your Mixed Reality Home.
2. Move your controllers and verify tracking, test buttons, and verify [teleportation](#) works. If they don't, then check out [motion controller troubleshooting](#).

Gazing and pointing

Windows Mixed Reality supports two key models for interaction, **gaze and commit** and **point and commit**:

- With **gaze and commit**, users target an object with their [gaze](#) and then select objects with hand air-taps, a gamepad, a clicker or their voice.
- With **point and commit**, a user can aim a pointing-capable motion controller at the target object and then select objects with the controller's trigger.

Apps that support pointing with motion controllers should also enable gaze-driven interactions where possible, to give users choice in what input devices they use.

Managing recoil when pointing

When using motion controllers to point and commit, your users will use the controller to target and then take action by pulling its trigger. Users who pull the trigger vigorously may end up aiming the controller higher at the end of their trigger pull than they'd intended.

To manage any such recoil that may occur when users pull the trigger, your app can snap its targeting ray when the trigger's analog axis value rises above 0.0. You can then take action using that targeting ray a few frames later once the trigger value reaches 1.0, as long as the final press occurs within a short time window. When using the higher-level [composite Tap gesture](#), Windows will manage this targeting ray capture and timeout for you.

Grip pose vs. pointing pose

Windows Mixed Reality supports motion controllers in a variety of form factors, with each controller's design differing in its relationship between the user's hand position and the natural "forward" direction that apps should use for pointing when rendering the controller.

To better represent these controllers, there are two kinds of poses you can investigate for each interaction source, the **grip pose** and the **pointer pose**.

Grip pose

The **grip pose** represents the location of either the palm of a hand detected by a HoloLens, or the palm holding a motion controller.

On immersive headsets, the grip pose is best used to render **the user's hand** or **an object held in the user's hand**, such as a sword or gun. The grip pose is also used when visualizing a motion controller, as the **renderable model** provided by Windows for a motion controller uses the grip pose as its origin and center of rotation.

The grip pose is defined specifically as follows:

- The **grip position**: The palm centroid when holding the controller naturally, adjusted left or right to center the position within the grip. On the Windows Mixed Reality motion controller, this position generally aligns with the

Grasp button.

- The **grip orientation's Right axis**: When you completely open your hand to form a flat 5-finger pose, the ray that is normal to your palm (forward from left palm, backward from right palm)
- The **grip orientation's Forward axis**: When you close your hand partially (as if holding the controller), the ray that points "forward" through the tube formed by your non-thumb fingers.
- The **grip orientation's Up axis**: The Up axis implied by the Right and Forward definitions.

Pointer pose

The **pointer pose** represents the tip of the controller pointing forward.

The system-provided pointer pose is best used to raycast when you are **rendering the controller model itself**. If you are rendering some other virtual object in place of the controller, such as a virtual gun, you should point with a ray that is most natural for that virtual object, such as a ray that travels along the barrel of the app-defined gun model. Because users can see the virtual object and not the physical controller, pointing with the virtual object will likely be more natural for those using your app.

Controller tracking state

Like the headsets, the Windows Mixed Reality motion controller requires no setup of external tracking sensors. Instead, the controllers are tracked by sensors in the headset itself.

If the user moves the controllers out of the headset's field of view, in most cases Windows will continue to infer controller positions and provide them to the app. When the controller has lost visual tracking for long enough, the controller's positions will drop to approximate-accuracy positions.

At this point, the system will body-lock the controller to the user, tracking the user's position as they move around, while still exposing the controller's true orientation using its internal orientation sensors. Many apps that use controllers to point at and activate UI elements can operate normally while in approximate accuracy without the user noticing.

Reasoning about tracking state explicitly

Apps that wish to treat positions differently based on tracking state may go further and inspect properties on the controller's state, such as `SourceLossRisk` and `PositionAccuracy`:

TRACKING STATE	SOURCELOSSRISK	POSITIONACCURACY	TRYGETPOSITION
High accuracy	< 1.0	High	true
High accuracy (at risk of losing)	== 1.0	High	true
Approximate accuracy	== 1.0	Approximate	true
No position	== 1.0	Approximate	false

These motion controller tracking states are defined as follows:

- **High accuracy**: While the motion controller is within the headset's field of view, it will generally provide high-accuracy positions, based on visual tracking. Note that a moving controller that momentarily leaves the field of view or is momentarily obscured from the headset sensors (e.g. by the user's other hand) will continue to return high-accuracy poses for a short time, based on inertial tracking of the controller itself.
- **High accuracy (at risk of losing)**: When the user moves the motion controller past the edge of the headset's

field of view, the headset will soon be unable to visually track the controller's position. The app knows when the controller has reached this FOV boundary by seeing the **SourceLossRisk** reach 1.0. At that point, the app may choose to pause controller gestures that require a steady stream of very high-quality poses.

- **Approximate accuracy:** When the controller has lost visual tracking for long enough, the controller's positions will drop to approximate-accuracy positions. At this point, the system will body-lock the controller to the user, tracking the user's position as they move around, while still exposing the controller's true orientation using its internal orientation sensors. Many apps that use controllers to point at and activate UI elements can operate as normal while in approximate accuracy without the user noticing. Apps with heavier input requirements may choose to sense this drop from **High** accuracy to **Approximate** accuracy by inspecting the **PositionAccuracy** property, for example to give the user a more generous hitbox on off-screen targets during this time.
- **No position:** While the controller can operate at approximate accuracy for a long time, sometimes the system knows that even a body-locked position is not meaningful at the moment. For example, a controller that was just turned on may have never been observed visually, or a user may put down a controller that's then picked up by someone else. At those times, the system will not provide any position to the app, and **TryGetPosition** will return false.

Interactions: Low-level spatial input

The core interactions across hands and motion controllers are **Select**, **Menu**, **Grasp**, **Touchpad**, **Thumbstick**, and **Home**.

- **Select** is the primary interaction to activate a hologram, consisting of a press followed by a release. For motion controllers, you perform a Select press using the controller's trigger. Other ways to perform a Select are by speaking the [voice command](#) "Select". The same select interaction can be used within any app. Think of Select as the equivalent of a mouse click, a universal action that you learn once and then apply across all your apps.
- **Menu** is the secondary interaction for acting on an object, used to pull up a context menu or take some other secondary action. With motion controllers, you can take a menu action using the controller's *menu* button. (i.e. the button with the hamburger "menu" icon on it)
- **Grasp** is how users can directly take action on objects at their hand to manipulate them. With motion controllers, you can do a grasp action by squeezing your fist tightly. A motion controller may detect a Grasp with a grab button, palm trigger or other sensor.
- **Touchpad** allows the user to adjust an action in two dimensions along the surface of a motion controller's touchpad, committing the action by clicking down on the touchpad. Touchpads provide a pressed state, touched state and normalized XY coordinates. X and Y range from -1 to 1 across the range of the circular touchpad, with a center at (0, 0). For X, -1 is on the left and 1 is on the right. For Y, -1 is on the bottom and 1 is on the top.
- **Thumbstick** allows the user to adjust an action in two dimensions by moving a motion controller's thumbstick within its circular range, committing the action by clicking down on the thumbstick. Thumbsticks also provide a pressed state and normalized XY coordinates. X and Y range from -1 to 1 across the range of the circular touchpad, with a center at (0, 0). For X, -1 is on the left and 1 is on the right. For Y, -1 is on the bottom and 1 is on the top.
- **Home** is a special system action that is used to go back to the Start Menu. It is similar to pressing the Windows key on a keyboard or the Xbox button on an Xbox controller. You can go home by pressing the Windows button on a motion controller. Note, you can also always return to Start by saying "Hey Cortana, Go Home". Apps cannot react specifically to home actions, as these are handled by the system.

Composite gestures: High-level spatial input

Both [hand gestures](#) and motion controllers can be tracked over time to detect a common set of high-level **composite gestures**. This enables your app to detect high-level **tap**, **hold**, **manipulation** and **navigation** gestures, whether users end up using hands or controllers.

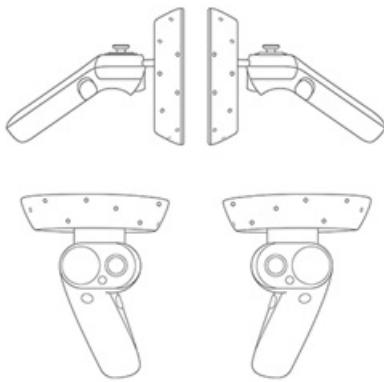
Rendering the motion controller model

3D controller models Windows makes available to apps a renderable model of each motion controller currently active in the system. By having your app dynamically load and articulate these system-provided controller models at runtime, you can ensure your app is forward-compatible to any future controller designs.

These renderable models should all be rendered at the **grip pose** of the controller, as the origin of the model is aligned with this point in the physical world. If you are rendering controller models, you may then wish to raycast into your scene from the **pointer pose**, which represents the ray along which users will naturally expect to point, given that controller's physical design.

For more information about how to load controller models dynamically in Unity, see the [Rendering the motion controller model in Unity](#) section.

2D controller line art While we recommend attaching in-app controller tips and commands to the in-app controller models themselves, some developers may want to use 2D line art representations of the motion controllers in flat "tutorial" or "how-to" UI. For those developers, we've made .png motion controller line art files available in both black and white below (right-click to save).



[Full-resolution motion controllers line art in "white"](#)

[Full-resolution motion controllers line art in "black"](#)

FAQ

Can I pair motion controllers to multiple PCs?

Motion controllers support pairing with a single PC. Follow instructions on [motion controller setup](#) to pair your controllers.

How do I update motion controller firmware?

Motion controller firmware is part of the headset driver and will be updated automatically on connection if required. Firmware updates typically take 1-2 minutes depending on Bluetooth radio and link quality. In rare cases, controller firmware updates may take up to 10 minutes, which can indicate poor Bluetooth connectivity or radio interference. Please see [Bluetooth best practices in the Enthusiast Guide](#) to troubleshoot connectivity issues. After a firmware update, controllers will reboot and reconnect to the host PC (you may notice the LEDs go bright for tracking). If a firmware update is interrupted (for example, the controllers lose power), it will be attempted again the next time the controllers are powered on.

How I can check battery level?

In the [Windows Mixed Reality home](#), you can turn your controller over to see its battery level on the reverse side of the virtual model. There is no physical battery level indicator.

Can you use these controllers without a headset? Just for the joystick/trigger/etc input?

Not for Universal Windows Applications.

Troubleshooting

See [motion controller troubleshooting](#) in the Enthusiast Guide.

Filing motion controller feedback/bugs

Give us [feedback](#) in Feedback Hub, using the "Mixed Reality -> Input" category.

See also

- [Gestures and motion controllers in Unity](#)
- [Gaze, gestures, and motion controllers in DirectX](#)
- [Gestures](#)
- [MR Input 213: Motion controllers](#)
- [Enthusiast's Guide: Your Windows Mixed Reality home](#)
- [Enthusiast's Guide: Using games & apps in Windows Mixed Reality](#)
- [How inside-out tracking works](#)

Voice input

11/6/2018 • 4 minutes to read • [Edit Online](#)

Voice is one of the three key forms of input on HoloLens. It allows you to directly command a hologram without having to use [gestures](#). You simply [gaze](#) at a hologram and speak your command. Voice input can be a natural way to communicate your intent. Voice is especially good at traversing complex interfaces because it lets users cut through nested menus with one command.

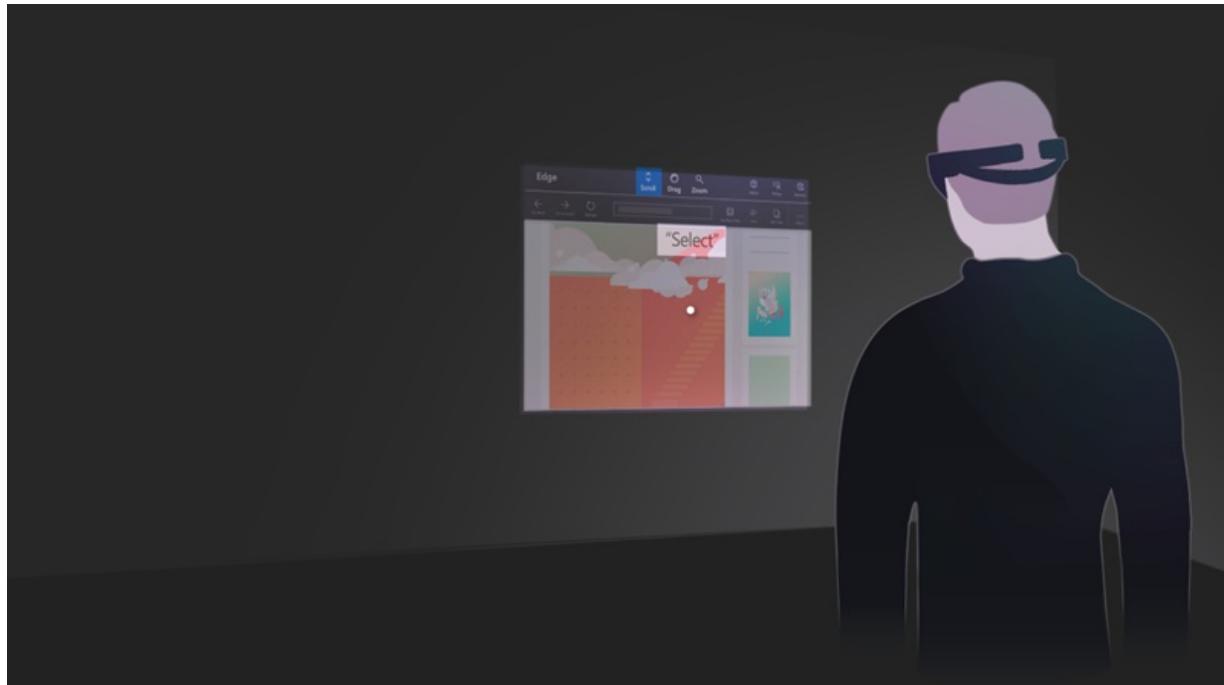
Voice input is powered by the [same engine](#) that supports speech in all other Universal Windows Apps.

Device support

FEATURE	HOOLENS	IMMERSIVE HEADSETS
Voice input	✓ <input type="checkbox"/>	✓ <input type="checkbox"/> (with microphone)

The "select" command

Even without specifically adding voice support to your app, your users can activate holograms simply by saying "select". This behaves the same as an [air tap](#) on HoloLens, pressing the select button on the [HoloLens clicker](#), or pressing the trigger on a [Windows Mixed Reality motion controller](#). You will hear a sound and see a tooltip with "select" appear as confirmation. "Select" is enabled by a low power keyword detection algorithm so it is always available for you to say at any time with minimal battery life impact, even with your hands at your side.



Say "select" to use the voice command for selection

Hey Cortana

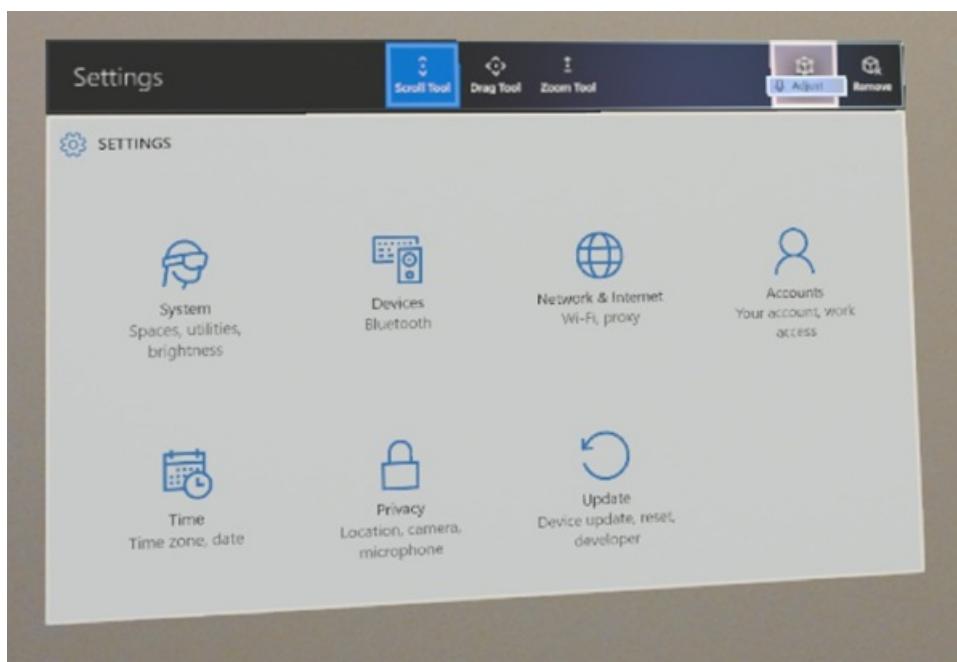
You can also say "Hey Cortana" to bring up Cortana at anytime. You don't have to wait for her to appear to continue asking her your question or giving her an instruction - for example, try saying "Hey Cortana what's the weather?" as a single sentence. For more information about Cortana and what you can do, simply ask her! Say "Hey Cortana what can I say?" and she'll pull up a list of working and suggested commands. If you're already in the Cortana app you can also click the ? icon on the sidebar to pull up this same menu.

HoloLens-specific commands

- "What can I say?"
- "Go home" or "Go to Start" - instead of [bloom](#) to get to [Start Menu](#)
- "Launch "
- "Move here"
- "Take a picture"
- "Start recording"
- "Stop recording"
- "Increase the brightness"
- "Decrease the brightness"
- "Increase the volume"
- "Decrease the volume"
- "Mute" or "Unmute"
- "Shut down the device"
- "Restart the device"
- "Go to sleep"
- "What time is it?"
- "How much battery do I have left?"
- "Call " (requires Skype for HoloLens)

"See It, Say It"

HoloLens has a "see it, say it" model for voice input, where labels on buttons tell users what voice commands they can say as well. For example, when looking at a 2D app, a user can say the "Adjust" command which they see in the App bar to adjust the position of the app in the world.



When apps follow this rule, users can easily understand what to say to control the system. To reinforce this, while gazing at a button, you will see a "voice dwell" tooltip that comes up after a second if the button is voice-enabled and displays the command to speak to "press" it.



"See it, say it" commands appear below the buttons

Voice commands for fast Hologram Manipulation

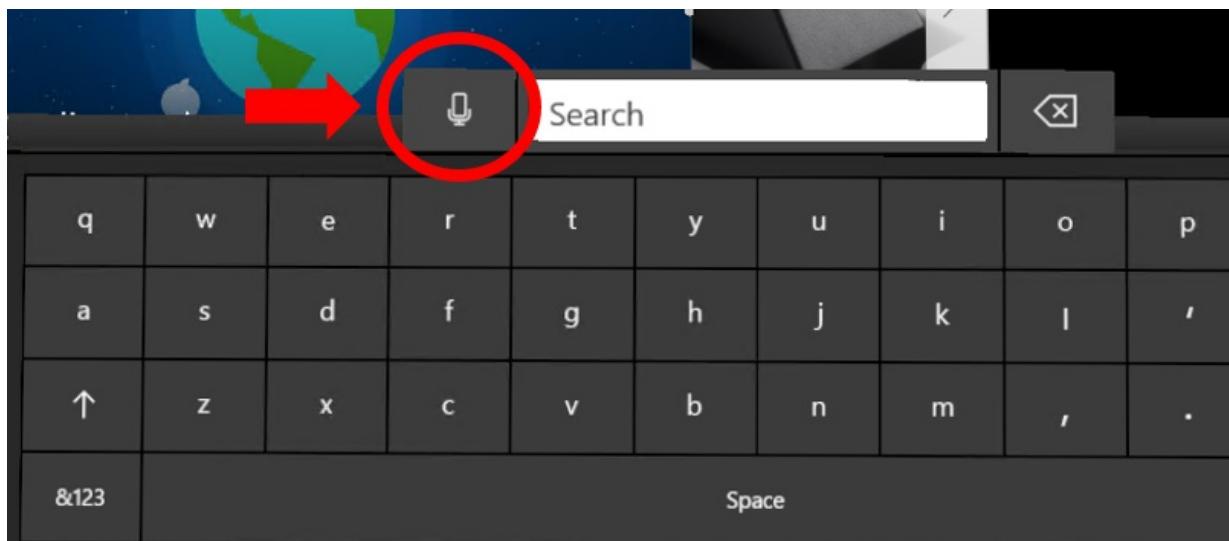
There are also a number of voice commands you can say while gazing at a hologram to quickly perform manipulation tasks. These voice commands work on 2D apps as well as 3D objects you have placed in the world.

Hologram Manipulation Commands

- Face me
- Bigger | Enhance
- Smaller

Dictation

Rather than typing with [air taps](#), voice dictation can be more efficient to enter text into an app. This can greatly accelerate input with less effort for the user.



Voice dictation starts by selecting the microphone button on the keyboard

Any time the holographic keyboard is active, you can switch to dictation mode instead of typing. Select the microphone on the side of the text input box to get started.

Communication

For applications that want to take advantage of the customized audio input processing options provided by HoloLens, it is important to understand the various [audio stream categories](#) your app can consume. Windows 10 supports several different stream categories and HoloLens makes use of three of these to enable custom processing to optimize the microphone audio quality tailored for speech, communication and other which can be used for ambient environment audio capture (i.e. "camcorder") scenarios.

- The AudioCategory_Communications stream category is customized for call quality and narration scenarios and provides the client with a 16kHz 24bit mono audio stream of the user's voice
- The AudioCategory_Speech stream category is customized for the HoloLens (Windows) speech engine and provides it with a 16kHz 24bit mono stream of the user's voice. This category can be used by 3rd party speech engines if needed.
- The AudioCategory_Other stream category is customized for ambient environment audio recording and provides the client with a 48kHz 24 bit stereo audio stream.

All this audio processing is hardware accelerated which means the features drain a lot less power than if the same processing was done on the HoloLens CPU. Avoid running other audio input processing on the CPU to maximize system battery life and take advantage of the built in, offloaded audio input processing.

Troubleshooting

If you're having any issues using "select" and "Hey Cortana", try moving to a quieter space, turning away from the source of noise, or by speaking louder. At this time, all speech recognition on HoloLens is tuned and optimized specifically to native speakers of United States English.

For the Windows Mixed Reality Developer Edition release 2017, the audio endpoint management logic will work fine (forever) after logging out and back in to the PC desktop after the initial HMD connection. Prior to that first sign out/in event after going through WMR OOBE, the user could experience various audio functionality issues ranging from no audio to no audio switching depending on how the system was set up prior to connecting the HMD for the first time.

See also

- [Voice input in DirectX](#)
- [Voice input in Unity](#)
- [MR Input 212: Voice](#)

Spatial mapping

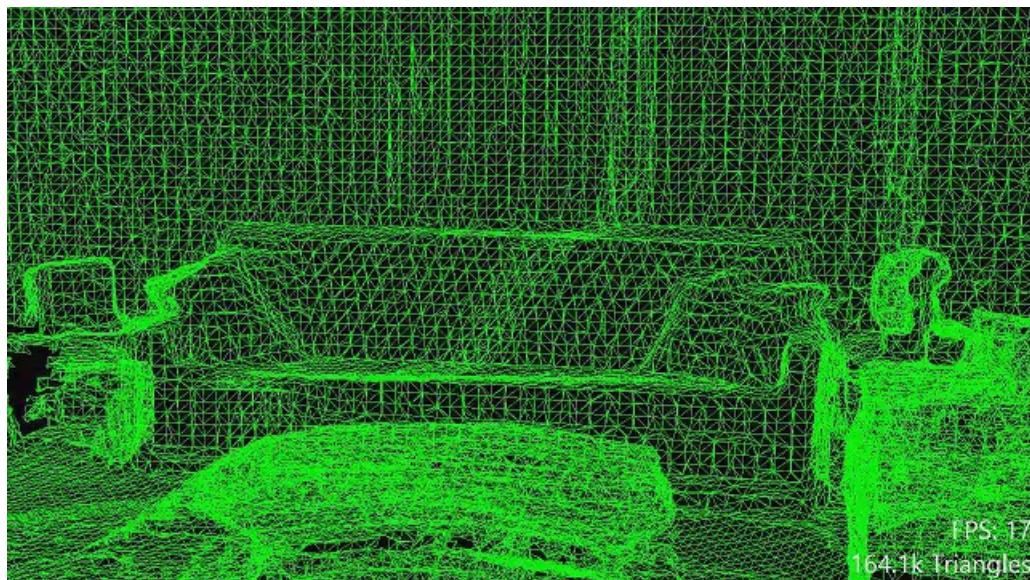
11/6/2018 • 23 minutes to read • [Edit Online](#)

Spatial mapping provides a detailed representation of real-world surfaces in the environment around the HoloLens, allowing developers to create a convincing mixed reality experience. By merging the real world with the virtual world, an application can make holograms seem real. Applications can also more naturally align with user expectations by providing familiar real-world behaviors and interactions.

Device support

FEATURE	HOLOLENS	IMMERSIVE HEADSETS
Spatial mapping	✓ <input type="checkbox"/>	

Conceptual overview



An example of a spatial mapping mesh covering a room

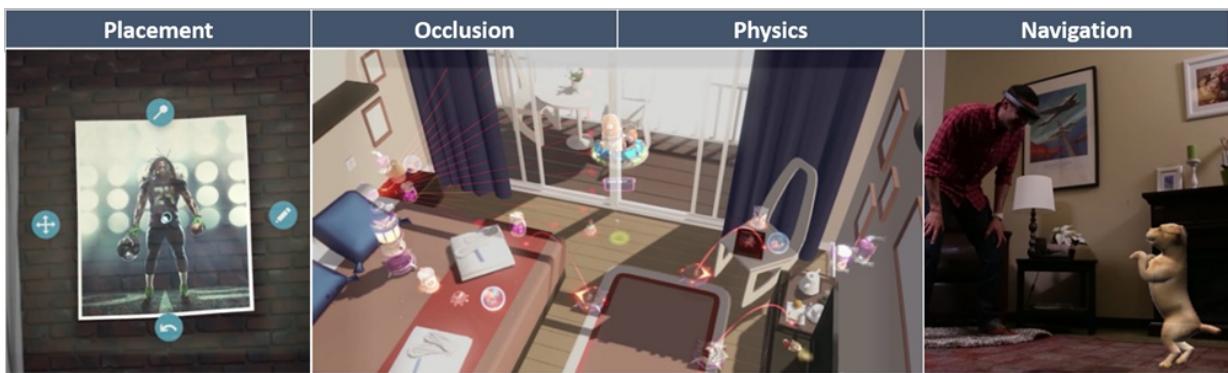
The two primary object types used for spatial mapping are the 'Spatial Surface Observer' and the 'Spatial Surface'.

The application provides the Spatial Surface Observer with one or more bounding volumes, to define the regions of space in which the application wishes to receive spatial mapping data. For each of these volumes, spatial mapping will provide the application with a set of Spatial Surfaces.

These volumes may be stationary (in a fixed location with respect to the real world) or they may be attached to the HoloLens (they move, but do not rotate, with the HoloLens as it moves through the environment). Each spatial surface describes real-world surfaces in a small volume of space, represented as a triangle mesh attached to a world-locked [spatial coordinate system](#).

As the HoloLens gathers new data about the environment, and as changes to the environment occur, spatial surfaces will appear, disappear and change.

Common usage scenarios



Placement

Spatial mapping provides applications with the opportunity to present natural and familiar forms of interaction to the user; what could be more natural than placing your phone down on the desk?

Constraining the placement of holograms (or more generally, any selection of spatial locations) to lie on surfaces provides a natural mapping from 3D (point in space) to 2D (point on surface). This reduces the amount of information the user needs to provide to the application and thus makes the user's interactions faster, easier and more precise. This is particularly true because 'distance away' is not something that we are used to physically communicating to other people or to computers. When we point with our finger, we are specifying a direction but not a distance.

An important caveat here is that when an application infers distance from direction (for example by performing a raycast along the user's gaze direction to find the nearest spatial surface), this must yield results that the user is able to reliably predict. Otherwise, the user will lose their sense of control and this can quickly become frustrating. One method that helps with this is to perform multiple raycasts instead of just one. The aggregate results should be smoother and more predictable, less susceptible to influence from transient 'outlier' results (as can be caused by rays passing through tiny holes or hitting small bits of geometry that the user is not aware of). Aggregation or smoothing can also be performed over time; for example you can limit the maximum speed at which a hologram can vary in distance from the user. Simply limiting the minimum and maximum distance value can also help, so the hologram being moved does not suddenly fly away into the distance or come crashing back into the user's face.

Applications can also use the shape and direction of surfaces to guide hologram placement. A holographic chair should not penetrate through walls and should sit flush with the floor even if it is slightly uneven. This kind of functionality would likely rely upon the use of physics collisions rather than just raycasts, however similar concerns will apply. If the hologram being placed has many small polygons that stick out, like the legs on a chair, it may make sense to expand the physics representation of those polygons to something wider and smoother so that they are more able to slide over spatial surfaces without snagging.

At its extreme, user input can be simplified away entirely and spatial surfaces can be used to perform entirely automatic hologram placement. For example, the application could place a holographic light-switch somewhere on the wall for the user to press. The same caveat about predictability applies doubly here; if the user expects control over hologram placement, but the application does not always place holograms where they expect (if the light-switch appears somewhere that the user cannot reach), then this will be a frustrating experience. It can actually be worse to perform automatic placement that requires user correction some of the time, than to just require the user to always perform placement themselves; because successful automatic placement is *expected*, manual correction feels like a burden!

Note also that the ability of an application to use spatial surfaces for placement depends heavily on the application's [scanning experience](#). If a surface has not been scanned, then it cannot be used for placement. It is up to the application to make this clear to the user, so that they can either help scan new surfaces or select a new location.

Visual feedback to the user is of paramount importance during placement. The user needs to know where the hologram is in relation to the nearest surface with [grounding effects](#). They should understand why the movement of their hologram is being constrained (for example, due to collision with another nearby surface). If they cannot place a hologram in the current location, then visual feedback should make it clear why not. For example, if the user is trying to place a holographic couch stuck half-way into the wall, then the portions of the couch that are behind the wall should pulsate in an angry color. Or conversely, if the application cannot find a spatial surface in a location where the user can see a real-world surface, then the application should make this clear. The obvious absence of a grounding effect in this area may achieve this purpose.

Occlusion

One of the primary uses of spatial mapping surfaces is simply to occlude holograms. This simple behavior has a huge impact on the perceived realism of holograms, helping to create a visceral sense that really inhabit the same physical space as the user.

Occlusion also provides information to the user; when a hologram appears to be occluded by a real-world surface, this provides additional visual feedback as to the spatial location of that hologram in the world. Conversely, occlusion can also usefully *hide* information from the user; occluding holograms behind walls can reduce visual clutter in an intuitive way. To hide or reveal a hologram, the user merely has to move their head.

Occlusion can also be used to prime expectations for a natural user interface based upon familiar physical interactions; if a hologram is occluded by a surface it is because that surface is solid, so the user should expect that the hologram will *collide* with that surface and not simply pass through it.

Sometimes, occlusion of holograms is undesirable. If a user needs to be able to interact with a hologram, then they need to be able to see it - even if it is behind a real-world surface. In such cases, it usually makes sense to render such a hologram differently when it is occluded (for example, by reducing its brightness). This way, the user will be able to visually locate the hologram, but they will still be aware that it is behind something.

Physics

The use of physics simulation is another way in which spatial mapping can be used to reinforce the *presence* of holograms in the user's physical space. When my holographic rubber ball rolls realistically off my desk, bounces across the floor and disappears under the couch, it might be hard for me to believe that it's not really there.

Physics simulation also provides the opportunity for an application to use natural and familiar physics-based interactions. Moving a piece of holographic furniture around on the floor will likely be easier for the user if the furniture responds as if it were sliding across the floor with the appropriate inertia and friction.

In order to generate realistic physical behaviors, you will likely need to perform some [mesh processing](#) such as filling holes, removing floating hallucinations and smoothing rough surfaces.

You will also need to consider how your application's [scanning experience](#) influences its physics simulation. Firstly, missing surfaces won't collide with anything; what happens when the rubber ball rolls off down the corridor and off the end of the known world? Secondly, you need to decide whether you will continue to respond to changes in the environment over time. In some cases, you will want to respond as quickly as possible; say if the user is using doors and furniture as movable barricades in defense against a tempest of incoming Roman arrows. In other cases though, you may want to ignore new updates; driving your holographic sports car around the racetrack on your floor may suddenly not be so fun if your dog decides to sit in the middle of the track.

Navigation

Applications can use spatial mapping data to grant holographic characters (or agents) the ability to navigate the real world in the same way a real person would. This can help reinforce the presence of holographic characters by restricting them to the same set of natural, familiar behaviors as those of the user and their friends.

Navigation capabilities could be useful to users as well. Once a navigation map has been built in a given area, it could be shared to provide holographic directions for new users unfamiliar with that location. This map could be designed to help keep pedestrian 'traffic' flowing smoothly, or to avoid accidents in dangerous locations like construction sites.

The key technical challenges involved in implementing navigation functionality will be reliable detection of walkable surfaces (humans don't walk on tables!) and graceful adaptation to changes in the environment (humans don't walk through closed doors!). The mesh may require some [processing](#) before it is usable for path-planning and navigation by a virtual character. Smoothing the mesh and removing hallucinations may help avoid characters becoming stuck. You may also wish to drastically simplify the mesh in order to speed up your character's path-planning and navigation calculations. These challenges have received a great deal of attention in the development of videogame technology, and there is a wealth of available research literature on these topics.

Note that the built-in NavMesh functionality in Unity cannot be used with spatial mapping surfaces. This is because spatial mapping surfaces are not known until the application starts, whereas NavMesh data files need to be generated from source assets ahead of time. Also note that, the spatial mapping system will not provide [information about surfaces very far away](#) from the user's current location. So the application must 'remember' surfaces itself if it is to build a map of a very large area.

Visualization

Most of the time it is appropriate for spatial surfaces to be invisible; to minimize visual clutter and let the real world speak for itself. However, sometimes it is useful to visualize spatial mapping surfaces directly, despite the fact that their real-world counterparts are already visible.

For example, when the user is trying to place a hologram onto a surface (placing a holographic cabinet on the wall, say) it can be useful to 'ground' the hologram by casting a shadow onto the surface. This gives the user a much clearer sense of the exact physical proximity between the hologram and the surface. This is also an example of the more general practice of visually 'previewing' a change before the user commits to it.

By visualizing surfaces, the application can share with the user its understanding of the environment. For example, a holographic board game could visualize the horizontal surfaces that it has identified as 'tables', so the user knows where they should go to interact.

Visualizing surfaces can be a useful way to show the user nearby spaces that are hidden from view. This could provide a simple way to give the user access to their kitchen (and all of its contained holograms) from their living room.

The surface meshes provided by spatial mapping may not be particularly 'clean'. Thus it is important to visualize them appropriately. Traditional lighting calculations may highlight errors in surface normals in a visually distracting manner, whilst 'clean' textures projected onto the surface may help to give it a tidier appearance. It is also possible to perform [mesh processing](#) to improve mesh properties, before the surfaces are rendered.

Using The Surface Observer

The starting point for spatial mapping is the surface observer. Program flow is as follows:

- Create a surface observer object
 - Provide one or more spatial volumes, to define the regions of interest in which the application wishes to receive spatial mapping data. A spatial volume is simply a shape defining a region of space, such as a sphere or a box.
 - Use a spatial volume with a world-locked spatial coordinate system to identify a fixed region of the physical world.
 - Use a spatial volume, updated each frame with a body-locked spatial coordinate system, to identify a

region of space that moves (but does not rotate) with the user.

- These spatial volumes may be changed later at any time, as the status of the application or the user changes.
- Use polling or notification to retrieve information about spatial surfaces
 - You may 'poll' the surface observer for spatial surface status at any time. Alternatively, you may register for the surface observer's 'surfaces changed' event, which will notify the application when spatial surfaces have changed.
 - For a dynamic spatial volume, such as the view frustum, or a body-locked volume, applications will need to poll for changes each frame by setting the region of interest and then obtaining the current set of spatial surfaces.
 - For a static volume, such as a world-locked cube covering a single room, applications may register for the 'surfaces changed' event to be notified when spatial surfaces inside that volume may have changed.
- Process surfaces changes
 - Iterate the provided set of spatial surfaces.
 - Classify spatial surfaces as added, changed or removed.
 - For each added or changed spatial surface, if appropriate submit an asynchronous request to receive updated mesh representing the surface's current state at the desired level of detail.
- Process the asynchronous mesh request (more details in following sections).

Mesh Caching

Spatial surfaces are represented by dense triangle meshes. Storing, rendering and processing these meshes can consume significant computational and storage resources. As such, each application should adopt a mesh caching scheme appropriate to its needs, in order to minimize the resources used for mesh processing and storage. This scheme should determine which meshes to retain and which to discard, as well as when to update the mesh for each spatial surface.

Many of the considerations discussed there will directly inform how your application should approach mesh caching. You should consider how the user moves through the environment, which surfaces are needed, when different surfaces will be observed and when changes in the environment should be captured.

When interpreting the 'surfaces changed' event provided by the surface observer, the basic mesh caching logic is as follows:

- If the application sees a spatial surface ID that it has not seen before, it should treat this as a new spatial surface.
- If the application sees a spatial surface with a known ID but with a new update time, it should treat this as an updated spatial surface.
- If the application no longer sees a spatial surface with a known ID, it should treat this as a removed spatial surface.

It is up to each application to then make the following choices:

- For new spatial surfaces, should mesh be requested?
 - Generally mesh should be requested immediately for new spatial surfaces, which may provide useful new information to the user.
 - However, new spatial surfaces near and in front of the user should be given priority and their mesh should be requested first.
 - If the new mesh is not needed, if for example the application has permanently or temporarily 'frozen' its model of the environment, then it should not be requested.
- For updated spatial surfaces, should mesh be requested?

- Updated spatial surfaces near and in front of the user should be given priority and their mesh should be requested first.
- It may also be appropriate to give higher priority to new surfaces than to updated surfaces, especially during the scanning experience.
- To limit processing costs, applications may wish to throttle the rate at which they process updates to spatial surfaces.
- It may be possible to infer that changes to a spatial surface are minor, for example if the bounds of the surface are small, in which case the update may not be important enough to process.
- Updates to spatial surfaces outside the current region of interest of the user may be ignored entirely, though in this case it may be more efficient to modify the spatial bounding volumes in use by the surface observer.
- For removed spatial surfaces, should mesh be discarded?
 - Generally mesh should be discarded immediately for removed spatial surfaces, so that hologram occlusion remains correct.
 - However, if the application has reason to believe that a spatial surface will reappear shortly (perhaps based upon the design of the user experience), then it may be more efficient to retain it than to discard its mesh and recreate it again later.
 - If the application is building a large-scale model of the user's environment then it may not wish to discard any meshes at all. It will still need to limit resource usage though, possibly by spooling meshes to disk as spatial surfaces disappear.
 - Note that some relatively rare events during spatial surface generation can cause spatial surfaces to be replaced by new spatial surfaces in a similar location but with different IDs. Consequently, applications that choose not to discard a removed surface should take care not to end up with multiple highly-overlapped spatial surface meshes covering the same location.
- Should mesh be discarded for any other spatial surfaces?
 - Even while a spatial surface exists, if it is no longer useful to the user's experience then it should be discarded. For example, if the application 'replaces' the room on the other side of a doorway with an alternate virtual space then the spatial surfaces in that room no longer matter.

Here is an example mesh caching strategy, using spatial and temporal hysteresis:

- Consider an application that wishes to use a frustum-shaped spatial volume of interest that follows the user's gaze as they look around and walk around.
- A spatial surface may disappear temporarily from this volume simply because the user looks away from the surface or steps further away from it... only to look back or moves closer again a moment later. In this case, discarding and re-creating the mesh for this surface represents a lot of redundant processing.
- To reduce the number of changes processed, the application uses two spatial surface observers, one contained within the other. The larger volume is spherical and follows the user 'lazily'; it only moves when necessary to ensure that its centre is within 2.0 metres of the user.
- New and updated spatial surface meshes are always processed from the smaller inner surface observer, but meshes are cached until they disappear from the larger outer surface observer. This allows the application to avoid processing many redundant changes due to local user movement.
- Since a spatial surface may also disappear temporarily due to tracking loss, the application also defers discarding removed spatial surfaces during tracking loss.
- In general, an application should evaluate the tradeoff between reduced update processing and increased memory usage to determine its ideal caching strategy.

Rendering

There are three primary ways in which spatial mapping meshes tend to be used for rendering:

- For surface visualization
 - It is often useful to visualize spatial surfaces directly. For example, casting 'shadows' from objects onto spatial surfaces can provide helpful visual feedback to the user while they are placing holograms on surfaces.
 - One thing to bear in mind is that spatial meshes are different to the kind of meshes that a 3D artist might create. The triangle topology will not be as 'clean' as human-created topology, and the mesh will suffer from [various errors](#).
 - In order to create a pleasing visual aesthetic, you may thus want to perform some [mesh processing](#), for example to fill holes or smooth surface normals. You may also wish to use a shader to project artist-designed textures onto your mesh instead of directly visualizing mesh topology and normals.
- For occluding holograms behind real-world surfaces
 - Spatial surfaces can be rendered in a depth-only pass which only affects the [depth buffer](#) and does not affect color render targets.
 - This primes the depth buffer to occlude subsequently-rendered holograms behind spatial surfaces. Accurate occlusion of holograms enhances the sense that holograms really exist within the user's physical space.
 - To enable depth-only rendering, update your blend state to set the [RenderTargetWriteMask](#) to zero for all color render targets.
- For modifying the appearance of holograms occluded by real-world surfaces
 - Normally rendered geometry is hidden when it is occluded. This is achieved by setting the depth function in your [depth-stencil state](#) to "less than or equal", which causes geometry to be visible only where it is **closer** to the camera than all previously rendered geometry.
 - However, it may be useful to keep certain geometry visible even when it is occluded, and to modify its appearance when occluded as a way of providing visual feedback to the user. For example, this allows the application to show the user the location of an object whilst making it clear that is behind a real-world surface.
 - To achieve this, render the geometry a second time with a different shader that creates the desired 'occluded' appearance. Before rendering the geometry for the second time, make two changes to your [depth-stencil state](#). First, set the depth function to "greater than or equal" so that the geometry will be visible only where it is **further** from the camera than all previously rendered geometry. Second, set the DepthWriteMask to zero, so that the depth buffer will not be modified (the depth buffer should continue to represent the depth of the geometry **closest** to the camera).

Performance is an important concern when rendering spatial mapping meshes. Here are some rendering performance techniques specific to rendering spatial mapping meshes:

- Adjust triangle density
 - When requesting spatial surface meshes from your surface observer, request the lowest density of triangle meshes that will suffice for your needs.
 - It may make sense to vary triangle density on a surface by surface basis, depending on the surface's distance from the user, and its relevance to the user experience.
 - Reducing triangle counts will reduce memory usage and vertex processing costs on the GPU, though it will not affect pixel processing costs.
- Perform frustum culling
 - Frustum culling skips drawing objects that cannot be seen because they are outside the current display frustum. This reduces both CPU and GPU processing costs.
 - Since culling is performed on a per-mesh basis and spatial surfaces can be large, breaking each spatial surface mesh into smaller chunks may result in more efficient culling (in that fewer offscreen triangles are rendered). There is a tradeoff, however; the more meshes you have, the more draw calls you must make, which can increase CPU costs. In an extreme case, the frustum culling calculations themselves could even have a measurable CPU cost.

- Adjust rendering order
 - Spatial surfaces tend to be large, because they represent the user's entire environment surrounding them. Pixel processing costs on the GPU can thus be high, especially in cases where there is more than one layer of visible geometry (including both spatial surfaces and other holograms). In this case, the layer nearest to the user will be occluding any layers further away, so any GPU time spent rendering those more distant layers is wasted.
 - To reduce this redundant work on the GPU, it helps to render opaque surfaces in front-to-back order (closer ones first, more distant ones last). By 'opaque' we mean surfaces for which the DepthWriteMask is set to one in your [depth-stencil state](#). When the nearest surfaces are rendered, they will prime the depth buffer so that more distant surfaces are efficiently skipped by the pixel processor on the GPU.

Mesh Processing

An application may want to perform [various operations](#) on spatial surface meshes to suit its needs. The index and vertex data provided with each spatial surface mesh uses the same familiar layout as the [vertex and index buffers](#) that are used for rendering triangle meshes in all modern rendering APIs. However, one key fact to be aware of is that spatial mapping triangles have a **front-clockwise winding order**. Each triangle is represented by three vertex indices in the mesh's index buffer and these indices will identify the triangle's vertices in a **clockwise** order, when the triangle is viewed from the **front** side. The front side (or outside) of spatial surface meshes corresponds as you would expect to the front (visible) side of real world surfaces.

Applications should only perform mesh simplification if the coarsest triangle density provided by the surface observer is still insufficiently coarse - this work is computationally expensive and already being performed by the runtime to generate the various provided levels of detail.

Because each surface observer can provide multiple unconnected spatial surfaces, some applications may wish to clip these spatial surface meshes against each other, then zipper them together. In general, the clipping step is required, as nearby spatial surface meshes often overlap slightly.

Raycasting and Collision

In order for a physics API (such as [Havok](#)) to provide an application with raycasting and collision functionality for spatial surfaces, the application must provide spatial surface meshes to the physics API. Meshes used for physics often have the following properties:

- They contain only small numbers of triangles. Physics operations are more computationally intensive than rendering operations.
- They are 'water-tight'. Surfaces intended to be solid should not have small holes in them; even holes too small to be visible can cause problems.
- They are converted into convex hulls. Convex hulls have few polygons and are free of holes, and they are much more computationally efficient to process than raw triangle meshes.

When performing raycasts against spatial surfaces, bear in mind that these surfaces are often complex, cluttered shapes full of messy little details - just like your desk! This means that a single raycast is often insufficient to give you enough information about the shape of the surface and the shape of the empty space near it. It is thus usually a good idea to perform many raycasts within a small area and to use the aggregate results to derive a more reliable understanding of the surface. For example, using the average of 10 raycasts to guide hologram placement on a surface will yield a far smoother and less 'jittery' result than using just a single raycast.

However, bear in mind that each raycast can have a high computational cost. Thus depending on your usage scenario you should trade off the computational cost of additional raycasts (performed every frame) against the computational cost of [mesh processing](#) to smooth and remove holes in spatial surfaces (performed when

spatial meshes are updated).

Troubleshooting

- In order for the surface meshes to be orientated correctly, each GameObject needs to be active before it is sent to the SurfaceObserver to have its mesh constructed. Otherwise, the meshes will show up in your space but rotated at weird angles.
- The GameObject that runs the script that communicates with the SurfaceObserver needs to be set to the origin. Otherwise, all of GameObjects that you create and send to the SurfaceObserver to have their meshes constructed will have an offset equal to the offset of the Parent Game Object. This can make your meshes show up several meters away which makes it very hard to debug what is going on.

See also

- [Coordinate systems](#)
- [Spatial mapping in DirectX](#)
- [Spatial mapping in Unity](#)
- [Spatial mapping design](#)
- [Case study - Looking through holes in your reality](#)

Spatial sound

11/6/2018 • 3 minutes to read • [Edit Online](#)

When objects are out of our line of sight, one of the ways that we can perceive what's going on around us is through sound. In Windows Mixed Reality, the audio engine provides the aural component of the mixed-reality experience by simulating 3D sound using direction, distance, and environmental simulations. Using spatial sound in an application allows developers to convincingly place sounds in a 3 dimensional space (sphere) all around the user. Those sounds will then seem as if they were coming from real physical objects or the mixed reality holograms in the user's surroundings. Given that [holograms](#) are objects made of light and sometimes sound, the sound component helps ground holograms making them more believable and creating a more immersive experience.

Although holograms can only appear visually where the user's gaze is pointing, your app's sound can come from all directions; above, below, behind, to the side, etc. You can use this feature to draw attention to an object that might not currently be in the user's view. A user can perceive sounds to be emanating from a source in the mixed-reality world. For example, as the user gets closer to an object or the object gets closer to them, the volume increases. Similarly, as objects move around a user, or vice versa, spatial sounds give the illusion that sounds are coming directly from the object.

Device support

FEATURE	HOLOLENS	IMMERSIVE HEADSETS
Spatial sound	✓ <input type="checkbox"/>	✓ <input type="checkbox"/> (with headphones attached)

Simulating the perceived location and distance of sounds

By analyzing how sound reaches both our ears, our brain determines the distance and direction of the object emitting the sound. An HRTF (or Head Related Transfer Function) simulates this interaction by modeling the spectral response that characterizes how an ear receives sound from a point in space. The spatial audio engine uses personalized HRTFs to expand the mixed reality experience, and simulate sounds that are coming from various directions and distances.

Left or right audio (azimuth) cues originate from differences in the time sound arrives at each ear. Up and down cues originate from spectral changes produced by the outer ear shape (pinnae). By designating where audio is coming from, the system can simulate the experience of sound arriving at different times to our ears. Note that on HoloLens, while azimuth spatialization is personalized, the simulation of elevation is based on an average set of anthropometrics. Thus, elevation accuracy may be less accurate than azimuth accuracy.

The characteristics of sounds also change based on the environment in which they exist. For instance, shouting in a cave will cause your voice to bounce off the walls, floors, and ceilings, creating an echo effect. The room model setting of spatial sound reproduces these reflections to place sounds in a particular audio environment. You can use this setting to match the user's actual location for simulation of sounds in that space to create a

more immersive audio experience.

Integrating spatial sound

Because the general principle of mixed reality is to ground [holograms](#) in the user's physical world or virtual environment, most sounds from holograms should be spatialized. On HoloLens, there are naturally CPU and memory budget considerations, but you can use 10-12 spatial sound voices there while using less than ~12% of the CPU (~70% of one of the four cores). Recommended use for spatial sound voices include:

- Gaze Mixing (highlighting objects, particularly when out of view). When a hologram needs a user's attention, play a sound on that hologram (e.g. have a virtual dog bark). This helps the user to find the hologram when it is not in view.
- Audio Haptics (reactive audio for touchless interactions). For example, play a sound when the user's hand or motion controller enters and exits the gesture frame. Or play a sound when the user selects a hologram.
- Immersion (ambient sounds surrounding the user).

It is also important to note that while blending standard stereo sounds with spatial sound can be effective in creating realistic environments, the stereo sounds should be relatively quiet to leave room for the subtle aspects of spatial sound, such as reflections (distance cues) that can be difficult to hear in a noisy environment.

Windows' spatial sound engine only supports a 48k sample rate for playback. Most middleware, such as Unity, will automatically convert sound files into the supported format, but when using Windows Audio APIs directly please match the format of the content to the format supported by the effect.

See also

- [MR Spatial 220](#)
- [Spatial sound in Unity](#)
- [Spatial sound in DirectX](#)
- [Spatial sound design](#)

Coordinate systems

11/6/2018 • 15 minutes to read • [Edit Online](#)

At their core, mixed reality apps place [holograms](#) in your world that look and sound like real objects. This involves precisely positioning and orienting those holograms at places in the world that are meaningful to the user, whether the world is their physical room or a virtual realm you've created. When reasoning about the position and orientation of your holograms, or any other geometry such as the [gaze](#) ray or [hand positions](#), Windows provides various real-world coordinate systems in which that geometry can be expressed, known as **spatial coordinate systems**.

Device support

FEATURE	HOLOLENS	IMMERSIVE HEADSETS
Stationary frame of reference	✓□	✓□
Attached frame of reference	✓□	✓□
Stage frame of reference	Not supported yet	✓□
Spatial anchors	✓□	✓□
Spatial mapping	✓□	

Mixed reality experience scales

Mixed reality apps can design for a broad range of user experiences, from 360-degree video viewers that just need the headset's orientation, to full world-scale apps and games, which need spatial mapping and spatial anchors:

EXPERIENCE SCALE	REQUIREMENTS	EXAMPLE EXPERIENCE
Orientation-only	Headset orientation (gravity-aligned)	360° video viewer
Seated-scale	Above, plus headset position relative to zero position	Racing game or space simulator
Standing-scale	Above, plus stage floor origin	Action game where you duck and dodge in place
Room-scale	Above, plus stage bounds polygon	Puzzle game where you walk around the puzzle
World-scale	Spatial anchors (and typically spatial mapping)	Game with enemies coming from your real walls, such as RoboRaid

These experience scales follow a "nesting dolls" model. The key design principle here for Windows Mixed Reality is that a given headset supports apps built for a target experience scale, as well as all lesser scales:

6DOF TRACKING	FLOOR DEFINED	360° TRACKING	BOUNDS DEFINED	Spatial Anchors	Max Experience
No	-	-	-	-	Orientation-only
Yes	No	-	-	-	Seated
Yes	Yes	No	-	-	Standing - Forward
Yes	Yes	Yes	No	-	Standing - 360°
Yes	Yes	Yes	Yes	No	Room
Yes	Yes	Yes	Yes	Yes	World

Note that the Stage frame of reference is not yet supported on HoloLens. A room-scale app on HoloLens currently needs to use [spatial mapping](#) to find the user's floor and walls.

Spatial coordinate systems

All 3D graphics applications use [Cartesian coordinate systems](#) to reason about the positions and orientations of objects in the virtual worlds they render. Such coordinate systems establish 3 perpendicular axes along which to position objects: an X, Y, and Z axis.

In [mixed reality](#), your apps will reason about both virtual and physical coordinate systems. Windows calls a coordinate system that has real meaning in the physical world a **spatial coordinate system**.

Spatial coordinate systems express their coordinate values in meters. This means that objects placed 2 units apart in either the X, Y or Z axis will appear 2 meters apart from one another when rendered in mixed reality. This lets you easily render objects and environments at real-world scale.

In general, Cartesian coordinate systems can be either right-handed or left-handed. Spatial coordinate systems on Windows are always right-handed, which means that the positive X-axis points right, the positive Y-axis points up (aligned to gravity) and the positive Z-axis points towards you.

In both kinds of coordinate systems, the positive X-axis points to the right and the positive Y-axis points up. The difference is whether the positive Z-axis points towards or away from you. You can remember which direction the positive Z-axis points by pointing the fingers of either your left or right hand in the positive X direction and curling them to the positive Y direction. The direction your thumb points, either toward or away from you, is the direction that the positive Z-axis points for that coordinate system.

Building an orientation-only or seated-scale experience

The key to holographic [rendering](#) is changing your app's view of its holograms each frame as the user moves around, to match their predicted head motion. You can build **seated-scale experiences** that respect changes to the user's head position and head orientation using a **stationary frame of reference**.

Some content must ignore head position updates, staying fixed at a chosen heading and distance from the user at all times. The primary example is 360-degree video: because the video is captured from a single fixed perspective, it would ruin the illusion for the view position to move relative to the content, even though the view orientation must change as the user looks around. You can build such **orientation-only experiences**

using an **attached frame of reference**.

Stationary frame of reference

The coordinate system provided by a stationary frame of reference works to keep the positions of objects near the user as stable as possible relative to the world, while respecting changes in the user's head position.

For seated-scale experiences in a game engine such as [Unity](#), a stationary frame of reference is what defines the engine's "world origin." Objects that are placed at a specific world coordinate use the stationary frame of reference to define their position in the real-world using those same coordinates. Content that stays put in the world, even as the user walks around, is known as **world-locked** content.

An app will typically create one stationary frame of reference on startup and use its coordinate system throughout the app's lifetime. As an app developer in Unity, you can just start placing content relative to the origin, which will be at the user's initial head position and orientation. If the user moves to a new place and wants to continue their seated-scale experience, you can recenter the world origin at that location.

Over time, as the system learns more about the user's environment, it may determine that distances between various points in the real-world are shorter or longer than the system previously believed. If you render holograms in a stationary frame of reference for an app on HoloLens where users wander beyond an area about 5 meters wide, your app may observe drift in the observed location of those holograms. If your experience has users wandering beyond 5 meters, you're building a [world-scale experience](#), which will require additional techniques to keep holograms stable, as described below.

Attached frame of reference

An attached frame of reference moves with the user as they walk around, with a fixed heading defined when the app first creates the frame. This lets the user comfortably look around at content placed within that frame of reference. Content rendered in this user-relative way is called **body-locked** content.

When the headset can't figure out where it is in the world, an attached frame of reference provides the only coordinate system which can be used to render holograms. This makes it ideal for displaying fallback UI to tell the user that their device can't find them in the world. Apps that are seated-scale or higher should include an orientation-only fallback to help the user get going again, with UI similar to that shown in the [Mixed Reality home](#).

Building a standing-scale or room-scale experience

To go beyond seated-scale on an immersive headset and build a **standing-scale experience**, you can use the **stage frame of reference**.

To provide a **room-scale experience**, letting users walk around within the 5-meter boundary they pre-defined, you can check for **stage bounds** as well.

Stage frame of reference

When first setting up an immersive headset, the user defines a **stage**, which represents the room in which they will experience mixed reality. The stage minimally defines a **stage origin**, a spatial coordinate system centered at the user's chosen floor position and forward orientation where they intend to use the device. By placing content in this stage coordinate system at the Y=0 floor plane, you can ensure your holograms appear comfortably on the floor when the user is standing, providing users with a **standing-scale experience**.

Stage bounds

The user may also optionally define **stage bounds**, an area within the room that they've cleared of furniture where they intend to move around in mixed reality. If so, the app can build a **room-scale experience**, using these bounds to ensure that holograms are always placed where the user can reach them.

Because the stage frame of reference provides a single fixed coordinate system within which to place floor-relative content, it is the easiest path for porting standing-scale and room-scale applications developed for

virtual reality headsets. However, as with those VR platforms, a single coordinate system can only stabilize content in about a 5 meter (15 foot) diameter, before lever-arm effects cause content far from the center to shift noticeably as the system adjusts. To go beyond 5 meters, spatial anchors are needed.

Building a world-scale experience

HoloLens allows for true **world-scale experiences** that let users wander beyond 5 meters. To build a world-scale app, you'll need new techniques beyond those used for room-scale experiences.

Why a single rigid coordinate system cannot be used beyond 5 meters

Today, when writing games, data visualization apps, or virtual reality apps, the typical approach is to establish one absolute world coordinate system that all other coordinates can reliably map back to. In that environment, you can always find a stable transform that defines a relationship between any two objects in that world. If you didn't move those objects, their relative transforms would always remain the same. This kind of global coordinate system works well when rendering a purely virtual world where you know all of the geometry in advance. Room-scale VR apps today typically establish this kind of absolute room-scale coordinate system with its origin on the floor.

In contrast, an untethered mixed reality device such as HoloLens has a dynamic sensor-driven understanding of the world, continuously adjusting its knowledge over time of the user's surroundings as they walk many meters across an entire floor of a building. In a world-scale experience, if you placed all your holograms in a single rigid coordinate system, those holograms would necessarily drift over time, either relative to the world or to each other.

For example, the headset may currently believe two locations in the world to be 4 meters apart, and then later refine that understanding, learning that the locations are in fact 3.9 meters apart. If those holograms had initially been placed 4 meters apart in a single rigid coordinate system, one of them would then always appear 0.1 meters off from the real world.

Spatial anchors

Windows Mixed Reality solves the issue described in the previous section by letting you create [spatial anchors](#) to mark important points in the world where the user has placed holograms. A spatial anchor represents an important point in the world that the system should keep track of over time.

As the device learns about the world, these spatial anchors can adjust their position relative to one another as needed to ensure that each anchor stays precisely where it was placed relative to the real-world. By placing a spatial anchor at the location where the user places a hologram and then positioning that hologram relative to its spatial anchor, you can ensure that the hologram maintains optimal stability, even as the user roams across tens of meters.

This continuous adjustment of spatial anchors relative to one another is the key difference between coordinate systems from spatial anchors and stationary frames of reference:

- Holograms placed in the stationary frame of reference all retain a rigid relationship to one another. However, as the user walks long distances, that frame's coordinate system may drift relative to the world to ensure that holograms next to the user appear stable.
- Holograms placed in the stage frame of reference also retain a rigid relationship to one another. In contrast to the stationary frame, the stage frame always remains fixed in place relative to its defined physical origin. However, content rendered in the stage's coordinate system beyond its 5-meter boundary will only appear stable while the user is standing within that boundary.
- Holograms placed using one spatial anchor may drift relative to holograms placed using another spatial anchor. This allows Windows to improve its understanding of the position of each spatial anchor, even if, for example, one anchor needs to adjust itself left and another anchor needs to adjust right.

In contrast to a stationary frame of reference, which always optimizes for stability near the user, the stage frame of reference and spatial anchors ensure stability near their origins. This helps those holograms stay precisely in place over time, but it also means that holograms rendered too far away from their coordinate system's origin will experience increasingly severe lever-arm effects. This is because small adjustments to the position and orientation of the stage or anchor are magnified proportional to the distance from that anchor.

A good rule of thumb is to ensure that anything you render based on a distant spatial anchor's coordinate system is within about 3 meters of its origin. For a nearby stage origin, rendering distant content is OK, as any increased positional error will affect only small holograms that will not shift much in the user's view.

Spatial anchor persistence

Spatial anchors can also allow your app to remember an important location even after your app suspends or the device is shut down.

You can save to disk the spatial anchors your app creates, and then load them back again later, by persisting them to your app's **spatial anchor store**. When saving or loading an anchor, you provide a string key that is meaningful to your app, in order to identify the anchor later. Think of this key as the filename for your anchor. If you want to associate other data with that anchor, such as a 3D model that the user placed at that location, save that to your app's local storage and associate it with the key you chose.

By persisting anchors to the store, your users can place individual holograms or place a workspace around which an app will place its various holograms, and then find those holograms later where they expect them, over many uses of your app.

Spatial anchor sharing

Your app can also share spatial anchors with other devices. By transferring a spatial anchor along with its supporting understanding of the environment and sensor data around it from one HoloLens to another, both devices can then reason about the same location. By having each device render a hologram using that shared spatial anchor, both users will see the hologram appear at the same place in the real world.

Avoid head-locked content

We strongly discourage rendering head-locked content, which stays at a fixed spot in the display (such as a HUD). In general, head-locked content is uncomfortable for users and does not feel like a natural part of their world.

Head-locked content should usually be replaced with holograms that are attached to the user or placed in the world itself. For example, [cursors](#) should generally be pushed out into the world, scaling naturally to reflect the position and distance of the object under the user's gaze.

Handling tracking errors

In some environments such as dark hallways, it may not be possible for a headset using inside-out tracking to locate itself correctly in the world. This can lead holograms to either not show up or appear at incorrect places if handled incorrectly. We now discuss the conditions in which this can happen, its impact on user experience, and tips to best handle this situation.

Headset cannot track due to insufficient sensor data

Sometimes, the headset's sensors are not able to figure out where the headset is. This can happen if the room is dark, or if the sensors are covered by hair or hands, or if the surroundings do not have enough texture.

When this happens, the headset will be unable to track its position with enough accuracy to render world-locked holograms. You won't be able to figure out where a spatial anchor, stationary frame or stage frame is relative to the device, but you can still render body-locked content in the attached frame of reference.

Your app should tell the user how to get positional tracking back, rendering some fallback body-locked

content that describes some tips, such as uncovering the sensors and turning on more lights.

Headset tracks incorrectly due to dynamic changes in the environment

Sometimes, the device cannot track properly if there are lots of dynamic changes in the environment, such as many people walking around in the room. In this case, the holograms may seem to jump or drift as the device tries to track itself in this dynamic environment. We recommend using the device in a less dynamic environment if you hit this scenario.

Headset tracks incorrectly because the environment has changed significantly over time

Sometimes, when you start using a headset in an environment which has undergone lot of changes (e.g. significant movement of furniture, wall hangings etc.), it is possible that some holograms may appear shifted from their original locations. The earlier holograms may also jump around as the user moves around in this new space. This is because the system's understanding of your space no longer holds and it tries to remap the environment while trying to reconcile the features of the room. In this scenario, it is advised to encourage users to re-place holograms they pinned in the world if they are not appearing where expected.

Headset tracks incorrectly due to identical spaces in an environment

Sometimes, a home or other space may have two identical areas. For example, two identical conference rooms, two identical corner areas, two large identical posters that cover the device's field of view. In such scenarios, the device may, at times, get confused between the identical parts and mark them as the same in its internal representation. This may cause the holograms from some areas to appear in other locations. The device may start to lose tracking often since its internal representation of the environment has been corrupted. In this case, it is advised to reset the system's environmental understanding. Please note that resetting the map leads to loss of all spatial anchor placements. This will cause the headset to track well in the unique areas of the environment. However, the problem may re-occur if the device gets confused between the identical areas again.

See also

- [GDC 2017 presentation on spatial coordinate systems and holographic rendering](#)
- [Coordinate systems in Unity](#)
- [Coordinate systems in DirectX](#)
- [Spatial anchors](#)
- [Shared experiences in mixed reality](#)
- [Case study - Looking through holes in your reality](#)

Spatial anchors

11/6/2018 • 5 minutes to read • [Edit Online](#)

A spatial anchor represents an important point in the world that the system should keep track of over time. Each anchor has a [coordinate system](#) that adjusts as needed, relative to other anchors or frames of reference, in order to ensure that anchored holograms stay precisely in place. By saving spatial anchors to disk and loading them back later, your app can reason about the same location in the real-world across multiple app sessions.

Rendering a hologram in an anchor's coordinate system gives you the most accurate positioning for that hologram at any given time. This comes at the cost of small adjustments over time to the hologram's position, as the system continually moves it back into place relative to the real world.

For standing-scale or room-scale experiences for tethered desktop headsets that will stay within a 5-meter diameter, you can usually just use the [stage frame of reference](#) instead of spatial anchors, providing you a single coordinate system in which to render all content. However, if your app intends to let users wander beyond 5 meters on HoloLens, perhaps operating throughout an entire floor of a building, you'll need spatial anchors to keep content stable.

While spatial anchors are great for holograms that should remain fixed in the world, once an anchor is placed, it can't be moved. There are alternatives to anchors that are more appropriate for dynamic holograms that should tag along with the user. It is best to position dynamic holograms using a stationary frame of reference (the foundation for Unity's world coordinates) or an attached frame of reference.

Best practices

These spatial anchor guidelines will help you render stable holograms that accurately track the real world.

Create spatial anchors where users place them

Most of the time, users should be the ones explicitly placing spatial anchors.

For example, on HoloLens, an app can intersect the user's [gaze](#) ray with the [spatial mapping](#) mesh to let the user decide where to place a hologram. When the user taps to place that hologram, create a spatial anchor at the intersection point and then place the hologram at the origin of that anchor's coordinate system.

Spatial anchors are quite cheap to create, and the system will consolidate their internal data if multiple anchors can share their underlying sensor data. You should typically create a new spatial anchor for each hologram that a user explicitly places, except in cases outlined below such as grouped holograms.

Always render anchored holograms within 3 meters of their anchor

Spatial anchors stabilize their coordinate system near the anchor's origin. If you render holograms more than about 3 meters from that origin, those holograms may experience noticeable positional errors in proportion to their distance from that origin, due to lever-arm effects. That works if the user stands near the anchor, since the hologram is far away from the user too, meaning the angular error of the distant hologram will be small. However, if the user walks up to that distant hologram, it will then be large in their view, making the lever-arm effects from the faraway anchor origin quite obvious.

Group holograms that should form a rigid cluster

Multiple holograms can share the same spatial anchor if the app expects those holograms to maintain fixed relationships to one another.

For example, if you are animating a holographic solar system in a room, it's better to tie all of the solar system objects to a single anchor in the center, so that they move smoothly relative to each other. In this case, it is the

solar system as a whole that is anchored, even though its component parts are moving dynamically around the anchor.

The key caveat here to maintain hologram stability is to follow the 3-meter rule above.

Render highly dynamic holograms using the stationary frame of reference instead of a spatial anchor

If you have a highly dynamic hologram, such as a character walking around the room, or a floating UI that follows along the wall near the user, it is best to skip spatial anchors and render those holograms directly in the coordinate system provided by the [stationary frame of reference](#) (i.e. in Unity, you achieve this by placing holograms directly in world coordinates without a WorldAnchor). Holograms in a stationary coordinate system may experience drift when the user is far from the hologram, but this is less likely to be noticeable for dynamic holograms: either the hologram is constantly moving anyway, or its motion constantly keeps it close to the user, where drift will be minimized.

One interesting case of dynamic holograms is an object that is animating from one anchored coordinate system to another. For example, you might have two castles 10 meters apart, each on their own spatial anchor, with one castle firing a cannonball at the other castle. At the moment the cannonball is fired, you can render it at the appropriate location in the stationary frame of reference, so as to coincide with the cannon in the first castle's anchored coordinate system. It can then follow its trajectory in the stationary frame of reference as it flies 10 meters through the air. As the cannonball reaches the other castle, you may choose to move it into the second castle's anchored coordinate system to allow for physics calculations with that castle's rigid bodies.

Avoid creating a grid of spatial anchors

You may be tempted to drop a regular grid of spatial anchors as the user walks around, transitioning dynamic objects from anchor to anchor as they move around. However, this involves a lot of management for your app, without the benefit of the deep sensor data that the system itself maintains internally. For these cases, you will usually achieve better results with less effort by placing your holograms in the stationary frame of reference, as described in the section above.

Release spatial anchors you no longer need

While a spatial anchor is active, the system will prioritize keeping around the sensor data that is near that anchor. If you are no longer using a spatial anchor, stop accessing its coordinate system. This will allow its underlying sensor data to be removed as necessary.

This is especially important for anchors you have persisted to the spatial anchor store. The sensor data behind these anchors will be kept around permanently to allow your app to find that anchor in future app sessions, which will reduce the space available to track other anchors. Persist only those anchors that you need to find again in future sessions and remove them from the store when they are no longer meaningful to the user.

See also

- [Persistence in Unity](#)
- [Spatial anchors in DirectX](#)
- [Coordinate systems](#)
- [Shared experiences in mixed reality](#)
- [Case study - Looking through holes in your reality](#)

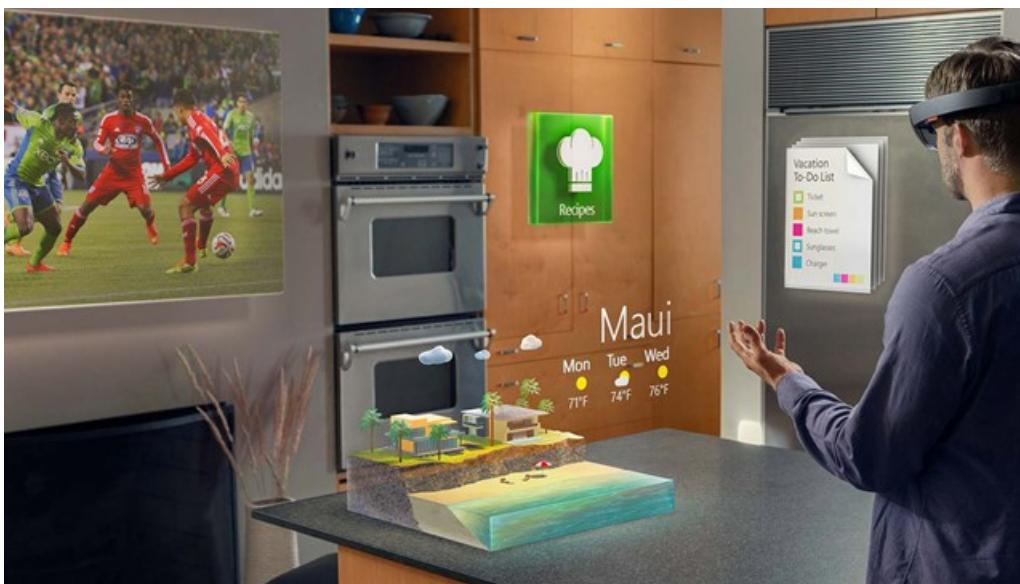
Types of mixed reality apps

11/6/2018 • 3 minutes to read • [Edit Online](#)

One of the advantages of developing apps for Windows Mixed Reality is that there is a spectrum of experiences that the platform can support. From fully immersive, virtual environments, to light information layering over a user's current environment, Windows Mixed Reality provides a robust set of tools to bring any experience to life. It is important for an app maker to understand early in their development process as to where along this spectrum their experience lies. This decision will ultimately impact both the app design makeup and the technological path for development.

Enhanced environment apps (HoloLens only)

One of the most powerful ways that mixed reality can bring value to users is by facilitating the placement of digital information or content in a user's current environment. This is an enhanced environment app. This approach is popular for apps where the contextual placement of digital content in the real world is paramount and/or keeping the user's real world environment "present" during their experience is key. This approach also allows users to easily move from real world tasks to digital tasks and back easily, lending even more credence to promise that the mixed reality apps the user sees are truly a part of their environment.



Enhanced environment apps

Example uses

- A mixed reality notepad style app that allows users to create and place notes around their environment
- A mixed reality television app placed in a comfortable spot for viewing
- A mixed reality cooking app placed above the kitchen island to assist in a cooking task
- A mixed reality app that gives users the feeling of "x-ray vision" (i.e. a hologram placed on top of and mimics a real world object, while allowing the user to see "inside it" holographically)
- Mixed reality annotations placed throughout a factory to give worker's necessary information
- Mixed reality wayfinding in an office space
- Mixed reality tabletop experiences (i.e. board game style experiences)
- Mixed reality communication apps like Skype

Blended environment apps

Given Windows Mixed Reality's ability to recognize and map the user's environment, it is capable of creating a digital layer that can be completely overlaid on the user's space. This layer respects the shape and boundaries of the user's environment, but the app may choose to transform certain elements best suited to immerse the user in the app. This is called a blended environment app. Unlike an enhanced environment app, blended environment apps may only care enough about the environment to best use its makeup for encouraging specific user behavior (like encouraging movement or exploration) or by replacing elements with changes (a kitchen counter is virtually skinned to show a different tile pattern). This type of experience may even transform an element into an entirely different object, but still retain the rough dimensions of the object as its base (a kitchen island is transformed into a dumpster for a crime thriller game).



Blended environment apps

Example uses

- A mixed reality interior design app that can paint walls, countertops or floors in different colors and patterns
- A mixed reality app that allows an automotive designer to layer new design iterations for an upcoming car refresh on top of an existing car
- A bed is "covered" and replaced by a mixed reality fruit stand in children's game
- A desk is "covered" and replaced with a mixed reality dumpster in a crime thriller game
- A hanging lantern is "covered" and replaced with signpost using roughly the same shape and dimension
- An app that allows users to blast holes in their real or immersive world walls, which reveal a magical world

Immersive environment apps

Immersive environment apps are centered around an environment that completely changes the user's world and can place them in a different time and space. These environments can feel very real, creating immersive and thrilling experiences that are only limited by the app creator's imagination. Unlike blended environment apps, once Windows Mixed Reality identifies the user's space, an immersive environment app may totally disregard the user's current environment and replace it whole stock with one of its own. These experiences may also completely separate time and space, meaning a user could walk the streets of Rome in an immersive experience, while remaining relatively still in their real world space. Context of the real world environment may not be important to an immersive environment app.



Immersive environment apps

Example uses

- An immersive app that lets a user tour a space completely separate from their own (i.e. walk through a famous building, museum, popular city)
- An immersive app that orchestrates an event or scenario around the user (i.e. a battle or a performance)

See also

- [Development overview](#)
- [App model](#)
- [App views](#)

App model

11/6/2018 • 10 minutes to read • [Edit Online](#)

Windows Mixed Reality uses the app model provided by the [Universal Windows Platform](#) (UWP), a model and environment for modern Windows apps. The UWP app model defines how apps are installed, updated, versioned and removed safely and completely. It governs the application life cycle - how apps execute, sleep and terminate - and how they can preserve state. It also covers integration and interaction with the operating system, files and other apps.



Apps with a 2D view arranged in the Windows Mixed Reality home

App lifecycle

The lifecycle of a mixed reality app involves standard app concepts such as placement, launch, termination and removal.

Placement is launch

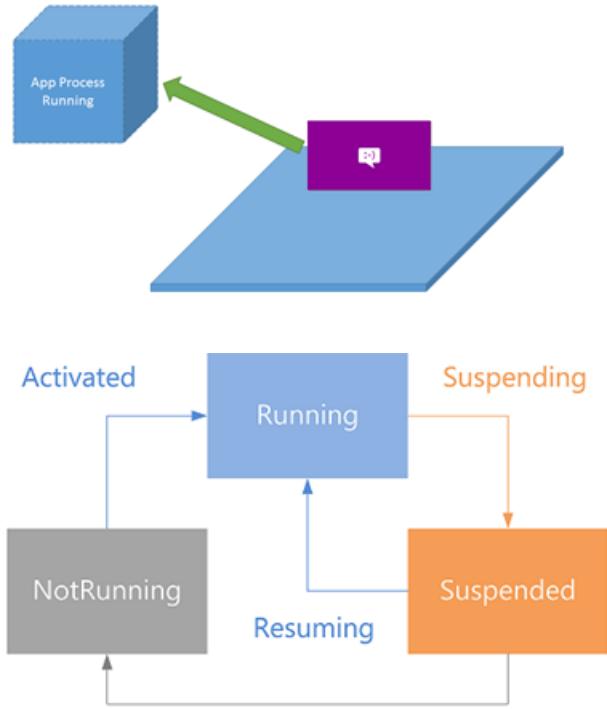
Every app starts in mixed reality by placing an app tile (just a [Windows secondary tile](#)) in the [Windows Mixed Reality home](#). These app tiles, on placement, will start running the app. These app tiles persist and stay at the location where they are placed, acting like launchers for anytime you want to get back to the app.



Placement puts a secondary tile in the world

As soon as placement completes (unless the placement was started by an [app to app](#) launch), the app starts launching. Windows Mixed Reality can run a limited number of apps at one time. As soon as you place and

launch an app, other active apps may suspend, leaving a screenshot of the app's last state on its app tile wherever you placed it. See [Windows 10 UWP app lifecycle](#) for more information on handling resume and other app life cycle events.



Left: after placing a tile, the app starts running. Right: state diagram for app running, suspended, or not running.

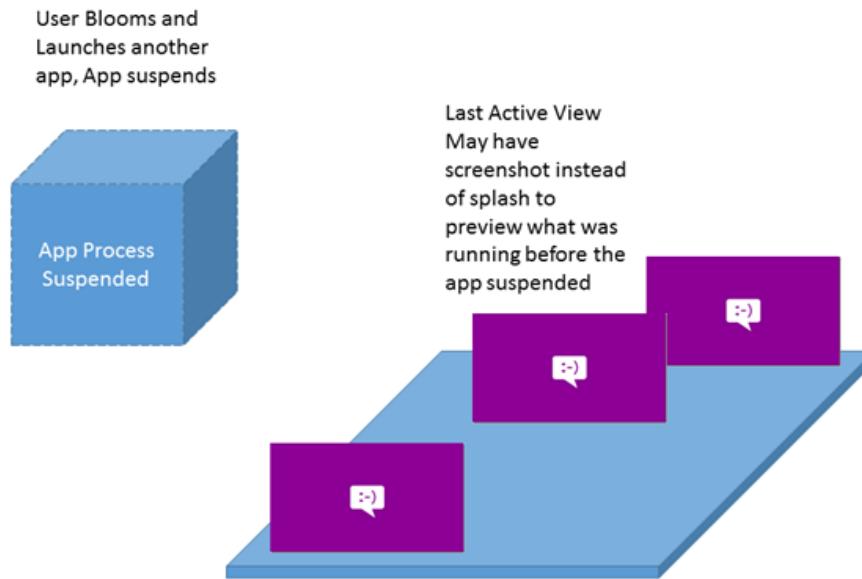
Remove is close/terminate process

When you remove a placed app tile from the world, this closes the underlying processes. This can be useful for ensuring your app is terminated or restarting a problematic app.

App suspension/termination

In the [Windows Mixed Reality home](#), the user is able to create multiple entry points for an app. They do this by launching your app from the Start menu and placing the app tile in the world. Each app tile behaves as a different entry point, and has a separate tile instance in the system. A query for [SecondaryTile.FindAllAsync](#) will result in a **SecondaryTile** for each app instance.

When a UWP app suspends, a screenshot is taken of the current state.



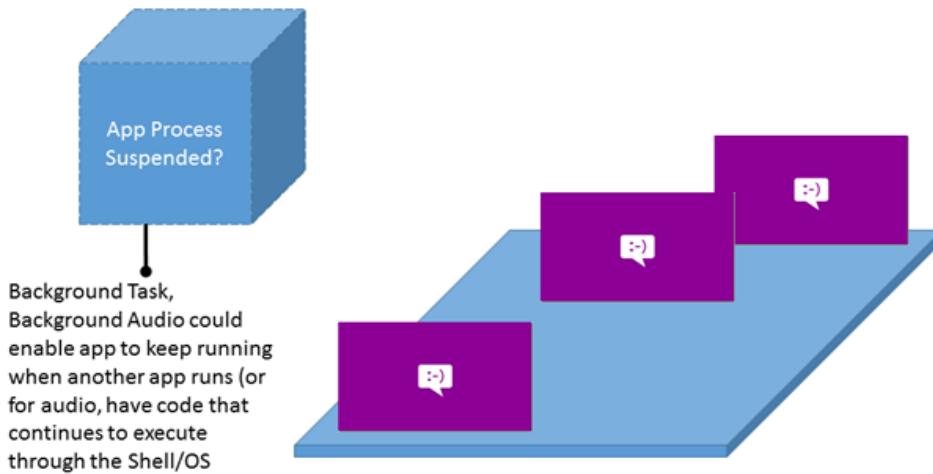
Screenshots are shown for suspended apps

One key difference from other Windows 10 shells is how the app is informed of an app instance activation via the [CoreApplication.Resuming](#) and [CoreWindow.Activated](#) events.

SCENARIO	RESUMING	ACTIVATED
Launch new instance of app from the Start menu		Activated with a new TileId
Launch second instance of app from the Start menu		Activated with a new TileId
Select the instance of the app that is not currently active		Activated with the TileId associated with the instance
Select a different app, then select the previously active instance	Resuming raised	
Select a different app, then select the instance that was previously inactive	Resuming raised	Then Activated with the TileId associated with the instance

Extended execution

Sometimes your app needs to continue doing work in the background or playing audio. [Background tasks](#) are available on HoloLens.

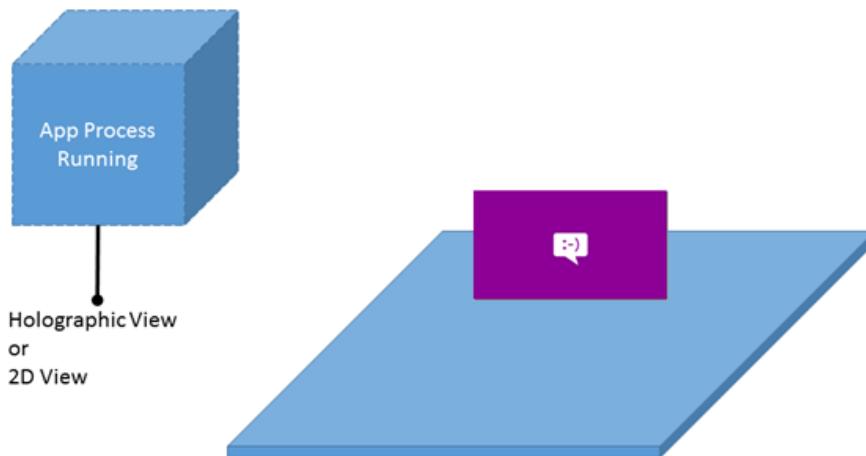


Apps can run in the background

App views

When your app activates, you can choose what type of view you'd like to display. For an app's **CoreApplication**, there is always a primary [app view](#) and any number of further app views you would like to create. On desktop, you can think of an app view as a window. Our mixed reality app templates create a Unity project where the primary app view is [immersive](#).

Your app can create an additional 2D app view using technology like XAML, to use Windows 10 features such as in-app purchase. If your app started as a UWP app for other Windows 10 devices, your primary view is 2D, but you could "light up" in mixed reality by adding an additional app view that's immersive to show an experience volumetrically. Imagine building a photo viewer app in XAML where the slideshow button switched to an immersive app view that flew photos from the app across the world and surfaces.



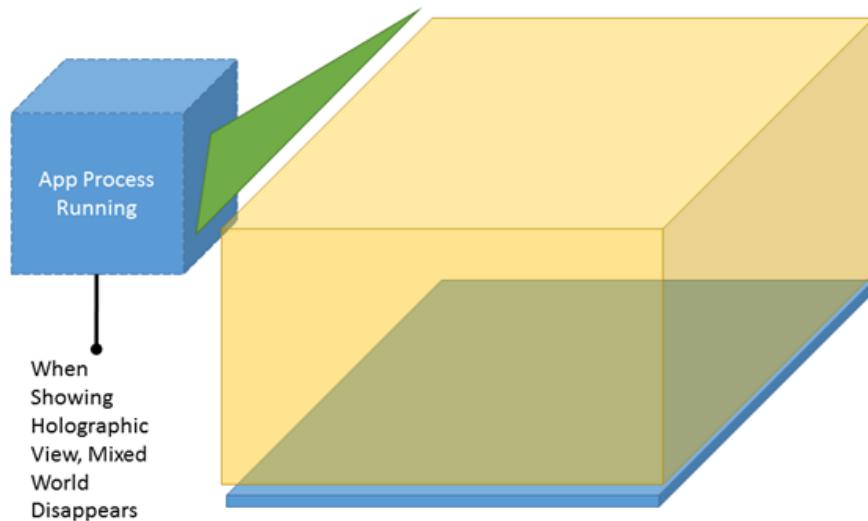
The running app can have a 2D view or an immersive view

Creating an immersive view

Mixed reality apps are those that create an immersive view, which is achieved with the [HolographicSpace](#) type.

An app that is purely immersive should always create an immersive view on launch, even if launched from the desktop. Immersive views always show up in the headset, regardless of where they were created from. Activating an immersive view will display the Mixed Reality Portal and guide the user to put on their headset.

An app that starts with a 2D view on the desktop monitor may create a secondary immersive view to show content in the headset. An example of this is a 2D Edge window on the monitor displaying a 360-degree video in the headset.

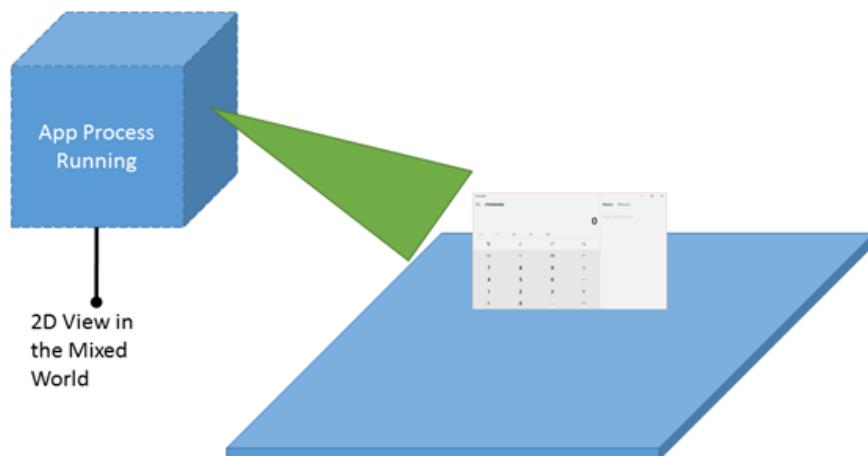


An app running in an immersive view is the only one visible

2D view in the Windows Mixed Reality home

Anything other than an immersive view is rendered as a 2D view in your world.

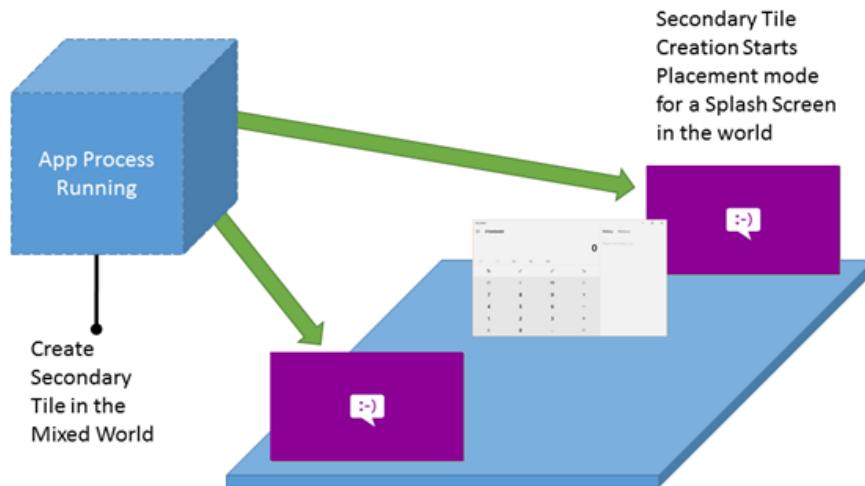
An app may have 2D views on both the desktop monitor and in the headset. Note that a new 2D view will be placed in the same shell as the view that created it, either on the monitor or in the headset. It is not currently possible for an app or a user to move a 2D view between the Mixed Reality home and the monitor.



Apps running in a 2D view share the space with other apps

Placement of additional app tiles

You can place as many apps with a 2D view in your world as you want with the [Secondary Tile APIs](#). These "pinned" tiles will appear as splash screens that users must place and then can later use to launch your app. Windows Mixed Reality does not currently support rendering any of the 2D tile content as live tiles.



Apps can have multiple placements using secondary tiles

Switching views

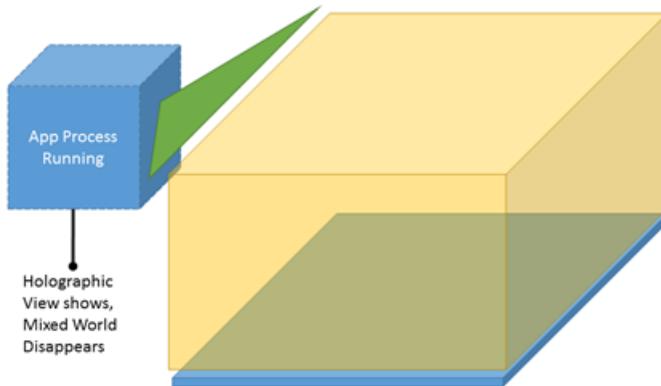
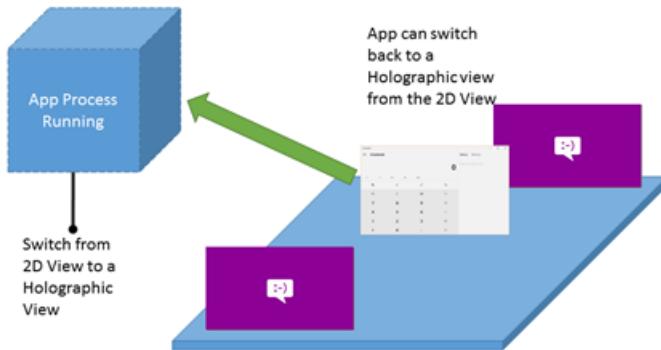
Switching from the 2D XAML view to the immersive view

If the app uses XAML, then the XAML [IFrameworkViewSource](#) will control the first view of the app. The app will need to switch to the immersive view before activating the [CoreWindow](#), to ensure the app launches directly into the immersive experience.

Use [CoreApplication.CreateNewView](#) and [ApplicationViewSwitcher.SwitchAsync](#) to make it the active view.

NOTE

- Do not specify the [ApplicationViewSwitchingOptions.ConsolidateViews](#) flag to **SwitchAsync** when switching from the XAML view to the immersive view, or the slate that launched the app will be removed from the world.
- **SwitchAsync** should be called using the [Dispatcher](#) associated with the view you are switching into.
- You will need to **SwitchAsync** back to the XAML view if you need to launch a virtual keyboard or want to activate another app.



Left: apps can switch between 2D view and immersive view. Right: when an app goes into an immersive view, the Windows Mixed Reality home and other apps disappear.

Switching from the immersive view back to a keyboard XAML view

One common reason for switching back-and-forth between views is displaying a keyboard in a mixed reality app. The shell is only able to display the system keyboard if the app is showing a 2D view. If the app needs to get text input, it may provide a custom XAML view with a text input field, switch to it, and then switch back after the input is complete.

Like in the previous section, you can use **ApplicationViewSwitcher.SwitchAsync** to transition back to a XAML view from your immersive view.

App size

2D app views always appear in a fixed virtual slate. This makes all 2D views show the exact same amount of content. Here are some further details about the size of your app's 2D view:

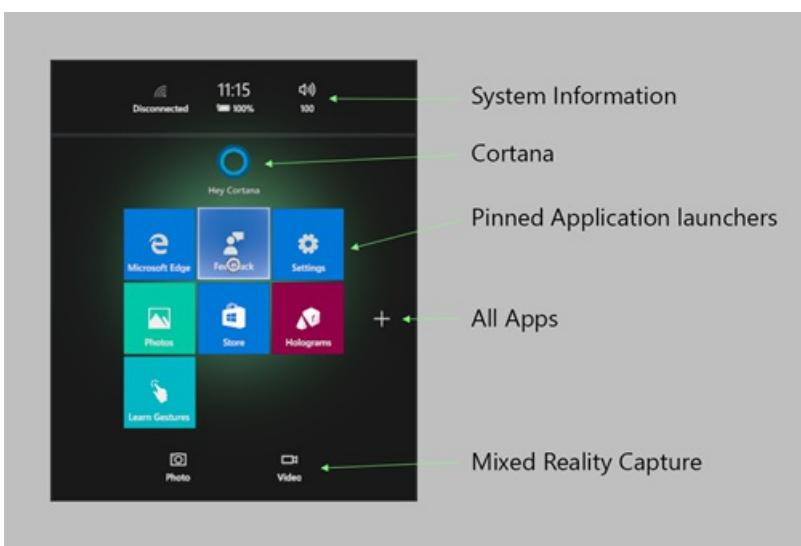
- The aspect ratio of the app is preserved while resizing.
- App [resolution and scale factor](#) are not changed by resizing.
- Apps are not able to query their actual size in the world.



Apps with a 2D view appear with fixed window sizes

App tiles

The Start menu uses the standard small tile and medium tile for pins and the **All Apps** list in mixed reality.



The Start menu for Windows Mixed Reality

App to app interactions

As you build apps, you have access to the rich app to app communication mechanisms available on Windows 10. Many of the new Protocol APIs and file registrations work perfectly on HoloLens to enable app launching and communication.

Note that for desktop headsets, the app associated with a given file extension or protocol may be a Win32 app that can only appear on the desktop monitor or in the desktop slate.

Protocols

HoloLens supports app to app launching via the [Windows.System.Launcher APIs](#).

There are some things to consider when launching another application:

- When doing a non-modal launch, such as [LaunchUriAsync](#), the user must place the app before interacting with it.
- When doing a modal launch, such as through [LaunchUriForResultsAsync](#), the modal app is placed on top of the window.
- Windows Mixed Reality cannot overlay applications on top of exclusive views. In order to show the launched app, Windows takes the user back to the world to display the application.

File pickers

HoloLens supports both [FileOpenPicker](#) and [FileSavePicker](#) contracts. However, no app comes pre-installed that fulfills the file picker contracts. These apps - OneDrive, for example - can be installed from the Microsoft Store.

If you have more than one file picker app installed, you will not see any disambiguation UI for choosing which app to launch; instead, the first file picker installed will be chosen. When saving a file, the filename is generated which includes the timestamp. This cannot be changed by the user.

By default, the following extensions are supported locally:

APP	EXTENSIONS
Photos	bmp, gif, jpg, png, avi, mov, mp4, wmv
Microsoft Edge	htm, html, pdf, svg, xml

App contracts and Windows Mixed Reality extensions

App contracts and extension points allow you to register your app to take advantage of deeper operating system features like handling a file extension or using background tasks. This is a list of the supported and unsupported contracts and extension points on HoloLens.

CONTRACT OR EXTENSION	SUPPORTED?
Account Picture Provider (extension)	Unsupported
Alarm	Unsupported
App service	Supported but not fully functional
Appointments provider	Unsupported
AutoPlay (extension)	Unsupported
Background tasks (extension)	Partially Supported (not all triggers work)
Update task (extension)	Supported
Cached file updater contract	Supported
Camera settings (extension)	Unsupported
Dial protocol	Unsupported
File activation (extension)	Supported
File Open Picker contract	Supported
File Save Picker contract	Supported
Lock screen call	Unsupported
Media playback	Unsupported

CONTRACT OR EXTENSION	SUPPORTED?
Play To contract	Unsupported
Preinstalled config task	Unsupported
Print 3D Workflow	Supported
Print task settings (extension)	Unsupported
URI activation (extension)	Supported
Restricted launch	Unsupported
Search contract	Unsupported
Settings contract	Unsupported
Share contract	Unsupported
SSL/certificates (extension)	Supported
Web account provider	Supported

App file storage

All storage is through the [Windows.Storage namespace](#). See the following documentation for more details.
HoloLens does not support app storage sync/roaming.

- [Files, folders, and libraries](#)
- [Store and retrieve settings and other app data](#)

Known folders

See [KnownFolders](#) for the full details for UWP apps.

PROPERTY	SUPPORTED ON HOOLENS	SUPPORTED ON IMMERSIVE HEADSETS	DESCRIPTION
AppCaptures	✓ <input type="checkbox"/>	✓ <input type="checkbox"/>	Gets the App Captures folder.
CameraRoll	✓ <input type="checkbox"/>	✓ <input type="checkbox"/>	Gets the Camera Roll folder.
DocumentsLibrary	✓ <input type="checkbox"/>	✓ <input type="checkbox"/>	Gets the Documents library. The Documents library is not intended for general use.
MusicLibrary	✓ <input type="checkbox"/>	✓ <input type="checkbox"/>	Gets the Music library.
Objects3D	✓ <input type="checkbox"/>	✓ <input type="checkbox"/>	Gets the Objects 3D folder.
PicturesLibrary	✓ <input type="checkbox"/>	✓ <input type="checkbox"/>	Gets the Pictures library.

Playlists	✓□	✓□	Gets the play lists folder.
SavedPictures	✓□	✓□	Gets the Saved Pictures folder.
VideosLibrary	✓□	✓□	Gets the Videos library.
HomeGroup		✓□	Gets the HomeGroup folder.
MediaServerDevices		✓□	Gets the folder of media server (Digital Living Network Alliance (DLNA)) devices.
RecordedCalls		✓□	Gets the recorded calls folder.
RemovableDevices		✓□	Gets the removable devices folder.

App package

With Windows 10 you no longer target an operating system but instead [target your app to one or more device families](#). A device family identifies the APIs, system characteristics, and behaviors that you can expect across devices within the device family. It also determines the set of devices on which your app can be installed from the [Microsoft Store](#).

- To target both desktop headsets and HoloLens, target your app to the **Windows.Universal** device family.
- To target just desktop headsets, target your app to the **Windows.Desktop** device family.
- To target just HoloLens, target your app to the **Windows.Holographic** device family.

See also

- [App views](#)
- [Updating 2D UWP apps for mixed reality](#)
- [3D app launcher design guidance](#)
- [Implementing 3D app launchers](#)

App views

11/6/2018 • 3 minutes to read • [Edit Online](#)

Windows apps can contain two kinds of views, **immersive views** and **2D views**. Apps can switch between their various immersive views and 2D views, showing their 2D views either on a monitor as a window or in a headset as a slate. Apps that have at least one immersive view are categorized as **mixed reality apps**. Apps that never have a immersive view are **2D apps**.

Immersive views

An immersive view gives your app the ability to create holograms in the world around you or immerse the user in a virtual environment. When an app is drawing in the immersive view, no other app is drawing at the same time—holograms from multiple apps are not composited together. By continually adjusting the perspective from which your [app renders](#) its scene to match the user's head movements, your app can render [world-locked](#) holograms that remain at a fixed point in the real world, or it can render a virtual world that holds its position as a user moves within it.



When in an immersive view, holograms can be placed in the world around you

On [HoloLens](#), your app renders its holograms on top of the user's real-world surroundings. On a [Windows Mixed Reality immersive headset](#), the user cannot see the real world, and so your app must render everything the user will see.

The [Windows Mixed Reality home](#) (including the Start menu and holograms you've placed around the environment) does not render while in an immersive view either. On HoloLens, any system notifications that occur while an immersive view is showing will be relayed audibly by Cortana, and the user can respond with voice input.

While in an immersive view, your app is also responsible for handling all input. Input in Windows Mixed Reality is made up of [gaze](#), [gesture](#) (HoloLens only), [voice](#) and [motion controllers](#) (immersive headsets only).

2D views



Multiple apps with a 2D view placed around the Windows Mixed Reality home

An app with a 2D view appears in the [Windows Mixed Reality home](#) (sometimes called the "shell") as a virtual slate, rendered alongside the app launchers and other holograms the user has placed in their world. The user can adjust this slate to move and scale it, though it remains at a fixed resolution regardless of its size. If your app's first view is a 2D view, your 2D content will fill the same slate used to launch the app.

In a desktop headset, you can run any Universal Windows Platform (UWP) apps that run on your desktop monitor today. These apps are already rendering 2D views today, and their content will automatically appear on a slate in the user's world when launched. 2D UWP apps can target the **Windows.Universal** device family to run on both desktop headsets and on HoloLens as slates.

One key use of 2D views is to show a text entry form that can make use of the system keyboard. Because the shell cannot render on top of an immersive view, the app must switch to a 2D view to show the system keyboard. Apps that want to accept text input can switch to a 2D view with a text box. While that text box has focus, the system will show the system keyboard, allowing the user to enter text.

Note that on a desktop PC, an app can have 2D views on both the desktop monitor and in an attached headset. For example, you can browse Edge on your desktop monitor using its main 2D view to find a 360-degree video. When you play that video, Edge will launch a secondary immersive view inside the headset to display the immersive video content.

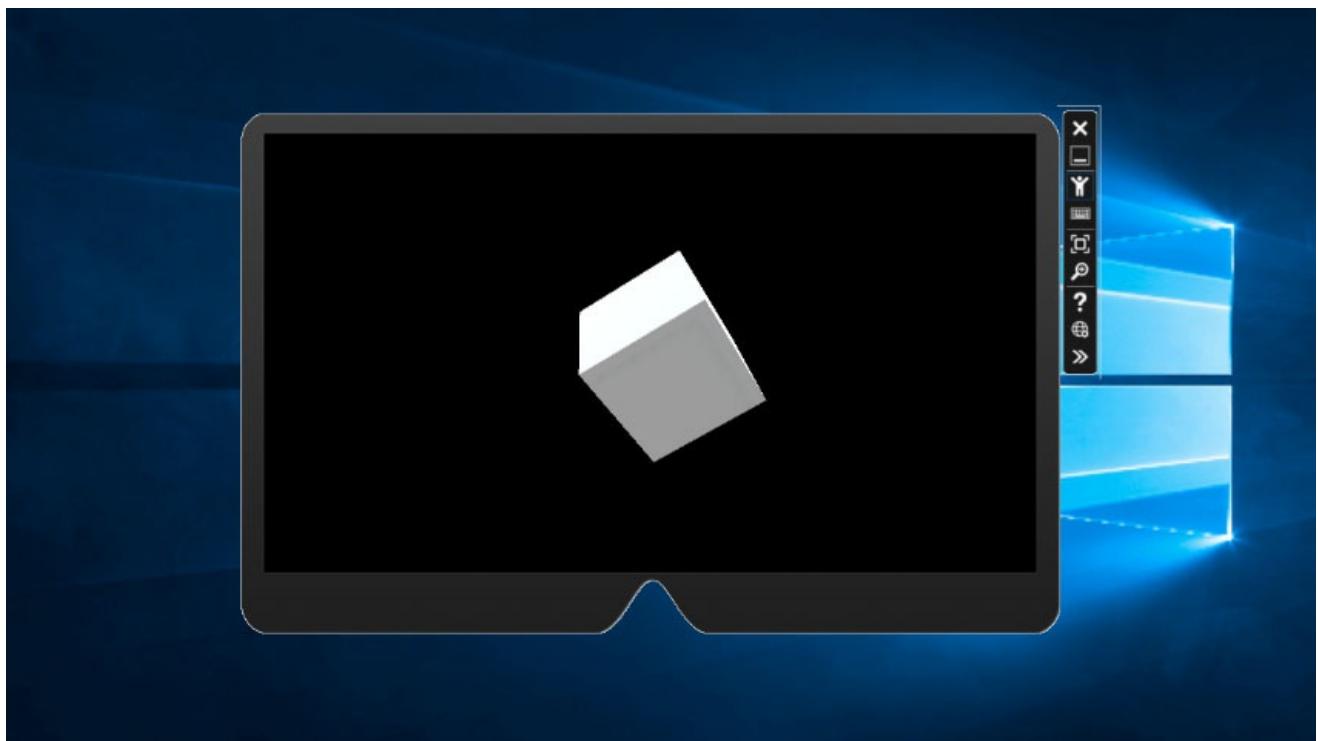
See also

- [App model](#)
- [Updating 2D UWP apps for mixed reality](#)
- [Getting a HolographicSpace](#)
- [Navigating the Windows Mixed Reality home](#)
- [Types of mixed reality apps](#)

The Mixed Reality Academy is a set of online step-by-step tutorials with corresponding project files:

- The tutorials cover 100, 200, and 300 level topics, in which: 100-level covers project basics, 200-level covers core MR building blocks, and 300-level covers cloud service integration.
- Most courses cover concepts applicable to both HoloLens and immersive (VR) headsets.
- Each tutorial is organized by chapter, and most include video demonstrations of the key concepts.
- A Windows 10 PC with the correct [tools installed](#) is a common prerequisite to complete each tutorial.

We also have an in-person Mixed Reality Academy at the Reactor space in San Francisco. If you're looking for information about the physical Academy space, or upcoming events, [click here](#) or scroll to the bottom of this page.



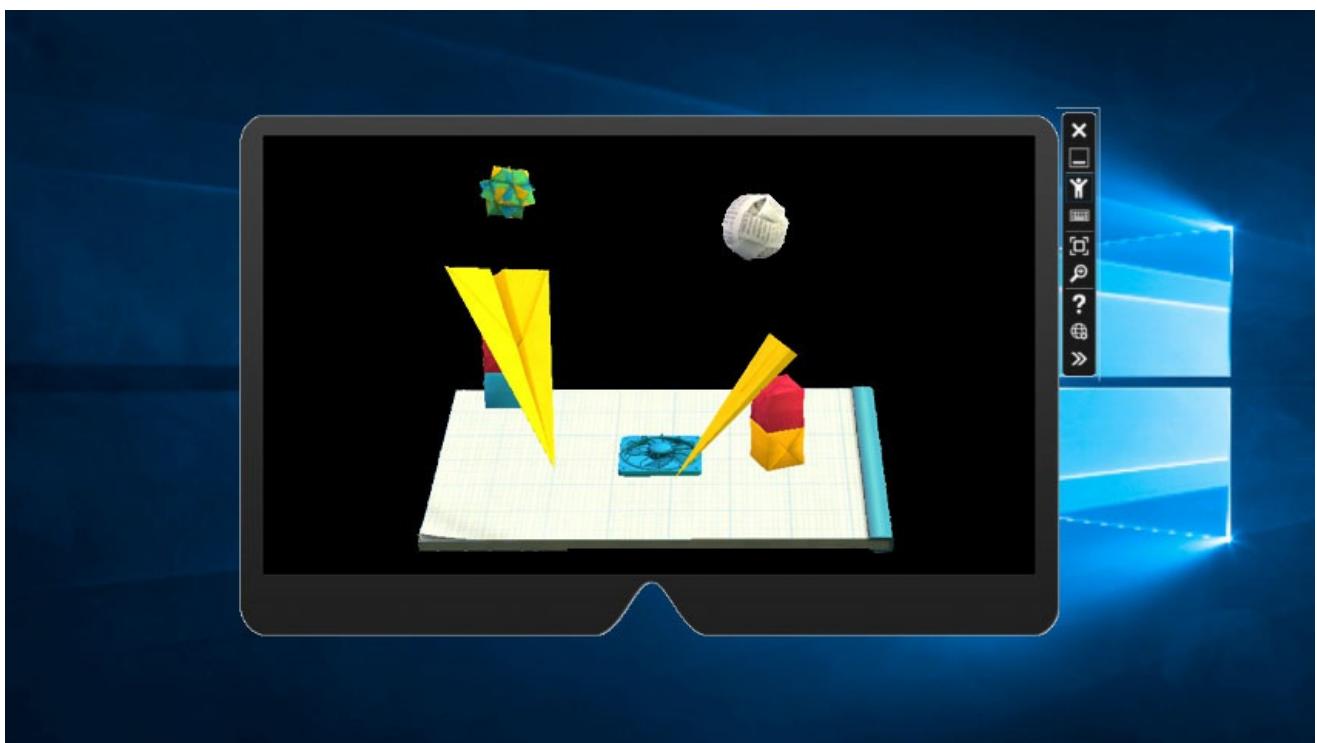
MR Basics 100: Getting started with Unity

Create a basic mixed reality app with Unity. This project can then serve as a starting template for any MR app you might want to build in Unity.



MR Basics 101: Complete project with device

Set up a complete project, introducing core mixed reality features (gaze, gesture, voice, spatial sound, and spatial mapping) using a HoloLens device.



MR Basics 101E: Complete project with emulator

Set up a complete project, introducing core mixed reality features (gaze, gesture, voice, spatial sound, and spatial mapping) using the HoloLens emulator.



MR Input 210: Gaze

Gaze is the first form of input, and reveals the user's intent and awareness. You will add contextual awareness to your cursor and holograms, taking full advantage of what your app knows about the user's gaze.



MR Input 211: Gesture

Gestures turn user intention into action. With gestures, users can interact with holograms. In this course, you will learn to track the user's hands, respond to user input, and give feedback based on hand state and location.



MR Input 212: Voice

Voice allows us to interact with our holograms in an easy and natural way. In this course, you will learn to make users aware of available voice commands, give feedback that a voice command was heard, and use dictation to understand what the user is saying.



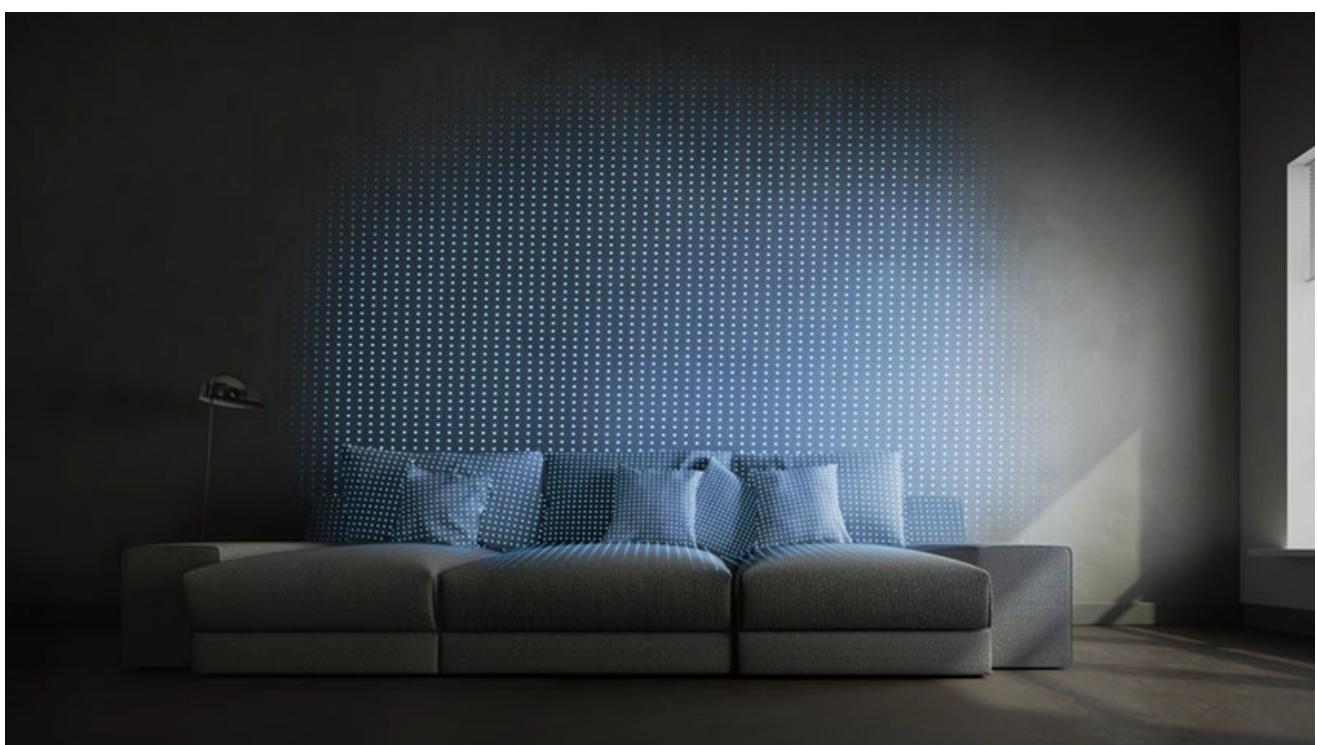
MR Input 213: Motion controllers

This course will explore ways of visualizing motion controllers in immersive (VR) headsets, handling input events, and attaching custom UI elements to the controllers.



MR Spatial 220: Spatial sound

Spatial sound breathes life into holograms and gives them presence. In this course, you will learn to use spatial sound to ground holograms in the surrounding world, give feedback during interactions, and use audio to find your holograms.



MR Spatial 230: Spatial mapping

Spatial mapping brings the real world and virtual world together. You'll explore shaders and use them to visualize your space. Then you'll learn to simplify the room mesh into simple planes, give feedback on placing holograms on real-world surfaces, and explore occlusion visual effects.



MR Sharing 240: Multiple HoloLens devices

Our //Build 2016 project! Set up a complete project with coordinate systems shared between HoloLens devices, allowing users to take part in a shared holographic world.



MR Sharing 250: HoloLens and immersive headsets

In our //Build 2017 project, we demonstrate building an app that leverages the unique strengths of HoloLens and immersive (VR) headsets within a shared, cross-device experience.



MR and Azure 301: Language translation

Using the Azure Translator Text API, your mixed reality app can translate speech to text in another language. Learn how in this course!



MR and Azure 302: Computer vision

Use Azure Computer Vision APIs in a mixed reality app for image processing and analysis, without training a model.



MR and Azure 302b: Custom vision

Learn how to train a machine learning model, and use the trained model for image processing and analysis.



MR and Azure 303: Natural language understanding

This course will teach you how to use the Azure Language Understanding (LUIS) service to add natural language understanding into your mixed reality app.



MR and Azure 304: Face recognition

Learn how to use the Azure Face API to perform face detection and recognition in your mixed reality app.



MR and Azure 305: Functions and storage

In this course you will learn how to create and use Azure Functions, and store data within Azure Storage, within a mixed reality app.



MR and Azure 306: Streaming video

Learn how to use Azure Media Services to stream 360-degree video within a Windows Mixed Reality immersive (VR) experience.



MR and Azure 307: Machine learning

Leverage Azure Machine Learning Studio within your mixed reality app to deploy a large number of machine learning (ML) algorithms.



MR and Azure 308: Cross-device notifications

In this course, you'll learn how to use several Azure services to deliver push notifications and scene changes from a PC app to a mixed reality app.



MR and Azure 309: Application insights

Use the Azure Application Insights service to collect analytics on user behavior within a mixed reality app.



MR and Azure 310: Object detection

Train a machine learning model, and use the trained model to recognize similar objects and their positions in the physical world.



MR and Azure 311: Microsoft Graph

Learn how to connect to Microsoft Graph services from within a mixed reality app.



MR and Azure 312: Bot integration

Create and deploy a bot using Microsoft Bot Framework v4, and communicate with it in a mixed reality app.



MR and Azure 313: IoT Hub Service

Learn how to implement Azure IoT Hub service on a virtual machine, and visualize the data on HoloLens.

Microsoft Reactor



The Microsoft Reactor in San Francisco

The Microsoft Reactor in San Francisco, located at 680 Folsom in SOMA, serves as the flagship location for the Mixed Reality Capture Studio and the Mixed Reality Academy. It is a place where developers and creators can begin their journey building mixed reality experiences for Microsoft HoloLens and Windows Mixed Reality headsets.

[Check for announcements of upcoming sessions, workshops, and hackathons](#)

MR Basics 100: Getting started with Unity

11/13/2018 • 9 minutes to read • [Edit Online](#)

This tutorial will walk you through creating a basic mixed reality app built with Unity.

Device support

COURSE	HOLOLENS	IMMERSIVE HEADSETS
MR Basics 100: Getting started with Unity	✓ <input type="checkbox"/>	✓ <input type="checkbox"/>

Prerequisites

- A Windows 10 PC configured with the correct [tools installed](#).

Chapter 1 - Create a New Project

To build an app with Unity, you first need to create a project. This project is organized into a few folders, the most important of which is your Assets folder. This is the folder that holds all assets you import from digital content creation tools such as Maya, Max Cinema 4D or Photoshop, all code you create with Visual Studio or your favorite code editor, and any number of content files that Unity creates as you compose scenes, animations and other Unity asset types in the editor.

To build and deploy UWP apps, Unity can export the project as a Visual Studio solution that will contain all necessary asset and code files.

1. Start Unity
2. Select **New**
3. Enter a project name (e.g. "MixedRealityIntroduction")
4. Enter a location to save your project
5. Ensure the **3D** toggle is selected
6. Select **Create project**

Congrats, you are all setup to get started with your mixed reality customizations now.

Chapter 2 - Setup the Camera

The Unity Main Camera handles head tracking and stereoscopic rendering. There are a few changes to make to the Main Camera to use it with mixed reality.

1. Select File > New Scene

First, it will be easier to lay out your app if you imagine the starting position of the user as (**X**: 0, **Y**: 0, **Z**: 0). Since the Main Camera is tracking movement of the user's head, the starting position of the user can be set by setting the starting position of the Main Camera.

1. Select **Main Camera** in the **Hierarchy** panel

2. In the **Inspector** panel, find the **Transform** component and change the **Position** from (**X**: 0, **Y**: 1, **Z**: -10) to (**X**: 0, **Y**: 0, **Z**: 0)

Second, the default Camera background needs some thought.

For HoloLens applications, the real world should appear behind everything the camera renders, not a Skybox texture.

1. With the **Main Camera** still selected in the **Hierarchy** panel, find the **Camera** component in the **Inspector** panel and change the **Clear Flags** dropdown from **Skybox** to **Solid Color**.
2. Select the **Background** color picker and change the **RGBA** values to (0, 0, 0, 0)

For mixed reality applications targeted to immersive headsets, we can use the default Skybox texture that Unity provides.

1. With the **Main Camera** still selected in the **Hierarchy** panel, find the **Camera** component in the **Inspector** panel and keep the **Clear Flags** dropdown to **Skybox**.

Third, let us consider the near clip plane in Unity and prevent objects from being rendered too close to the users eyes as a user approaches an object or an object approaches a user.

For HoloLens applications, the near clip plane can be set to the [HoloLens recommended](#) 0.85 meters.

1. With the **Main Camera** still selected in the **Hierarchy** panel, find the **Camera** component in the **Inspector** panel and change the **Near Clip Plane** field from the default **0.3** to the HoloLens recommended **0.85**.

For mixed reality applications targeted to immersive headsets, we can use the default setting that Unity provides.

1. With the **Main Camera** still selected in the **Hierarchy** panel, find the **Camera** component in the **Inspector** panel and keep the **Near Clip Plane** field to the default **0.3**.

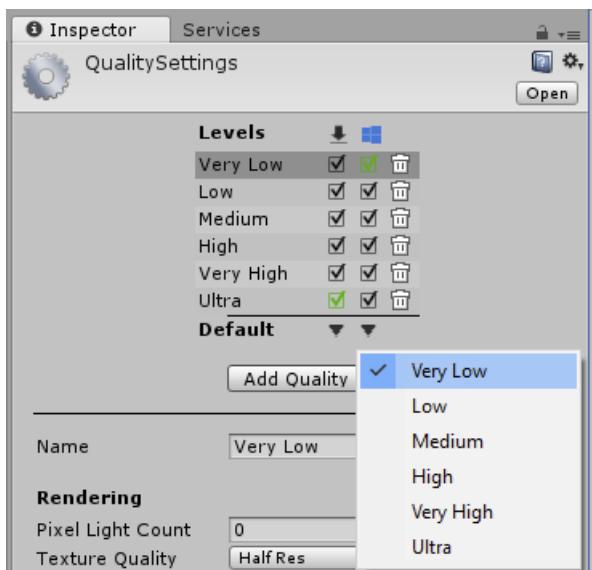
Finally, let us save our progress so far. To save the scene changes, select **File > Save Scene As**, name the scene **Main**, and select **Save**.

Chapter 3 - Setup the Project Settings

In this chapter, we will set some Unity project settings that help us target the Windows Holographic SDK for development. We will also set some quality settings for our application. Finally, we will ensure our build targets are set to Windows Store.

Unity performance and quality settings

Unity quality settings for HoloLens



Since maintaining high framerate on HoloLens is so important, we want the quality settings tuned for fastest performance. For more detailed performance information, [Performance recommendations for Unity](#).

1. Select **Edit > Project Settings > Quality**
2. Select the **dropdown** under the **Windows Store** logo and select **Very Low**. You'll know the setting is applied correctly when the box in the Windows Store column and Fastest row is green.

For mixed reality applications targeted to occluded displays, you can leave the quality settings to its default values.

Target Windows 10 SDK

Target Windows Holographic SDK



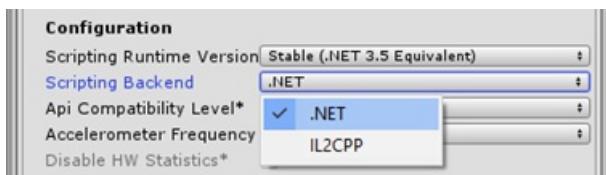
We need to let Unity know that the app we are trying to export should create an [immersive view](#) instead of a 2D view. We do this by enabling Virtual Reality support on Unity targeting the Windows 10 SDK.

1. Go to **Edit > Project Settings > Player**.
2. In the **Inspector Panel** for Player Settings, select the **Windows Store** icon.
3. Expand the **XR Settings** group.
4. In the **Rendering** section, check the **Virtual Reality Supported** checkbox to add a new **Virtual Reality SDKs** list.
5. Verify that **Windows Mixed Reality** appears in the list. If not, select the **+** button at the bottom of the list and choose **Windows Mixed Reality**.

NOTE

If you do not see the **Windows Store** icon, double check to make sure you selected the Windows Store .NET Scripting Backend prior to installation. If not, you may need to reinstall Unity with the correct Windows installation.

Verify .NET Configuration



1. Go to **Edit > Project Settings > Player** (you may still have this up from the previous step).
2. In the **Inspector Panel** for Player Settings, select the **Windows Store** icon.
3. In the **Other Settings** Configuration section, make sure that **Scripting Backend** is set to **.NET**

Awesome job on getting all the project settings applied. Next, let us add a hologram!

Chapter 4 - Create a cube

Creating a cube in your Unity project is just like creating any other object in Unity. Placing a cube in front of the user is easy because Unity's coordinate system is mapped to the real world - where one meter in Unity is approximately one meter in the real world.

1. In the top left corner of the **Hierarchy** panel, select the **Create** dropdown and choose **3D Object > Cube**.
2. Select the newly created **Cube** in the **Hierarchy** panel
3. In the **Inspector** find the **Transform** component and change **Position** to **(X: 0, Y: 0, Z: 2)**. *This positions the cube 2 meters in front of the user's starting position.*
4. In the **Transform** component, change **Rotation** to **(X: 45, Y: 45, Z: 45)** and change **Scale** to **(X: 0.25, Y: 0.25, Z: 0.25)**. *This scales the cube to 0.25 meters.*
5. To save the scene changes, select **File > Save Scene**.

Chapter 5 - Verify on device from Unity editor

Now that we have created our cube, it is time to do a quick check in device. You can do this directly from within the Unity editor.

For HoloLens use Unity Remoting

1. On your HoloLens, install and run the [Holographic Remoting Player](#), available from the Windows Store.
Launch the application on the device, and it will enter a waiting state and show the IP address of the device.
Note down the IP.
2. On your development PC, in Unity, open **File > Build Settings** window.
3. Change **Platform** to **Universal Windows Platform** and click **Switch Platform**.
4. Open **Window > XR > Holographic Emulation**.
5. Change **Emulation Mode** from **None** to **Remote to Device**.
6. In **Remote Machine**, enter the IP address of your HoloLens noted earlier.
7. Click **Connect**.
8. Ensure the **Connection Status** changes to green **Connected**.
9. Now you can now click **Play** in the Unity editor.

You will now be able to see the cube in device and in the editor. You can pause, inspect objects, and debug just like you are running an app in the editor, because that's essentially what's happening, but with video, audio, and device input transmitted back and forth across the network between the host machine and the device.

For other mixed reality supported headsets

1. Connect the headset to your development PC using the USB cable and the HDMI or display port cable.
2. Launch the **Mixed Reality Portal** and ensure you have completed the first run experience.

3. From Unity, you can now press the Play button.

You will now be able to see the cube rendering in your mixed reality headset and in the editor.

Chapter 6 - Build and deploy to device from Visual Studio

We are now ready to compile our project to Visual Studio and deploy to our target device.

Export to the Visual Studio solution

1. Open **File > Build Settings** window.
2. Click **Add Open Scenes** to add the scene.
3. Change **Platform** to **Universal Windows Platform** and click **Switch Platform**.
4. In **Windows Store** settings ensure, **SDK is Universal 10**.
5. For Target device, leave to **Any Device** for occluded displays or switch to **HoloLens**.
6. **UWP Build Type** should be **D3D**.
7. **UWP SDK** could be left at **Latest installed**.
8. Check **Unity C# Projects** under Debugging.
9. Click **Build**.
10. In the file explorer, click **New Folder** and name the folder "**App**".
11. With the **App** folder selected, click the **Select Folder** button.
12. When Unity is done building, a Windows File Explorer window will appear.
13. Open the **App** folder in file explorer.
14. Open the generated Visual Studio solution (MixedRealityIntroduction.sln in this example)

Compile the Visual Studio solution

Finally, we will compile the exported Visual Studio solution, deploy it, and try it out on the device.

1. Using the top toolbar in Visual Studio, change the target from **Debug** to **Release** and from **ARM** to **X86**.

The instructions differ for deploying to a device versus the emulator. Follow the instructions that match your setup.

Deploy to mixed reality device over Wi-Fi

1. Click on the arrow next to the **Local Machine** button, and change the deployment target to **Remote Machine**.
2. Enter the IP address of your mixed reality device and change **Authentication Mode** to Universal (Unencrypted Protocol) for HoloLens and **Windows** for other devices.
3. Click **Debug > Start without debugging**.

For HoloLens, If this is the first time deploying to your device, you will need to pair [using Visual Studio](#).

Deploy to mixed reality device over USB

Ensure your device is plugged in via the USB cable.

1. **For HoloLens**, click on the arrow next to the **Local Machine** button, and change the deployment target to **Device**.
2. **For targeting occluded devices attached to your PC**, keep the setting to Local Machine. Ensure you have the **Mixed Reality Portal** running.
3. Click **Debug > Start without debugging**.

Deploy to Emulator

1. Click on the arrow next to the **Device** button, and from drop down select **HoloLens Emulator**.
2. Click **Debug > Start without debugging**.

Try out your app

Now that your app is deployed, try moving all around the cube and observe that it stays in the world in front of you.

See also

- [Unity development overview](#)
- [Best practices for working with Unity and Visual Studio](#)
- [MR Basics 101](#)
- [MR Basics 101E](#)

MR Basics 101: Complete project with device

11/6/2018 • 15 minutes to read • [Edit Online](#)

This tutorial will walk you through a complete project, built in Unity, that demonstrates core Windows Mixed Reality features on HoloLens including [gaze](#), [gestures](#), [voice input](#), [spatial sound](#) and [spatial mapping](#).

The tutorial will take approximately 1 hour to complete.

Device support

COURSE	HOLOLENS	IMMERSIVE HEADSETS
MR Basics 101: Complete project with device	✓ <input type="checkbox"/>	

Before you start

Prerequisites

- A Windows 10 PC configured with the correct [tools installed](#).
- A HoloLens device [configured for development](#).

Project files

- Download the [files](#) required by the project. Requires Unity 2017.2 or later.
 - If you still need Unity 5.6 support, please use [this release](#).
 - If you still need Unity 5.5 support, please use [this release](#).
 - If you still need Unity 5.4 support, please use [this release](#).
- Un-archive the files to your desktop or other easy to reach location. Keep the folder name as **Origami**.

NOTE

If you want to look through the source code before downloading, it's [available on GitHub](#).

Chapter 1 - "Holo" world

In this chapter, we'll setup our first Unity project and step through the build and deploy process.

Objectives

- Set up Unity for holographic development.
- Make a hologram.
- See a hologram that you made.

Instructions

- Start Unity.
- Select **Open**.
- Enter location as the **Origami** folder you previously un-archived.
- Select **Origami** and click **Select Folder**.

- Since the **Origami** project does not contain a scene, save the empty default scene to a new file using: **File / Save Scene As**.
- Name the new scene **Origami** and press the **Save** button.

Setup the main virtual camera

- In the **Hierarchy Panel**, select **Main Camera**.
- In the **Inspector** set its transform position to **0,0,0**.
- Find the **Clear Flags** property, and change the dropdown from **Skybox** to **Solid color**.
- Click on the **Background** field to open a color picker.
- Set **R, G, B, and A** to **0**.

Setup the scene

- In the **Hierarchy Panel**, click on **Create** and **Create Empty**.
- Right-click the new **GameObject** and select **Rename**. Rename the GameObject to **OrigamiCollection**.
- From the **Holograms** folder in the Project Panel (expand Assets and select Holograms or double click the Holograms folder in the Project Panel):
 - Drag **Stage** into the Hierarchy to be a child of **OrigamiCollection**.
 - Drag **Sphere1** into the Hierarchy to be a child of **OrigamiCollection**.
 - Drag **Sphere2** into the Hierarchy to be a child of **OrigamiCollection**.
- Right-click the **Directional Light** object in the **Hierarchy Panel** and select **Delete**.
- From the **Holograms** folder, drag **Lights** into the root of the **Hierarchy Panel**.
- In the **Hierarchy**, select the **OrigamiCollection**.
- In the **Inspector**, set the transform position to **0, -0.5, 2.0**.
- Press the **Play** button in Unity to preview your holograms.
- You should see the Origami objects in the preview window.
- Press **Play** a second time to stop preview mode.

Export the project from Unity to Visual Studio

- In Unity select **File > Build Settings**.
- Select **Universal Windows Platform** in the **Platform** list and click **Switch Platform**.
- Set **SDK** to **Universal 10** and **Build Type** to **D3D**.
- Check **Unity C# Projects**.
- Click **Add Open Scenes** to add the scene.
- Click **Build**.
- In the file explorer window that appears, create a **New Folder** named "App".
- Single click the **App Folder**.
- Press **Select Folder**.
- When Unity is done, a File Explorer window will appear.
- Open the **App** folder.
- Open (double click) **Origami.sln**.
- Using the top toolbar in Visual Studio, change the target from Debug to **Release** and from ARM to **X86**.
- Click on the arrow next to the Device button, and select **Remote Machine** to deploy over Wi-Fi.
 - Set the **Address** to the name or IP address of your HoloLens. If you do not know your device IP address, look in **Settings > Network & Internet > Advanced Options** or ask Cortana "**Hey Cortana, What's my IP address?**"
 - If the HoloLens is attached over USB, you may instead select **Device** to deploy over USB.
 - Leave the **Authentication Mode** set to **Universal**.
 - Click **Select**
- Click **Debug > Start Without debugging** or press **Ctrl + F5**. If this is the first time deploying to your

device, you will need to [pair it with Visual Studio](#).

- The Origami project will now build, deploy to your HoloLens, and then run.
- Put on your HoloLens and look around to see your new holograms.

Chapter 2 - Gaze

In this chapter, we are going to introduce the first of three ways of interacting with your holograms --[gaze](#).

Objectives

- Visualize your gaze using a world-locked cursor.

Instructions

- Go back to your Unity project, and close the Build Settings window if it's still open.
- Select the **Holograms** folder in the **Project panel**.
- Drag the **Cursor** object into the **Hierarchy panel** at the root level.
- Double-click on the **Cursor** object to take a closer look at it.
- Right-click on the **Scripts** folder in the Project panel.
- Click the **Create** sub-menu.
- Select **C# Script**.
- Name the script **WorldCursor**. Note: The name is case-sensitive. You do not need to add the .cs extension.
- Select the **Cursor** object in the **Hierarchy panel**.
- Drag and drop the **WorldCursor** script into the **Inspector panel**.
- Double-click the **WorldCursor** script to open it in Visual Studio.
- Copy and paste this code into **WorldCursor.cs** and **Save All**.

```

using UnityEngine;

public class WorldCursor : MonoBehaviour
{
    private MeshRenderer meshRenderer;

    // Use this for initialization
    void Start()
    {
        // Grab the mesh renderer that's on the same object as this script.
        meshRenderer = this.gameObject.GetComponentInChildren<MeshRenderer>();
    }

    // Update is called once per frame
    void Update()
    {
        // Do a raycast into the world based on the user's
        // head position and orientation.
        var headPosition = Camera.main.transform.position;
        var gazeDirection = Camera.main.transform.forward;

        RaycastHit hitInfo;

        if (Physics.Raycast(headPosition, gazeDirection, out hitInfo))
        {
            // If the raycast hit a hologram...
            // Display the cursor mesh.
            meshRenderer.enabled = true;

            // Move the cursor to the point where the raycast hit.
            this.transform.position = hitInfo.point;

            // Rotate the cursor to hug the surface of the hologram.
            this.transform.rotation = Quaternion.FromToRotation(Vector3.up, hitInfo.normal);
        }
        else
        {
            // If the raycast did not hit a hologram, hide the cursor mesh.
            meshRenderer.enabled = false;
        }
    }
}

```

- Rebuild the app from **File > Build Settings**.
- Return to the Visual Studio solution previously used to deploy to your HoloLens.
- Select 'Reload All' when prompted.
- Click **Debug -> Start Without debugging** or press **Ctrl + F5**.
- Now look around the scene and notice how the cursor interacts with the shape of objects.

Chapter 3 - Gestures

In this chapter, we'll add support for [gestures](#). When the user selects a paper sphere, we'll make the sphere fall by turning on gravity using Unity's physics engine.

Objectives

- Control your holograms with the Select gesture.

Instructions

We'll start by creating a script then can detect the Select gesture.

- In the **Scripts** folder, create a script named **GazeGestureManager**.
- Drag the **GazeGestureManager** script onto the **OrigamiCollection** object in the Hierarchy.
- Open the **GazeGestureManager** script in Visual Studio and add the following code:

```

using UnityEngine;
using UnityEngine.XR.WSA.Input;

public class GazeGestureManager : MonoBehaviour
{
    public static GazeGestureManager Instance { get; private set; }

    // Represents the hologram that is currently being gazed at.
    public GameObject FocusedObject { get; private set; }

    GestureRecognizer recognizer;

    // Use this for initialization
    void Awake()
    {
        Instance = this;

        // Set up a GestureRecognizer to detect Select gestures.
        recognizer = new GestureRecognizer();
        recognizer.Tapped += (args) =>
        {
            // Send an OnSelect message to the focused object and its ancestors.
            if (FocusedObject != null)
            {
                FocusedObject.SendMessageUpwards("OnSelect", SendMessageOptions.DontRequireReceiver);
            }
        };
        recognizer.StartCapturingGestures();
    }

    // Update is called once per frame
    void Update()
    {
        // Figure out which hologram is focused this frame.
        GameObject oldFocusObject = FocusedObject;

        // Do a raycast into the world based on the user's
        // head position and orientation.
        var headPosition = Camera.main.transform.position;
        var gazeDirection = Camera.main.transform.forward;

        RaycastHit hitInfo;
        if (Physics.Raycast(headPosition, gazeDirection, out hitInfo))
        {
            // If the raycast hit a hologram, use that as the focused object.
            FocusedObject = hitInfo.collider.gameObject;
        }
        else
        {
            // If the raycast did not hit a hologram, clear the focused object.
            FocusedObject = null;
        }

        // If the focused object changed this frame,
        // start detecting fresh gestures again.
        if (FocusedObject != oldFocusObject)
        {
            recognizer.CancelGestures();
            recognizer.StartCapturingGestures();
        }
    }
}

```

- Create another script in the Scripts folder, this time named **SphereCommands**.
- Expand the **OrigamiCollection** object in the Hierarchy view.

- Drag the **SphereCommands** script onto the **Sphere1** object in the Hierarchy panel.
- Drag the **SphereCommands** script onto the **Sphere2** object in the Hierarchy panel.
- Open the script in Visual Studio for editing, and replace the default code with this:

```
using UnityEngine;

public class SphereCommands : MonoBehaviour
{
    // Called by GazeGestureManager when the user performs a Select gesture
    void OnSelect()
    {
        // If the sphere has no Rigidbody component, add one to enable physics.
        if (!this.GetComponent<Rigidbody>())
        {
            var rigidbody = this.gameObject.AddComponent<Rigidbody>();
            rigidbody.collisionDetectionMode = CollisionDetectionMode.Continuous;
        }
    }
}
```

- Export, build and deploy the app to your HoloLens.
- Look at one of the spheres.
- Perform the select gesture and watch the sphere drop onto the surface below.

Chapter 4 - Voice

In this chapter, we'll add support for two [voice commands](#): "Reset world" to return the dropped spheres to their original location, and "Drop sphere" to make the sphere fall.

Objectives

- Add voice commands that always listen in the background.
- Create a hologram that reacts to a voice command.

Instructions

- In the **Scripts** folder, create a script named **SpeechManager**.
- Drag the **SpeechManager** script onto the **OrigamiCollection** object in the Hierarchy
- Open the **SpeechManager** script in Visual Studio.
- Copy and paste this code into **SpeechManager.cs** and **Save All**:

```

using System.Collections.Generic;
using System.Linq;
using UnityEngine;
using UnityEngine.Windows.Speech;

public class SpeechManager : MonoBehaviour
{
    KeywordRecognizer keywordRecognizer = null;
    Dictionary<string, System.Action> keywords = new Dictionary<string, System.Action>();

    // Use this for initialization
    void Start()
    {
        keywords.Add("Reset world", () =>
        {
            // Call the OnReset method on every descendant object.
            this.BroadcastMessage("OnReset");
        });
    }

    keywords.Add("Drop Sphere", () =>
    {
        var focusObject = GazeGestureManager.Instance.FocusedObject;
        if (focusObject != null)
        {
            // Call the OnDrop method on just the focused object.
            focusObject.SendMessage("OnDrop", SendMessageOptions.DontRequireReceiver);
        }
    });

    // Tell the KeywordRecognizer about our keywords.
    keywordRecognizer = new KeywordRecognizer(keywords.Keys.ToArray());

    // Register a callback for the KeywordRecognizer and start recognizing!
    keywordRecognizer.OnPhraseRecognized += KeywordRecognizer_OnPhraseRecognized;
    keywordRecognizer.Start();
}

private void KeywordRecognizer_OnPhraseRecognized(PhraseRecognizedEventArgs args)
{
    System.Action keywordAction;
    if (keywords.TryGetValue(args.text, out keywordAction))
    {
        keywordAction.Invoke();
    }
}
}

```

- Open the **SphereCommands** script in Visual Studio.
- Update the script to read as follows:

```

using UnityEngine;

public class SphereCommands : MonoBehaviour
{
    Vector3 originalPosition;

    // Use this for initialization
    void Start()
    {
        // Grab the original local position of the sphere when the app starts.
        originalPosition = this.transform.localPosition;
    }

    // Called by GazeGestureManager when the user performs a Select gesture
    void OnSelect()
    {
        // If the sphere has no Rigidbody component, add one to enable physics.
        if (!this.GetComponent<Rigidbody>())
        {
            var rigidbody = this.gameObject.AddComponent<Rigidbody>();
            rigidbody.collisionDetectionMode = CollisionDetectionMode.Continuous;
        }
    }

    // Called by SpeechManager when the user says the "Reset world" command
    void OnReset()
    {
        // If the sphere has a Rigidbody component, remove it to disable physics.
        var rigidbody = this.GetComponent<Rigidbody>();
        if (rigidbody != null)
        {
            rigidbody.isKinematic = true;
            Destroy(rigidbody);
        }

        // Put the sphere back into its original local position.
        this.transform.localPosition = originalPosition;
    }

    // Called by SpeechManager when the user says the "Drop sphere" command
    void OnDrop()
    {
        // Just do the same logic as a Select gesture.
        OnSelect();
    }
}

```

- Export, build and deploy the app to your HoloLens.
- Look at one of the spheres, and say "**Drop Sphere**".
- Say "**Reset World**" to bring them back to their initial positions.

Chapter 5 - Spatial sound

In this chapter, we'll add music to the app, and then trigger sound effects on certain actions. We'll be using [spatial sound](#) to give sounds a specific location in 3D space.

Objectives

- Hear holograms in your world.

Instructions

- In Unity select from the top menu **Edit > Project Settings > Audio**

- In the Inspector Panel on the right side, find the **Spatializer Plugin** setting and select **MS HRTF Spatializer**.
- From the **Holograms** folder in the Project panel, drag the **Ambience** object onto the **OrigamiCollection** object in the Hierarchy Panel.
- Select **OrigamiCollection** and find the **Audio Source** component in the Inspector panel. Change these properties:
 - Check the **Spatialize** property.
 - Check the **Play On Awake**.
 - Change **Spatial Blend** to **3D** by dragging the slider all the way to the right. The value should change from 0 to 1 when you move the slider.
 - Check the **Loop** property.
 - Expand **3D Sound Settings**, and enter **0.1** for **Doppler Level**.
 - Set **Volume Rolloff** to **Logarithmic Rolloff**.
 - Set **Max Distance** to **20**.
- In the **Scripts** folder, create a script named **SphereSounds**.
- Drag and drop **SphereSounds** to the **Sphere1** and **Sphere2** objects in the Hierarchy.
- Open **SphereSounds** in Visual Studio, update the following code and **Save All**.

```

using UnityEngine;

public class SphereSounds : MonoBehaviour
{
    AudioSource impact AudioSource = null;
    AudioSource rolling AudioSource = null;

    bool rolling = false;

    void Start()
    {
        // Add an AudioSource component and set up some defaults
        impact AudioSource = gameObject.AddComponent< AudioSource >();
        impact AudioSource.playOnAwake = false;
        impact AudioSource.spatialize = true;
        impact AudioSource.spatialBlend = 1.0f;
        impact AudioSource.dopplerLevel = 0.0f;
        impact AudioSource.rolloffMode = AudioRolloffMode.Logarithmic;
        impact AudioSource.maxDistance = 20f;

        rolling AudioSource = gameObject.AddComponent< AudioSource >();
        rolling AudioSource.playOnAwake = false;
        rolling AudioSource.spatialize = true;
        rolling AudioSource.spatialBlend = 1.0f;
        rolling AudioSource.dopplerLevel = 0.0f;
        rolling AudioSource.rolloffMode = AudioRolloffMode.Logarithmic;
        rolling AudioSource.maxDistance = 20f;
        rolling AudioSource.loop = true;

        // Load the Sphere sounds from the Resources folder
        impact AudioSource.clip = Resources.Load< AudioClip >("Impact");
        rolling AudioSource.clip = Resources.Load< AudioClip >("Rolling");
    }

    // Occurs when this object starts colliding with another object
    void OnCollisionEnter(Collision collision)
    {
        // Play an impact sound if the sphere impacts strongly enough.
        if (collision.relativeVelocity.magnitude >= 0.1f)
        {
            impact AudioSource.Play();
        }
    }
}

```

```

// Occurs each frame that this object continues to collide with another object
void OnCollisionStay(Collision collision)
{
    Rigidbody rigid = gameObject.GetComponent<Rigidbody>();

    // Play a rolling sound if the sphere is rolling fast enough.
    if (!rolling && rigid.velocity.magnitude >= 0.01f)
    {
        rolling = true;
        rolling AudioSource.Play();
    }
    // Stop the rolling sound if rolling slows down.
    else if (rolling && rigid.velocity.magnitude < 0.01f)
    {
        rolling = false;
        rolling AudioSource.Stop();
    }
}

// Occurs when this object stops colliding with another object
void OnCollisionExit(Collision collision)
{
    // Stop the rolling sound if the object falls off and stops colliding.
    if (rolling)
    {
        rolling = false;
        impact AudioSource.Stop();
        rolling AudioSource.Stop();
    }
}
}

```

- Save the script, and return to Unity.
- Export, build and deploy the app to your HoloLens.
- Move closer and further from the Stage and turn side-to-side to hear the sounds change.

Chapter 6 - Spatial mapping

Now we are going to use [spatial mapping](#) to place the game board on a real object in the real world.

Objectives

- Bring your real world into the virtual world.
- Place your holograms where they matter most to you.

Instructions

- In Unity, click on the **Holograms** folder in the Project panel.
- Drag the **Spatial Mapping** asset into the root of the **Hierarchy**.
- Click on the **Spatial Mapping** object in the Hierarchy.
- In the **Inspector panel**, change the following properties:
 - Check the **Draw Visual Meshes** box.
 - Locate **Draw Material** and click the circle on the right. Type "**wireframe**" into the search field at the top. Click on the result and then close the window. When you do this, the value for Draw Material should get set to Wireframe.
- Export, build and deploy the app to your HoloLens.
- When the app runs, a wireframe mesh will overlay your real world.
- Watch how a rolling sphere will fall off the stage, and onto the floor!

Now we'll show you how to move the OrigamiCollection to a new location:

- In the **Scripts** folder, create a script named **TapToPlaceParent**.
- In the **Hierarchy**, expand the **OrigamiCollection** and select the **Stage** object.
- Drag the **TapToPlaceParent** script onto the Stage object.
- Open the **TapToPlaceParent** script in Visual Studio, and update it to be the following:

```

using UnityEngine;

public class TapToPlaceParent : MonoBehaviour
{
    bool placing = false;

    // Called by GazeGestureManager when the user performs a Select gesture
    void OnSelect()
    {
        // On each Select gesture, toggle whether the user is in placing mode.
        placing = !placing;

        // If the user is in placing mode, display the spatial mapping mesh.
        if (placing)
        {
            SpatialMapping.Instance.DrawVisualMeshes = true;
        }
        // If the user is not in placing mode, hide the spatial mapping mesh.
        else
        {
            SpatialMapping.Instance.DrawVisualMeshes = false;
        }
    }

    // Update is called once per frame
    void Update()
    {
        // If the user is in placing mode,
        // update the placement to match the user's gaze.

        if (placing)
        {
            // Do a raycast into the world that will only hit the Spatial Mapping mesh.
            var headPosition = Camera.main.transform.position;
            var gazeDirection = Camera.main.transform.forward;

            RaycastHit hitInfo;
            if (Physics.Raycast(headPosition, gazeDirection, out hitInfo,
                30.0f, SpatialMapping.PhysicsRaycastMask))
            {
                // Move this object's parent object to
                // where the raycast hit the Spatial Mapping mesh.
                this.transform.parent.position = hitInfo.point;

                // Rotate this object's parent object to face the user.
                Quaternion toQuat = Camera.main.transform.localRotation;
                toQuat.x = 0;
                toQuat.z = 0;
                this.transform.parent.rotation = toQuat;
            }
        }
    }
}

```

- Export, build and deploy the app.
- Now you should now be able to place the game in a specific location by gazing at it, using the Select gesture and then moving to a new location, and using the Select gesture again.

Chapter 7 - Holographic fun

Objectives

- Reveal the entrance to a holographic underworld.

Instructions

Now we'll show you how to uncover the holographic underworld:

- From the **Holograms** folder in the Project Panel:
 - Drag **Underworld** into the Hierarchy to be a child of **OrigamiCollection**.
- In the **Scripts** folder, create a script named **HitTarget**.
- In the **Hierarchy**, expand the **OrigamiCollection**.
- Expand the **Stage** object and select the **Target** object (blue fan).
- Drag the **HitTarget** script onto the **Target** object.
- Open the **HitTarget** script in Visual Studio, and update it to be the following:

```
using UnityEngine;

public class HitTarget : MonoBehaviour
{
    // These public fields become settable properties in the Unity editor.
    public GameObject underworld;
    public GameObject objectToHide;

    // Occurs when this object starts colliding with another object
    void OnCollisionEnter(Collision collision)
    {
        // Hide the stage and show the underworld.
        objectToHide.SetActive(false);
        underworld.SetActive(true);

        // Disable Spatial Mapping to let the spheres enter the underworld.
        SpatialMapping.Instance.MappingEnabled = false;
    }
}
```

- In Unity, select the **Target** object.
- Two public properties are now visible on the **Hit Target** component and need to reference objects in our scene:
 - Drag **Underworld** from the **Hierarchy** panel to the **Underworld** property on the **Hit Target** component.
 - Drag **Stage** from the **Hierarchy** panel to the **Object to Hide** property on the **Hit Target** component.
- Export, build and deploy the app.
- Place the Origami Collection on the floor, and then use the Select gesture to make a sphere drop.
- When the sphere hits the target (blue fan), an explosion will occur. The collection will be hidden and a hole to the underworld will appear.

The end

And that's the end of this tutorial!

You learned:

- How to create a holographic app in Unity.
- How to make use of gaze, gesture, voice, sound, and spatial mapping.
- How to build and deploy an app using Visual Studio.

You are now ready to start creating your own holographic experience!

See also

- [MR Basics 101E: Complete project with emulator](#)
- [Gaze](#)
- [Gestures](#)
- [Voice input](#)
- [Spatial sound](#)
- [Spatial mapping](#)

MR Basics 101E: Complete project with emulator

11/6/2018 • 13 minutes to read • [Edit Online](#)

This tutorial will walk you through a complete project, built in Unity, that demonstrates core Windows Mixed Reality features on HoloLens including [gaze](#), [gestures](#), [voice input](#), [spatial sound](#) and [spatial mapping](#). The tutorial will take approximately 1 hour to complete.

Device support

COURSE	HOLOLENS	IMMERSIVE HEADSETS
MR Basics 101E: Complete project with emulator	✓ <input type="checkbox"/>	

Before you start

Prerequisites

- A Windows 10 PC configured with the correct [tools installed](#).

Project files

- Download the [files](#) required by the project. Requires Unity 2017.2 or later.
 - If you still need Unity 5.6 support, please use [this release](#).
 - If you still need Unity 5.5 support, please use [this release](#).
 - If you still need Unity 5.4 support, please use [this release](#).
- Un-archive the files to your desktop or other easy to reach location. Keep the folder name as **Origami**.

NOTE

If you want to look through the source code before downloading, it's [available on GitHub](#).

Chapter 1 - "Holo" world

In this chapter, we'll setup our first Unity project and step through the build and deploy process.

Objectives

- Set up Unity for holographic development.
- Make a hologram.
- See a hologram that you made.

Instructions

- Start Unity.
- Select **Open**.
- Enter location as the **Origami** folder you previously un-archived.
- Select **Origami** and click **Select Folder**.
- Save the new scene: **File / Save Scene As**.

- Name the scene **Origami** and press the **Save** button.

Setup the main camera

- In the **Hierarchy Panel**, select **Main Camera**.
- In the **Inspector** set its transform position to **0,0,0**.
- Find the **Clear Flags** property, and change the dropdown from **Skybox** to **Solid color**.
- Click on the **Background** field to open a color picker.
- Set **R, G, B, and A** to **0**.

Setup the scene

- In the **Hierarchy Panel**, click on **Create** and **Create Empty**.
- Right-click the new **GameObject** and select **Rename**. Rename the GameObject to **OrigamiCollection**.
- From the **Holograms** folder in the **Project Panel**:
 - Drag **Stage** into the Hierarchy to be a child of **OrigamiCollection**.
 - Drag **Sphere1** into the Hierarchy to be a child of **OrigamiCollection**.
 - Drag **Sphere2** into the Hierarchy to be a child of **OrigamiCollection**.
- Right-click the **Directional Light** object in the **Hierarchy Panel** and select **Delete**.
- From the **Holograms** folder, drag **Lights** into the root of the **Hierarchy Panel**.
- In the **Hierarchy**, select the **OrigamiCollection**.
- In the **Inspector**, set the transform position to **0, -0.5, 2.0**.
- Press the **Play** button in Unity to preview your holograms.
- You should see the Origami objects in the preview window.
- Press **Play** a second time to stop preview mode.

Export the project from Unity to Visual Studio

- In Unity select **File > Build Settings**.
- Select **Windows Store** in the **Platform** list and click **Switch Platform**.
- Set **SDK** to **Universal 10** and **Build Type** to **D3D**.
- Check **Unity C# Projects**.
- Click **Add Open Scenes** to add the scene.
- Click **Player Settings...**
- In the Inspector Panel select the **Windows Store logo**. Then select **Publishing Settings**.
- In the **Capabilities** section, select the **Microphone** and **SpatialPerception** capabilities.
- Back in the Build Settings window, click **Build**.
- Create a **New Folder** named "App".
- Single click the **App Folder**.
- Press **Select Folder**.
- When Unity is done, a File Explorer window will appear.
- Open the **App** folder.
- Open the **Origami Visual Studio Solution**.
- Using the top toolbar in Visual Studio, change the target from Debug to **Release** and from ARM to **X86**.
 - Click on the arrow next to the Device button, and select **HoloLens Emulator**.
 - Click **Debug -> Start Without debugging** or press **Ctrl + F5**.
 - After some time the emulator will start with the Origami project. When first launching the [emulator](#), it can take as long as 15 minutes for the emulator to start up. Once it starts, do not close it.

Chapter 2 - Gaze

In this chapter, we are going to introduce the first of three ways of interacting with your holograms --[gaze](#).

Objectives

- Visualize your gaze using a world-locked cursor.

Instructions

- Go back to your Unity project, and close the Build Settings window if it's still open.
- Select the **Holograms** folder in the **Project panel**.
- Drag the **Cursor** object into the **Hierarchy panel** at the root level.
- Double-click on the **Cursor** object to take a closer look at it.
- Right-click on the **Scripts** folder in the Project panel.
- Click the **Create** sub-menu.
- Select **C# Script**.
- Name the script **WorldCursor**. Note: The name is case-sensitive. You do not need to add the .cs extension.
- Select the **Cursor** object in the **Hierarchy panel**.
- Drag and drop the **WorldCursor** script into the **Inspector panel**.
- Double-click the **WorldCursor** script to open it in Visual Studio.
- Copy and paste this code into **WorldCursor.cs** and **Save All**.

```

using UnityEngine;

public class WorldCursor : MonoBehaviour
{
    private MeshRenderer meshRenderer;

    // Use this for initialization
    void Start()
    {
        // Grab the mesh renderer that's on the same object as this script.
        meshRenderer = this.gameObject.GetComponentInChildren<MeshRenderer>();
    }

    // Update is called once per frame
    void Update()
    {
        // Do a raycast into the world based on the user's
        // head position and orientation.
        var headPosition = Camera.main.transform.position;
        var gazeDirection = Camera.main.transform.forward;

        RaycastHit hitInfo;

        if (Physics.Raycast(headPosition, gazeDirection, out hitInfo))
        {
            // If the raycast hit a hologram...
            // Display the cursor mesh.
            meshRenderer.enabled = true;

            // Move the cursor to the point where the raycast hit.
            this.transform.position = hitInfo.point;

            // Rotate the cursor to hug the surface of the hologram.
            this.transform.rotation = Quaternion.FromToRotation(Vector3.up, hitInfo.normal);
        }
        else
        {
            // If the raycast did not hit a hologram, hide the cursor mesh.
            meshRenderer.enabled = false;
        }
    }
}

```

- Rebuild the app from **File > Build Settings**.
- Return to the Visual Studio solution previously used to deploy to the emulator.
- Select 'Reload All' when prompted.
- Click **Debug -> Start Without debugging** or press **Ctrl + F5**.
- Use the Xbox controller to look around the scene. Notice how the cursor interacts with the shape of objects.

Chapter 3 - Gestures

In this chapter, we'll add support for [gestures](#). When the user selects a paper sphere, we'll make the sphere fall by turning on gravity using Unity's physics engine.

Objectives

- Control your holograms with the Select gesture.

Instructions

We'll start by creating a script than can detect the Select gesture.

- In the **Scripts** folder, create a script named **GazeGestureManager**.
- Drag the **GazeGestureManager** script onto the **OrigamiCollection** object in the Hierarchy.
- Open the **GazeGestureManager** script in Visual Studio and add the following code:

```

using UnityEngine;
using UnityEngine.XR.WSA.Input;

public class GazeGestureManager : MonoBehaviour
{
    public static GazeGestureManager Instance { get; private set; }

    // Represents the hologram that is currently being gazed at.
    public GameObject FocusedObject { get; private set; }

    GestureRecognizer recognizer;

    // Use this for initialization
    void Start()
    {
        Instance = this;

        // Set up a GestureRecognizer to detect Select gestures.
        recognizer = new GestureRecognizer();
        recognizer.Tapped += (args) =>
        {
            // Send an OnSelect message to the focused object and its ancestors.
            if (FocusedObject != null)
            {
                FocusedObject.SendMessageUpwards("OnSelect", SendMessageOptions.DontRequireReceiver);
            }
        };
        recognizer.StartCapturingGestures();
    }

    // Update is called once per frame
    void Update()
    {
        // Figure out which hologram is focused this frame.
        GameObject oldFocusObject = FocusedObject;

        // Do a raycast into the world based on the user's
        // head position and orientation.
        var headPosition = Camera.main.transform.position;
        var gazeDirection = Camera.main.transform.forward;

        RaycastHit hitInfo;
        if (Physics.Raycast(headPosition, gazeDirection, out hitInfo))
        {
            // If the raycast hit a hologram, use that as the focused object.
            FocusedObject = hitInfo.collider.gameObject;
        }
        else
        {
            // If the raycast did not hit a hologram, clear the focused object.
            FocusedObject = null;
        }

        // If the focused object changed this frame,
        // start detecting fresh gestures again.
        if (FocusedObject != oldFocusObject)
        {
            recognizer.CancelGestures();
            recognizer.StartCapturingGestures();
        }
    }
}

```

- Create another script in the Scripts folder, this time named **SphereCommands**.
- Expand the **OrigamiCollection** object in the Hierarchy view.

- Drag the **SphereCommands** script onto the **Sphere1** object in the Hierarchy panel.
- Drag the **SphereCommands** script onto the **Sphere2** object in the Hierarchy panel.
- Open the script in Visual Studio for editing, and replace the default code with this:

```
using UnityEngine;

public class SphereCommands : MonoBehaviour
{
    // Called by GazeGestureManager when the user performs a Select gesture
    void OnSelect()
    {
        // If the sphere has no Rigidbody component, add one to enable physics.
        if (!this.GetComponent<Rigidbody>())
        {
            var rigidbody = this.gameObject.AddComponent<Rigidbody>();
            rigidbody.collisionDetectionMode = CollisionDetectionMode.Continuous;
        }
    }
}
```

- Export, build and deploy the app to the HoloLens emulator.
- Look around the scene, and center on one of the spheres.
- Press the **A** button on the Xbox controller or press the Spacebar to simulate the Select gesture.

Chapter 4 - Voice

In this chapter, we'll add support for two [voice commands](#): "Reset world" to returns the dropped spheres to their original location, and "Drop sphere" to make the sphere fall.

Objectives

- Add voice commands that always listen in the background.
- Create a hologram that reacts to a voice command.

Instructions

- In the **Scripts** folder, create a script named **SpeechManager**.
- Drag the **SpeechManager** script onto the **OrigamiCollection** object in the Hierarchy
- Open the **SpeechManager** script in Visual Studio.
- Copy and paste this code into **SpeechManager.cs** and **Save All**:

```

using System.Collections.Generic;
using System.Linq;
using UnityEngine;
using UnityEngine.Windows.Speech;

public class SpeechManager : MonoBehaviour
{
    KeywordRecognizer keywordRecognizer = null;
    Dictionary<string, System.Action> keywords = new Dictionary<string, System.Action>();

    // Use this for initialization
    void Start()
    {
        keywords.Add("Reset world", () =>
        {
            // Call the OnReset method on every descendant object.
            this.BroadcastMessage("OnReset");
        });
    }

    keywords.Add("Drop Sphere", () =>
    {
        var focusObject = GazeGestureManager.Instance.FocusedObject;
        if (focusObject != null)
        {
            // Call the OnDrop method on just the focused object.
            focusObject.SendMessage("OnDrop", SendMessageOptions.DontRequireReceiver);
        }
    });
}

// Tell the KeywordRecognizer about our keywords.
keywordRecognizer = new KeywordRecognizer(keywords.Keys.ToArray());

// Register a callback for the KeywordRecognizer and start recognizing!
keywordRecognizer.OnPhraseRecognized += KeywordRecognizer_OnPhraseRecognized;
keywordRecognizer.Start();
}

private void KeywordRecognizer_OnPhraseRecognized(PhraseRecognizedEventArgs args)
{
    System.Action keywordAction;
    if (keywords.TryGetValue(args.text, out keywordAction))
    {
        keywordAction.Invoke();
    }
}
}

```

- Open the **SphereCommands** script in Visual Studio.
- Update the script to read as follows:

```

using UnityEngine;

public class SphereCommands : MonoBehaviour
{
    Vector3 originalPosition;

    // Use this for initialization
    void Start()
    {
        // Grab the original local position of the sphere when the app starts.
        originalPosition = this.transform.localPosition;
    }

    // Called by GazeGestureManager when the user performs a Select gesture
    void OnSelect()
    {
        // If the sphere has no Rigidbody component, add one to enable physics.
        if (!this.GetComponent<Rigidbody>())
        {
            var rigidbody = this.gameObject.AddComponent<Rigidbody>();
            rigidbody.collisionDetectionMode = CollisionDetectionMode.Continuous;
        }
    }

    // Called by SpeechManager when the user says the "Reset world" command
    void OnReset()
    {
        // If the sphere has a Rigidbody component, remove it to disable physics.
        var rigidbody = this.GetComponent<Rigidbody>();
        if (rigidbody != null)
        {
            rigidbody.isKinematic = true;
            Destroy(rigidbody);
        }

        // Put the sphere back into its original local position.
        this.transform.localPosition = originalPosition;
    }

    // Called by SpeechManager when the user says the "Drop sphere" command
    void OnDrop()
    {
        // Just do the same logic as a Select gesture.
        OnSelect();
    }
}

```

- Export, build and deploy the app to the HoloLens emulator.
- The emulator will support your PC's microphone and respond to your voice: adjust the view so the cursor is on one of the spheres, and say "Drop Sphere".
- Say "**Reset World**" to bring them back to their initial positions.

Chapter 5 - Spatial sound

In this chapter, we'll add music to the app, and then trigger sound effects on certain actions. We'll be using [spatial sound](#) to give sounds a specific location in 3D space.

Objectives

- Hear holograms in your world.

Instructions

- In the Unity select from the top menu **Edit > Project Settings > Audio**
- Find the **Spatializer Plugin** setting and select **MS HRTF Spatializer**.
- From the **Holograms** folder, drag the **Ambience** object onto the **OrigamiCollection** object in the Hierarchy Panel.
- Select **OrigamiCollection** and find the **Audio Source** component. Change these properties:
 - Check the **Spatialize** property.
 - Check the **Play On Awake**.
 - Change **Spatial Blend** to **3D** by dragging the slider all the way to the right.
 - Check the **Loop** property.
 - Expand **3D Sound Settings**, and enter **0.1** for **Doppler Level**.
 - Set **Volume Rolloff** to **Logarithmic Rolloff**.
 - Set **Max Distance** to **20**.
- In the **Scripts** folder, create a script named **SphereSounds**.
- Drag **SphereSounds** to the **Sphere1** and **Sphere2** objects in the Hierarchy.
- Open **SphereSounds** in Visual Studio, update the following code and **Save All**.

```

using UnityEngine;

public class SphereSounds : MonoBehaviour
{
    AudioSource impact AudioSource = null;
    AudioSource rolling AudioSource = null;

    bool rolling = false;

    void Start()
    {
        // Add an AudioSource component and set up some defaults
        impact AudioSource = gameObject.AddComponent< AudioSource >();
        impact AudioSource.playOnAwake = false;
        impact AudioSource.spatialize = true;
        impact AudioSource.spatialBlend = 1.0f;
        impact AudioSource.dopplerLevel = 0.0f;
        impact AudioSource.rolloffMode = AudioRolloffMode.Logarithmic;
        impact AudioSource.maxDistance = 20f;

        rolling AudioSource = gameObject.AddComponent< AudioSource >();
        rolling AudioSource.playOnAwake = false;
        rolling AudioSource.spatialize = true;
        rolling AudioSource.spatialBlend = 1.0f;
        rolling AudioSource.dopplerLevel = 0.0f;
        rolling AudioSource.rolloffMode = AudioRolloffMode.Logarithmic;
        rolling AudioSource.maxDistance = 20f;
        rolling AudioSource.loop = true;

        // Load the Sphere sounds from the Resources folder
        impact AudioSource.clip = Resources.Load< AudioClip >("Impact");
        rolling AudioSource.clip = Resources.Load< AudioClip >("Rolling");
    }

    // Occurs when this object starts colliding with another object
    void OnCollisionEnter(Collision collision)
    {
        // Play an impact sound if the sphere impacts strongly enough.
        if (collision.relativeVelocity.magnitude >= 0.1f)
        {
            impact AudioSource.Play();
        }
    }

    // Occurs each frame that this object continues to collide with another object
}

```

```

void OnCollisionStay(Collision collision)
{
    Rigidbody rigid = gameObject.GetComponent<Rigidbody>();

    // Play a rolling sound if the sphere is rolling fast enough.
    if (!rolling && rigid.velocity.magnitude >= 0.01f)
    {
        rolling = true;
        rollingAudioSource.Play();
    }
    // Stop the rolling sound if rolling slows down.
    else if (rolling && rigid.velocity.magnitude < 0.01f)
    {
        rolling = false;
        rollingAudioSource.Stop();
    }
}

// Occurs when this object stops colliding with another object
void OnCollisionExit(Collision collision)
{
    // Stop the rolling sound if the object falls off and stops colliding.
    if (rolling)
    {
        rolling = false;
        impactAudioSource.Stop();
        rollingAudioSource.Stop();
    }
}
}

```

- Save the script, and return to Unity.
- Export, build and deploy the app to the HoloLens emulator.
- Wear headphones to get the full effect, and move closer and further from the Stage to hear the sounds change.

Chapter 6 - Spatial mapping

Now we are going to use [spatial mapping](#) to place the game board on a real object in the real world.

Objectives

- Bring your real world into the virtual world.
- Place your holograms where they matter most to you.

Instructions

- Click on the **Holograms** folder in the Project panel.
- Drag the **Spatial Mapping** asset into the root of the **Hierarchy**.
- Click on the **Spatial Mapping** object in the Hierarchy.
- In the **Inspector panel**, change the following properties:
 - Check the **Draw Visual Meshes** box.
 - Locate **Draw Material** and click the circle on the right. Type "**wireframe**" into the search field at the top. Click on the result and then close the window.
- Export, build and deploy the app to the HoloLens emulator.
- When the app runs, a mesh of a previously scanned real-world living room will be rendered in wireframe.
- Watch how a rolling sphere will fall off the stage, and onto the floor!

Now we'll show you how to move the OrigamiCollection to a new location:

- In the **Scripts** folder, create a script named **TapToPlaceParent**.

- In the **Hierarchy**, expand the **OrigamiCollection** and select the **Stage** object.
- Drag the **TapToPlaceParent** script onto the Stage object.
- Open the **TapToPlaceParent** script in Visual Studio, and update it to be the following:

```

using UnityEngine;

public class TapToPlaceParent : MonoBehaviour
{
    bool placing = false;

    // Called by GazeGestureManager when the user performs a Select gesture
    void OnSelect()
    {
        // On each Select gesture, toggle whether the user is in placing mode.
        placing = !placing;

        // If the user is in placing mode, display the spatial mapping mesh.
        if (placing)
        {
            SpatialMapping.Instance.DrawVisualMeshes = true;
        }
        // If the user is not in placing mode, hide the spatial mapping mesh.
        else
        {
            SpatialMapping.Instance.DrawVisualMeshes = false;
        }
    }

    // Update is called once per frame
    void Update()
    {
        // If the user is in placing mode,
        // update the placement to match the user's gaze.

        if (placing)
        {
            // Do a raycast into the world that will only hit the Spatial Mapping mesh.
            var headPosition = Camera.main.transform.position;
            var gazeDirection = Camera.main.transform.forward;

            RaycastHit hitInfo;
            if (Physics.Raycast(headPosition, gazeDirection, out hitInfo,
                30.0f, SpatialMapping.PhysicsRaycastMask))
            {
                // Move this object's parent object to
                // where the raycast hit the Spatial Mapping mesh.
                this.transform.parent.position = hitInfo.point;

                // Rotate this object's parent object to face the user.
                Quaternion toQuat = Camera.main.transform.localRotation;
                toQuat.x = 0;
                toQuat.z = 0;
                this.transform.parent.rotation = toQuat;
            }
        }
    }
}

```

- Export, build and deploy the app.
- Now you should now be able to place the game in a specific location by gazing at it, using the Select gesture (**A** or Spacebar) and then moving to a new location, and using the Select gesture again.

The end

And that's the end of this tutorial!

You learned:

- How to create a holographic app in Unity.
- How to make use of gaze, gesture, voice, sounds, and spatial mapping.
- How to build and deploy an app using Visual Studio.

You are now ready to start creating your own holographic apps!

See also

- [MR Basics 101: Complete project with device](#)
- [Gaze](#)
- [Gestures](#)
- [Voice input](#)
- [Spatial sound](#)
- [Spatial mapping](#)

MR Input 210: Gaze

11/6/2018 • 12 minutes to read • [Edit Online](#)

Gaze is the first form of input and reveals the user's intent and awareness. MR Input 210 (aka Project Explorer) is a deep dive into gaze-related concepts for Windows Mixed Reality. We will be adding contextual awareness to our cursor and holograms, taking full advantage of what your app knows about the user's gaze.

We have a friendly astronaut here to help you learn gaze concepts. In [MR Basics 101](#), we had a simple cursor that just followed your gaze. Today we're moving a step beyond the simple cursor:

- We're making the cursor and our holograms gaze-aware: both will change based on where the user is looking - or where the user is *not* looking. This makes them context-aware.
- We will add feedback to our cursor and holograms to give the user more context on what is being targeted. This feedback can be audio and visual.
- We'll show you targeting techniques to help users hit smaller targets.
- We'll show you how to draw the user's attention to your holograms with a directional indicator.
- We'll teach you techniques to take your holograms with the user as she moves around in your world.

IMPORTANT

The videos embedded in each of the chapters below were recorded using an older version of Unity and the Mixed Reality Toolkit. While the step-by-step instructions are accurate and current, you may see scripts and visuals in the corresponding videos that are out-of-date. The videos remain included for posterity and because the concepts covered still apply.

Device support

COURSE	HOLOLENS	IMMERSIVE HEADSETS
MR Input 210: Gaze	✓ <input type="checkbox"/>	✓ <input type="checkbox"/>

Before you start

Prerequisites

- A Windows 10 PC configured with the correct [tools installed](#).
- Some basic C# programming ability.
- You should have completed [MR Basics 101](#).
- A HoloLens device [configured for development](#).

Project files

- Download the [files](#) required by the project. Requires Unity 2017.2 or later.
- Un-archive the files to your desktop or other easy to reach location.

NOTE

If you want to look through the source code before downloading, it's [available on GitHub](#).

Errata and Notes

- In Visual Studio, "Just My Code" needs to be disabled (unchecked) under Tools->Options->Debugging in order to hit breakpoints in your code.

Chapter 1 - Unity Setup

Objectives

- Optimize Unity for HoloLens development.
- Import assets and setup the scene.
- View the astronaut in the HoloLens.

Instructions

1. Start Unity.
2. Select **New Project**.
3. Name the project **ModelExplorer**.
4. Enter location as the **Gaze** folder you previously un-archived.
5. Make sure the project is set to **3D**.
6. Click **Create Project**.

Unity settings for HoloLens

We need to let Unity know that the app we are trying to export should create an [immersive view](#) instead of a 2D view. We do that by adding HoloLens as a virtual reality device.

1. Go to **Edit > Project Settings > Player**.
2. In the **Inspector Panel** for Player Settings, select the **Windows Store** icon.
3. Expand the **XR Settings** group.
4. In the **Rendering** section, check the **Virtual Reality Supported** checkbox to add a new **Virtual Reality SDKs** list.
5. Verify that **Windows Mixed Reality** appears in the list. If not, select the **+** button at the bottom of the list and choose **Windows Holographic**.

Next, we need to set our scripting backend to .NET.

1. Go to **Edit > Project Settings > Player** (you may still have this up from the previous step).
2. In the **Inspector Panel** for Player Settings, select the **Windows Store** icon.
3. In the **Other Settings** Configuration section, make sure that **Scripting Backend** is set to **.NET**

Finally, we'll update our quality settings to achieve a fast performance on HoloLens.

1. Go to **Edit > Project Settings > Quality**.
2. Click on downward pointing arrow in the **Default** row under the Windows Store icon.
3. Select **Fastest** for **Windows Store Apps**.

Import project assets

1. Right click the **Assets** folder in the **Project** panel.
2. Click on **Import Package > Custom Package**.
3. Navigate to the project files you downloaded and click on **ModelExplorer.unitypackage**.
4. Click **Open**.
5. After the package loads, click on the **Import** button.

Setup the scene

1. In the Hierarchy, delete the **Main Camera**.
2. In the **HoloToolkit** folder, open the **Input** folder, then open the **Prefabs** folder.
3. Drag and drop the **MixedRealityCameraParent** prefab from the **Prefabs** folder into the **Hierarchy**.
4. Right-click the **Directional Light** in the Hierarchy and select **Delete**.
5. In the **Holograms** folder, drag and drop the following assets into the root of the **Hierarchy**:
 - **AstroMan**
 - **Lights**
 - **Space AudioSource**
 - **Space Background**
6. Start **Play Mode ▶** to view the astronaut.
7. Click **Play Mode ▶** again to **Stop**.
8. In the **Holograms** folder, find the **Fitbox** asset and drag it to the root of the **Hierarchy**.
9. Select the **Fitbox** in the **Hierarchy** panel.
10. Drag the **AstroMan** collection from the **Hierarchy** to the **Hologram Collection** property of the Fitbox in the **Inspector** panel.

Save the project

1. Save the new scene: **File > Save Scene As**.
2. Click **New Folder** and name the folder **Scenes**.
3. Name the file "**ModelExplorer**" and save it in the **Scenes** folder.

Build the project

1. In Unity, select **File > Build Settings**.
2. Click **Add Open Scenes** to add the scene.
3. Select **Universal Windows Platform** in the **Platform** list and click **Switch Platform**.
4. If you're specifically developing for HoloLens, set **Target device** to **HoloLens**. Otherwise, leave it on **Any device**.
5. Ensure **Build Type** is set to **D3D** and **SDK** is set to **Latest installed** (which should be SDK 16299 or newer).
6. Click **Build**.
7. Create a **New Folder** named "App".
8. Single click the **App** folder.
9. Press **Select Folder**.

When Unity is done, a File Explorer window will appear.

1. Open the **App** folder.
2. Open the **ModelExplorer Visual Studio Solution**.

If deploying to HoloLens:

1. Using the top toolbar in Visual Studio, change the target from Debug to **Release** and from ARM to **x86**.
2. Click on the drop down arrow next to the Local Machine button, and select **Remote Machine**.
3. Enter **your HoloLens device IP address** and set Authentication Mode to **Universal (Unencrypted Protocol)**. Click **Select**. If you do not know your device IP address, look in **Settings > Network & Internet > Advanced Options**.
4. In the top menu bar, click **Debug -> Start Without debugging** or press **Ctrl + F5**. If this is the first time deploying to your device, you will need to [pair it with Visual Studio](#).
5. When the app has deployed, dismiss the **Fitbox** with a **select gesture**.

If deploying to an immersive headset:

1. Using the top toolbar in Visual Studio, change the target from Debug to **Release** and from ARM to **x64**.
2. Make sure the deployment target is set to **Local Machine**.
3. In the top menu bar, click **Debug -> Start Without debugging** or press **Ctrl + F5**.
4. When the app has deployed, dismiss the **Fitbox** by pulling the trigger on a motion controller.

Chapter 2 - Cursor and target feedback

Objectives

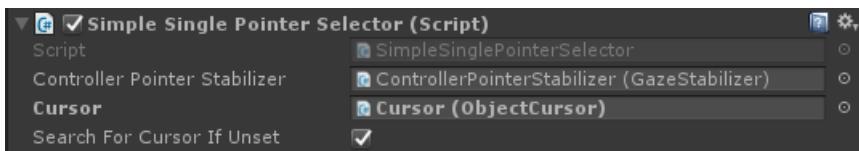
- Cursor visual design and behavior.
- Gaze-based cursor feedback.
- Gaze-based hologram feedback.

We're going to base our work on some cursor design principles, namely:

- The cursor is always present.
- Don't let the cursor get too small or big.
- Avoid obstructing content.

Instructions

1. In the **HoloToolkit\Input\Prefabs** folder, find the **InputManager** asset.
2. Drag and drop the **InputManager** onto the **Hierarchy**.
3. In the **HoloToolkit\Input\Prefabs** folder, find the **Cursor** asset.
4. Drag and drop the **Cursor** onto the **Hierarchy**.
5. Select the **InputManager** object in the **Hierarchy**.
6. Drag the **Cursor** object from the **Hierarchy** into the InputManager's **SimpleSinglePointerSelector**'s **Cursor** field, at the bottom of the **Inspector**.



Build and Deploy

1. Rebuild the app from **File > Build Settings**.
2. Open the **App folder**.
3. Open the **ModelExplorer Visual Studio Solution**.
4. Click **Debug -> Start Without debugging** or press **Ctrl + F5**.
5. Observe how the cursor is drawn, and how it changes appearance if it is touching a hologram.

Instructions

1. In the **Hierarchy** panel, expand the **AstroMan->GEO_G->Back_Center** object.
2. Double click on **Interactable.cs** to open it in Visual Studio.
3. Uncomment the lines in the **IFocusable.OnFocusEnter()** and **IFocusable.OnFocusExit()** callbacks in **Interactable.cs**. These are called by the Mixed Reality Toolkit's InputManager when focus (either by gaze or by controller pointing) enters and exits the specific GameObject's collider.

```

/* TODO: DEVELOPER CODING EXERCISE 2.d */

void IFocusable.OnFocusEnter()
{
    for (int i = 0; i < defaultMaterials.Length; i++)
    {
        // 2.d: Uncomment the below line to highlight the material when gaze enters.
        defaultMaterials[i].EnableKeyword("_ENVIRONMENT_COLORING");
    }
}

void IFocusable.OnFocusExit()
{
    for (int i = 0; i < defaultMaterials.Length; i++)
    {
        // 2.d: Uncomment the below line to remove highlight on material when gaze exits.
        defaultMaterials[i].DisableKeyword("_ENVIRONMENT_COLORING");
    }
}

```

NOTE

We use `EnableKeyword` and `DisableKeyword` above. In order to make use of these in your own app with the Toolkit's Standard shader, you'll need to follow the [Unity guidelines for accessing materials via script](#). In this case, we've already included the [three variants of highlighted material](#) needed in the Resources folder (look for the three materials with highlight in the name).

Build and Deploy

1. As before, build the project and deploy to the HoloLens.
2. Observe what happens when the gaze is aimed at an object and when it's not.

Chapter 3 - Targeting Techniques

Objectives

- Make it easier to target holograms.
- Stabilize natural head movements.

Instructions

1. In the **Hierarchy** panel, select the **InputManager** object.
2. In the **Inspector** panel, find the **Gaze Stabilizer** script. Click it to open in Visual Studio, if you want to take a look.
 - This script iterates over samples of Raycast data and helps stabilize the user's gaze for precision targeting.
3. In the **Inspector**, you can edit the **Stored Stability Samples** value. This value represents the number of samples that the stabilizer iterates on to calculate the stabilized value.

Chapter 4 - Directional indicator

Objectives

- Add a directional indicator on the cursor to help find holograms.

Instructions

We're going to use the **DirectionIndicator.cs** file which will:

1. Show the directional indicator if the user is not gazing at the holograms.
2. Hide the directional indicator if the user is gazing at the holograms.
3. Update the directional indicator to point to the holograms.

Let's get started.

1. Click on the **AstroMan** object in the **Hierarchy** panel and **click the arrow** to expand it.
2. In the **Hierarchy** panel, select the **DirectionalIndicator** object under **AstroMan**.
3. In the **Inspector** panel, click the **Add Component** button.
4. In the menu, type in the search box **Direction Indicator**. Select the search result.
5. In the **Hierarchy** panel, drag and drop the **Cursor** object onto the **Cursor** property in the **Inspector**.
6. In the **Project** panel, in the **Holograms** folder, drag and drop the **DirectionalIndicator** asset onto the **Directional Indicator** property in the **Inspector**.
7. Build and deploy the app.
8. Watch how the directional indicator object helps you find the astronaut.

Chapter 5 - Billboarding

Objectives

- Use billboarding to have holograms always face towards you.

We will be using the **Billboard.cs** file to keep a GameObject oriented such that it is facing the user at all times.

1. In the **Hierarchy** panel, select the **AstroMan** object.
2. In the **Inspector** panel, click the **Add Component** button.
3. In the menu, type in the search box **Billboard**. Select the search result.
4. In the **Inspector** set the **Pivot Axis** to **Y**.
5. Try it! Build and deploy the app as before.
6. See how the Billboard object faces you no matter how you change the viewpoint.
7. Delete the script from the **AstroMan** for now.

Chapter 6 - Tag-Along

Objectives

- Use Tag-Along to have our holograms follow us around the room.

As we work on this issue, we'll be guided by the following design constraints:

- Content should always be a glance away.
- Content should not be in the way.
- Head-locking content is uncomfortable.

The solution used here is to use a "tag-along" approach.

A tag-along object never fully leaves the user's view. You can think of the a tag-along as being an object attached to the user's head by rubber bands. As the user moves, the content will stay within an easy glance by sliding towards the edge of the view without completely leaving. When the user gazes towards the tag-along object, it comes more fully into view.

We're going to use the **SimpleTagalong.cs** file which will:

1. Determine if the Tag-Along object is within the camera bounds.
2. If not within the view frustum, position the Tag-Along to partially within the view frustum.
3. Otherwise, position the Tag-Along to a default distance from the user.

To do this, we first must change the **Interactable.cs** script to call the **TagalongAction**.

1. Edit **Interactable.cs** by completing coding exercise 6.a (uncommenting lines 84 to 87).

```
/* TODO: DEVELOPER CODING EXERCISE 6.a */
// 6.a: Uncomment the lines below to perform a Tagalong action.
if (interactableAction != null)
{
    interactableAction.PerformAction();
}
```

The **InteractableAction.cs** script, paired with **Interactable.cs** performs custom actions when you tap on holograms. In this case, we'll use one specifically for tag-along.

- In the **Scripts** folder click on **TagalongAction.cs** asset to open in Visual Studio.
- Complete the coding exercise or change it to this:
 - At the top of the **Hierarchy**, in the search bar type **ChestButton_Center** and select the result.
 - In the **Inspector** panel, click the **Add Component** button.
 - In the menu, type in the search box **Tagalong Action**. Select the search result.
 - In **Holograms** folder find the **Tagalong** asset.
 - Select the **ChestButton_Center** object in the **Hierarchy**. Drag and drop the **TagAlong** object from the **Project** panel onto the **Inspector** into the **Object To Tagalong** property.
 - Drag the **Tagalong Action** object from the **Inspector** into the **Interactable Action** field on the **Interactable** script.
- Double click the **TagalongAction** script to open it in Visual Studio.



We need to add the following:

- Add functionality to the `PerformAction` function in the `TagalongAction` script (inherited from `InteractableAction`).
- Add billboarding to the gazed-upon object, and set the pivot axis to XY.
- Then add simple Tag-Along to the object.

Here's our solution, from **TagalongAction.cs**:

```

// Copyright (c) Microsoft Corporation. All rights reserved.
// Licensed under the MIT License. See LICENSE in the project root for license information.

using HoloToolkit.Unity;
using UnityEngine;

public class TagalongAction : InteractableAction
{
    [SerializeField]
    [Tooltip("Drag the Tagalong prefab asset you want to display.")]
    private GameObject objectToTagalong;

    private void Awake()
    {
        if (objectToTagalong != null)
        {
            objectToTagalong = Instantiate(objectToTagalong);
            objectToTagalong.SetActive(false);

            /* TODO: DEVELOPER CODING EXERCISE 6.b */

            // 6.b: AddComponent Billboard to objectToTagAlong,
            // so it's always facing the user as they move.
            Billboard billboard = objectToTagalong.AddComponent<Billboard>();

            // 6.b: AddComponent SimpleTagalong to objectToTagAlong,
            // so it's always following the user as they move.
            objectToTagalong.AddComponent<SimpleTagalong>();

            // 6.b: Set any public properties you wish to experiment with.
            billboard.PivotAxis = PivotAxis.XY; // Already the default, but provided in case you want to edit
        }
    }

    public override void PerformAction()
    {
        // Recommend having only one tagalong.
        if (objectToTagalong == null || objectToTagalong.activeSelf)
        {
            return;
        }

        objectToTagalong.SetActive(true);
    }
}

```

- Try it! Build and deploy the app.
- Watch how the content follows the center of the gaze point, but not continuously and without blocking it.

MR Input 211: Gesture

11/6/2018 • 15 minutes to read • [Edit Online](#)

Gestures turn user intention into action. With gestures, users can interact with holograms. In this course, we'll learn how to track the user's hands, respond to user input, and give feedback to the user based on hand state and location.

In [MR Basics 101](#), we used a simple air-tap gesture to interact with our holograms. Now, we'll move beyond the air-tap gesture and explore new concepts to:

- Detect when the user's hand is being tracked and provide feedback to the user.
- Use a navigation gesture to rotate our holograms.
- Provide feedback when the user's hand is about to go out of view.
- Use manipulation events to allow users to move holograms with their hands.

In this course, we'll revisit the Unity project **Model Explorer**, which we built in [MR Input 210](#). Our astronaut friend is back to assist us in our exploration of these new gesture concepts.

IMPORTANT

The videos embedded in each of the chapters below were recorded using an older version of Unity and the Mixed Reality Toolkit. While the step-by-step instructions are accurate and current, you may see scripts and visuals in the corresponding videos that are out-of-date. The videos remain included for posterity and because the concepts covered still apply.

Device support

COURSE	HOOLENS	IMMERSIVE HEADSETS
MR Input 211: Gesture	✓ <input type="checkbox"/>	✓ <input type="checkbox"/>

Before you start

Prerequisites

- A Windows 10 PC configured with the correct [tools installed](#).
- Some basic C# programming ability.
- You should have completed [MR Basics 101](#).
- You should have completed [MR Input 210](#).
- A HoloLens device [configured for development](#).

Project files

- Download the [files](#) required by the project. Requires Unity 2017.2 or later.
- Un-archive the files to your desktop or other easy to reach location.

NOTE

If you want to look through the source code before downloading, it's [available on GitHub](#).

Errata and Notes

- "Enable Just My Code" needs to be disabled (*unchecked*) in Visual Studio under Tools->Options->Debugging in order to hit breakpoints in your code.

Chapter 0 - Unity Setup

Instructions

1. Start Unity.
2. Select **Open**.
3. Navigate to the **Gesture** folder you previously un-archived.
4. Find and select the **Starting/Model Explorer** folder.
5. Click the **Select Folder** button.
6. In the **Project** panel, expand the **Scenes** folder.
7. Double-click **ModelExplorer** scene to load it in Unity.

Building

1. In Unity, select **File > Build Settings**.
2. If **Scenes/ModelExplorer** is not listed in **Scenes In Build**, click **Add Open Scenes** to add the scene.
3. If you're specifically developing for HoloLens, set **Target device** to **HoloLens**. Otherwise, leave it on **Any device**.
4. Ensure **Build Type** is set to **D3D** and **SDK** is set to **Latest installed** (which should be SDK 16299 or newer).
5. Click **Build**.
6. Create a **New Folder** named "App".
7. Single click the **App** folder.
8. Press **Select Folder** and Unity will start building the project for Visual Studio.

When Unity is done, a File Explorer window will appear.

1. Open the **App** folder.
2. Open the **ModelExplorer Visual Studio Solution**.

If deploying to HoloLens:

1. Using the top toolbar in Visual Studio, change the target from Debug to **Release** and from ARM to **x86**.
2. Click on the drop down arrow next to the Local Machine button, and select **Remote Machine**.
3. Enter **your HoloLens device IP address** and set Authentication Mode to **Universal (Unencrypted Protocol)**. Click **Select**. If you do not know your device IP address, look in **Settings > Network & Internet > Advanced Options**.
4. In the top menu bar, click **Debug -> Start Without debugging** or press **Ctrl + F5**. If this is the first time deploying to your device, you will need to [pair it with Visual Studio](#).
5. When the app has deployed, dismiss the **Fitbox** with a **select gesture**.

If deploying to an immersive headset:

1. Using the top toolbar in Visual Studio, change the target from Debug to **Release** and from ARM to **x64**.
2. Make sure the deployment target is set to **Local Machine**.
3. In the top menu bar, click **Debug -> Start Without debugging** or press **Ctrl + F5**.
4. When the app has deployed, dismiss the **Fitbox** by pulling the trigger on a motion controller.

NOTE

You might notice some red errors in the Visual Studio Errors panel. It is safe to ignore them. Switch to the Output panel to view actual build progress. Errors in the Output panel will require you to make a fix (most often they are caused by a mistake in a script).

Chapter 1 - Hand detected feedback

Objectives

- Subscribe to hand tracking events.
- Use cursor feedback to show users when a hand is being tracked.

Instructions

- In the **Hierarchy** panel, expand the **InputManager** object.
- Look for and select the **GesturesInput** object.

The **InteractionInputSource.cs** script performs these steps:

1. Subscribes to the InteractionSourceDetected and InteractionSourceLost events.
2. Sets the HandDetected state.
3. Unsubscribes from the InteractionSourceDetected and InteractionSourceLost events.

Next, we'll upgrade our cursor from [MR Input 210](#) into one that shows feedback depending on the user's actions.

1. In the **Hierarchy** panel, select the **Cursor** object and delete it.
2. In the **Project** panel, search for **CursorWithFeedback** and drag it into the **Hierarchy** panel.
3. Click on **InputManager** in the **Hierarchy** panel, then drag the **CursorWithFeedback** object from the **Hierarchy** into the InputManager's **SimpleSinglePointerSelector**'s **Cursor** field, at the bottom of the **Inspector**.
4. Click on the **CursorWithFeedback** in the **Hierarchy**.
5. In the **Inspector** panel, expand **Cursor State Data** on the **Object Cursor** script.

The **Cursor State Data** works like this:

- Any **Observe** state means that no hand is detected and the user is simply looking around.
- Any **Interact** state means that a hand or controller is detected.
- Any **Hover** state means the user is looking at a hologram.

Build and Deploy

- In Unity, use **File > Build Settings** to rebuild the application.
- Open the **App** folder.
- If it's not already open, open the **ModelExplorer Visual Studio Solution**.
 - (If you already built/deployed this project in Visual Studio during set-up, then you can open that instance of VS and click 'Reload All' when prompted).
- In Visual Studio, click **Debug -> Start Without debugging** or press **Ctrl + F5**.
- After the application deploys to the HoloLens, dismiss the fitbox using the air-tap gesture.
- Move your hand into view and point your index finger to the sky to start hand tracking.
- Move your hand left, right, up and down.
- Watch how the cursor changes when your hand is detected and then lost from view.
- If you're on an immersive headset, you'll have to connect and disconnect your controller. This feedback

becomes less interesting on an immersive device, as a connected controller will always be "available".

Chapter 2 - Navigation

Objectives

- Use Navigation gesture events to rotate the astronaut.

Instructions

To use Navigation gestures in our app, we are going to edit **GestureAction.cs** to rotate objects when the Navigation gesture occurs. Additionally, we'll add feedback to the cursor to display when Navigation is available.

1. In the **Hierarchy** panel, expand **CursorWithFeedback**.
2. In the **Holograms** folder, find the **ScrollFeedback** asset.
3. Drag and drop the **ScrollFeedback** prefab onto the **CursorWithFeedback** GameObject in the **Hierarchy**.
4. Click on **CursorWithFeedback**.
5. In the **Inspector** panel, click the **Add Component** button.
6. In the menu, type in the search box **CursorFeedback**. Select the search result.
7. Drag and drop the **ScrollFeedback** object from the **Hierarchy** onto the **Scroll Detected Game Object** property in the **Cursor Feedback** component in the **Inspector**.
8. In the **Hierarchy** panel, select the **AstroMan** object.
9. In the **Inspector** panel, click the **Add Component** button.
10. In the menu, type in the search box **Gesture Action**. Select the search result.

Next, open **GestureAction.cs** in Visual Studio. In coding exercise 2.c, edit the script to do the following:

1. **Rotate the AstroMan** object whenever a Navigation gesture is performed.
2. Calculate the **rotationFactor** to control the amount of rotation applied to the object.
3. **Rotate the object** around the y-axis when the user moves their hand left or right.

Complete coding exercises 2.c in the script, or replace the code with the completed solution below:

```
using HoloToolkit.Unity.InputModule;
using UnityEngine;

/// <summary>
/// GestureAction performs custom actions based on
/// which gesture is being performed.
/// </summary>
public class GestureAction : MonoBehaviour, INavigationHandler, IManipulationHandler, ISpeechHandler
{
    [Tooltip("Rotation max speed controls amount of rotation.")]
    [SerializeField]
    private float RotationSensitivity = 10.0f;

    private bool isNavigationEnabled = true;
    public bool IsNavigationEnabled
    {
        get { return isNavigationEnabled; }
        set { isNavigationEnabled = value; }
    }

    private Vector3 manipulationOriginalPosition = Vector3.zero;

    void INavigationHandler.OnNavigationStarted(NavigationEventData eventData)
    {
        InputManager.Instance.PushModalInputHandler(gameObject);
    }
}
```

```

void INavigationHandler.OnNavigationUpdated(NavigationEventData eventData)
{
    if (isNavigationEnabled)
    {
        /* TODO: DEVELOPER CODING EXERCISE 2.c */

        // 2.c: Calculate a float rotationFactor based on eventData's NormalizedOffset.x multiplied by
        RotationSensitivity.
        // This will help control the amount of rotation.
        float rotationFactor = eventData.NormalizedOffset.x * RotationSensitivity;

        // 2.c: transform.Rotate around the Y axis using rotationFactor.
        transform.Rotate(new Vector3(0, -1 * rotationFactor, 0));
    }
}

void INavigationHandler.OnNavigationCompleted(NavigationEventData eventData)
{
    InputManager.Instance.PopModalInputHandler();
}

void INavigationHandler.OnNavigationCanceled(NavigationEventData eventData)
{
    InputManager.Instance.PopModalInputHandler();
}

void IManipulationHandler.OnManipulationStarted(ManipulationEventData eventData)
{
    if (!isNavigationEnabled)
    {
        InputManager.Instance.PushModalInputHandler(gameObject);

        manipulationOriginalPosition = transform.position;
    }
}

void IManipulationHandler.OnManipulationUpdated(ManipulationEventData eventData)
{
    if (!isNavigationEnabled)
    {
        /* TODO: DEVELOPER CODING EXERCISE 4.a */

        // 4.a: Make this transform's position be the manipulationOriginalPosition +
        eventData.CumulativeDelta
    }
}

void IManipulationHandler.OnManipulationCompleted(ManipulationEventData eventData)
{
    InputManager.Instance.PopModalInputHandler();
}

void IManipulationHandler.OnManipulationCanceled(ManipulationEventData eventData)
{
    InputManager.Instance.PopModalInputHandler();
}

void ISpeechHandler.OnSpeechKeywordRecognized(SpeechEventData eventData)
{
    if (eventData.RecognizedText.Equals("Move Astronaut"))
    {
        isNavigationEnabled = false;
    }
    else if (eventData.RecognizedText.Equals("Rotate Astronaut"))
    {
        isNavigationEnabled = true;
    }
    else
}

```

```

    {
        return;
    }

    eventData.Use();
}

```

You'll notice that the other navigation events are already filled in with some info. We push the GameObject onto the Toolkit's InputSystem's modal stack, so the user doesn't have to maintain focus on the Astronaut once rotation has begun. Correspondingly, we pop the GameObject off the stack once the gesture is completed.

Build and Deploy

1. Rebuild the application in Unity and then build and deploy from Visual Studio to run it in the HoloLens.
2. Gaze at the astronaut, two arrows should appear on either side of the cursor. This new visual indicates that the astronaut can be rotated.
3. Place your hand in the ready position (index finger pointed towards the sky) so the HoloLens will start tracking your hand.
4. To rotate the astronaut, lower your index finger to a pinch position, and then move your hand left or right to trigger the NavigationX gesture.

Chapter 3 - Hand Guidance

Objectives

- Use **hand guidance score** to help predict when hand tracking will be lost.
- Provide **feedback on the cursor** to show when the user's hand nears the camera's edge of view.

Instructions

1. In the **Hierarchy** panel, select the **CursorWithFeedback** object.
2. In the **Inspector** panel, click the **Add Component** button.
3. In the menu, type in the search box **Hand Guidance**. Select the search result.
4. In the **Project** panel **Holograms** folder, find the **HandGuidanceFeedback** asset.
5. Drag and drop the **HandGuidanceFeedback** asset onto the **Hand Guidance Indicator** property in the **Inspector** panel.

Build and Deploy

- Rebuild the application in Unity and then build and deploy from Visual Studio to experience the app on HoloLens.
- Bring your hand into view and raise your index finger to get tracked.
- Start rotating the astronaut with the Navigation gesture (pinch your index finger and thumb together).
- Move your hand far left, right, up, and down.
- As your hand nears the edge of the gesture frame, an arrow should appear next to the cursor to warn you that hand tracking will be lost. The arrow indicates which direction to move your hand in order to prevent tracking from being lost.

Chapter 4 - Manipulation

Objectives

- Use Manipulation events to move the astronaut with your hands.

- Provide feedback on the cursor to let the user know when Manipulation can be used.

Instructions

`GestureManager.cs` and `AstronautManager.cs` will allow us to do the following:

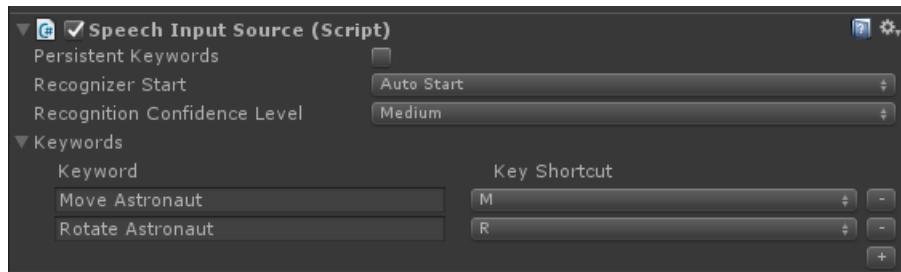
1. Use the speech keyword "**Move Astronaut**" to enable **Manipulation** gestures and "**Rotate Astronaut**" to disable them.
2. Switch to responding to the **Manipulation Gesture Recognizer**.

Let's get started.

1. In the **Hierarchy** panel, create a new empty GameObject. Name it "**AstronautManager**".
2. In the **Inspector** panel, click the **Add Component** button.
3. In the menu, type in the search box **Astronaut Manager**. Select the search result.
4. In the **Inspector** panel, click the **Add Component** button.
5. In the menu, type in the search box **Speech Input Source**. Select the search result.

We'll now add the speech commands required to control the interaction state of the astronaut.

1. Expand the **Keywords** section in the **Inspector**.
2. Click the **+** on the right hand side to add a new keyword.
3. Type the Keyword as **Move Astronaut**. Feel free to add a Key Shortcut if desired.
4. Click the **+** on the right hand side to add a new keyword.
5. Type the Keyword as **Rotate Astronaut**. Feel free to add a Key Shortcut if desired.
6. The corresponding handler code can be found in **GestureAction.cs**, in the **ISpeechHandler.OnSpeechKeywordRecognized** handler.



Next, we'll setup the manipulation feedback on the cursor.

1. In the **Project** panel **Holograms** folder, find the **PathingFeedback** asset.
2. Drag and drop the **PathingFeedback** prefab onto the **CursorWithFeedback** object in the **Hierarchy**.
3. In the **Hierarchy** panel, click on **CursorWithFeedback**.
4. Drag and drop the **PathingFeedback** object from the **Hierarchy** onto the **Pathing Detected Game Object** property in the **Cursor Feedback** component in the **Inspector**.

Now we need to add code to **GestureAction.cs** to enable the following:

1. Add code to the **IManipulationHandler.OnManipulationUpdated** function, which will move the astronaut when a **Manipulation** gesture is detected.
2. Calculate the **movement vector** to determine where the astronaut should be moved to based on hand position.
3. **Move** the astronaut to the new position.

Complete coding exercise 4.a in **GestureAction.cs**, or use our completed solution below:

```
using HoloToolkit.UnityInputModule;
using UnityEngine;
```

```

/// <summary>
/// GestureAction performs custom actions based on
/// which gesture is being performed.
/// </summary>
public class GestureAction : MonoBehaviour, INavigationHandler, IManipulationHandler, ISpeechHandler
{
    [Tooltip("Rotation max speed controls amount of rotation.")]
    [SerializeField]
    private float RotationSensitivity = 10.0f;

    private bool isNavigationEnabled = true;
    public bool IsNavigationEnabled
    {
        get { return isNavigationEnabled; }
        set { isNavigationEnabled = value; }
    }

    private Vector3 manipulationOriginalPosition = Vector3.zero;

    void INavigationHandler.OnNavigationStarted(NavigationEventData eventData)
    {
        InputManager.Instance.PushModalInputHandler(gameObject);
    }

    void INavigationHandler.OnNavigationUpdated(NavigationEventData eventData)
    {
        if (isNavigationEnabled)
        {
            /* TODO: DEVELOPER CODING EXERCISE 2.c */

            // 2.c: Calculate a float rotationFactor based on eventData's NormalizedOffset.x multiplied by
            RotationSensitivity.
            // This will help control the amount of rotation.
            float rotationFactor = eventData.NormalizedOffset.x * RotationSensitivity;

            // 2.c: transform.Rotate around the Y axis using rotationFactor.
            transform.Rotate(new Vector3(0, -1 * rotationFactor, 0));
        }
    }

    void INavigationHandler.OnNavigationCompleted(NavigationEventData eventData)
    {
        InputManager.Instance.PopModalInputHandler();
    }

    void INavigationHandler.OnNavigationCanceled(NavigationEventData eventData)
    {
        InputManager.Instance.PopModalInputHandler();
    }

    void IManipulationHandler.OnManipulationStarted(ManipulationEventData eventData)
    {
        if (!isNavigationEnabled)
        {
            InputManager.Instance.PushModalInputHandler(gameObject);

            manipulationOriginalPosition = transform.position;
        }
    }

    void IManipulationHandler.OnManipulationUpdated(ManipulationEventData eventData)
    {
        if (!isNavigationEnabled)
        {
            /* TODO: DEVELOPER CODING EXERCISE 4.a */

            // 4.a: Make this transform's position be the manipulationOriginalPosition +
            eventData.CumulativeDelta
        }
    }
}

```

```

        transform.position = manipulationOriginalPosition + eventData.CumulativeDelta;
    }

    void IManipulationHandler.OnManipulationCompleted(ManipulationEventData eventData)
    {
        InputManager.Instance.PopModalInputHandler();
    }

    void IManipulationHandler.OnManipulationCanceled(ManipulationEventData eventData)
    {
        InputManager.Instance.PopModalInputHandler();
    }

    void ISpeechHandler.OnSpeechKeywordRecognized(SpeechEventData eventData)
    {
        if (eventData.RecognizedText.Equals("Move Astronaut"))
        {
            isNavigationEnabled = false;
        }
        else if (eventData.RecognizedText.Equals("Rotate Astronaut"))
        {
            isNavigationEnabled = true;
        }
        else
        {
            return;
        }

        eventData.Use();
    }
}

```

Build and Deploy

- Rebuild in Unity and then build and deploy from Visual Studio to run the app in HoloLens.
- Move your hand in front of the HoloLens and raise your index finger so that it can be tracked.
- Focus the cursor over the astronaut.
- Say 'Move Astronaut' to move the astronaut with a Manipulation gesture.
- Four arrows should appear around the cursor to indicate that the program will now respond to Manipulation events.
- Lower your index finger down to your thumb, and keep them pinched together.
- As you move your hand around, the astronaut will move too (this is Manipulation).
- Raise your index finger to stop manipulating the astronaut.
- Note: If you do not say 'Move Astronaut' before moving your hand, then the Navigation gesture will be used instead.
- Say 'Rotate Astronaut' to return to the rotatable state.

Chapter 5 - Model expansion

Objectives

- Expand the Astronaut model into multiple, smaller pieces that the user can interact with.
- Move each piece individually using Navigation and Manipulation gestures.

Instructions

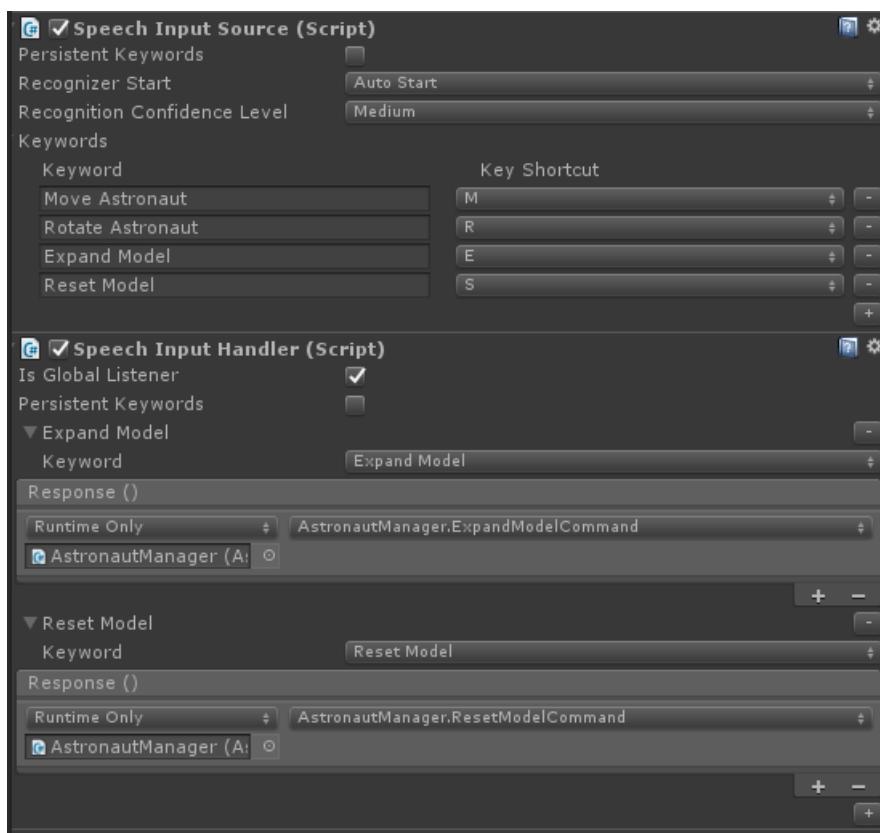
In this section, we will accomplish the following tasks:

1. Add a new keyword "**Expand Model**" to expand the astronaut model.

2. Add a new Keyword "Reset Model" to return the model to its original form.

We'll do this by adding two more keywords to the Speech Input Source from the previous chapter. We'll also demonstrate another way to handle recognition events.

1. Click back on **AstronautManager** in the **Inspector** and expand the **Keywords** section in the **Inspector**.
2. Click the **+** on the right hand side to add a new keyword.
3. Type the Keyword as **Expand Model**. Feel free to add a Key Shortcut if desired.
4. Click the **+** on the right hand side to add a new keyword.
5. Type the Keyword as **Reset Model**. Feel free to add a Key Shortcut if desired.
6. In the **Inspector** panel, click the **Add Component** button.
7. In the menu, type in the search box **Speech Input Handler**. Select the search result.
8. Check **Is Global Listener**, since we want these commands to work regardless of the GameObject we're focusing.
9. Click the **+** button and select **Expand Model** from the Keyword dropdown.
10. Click the **+** under Response, and drag the **AstronautManager** from the **Hierarchy** into the **None (Object)** field.
11. Now, click the **No Function** dropdown, select **AstronautManager**, then **ExpandModelCommand**.
12. Click the Speech Input Handler's **+** button and select **Reset Model** from the Keyword dropdown.
13. Click the **+** under Response, and drag the **AstronautManager** from the **Hierarchy** into the **None (Object)** field.
14. Now, click the **No Function** dropdown, select **AstronautManager**, then **ResetModelCommand**.



Build and Deploy

- Try it! Build and deploy the app to the HoloLens.
- Say **Expand Model** to see the expanded astronaut model.
- Use **Navigation** to rotate individual pieces of the astronaut suit.
- Say **Move Astronaut** and then use **Manipulation** to move individual pieces of the astronaut suit.
- Say **Rotate Astronaut** to rotate the pieces again.

- Say **Reset Model** to return the astronaut to its original form.

The End

Congratulations! You have now completed **MR Input 211: Gesture**.

- You know how to detect and respond to hand tracking, navigation and manipulation events.
- You understand the difference between Navigation and Manipulation gestures.
- You know how to change the cursor to provide visual feedback for when a hand is detected, when a hand is about to be lost, and for when an object supports different interactions (Navigation vs Manipulation).

MR Input 212: Voice

11/6/2018 • 18 minutes to read • [Edit Online](#)

Voice input gives us another way to interact with our holograms. Voice commands work in a very natural and easy way. Design your voice commands so that they are:

- Natural
- Easy to remember
- Context appropriate
- Sufficiently distinct from other options within the same context

In [MR Basics 101](#), we used the KeywordRecognizer to build two simple voice commands. In MR Input 212, we'll dive deeper and learn how to:

- Design voice commands that are optimized for the HoloLens speech engine.
- Make the user aware of what voice commands are available.
- Acknowledge that we've heard the user's voice command.
- Understand what the user is saying, using a Dictation Recognizer.
- Use a Grammar Recognizer to listen for commands based on an SRGS, or Speech Recognition Grammar Specification, file.

In this course, we'll revisit Model Explorer, which we built in [MR Input 210](#) and [MR Input 211](#).

IMPORTANT

The videos embedded in each of the chapters below were recorded using an older version of Unity and the Mixed Reality Toolkit. While the step-by-step instructions are accurate and current, you may see scripts and visuals in the corresponding videos that are out-of-date. The videos remain included for posterity and because the concepts covered still apply.

Device support

COURSE	HOLOLENS	IMMERSIVE HEADSETS
MR Input 212: Voice	✓ <input type="checkbox"/>	✓ <input type="checkbox"/>

Before you start

Prerequisites

- A Windows 10 PC configured with the correct [tools installed](#).
- Some basic C# programming ability.
- You should have completed [MR Basics 101](#).
- You should have completed [MR Input 210](#).
- You should have completed [MR Input 211](#).
- A HoloLens device [configured for development](#).

Project files

- Download the [files](#) required by the project. Requires Unity 2017.2 or later.
- Un-archive the files to your desktop or other easy to reach location.

NOTE

If you want to look through the source code before downloading, it's [available on GitHub](#).

Errata and Notes

- "Enable Just My Code" needs to be disabled (*unchecked*) in Visual Studio under Tools->Options->Debugging in order to hit breakpoints in your code.

Unity Setup

Instructions

1. Start Unity.
2. Select **Open**.
3. Navigate to the **HolographicAcademy-Holograms-212-Voice** folder you previously un-archived.
4. Find and select the **Starting/Model Explorer** folder.
5. Click the **Select Folder** button.
6. In the **Project** panel, expand the **Scenes** folder.
7. Double-click **ModelExplorer** scene to load it in Unity.

Building

1. In Unity, select **File > Build Settings**.
2. If **Scenes/ModelExplorer** is not listed in **Scenes In Build**, click **Add Open Scenes** to add the scene.
3. If you're specifically developing for HoloLens, set **Target device** to **HoloLens**. Otherwise, leave it on **Any device**.
4. Ensure **Build Type** is set to **D3D** and **SDK** is set to **Latest installed** (which should be SDK 16299 or newer).
5. Click **Build**.
6. Create a **New Folder** named "App".
7. Single click the **App** folder.
8. Press **Select Folder** and Unity will start building the project for Visual Studio.

When Unity is done, a File Explorer window will appear.

1. Open the **App** folder.
2. Open the **ModelExplorer Visual Studio Solution**.

If deploying to HoloLens:

1. Using the top toolbar in Visual Studio, change the target from Debug to **Release** and from ARM to **x86**.
2. Click on the drop down arrow next to the Local Machine button, and select **Remote Machine**.
3. Enter **your HoloLens device IP address** and set Authentication Mode to **Universal (Unencrypted Protocol)**. Click **Select**. If you do not know your device IP address, look in **Settings > Network & Internet > Advanced Options**.
4. In the top menu bar, click **Debug -> Start Without debugging** or press **Ctrl + F5**. If this is the first time deploying to your device, you will need to [pair it with Visual Studio](#).
5. When the app has deployed, dismiss the **Fitbox** with a **select gesture**.

If deploying to an immersive headset:

1. Using the top toolbar in Visual Studio, change the target from Debug to **Release** and from ARM to **x64**.

2. Make sure the deployment target is set to **Local Machine**.
3. In the top menu bar, click **Debug -> Start Without debugging** or press **Ctrl + F5**.
4. When the app has deployed, dismiss the **Fitbox** by pulling the trigger on a motion controller.

NOTE

You might notice some red errors in the Visual Studio Errors panel. It is safe to ignore them. Switch to the Output panel to view actual build progress. Errors in the Output panel will require you to make a fix (most often they are caused by a mistake in a script).

Chapter 1 - Awareness

Objectives

- Learn the **Dos and Don'ts** of voice command design.
- Use **KeywordRecognizer** to add gaze based voice commands.
- Make users aware of voice commands using cursor **feedback**.

Voice Command Design

In this chapter, you'll learn about designing voice commands. When creating voice commands:

DO

- Create concise commands. You don't want to use "*Play the currently selected video*", because that command is not concise and would easily be forgotten by the user. Instead, you should use: "*Play Video*", because it is concise and has multiple syllables.
- Use a simple vocabulary. Always try to use common words and phrases that are easy for the user to discover and remember. For example, if your application had a note object that could be displayed or hidden from view, you would not use the command "*Show Placard*", because "placard" is a rarely used term. Instead, you would use the command: "*Show Note*", to reveal the note in your application.
- Be consistent. Voice commands should be kept consistent across your application. Imagine that you have two scenes in your application and both scenes contain a button for closing the application. If the first scene used the command "*Exit*" to trigger the button, but the second scene used the command "*Close App*", then the user is going to get very confused. If the same functionality persists across multiple scenes, then the same voice command should be used to trigger it.

DON'T

- Use single syllable commands. As an example, if you were creating a voice command to play a video, you should avoid using the simple command "*Play*", as it is only a single syllable and could easily be missed by the system. Instead, you should use: "*Play Video*", because it is concise and has multiple syllables.
- Use system commands. The "*Select*" command is reserved by the system to trigger a Tap event for the currently focused object. Do not re-use the "*Select*" command in a keyword or phrase, as it might not work as you expect. For example, if the voice command for selecting a cube in your application was "*Select cube*", but the user was looking at a sphere when they uttered the command, then the sphere would be selected instead. Similarly app bar commands are voice enabled. Don't use the following speech commands in your CoreWindow View:

1. Go Back
2. Scroll Tool
3. Zoom Tool
4. Drag Tool
5. Adjust
6. Remove

- Use similar sounds. Try to avoid using voice commands that rhyme. If you had a shopping application which supported "Show Store" and "Show More" as voice commands, then you would want to disable one of the commands while the other was in use. For example, you could use the "Show Store" button to open the store, and then disable that command when the store was displayed so that the "Show More" command could be used for browsing.

Instructions

- In Unity's **Hierarchy** panel, use the search tool to find the **holoComm_screen_mesh** object.
- Double-click on the **holoComm_screen_mesh** object to view it in the **Scene**. This is the astronaut's watch, which will respond to our voice commands.
- In the **Inspector** panel, locate the **Speech Input Source (Script)** component.
- Expand the **Keywords** section to see the supported voice command: **Open Communicator**.
- Click the cog to the right side, then select **Edit Script**.
- Explore **SpeechInputSource.cs** to understand how it uses the **KeywordRecognizer** to add voice commands.

Build and Deploy

- In Unity, use **File > Build Settings** to rebuild the application.
- Open the **App** folder.
- Open the **ModelExplorer Visual Studio Solution**.

(If you already built/deployed this project in Visual Studio during set-up, then you can open that instance of VS and click 'Reload All' when prompted).

- In Visual Studio, click **Debug -> Start Without debugging** or press **Ctrl + F5**.
- After the application deploys to the HoloLens, dismiss the fit box using the **air-tap** gesture.
- Gaze at the astronaut's watch.
- When the watch has focus, verify that the cursor changes to a microphone. This provides feedback that the application is listening for voice commands.
- Verify that a tooltip appears on the watch. This helps users discover the "*Open Communicator*" command.
- While gazing at the watch, say "*Open Communicator*" to open the communicator panel.

Chapter 2 - Acknowledgement

Objectives

- Record a message using the Microphone input.
- Give feedback to the user that the application is listening to their voice.

NOTE

The **Microphone** capability must be declared for an app to record from the microphone. This is done for you already in MR Input 212, but keep this in mind for your own projects.

1. In the Unity Editor, go to the player settings by navigating to "Edit > Project Settings > Player"
2. Click on the "Universal Windows Platform" tab
3. In the "Publishing Settings > Capabilities" section, check the **Microphone** capability

Instructions

- In Unity's **Hierarchy** panel, verify that the **holoComm_screen_mesh** object is selected.
- In the **Inspector** panel, find the **Astronaut Watch (Script)** component.
- Click on the small, blue cube which is set as the value of the **Communicator Prefab** property.

- In the **Project** panel, the **Communicator** prefab should now have focus.
- Click on the **Communicator** prefab in the **Project** panel to view its components in the **Inspector**.
- Look at the **Microphone Manager (Script)** component, this will allow us to record the user's voice.
- Notice that the **Communicator** object has a **Speech Input Handler (Script)** component for responding to the **Send Message** command.
- Look at the **Communicator (Script)** component and double-click on the script to open it in Visual Studio.

`Communicator.cs` is responsible for setting the proper button states on the communicator device. This will allow our users to record a message, play it back, and send the message to the astronaut. It will also start and stop an animated wave form, to acknowledge to the user that their voice was heard.

- In **Communicator.cs**, delete the following lines (81 and 82) from the **Start** method. This will enable the 'Record' button on the communicator.

```
// TODO: 2.a Delete the following two lines:  
RecordButton.SetActive(false);  
MessageUIRenderer.gameObject.SetActive(false);
```

Build and Deploy

- In Visual Studio, rebuild your application and deploy to the device.
- Gaze at the astronaut's watch and say "*Open Communicator*" to show the communicator.
- Press the **Record** button (microphone) to start recording a verbal message for the astronaut.
- Start speaking, and verify that the wave animation plays on the communicator, which provides feedback to the user that their voice is heard.
- Press the **Stop** button (left square), and verify that the wave animation stops running.
- Press the **Play** button (right triangle) to play back the recorded message and hear it on the device.
- Press the **Stop** button (right square) to stop playback of the recorded message.
- Say "*Send Message*" to close the communicator and receive a 'Message Received' response from the astronaut.

Chapter 3 - Understanding and the Dictation Recognizer

Objectives

- Use the Dictation Recognizer to convert the user's speech to text.
- Show the Dictation Recognizer's hypothesized and final results in the communicator.

In this chapter, we'll use the Dictation Recognizer to create a message for the astronaut. When using the Dictation Recognizer, keep in mind that:

- You must be connected to WiFi for the Dictation Recognizer to work.
- Timeouts occur after a set period of time. There are two timeouts to be aware of:
 - If the recognizer starts and doesn't hear any audio for the first five seconds, it will timeout.
 - If the recognizer has given a result but then hears silence for twenty seconds, it will timeout.
- Only one type of recognizer (Keyword or Dictation) can run at a time.

NOTE

The **Microphone** capability must be declared for an app to record from the microphone. This is done for you already in MR Input 212, but keep this in mind for your own projects.

1. In the Unity Editor, go to the player settings by navigating to "Edit > Project Settings > Player"
2. Click on the "Universal Windows Platform" tab
3. In the "Publishing Settings > Capabilities" section, check the **Microphone** capability

Instructions

We're going to edit **MicrophoneManager.cs** to use the Dictation Recognizer. This is what we'll add:

1. When the **Record button** is pressed, we'll **start the DictationRecognizer**.
2. Show the **hypothesis** of what the DictationRecognizer understood.
3. Lock in the **results** of what the DictationRecognizer understood.
4. Check for timeouts from the DictationRecognizer.
5. When the **Stop button** is pressed, or the mic session times out, **stop the DictationRecognizer**.
6. Restart the **KeywordRecognizer**, which will listen for the **Send Message** command.

Let's get started. Complete all coding exercises for 3.a in **MicrophoneManager.cs**, or copy and paste the finished code found below:

```
// Copyright (c) Microsoft Corporation. All rights reserved.  
// Licensed under the MIT License. See LICENSE in the project root for license information.  
  
using System.Collections;  
using System.Text;  
using UnityEngine;  
using UnityEngine.UI;  
using UnityEngine.Windows.Speech;  
  
namespace Academy  
{  
    public class MicrophoneManager : MonoBehaviour  
    {  
        [Tooltip("A text area for the recognizer to display the recognized strings.")]  
        [SerializeField]  
        private Text dictationDisplay;  
  
        private DictationRecognizer dictationRecognizer;  
  
        // Use this string to cache the text currently displayed in the text box.  
        private StringBuilder textSoFar;  
  
        // Using an empty string specifies the default microphone.  
        private static string deviceName = string.Empty;  
        private int samplingRate;  
        private const int messageLength = 10;  
  
        // Use this to reset the UI once the Microphone is done recording after it was started.  
        private bool hasRecordingStarted;  
  
        void Awake()  
        {  
            /* TODO: DEVELOPER CODING EXERCISE 3.a */  
  
            // 3.a: Create a new DictationRecognizer and assign it to dictationRecognizer variable.  
            dictationRecognizer = new DictationRecognizer();  
  
            // 3.a: Register for dictationRecognizer.DictationHypothesis and implement DictationHypothesis  
            below  
            // This event is fired while the user is talking. As the recognizer listens, it provides text of
```

```

// This event is fired while the user is talking. As the recognizer listens, it provides text of
what it's heard so far.
    dictationRecognizer.DictationHypothesis += DictationRecognizer_DictationHypothesis;

    // 3.a: Register for dictationRecognizer.DictationResult and implement DictationResult below
    // This event is fired after the user pauses, typically at the end of a sentence. The full
recognized string is returned here.
    dictationRecognizer.DictationResult += DictationRecognizer_DictationResult;

    // 3.a: Register for dictationRecognizer.DictationComplete and implement DictationComplete below
    // This event is fired when the recognizer stops, whether from Stop() being called, a timeout
occurring, or some other error.
    dictationRecognizer.DictationComplete += DictationRecognizer_DictationComplete;

    // 3.a: Register for dictationRecognizer.DictationError and implement DictationError below
    // This event is fired when an error occurs.
    dictationRecognizer.DictationError += DictationRecognizer_DictationError;

    // Query the maximum frequency of the default microphone. Use 'unused' to ignore the minimum
frequency.
    int unused;
    Microphone.GetDeviceCaps(deviceName, out unused, out samplingRate);

    // Use this string to cache the text currently displayed in the text box.
    textSoFar = new StringBuilder();

    // Use this to reset the UI once the Microphone is done recording after it was started.
    hasRecordingStarted = false;
}

void Update()
{
    // 3.a: Add condition to check if dictationRecognizer.Status is Running
    if (hasRecordingStarted && !Microphone.IsRecording(deviceName) && dictationRecognizer.Status ==
SpeechSystemStatus.Running)
    {
        // Reset the flag now that we're cleaning up the UI.
        hasRecordingStarted = false;

        // This acts like pressing the Stop button and sends the message to the Communicator.
        // If the microphone stops as a result of timing out, make sure to manually stop the dictation
recognizer.
        // Look at the StopRecording function.
        SendMessage("RecordStop");
    }
}

/// <summary>
/// Turns on the dictation recognizer and begins recording audio from the default microphone.
/// </summary>
/// <returns>The audio clip recorded from the microphone.</returns>
public AudioClip StartRecording()
{
    // 3.a Shutdown the PhraseRecognitionSystem. This controls the KeywordRecognizers
    PhraseRecognitionSystemShutdown();

    // 3.a: Start dictationRecognizer
    dictationRecognizer.Start();

    // 3.a Uncomment this line
    dictationDisplay.text = "Dictation is starting. It may take time to display your text the first
time, but begin speaking now...";

    // Set the flag that we've started recording.
    hasRecordingStarted = true;

    // Start recording from the microphone for 10 seconds.
    return Microphone.Start(deviceName, false, messageLength, samplingRate);
}

```

```

/// <summary>
/// Ends the recording session.
/// </summary>
public void StopRecording()
{
    // 3.a: Check if dictationRecognizer.Status is Running and stop it if so
    if (dictationRecognizer.Status == SpeechSystemStatus.Running)
    {
        dictationRecognizer.Stop();
    }

    Microphone.End(deviceName);
}

/// <summary>
/// This event is fired while the user is talking. As the recognizer listens, it provides text of what
it's heard so far.
/// </summary>
/// <param name="text">The currently hypothesized recognition.</param>
private void DictationRecognizer_DictationHypothesis(string text)
{
    // 3.a: Set DictationDisplay text to be textSoFar and new hypothesized text
    // We don't want to append to textSoFar yet, because the hypothesis may have changed on the next
event
    dictationDisplay.text = textSoFar.ToString() + " " + text + "...";
}

/// <summary>
/// This event is fired after the user pauses, typically at the end of a sentence. The full recognized
string is returned here.
/// </summary>
/// <param name="text">The text that was heard by the recognizer.</param>
/// <param name="confidence">A representation of how confident (rejected, low, medium, high) the
recognizer is of this recognition.</param>
private void DictationRecognizer_DictationResult(string text, ConfidenceLevel confidence)
{
    // 3.a: Append textSoFar with latest text
    textSoFar.Append(text + ". ");

    // 3.a: Set DictationDisplay text to be textSoFar
    dictationDisplay.text = textSoFar.ToString();
}

/// <summary>
/// This event is fired when the recognizer stops, whether from Stop() being called, a timeout
occurring, or some other error.
/// Typically, this will simply return "Complete". In this case, we check to see if the recognizer
timed out.
/// </summary>
/// <param name="cause">An enumerated reason for the session completing.</param>
private void DictationRecognizer_DictationComplete(DictationCompletionCause cause)
{
    // If Timeout occurs, the user has been silent for too long.
    // With dictation, the default timeout after a recognition is 20 seconds.
    // The default timeout with initial silence is 5 seconds.
    if (cause == DictationCompletionCause.TimeoutExceeded)
    {
        Microphone.End(deviceName);

        dictationDisplay.text = "Dictation has timed out. Please press the record button again.";
        SendMessage("ResetAfterTimeout");
    }
}

/// <summary>
/// This event is fired when an error occurs.
/// </summary>
/// <param name="error">The string representation of the error reason.</param>

```

```

/// <param name="HRESULT">The int representation of the HRESULT.</param>
private void DictationRecognizer_DictationError(string error, int HRESULT)
{
    // 3.a: Set DictationDisplay text to be the error string
    dictationDisplay.text = error + "\nHRESULT: " + HRESULT;
}

/// <summary>
/// The dictation recognizer may not turn off immediately, so this call blocks on
/// the recognizer reporting that it has actually stopped.
/// </summary>
public IEnumerator WaitForDictationToStop()
{
    while (dictationRecognizer != null && dictationRecognizer.Status == SpeechSystemStatus.Running)
    {
        yield return null;
    }
}
}

```

Build and Deploy

- Rebuild in Visual Studio and deploy to your device.
- Dismiss the fit box with an air-tap gesture.
- Gaze at the astronaut's watch and say "*Open Communicator*".
- Select the **Record** button (microphone) to record your message.
- Start speaking. The **Dictation Recognizer** will interpret your speech and show the hypothesized text in the communicator.
- Try saying "*Send Message*" while you are recording a message. Notice that the **Keyword Recognizer** does not respond because the **Dictation Recognizer** is still active.
- Stop speaking for a few seconds. Watch as the Dictation Recognizer completes its hypothesis and shows the final result.
- Begin speaking and then pause for 20 seconds. This will cause the **Dictation Recognizer** to timeout.
- Notice that the **Keyword Recognizer** is re-enabled after the above timeout. The communicator will now respond to voice commands.
- Say "*Send Message*" to send the message to the astronaut.

Chapter 4 - Grammar Recognizer

Objectives

- Use the Grammar Recognizer to recognize the user's speech according to an SRGS, or Speech Recognition Grammar Specification, file.

NOTE

The **Microphone** capability must be declared for an app to record from the microphone. This is done for you already in MR Input 212, but keep this in mind for your own projects.

1. In the Unity Editor, go to the player settings by navigating to "Edit > Project Settings > Player"
2. Click on the "Universal Windows Platform" tab
3. In the "Publishing Settings > Capabilities" section, check the **Microphone** capability

Instructions

1. In the **Hierarchy** panel, search for **Jetpack_Center** and select it.

2. Look for the **Tagalong Action** script in the **Inspector** panel.
3. Click the little circle to the right of the **Object To Tag Along** field.
4. In the window that pops up, search for **SRGSToolbox** and select it from the list.
5. Take a look at the **SRGSColor.xml** file in the **StreamingAssets** folder.
 - The SRGS design spec can be found on the W3C website [here](#).
 - In our SRGS file, we have three types of rules:
 - A rule which lets you say one color from a list of twelve colors.
 - Three rules which listen for a combination of the color rule and one of the three shapes.
 - The root rule, `colorChooser`, which listens for any combination of the three "color + shape" rules. The shapes can be said in any order and in any amount from just one to all three. This is the only rule that is listened for, as it's specified as the root rule at the top of the file in the initial `<grammar>` tag.

Build and Deploy

- Rebuild the application in Unity, then build and deploy from Visual Studio to experience the app on HoloLens.
- Dismiss the fit box with an air-tap gesture.
- Gaze at the astronaut's jetpack and perform an air-tap gesture.
- Start speaking. The **Grammar Recognizer** will interpret your speech and change the colors of the shapes based on the recognition. An example command is "blue circle, yellow square".
- Perform another air-tap gesture to dismiss the toolbox.

The End

Congratulations! You have now completed **MR Input 212: Voice**.

- You know the Dos and Don'ts of voice commands.
- You saw how tooltips were employed to make users aware of voice commands.
- You saw several types of feedback used to acknowledge that the user's voice was heard.
- You know how to switch between the Keyword Recognizer and the Dictation Recognizer, and how these two features understand and interpret your voice.
- You learned how to use an SRGS file and the Grammar Recognizer for speech recognition in your application.

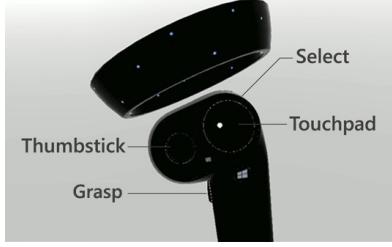
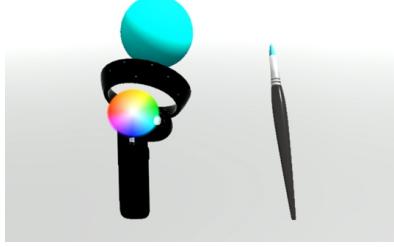
MR Input 213: Motion controllers

11/6/2018 • 20 minutes to read • [Edit Online](#)

Motion controllers in the mixed reality world add another level of interactivity. With [motion controllers](#), we can directly interact with objects in a more natural way, similar to our physical interactions in real life, increasing immersion and delight in your app experience.

In MR Input 213, we will explore the motion controller's input events by creating a simple spatial painting experience. With this app, users can paint in three-dimensional space with various types of brushes and colors.

Topics covered in this tutorial

		
Controller visualization	Controller input events	Custom controller and UI
Learn how to render motion controller models in Unity's game mode and runtime.	Understand different types of button events and their applications.	Learn how to overlay UI elements on top of the controller or fully customize it.

Device support

COURSE	HOOLENS	IMMERSIVE HEADSETS
MR Input 213: Motion controllers		<input checked="" type="checkbox"/> <input type="checkbox"/>

Before you start

Prerequisites

See the installation checklist for immersive headsets on [this page](#).

- This tutorial requires [Unity 2017.2.1p2](#)

Project files

- [Download the files](#) required by the project and extract the files to the Desktop.

NOTE

If you want to look through the source code before downloading, it's [available on GitHub](#).

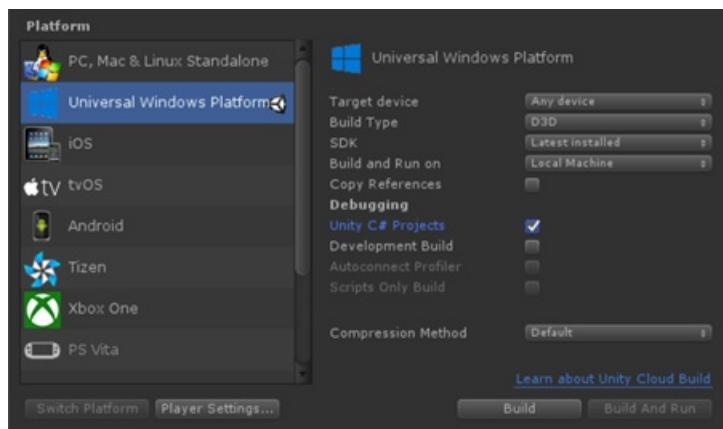
Unity setup

Objectives

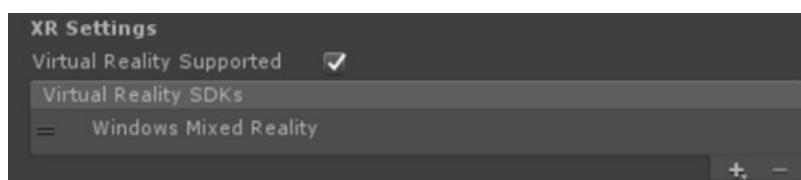
- Optimize Unity for Windows Mixed Reality development
- Setup Mixed Reality Camera
- Setup environment

Instructions

- Start Unity.
- Select **Open**.
- Navigate to your Desktop and find the **MixedReality213-master** folder you previously unarchived.
- Click **Select Folder**.
- Once Unity finishes loading project files, you will be able to see Unity editor.
- In Unity, select **File > Build Settings**.



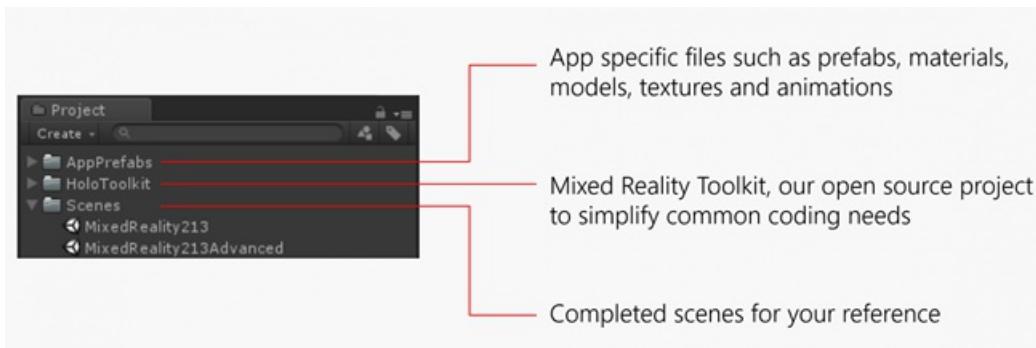
- Select **Universal Windows Platform** in the **Platform** list and click the **Switch Platform** button.
- Set Target Device to **Any device**
- Set Build Type to **D3D**
- Set SDK to **Latest Installed**
- Check **Unity C# Projects**
 - This allows you modify script files in the Visual Studio project without rebuilding Unity project.
- Click **Player Settings**.
- In the **Inspector** panel, scroll down to the bottom
- In XR Settings, check **Virtual Reality Supported**
- Under Virtual Reality SDKs, select **Windows Mixed Reality**



- Close **Build Settings** window.

Project structure

This tutorial uses **MixedReality Toolkit - Unity**. You can find the releases on [this page](#).

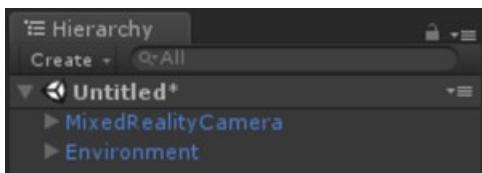


Completed scenes for your reference

- You will find two completed Unity scenes under **Scenes** folder:
 - **MixedReality213**: Completed scene with single brush
 - **MixedReality213Advanced**: Completed scene for advanced design with multiple brushes

New Scene setup for the tutorial

- In Unity, click **File > New Scene**
- Delete **Main Camera** and **Directional Light**
- From the **Project panel**, search and drag the following prefabs into the **Hierarchy** panel:
 - Assets/HoloToolkit/Input/Prefabs/**MixedRealityCamera**
 - Assets/AppPrefabs/**Environment**

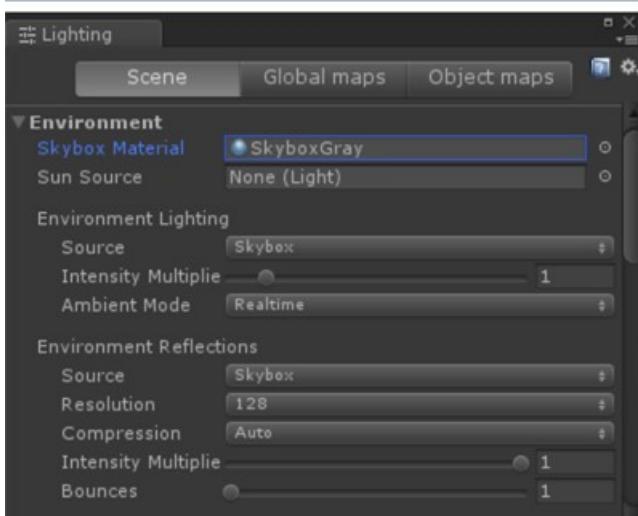


- There are two camera prefabs in Mixed Reality Toolkit:
 - **MixedRealityCamera.prefab**: Camera only
 - **MixedRealityCameraParent.prefab**: Camera + Teleportation + Boundary
- In this tutorial, we will use **MixedRealityCamera** without teleportation feature. Because of this, we added simple **Environment** prefab which contains a basic floor to make the user feel grounded.
- To learn more about the teleportation with **MixedRealityCameraParent**, see [Advanced design - Teleportation and locomotion](#)

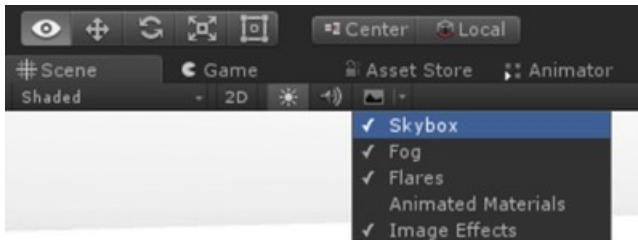
Skybox setup

- Click **Window > Lighting > Settings**
- Click the circle on the right side of the **Skybox Material** field
- Type in 'gray' and select **SkyboxGray**

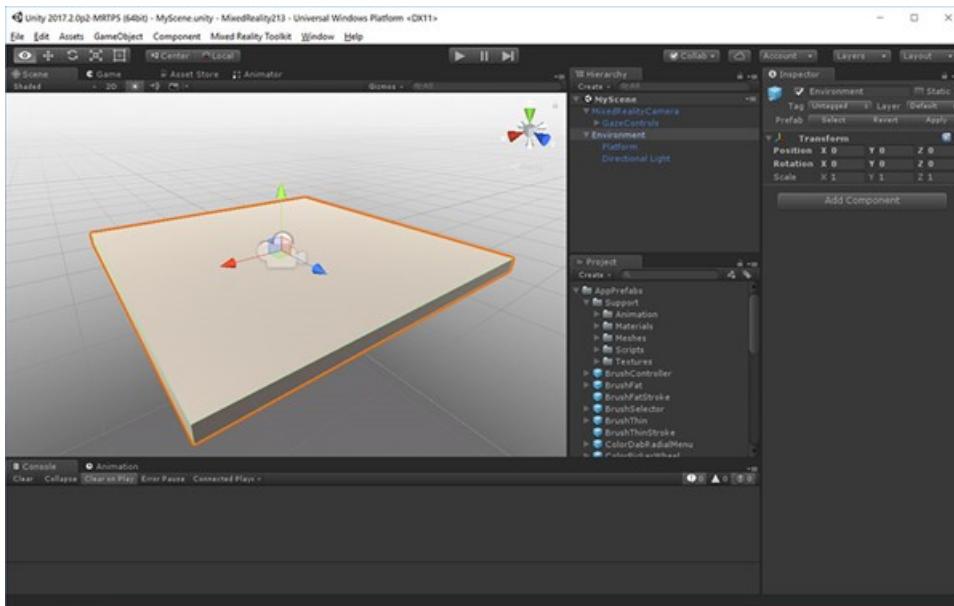
(Assets/AppPrefabs/Support/Materials/SkyboxGray.mat)



- Check **Skybox** option to be able to see assigned gray gradient skybox



- The scene with MixedRealityCamera, Environment and gray skybox will look like this.



- Click **File > Save Scene as**
- Save** your scene under Scenes folder with any name

Chapter 1 - Controller visualization

Objectives

- Learn how to render motion controller models in Unity's game mode and at runtime.

Windows Mixed Reality provides an animated controller model for controller visualization. There are several approaches you can take for controller visualization in your app:

- Default - Using default controller without modification
- Hybrid - Using default controller, but customizing some of its elements or overlaying UI components
- Replacement - Using your own customized 3D model for the controller

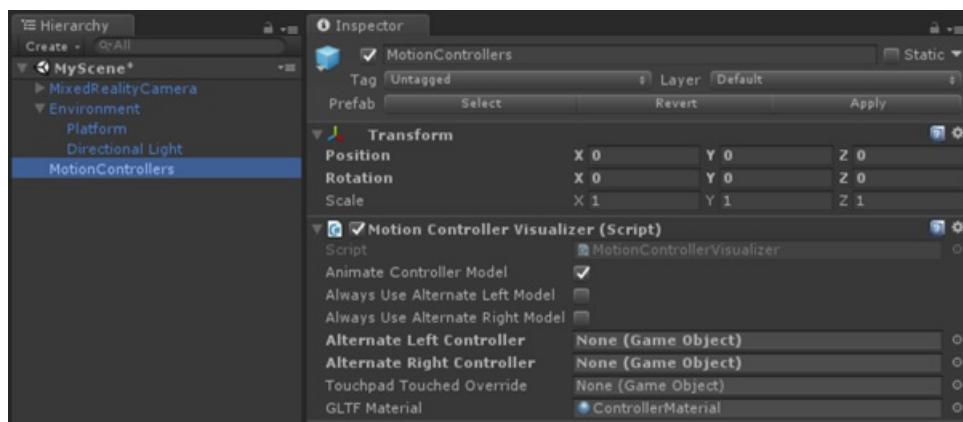
In this chapter, we will learn about the examples of these controller customizations.

Instructions

- In the **Project** panel, type **MotionControllers** in the search box . You can also find it under Assets/HoloToolkit/Input/Prefabs/.
- Drag the **MotionControllers** prefab into the **Hierarchy** panel.
- Click on the **MotionControllers** prefab in the **Hierarchy** panel.

MotionControllers prefab

MotionControllers prefab has a **MotionControllerVisualizer** script which provides the slots for alternate controller models. If you assign your own custom 3D models such as a hand or a sword and check 'Always Use Alternate Left/Right Model', you will see them instead of the default model. We will use this slot in Chapter 4 to replace the controller model with a brush.



Instructions

- In the **Inspector** panel, double click **MotionControllerVisualizer** script to see the code in the Visual Studio

MotionControllerVisualizer script

The **MotionControllerVisualizer** and **MotionControllerInfo** classes provide the means to access & modify the default controller models. **MotionControllerVisualizer** subscribes to Unity's **InteractionSourceDetected** event and automatically instantiates controller models when they are found.

```
protected override void Awake()
{
    ...
    InteractionManager.InteractionSourceDetected += InteractionManager_InteractionSourceDetected;
    InteractionManager.InteractionSourceLost += InteractionManager_InteractionSourceLost;
    ...
}
```

The controller models are delivered according to [the glTF specification](#). This format has been created to provide a common format, while improving the process behind transmitting and unpacking 3D assets. In this case, we need to retrieve and load the controller models at runtime, as we want to make the user's experience as seamless as possible, and it's not guaranteed which version of the motion controllers the user might be using. This course, via the Mixed Reality Toolkit, uses a version of the Khronos Group's [UnityGLTF project](#).

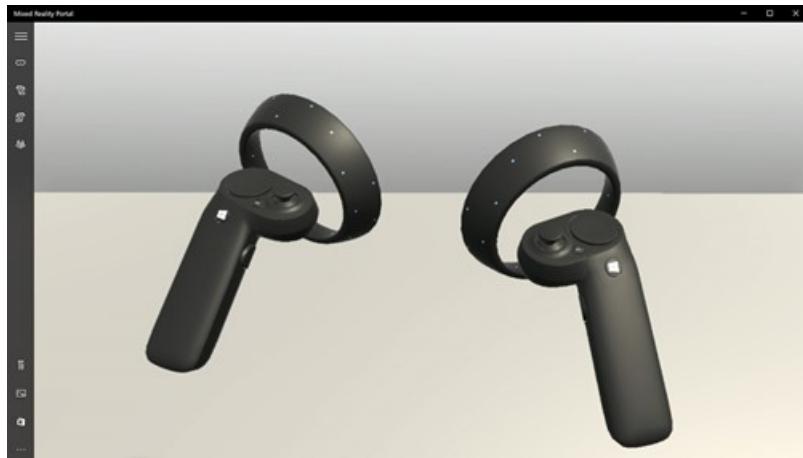
Once the controller has been delivered, scripts can use **MotionControllerInfo** to find the transforms for specific controller elements so they can correctly position themselves.

In a later chapter, we will learn how to use these scripts to attach UI elements to the controllers.

In some scripts, you will find code blocks with `#if !UNITY_EDITOR` or `UNITY_WSA`. These code blocks run only on the UWP runtime when you deploy to Windows. This is because the set of APIs used by the Unity editor and the UWP app runtime are different.

- **Save** the scene and click the **play** button.

You will be able to see the scene with motion controllers in your headset. You can see detailed animations for button clicks, thumbstick movement, and touchpad touch highlighting.



Chapter 2 - Attaching UI elements to the controller

Objectives

- Learn about the elements of the motion controllers
- Learn how to attach objects to specific parts of the controllers

In this chapter, you will learn how to add user interface elements to the controller which the user can easily access and manipulate at anytime. You will also learn how to add a simple color picker UI using the touchpad input.

Instructions

- In the **Project** panel, search **MotionControllerInfo** script.
- From the search result, double click **MotionControllerInfo** script to see the code in Visual Studio.

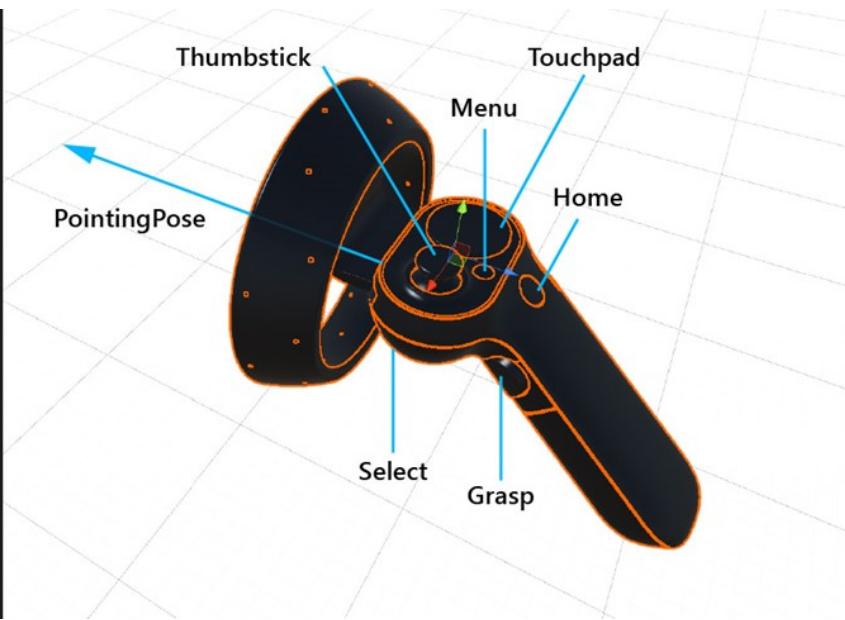
MotionControllerInfo script

The first step is to choose which element of the controller you want the UI to attach to. These elements are defined in **ControllerElementEnum** in **MotionControllerInfo.cs**.

```

public enum ControllerElementEnum
{
    // Controller button elements
    Home,
    Menu,
    Grasp,
    Thumbstick,
    Select,
    Touchpad,
    // Controller body elements & poses
    PointingPose
}

```



- **Home**
- **Menu**
- **Grasp**
- **Thumbstick**
- **Select**
- **Touchpad**
- **Pointing pose** – this element represents the tip of the controller pointing forward direction.

Instructions

- In the **Project** panel, search **AttachToController** script.
- From the search result, double click **AttachToController** script to see the code in Visual Studio.

AttachToController script

The **AttachToController** script provides a simple way to attach any objects to a specified controller handedness and element.

In **AttachElementToController()**,

- Check handedness using **MotionControllerInfo.Handedness**
- Get specific element of the controller using **MotionControllerInfo.TryGetElement()**
- After retrieving the element's transform from the controller model, parent the object under it and set object's local position & rotation to zero.

```

public MotionControllerInfo.ControllerElementEnum Element { get { return element; } }

private void AttachElementToController(MotionControllerInfo newController)
{
    if (!IsAttached && newController.Handedness == handedness)
    {
        if (!newController.TryGetElement(element, out elementTransform))
        {
            Debug.LogError("Unable to find element of type " + element + " under controller " +
newController.ControllerParent.name + "; not attaching.");
            return;
        }

        controller = newController;

        SetChildrenActive(true);

        // Parent ourselves under the element and set our offsets
        transform.parent = elementTransform;
        transform.localPosition = positionOffset;
        transform.localEulerAngles = rotationOffset;
        if (setScaleOnAttach)
        {
            transform.localScale = scale;
        }

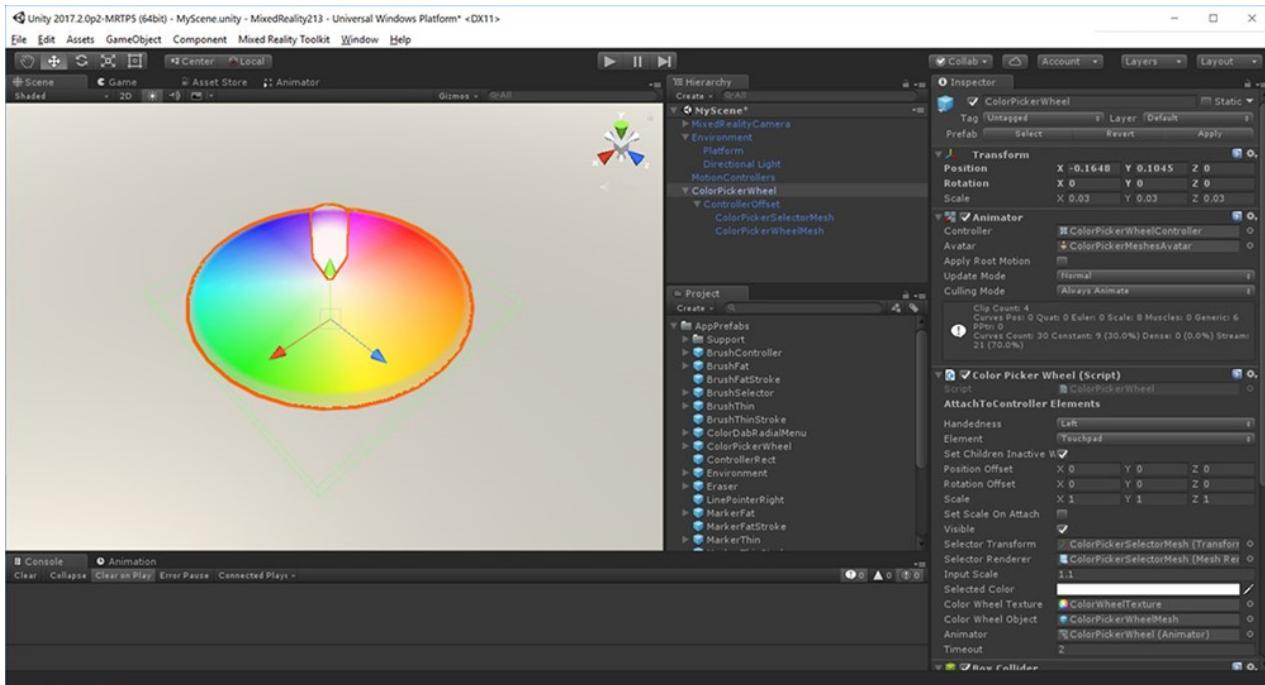
        // Announce that we're attached
        OnAttachToController();
        IsAttached = true;
    }
}

```

The simplest way to use **AttachToController** script is to inherit from it, as we've done in the case of **ColorPickerWheel**. Simply override the **OnAttachToController** and **OnDetachFromController** functions to perform your setup / breakdown when the controller is detected / disconnected.

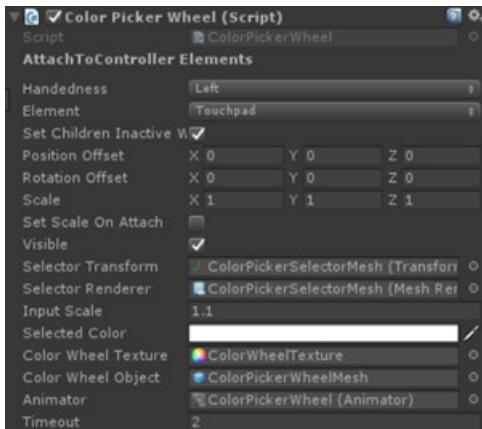
Instructions

- In the **Project** panel, type in the search box **ColorPickerWheel**. You can also find it under Assets/AppPrefabs/.
- Drag **ColorPickerWheel** prefab into the **Hierarchy** panel.
- Click the **ColorPickerWheel** prefab in the **Hierarchy** panel.
- In the **Inspector** panel, double click **ColorPickerWheel** Script to see the code in Visual Studio.



ColorPickerWheel script

Since **ColorPickerWheel** inherits **AttachToController**, it shows **Handedness** and **Element** in the **Inspector** panel. We'll be attaching the UI to the Touchpad element on the left controller.



ColorPickerWheel overrides the **OnAttachToController** and **OnDetachFromController** to subscribe to the input event which will be used in next chapter for color selection with touchpad input.

```
public class ColorPickerWheel : AttachToController, IPointerTarget
{
    protected override void OnAttachToController()
    {
        // Subscribe to input now that we're parented under the controller
        InteractionManager.InteractionSourceUpdated += InteractionSourceUpdated;
    }

    protected override void OnDetachFromController()
    {
        Visible = false;

        // Unsubscribe from input now that we've detached from the controller
        InteractionManager.InteractionSourceUpdated -= InteractionSourceUpdated;
    }
}
```

- Save the scene and click the play button.

Alternative method for attaching objects to the controllers

We recommend that your scripts inherit from **AttachToController** and override **OnAttachToController**. However, this may not always be possible. An alternative is using it as a standalone component. This can be useful when you want to attach an existing prefab to a controller without refactoring your scripts. Simply have your class wait for `IsAttached` to be set to true before performing any setup. The simplest way to do this is by using a coroutine for 'Start.'

```
private IEnumerator Start() {
    AttachToController attach = gameObject.GetComponent<AttachToController>();

    while (!attach.IsAttached) {
        yield return null;
    }

    // Perform setup here
}
```

Chapter 3 - Working with touchpad input

Objectives

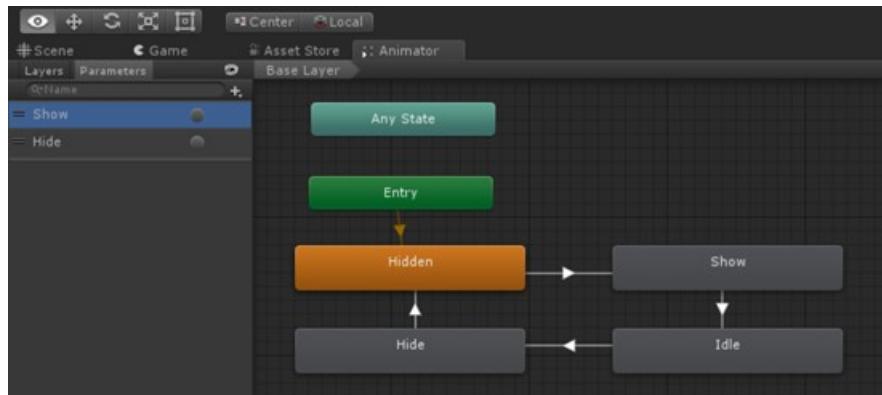
- Learn how to get touchpad input data events
- Learn how to use touchpad axis position information for your app experience

Instructions

- In the **Hierarchy** panel, click **ColorPickerWheel**
- In the **Inspector** panel, under **Animator**, double click **ColorPickerWheelController**
- You will be able to see **Animator** tab opened

Showing/hiding UI with Unity's Animation controller

To show and hide the **ColorPickerWheel** UI with animation, we are using [Unity's animation system](#). Setting the **ColorPickerWheel**'s **Visible** property to true or false triggers **Show** and **Hide** animation triggers. **Show** and **Hide** parameters are defined in the **ColorPickerWheelController** animation controller.



Instructions

- In the **Hierarchy** panel, select **ColorPickerWheel** prefab
- In the **Inspector** panel, double click **ColorPickerWheel** script to see the code in the Visual Studio

ColorPickerWheel script

ColorPickerWheel subscribes to Unity's **InteractionSourceUpdated** event to listen for touchpad events.

In **InteractionSourceUpdated()**, the script first checks to ensure that it:

- is actually a touchpad event (`obj.state.touchpadTouched`)
- originates from the left controller (`obj.state.source.handedness`)

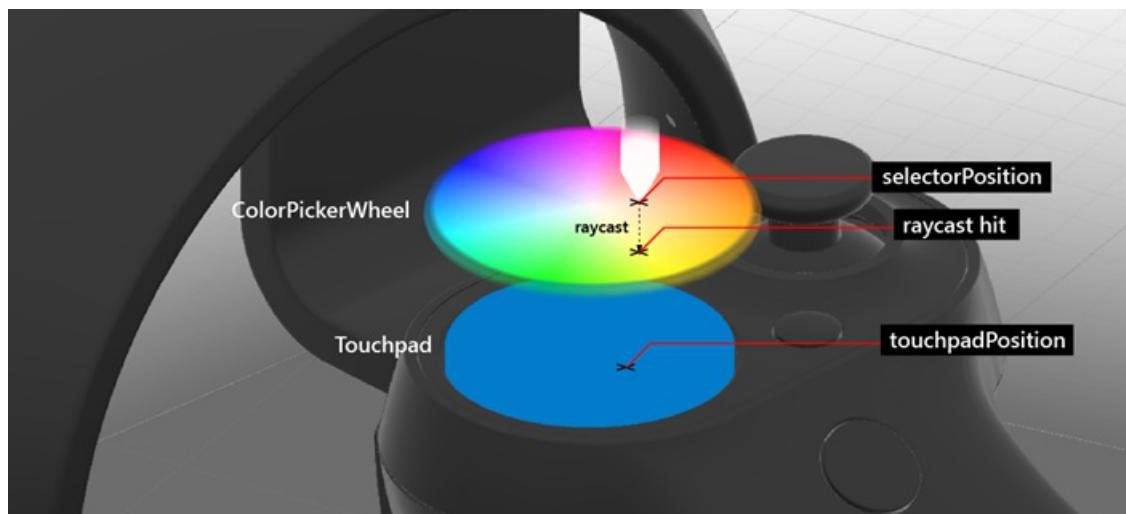
If both are true, the touchpad position (`obj.state.touchpadPosition`) is assigned to **selectorPosition**.

```
private void InteractionSourceUpdated(InteractionSourceUpdatedEventArgs obj)
{
    if (obj.state.source.handedness == handedness && obj.state.touchpadTouched)
    {
        Visible = true;
        selectorPosition = obj.state.touchpadPosition;
    }
}
```

In **Update()**, based on **visible** property, it triggers Show and Hide animation triggers in the color picker's animator component

```
if (visible != visibleLastFrame)
{
    if (visible)
    {
        animator.SetTrigger("Show");
    }
    else
    {
        animator.SetTrigger("Hide");
    }
}
```

In **Update()**, **selectorPosition** is used to cast a ray at the color wheel's mesh collider, which returns a UV position. This position can then be used to find the pixel coordinate and color value of the color wheel's texture. This value is accessible to other scripts via the **SelectedColor** property.



```

...
    // Clamp selector position to a radius of 1
    Vector3 localPosition = new Vector3(selectorPosition.x * inputScale, 0.15f, selectorPosition.y * inputScale);
    if (localPosition.magnitude > 1)
    {
        localPosition = localPosition.normalized;
    }
    selectorTransform.localPosition = localPosition;

    // Raycast the wheel mesh and get its UV coordinates
    Vector3 raycastStart = selectorTransform.position + selectorTransform.up * 0.15f;
    RaycastHit hit;
    Debug.DrawLine(raycastStart, raycastStart - (selectorTransform.up * 0.25f));

    if (Physics.Raycast(raycastStart, -selectorTransform.up, out hit, 0.25f, 1 << colorWheelObject.layer,
QueryTriggerInteraction.Ignore))
    {
        // Get pixel from the color wheel texture using UV coordinates
        Vector2 uv = hit.textureCoord;
        int pixelX = Mathf.FloorToInt(colorWheelTexture.width * uv.x);
        int pixelY = Mathf.FloorToInt(colorWheelTexture.height * uv.y);
        selectedColor = colorWheelTexture.GetPixel(pixelX, pixelY);
        selectedColor.a = 1f;
    }
    // Set the selector's color and blend it with white to make it visible on top of the wheel
    selectorRenderer.material.color = Color.Lerp (selectedColor, Color.white, 0.5f);
}

```

Chapter 4 - Overriding controller model

Objectives

- Learn how to override the controller model with a custom 3D model.



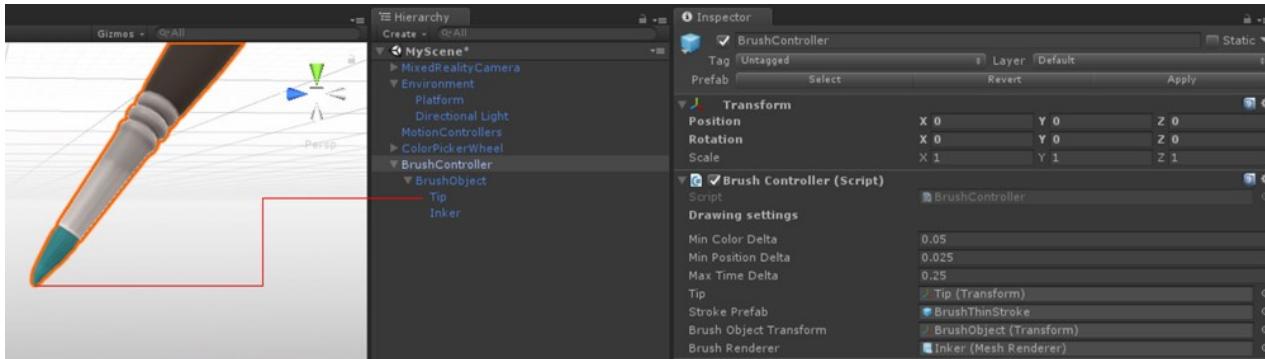
Instructions

- Click **MotionControllers** in the **Hierarchy** panel.
- Click the circle on the right side of the **Alternate Right Controller** field.
- Type in '**BrushController**' and select the prefab from the result. You can find it under Assets/AppPrefabs/**BrushController**.
- Check **Always Use Alternate Right Model**



The **BrushController** prefab does not have to be included in the **Hierarchy** panel. However, to check out its child components:

- In the **Project** panel, type in **BrushController** and drag **BrushController** prefab into the **Hierarchy** panel.



You will find the **Tip** component in **BrushController**. We will use its transform to start/stop drawing lines.

- Delete the **BrushController** from the **Hierarchy** panel.
- Save** the scene and click the **play** button. You will be able to see the brush model replaced the right-hand motion controller.

Chapter 5 - Painting with Select input

Objectives

- Learn how to use the Select button event to start and stop a line drawing

Instructions

- Search **BrushController** prefab in the **Project** panel.
- In the **Inspector** panel, double click **BrushController** Script to see the code in Visual Studio

BrushController script

BrushController subscribes to the **InteractionManager's** **InteractionSourcePressed** and **InteractionSourceReleased** events. When **InteractionSourcePressed** event is triggered, the brush's **Draw** property is set to true; when **InteractionSourceReleased** event is triggered, the brush's **Draw** property is set to false.

```

private void InteractionSourcePressed(InteractionSourcePressedEventArgs obj)
{
    if (obj.state.source.handedness == InteractionSourceHandedness.Right && obj.pressType ==
InteractionSourcePressType.Select)
    {
        Draw = true;
    }
}

private void InteractionSourceReleased(InteractionSourceReleasedEventArgs obj)
{
    if (obj.state.source.handedness == InteractionSourceHandedness.Right && obj.pressType ==
InteractionSourcePressType.Select)
    {
        Draw = false;
    }
}

```

While **Draw** is set to true, the brush will generate points in an instantiated Unity **LineRenderer**. A reference to this prefab is kept in the brush's **Stroke Prefab** field.

```

private IEnumerator DrawOverTime()
{
    // Get the position of the tip
    Vector3 lastPointPosition = tip.position;

    ...

    // Create a new brush stroke
    GameObject newStroke = Instantiate(strokePrefab);
    LineRenderer line = newStroke.GetComponent<LineRenderer>();
    newStroke.transform.position = startPosition;
    line.SetPosition(0, tip.position);
    float initialWidth = line.widthMultiplier;

    // Generate points in an instantiated Unity LineRenderer
    while (draw)
    {
        // Move the last point to the draw point position
        line.SetPosition(line.positionCount - 1, tip.position);
        line.material.color = colorPicker.SelectedColor;
        brushRenderer.material.color = colorPicker.SelectedColor;
        lastPointAddedTime = Time.unscaledTime;
        // Adjust the width between 1x and 2x width based on strength of trigger pull
        line.widthMultiplier = Mathf.Lerp(initialWidth, initialWidth * 2, width);

        if (Vector3.Distance(lastPointPosition, tip.position) > minPositionDelta || Time.unscaledTime >
lastPointAddedTime + maxTimeDelta)
        {
            // Spawn a new point
            lastPointAddedTime = Time.unscaledTime;
            lastPointPosition = tip.position;
            line.positionCount += 1;
            line.SetPosition(line.positionCount - 1, lastPointPosition);
        }
        yield return null;
    }
}

```

To use the currently selected color from the color picker wheel UI, **BrushController** needs to have a reference to the **ColorPickerWheel** object. Because the **BrushController** prefab is instantiated at runtime as a replacement controller, any references to objects in the scene will have to be set at runtime. In this case we use **GameObject.FindObjectOfType** to locate the **ColorPickerWheel**:

```

private void OnEnable()
{
    // Locate the ColorPickerWheel
    colorPicker = FindObjectOfType<ColorPickerWheel>();

    // Assign currently selected color to the brush's material color
    brushRenderer.material.color = colorPicker.SelectedColor;
    ...
}

```

- **Save** the scene and click the **play** button. You will be able to draw the lines and paint using the select button on the right-hand controller.

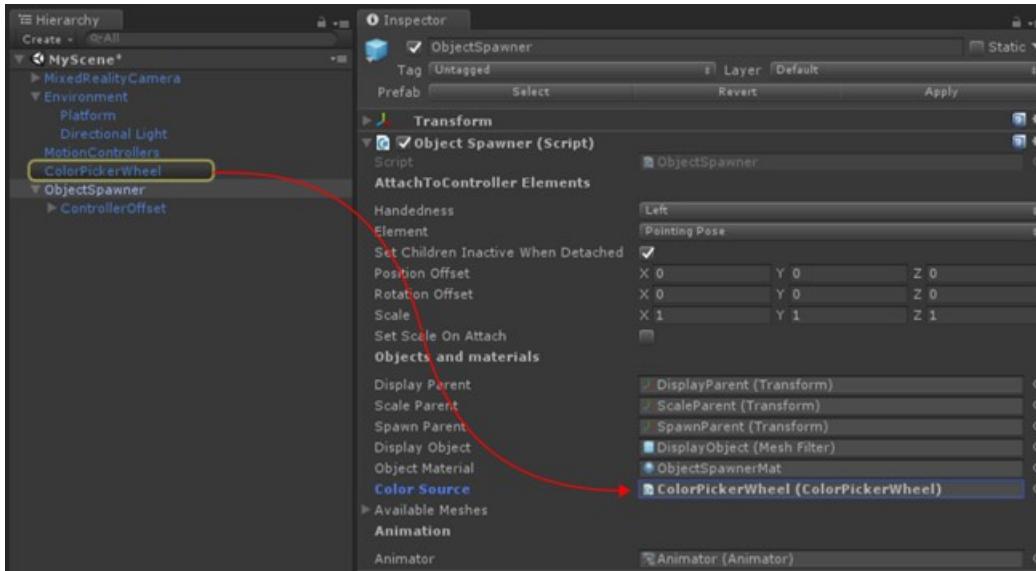
Chapter 6 - Object spawning with Select input

Objectives

- Learn how to use Select and Grasp button input events
- Learn how to instantiate objects

Instructions

- In the **Project** panel, type **ObjectSpawner** in the search box. You can also find it under Assets/AppPrefabs/
- Drag the **ObjectSpawner** prefab into the **Hierarchy** panel.
- Click **ObjectSpawner** in the **Hierarchy** panel.
- **ObjectSpawner** has a field named **Color Source**.
- From the **Hierarchy** panel, drag the **ColorPickerWheel** reference into this field.



- Click the **ObjectSpawner** prefab in the **Hierarchy** panel.
- In the **Inspector** panel, double click **ObjectSpawner** Script to see the code in Visual Studio.

ObjectSpawner script

The **ObjectSpawner** instantiates copies of a primitive mesh (cube, sphere, cylinder) into the space. When a **InteractionSourcePressed** is detected it checks the handedness and if it's an **InteractionSourcePressType.Grasp** or **InteractionSourcePressType.Select** event.

For a **Grasp** event, it increments the index of current mesh type (sphere, cube, cylinder)

```

private void InteractionSourcePressed(InteractionSourcePressedEventArgs obj)
{
    // Check handedness, see if it is left controller
    if (obj.state.source.handedness == handedness)
    {
        switch (obj.pressType)
        {
            // If it is Select button event, spawn object
            case InteractionSourcePressType.Select:
                if (state == StateEnum.Idle)
                {
                    // We've pressed the grasp - enter spawning state
                    state = StateEnum.Spawning;
                    SpawnObject();
                }
                break;

            // If it is Grasp button event
            case InteractionSourcePressType.Grasp:

                // Increment the index of current mesh type (sphere, cube, cylinder)
                meshIndex++;
                if (meshIndex >= NumAvailableMeshes)
                {
                    meshIndex = 0;
                }
                break;

            default:
                break;
        }
    }
}

```

For a **Select** event, in **SpawnObject()**, a new object is instantiated, un-parented and released into the world.

```

private void SpawnObject()
{
    // Instantiate the spawned object
    GameObject newObjet = Instantiate(displayObject.gameObject, spawnParent);
    // Detatch the newly spawned object
    newObjet.transform.parent = null;
    // Reset the scale transform to 1
    scaleParent.localScale = Vector3.one;
    // Set its material color so its material gets instantiated
    newObjet.GetComponent<Renderer>().material.color = colorSource.SelectedColor;
}

```

The **ObjectSpawner** uses the **ColorPickerWheel** to set the color of the display object's material. Spawns objects are given an instance of this material so they will retain their color.

- **Save** the scene and click the **play** button.

You will be able to change the objects with the Grasp button and spawn objects with the Select button.

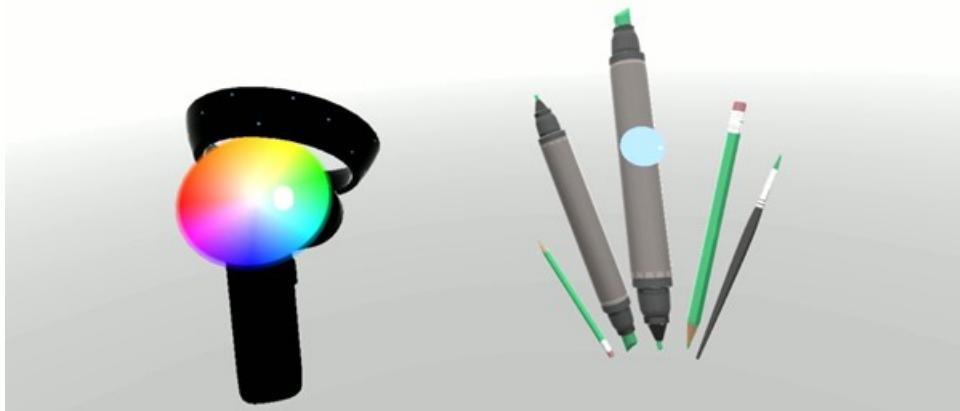
Build and deploy app to Mixed Reality Portal

- In Unity, select **File > Build Settings**.
- Click **Add Open Scenes** to add current scene to the **Scenes In Build**.
- Click **Build**.
- Create a **New Folder** named "App".

- Single click the **App** folder.
- Click **Select Folder**.
- When Unity is done, a File Explorer window will appear.
- Open the **App** folder.
- Double click **YourSceneName.sln** Visual Studio Solution file.
- Using the top toolbar in Visual Studio, change the target from Debug to **Release** and from ARM to **X64**.
- Click on the drop-down arrow next to the Device button, and select **Local Machine**.
- Click **Debug -> Start Without debugging** in the menu or press **Ctrl + F5**.

Now the app is built and installed in Mixed Reality Portal. You can launch it again through Start menu in Mixed Reality Portal.

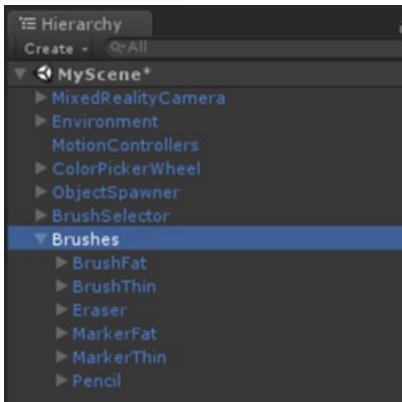
Advanced design - Brush tools with radial layout



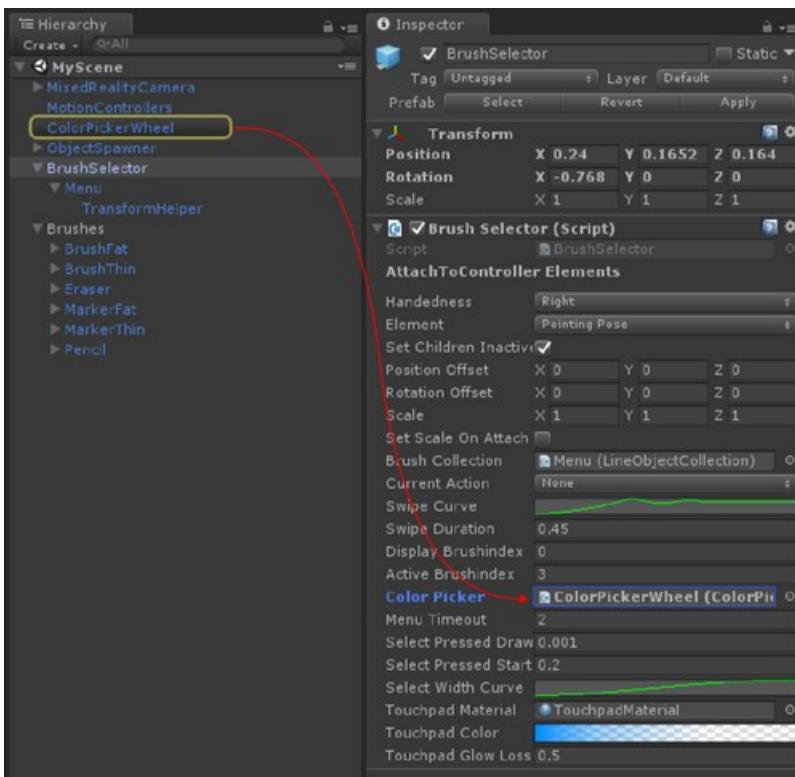
In this chapter, you will learn how to replace the default motion controller model with a custom brush tool collection. For your reference, you can find the completed scene **MixedReality213Advanced** under **Scenes** folder.

Instructions

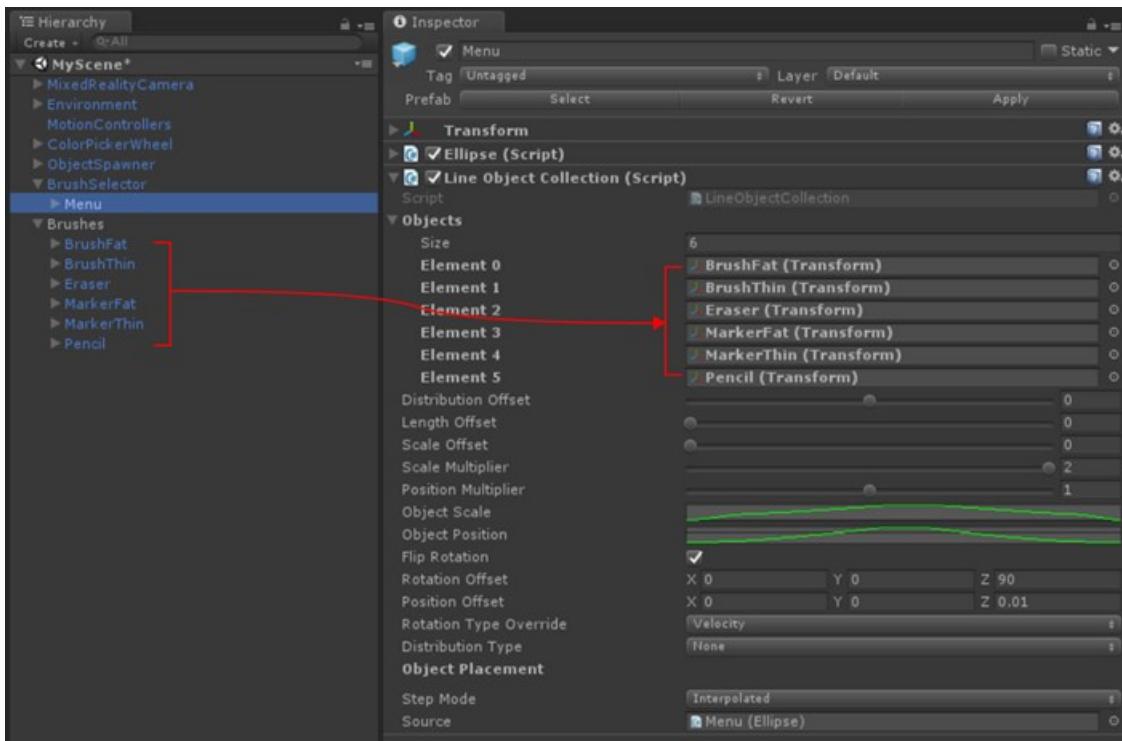
- In the **Project** panel, type **BrushSelector** in the search box . You can also find it under Assets/AppPrefabs/
- Drag the **BrushSelector** prefab into the **Hierarchy** panel.
- For organization, create an empty GameObject called **Brushes**
- Drag following prefabs from the **Project** panel into **Brushes**
 - Assets/AppPrefabs/**BrushFat**
 - Assets/AppPrefabs/**BrushThin**
 - Assets/AppPrefabs/**Eraser**
 - Assets/AppPrefabs/**MarkerFat**
 - Assets/AppPrefabs/**MarkerThin**
 - Assets/AppPrefabs/**Pencil**



- Click **MotionControllers** prefab in the **Hierarchy** panel.
- In the **Inspector** panel, uncheck **Always Use Alternate Right Model** on the **Motion Controller Visualizer**
- In the **Hierarchy** panel, click **BrushSelector**
- **BrushSelector** has a field named **ColorPicker**
- From the **Hierarchy** panel, drag the **ColorPickerWheel** into **ColorPicker** field in the **Inspector** panel.



- In the **Hierarchy** panel, under **BrushSelector** prefab, select the **Menu** object.
- In the **Inspector** panel, under the **LineObjectCollection** component, open the **Objects** array dropdown. You will see 6 empty slots.
- From the **Hierarchy** panel, drag each of the prefabs parented under the **Brushes** GameObject into these slots in any order. (Make sure you're dragging the prefabs from the scene, not the prefabs in the project folder.)



BrushSelector prefab

Since the **BrushSelector** inherits **AttachToController**, it shows **Handedness** and **Element** options in the **Inspector** panel. We selected **Right** and **Pointing Pose** to attach brush tools to the right hand controller with forward direction.

The **BrushSelector** makes use of two utilities:

- **Ellipse**: used to generate points in space along an ellipse shape.
- **LineObjectCollection**: distributes objects using the points generated by any Line class (eg, Ellipse). This is what we'll be using to place our brushes along the Ellipse shape.

When combined, these utilities can be used to create a radial menu.

LineObjectCollection script

LineObjectCollection has controls for the size, position and rotation of objects distributed along its line. This is useful for creating radial menus like the brush selector. To create the appearance of brushes that scale up from nothing as they approach the center selected position, the **ObjectScale** curve peaks in the center and tapers off at the edges.

BrushSelector script

In the case of the **BrushSelector**, we've chosen to use procedural animation. First, brush models are distributed in an ellipse by the **LineObjectCollection** script. Then, each brush is responsible for maintaining its position in the user's hand based on its **DisplayMode** value, which changes based on the selection. We chose a procedural approach because of the high probability of brush position transitions being interrupted as the user selects brushes. Mecanim animations can handle interruptions gracefully, but it tends to be more complicated than a simple Lerp operation.

BrushSelector uses a combination of both. When touchpad input is detected, brush options become visible and scale up along the radial menu. After a timeout period (which indicates that the user has made a selection) the brush options scale down again, leaving only the selected brush.

Visualizing touchpad input

Even in cases where the controller model has been completely replaced, it can be helpful to show input on the

original model inputs. This helps to ground the user's actions in reality. For the **BrushSelector** we've chosen to make the touchpad briefly visible when the input is received. This was done by retrieving the Touchpad element from the controller, replacing its material with a custom material, then applying a gradient to that material's color based on the last time touchpad input was received.

```
protected override void OnAttachToController()
{
    // Turn off the default controller's renderers
    controller.SetRenderersVisible(false);

    // Get the touchpad and assign our custom material to it
    Transform touchpad;
    if (controller.TryGetElement(MotionControllerInfo.ControllerElementEnum.Touchpad, out touchpad))
    {
        touchpadRenderer = touchpad.GetComponentInChildren<MeshRenderer>();
        originalTouchpadMaterial = touchpadRenderer.material;
        touchpadRenderer.material = touchpadMaterial;
        touchpadRenderer.enabled = true;
    }

    // Subscribe to input now that we're parented under the controller
    InteractionManager.InteractionSourceUpdated += InteractionSourceUpdated;
}

private void Update()
{
    ...
    // Update our touchpad material
    Color glowColor = touchpadColor.Evaluate((Time.unscaledTime - touchpadTouchTime) / touchpadGlowLossTime);
    touchpadMaterial.SetColor("_EmissionColor", glowColor);
    touchpadMaterial.SetColor("_Color", glowColor);
    ...
}
```

Brush tool selection with touchpad input

When the brush selector detects touchpad's pressed input, it checks the position of the input to determine if it was to the left or right.

Stroke thickness with selectPressedAmount

Instead of the **InteractionSourcePressType.Select** event in the **InteractionSourcePressed()**, you can get the analog value of the pressed amount through **selectPressedAmount**. This value can be retrieved in **InteractionSourceUpdated()**.

```

private void InteractionSourceUpdated(InteractionSourceUpdatedEventArgs obj)
{
    if (obj.state.source.handedness == handedness)
    {
        if (obj.state.touchpadPressed)
        {
            // Check which side we clicked
            if (obj.state.touchpadPosition.x < 0)
            {
                currentAction = SwipeEnum.Left;
            }
            else
            {
                currentAction = SwipeEnum.Right;
            }

            // Ping the touchpad material so it gets bright
            touchpadTouchTime = Time.unscaledTime;
        }
    }

    if (activeBrush != null)
    {
        // If the pressed amount is greater than our threshold, draw
        if (obj.state.selectPressedAmount >= selectPressedDrawThreshold)
        {
            activeBrush.Draw = true;
            activeBrush.Width = ProcessSelectPressedAmount(obj.state.selectPressedAmount);
        }
        else
        {
            // Otherwise, stop drawing
            activeBrush.Draw = false;
            selectPressedSmooth = 0f;
        }
    }
}
}

```

Eraser script

Eraser is a special type of brush that overrides the base **Brush's DrawOverTime()** function. While Draw is true, the eraser checks to see if its tip intersects with any existing brush strokes. If it does, they are added to a queue to be shrunk down and deleted.

Advanced design - Teleportation and locomotion

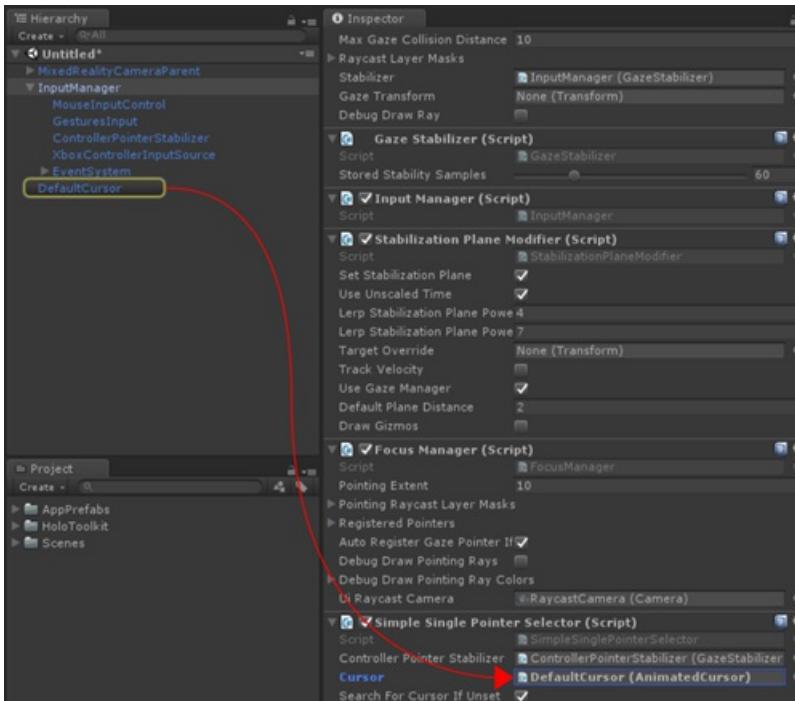
If you want to allow the user to move around the scene with teleportation using thumbstick, use **MixedRealityCameraParent** instead of **MixedRealityCamera**. You also need to add **InputManager** and **DefaultCusor**. Since **MixedRealityCameraParent** already includes **MotionControllers** and **Boundary** as child components, you should remove existing **MotionControllers** and **Environment** prefab.

Instructions

- In the **Hierarchy** panel, delete **MixedRealityCamera**, **Environment** and **MotionControllers**
- From the **Project panel**, search and drag the following prefabs into the **Hierarchy** panel:
 - Assets/AppPrefabs/Input/Prefabs/**MixedRealityCameraParent**
 - Assets/AppPrefabs/Input/Prefabs/**InputManager**
 - Assets/AppPrefabs/Input/Prefabs/Cursor/**DefaultCursor**



- In the **Hierarchy** panel, click **Input Manager**
- In the **Inspector** panel, scroll down to the **Simple Single Pointer Selector** section
- From the **Hierarchy** panel, drag **DefaultCursor** into **Cursor** field



- **Save** the scene and click the **play** button. You will be able to use the thumbstick to rotate left/right or teleport.

The end

And that's the end of this tutorial! You learned:

- How to work with motion controller models in Unity's game mode and runtime.
- How to use different types of button events and their applications.
- How to overlay UI elements on top of the controller or fully customize it.

You are now ready to start creating your own immersive experience with motion controllers!

Completed scenes

- In Unity's **Project** panel click on the **Scenes** folder.
- You will find two Unity scenes **MixedReality213** and **MixedReality213Advanced**.
 - **MixedReality213**: Completed scene with single brush
 - **MixedReality213Advanced**: Completed scene with multiple brush with select button's press amount example

See also

- [MR Input 213 project files](#)

- [Mixed Reality Toolkit - Motion Controller Test scene](#)
- [Mixed Reality Toolkit - Grab Mechanics](#)
- [Motion controller development guidelines](#)

MR Spatial 220: Spatial sound

11/6/2018 • 18 minutes to read • [Edit Online](#)

Spatial sound breathes life into holograms and gives them presence in our world. Holograms are composed of both light and sound, and if you happen to lose sight of your holograms, spatial sound can help you find them. Spatial sound is not like the typical sound that you would hear on the radio, it is sound that is positioned in 3D space. With spatial sound, you can make holograms sound like they're behind you, next to you, or even on your head! In this course, you will:

- Configure your development environment to use Microsoft Spatial Sound.
- Use Spatial Sound to enhance interactions.
- Use Spatial Sound in conjunction with Spatial Mapping.
- Understand sound design and mixing best practices.
- Use sound to enhance special effects and bring the user into the Mixed Reality world.

Device support

COURSE	HOLOLENS	IMMERSIVE HEADSETS
MR Spatial 220: Spatial sound	✓ <input type="checkbox"/>	✓ <input type="checkbox"/>

Before you start

Prerequisites

- A Windows 10 PC configured with the correct [tools installed](#).
- Some basic C# programming ability.
- You should have completed [MR Basics 101](#).
- A HoloLens device [configured for development](#).

Project files

- Download the [files](#) required by the project. Requires Unity 2017.2 or later.
 - If you still need Unity 5.6 support, please use [this release](#). This release may no longer be up-to-date.
 - If you still need Unity 5.5 support, please use [this release](#). This release may no longer be up-to-date.
 - If you still need Unity 5.4 support, please use [this release](#). This release may no longer be up-to-date.
- Un-archive the files to your desktop or other easy to reach location.

NOTE

If you want to look through the source code before downloading, it's [available on GitHub](#).

Errata and Notes

- "Enable Just My Code" needs to be disabled (*unchecked*) in Visual Studio under Tools->Options->Debugging in order to hit breakpoints in your code.

Chapter 1 - Unity Setup

Objectives

- Change Unity's sound configuration to use Microsoft Spatial Sound.
- Add 3D sound to an object in Unity.

Instructions

- Start Unity.
- Select **Open**.
- Navigate to your Desktop and find the folder you previously un-archived.
- Click on the **Starting\Decibel** folder and then press the **Select Folder** button.
- Wait for the project to load in Unity.
- In the **Project** panel, open **Scenes\Decibel.unity**.
- In the **Hierarchy** panel, expand **HologramCollection** and select **POLY**.
- In the Inspector, expand **AudioSource** and notice that there is no **Spatialize** check box.

By default, Unity does not load a spatializer plugin. The following steps will enable Spatial Sound in the project.

- In Unity's top menu, go to **Edit > Project Settings > Audio**.
- Find the **Spatializer Plugin** dropdown, and select **MS HRTF Spatializer**.
- In the **Hierarchy** panel, select **HologramCollection > POLY**.
- In the **Inspector** panel, find the **Audio Source** component.
- Check the **Spatialize** checkbox.
- Drag the **Spatial Blend** slider all the way to **3D**, or enter **1** in the edit box.

We will now build the project in Unity and configure the solution in Visual Studio.

1. In Unity, select **File > Build Settings**.
2. Click **Add Open Scenes** to add the scene.
3. Select **Universal Windows Platform** in the **Platform** list and click **Switch Platform**.
4. If you're specifically developing for HoloLens, set **Target device** to **HoloLens**. Otherwise, leave it on **Any device**.
5. Ensure **Build Type** is set to **D3D** and **SDK** is set to **Latest installed** (which should be SDK 16299 or newer).
6. Click **Build**.
7. Create a **New Folder** named "App".
8. Single click the **App** folder.
9. Press **Select Folder**.

When Unity is done, a File Explorer window will appear.

1. Open the **App** folder.
2. Open the **Decibel Visual Studio Solution**.

If deploying to HoloLens:

1. Using the top toolbar in Visual Studio, change the target from Debug to **Release** and from ARM to **x86**.
2. Click on the drop down arrow next to the Local Machine button, and select **Remote Machine**.
3. Enter **your HoloLens device IP address** and set Authentication Mode to **Universal (Unencrypted Protocol)**. Click **Select**. If you do not know your device IP address, look in **Settings > Network & Internet > Advanced Options**.
4. In the top menu bar, click **Debug -> Start Without debugging** or press **Ctrl + F5**. If this is the first time deploying to your device, you will need to [pair it with Visual Studio](#).

If deploying to an immersive headset:

1. Using the top toolbar in Visual Studio, change the target from Debug to **Release** and from ARM to **x64**.
2. Make sure the deployment target is set to **Local Machine**.
3. In the top menu bar, click **Debug -> Start Without debugging** or press **Ctrl + F5**.

Chapter 2 - Spatial Sound and Interaction

Objectives

- Enhance hologram realism using sound.
- Direct the user's gaze using sound.
- Provide gesture feedback using sound.

Part 1 - Enhancing Realism

Key Concepts

- Spatialize hologram sounds.
- Sound sources should be placed at an appropriate location on the hologram.

The appropriate location for the sound is going to depend on the hologram. For example, if the hologram is of a human, the sound source should be located near the mouth and not the feet.

Instructions

The following instructions will attach a spatialized sound to a hologram.

- In the **Hierarchy** panel, expand **HologramCollection** and select **POLY**.
- In the **Inspector** panel, in the **AudioSource**, click the circle next to **AudioClip** and select **PolyHover** from the pop-up.
- Click the circle next to **Output** and select **SoundEffects** from the pop-up.

Project Decibel uses a Unity **AudioMixer** component to enable adjusting sound levels for groups of sounds. By grouping sounds this way, the overall volume can be adjusted while maintaining the relative volume of each sound.

- In the **AudioSource**, expand **3D Sound Settings**.
- Set **Doppler Level** to **0**.

Setting Doppler level to zero disables changes in pitch caused by motion (either of the hologram or the user). A classic example of Doppler is a fast-moving car. As the car approaches a stationary listener, the pitch of the engine rises. When it passes the listener, the pitch lowers with distance.

Part 2 - Directing the User's Gaze

Key Concepts

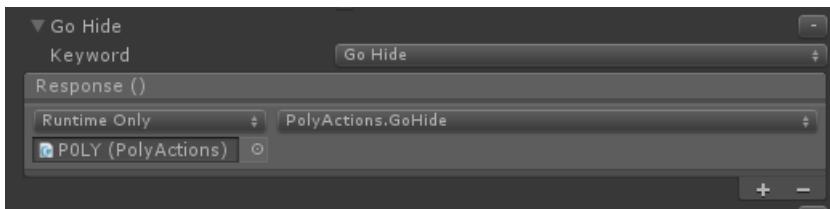
- Use sound to call attention to important holograms.
- The ears help direct where the eyes should look.
- The brain has some learned expectations.

One example of learned expectations is that birds are generally above the heads of humans. If a user hears a bird sound, their initial reaction is to look up. Placing a bird below the user can lead to them facing the correct direction of the sound, but being unable to find the hologram based on the expectation of needing to look up.

Instructions

The following instructions enable POLY to hide behind you, so that you can use sound to locate the hologram.

- In the **Hierarchy** panel, select **Managers**.
- In the **Inspector** panel, find **Speech Input Handler**.
- In **Speech Input Handler**, expand **Go Hide**.
- Change **No Function** to **PolyActions.GoHide**.



Part 3 - Gesture Feedback

Key Concepts

- Provide the user with positive gesture confirmation using sound
- Do not overwhelm the user - overly loud sounds get in the way
- Subtle sounds work best - do not overshadow the experience

Instructions

- In the **Hierarchy** panel, expand **HologramCollection**.
- Expand **EnergyHub** and select **Base**.
- In the **Inspector** panel, click **Add Component** and add **Gesture Sound Handler**.
- In **Gesture Sound Handler**, click the circle next to **Navigation Started Clip** and **Navigation Updated Clip** and select **RotateClick** from the pop-up for both.
- Double click on "GestureSoundHandler" to load in Visual Studio.

Gesture Sound Handler performs the following tasks:

- Create and configure an **AudioSource**.
- Place the **AudioSource** at the location of the appropriate **GameObject**.
- Plays the **AudioClip** associated with the gesture.

Build and Deploy

1. In Unity, select **File > Build Settings**.
2. Click **Build**.
3. Single click the **App** folder.
4. Press **Select Folder**.

Check that the Toolbar says "Release", "x86" or "x64", and "Remote Device". If not, this is the coding instance of Visual Studio. You may need to re-open the solution from the App folder.

- If prompted, reload the project files.
- As before, deploy from Visual Studio.

After the application is deployed:

- Observe how the sound changes as you move around POLY.
- Say "Go Hide" to make POLY move to a location behind you. Find it by the sound.
- Gaze at the base of the Energy Hub. Tap and drag left or right to rotate the hologram and notice how the clicking sound confirms the gesture.

Note: There is a text panel that will tag-along with you. This will contain the available voice commands that you can use throughout this course.

Chapter 3 - Spatial Sound and Spatial Mapping

Objectives

- Confirm interaction between holograms and the real world using sound.
- Occlude sound using the physical world.

Part 1 - Physical World Interaction

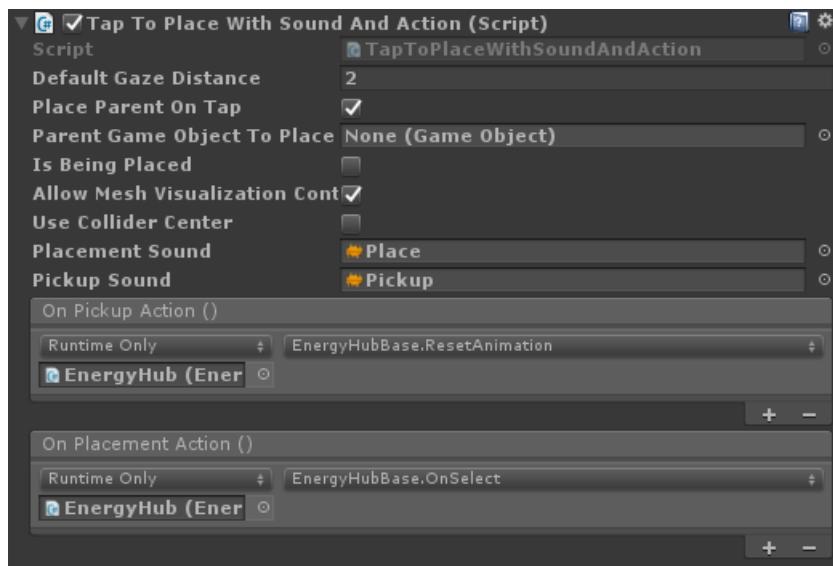
Key Concepts

- Physical objects generally make a sound when encountering a surface or another object.
- Sounds should be context appropriate within the experience.

For example, setting a cup on a table should make a quieter sound than dropping a boulder on a piece of metal.

Instructions

- In the **Hierarchy** panel, expand **HologramCollection**.
- Expand **EnergyHub**, select **Base**.
- In the **Inspector** panel, click **Add Component** and add **Tap To Place With Sound and Action**.
- In **Tap To Place With Sound and Action**:
 - Check **Place Parent On Tap**.
 - Set **Placement Sound** to **Place**.
 - Set **Pickup Sound** to **Pickup**.
 - Press the + in the bottom right under both **On Pickup Action** and **On Placement Action**. Drag EnergyHub from the scene into the **None (Object)** fields.
 - Under **On Pickup Action**, click on **No Function** -> **EnergyHubBase** -> **ResetAnimation**.
 - Under **On Placement Action**, click on **No Function** -> **EnergyHubBase** -> **OnSelect**.



Part 2 - Sound Occlusion

Key Concepts

- Sound, like light, can be occluded.

A classic example is a concert hall. When a listener is standing outside of the hall and the door is closed, the music sounds muffled. There is also typically a reduction in volume. When the door is opened, the full spectrum of the sound is heard at the actual volume. High frequency sounds are generally absorbed more than low frequencies.

Instructions

- In the **Hierarchy** panel, expand **HologramCollection** and select **POLY**.
- In the **Inspector** panel, click **Add Component** and add **Audio Emitter**.

The Audio Emitter class provides the following features:

- Restores any changes to the volume of the **AudioSource**.
- Performs a **Physics.RaycastNonAlloc** from the user's position in the direction of the **GameObject** to which the **AudioEmitter** is attached.

The RaycastNonAlloc method is used as a performance optimization to limit allocations as well as the number of

results returned.

- For each **IAudioInfluencer** encountered, calls the **ApplyEffect** method.
- For each previous **IAudioInfluencer** that is no longer encountered, call the **RemoveEffect** method.

Note that AudioEmitter updates on human time scales, as opposed to on a per frame basis. This is done because humans generally do not move fast enough for the effect to need to be updated more frequently than every quarter or half of a second. Holograms that teleport rapidly from one location to another can break the illusion.

- In the **Hierarchy** panel, expand **HologramCollection**.
- Expand **EnergyHub** and select **BlobOutside**.
- In the **Inspector** panel, click **Add Component** and add **Audio Occluder**.
- In **Audio Occluder**, set **Cutoff Frequency** to **1500**.

This setting limits the AudioSource frequencies to 1500 Hz and below.

- Set **Volume Pass Through** to **0.9**.

This setting reduces the volume of the AudioSource to 90% of its current level.

Audio Occluder implements **IAudioInfluencer** to:

- Apply an occlusion effect using an **AudioLowPassFilter** which gets attached to the **AudioSource** managed by the **AudioEmitter**.
- Applies volume attenuation to the AudioSource.
- Disables the effect by setting a neutral cutoff frequency and disabling the filter.

The frequency used as neutral is 22 kHz (22000 Hz). This frequency was chosen due to it being above the nominal maximum frequency that can be heard by the human ear, this making no discernable impact to the sound.

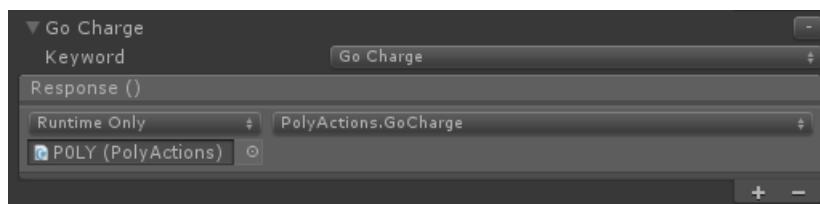
- In the **Hierarchy** panel, select **SpatialMapping**.
- In the **Inspector** panel, click **Add Component** and add **Audio Occluder**.
- In **Audio Occluder**, set **Cutoff Frequency** to **750**.

When multiple occluders are in the path between the user and the **AudioEmitter**, the lowest frequency is applied to the filter.

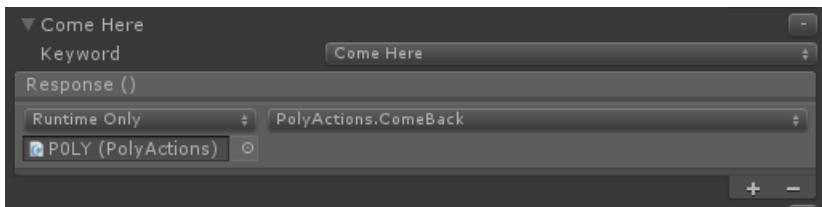
- Set **Volume Pass Through** to **0.75**.

When multiple occluders are in the path between the user and the **AudioEmitter**, the volume pass through is applied additively.

- In the **Hierarchy** panel, select **Managers**.
- In the **Inspector** panel, expand **Speech Input Handler**.
- In **Speech Input Handler**, expand **Go Charge**.
- Change **No Function** to **PolyActions.GoCharge**.



- Expand **Come Here**.
- Change **No Function** to **PolyActions.ComeBack**.



Build and Deploy

- As before, build the project in Unity and deploy in Visual Studio.

After the application is deployed:

- Say "Go Charge" to have POLY enter the Energy Hub.

Note the change in the sound. It should sound muffled and a little quieter. If you are able to position yourself with a wall or other object between you and the Energy Hub, you should notice a further muffling of the sound due to the occlusion by the real world.

- Say "Come Here" to have POLY leave the Energy Hub and position itself in front of you.

Note that the sound occlusion is removed once POLY exits the Energy Hub. If you are still hearing occlusion, POLY may be occluded by the real world. Try moving to ensure you have a clear line of sight to POLY.

Part 3 - Room Models

Key Concepts

- The size of the space provides subliminal queues that contribute to sound localization.
- Room models are set per- **AudioSource**.
- The [MixedRealityToolkit for Unity](#) provides code for setting the room model.
- For Mixed Reality experiences, select the room model that best fits the real world space.

If you are creating a Virtual Reality scenario, select the room model that best fits the virtual environment.

Chapter 4 - Sound Design

Objectives

- Understand considerations for effective sound design.
- Learn mixing techniques and guidelines.

Part 1 - Sound and Experience Design

This section discusses key sound and experience design considerations and guidelines.

Normalize all sounds

This avoids the need for special case code to adjust volume levels per sound, which can be time consuming and limits the ability to easily update sound files.

Design for an untethered experience

HoloLens is a fully contained, untethered holographic computer. Your users can and will use your experiences while moving. Be sure to test your audio mix by walking around.

Emit sound from logical locations on your holograms

In the real world, a dog does not bark from its tail and a human's voice does not come from his/her feet. Avoid having your sounds emit from unexpected portions of your holograms.

For small holograms, it is reasonable to have sound emit from the center of the geometry.

Familiar sounds are most localizable

The human voice and music are very easy to localize. If someone calls your name, you are able to very accurately determine from what direction the voice came and from how far away. Short, unfamiliar sounds are harder to

localize.

Be cognizant of user expectations

Life experience plays a part in our ability to identify the location of a sound. This is one reason why the human voice is particularly easy to localize. It is important to be aware of your user's learned expectations when placing your sounds.

For example, when someone hears a bird song they generally look up, as birds tend to be above the line of sight (flying or in a tree). It is not uncommon for a user to turn in the correct direction of a sound, but look in the wrong vertical direction and become confused or frustrated when they are unable to find the hologram.

Avoid hidden emitters

In the real world, if we hear a sound, we can generally identify the object that is emitting the sound. This should also hold true in your experiences. It can be very disconcerting for users to hear a sound, know from where the sound originates and be unable to see an object.

There are some exceptions to this guideline. For example, ambient sounds such as crickets in a field need not be visible. Life experience gives us familiarity with the source of these sounds without the need to see it.

Part 2 - Sound Mixing

Target your mix for 70% volume on the HoloLens

Mixed Reality experiences allow holograms to be seen in the real world. They should also allow real world sounds to be heard. A 70% volume target enables the user to hear the world around them along with the sound of your experience.

HoloLens at 100% volume should drown out external sounds

A volume level of 100% is akin to a Virtual Reality experience. Visually, the user is transported to a different world. The same should hold true audibly.

Use the Unity AudioMixer to adjust categories of sounds

When designing your mix, it is often helpful to create sound categories and have the ability to increase or decrease their volume as a unit. This retains the relative levels of each sound while enabling quick and easy changes to the overall mix. Common categories include; sound effects, ambience, voice overs and background music.

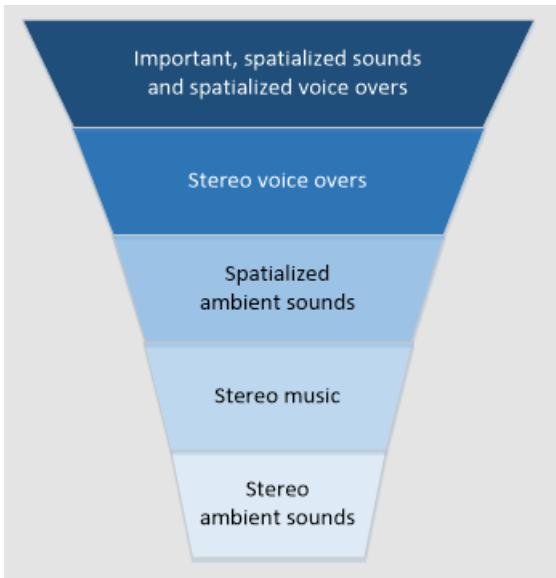
Mix sounds based on the user's gaze

It can often be useful to change the sound mix in your experience based on where a user is (or is not) looking. One common use for this technique are to reduce the volume level for holograms that are outside of the Holographic Frame to make it easier for the user to focus on the information in front of them. Another use is to increase the volume of a sound to draw the user's attention to an important event.

Building your mix

When building your mix, it is recommended to start with your experience's background audio and add layers based on importance. Often, this results in each layer being louder than the previous.

Imagining your mix as an inverted funnel, with the least important (and generally quietest sounds) at the bottom, it is recommended to structure your mix similar to the following diagram.



Voice overs are an interesting scenario. Based on the experience you are creating you may wish to have a stereo (not localized) sound or to spatialize your voice overs. Two Microsoft published experiences illustrate excellent examples of each scenario.

[HoloTour](#) uses a stereo voice over. When the narrator is describing the location being viewed, the sound is consistent and does not vary based on the user's position. This enables the narrator to describe the scene without taking away from the spatialized sounds of the environment.

[Fragments](#) utilizes a spatialized voice over in the form of a detective. The detective's voice is used to help bring the user's attention to an important clue as if an actual human was in the room. This enables an even greater sense of immersion into the experience of solving the mystery.

Part 3 -Performance

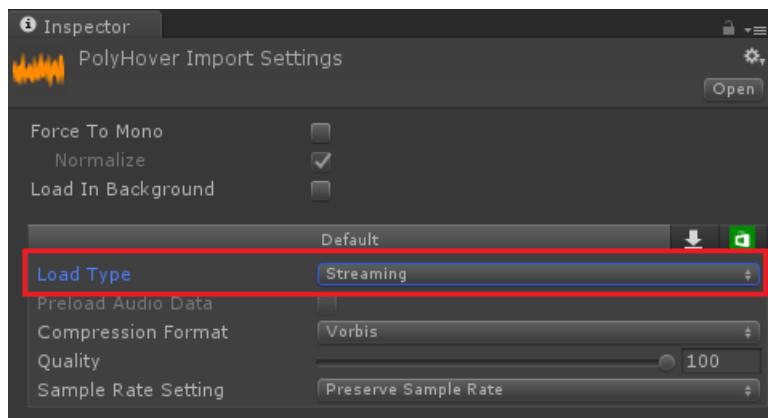
CPU usage

When using Spatial Sound, 10 - 12 emitters will consume approximately 12% of the CPU.

Stream long audio files

Audio data can be large, especially at common sample rates (44.1 and 48 kHz). A general rule is that audio files longer than 5 - 10 seconds should be streamed to reduce application memory usage.

In Unity, you can mark an audio file for streaming in the file's import settings.



Chapter 5 - Special Effects

Objectives

- Add depth to "Magic Windows".
- Bring the user into the virtual world.

Magic Windows

Key Concepts

- Creating views into a hidden world, is visually compelling.
- Enhance realism by adding audio effects when a hologram or the user is near the hidden world.

Instructions

- In the **Hierarchy** panel, expand **HologramCollection** and select **Underworld**.
- Expand **Underworld** and select **VoiceSource**.
- In the **Inspector** panel, click **Add Component** and add **User Voice Effect**.

An **AudioSource** component will be added to **VoiceSource**.

- In **AudioSource**, set **Output** to **UserVoice (Mixer)**.
- Check the **Spatialize** checkbox.
- Drag the **Spatial Blend** slider all the way to **3D**, or enter **1** in the edit box.
- Expand **3D Sound Settings**.
- Set **Doppler Level** to **0**.
- In **User Voice Effect**, set **Parent Object** to the **Underworld** from the scene.
- Set **Max Distance** to **1**.

Setting **Max Distance** tells **User Voice Effect** how close the user must be to the parent object before the effect is enabled.

- In **User Voice Effect**, expand **Chorus Parameters**.
- Set **Depth** to **0.1**.
- Set **Tap 1 Volume**, **Tap 2 Volume** and **Tap 3 Volume** to **0.8**.
- Set **Original Sound Volume** to **0.5**.

The previous settings configure the parameters of the Unity **AudioChorusFilter** used to add richness to the user's voice.

- In **User Voice Effect**, expand **Echo Parameters**.
- Set **Delay** to **300**
- Set **Decay Ratio** to **0.2**.
- Set **Original Sound Volume** to **0**.

The previous settings configure the parameters of the Unity **AudioEchoFilter** used to cause the user's voice to echo.

The User Voice Effect script is responsible for:

- Measuring the distance between the user and the **GameObject** to which the script is attached.
- Determining whether or not the user is facing the **GameObject**.

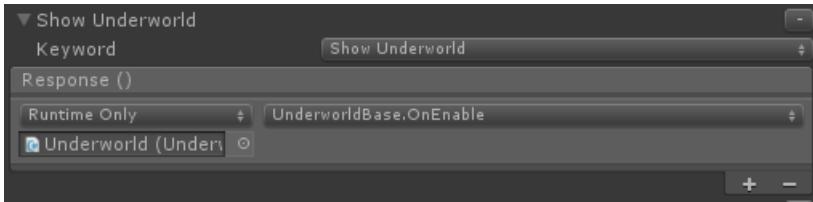
The user must be facing the **GameObject**, regardless of distance, for the effect to be enabled.

- Applying and configuring an **AudioChorusFilter** and an **AudioEchoFilter** to the **AudioSource**.
- Disabling the effect by disabling the filters.

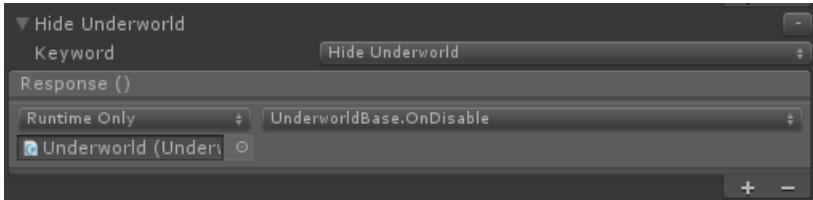
User Voice Effect uses the Mic Stream Selector component, from the [MixedRealityToolkit for Unity](#), to select the high quality voice stream and route it into Unity's audio system.

- In the **Hierarchy** panel, select **Managers**.
- In the **Inspector** panel, expand **Speech Input Handler**.
- In **Speech Input Handler**, expand **Show Underworld**.

- Change **No Function** to **UnderworldBase.OnEnable**.



- Expand **Hide Underworld**.
- Change **No Function** to **UnderworldBase.OnDisable**.



Build and Deploy

- As before, build the project in Unity and deploy in Visual Studio.

After the application is deployed:

- Face a surface (wall, floor, table) and say "*Show Underworld*".

The underworld will be shown and all other holograms will be hidden. If you do not see the underworld, ensure that you are facing a real-world surface.

- Approach within 1 meter of the underworld hologram and start talking.

There are now audio effects applied to your voice!

- Turn away from the underworld and notice how the effect is no longer applied.
- Say "*Hide Underworld*" to hide the underworld.

The underworld will be hidden and the previously hidden holograms will reappear.

The End

Congratulations! You have now completed **MR Spatial 220: Spatial sound**.

Listen to the world and bring your experiences to life with sound!

MR Spatial 230: Spatial mapping

11/6/2018 • 31 minutes to read • [Edit Online](#)

Spatial mapping combines the real world and virtual world together by teaching holograms about the environment. In MR Spatial 230 (Project Planetarium) we'll learn how to:

- Scan the environment and transfer data from the HoloLens to your development machine.
- Explore shaders and learn how to use them for visualizing your space.
- Break down the room mesh into simple planes using mesh processing.
- Go beyond the placement techniques we learned in [MR Basics 101](#), and provide feedback about where a hologram can be placed in the environment.
- Explore occlusion effects, so when your hologram is behind a real-world object, you can still see it with x-ray vision!

Device support

COURSE	HOOLENS	IMMERSIVE HEADSETS
MR Spatial 230: Spatial mapping	✓ <input type="checkbox"/>	

Before you start

Prerequisites

- A Windows 10 PC configured with the correct [tools installed](#).
- Some basic C# programming ability.
- You should have completed [MR Basics 101](#).
- A HoloLens device [configured for development](#).

Project files

- Download the [files](#) required by the project. Requires Unity 2017.2 or later.
 - If you still need Unity 5.6 support, please use [this release](#).
 - If you still need Unity 5.5 support, please use [this release](#).
 - If you still need Unity 5.4 support, please use [this release](#).
- Un-archive the files to your desktop or other easy to reach location.

NOTE

If you want to look through the source code before downloading, it's [available on GitHub](#).

Notes

- "Enable Just My Code" in Visual Studio needs to be disabled (*unchecked*) under Tools > Options > Debugging in order to hit breakpoints in your code.

Unity setup

- Start **Unity**.
- Select **New** to create a new project.
- Name the project **Planetarium**.
- Verify that the **3D** setting is selected.
- Click **Create Project**.
- Once Unity launches, go to **Edit > Project Settings > Player**.
- In the **Inspector** panel, find and select the green **Windows Store** icon.
- Expand **Other Settings**.
- In the **Rendering** section, check the **Virtual Reality Supported** option.
- Verify that **Windows Holographic** appears in the list of **Virtual Reality SDKs**. If not, select the + button at the bottom of the list and choose **Windows Holographic**.
- Expand **Publishing Settings**.
- In the **Capabilities** section, check the following settings:
 - InternetClientServer
 - PrivateNetworkClientServer
 - Microphone
 - SpatialPerception
- Go to **Edit > Project Settings > Quality**
- In the **Inspector** panel, under the Windows Store icon, select the black drop-down arrow under the 'Default' row and change the default setting to **Fastest**.
- Go to **Assets > Import Package > Custom Package**.
- Navigate to the ...\\HolographicAcademy-Holograms-230-SpatialMapping\\Starting folder.
- Click on **Planetarium.unitypackage**.
- Click **Open**.
- An **Import Unity Package** window should appear, click on the **Import** button.
- Wait for Unity to import all of the assets that we will need to complete this project.
- In the **Hierarchy** panel, delete the **Main Camera**.
- In the **Project** panel, **HoloToolkit-SpatialMapping-230\Utilities\Prefabs** folder, find the **Main Camera** object.
- Drag and drop the **Main Camera** prefab into the **Hierarchy** panel.
- In the **Hierarchy** panel, delete the **Directional Light** object.
- In the **Project** panel, **Holograms** folder, locate the **Cursor** object.
- Drag & drop the **Cursor** prefab into the **Hierarchy**.
- In the **Hierarchy** panel, select the **Cursor** object.
- In the **Inspector** panel, click the **Layer** drop-down and select **Edit Layers....**
- Name **User Layer 31** as "**SpatialMapping**".
- Save the new scene: **File > Save Scene As...**
- Click **New Folder** and name the folder **Scenes**.
- Name the file "**Planetarium**" and save it in the **Scenes** folder.

Chapter 1 - Scanning

Objectives

- Learn about the SurfaceObserver and how its settings impact experience and performance.
- Create a room scanning experience to collect the meshes of your room.

Instructions

- In the **Project** panel **HoloToolkit-SpatialMapping-230\SpatialMapping\Prefabs** folder, find the **SpatialMapping** prefab.
- Drag & drop the **SpatialMapping** prefab into the **Hierarchy** panel.

Build and Deploy (part 1)

- In Unity, select **File > Build Settings**.
- Click **Add Open Scenes** to add the **Planetarium** scene to the build.
- Select **Universal Windows Platform** in the **Platform** list and click **Switch Platform**.
- Set **SDK** to **Universal 10** and **UWP Build Type** to **D3D**.
- Check **Unity C# Projects**.
- Click **Build**.
- Create a **New Folder** named "App".
- Single click the **App** folder.
- Press the **Select Folder** button.
- When Unity is done building, a File Explorer window will appear.
- Double-click on the **App** folder to open it.
- Double-click on **Planetarium.sln** to load the project in Visual Studio.
- In Visual Studio, use the top toolbar to change the Configuration to **Release**.
- Change the Platform to **x86**.
- Click on the drop-down arrow to the right of 'Local Machine', and select **Remote Machine**.
- Enter [your device's IP address](#) in the Address field and change Authentication Mode to **Universal (Unencrypted Protocol)**.
- Click **Debug -> Start Without debugging** or press **Ctrl + F5**.
- Watch the **Output** panel in Visual Studio for build and deploy status.
- Once your app has deployed, walk around the room. You will see the surrounding surfaces covered by black and white wireframe meshes.
- Scan your surroundings. Be sure to look at walls, ceilings, and floors.

Build and Deploy (part 2)

Now let's explore how Spatial Mapping can affect performance.

- In Unity, select **Window > Profiler**.
- Click **Add Profiler > GPU**.
- Click **Active Profiler >**.
- Enter the **IP address** of your HoloLens.
- Click **Connect**.
- Observe the number of milliseconds it takes for the GPU to render a frame.
- Stop the application from running on the device.
- Return to Visual Studio and open **SpatialMappingObserver.cs**. You will find it in the **HoloToolkit\SpatialMapping** folder of the Assembly-CSharp (Universal Windows) project.
- Find the **Awake()** function, and add the following line of code: **TrianglesPerCubicMeter = 1200;**
- Re-deploy the project to your device, and then **reconnect the profiler**. Observe the change in the number of milliseconds to render a frame.
- Stop the application from running on the device.

Save and load in Unity

Finally, let's save our room mesh and load it into Unity.

- Return to Visual Studio and remove the **TrianglesPerCubicMeter** line that you added in the **Awake()** function during the previous section.
- Redeploy the project to your device. We should now be running with **500** triangles per cubic meter.
- Open a browser and enter in your HoloLens IPAddress to navigate to the **Windows Device Portal**.
- Select the **3D View** option in the left panel.
- Under **Surface reconstruction** select the **Update** button.
- Watch as the areas that you have scanned on your HoloLens appear in the display window.
- To save your room scan, press the **Save** button.
- Open your **Downloads** folder to find the saved room model **SRMesh.obj**.
- Copy **SRMesh.obj** to the **Assets** folder of your Unity project.
- In Unity, select the **SpatialMapping** object in the **Hierarchy** panel.
- Locate the **Object Surface Observer (Script)** component.
- Click the circle to the right of the **Room Model** property.
- Find and select the **SRMesh** object and then close the window.
- Verify that the **Room Model** property in the **Inspector** panel is now set to **SRMesh**.
- Press the **Play** button to enter Unity's preview mode.
- The SpatialMapping component will load the meshes from the saved room model so you can use them in Unity.
- Switch to **Scene** view to see all of your room model displayed with the wireframe shader.
- Press the **Play** button again to exit preview mode.

NOTE: The next time that you enter preview mode in Unity, it will load the saved room mesh by default.

Chapter 2 - Visualization

Objectives

- Learn the basics of shaders.
- Visualize your surroundings.

Instructions

- In Unity's **Hierarchy** panel, select the **SpatialMapping** object.
- In the **Inspector** panel, find the **Spatial Mapping Manager (Script)** component.
- Click the circle to the right of the **Surface Material** property.
- Find and select the **BlueLinesOnWalls** material and close the window.
- In the **Project** panel **Shaders** folder, double-click on **BlueLinesOnWalls** to open the shader in Visual Studio.
- This is a simple pixel (vertex to fragment) shader, which accomplishes the following tasks:
 1. Converts a vertex's location to world space.
 2. Checks the vertex's normal to determine if a pixel is vertical.
 3. Sets the color of the pixel for rendering.

Build and Deploy

- Return to Unity and press **Play** to enter preview mode.
- Blue lines will be rendered on all vertical surfaces of the room mesh (which automatically loaded from our saved scanning data).
- Switch to the **Scene** tab to adjust your view of the room and see how the entire room mesh appears in Unity.
- In the **Project** panel, find the **Materials** folder and select the **BlueLinesOnWalls** material.
- Modify some properties and see how the changes appear in the Unity editor.

- In the **Inspector** panel, adjust the **LineScale** value to make the lines appear thicker or thinner.
- In the **Inspector** panel, adjust the **LinesPerMeter** value to change how many lines appear on each wall.
- Click **Play** again to exit preview mode.
- Build and deploy to the HoloLens and observe how the shader rendering appears on real surfaces.

Unity does a great job of previewing materials, but it's always a good idea to check-out rendering in the device.

Chapter 3 - Processing

Objectives

- Learn techniques to process spatial mapping data for use in your application.
- Analyze spatial mapping data to find planes and remove triangles.
- Use planes for hologram placement.

Instructions

- In Unity's **Project** panel, **Holograms** folder, find the **SpatialProcessing** object.
- Drag & drop the **SpatialProcessing** object into the **Hierarchy** panel.

The SpatialProcessing prefab includes components for processing the spatial mapping data.

SurfaceMeshesToPlanes.cs will find and generate planes based on the spatial mapping data. We will use planes in our application to represent walls, floors and ceilings. This prefab also includes **RemoveSurfaceVertices.cs** which can remove vertices from the spatial mapping mesh. This can be used to create holes in the mesh, or to remove excess triangles that are no longer needed (because planes can be used instead).

- In Unity's **Project** panel, **Holograms** folder, find the **SpaceCollection** object.
- Drag and drop the **SpaceCollection** object into the **Hierarchy** panel.
- In the **Hierarchy** panel, select the **SpatialProcessing** object.
- In the **Inspector** panel, find the **Play Space Manager (Script)** component.
- Double-click on **PlaySpaceManager.cs** to open it in Visual Studio.

PlaySpaceManager.cs contains application-specific code. We will add functionality to this script to enable the following behavior:

1. Stop collecting spatial mapping data after we exceed the scanning time limit (10 seconds).
2. Process the spatial mapping data:
 - a. Use SurfaceMeshesToPlanes to create a simpler representation of the world as planes (walls, floors, ceilings, etc).
 - b. Use RemoveSurfaceVertices to remove surface triangles that fall within plane boundaries.
3. Generate a collection of holograms in the world and place them on wall and floor planes near the user.

Complete the coding exercises marked in PlaySpaceManager.cs, or replace the script with the finished solution from below:

```
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Windows.Speech;
using Academy.HoloToolkit.Unity;

/// <summary>
/// The SurfaceManager class allows applications to scan the environment for a specified amount of time
/// and then process the Spatial Mapping Mesh (find planes, remove vertices) after that time has expired.
/// </summary>
public class PlaySpaceManager : Singleton<PlaySpaceManager>
{
```

```

    {
        [Tooltip("When checked, the SurfaceObserver will stop running after a specified amount of time.")]
        public bool limitScanningByTime = true;

        [Tooltip("How much time (in seconds) that the SurfaceObserver will run after being started; used when
        'Limit Scanning By Time' is checked.")]
        public float scanTime = 30.0f;

        [Tooltip("Material to use when rendering Spatial Mapping meshes while the observer is running.")]
        public Material defaultMaterial;

        [Tooltip("Optional Material to use when rendering Spatial Mapping meshes after the observer has been
        stopped.")]
        public Material secondaryMaterial;

        [Tooltip("Minimum number of floor planes required in order to exit scanning/processing mode.")]
        public uint minimumFloors = 1;

        [Tooltip("Minimum number of wall planes required in order to exit scanning/processing mode.")]
        public uint minimumWalls = 1;

        /// <summary>
        /// Indicates if processing of the surface meshes is complete.
        /// </summary>
        private bool meshesProcessed = false;

        /// <summary>
        /// GameObject initialization.
        /// </summary>
        private void Start()
        {
            // Update surfaceObserver and storedMeshes to use the same material during scanning.
            SpatialMappingManager.Instance.SetSurfaceMaterial(defaultMaterial);

            // Register for the MakePlanesComplete event.
            SurfaceMeshesToPlanes.Instance.MakePlanesComplete += SurfaceMeshesToPlanes_MakePlanesComplete;
        }

        /// <summary>
        /// Called once per frame.
        /// </summary>
        private void Update()
        {
            // Check to see if the spatial mapping data has been processed
            // and if we are limiting how much time the user can spend scanning.
            if (!meshesProcessed && limitScanningByTime)
            {
                // If we have not processed the spatial mapping data
                // and scanning time is limited...

                // Check to see if enough scanning time has passed
                // since starting the observer.
                if (limitScanningByTime && ((Time.time - SpatialMappingManager.Instance.StartTime) < scanTime))
                {
                    // If we have a limited scanning time, then we should wait until
                    // enough time has passed before processing the mesh.
                }
                else
                {
                    // The user should be done scanning their environment,
                    // so start processing the spatial mapping data...

                    /* TODO: 3.a DEVELOPER CODING EXERCISE 3.a */

                    // 3.a: Check if IsObserverRunning() is true on the
                    // SpatialMappingManager.Instance.
                    if(SpatialMappingManager.Instance.IsObserverRunning())
                    {
                        // 3.a: If running, Stop the observer by calling

```

```

        // StopObserver() on the SpatialMappingManager.Instance.
        SpatialMappingManager.Instance.StopObserver();
    }

    // 3.a: Call CreatePlanes() to generate planes.
    CreatePlanes();

    // 3.a: Set meshesProcessed to true.
    meshesProcessed = true;
}
}

/// <summary>
/// Handler for the SurfaceMeshesToPlanes MakePlanesComplete event.
/// </summary>
/// <param name="source">Source of the event.</param>
/// <param name="args">Args for the event.</param>
private void SurfaceMeshesToPlanes_MakePlanesComplete(object source, System.EventArgs args)
{
    /* TODO: 3.a DEVELOPER CODING EXERCISE 3.a */

    // Collection of floor and table planes that we can use to set horizontal items on.
    List<GameObject> horizontal = new List<GameObject>();

    // Collection of wall planes that we can use to set vertical items on.
    List<GameObject> vertical = new List<GameObject>();

    // 3.a: Get all floor and table planes by calling
    // SurfaceMeshesToPlanes.Instance.GetActivePlanes().
    // Assign the result to the 'horizontal' list.
    horizontal = SurfaceMeshesToPlanes.Instance.GetActivePlanes(PlaneTypes.Table | PlaneTypes.Floor);

    // 3.a: Get all wall planes by calling
    // SurfaceMeshesToPlanes.Instance.GetActivePlanes().
    // Assign the result to the 'vertical' list.
    vertical = SurfaceMeshesToPlanes.Instance.GetActivePlanes(PlaneTypes.Wall);

    // Check to see if we have enough horizontal planes (minimumFloors)
    // and vertical planes (minimumWalls), to set holograms on in the world.
    if (horizontal.Count >= minimumFloors && vertical.Count >= minimumWalls)
    {
        // We have enough floors and walls to place our holograms on...

        // 3.a: Let's reduce our triangle count by removing triangles
        // from SpatialMapping meshes that intersect with our active planes.
        // Call RemoveVertices().
        // Pass in all activePlanes found by SurfaceMeshesToPlanes.Instance.
        RemoveVertices(SurfaceMeshesToPlanes.Instance.ActivePlanes);

        // 3.a: We can indicate to the user that scanning is over by
        // changing the material applied to the Spatial Mapping meshes.
        // Call SpatialMappingManager.Instance.SetSurfaceMaterial().
        // Pass in the secondaryMaterial.
        SpatialMappingManager.Instance.SetSurfaceMaterial(secondaryMaterial);

        // 3.a: We are all done processing the mesh, so we can now
        // initialize a collection of Placeable holograms in the world
        // and use horizontal/vertical planes to set their starting positions.
        // Call SpaceCollectionManager.Instance.GenerateItemsInWorld().
        // Pass in the lists of horizontal and vertical planes that we found earlier.
        SpaceCollectionManager.Instance.GenerateItemsInWorld(horizontal, vertical);
    }
    else
    {
        // We do not have enough floors/walls to place our holograms on...

        // 3.a: Re-enter scanning mode so the user can find more surfaces by
        // calling StartObserver() on the SpatialMappingManager.Instance.
    }
}

```

```

        SpatialMappingManager.Instance.StartObserver();

        // 3.a: Re-process spatial data after scanning completes by
        // re-setting meshesProcessed to false.
        meshesProcessed = false;
    }
}

/// <summary>
/// Creates planes from the spatial mapping surfaces.
/// </summary>
private void CreatePlanes()
{
    // Generate planes based on the spatial map.
    SurfaceMeshesToPlanes surfaceToPlanes = SurfaceMeshesToPlanes.Instance;
    if (surfaceToPlanes != null && surfaceToPlanes.enabled)
    {
        surfaceToPlanes.MakePlanes();
    }
}

/// <summary>
/// Removes triangles from the spatial mapping surfaces.
/// </summary>
/// <param name="boundingObjects"></param>
private void RemoveVertices(IEnumerable<GameObject> boundingObjects)
{
    RemoveSurfaceVertices removeVerts = RemoveSurfaceVertices.Instance;
    if (removeVerts != null && removeVerts.enabled)
    {
        removeVerts.RemoveSurfaceVerticesWithinBounds(boundingObjects);
    }
}

/// <summary>
/// Called when the GameObject is unloaded.
/// </summary>
private void OnDestroy()
{
    if (SurfaceMeshesToPlanes.Instance != null)
    {
        SurfaceMeshesToPlanes.Instance.MakePlanesComplete -= SurfaceMeshesToPlanes_MakePlanesComplete;
    }
}
}

```

Build and Deploy

- Before deploying to the HoloLens, press the **Play** button in Unity to enter play mode.
- After the room mesh is loaded from file, wait for 10 seconds before processing starts on the spatial mapping mesh.
- When processing is complete, planes will appear to represent the floor, walls, ceiling, etc.
- After all of the planes have been found, you should see a solar system appear on a table of floor near the camera.
- Two posters should appear on walls near the camera too. Switch to the **Scene** tab if you cannot see them in **Game** mode.
- Press the **Play** button again to exit play mode.
- Build and deploy to the HoloLens, as usual.
- Wait for scanning and processing of the spatial mapping data to complete.
- Once you see planes, try to find the solar system and posters in your world.

Chapter 4 - Placement

Objectives

- Determine if a hologram will fit on a surface.
- Provide feedback to the user when a hologram can/cannot fit on a surface.

Instructions

- In Unity's **Hierarchy** panel, select the **SpatialProcessing** object.
- In the **Inspector** panel, find the **Surface Meshes To Planes (Script)** component.
- Change the **Draw Planes** property to **Nothing** to clear the selection.
- Change the **Draw Planes** property to **Wall**, so that only wall planes will be rendered.
- In the **Project** panel, **Scripts** folder, double-click on **Placeable.cs** to open it in Visual Studio.

The **Placeable** script is already attached to the posters and projection box that are created after plane finding completes. All we need to do is uncomment some code, and this script will achieve the following:

1. Determine if a hologram will fit on a surface by raycasting from the center and four corners of the bounding cube.
2. Check the surface normal to determine if it is smooth enough for the hologram to sit flush on.
3. Render a bounding cube around the hologram to show its actual size while being placed.
4. Cast a shadow under/behind the hologram to show where it will be placed on the floor/wall.
5. Render the shadow as red, if the hologram cannot be placed on the surface, or green, if it can.
6. Re-orient the hologram to align with the surface type (vertical or horizontal) that it has affinity to.
7. Smoothly place the hologram on the selected surface to avoid jumping or snapping behavior.

Uncomment all code in the coding exercise below, or use this completed solution in **Placeable.cs**:

```
using System.Collections.Generic;
using UnityEngine;
using Academy.HoloToolkit.Unity;

/// <summary>
/// Enumeration containing the surfaces on which a GameObject
/// can be placed. For simplicity of this sample, only one
/// surface type is allowed to be selected.
/// </summary>
public enum PlacementSurfaces
{
    // Horizontal surface with an upward pointing normal.
    Horizontal = 1,

    // Vertical surface with a normal facing the user.
    Vertical = 2,
}

/// <summary>
/// The Placeable class implements the logic used to determine if a GameObject
/// can be placed on a target surface. Constraints for placement include:
/// * No part of the GameObject's box collider impacts with another object in the scene
/// * The object lays flat (within specified tolerances) against the surface
/// * The object would not fall off of the surface if gravity were enabled.
/// This class also provides the following visualizations.
/// * A transparent cube representing the object's box collider.
/// * Shadow on the target surface indicating whether or not placement is valid.
/// </summary>
public class Placeable : MonoBehaviour
{
    [Tooltip("The base material used to render the bounds asset when placement is allowed.")]
    public Material PlaceableBoundsMaterial = null;
```

```

[Tooltip("The base material used to render the bounds asset when placement is not allowed.")]
public Material NotPlaceableBoundsMaterial = null;

[Tooltip("The material used to render the placement shadow when placement it allowed.")]
public Material PlaceableShadowMaterial = null;

[Tooltip("The material used to render the placement shadow when placement it not allowed.")]
public Material NotPlaceableShadowMaterial = null;

[Tooltip("The type of surface on which the object can be placed.")]
public PlacementSurfaces PlacementSurface = PlacementSurfaces.Horizontal;

[Tooltip("The child object(s) to hide during placement.")]
public List<GameObject> ChildrenToHide = new List<GameObject>();

/// <summary>
/// Indicates if the object is in the process of being placed.
/// </summary>
public bool IsPlacing { get; private set; }

// The most recent distance to the surface. This is used to
// locate the object when the user's gaze does not intersect
// with the Spatial Mapping mesh.
private float lastDistance = 2.0f;

// The distance away from the target surface that the object should hover prior while being placed.
private float hoverDistance = 0.15f;

// Threshold (the closer to 0, the stricter the standard) used to determine if a surface is flat.
private float distanceThreshold = 0.02f;

// Threshold (the closer to 1, the stricter the standard) used to determine if a surface is vertical.
private float upNormalThreshold = 0.9f;

// Maximum distance, from the object, that placement is allowed.
// This is used when raycasting to see if the object is near a placeable surface.
private float maximumPlacementDistance = 5.0f;

// Speed (1.0 being fastest) at which the object settles to the surface upon placement.
private float placementVelocity = 0.06f;

// Indicates whether or not this script manages the object's box collider.
private bool managingBoxCollider = false;

// The box collider used to determine of the object will fit in the desired location.
// It is also used to size the bounding cube.
private BoxCollider boxCollider = null;

// Visible asset used to show the dimensions of the object. This asset is sized
// using the box collider's bounds.
private GameObject boundsAsset = null;

// Visible asset used to show the where the object is attempting to be placed.
// This asset is sized using the box collider's bounds.
private GameObject shadowAsset = null;

// The location at which the object will be placed.
private Vector3 targetPosition;

/// <summary>
/// Called when the GameObject is created.
/// </summary>
private void Awake()
{
    targetPosition = gameObject.transform.position;

    // Get the object's collider.
    boxCollider = gameObject.GetComponent<BoxCollider>();
    if (boxCollider == null)

```

```

    // (boxCollider == null)
    {
        // The object does not have a collider, create one and remember that
        // we are managing it.
        managingBoxCollider = true;
        boxCollider = gameObject.AddComponent<BoxCollider>();
        boxCollider.enabled = false;
    }

    // Create the object that will be used to indicate the bounds of the GameObject.
    boundsAsset = GameObject.CreatePrimitive(PrimitiveType.Cube);
    boundsAsset.transform.parent = gameObject.transform;
    boundsAsset.SetActive(false);

    // Create a object that will be used as a shadow.
    shadowAsset = GameObject.CreatePrimitive(PrimitiveType.Quad);
    shadowAsset.transform.parent = gameObject.transform;
    shadowAsset.SetActive(false);
}

/// <summary>
/// Called when our object is selected. Generally called by
/// a gesture management component.
/// </summary>
public void OnSelect()
{
    /* TODO: 4.a CODE ALONG 4.a */

    if (!IsPlacing)
    {
        OnPlacementStart();
    }
    else
    {
        OnPlacementStop();
    }
}

/// <summary>
/// Called once per frame.
/// </summary>
private void Update()
{
    /* TODO: 4.a CODE ALONG 4.a */

    if (IsPlacing)
    {
        // Move the object.
        Move();

        // Set the visual elements.
        Vector3 targetPosition;
        Vector3 surfaceNormal;
        bool canBePlaced = ValidatePlacement(out targetPosition, out surfaceNormal);
        DisplayBounds(canBePlaced);
        DisplayShadow(targetPosition, surfaceNormal, canBePlaced);
    }
    else
    {
        // Disable the visual elements.
        boundsAsset.SetActive(false);
        shadowAsset.SetActive(false);

        // Gracefully place the object on the target surface.
        float dist = (gameObject.transform.position - targetPosition).magnitude;
        if (dist > 0)
        {
            gameObject.transform.position = Vector3.Lerp(gameObject.transform.position, targetPosition,
placementVelocity / dist);
        }
    }
}

```

```

        }

        else
        {
            // Unhide the child object(s) to make placement easier.
            for (int i = 0; i < ChildrenToHide.Count; i++)
            {
                ChildrenToHide[i].SetActive(true);
            }
        }
    }

}

/// <summary>
/// Verify whether or not the object can be placed.
/// </summary>
/// <param name="position">
/// The target position on the surface.
/// </param>
/// <param name="surfaceNormal">
/// The normal of the surface on which the object is to be placed.
/// </param>
/// <returns>
/// True if the target position is valid for placing the object, otherwise false.
/// </returns>
private bool ValidatePlacement(out Vector3 position, out Vector3 surfaceNormal)
{
    Vector3 raycastDirection = gameObject.transform.forward;

    if (PlacementSurface == PlacementSurfaces.Horizontal)
    {
        // Placing on horizontal surfaces.
        // Raycast from the bottom face of the box collider.
        raycastDirection = -(Vector3.up);
    }

    // Initialize out parameters.
    position = Vector3.zero;
    surfaceNormal = Vector3.zero;

    Vector3[] facePoints = GetColliderFacePoints();

    // The origin points we receive are in local space and we
    // need to raycast in world space.
    for (int i = 0; i < facePoints.Length; i++)
    {
        facePoints[i] = gameObject.transform.TransformVector(facePoints[i]) +
gameObject.transform.position;
    }

    // Cast a ray from the center of the box collider face to the surface.
    RaycastHit centerHit;
    if (!Physics.Raycast(facePoints[0],
                        raycastDirection,
                        out centerHit,
                        maximumPlacementDistance,
                        SpatialMappingManager.Instance.LayerMask))
    {
        // If the ray failed to hit the surface, we are done.
        return false;
    }

    // We have found a surface. Set position and surfaceNormal.
    position = centerHit.point;
    surfaceNormal = centerHit.normal;

    // Cast a ray from the corners of the box collider face to the surface.
    for (int i = 1; i < facePoints.Length; i++)
    {
        RaycastHit hitInfo;
    }
}

```

```

        if (Physics.Raycast(facePoints[i],
                            raycastDirection,
                            out hitInfo,
                            maximumPlacementDistance,
                            SpatialMappingManager.Instance.LayerMask))
    {
        // To be a valid placement location, each of the corners must have a similar
        // enough distance to the surface as the center point
        if (!IsEquivalentDistance(centerHit.distance, hitInfo.distance))
        {
            return false;
        }
    }
    else
    {
        // The raycast failed to intersect with the target layer.
        return false;
    }
}

return true;
}

/// <summary>
/// Determine the coordinates, in local space, of the box collider face that
/// will be placed against the target surface.
/// </summary>
/// <returns>
/// Vector3 array with the center point of the face at index 0.
/// </returns>
private Vector3[] GetColliderFacePoints()
{
    // Get the collider extents.
    // The size values are twice the extents.
    Vector3 extents = boxCollider.size / 2;

    // Calculate the min and max values for each coordinate.
    float minX = boxCollider.center.x - extents.x;
    float maxX = boxCollider.center.x + extents.x;
    float minY = boxCollider.center.y - extents.y;
    float maxY = boxCollider.center.y + extents.y;
    float minZ = boxCollider.center.z - extents.z;
    float maxZ = boxCollider.center.z + extents.z;

    Vector3 center;
    Vector3 corner0;
    Vector3 corner1;
    Vector3 corner2;
    Vector3 corner3;

    if (PlacementSurface == PlacementSurfaces.Horizontal)
    {
        // Placing on horizontal surfaces.
        center = new Vector3(boxCollider.center.x, minY, boxCollider.center.z);
        corner0 = new Vector3(minX, minY, minZ);
        corner1 = new Vector3(minX, minY, maxZ);
        corner2 = new Vector3(maxX, minY, minZ);
        corner3 = new Vector3(maxX, minY, maxZ);
    }
    else
    {
        // Placing on vertical surfaces.
        center = new Vector3(boxCollider.center.x, boxCollider.center.y, maxZ);
        corner0 = new Vector3(minX, minY, maxZ);
        corner1 = new Vector3(minX, maxY, maxZ);
        corner2 = new Vector3(maxX, minY, maxZ);
        corner3 = new Vector3(maxX, maxY, maxZ);
    }
}

```

```

        return new Vector3[] { center, corner0, corner1, corner2, corner3 };
    }

/// <summary>
/// Put the object into placement mode.
/// </summary>
public void OnPlacementStart()
{
    // If we are managing the collider, enable it.
    if (managingBoxCollider)
    {
        boxCollider.enabled = true;
    }

    // Hide the child object(s) to make placement easier.
    for (int i = 0; i < ChildrenToHide.Count; i++)
    {
        ChildrenToHide[i].SetActive(false);
    }

    // Tell the gesture manager that it is to assume
    // all input is to be given to this object.
    GestureManager.Instance.OverrideFocusedObject = gameObject;

    // Enter placement mode.
    IsPlacing = true;
}

/// <summary>
/// Take the object out of placement mode.
/// </summary>
/// <remarks>
/// This method will leave the object in placement mode if called while
/// the object is in an invalid location. To determine whether or not
/// the object has been placed, check the value of the IsPlacing property.
/// </remarks>
public void OnPlacementStop()
{
    // ValidatePlacement requires a normal as an out parameter.
    Vector3 position;
    Vector3 surfaceNormal;

    // Check to see if we can exit placement mode.
    if (!ValidatePlacement(out position, out surfaceNormal))
    {
        return;
    }

    // The object is allowed to be placed.
    // We are placing at a small buffer away from the surface.
    targetPosition = position + (0.01f * surfaceNormal);

    OrientObject(true, surfaceNormal);

    // If we are managing the collider, disable it.
    if (managingBoxCollider)
    {
        boxCollider.enabled = false;
    }

    // Tell the gesture manager that it is to resume
    // its normal behavior.
    GestureManager.Instance.OverrideFocusedObject = null;

    // Exit placement mode.
    IsPlacing = false;
}

/// <summary>

```

```

/// Positions the object along the surface toward which the user is gazing.
/// </summary>
/// <remarks>
/// If the user's gaze does not intersect with a surface, the object
/// will remain at the most recently calculated distance.
/// </remarks>
private void Move()
{
    Vector3 moveTo = gameObject.transform.position;
    Vector3 surfaceNormal = Vector3.zero;
    RaycastHit hitInfo;

    bool hit = Physics.Raycast(Camera.main.transform.position,
                               Camera.main.transform.forward,
                               out hitInfo,
                               20f,
                               SpatialMappingManager.Instance.LayerMask);

    if (hit)
    {
        float offsetDistance = hoverDistance;

        // Place the object a small distance away from the surface while keeping
        // the object from going behind the user.
        if (hitInfo.distance <= hoverDistance)
        {
            offsetDistance = 0f;
        }

        moveTo = hitInfo.point + (offsetDistance * hitInfo.normal);

        lastDistance = hitInfo.distance;
        surfaceNormal = hitInfo.normal;
    }
    else
    {
        // The raycast failed to hit a surface. In this case, keep the object at the distance of the last
        // intersected surface.
        moveTo = Camera.main.transform.position + (Camera.main.transform.forward * lastDistance);
    }

    // Follow the user's gaze.
    float dist = Mathf.Abs((gameObject.transform.position - moveTo).magnitude);
    gameObject.transform.position = Vector3.Lerp(gameObject.transform.position, moveTo, placementVelocity
/ dist);

    // Orient the object.
    // We are using the return value from Physics.Raycast to instruct
    // the OrientObject function to align to the vertical surface if appropriate.
    OrientObject(hit, surfaceNormal);
}

/// <summary>
/// Orients the object so that it faces the user.
/// </summary>
/// <param name="alignToVerticalSurface">
/// If true and the object is to be placed on a vertical surface,
/// orient parallel to the target surface. If false, orient the object
/// to face the user.
/// </param>
/// <param name="surfaceNormal">
/// The target surface's normal vector.
/// </param>
/// <remarks>
/// The alignToVerticalSurface parameter is ignored if the object
/// is to be placed on a horizontalSurface
/// </remarks>
private void OrientObject(bool alignToVerticalSurface, Vector3 surfaceNormal)
{

```

```

Quaternion rotation = Camera.main.transform.localRotation;

// If the user's gaze does not intersect with the Spatial Mapping mesh,
// orient the object towards the user.
if (alignToVerticalSurface && (PlacementSurface == PlacementSurfaces.Vertical))
{
    // We are placing on a vertical surface.
    // If the normal of the Spatial Mapping mesh indicates that the
    // surface is vertical, orient parallel to the surface.
    if (Mathf.Abs(surfaceNormal.y) <= (1 - upNormalThreshold))
    {
        rotation = Quaternion.LookRotation(-surfaceNormal, Vector3.up);
    }
}
else
{
    rotation.x = 0f;
    rotation.z = 0f;
}

gameObject.transform.rotation = rotation;
}

/// <summary>
/// Displays the bounds asset.
/// </summary>
/// <param name="canBePlaced">
/// Specifies if the object is in a valid placement location.
/// </param>
private void DisplayBounds(bool canBePlaced)
{
    // Ensure the bounds asset is sized and positioned correctly.
    boundsAsset.transform.localPosition = boxCollider.center;
    boundsAsset.transform.localScale = boxCollider.size;
    boundsAsset.transform.rotation = gameObject.transform.rotation;

    // Apply the appropriate material.
    if (canBePlaced)
    {
        boundsAsset.GetComponent<Renderer>().sharedMaterial = PlaceableBoundsMaterial;
    }
    else
    {
        boundsAsset.GetComponent<Renderer>().sharedMaterial = NotPlaceableBoundsMaterial;
    }

    // Show the bounds asset.
    boundsAsset.SetActive(true);
}

/// <summary>
/// Displays the placement shadow asset.
/// </summary>
/// <param name="position">
/// The position at which to place the shadow asset.
/// </param>
/// <param name="surfaceNormal">
/// The normal of the surface on which the asset will be placed
/// </param>
/// <param name="canBePlaced">
/// Specifies if the object is in a valid placement location.
/// </param>
private void DisplayShadow(Vector3 position,
                           Vector3 surfaceNormal,
                           bool canBePlaced)
{
    // Rotate and scale the shadow so that it is displayed on the correct surface and matches the object.
    float rotationX = 0.0f;
}

```

```

        if (PlacementSurface == PlacementSurfaces.Horizontal)
        {
            rotationX = 90.0f;
            shadowAsset.transform.localScale = new Vector3(boxCollider.size.x, boxCollider.size.z, 1);
        }
        else
        {
            shadowAsset.transform.localScale = boxCollider.size;
        }

        Quaternion rotation = Quaternion.Euler(rotationX, gameObject.transform.rotation.eulerAngles.y, 0);
        shadowAsset.transform.rotation = rotation;

        // Apply the appropriate material.
        if (canBePlaced)
        {
            shadowAsset.GetComponent<Renderer>().sharedMaterial = PlaceableShadowMaterial;
        }
        else
        {
            shadowAsset.GetComponent<Renderer>().sharedMaterial = NotPlaceableShadowMaterial;
        }

        // Show the shadow asset as appropriate.
        if (position != Vector3.zero)
        {
            // Position the shadow a small distance from the target surface, along the normal.
            shadowAsset.transform.position = position + (0.01f * surfaceNormal);
            shadowAsset.SetActive(true);
        }
        else
        {
            shadowAsset.SetActive(false);
        }
    }

    /// <summary>
    /// Determines if two distance values should be considered equivalent.
    /// </summary>
    /// <param name="d1">
    /// Distance to compare.
    /// </param>
    /// <param name="d2">
    /// Distance to compare.
    /// </param>
    /// <returns>
    /// True if the distances are within the desired tolerance, otherwise false.
    /// </returns>
    private bool IsEquivalentDistance(float d1, float d2)
    {
        float dist = Mathf.Abs(d1 - d2);
        return (dist <= distanceThreshold);
    }

    /// <summary>
    /// Called when the GameObject is unloaded.
    /// </summary>
    private void OnDestroy()
    {
        // Unload objects we have created.
        Destroy(boundsAsset);
        boundsAsset = null;
        Destroy(shadowAsset);
        shadowAsset = null;
    }
}

```

Build and Deploy

- As before, build the project and deploy to the HoloLens.
- Wait for scanning and processing of the spatial mapping data to complete.
- When you see the solar system, gaze at the projection box below and perform a select gesture to move it around. While the projection box is selected, a bounding cube will be visible around the projection box.
- Move you head to gaze at a different location in the room. The projection box should follow your gaze. When the shadow below the projection box turns red, you cannot place the hologram on that surface. When the shadow below the projection box turns green, you can place the hologram by performing another select gesture.
- Find and select one of the holographic posters on a wall to move it to a new location. Notice that you cannot place the poster on the floor or ceiling, and that it stays correctly oriented to each wall as you move around.

Chapter 5 - Occlusion

Objectives

- Determine if a hologram is occluded by the spatial mapping mesh.
- Apply different occlusion techniques to achieve a fun effect.

Instructions

First, we are going to allow the spatial mapping mesh to occlude other holograms without occluding the real world:

- In the **Hierarchy** panel, select the **SpatialProcessing** object.
- In the **Inspector** panel, find the **Play Space Manager (Script)** component.
- Click the circle to the right of the **Secondary Material** property.
- Find and select the **Occlusion** material and close the window.

Next, we are going to add a special behavior to Earth, so that it has a blue highlight whenever it becomes occluded by another hologram (like the sun), or by the spatial mapping mesh:

- In the **Project** panel, in the **Holograms** folder, expand the **SolarSystem** object.
- Click on **Earth**.
- In the **Inspector** panel, find the Earth's material (bottom component).
- In the **Shader drop-down**, change the shader to **Custom > OcclusionRim**. This will render a blue highlight around Earth whenever it is occluded by another object.

Finally, we are going to enable an x-ray vision effect for planets in our solar system. We will need to edit **PlanetOcclusion.cs** (found in the Scripts\SolarSystem folder) in order to achieve the following:

1. Determine if a planet is occluded by the SpatialMapping layer (room meshes and planes).
2. Show the wireframe representation of a planet whenever it is occluded by the SpatialMapping layer.
3. Hide the wireframe representation of a planet when it is not blocked by the SpatialMapping layer.

Follow the coding exercise in PlanetOcclusion.cs, or use the following solution:

```
using UnityEngine;
using Academy.HoloToolkit.Unity;

/// <summary>
/// Determines when the occluded version of the planet should be visible.
/// This script allows us to do selective occlusion, so the occlusionObject
/// will only be rendered when a Spatial Mapping surface is occluding the planet,
```

```

/// not when another hologram is responsible for the occlusion.
/// </summary>
public class PlanetOcclusion : MonoBehaviour
{
    [Tooltip("Object to display when the planet is occluded.")]
    public GameObject occlusionObject;

    /// <summary>
    /// Points to raycast to when checking for occlusion.
    /// </summary>
    private Vector3[] checkPoints;

    // Use this for initialization
    void Start()
    {
        occlusionObject.SetActive(false);

        // Set the check points to use when testing for occlusion.
        MeshFilter filter = gameObject.GetComponent<MeshFilter>();
        Vector3 extents = filter.mesh.bounds.extents;
        Vector3 center = filter.mesh.bounds.center;
        Vector3 top = new Vector3(center.x, center.y + extents.y, center.z);
        Vector3 left = new Vector3(center.x - extents.x, center.y, center.z);
        Vector3 right = new Vector3(center.x + extents.x, center.y, center.z);
        Vector3 bottom = new Vector3(center.x, center.y - extents.y, center.z);

        checkPoints = new Vector3[] { center, top, left, right, bottom };
    }

    // Update is called once per frame
    void Update()
    {
        /* TODO: 5.a DEVELOPER CODING EXERCISE 5.a */

        // Check to see if any of the planet's boundary points are occluded.
        for (int i = 0; i < checkPoints.Length; i++)
        {
            // 5.a: Convert the current checkPoint to world coordinates.
            // Call gameObject.transform.TransformPoint(checkPoints[i]).
            // Assign the result to a new Vector3 variable called 'checkPt'.
            Vector3 checkPt = gameObject.transform.TransformPoint(checkPoints[i]);

            // 5.a: Call Vector3.Distance() to calculate the distance
            // between the Main Camera's position and 'checkPt'.
            // Assign the result to a new float variable called 'distance'.
            float distance = Vector3.Distance(Camera.main.transform.position, checkPt);

            // 5.a: Take 'checkPt' and subtract the Main Camera's position from it.
            // Assign the result to a new Vector3 variable called 'direction'.
            Vector3 direction = checkPt - Camera.main.transform.position;

            // Used to indicate if the call to Physics.Raycast() was successful.
            bool raycastHit = false;

            // 5.a: Check if the planet is occluded by a spatial mapping surface.
            // Call Physics.Raycast() with the following arguments:
            // - Pass in the Main Camera's position as the origin.
            // - Pass in 'direction' for the direction.
            // - Pass in 'distance' for the maxDistance.
            // - Pass in SpatialMappingManager.Instance.LayerMask as layerMask.
            // Assign the result to 'raycastHit'.
            raycastHit = Physics.Raycast(Camera.main.transform.position, direction, distance,
                SpatialMappingManager.Instance.LayerMask);

            if (raycastHit)
            {
                // 5.a: Our raycast hit a surface, so the planet is occluded.
                // Set the occlusionObject to active.
                occlusionObject.SetActive(true);
            }
        }
    }
}

```

```
// At least one point is occluded, so break from the loop.  
break;  
}  
else  
{  
    // 5.a: The Raycast did not hit, so the planet is not occluded.  
    // Deactivate the occlusionObject.  
    occlusionObject.SetActive(false);  
}  
}  
}  
}
```

Build and Deploy

- Build and deploy the application to HoloLens, as usual.
 - Wait for scanning and processing of the spatial mapping data to be complete (you should see blue lines appear on walls).
 - Find and select the solar system's projection box and then set the box next to a wall or behind a counter.
 - You can view basic occlusion by hiding behind surfaces to peer at the poster or projection box.
 - Look for the Earth, there should be a blue highlight effect whenever it goes behind another hologram or surface.
 - Watch as the planets move behind the wall or other surfaces in the room. You now have x-ray vision and can see their wireframe skeletons!

The End

Congratulations! You have now completed **MR Spatial 230: Spatial mapping**.

- You know how to scan your environment and load spatial mapping data to Unity.
 - You understand the basics of shaders and how materials can be used to re-visualize the world.
 - You learned of new processing techniques for finding planes and removing triangles from a mesh.
 - You were able to move and place holograms on surfaces that made sense.
 - You experienced different occlusion techniques and harnessed the power of x-ray vision!

MR Sharing 240: Multiple HoloLens devices

11/6/2018 • 21 minutes to read • [Edit Online](#)

Holograms are given presence in our world by remaining in place as we move about in space. HoloLens keeps holograms in place by using various [coordinate systems](#) to keep track of the location and orientation of objects. When we share these coordinate systems between devices, we can create a shared experience that allows us to take part in a shared holographic world.

In this tutorial, we will:

- Setup a network for a shared experience.
- Share holograms across HoloLens devices.
- Discover other people in our shared holographic world.
- Create a shared interactive experience where you can target other players - and launch projectiles at them!

Device support

COURSE	HOLOLENS	IMMERSIVE HEADSETS
MR Sharing 240: Multiple HoloLens devices	✓ <input type="checkbox"/>	

Before you start

Prerequisites

- A Windows 10 PC configured with the correct [tools installed](#) with Internet access.
- At least two HoloLens devices [configured for development](#).

Project files

- Download the [files](#) required by the project. Requires Unity 2017.2 or later.
 - If you still need Unity 5.6 support, please use [this release](#).
 - If you still need Unity 5.5 support, please use [this release](#).
 - If you still need Unity 5.4 support, please use [this release](#).
- Un-archive the files to your desktop or other easy to reach location. Keep the folder name as **SharedHolograms**.

NOTE

If you want to look through the source code before downloading, it's [available on GitHub](#).

Chapter 1 - Holo World

In this chapter, we'll setup our first Unity project and step through the build and deploy process.

Objectives

- Setup Unity to develop holographic apps.
- See your hologram!

Instructions

- Start Unity.
- Select **Open**.
- Enter location as the **SharedHolograms** folder you previously unarchived.
- Select **Project Name** and click **Select Folder**.
- In the **Hierarchy**, right-click the **Main Camera** and select **Delete**.
- In the **HoloToolkit-Sharing-240/Prefabs/Camera** folder, find the **Main Camera** prefab.
- Drag and drop the **Main Camera** into the **Hierarchy**.
- In the **Hierarchy**, click on **Create** and **Create Empty**.
- Right-click the new **GameObject** and select **Rename**.
- Rename the GameObject to **HologramCollection**.
- Select the **HologramCollection** object in the **Hierarchy**.
- In the **Inspector** set the **transform position** to: **X: 0, Y: -0.25, Z: 2**.
- In the **Holograms** folder in the **Project panel**, find the **EnergyHub** asset.
- Drag and drop the **EnergyHub** object from the **Project panel** to the **Hierarchy** as a **child of HologramCollection**.
- Select **File > Save Scene As...**
- Name the scene **SharedHolograms** and click **Save**.
- Press the **Play** button in Unity to preview your holograms.
- Press **Play** a second time to stop preview mode.

Export the project from Unity to Visual Studio

- In Unity select **File > Build Settings**.
- Click **Add Open Scenes** to add the scene.
- Select **Universal Windows Platform** in the **Platform** list and click **Switch Platform**.
- Set **SDK** to **Universal 10**.
- Set **Target device** to **HoloLens** and **UWP Build Type** to **D3D**.
- Check **Unity C# Projects**.
- Click **Build**.
- In the file explorer window that appears, create a **New Folder** named "App".
- Single click the **App** folder.
- Press **Select Folder**.
- When Unity is done, a File Explorer window will appear.
- Open the **App** folder.
- Open **SharedHolograms.sln** to launch Visual Studio.
- Using the top toolbar in Visual Studio, change the target from Debug to **Release** and from ARM to **X86**.
- Click on the drop-down arrow next to Local Machine, and select **Remote Device**.
 - Set the **Address** to the name or IP address of your HoloLens. If you do not know your device IP address, look in **Settings > Network & Internet > Advanced Options** or ask Cortana "**Hey Cortana, What's my IP address?**"
 - Leave the **Authentication Mode** set to **Universal**.
 - Click **Select**
- Click **Debug > Start Without debugging** or press **Ctrl + F5**. If this is the first time deploying to your device, you will need to [pair it with Visual Studio](#).
- Put on your HoloLens and find the EnergyHub hologram.

Chapter 2 - Interaction

In this chapter, we'll interact with our holograms. First, we'll add a cursor to visualize our [Gaze](#). Then, we'll add [Gestures](#) and use our hand to place our holograms in space.

Objectives

- Use gaze input to control a cursor.
- Use gesture input to interact with holograms.

Instructions

Gaze

- In the **Hierarchy panel** select the **HologramCollection** object.
- In the **Inspector panel** click the **Add Component** button.
- In the menu, type in the search box **Gaze Manager**. Select the search result.
- In the **HoloToolkit-Sharing-240\Prefabs\Input** folder, find the **Cursor** asset.
- Drag and drop the **Cursor** asset onto the **Hierarchy**.

Gesture

- In the **Hierarchy panel** select the **HologramCollection** object.
- Click **Add Component** and type **Gesture Manager** in the search field. Select the search result.
- In the **Hierarchy panel**, expand **HologramCollection**.
- Select the child **EnergyHub** object.
- In the **Inspector panel** click the **Add Component** button.
- In the menu, type in the search box **Hologram Placement**. Select the search result.
- Save the scene by selecting **File > Save Scene**.

Deploy and enjoy

- Build and deploy to your HoloLens, using the instructions from the previous chapter.
- Once the app launches on your HoloLens, move your head around and notice how the EnergyHub follows your gaze.
- Notice how the cursor appears when you gaze upon the hologram, and changes to a point light when not gazing at a hologram.
- Perform an air-tap to place the hologram. At this time in our project, you can only place the hologram once (redeploy to try again).

Chapter 3 - Shared Coordinates

It's fun to see and interact with holograms, but let's go further. We'll set up our first shared experience - a hologram everyone can see together.

Objectives

- Setup a network for a shared experience.
- Establish a common reference point.
- Share coordinate systems across devices.
- Everyone sees the same hologram!

NOTE

The **InternetClientServer** and **PrivateNetworkClientServer** capabilities must be declared for an app to connect to the sharing server. This is done for you already in Holograms 240, but keep this in mind for your own projects.

1. In the Unity Editor, go to the player settings by navigating to "Edit > Project Settings > Player"
2. Click on the "Windows Store" tab
3. In the "Publishing Settings > Capabilities" section, check the **InternetClientServer** capability and the **PrivateNetworkClientServer** capability

Instructions

- In the **Project panel** navigate to the **HoloToolkit-Sharing-240\Prefabs\Sharing** folder.
- Drag and drop the **Sharing** prefab into the **Hierarchy panel**.

Next we need to launch the sharing service. Only **one PC** in the shared experience needs to do this step.

- In Unity - in the top-hand menu - select the **HoloToolkit-Sharing-240 menu**.
- Select the **Launch Sharing Service** item in the drop-down.
- Check the **Private Network** option and click **Allow Access** when the firewall prompt appears.
- Note down the IPv4 address displayed in the Sharing Service console window. This is the same IP as the machine the service is being run on.

Follow the rest of the instructions on **all PCs** that will join the shared experience.

- In the **Hierarchy**, select the **Sharing** object.
- In the **Inspector**, on the **Sharing Stage** component, change the **Server Address** from 'localhost' to the IPv4 address of the machine running SharingService.exe.
- In the **Hierarchy** select the **HologramCollection** object.
- In the **Inspector** click the **Add Component** button.
- In the search box, type **Import Export Anchor Manager**. Select the search result.
- In the **Project panel** navigate to the **Scripts** folder.
- Double-click the **HologramPlacement** script to open it in Visual Studio.
- Replace the contents with the code below.

```
using UnityEngine;
using System.Collections.Generic;
using UnityEngine.Windows.Speech;
using Academy.HoloToolkit.Unity;
using Academy.HoloToolkit.Sharing;

public class HologramPlacement : Singleton<HologramPlacement>
{
    /// <summary>
    /// Tracks if we have been sent a transform for the anchor model.
    /// The anchor model is rendered relative to the actual anchor.
    /// </summary>
    public bool GotTransform { get; private set; }

    private bool animationPlayed = false;

    void Start()
    {
        // We care about getting updates for the anchor transform.
        CustomMessages.Instance.MessageHandlers[CustomMessages.TestMessageID.StageTransform] =
        this.OnStageTransform;

        // And when a new user join we will send the anchor transform we have.
        SharingSessionTracker.Instance.SessionJoined += Instance_SessionJoined;
    }

    void OnStageTransform(CustomMessage message)
    {
        if (GotTransform)
            return;

        GotTransform = true;
        transform.position = message.Data;
    }

    void Instance_SessionJoined(Session session)
    {
        if (session != null)
            session.SessionJoined += Session_SessionJoined;
    }

    void Session_SessionJoined(Session session)
    {
        if (session != null)
            session.SessionJoined -= Session_SessionJoined;
    }
}
```

```

}

/// <summary>
/// When a new user joins we want to send them the relative transform for the anchor if we have it.
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void Instance_SessionJoined(object sender, SharingSessionTracker.SessionJoinedEventArgs e)
{
    if (GotTransform)
    {
        CustomMessages.Instance.SendStageTransform(transform.localPosition, transform.localRotation);
    }
}

void Update()
{
    if (GotTransform)
    {
        if (ImportExportAnchorManager.Instance.AnchorEstablished &&
            animationPlayed == false)
        {
            // This triggers the animation sequence for the anchor model and
            // puts the cool materials on the model.
            GetComponent<EnergyHubBase>().SendMessage("OnSelect");
            animationPlayed = true;
        }
    }
    else
    {
        transform.position = Vector3.Lerp(transform.position, ProposeTransformPosition(), 0.2f);
    }
}

Vector3 ProposeTransformPosition()
{
    // Put the anchor 2m in front of the user.
    Vector3 retval = Camera.main.transform.position + Camera.main.transform.forward * 2;

    return retval;
}

public void OnSelect()
{
    // Note that we have a transform.
    GotTransform = true;

    // And send it to our friends.
    CustomMessages.Instance.SendStageTransform(transform.localPosition, transform.localRotation);
}

/// <summary>
/// When a remote system has a transform for us, we'll get it here.
/// </summary>
/// <param name="msg"></param>
void OnStageTransform(NetworkInMessage msg)
{
    // We read the user ID but we don't use it here.
    msg.ReadInt64();

    transform.localPosition = CustomMessages.Instance.ReadVector3(msg);
    transform.localRotation = CustomMessages.Instance.ReadQuaternion(msg);

    // The first time, we'll want to send the message to the anchor to do its animation and
    // swap its materials.
    if (GotTransform == false)
    {
        GetComponent<EnergyHubBase>().SendMessage("OnSelect");
    }
}

```

```

        GotTransform = true;
    }

    public void ResetStage()
    {
        // We'll use this later.
    }
}

```

- Back in Unity, select the **HologramCollection** in the **Hierarchy panel**.
- In the **Inspector panel** click the **Add Component** button.
- In the menu, type in the search box **App State Manager**. Select the search result.

Deploy and enjoy

- Build the project for your HoloLens devices.
- Designate one HoloLens to deploy to first. You will need to wait for the Anchor to be uploaded to the service before you can place the EnergyHub (this can take ~30-60 seconds). Until the upload is done, your tap gestures will be ignored.
- After the EnergyHub has been placed, its location will be uploaded to the service and you can then deploy to all other HoloLens devices.
- When a new HoloLens first joins the session, the location of the EnergyHub may not be correct on that device. However, as soon as the anchor and EnergyHub locations have been downloaded from the service, the EnergyHub should jump to the new, shared location. If this does not happen within ~30-60 seconds, walk to where the original HoloLens was when setting the anchor to gather more environment clues. If the location still does not lock on, redeploy to the device.
- When the devices are all ready and running the app, look for the EnergyHub. Can you all agree on the hologram's location and which direction the text is facing?

Chapter 4 - Discovery

Everyone can now see the same hologram! Now let's see everyone else connected to our shared holographic world. In this chapter, we'll grab the head location and rotation of all other HoloLens devices in the same sharing session.

Objectives

- Discover each other in our shared experience.
- Choose and share a player avatar.
- Attach the player avatar next to everyone's heads.

Instructions

- In the **Project panel** navigate to the **Holograms** folder.
- Drag and drop the **PlayerAvatarStore** into the **Hierarchy**.
- In the **Project panel** navigate to the **Scripts** folder.
- Double-click the **AvatarSelector** script to open it in Visual Studio.
- Replace the contents with the code below.

```

using UnityEngine;
using Academy.HoloToolkit.Unity;

/// <summary>
/// Script to handle the user selecting the avatar.
/// </summary>
public class AvatarSelector : MonoBehaviour
{
    /// <summary>
    /// This is the index set by the PlayerAvatarStore for the avatar.
    /// </summary>
    public int AvatarIndex { get; set; }

    /// <summary>
    /// Called when the user is gazing at this avatar and air-taps it.
    /// This sends the user's selection to the rest of the devices in the experience.
    /// </summary>
    void OnSelect()
    {
        PlayerAvatarStore.Instance.DismissAvatarPicker();

        LocalPlayerManager.Instance.SetUserAvatar(AvatarIndex);
    }

    void Start()
    {
        // Add Billboard component so the avatar always faces the user.
        Billboard billboard = gameObject.GetComponent<Billboard>();
        if (billboard == null)
        {
            billboard = gameObject.AddComponent<Billboard>();
        }

        // Lock rotation along the Y axis.
        billboard.PivotAxis = PivotAxis.Y;
    }
}

```

- In the **Hierarchy** select the **HologramCollection** object.
- In the **Inspector** click **Add Component**.
- In the search box, type **Local Player Manager**. Select the search result.
- In the **Hierarchy** select the **HologramCollection** object.
- In the **Inspector** click **Add Component**.
- In the search box, type **Remote Player Manager**. Select the search result.
- Open the **HologramPlacement** script in Visual Studio.
- Replace the contents with the code below.

```

using UnityEngine;
using System.Collections.Generic;
using UnityEngine.Windows.Speech;
using Academy.HoloToolkit.Unity;
using Academy.HoloToolkit.Sharing;

public class HologramPlacement : Singleton<HologramPlacement>
{
    /// <summary>
    /// Tracks if we have been sent a transform for the model.
    /// The model is rendered relative to the actual anchor.
    /// </summary>
    public bool GotTransform { get; private set; }

    /// <summary>
    /// When the experience starts we disable all of the rendering of the model
    /// </summary>
}
```

```

/// When the experience starts, we disable all of the rendering of the model.
/// </summary>
List<MeshRenderer> disabledRenderers = new List<MeshRenderer>();

void Start()
{
    // When we first start, we need to disable the model to avoid it obstructing the user picking a hat.
    DisableModel();

    // We care about getting updates for the model transform.
    CustomMessages.Instance.MessageHandlers[CustomMessages.TestMessageID.StageTransform] =
this.OnStageTransform;

    // And when a new user join we will send the model transform we have.
    SharingSessionTracker.Instance.SessionJoined += Instance_SessionJoined;
}

/// <summary>
/// When a new user joins we want to send them the relative transform for the model if we have it.
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void Instance_SessionJoined(object sender, SharingSessionTracker.SessionJoinedEventArgs e)
{
    if (GotTransform)
    {
        CustomMessages.Instance.SendStageTransform(transform.localPosition, transform.localRotation);
    }
}

/// <summary>
/// Turns off all renderers for the model.
/// </summary>
void DisableModel()
{
    foreach (MeshRenderer renderer in gameObject.GetComponentsInChildren<MeshRenderer>())
    {
        if (renderer.enabled)
        {
            renderer.enabled = false;
            disabledRenderers.Add(renderer);
        }
    }

    foreach (MeshCollider collider in gameObject.GetComponentsInChildren<MeshCollider>())
    {
        collider.enabled = false;
    }
}

/// <summary>
/// Turns on all renderers that were disabled.
/// </summary>
void EnableModel()
{
    foreach (MeshRenderer renderer in disabledRenderers)
    {
        renderer.enabled = true;
    }

    foreach (MeshCollider collider in gameObject.GetComponentsInChildren<MeshCollider>())
    {
        collider.enabled = true;
    }

    disabledRenderers.Clear();
}

```

```

void update()
{
    // Wait till users pick an avatar to enable renderers.
    if (disabledRenderers.Count > 0)
    {
        if (!PlayerAvatarStore.Instance.PickerActive &&
            ImportExportAnchorManager.Instance.AnchorEstablished)
        {
            // After which we want to start rendering.
            EnableModel();

            // And if we've already been sent the relative transform, we will use it.
            if (GotTransform)
            {
                // This triggers the animation sequence for the model and
                // puts the cool materials on the model.
                GetComponent<EnergyHubBase>().SendMessage("OnSelect");
            }
        }
    }
    else if (GotTransform == false)
    {
        transform.position = Vector3.Lerp(transform.position, ProposeTransformPosition(), 0.2f);
    }
}

Vector3 ProposeTransformPosition()
{
    // Put the model 2m in front of the user.
    Vector3 retval = Camera.main.transform.position + Camera.main.transform.forward * 2;

    return retval;
}

public void OnSelect()
{
    // Note that we have a transform.
    GotTransform = true;

    // And send it to our friends.
    CustomMessages.Instance.SendStageTransform(transform.localPosition, transform.localRotation);
}

/// <summary>
/// When a remote system has a transform for us, we'll get it here.
/// </summary>
/// <param name="msg"></param>
void OnStageTransform(NetworkInMessage msg)
{
    // We read the user ID but we don't use it here.
    msg.ReadInt64();

    transform.localPosition = CustomMessages.Instance.ReadVector3(msg);
    transform.localRotation = CustomMessages.Instance.ReadQuaternion(msg);

    // The first time, we'll want to send the message to the model to do its animation and
    // swap its materials.
    if (disabledRenderers.Count == 0 && GotTransform == false)
    {
        GetComponent<EnergyHubBase>().SendMessage("OnSelect");
    }

    GotTransform = true;
}

public void ResetStage()
{
    // We'll use this later.
}

```

```
}
```

- Open the **AppStateManager** script in Visual Studio.
- Replace the contents with the code below.

```

using UnityEngine;
using Academy.HoloToolkit.Unity;

/// <summary>
/// Keeps track of the current state of the experience.
/// </summary>
public class AppStateManager : Singleton<AppStateManager>
{
    /// <summary>
    /// Enum to track progress through the experience.
    /// </summary>
    public enum AppState
    {
        Starting = 0,
        WaitingForAnchor,
        WaitingForStageTransform,
        PickingAvatar,
        Ready
    }

    /// <summary>
    /// Tracks the current state in the experience.
    /// </summary>
    public AppState CurrentAppState { get; set; }

    void Start()
    {
        // We start in the 'picking avatar' mode.
        CurrentAppState = AppState.PickingAvatar;

        // We start by showing the avatar picker.
        PlayerAvatarStore.Instance.SpawnAvatarPicker();
    }

    void Update()
    {
        switch (CurrentAppState)
        {
            case AppState.PickingAvatar:
                // Avatar picking is done when the avatar picker has been dismissed.
                if (PlayerAvatarStore.Instance.PickerActive == false)
                {
                    CurrentAppState = AppState.WaitingForAnchor;
                }
                break;
            case AppState.WaitingForAnchor:
                if (ImportExportAnchorManager.Instance.AnchorEstablished)
                {
                    CurrentAppState = AppState.WaitingForStageTransform;
                    GestureManager.Instance.OverrideFocusedObject = HologramPlacement.Instance.gameObject;
                }
                break;
            case AppState.WaitingForStageTransform:
                // Now if we have the stage transform we are ready to go.
                if (HologramPlacement.Instance.GotTransform)
                {
                    CurrentAppState = AppState.Ready;
                    GestureManager.Instance.OverrideFocusedObject = null;
                }
                break;
        }
    }
}

```

Deploy and Enjoy

- Build and deploy the project to your HoloLens devices.
- When you hear a pinging sound, find the avatar selection menu and select an avatar with the air-tap gesture.
- If you're not looking at any holograms, the point light around your cursor will turn a different color when your HoloLens is communicating with the service: initializing (dark purple), downloading the anchor (green), importing/exporting location data (yellow), uploading the anchor (blue). If your point light around your cursor is the default color (light purple), then you are ready to interact with other players in your session!
- Look at other people connected to your space - there will be a holographic robot floating above their shoulder and mimicking their head motions!

Chapter 5 - Placement

In this chapter, we'll make the anchor able to be placed on real-world surfaces. We'll use shared coordinates to place that anchor in the middle point between everyone connected to the shared experience.

Objectives

- Place holograms on the spatial map based on players' head position.

Instructions

- In the **Project panel** navigate to the **Holograms** folder.
- Drag and drop the **CustomSpatialMapping** prefab onto the **Hierarchy**.
- In the **Project panel** navigate to the **Scripts** folder.
- Double-click the **AppStateManager** script to open it in Visual Studio.
- Replace the contents with the code below.

```
using UnityEngine;
using Academy.HoloToolkit.Unity;

/// <summary>
/// Keeps track of the current state of the experience.
/// </summary>
public class AppStateManager : Singleton<AppStateManager>
{
    /// <summary>
    /// Enum to track progress through the experience.
    /// </summary>
    public enum AppState
    {
        Starting = 0,
        PickingAvatar,
        WaitingForAnchor,
        WaitingForStageTransform,
        Ready
    }

    // The object to call to make a projectile.
    GameObject shootHandler = null;

    /// <summary>
    /// Tracks the current state in the experience.
    /// </summary>
    public AppState CurrentAppState { get; set; }

    void Start()
    {
        // The shootHandler shoots projectiles.
        if (GetComponent<ProjectileLauncher>() != null)
        {
            shootHandler = GetComponent<ProjectileLauncher>().gameObject;
        }
    }
}
```

```

// We start in the 'picking avatar' mode.
CurrentAppState = AppState.PickingAvatar;

// Spatial mapping should be disabled when we start up so as not
// to distract from the avatar picking.
SpatialMappingManager.Instance.StopObserver();
SpatialMappingManager.Instance.gameObject.SetActive(false);

// On device we start by showing the avatar picker.
PlayerAvatarStore.Instance.SpawnAvatarPicker();
}

public void ResetStage()
{
    // If we fall back to waiting for anchor, everything needed to
    // get us into setting the target transform state will be setup.
    if (CurrentAppState != AppState.PickingAvatar)
    {
        CurrentAppState = AppState.WaitingForAnchor;
    }

    // Reset the underworld.
    if (UnderworldBase.Instance)
    {
        UnderworldBase.Instance.ResetUnderworld();
    }
}

void Update()
{
    switch (CurrentAppState)
    {
        case AppState.PickingAvatar:
            // Avatar picking is done when the avatar picker has been dismissed.
            if (PlayerAvatarStore.Instance.PickerActive == false)
            {
                CurrentAppState = AppState.WaitingForAnchor;
            }
            break;

        case AppState.WaitingForAnchor:
            // Once the anchor is established we need to run spatial mapping for a
            // little while to build up some meshes.
            if (ImportExportAnchorManager.Instance.AnchorEstablished)
            {
                CurrentAppState = AppState.WaitingForStageTransform;
                GestureManager.Instance.OverrideFocusedObject = HologramPlacement.Instance.gameObject;

                SpatialMappingManager.Instance.gameObject.SetActive(true);
                SpatialMappingManager.Instance.DrawVisualMeshes = true;
                SpatialMappingDeformation.Instance.ResetGlobalRendering();
                SpatialMappingManager.Instance.StartObserver();
            }
            break;

        case AppState.WaitingForStageTransform:
            // Now if we have the stage transform we are ready to go.
            if (HologramPlacement.Instance.GotTransform)
            {
                CurrentAppState = AppState.Ready;
                GestureManager.Instance.OverrideFocusedObject = shootHandler;
            }
            break;
    }
}
}

```

- In the **Project panel** navigate to the **Scripts** folder.

- Double-click the **HologramPlacement** script to open it in Visual Studio.
- Replace the contents with the code below.

```

using UnityEngine;
using System.Collections.Generic;
using UnityEngine.Windows.Speech;
using Academy.HoloToolkit.Unity;
using Academy.HoloToolkit.Sharing;

public class HologramPlacement : Singleton<HologramPlacement>
{
    /// <summary>
    /// Tracks if we have been sent a transform for the model.
    /// The model is rendered relative to the actual anchor.
    /// </summary>
    public bool GotTransform { get; private set; }

    /// <summary>
    /// When the experience starts, we disable all of the rendering of the model.
    /// </summary>
    List<MeshRenderer> disabledRenderers = new List<MeshRenderer>();

    /// <summary>
    /// We use a voice command to enable moving the target.
    /// </summary>
    KeywordRecognizer keywordRecognizer;

    void Start()
    {
        // When we first start, we need to disable the model to avoid it obstructing the user picking a hat.
        DisableModel();

        // We care about getting updates for the model transform.
        CustomMessages.Instance.MessageHandlers[CustomMessages.TestMessageID.StageTransform] =
this.OnStageTransform;

        // And when a new user join we will send the model transform we have.
        SharingSessionTracker.Instance.SessionJoined += Instance_SessionJoined;

        // And if the users want to reset the stage transform.
        CustomMessages.Instance.MessageHandlers[CustomMessages.TestMessageID.ResetStage] = this.OnResetStage;

        // Setup a keyword recognizer to enable resetting the target location.
        List<string> keywords = new List<string>();
        keywords.Add("Reset Target");
        keywordRecognizer = new KeywordRecognizer(keywords.ToArray());
        keywordRecognizer.OnPhraseRecognized += KeywordRecognizer_OnPhraseRecognized;
        keywordRecognizer.Start();
    }

    /// <summary>
    /// When the keyword recognizer hears a command this will be called.
    /// In this case we only have one keyword, which will re-enable moving the
    /// target.
    /// </summary>
    /// <param name="args">information to help route the voice command.</param>
    private void KeywordRecognizer_OnPhraseRecognized(PhraseRecognizedEventArgs args)
    {
        ResetStage();
    }

    /// <summary>
    /// Resets the stage transform, so users can place the target again.
    /// </summary>
    public void ResetStage()
    {
        GotTransform = false;
    }
}

```

```

// AppStateManager needs to know about this so that
// the right objects get input routed to them.
AppStateManager.Instance.ResetStage();

// Other devices in the experience need to know about this as well.
CustomMessages.Instance.SendResetStage();

// And we need to reset the object to its start animation state.
GetComponent<EnergyHubBase>().ResetAnimation();
}

/// <summary>
/// When a new user joins we want to send them the relative transform for the model if we have it.
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void Instance_SessionJoined(object sender, SharingSessionTracker.SessionJoinedEventArgs e)
{
    if (GotTransform)
    {
        CustomMessages.Instance.SendStageTransform(transform.localPosition, transform.localRotation);
    }
}

/// <summary>
/// Turns off all renderers for the model.
/// </summary>
void DisableModel()
{
    foreach (MeshRenderer renderer in gameObject.GetComponentsInChildren<MeshRenderer>())
    {
        if (renderer.enabled)
        {
            renderer.enabled = false;
            disabledRenderers.Add(renderer);
        }
    }

    foreach (MeshCollider collider in gameObject.GetComponentsInChildren<MeshCollider>())
    {
        collider.enabled = false;
    }
}

/// <summary>
/// Turns on all renderers that were disabled.
/// </summary>
void EnableModel()
{
    foreach (MeshRenderer renderer in disabledRenderers)
    {
        renderer.enabled = true;
    }

    foreach (MeshCollider collider in gameObject.GetComponentsInChildren<MeshCollider>())
    {
        collider.enabled = true;
    }

    disabledRenderers.Clear();
}

void Update()
{
    // Wait till users pick an avatar to enable renderers.
    if (disabledRenderers.Count > 0)
    {
}

```

```

        if (!PlayerAvatarStore.Instance.PickerActive &&
            ImportExportAnchorManager.Instance.AnchorEstablished)
        {
            // After which we want to start rendering.
            EnableModel();

            // And if we've already been sent the relative transform, we will use it.
            if (GotTransform)
            {
                // This triggers the animation sequence for the model and
                // puts the cool materials on the model.
                GetComponent<EnergyHubBase>().SendMessage("OnSelect");
            }
        }

        else if (GotTransform == false)
        {
            transform.position = Vector3.Lerp(transform.position, ProposeTransformPosition(), 0.2f);
        }
    }

    Vector3 ProposeTransformPosition()
    {
        Vector3 retval;
        // We need to know how many users are in the experience with good transforms.
        Vector3 cumulatedPosition = Camera.main.transform.position;
        int playerCount = 1;
        foreach (RemotePlayerManager.RemoteHeadInfo remoteHead in
        RemotePlayerManager.Instance.remoteHeadInfos)
        {
            if (remoteHead.Anchored && remoteHead.Active)
            {
                playerCount++;
                cumulatedPosition += remoteHead.HeadObject.transform.position;
            }
        }

        // If we have more than one player ...
        if (playerCount > 1)
        {
            // Put the transform in between the players.
            retval = cumulatedPosition / playerCount;
            RaycastHit hitInfo;

            // And try to put the transform on a surface below the midpoint of the players.
            if (Physics.Raycast(retval, Vector3.down, out hitInfo, 5,
            SpatialMappingManager.Instance.LayerMask))
            {
                retval = hitInfo.point;
            }
        }
        // If we are the only player, have the model act as the 'cursor' ...
        else
        {
            // We prefer to put the model on a real world surface.
            RaycastHit hitInfo;

            if (Physics.Raycast(Camera.main.transform.position, Camera.main.transform.forward, out hitInfo,
            30, SpatialMappingManager.Instance.LayerMask))
            {
                retval = hitInfo.point;
            }
            else
            {
                // But if we don't have a ray that intersects the real world, just put the model 2m in
                // front of the user.
                retval = Camera.main.transform.position + Camera.main.transform.forward * 2;
            }
        }
    }
}

```

```

        return retval;
    }

    public void OnSelect()
    {
        // Note that we have a transform.
        GotTransform = true;

        // And send it to our friends.
        CustomMessages.Instance.SendStageTransform(transform.localPosition, transform.localRotation);
    }

    /// <summary>
    /// When a remote system has a transform for us, we'll get it here.
    /// </summary>
    /// <param name="msg"></param>
    void OnStageTransform(NetworkInMessage msg)
    {
        // We read the user ID but we don't use it here.
        msg.ReadInt64();

        transform.localPosition = CustomMessages.Instance.ReadVector3(msg);
        transform.localRotation = CustomMessages.Instance.ReadQuaternion(msg);

        // The first time, we'll want to send the message to the model to do its animation and
        // swap its materials.
        if (disabledRenderers.Count == 0 && GotTransform == false)
        {
            GetComponent<EnergyHubBase>().SendMessage("OnSelect");
        }

        GotTransform = true;
    }

    /// <summary>
    /// When a remote system has a transform for us, we'll get it here.
    /// </summary>
    void OnResetStage(NetworkInMessage msg)
    {
        GotTransform = false;

        GetComponent<EnergyHubBase>().ResetAnimation();
        AppStateManager.Instance.ResetStage();
    }
}

```

Deploy and enjoy

- Build and deploy the project to your HoloLens devices.
- When the app is ready, stand in a circle and notice how the EnergyHub appears in the center of everyone.
- Tap to place the EnergyHub.
- Try the voice command 'Reset Target' to pick the EnergyHub back up and work together as a group to move the hologram to a new location.

Chapter 6 - Real-World Physics

In this chapter we'll add holograms that bounce off real-world surfaces. Watch your space fill up with projects launched by both you and your friends!

Objectives

- Launch projectiles that bounce off real-world surfaces.
- Share the projectiles so other players can see them.

Instructions

- In the **Hierarchy** select the **HologramCollection** object.
- In the **Inspector** click **Add Component**.
- In the search box, type **Projectile Launcher**. Select the search result.

Deploy and enjoy

- Build and deploy to your HoloLens devices.
- When the app is running on all devices, perform an air-tap to launch projectile at real world surfaces.
- See what happens when your projectile collides with another player's avatar!

Chapter 7 - Grand Finale

In this chapter, we'll uncover a portal that can only be discovered with collaboration.

Objectives

- Work together to launch enough projectiles at the anchor to uncover a secret portal!

Instructions

- In the **Project panel** navigate to the **Holograms** folder.
- Drag and drop the **Underworld** asset as a **child of HologramCollection**.
- With **HologramCollection** selected, click the **Add Component** button in the **Inspector**.
- In the menu, type in the search box **ExplodeTarget**. Select the search result.
- With **HologramCollection** selected, from the **Hierarchy** drag the **EnergyHub** object to the **Target** field in the **Inspector**.
- With **HologramCollection** selected, from the **Hierarchy** drag the **Underworld** object to the **Underworld** field in the **Inspector**.

Deploy and enjoy

- Build and deploy to your HoloLens devices.
- When the app has launched, collaborate together to launch projectiles at the EnergyHub.
- When the underworld appears, launch projectiles at underworld robots (hit a robot three times for extra fun).

MR Sharing 250: HoloLens and immersive headsets

11/6/2018 • 19 minutes to read • [Edit Online](#)

With the flexibility of Universal Windows Platform (UWP), it is easy to create an application that spans multiple devices. With this flexibility, we can create experiences that leverage the strengths of each device. This tutorial will cover a basic shared experience that runs on both HoloLens and Windows Mixed Reality immersive headsets. This content was originally delivered at the Microsoft Build 2017 conference in Seattle, WA.

In this tutorial, we will:

- Setup a network using UNET.
- Share holograms across mixed reality devices.
- Establish a different view of the application depending on which mixed reality device is being used.
- Create a shared experience where HoloLens users guide immersive headsets users through some simple puzzles.

Device support

COURSE	HOLOLENS	IMMERSIVE HEADSETS
MR Sharing 250: HoloLens and immersive headsets	✓ <input type="checkbox"/>	✓ <input type="checkbox"/>

Before you start

Prerequisites

- A Windows 10 PC with the [necessary development tools](#) and [configured to support a Windows Mixed Reality immersive headset](#).
- An Xbox controller that works with your PC.
- At least one HoloLens device and one immersive headset.
- A network which allows UDP Broadcast for discovery.

Project files

- Download the [files](#) required by the project. Extract the files to an easy to remember location.
- This project requires the [a recommended version of Unity with Windows Mixed Reality support](#).

NOTE

If you want to look through the source code before downloading, it's [available on GitHub](#).

Chapter 1 - Holo World

Objectives

Make sure the development environment is ready to go with a simple project.

What we will build

An application that shows a hologram on either HoloLens or a Windows Mixed Reality immersive headset.

Steps

- Open Unity.
 - Select **Open**.
 - Navigate to where you extracted the project files.
 - Click **Select Folder**.
 - *It will take a little while for Unity to process the project the first time.*
- Check that Mixed Reality is enabled in Unity.
 - Open the build settings dialog (**Control+Shift+B** or **File > Build Settings...**).
 - Select **Universal Windows Platform** then click **Switch Platform**.
 - Select **Edit>Player Settings**.
 - In the **Inspector** panel on the right hand side, expand **XR Settings**.
 - Check the **Virtual Reality Supported** box.
 - *Windows Mixed Reality should be the Virtual Reality SDK.*
- Create a scene.
 - In the **Hierarchy** right click **Main Camera** select **Delete**.
 - From **HoloToolkit > Input > Prefabs** drag **MixedRealityCameraParent** to the **Hierarchy**.
- Add Holograms to the scene
 - From **AppPrefabs** drag **Skybox** to the **Scene View**.
 - From **AppPrefabs** drag **Managers** to the **Hierarchy**.
 - From **AppPrefabs** drag **Island** to the **Hierarchy**.
- Save And build
 - Save (either **Control+S** or **File > Save Scene**)
 - Since this is a new scene, you'll need to name it. Name doesn't matter, but we use **SharedMixedReality**.
- Export To Visual Studio
 - Open the build menu (**Control+Shift+B** or **File > Build Settings**)
 - Click **Add Open Scenes**.
 - Check **Unity C# Projects**
 - Click **Build**.
 - In the file explorer window that appears, create a New Folder named **App**.
 - Single click the **App** folder.
 - Press **Select Folder**.
 - **Wait for the build to complete**
 - In the file explorer window that appears, navigate into the **App** folder.
 - Double-click **SharedMixedReality.sln** to launch Visual Studio
- Build From Visual Studio
 - Using the top toolbar change target to **Release** and **x86**.
 - Click the arrow next to **Local Machine** and select **Device** to deploy to HoloLens
 - Click the arrow next to **Device** and select **Local Machine** to deploy for the mixed reality headset.
 - Click **Debug->Start Without Debugging** or **Control+F5** to start the application.

Digging into the code

In the project panel, navigate to **Assets\HoloToolkit\Input\Scripts\Utilities** and double click **MixedRealityCameraManager.cs** to open it.

Overview: MixedRealityCameraManager.cs is a simple script that adjusts quality level and background settings based on the device. Key here is `HolographicSettings.IsDisplayOpaque`, which allows a script to detect if the device is a HoloLens (`IsDisplayOpaque` returns false) or an immersive headset (`IsDisplayOpaque` returns true).

Enjoy your progress

At this point the application will just render a hologram. We will add interaction to the hologram later. Both devices will render the hologram the same. The immersive headset will also render a blue sky and clouds background.

Chapter 2 - Interaction

Objectives

Show how to handle input for a Windows Mixed Reality application.

What we will build

Building on the application from chapter 1, we will add functionality to allow the user to pick up the hologram and place it on a real world surface in HoloLens or on a virtual table in an immersive headset.

Input Refresher: On HoloLens the select gesture is the **air tap**. On immersive headsets, we will use the **A** button on the Xbox controller. For more information on input [start here](#).

Steps

- Add Input manager
 - From **HoloToolkit > Input > Prefabs** drag **InputManager** to **Hierarchy** as a child of **Managers**.
 - From **HoloToolkit > Input > Prefabs > Cursor** drag **Cursor** to **Hierarchy**.
- Add Spatial Mapping
 - From **HoloToolkit > SpatialMapping > Prefabs** drag **SpatialMapping** to **Hierarchy**.
- Add Virtual Playspace
 - In **Hierarchy** expand **MixedRealityCameraParent** select **Boundary**
 - In **Inspector** panel check the box to enable **Boundary**
 - From **AppPrefabs** drag **VRRoom** to **Hierarchy**.
- Add WorldAnchorManager
 - In **Hierarchy**, Select **Managers**.
 - In **Inspector**, click **Add Component**.
 - Type **World Anchor Manager**.
 - Select **World Anchor Manager** to add it.
- Add TapToPlace to the Island
 - In **Hierarchy**, expand **Island**.
 - Select **MixedRealityLand**.
 - In **Inspector**, click **Add Component**.
 - Type **Tap To Place** and select it.
 - Check **Place Parent On Tap**.
 - Set **Placement Offset** to **(0, 0.1, 0)**.
- Save and Build as before

Digging into the code

Script 1 - GamepadInput.cs

In the project panel navigate to **Assets\HoloToolkit\Input\Scripts\InputSources** and double click

GamepadInput.cs to open it. From the same path in the project panel, also double click

InteractionSourceInputSource.cs.

Note that both scripts have a common base class, **BaseInputSource**.

BaseInputSource keeps a reference to an **InputManager**, which allows a script to trigger events. In this case, the

InputClicked event is relevant. This will be important to remember when we get to script 2, TapToPlace. In the case of GamePadInput, we poll for the A button on the controller to be pressed, then we raise the InputClicked event. In the case of InteractionSourceInputSource, we raise the InputClicked event in response to the TappedEvent.

Script 2 - TapToPlace.cs

In the project panel navigate to **Assets\HoloToolkit\SpatialMapping\Scripts** and double click **TapToPlace.cs** to open it.

The first thing many developers want to implement when creating a Holographic application is moving Holograms with gesture input. As such, we've endeavored to thoroughly comment this script. A few things are worth highlighting for this tutorial.

First, note that TapToPlace implements IInputClickHandler. IInputClickHandler exposes the functions that handle the InputClicked event raised by GamePadInput.cs or InteractionSourceInputSource.cs. OnInputClicked is called when a BaseInputSource detects a click while the object with TapToPlace is in focus. Either airtapping on HoloLens or pressing the A button on the Xbox controller will trigger the event.

Second is the code be executed in update to see if a surface is being looked at so we can place the game object on a surface, like a table. The immersive headset doesn't have a concept of real surfaces, so the object that represents the table top (Vroom > TableThingy > Cube) has been marked with the SpatialMapping physics layer, so the ray cast in Update will collide with the virtual table top.

Enjoy your progress

This time you can select the island to move it. On HoloLens you can move the island to a real surface. In the immersive headset you can move the island to the virtual table we added.

Chapter 3 - Sharing

Objectives

Ensure that the network is correctly configured and detail how spatial anchors are shared between devices.

What we will build

We will convert our project to a multiplayer project. We will add UI and logic to host or join sessions. HoloLens users will see each other in the session with clouds over their heads, and immersive headset users have clouds near to where the anchor is. Users in the immersive headsets will see the HoloLens users relative to the origin of the scene. HoloLens users will all see the hologram of the island in the same place. It is key to note that the users in the immersive headsets will not be on the island during this chapter, but will behave very similarly to HoloLens, with a birds eye view of the island.

Steps

- Remove Island and VRRoom
 - In **Hierarchy** right-click **Island** select **Delete**
 - In **Hierarchy** right-click **VRRoom** select **Delete**
- Add Usland
 - From **AppPrefabs** drag **Usland** to **Hierarchy**.
- From **AppPrefabs** drag each of the following to **Hierarchy**:
 - **UNETSharingStage**
 - **UNetAnchorRoot**
 - **UIContainer**
 - **DebugPanelButton**
- Save and Build as before

Digging into the code

In the project panel, navigate to **Assets\AppPrefabs\Support\SharingWithUnet\Scripts** and double-click on **UnetAnchorManager.cs**. The ability for one HoloLens to share tracking information with another HoloLens such that both devices can share the same space is near magical. The power of mixed reality comes alive when two or more people can collaborate using the same digital data.

A few things to point out in this script:

In the start function, notice the check for **IsDisplayOpaque**. In this case, we pretend that the Anchor is established. This is because the immersive headsets do not expose a way to import or export anchors. If we are running on a HoloLens, however, this script implements sharing anchors between the devices. The device that starts the session will create an anchor for exporting. The device that joins a session will request the anchor from the device that started the session.

Exporting:

When a user creates a session, NetworkDiscoveryWithAnchors will call UNETAnchorManagers CreateAnchor function. Let's follow CreateAnchor flow.

We start by doing some housekeeping, clearing out any data we may have collected for previous anchors. Then we check if there is a cached anchor to load. The anchor data tends to be between 5 and 20 MB, so reusing cached anchors can save on the amount of data we need to transfer over the network. We'll see how this works a bit later. Even if we are reusing the anchor, we need to get the anchor data ready in case a new client joins that doesn't have the anchor.

Speaking of getting the anchor data ready, the WorldAnchorTransferBatch class exposes the functionality to prepare anchor data for sending to another device or application and the functionality to import the anchor data. Since we're on the export path, we will add our anchor to the WorldAnchorTransferBatch and call the ExportAsync function. ExportAsync will then call the WriteBuffer callback as it generates data for export. When all of the data has been exported ExportComplete will be called. In WriteBuffer we add the chunk of data to a list we keep for exporting. In ExportComplete we convert the list to an array. The AnchorName variable is also set, which will trigger other devices to request the anchor if they don't have it.

In some cases the anchor won't export or will create so little data that we will try again. Here we just call CreateAnchor again.

A final function in the export path is AnchorFoundRemotely. When another device finds the anchor, that device will tell the host, and the host will use that as a signal that the anchor is a "good anchor" and can be cached.

Importing:

When a HoloLens joins a session, it needs to import an anchor. In UNETAnchorManager's Update function, the AnchorName is polled. When the anchor name changes, the import process begins. First, we try to load the anchor with the specified name from the local anchor store. If we already have it, we can use it without downloading the data again. If we don't have it, then we call WaitForAnchor which will initiate the download.

When the download is completed, NetworkTransmitter_dataReadyEvent is called. This will signal the Update loop to call ImportAsync with the downloaded data. ImportAsync will call ImportComplete when the import process is complete. If the import is successful, the anchor will be saved in the local player store. PlayerController.cs actually makes the call to AnchorFoundRemotely to let the host know that a good anchor has been established.

Enjoy your progress

This time a user with a HoloLens will host a session using the **start session** button in the UI. Other users, both on HoloLens or an immersive headset, will select the session and then select the **join session** button in the UI. If you have multiple people with HoloLens devices, they will have red clouds over their heads. There will also be a blue cloud for each immersive headset, but the blue clouds will not be above the headsets, as the headsets are not trying to find the same world coordinate space as the HoloLens devices.

This point in the project is a contained sharing application; it doesn't do very much, and could act as a baseline. In the next chapters, we will start building an experience for people to enjoy. To get further guidance on shared experience design, go [here](#).

Chapter 4 - Immersion and teleporting

Objectives

Cater the experience to each type of mixed reality device.

What we will build

We will update the application to put immersive headset users on the island with an immersive view. HoloLens users will still have the bird's eye view of the island. Users of each device type can see other users as they appear in the world. For instance, immersive headset users can see the other avatars on other paths on the island, and they see the HoloLens users as giant clouds above the island. Immersive headset users will also see the cursor of the HoloLens user's gaze ray if the HoloLens user is looking at the island. HoloLens users will see an avatar on the island to represent each immersive headset user.

Updated Input for the Immersive device:

- The left bumper and right bumper buttons on the Xbox controller rotate the player
- Holding the Y button on the Xbox controller will enable a [teleport](#) cursor. If the cursor has a spinning arrow indicator when you release the Y button, you will be teleported to the cursor's location.

Steps

- Add MixedRealityTeleport to MixedRealityCameraParent
 - In **Hierarchy**, select **Usland**.
 - In **Inspector**, enable **Level Control**.
 - In **Hierarchy**, select **MixedRealityCameraParent**.
 - In **Inspector**, click **Add Component**.
 - Type **Mixed Reality Teleport** and select it.

Digging into the code

Immersive headset users will be tethered to their PCs with a cable, but our island is larger than the cable is long. To compensate, we need the ability to move the camera independently of the user's motion. Please see the [comfort page](#) for more information about designing your mixed reality application (in particular self motion and locomotion).

In order to describe this process it will be useful to define two terms. First, **dolly** will be the object that moves the camera independently from the user. A child game object of the **dolly** will be the **main camera**. The main camera is attached to the user's head.

In the project panel, navigate to **Assets\AppPrefabs\Support\Scripts\GameLogic** and double-click on **MixedRealityTeleport.cs**.

MixedRealityTeleport has two jobs. First, it handles rotation using the bumpers. In the update function we poll for 'ButtonUp' on LeftBumper and RightBumper. GetButtonUp only returns true on the first frame a button is up after having been down. If either button had been raised, then we know the user needs to rotate.

When we rotate we do a fade out and fade in using a simple script called 'fade control'. We do this to prevent the user from seeing an unnatural movement which could lead to discomfort. The fade in and out effect is fairly simple. We have a black quad hanging in front of the **main camera**. When fading out we transition the alpha value from 0 to 1. This gradually causes the black pixels of the quad to render and obscure anything behind them. When fading back in we transition the alpha value back to zero.

When we calculate the rotation, note that we are rotating our **dolly** but calculating the rotation around the **main camera**. This is important as the farther the **main camera** is away from 0,0,0, the less accurate a rotation around the dolly would become from the point of view of the user. In fact, if you do not rotate around the camera position, the user will move on an arc around the **dolly** rather than rotating.

The second job for MixedRealityTeleport is to handle moving the **dolly**. This is done in SetWorldPosition. SetWorldPosition takes the desired world position, the position where the user wants to perceive that they inhabit. We need to put our **dolly** at that position minus the local position of the **main camera**, as that offset will be added each frame.

A second script calls SetWorldPosition. Let's look at that script. In the project panel, navigate to **Assets\AppPrefabs\Support\Scripts\GameLogic** and double-click on **TeleportScript.cs**.

This script is a little more involved than MixedRealityTeleport. The script is checking for the Y Button on the Xbox controller to be held down. While the button is held down a teleport cursor is rendered and the script casts a ray from the user's gaze position. If that ray collides with a surface that is more or less pointing up, the surface will be considered a good surface to teleport to, and the animation on the teleport cursor will be enabled. If the ray does not collide with a surface more or less pointing up, then the animation on the cursor will be disabled. When the Y button is released and the calculated point of the ray is a valid position, the script calls SetWorldPosition with the position the ray intersected.

Enjoy your progress

This time you'll need to find a friend.

Once again, a user with the HoloLens will host a session. Other users will join the session. The application will place the first three users to join from an immersive headset on one of the three paths on the island. Feel free to explore the island in this section.

Details to notice:

1. You can see faces in the clouds, which helps an immersed user see which direction a HoloLens user is looking.
2. The avatars on the island have necks that rotate. They won't follow what the user is doing in real reality (we don't have that information) but it makes for a nice experience.
3. If the HoloLens user is looking at the Island, the immersed users can see their cursor.
4. The clouds that represent the HoloLens users cast shadows.

Chapter 5 - Finale

Objectives

Create a collaborative interactive experience between the two device types.

What we will build

Building on chapter 4, when a user with an immersive headset gets near a puzzle on the island, the HoloLens users will get a tool tip with a clue to the puzzle. Once all of the immersive headset users get past their puzzles and onto the "ready pad" in the rocket room, the rocket will launch.

Steps

- In **Hierarchy**, select **Usland**.
- In **Inspector**, in **Level Control**, check **Enable Collaboration**.

Digging into the code

Now let us look at LevelControl.cs. This script is the core of the game logic and maintains the game state. Since this is a multiplayer game using UNET we need to understand how data flows, at least well enough to modify this

tutorial. For a more complete overview of UNET, please refer to Unity's documentation.

In the project panel, navigate to **Assets\AppPrefabs\Support\Scripts\GameLogic** and double-click on **LevelControl.cs**.

Let us understand how an immersive headset indicates that they are ready for the rocket launch. Rocket Launch readiness is communicated by setting one of three bools in a list of bools that correspond to the three paths on the island. A path's bool will be set when the user assigned to the path is on top of the brown pad inside the rocket room. Okay, now to the details.

We will start in the Update() function. You will note that there is a 'cheat' function. We used this in development to test the rocket launch and reset sequence. It won't work in the multi user experience. Hopefully by the time you internalize the following information you can make it work. After we check to see if we should cheat, we check to see if the local player is immersed. We want to focus on how we find that we're at the goal. Inside of the if (Immersed) check, there is a call to CheckGoal hiding behind the **EnableCollaboration** bool. This corresponds to the checkbox you checked while completing the steps for this chapter. Inside of EnableCollaboration we see a call to CheckGoal().

CheckGoal does some math to see if we are more or less standing on the pad. When we are, we Debug.Log "Arrived at goal" and then we call 'SendAtGoalMessage()'. In SendAtGoalMessage we call playerController.SendAtGoal. To save you some time, here's the code:

```
private void CmdSendAtGoal(int GoalIndex)
{
    levelState.SetGoalIndex(GoalIndex);
}
```

```
public void SendAtGoal(int GoalIndex)
{
    if (isLocalPlayer)
    {
        Debug.Log("sending at goal " + GoalIndex);
        CmdSendAtGoal(GoalIndex);
    }
}
```

Note that SendAtGoalMessage calls CmdSendAtGoal, which calls levelState.SetGoalIndex, which is back in LevelControl.cs. At first glance this seems strange. Why not just call SetGoalIndex rather than this strange routing through the player controller? The reason is that we are conforming to the data model UNET uses to keep data in sync. To prevent cheating and thrashing, UNET requires that each object has a user who has authority to change the synchronized variables. Further, only the host (the user that started the session) can change data directly. Users who are not the host, but have authority, need to send a 'command' to the host which will change the variable. By default the host has authority over all objects, except for the object spawned to represent the user. In our case this object has the playercontroller script. There is a way to request authority for an object and then make changes, but we choose to leverage the fact that the player controller has self authority and route commands through the player controller.

Said another way, when we've found ourselves at our goal, the player needs to tell the host, and the host will tell everyone else.

Back in LevelControl.cs look at SetGoalIndex. Here we are setting the value of a value in a synclist (AtGoal). Remember that we are in the context of the host while we do this. Similar to a command, an RPC is something the host can issue that will cause all clients to run some code. Here we call 'RpcCheckAllGoals'. Each client will individually check to see if all three AtGoals are set, and if so, launch the rocket.

Enjoy your progress

Building on the previous chapter, we will start the session as before. This time as the users in the immersive headset get to the "door" on their path, a tooltip will appear that only the HoloLens users can see. The HoloLens users are responsible for communicating this clue to the users in the immersive headset. The rocket will launch to space once each avatar has stepped on its corresponding brown pad inside the volcano. The scene will reset after 60 seconds so you can do it again.

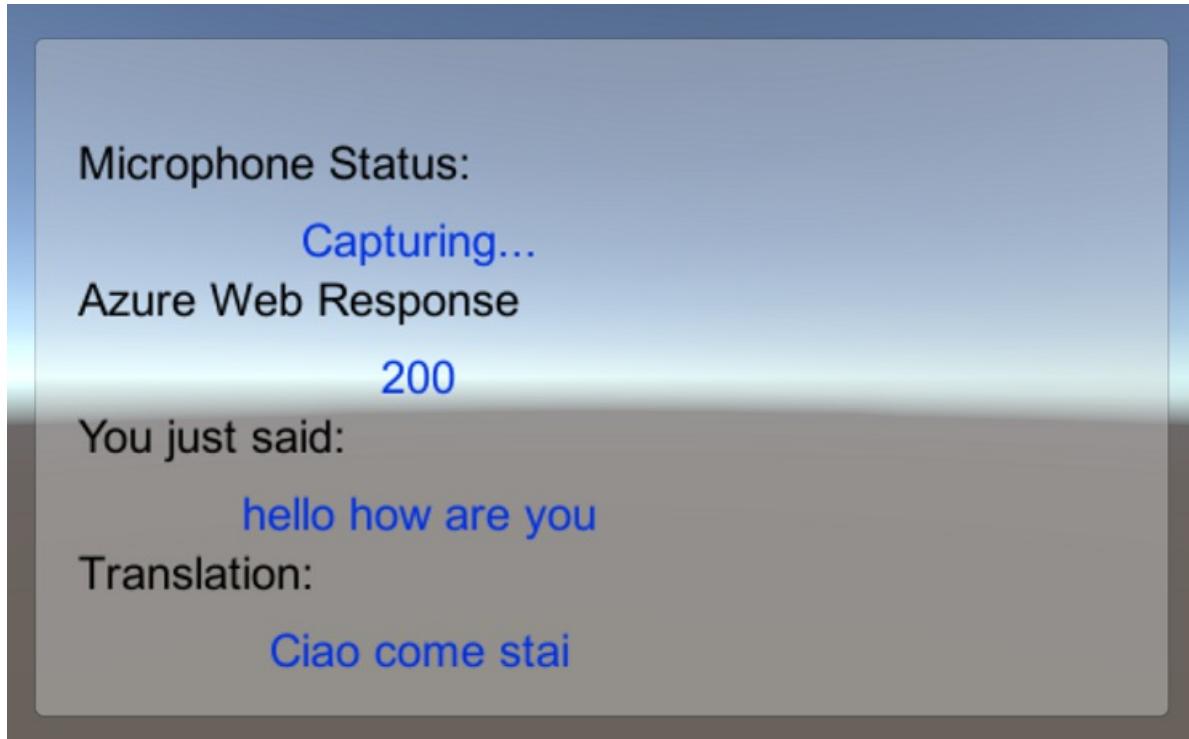
See also

- [MR Input 213: Motion controllers](#)

MR and Azure 301: Language translation

11/6/2018 • 23 minutes to read • [Edit Online](#)

In this course, you will learn how to add translation capabilities to a mixed reality application using Azure Cognitive Services, with the Translator Text API.



The Translator Text API is a translation Service which works in near real-time. The Service is cloud-based, and, using a REST API call, an app can make use of the neural machine translation technology to translate text to another language. For more information, visit the [Azure Translator Text API page](#).

Upon completion of this course, you will have a mixed reality application which will be able to do the following:

1. The user will speak into a microphone connected to an immersive (VR) headset (or the built-in microphone of HoloLens).
2. The app will capture the dictation and send it to the Azure Translator Text API.
3. The translation result will be displayed in a simple UI group in the Unity Scene.

This course will teach you how to get the results from the Translator Service into a Unity-based sample application. It will be up to you to apply these concepts to a custom application you might be building.

Device support

COURSE	HOLOLENS	IMMERSIVE HEADSETS
MR and Azure 301: Language translation	✓ <input type="checkbox"/>	✓ <input type="checkbox"/>

NOTE

While this course primarily focuses on Windows Mixed Reality immersive (VR) headsets, you can also apply what you learn in this course to Microsoft HoloLens. As you follow along with the course, you will see notes on any changes you might need to employ to support HoloLens. When using HoloLens, you may notice some echo during voice capture.

Prerequisites

NOTE

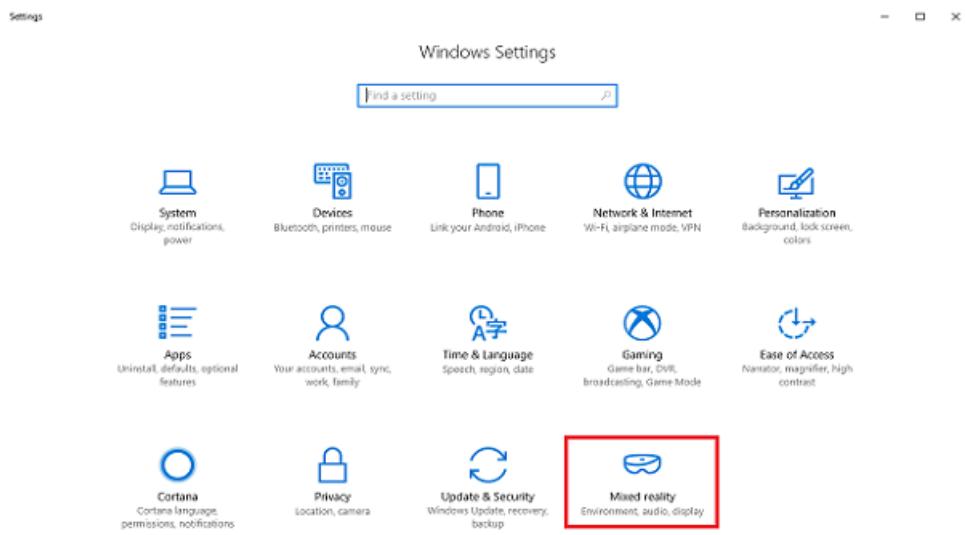
This tutorial is designed for developers who have basic experience with Unity and C#. Please also be aware that the prerequisites and written instructions within this document represent what has been tested and verified at the time of writing (May 2018). You are free to use the latest software, as listed within the [install the tools](#) article, though it should not be assumed that the information in this course will perfectly match what you'll find in newer software than what's listed below.

We recommend the following hardware and software for this course:

- A development PC, [compatible with Windows Mixed Reality](#) for immersive (VR) headset development
- [Windows 10 Fall Creators Update \(or later\)](#) with Developer mode enabled
- [The latest Windows 10 SDK](#)
- [Unity 2017.4](#)
- [Visual Studio 2017](#)
- A [Windows Mixed Reality immersive \(VR\) headset](#) or [Microsoft HoloLens](#) with Developer mode enabled
- A set of headphones with a built-in microphone (if the headset doesn't have a built-in mic and speakers)
- Internet access for Azure setup and translation retrieval

Before you start

- To avoid encountering issues building this project, it is strongly suggested that you create the project mentioned in this tutorial in a root or near-root folder (long folder paths can cause issues at build-time).
- The code in this tutorial will allow you to record from the default microphone device connected to your PC. Make sure the default microphone device is set to the device you plan to use to capture your voice.
- To allow your PC to enable dictation, go to **Settings > Privacy > Speech, inking & typing** and select the button **Turn On speech services and typing suggestions**.
- If you're using a microphone and headphones connected to (or built-in to) your headset, make sure the option "When I wear my headset, switch to headset mic" is turned on in **Settings > Mixed reality > Audio and speech**.



Home

Find a setting

- Mixed reality
- Audio and speech**
- Environment
- Headset display
- Uninstall

Audio and speech

Audio

Windows Mixed Reality works best with your device's built-in microphone and speakers, or with a headset connected to its audio port.

[Learn more](#)

When I wear my headset, switch to headset audio On

When I wear my headset, switch to headset mic On

Speech

Use speech recognition in Windows Mixed Reality. Speech recognition will always listen when Mixed Reality is running.

[Learn more](#)

WARNING

Be aware that if you are developing for an immersive headset for this lab, you may experience audio output device issues. This is due to an issue with Unity, which is fixed in later versions of Unity (Unity 2018.2). The issue prevents Unity from changing the default audio output device at run time. As a work around, ensure you have completed the above steps, and close and re-open the Editor, when this issue presents itself.

Chapter 1 – The Azure Portal

To use the Azure Translator API, you will need to configure an instance of the Service to be made available to your application.

1. Log in to the [Azure Portal](#).

NOTE

If you do not already have an Azure account, you will need to create one. If you are following this tutorial in a classroom or lab situation, ask your instructor or one of the proctors for help setting up your new account.

2. Once you are logged in, click on **New** in the top left corner and search for "Translator Text API." Select **Enter**.

The screenshot shows the Microsoft Azure portal's 'New' blade. On the left, there's a sidebar with 'New' and 'Dashboard' buttons, both circled in red. The main area shows search results for 'Translator Text API'. Below the search bar, there are sections for 'Azure Marketplace' and 'Popular' services like Windows Server 2016 VM and Ubuntu Server 16.04 LTS VM.

NOTE

The word **New** may have been replaced with **Create a resource**, in newer portals.

3. The new page will provide a description of the *Translator Text API* Service. At the bottom left of this page, select the **Create** button, to create an association with this Service.

This screenshot shows the 'Translator Text API' service details page. It includes sections for 'Extend the reach of your applications', 'Automatically detect languages', 'Translate real-life conversation', 'Build customized translation systems', 'Crowd-source translation improvement', and 'Legal Notice'. At the bottom left, there is a prominent blue 'Create' button, which is circled in red.

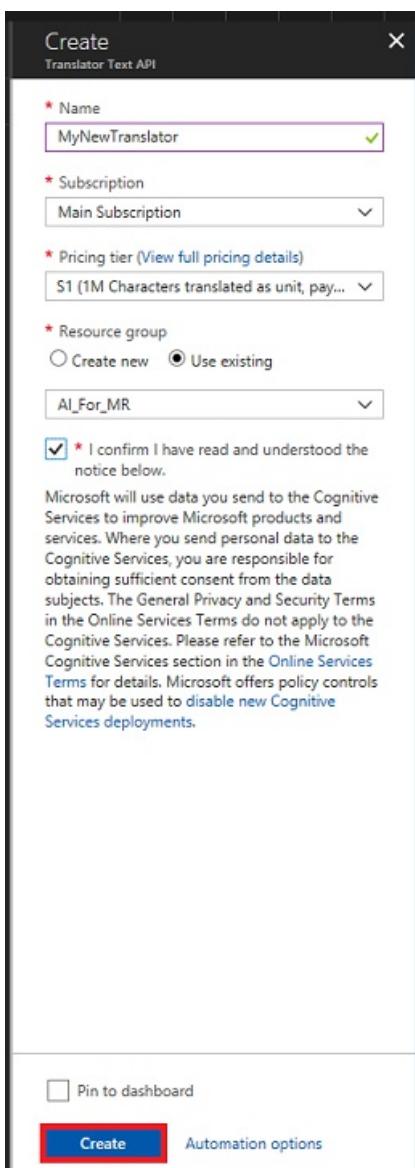
4. Once you have clicked on **Create**:

- a. Insert your desired **Name** for this Service instance.
- b. Select an appropriate **Subscription**.

- c. Select the **Pricing Tier** appropriate for you, if this is the first time creating a *Translator Text Service*, a free tier (named F0) should be available to you.
- d. Choose a **Resource Group** or create a new one. A resource group provides a way to monitor, control access, provision and manage billing for a collection of Azure assets. It is recommended to keep all the Azure Services associated with a single project (e.g. such as these labs) under a common resource group.

If you wish to read more about Azure Resource Groups, please [visit the resource group article](#).

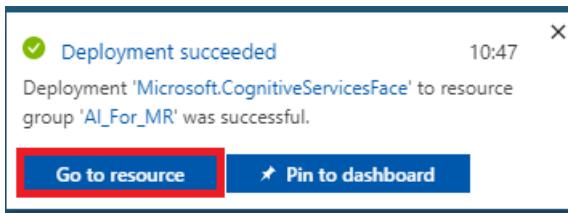
- e. Determine the **Location** for your resource group (if you are creating a new Resource Group). The location would ideally be in the region where the application would run. Some Azure assets are only available in certain regions.
- f. You will also need to confirm that you have understood the Terms and Conditions applied to this Service.
- g. Select **Create**.



- 5. Once you have clicked on **Create**, you will have to wait for the Service to be created, this might take a minute.
- 6. A notification will appear in the portal once the Service instance is created.



7. Click on the notification to explore your new Service instance.



8. Click the **Go to resource** button in the notification to explore your new Service instance. You will be taken to your new Translator Text API Service instance.

MyNewTranslator - Quick start

Congratulations! Your keys are ready.
Now explore the Quickstart guidance to get up and running with Translator Text API.

- 1 Grab your keys
Every call to Translator Text API requires a subscription key. This key needs to be either passed through a query string parameter or specified in the request header. You can find your keys in the API resource 'Overview' or 'Keys' from the left menu.
[Keys](#)
- 2 Make an API call
Get in-depth information about each properties and methods of the API. Test your keys with the built-in testing console without writing a single line of code.
[API reference](#)
- 3 Enjoy coding
Learn more about the features, tutorials, developer tools, examples and how-to guidance to speed up.
[Documentation](#)
[Code samples](#)

Additional resources

[Region availability](#)
[Support options](#)
[Provide feedback](#)

9. Within this tutorial, your application will need to make calls to your Service, which is done through using your Service's Subscription Key.
10. From the *Quick start* page of your *Translator Text* Service, navigate to the first step, *Grab your keys*, and click **Keys** (you can also achieve this by clicking the blue hyperlink Keys, located in the Services navigation menu, denoted by the key icon). This will reveal your Service Keys.
11. Take a copy of one of the displayed keys, as you will need this later in your project.

Chapter 2 – Set up the Unity project

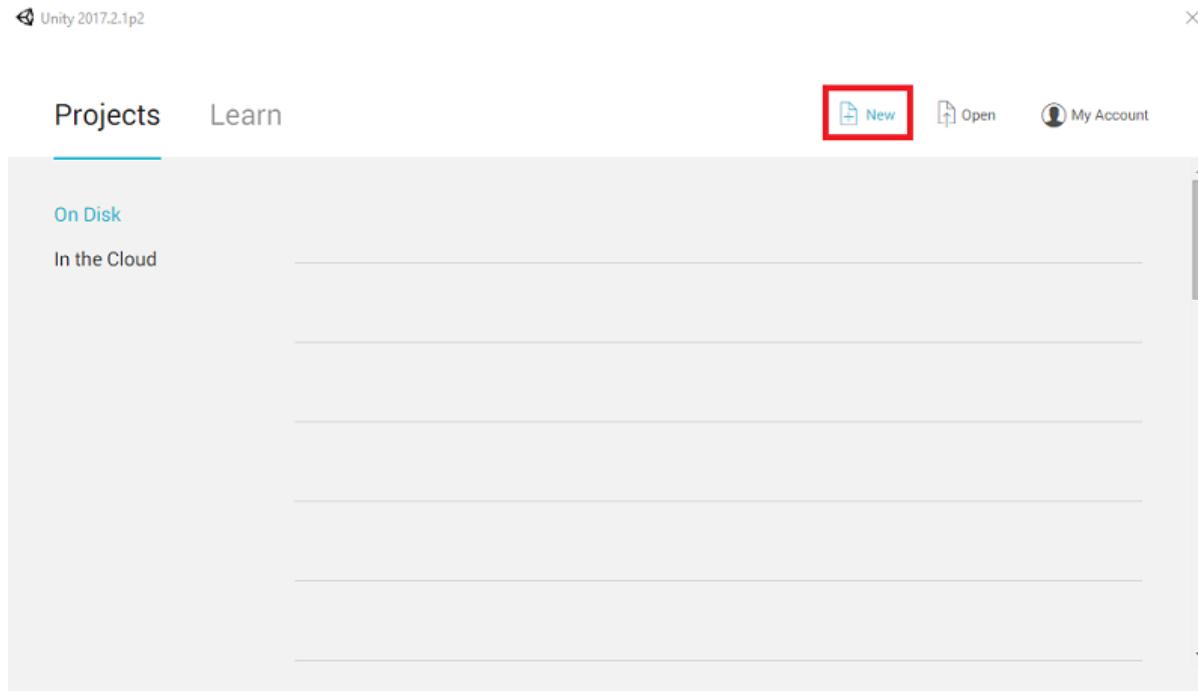
Set up and test your mixed reality immersive headset.

NOTE

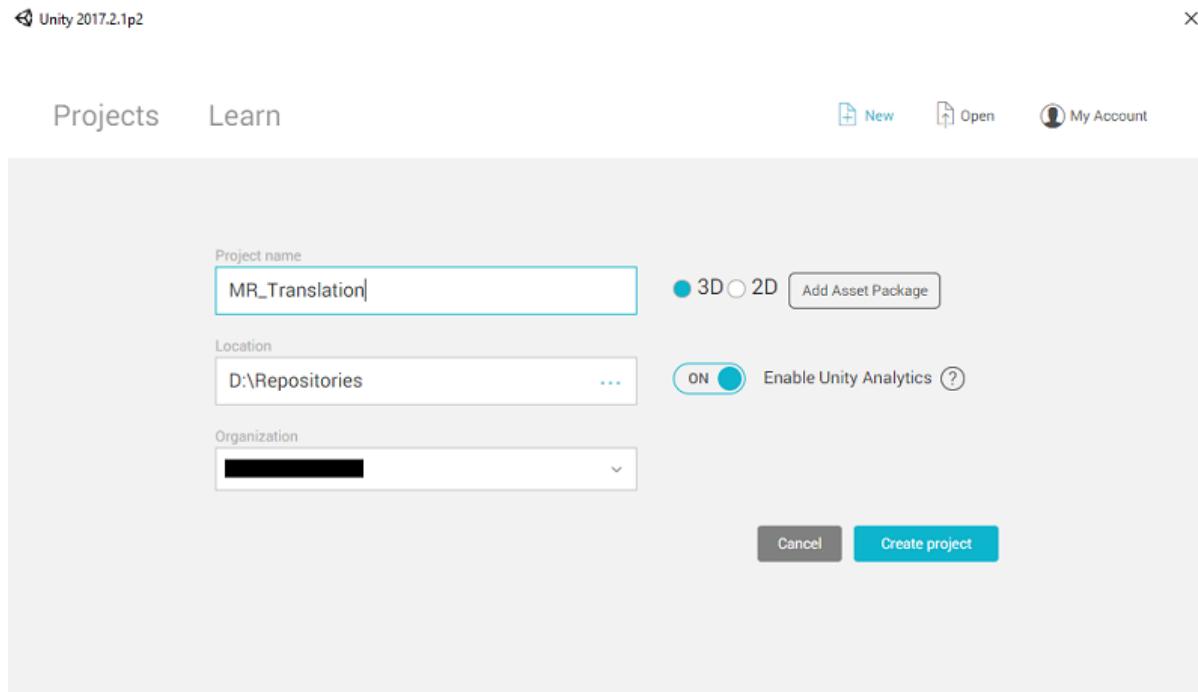
You will not need motion controllers for this course. If you need support setting up an immersive headset, please [follow these steps](#).

The following is a typical set up for developing with mixed reality and, as such, is a good template for other projects:

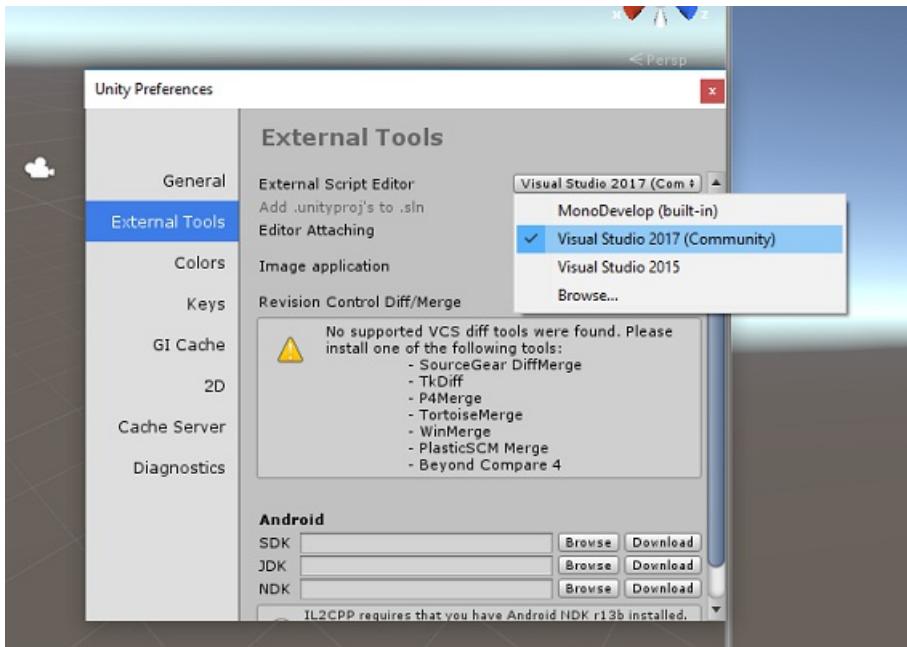
1. Open **Unity** and click **New**.



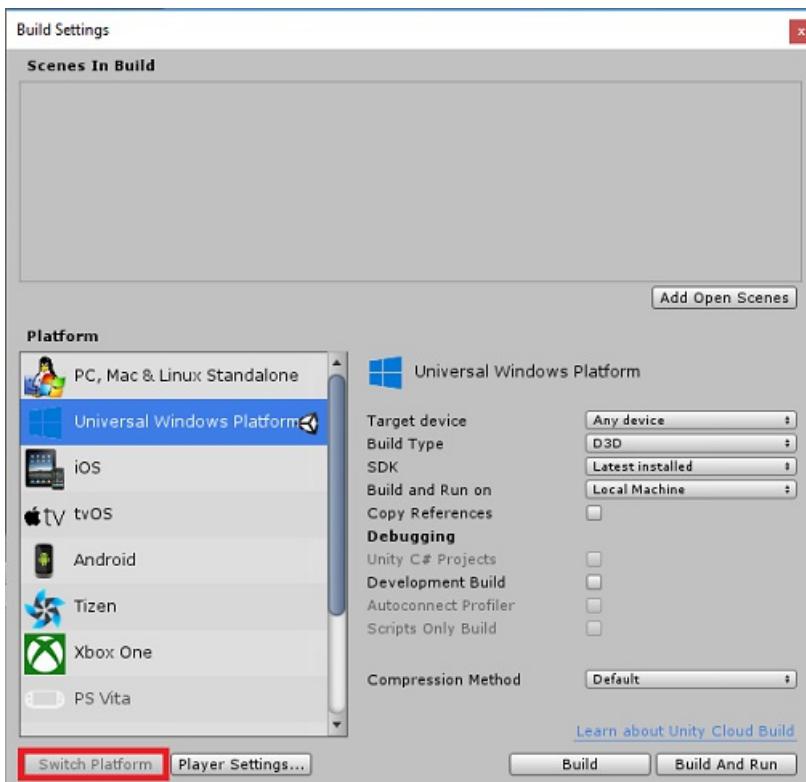
2. You will now need to provide a Unity Project name. Insert **MR_Translation**. Make sure the project type is set to **3D**. Set the *Location* to somewhere appropriate for you (remember, closer to root directories is better). Then, click **Create project**.



3. With Unity open, it is worth checking the default **Script Editor** is set to **Visual Studio**. Go to **Edit > Preferences** and then from the new window, navigate to **External Tools**. Change **External Script Editor** to **Visual Studio 2017**. Close the **Preferences** window.



- Next, go to **File > Build Settings** and switch the platform to **Universal Windows Platform**, by clicking on the **Switch Platform** button.



- Go to **File > Build Settings** and make sure that:

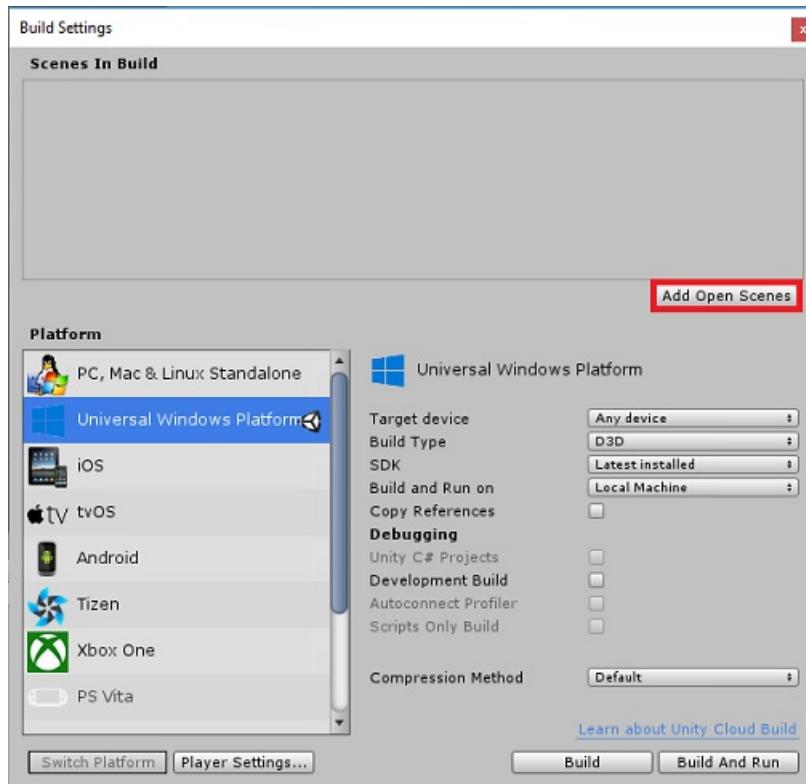
- Target Device** is set to **Any Device**.

For Microsoft HoloLens, set **Target Device** to *HoloLens*.

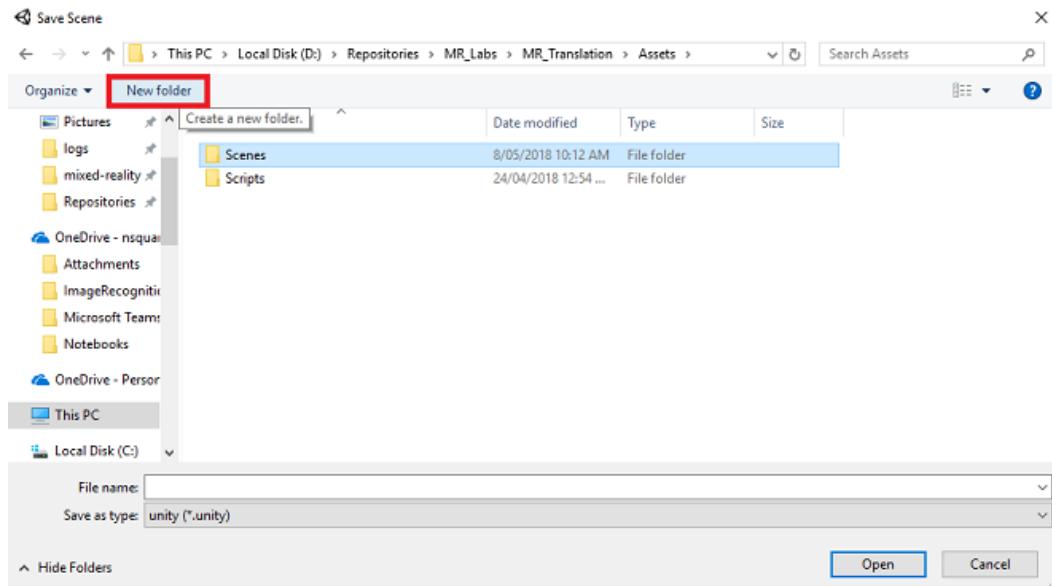
- Build Type** is set to **D3D**
- SDK** is set to **Latest installed**
- Visual Studio Version** is set to **Latest installed**
- Build and Run** is set to **Local Machine**

f. Save the scene and add it to the build.

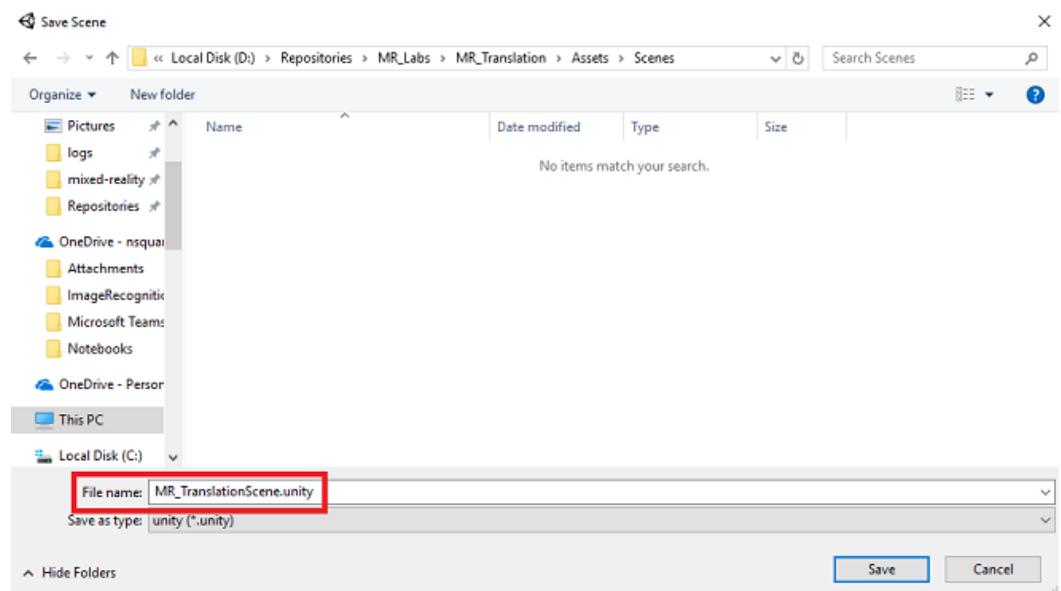
a. Do this by selecting **Add Open Scenes**. A save window will appear.



b. Create a new folder for this, and any future, scene, then select the **New folder** button, to create a new folder, name it **Scenes**.

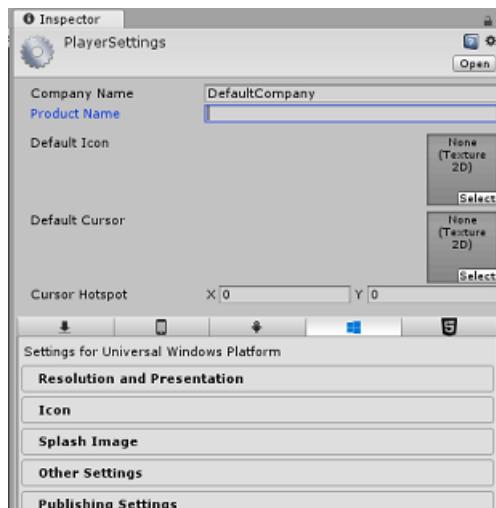


c. Open your newly created **Scenes** folder, and then in the *File name:* text field, type **MR_TranslationScene**, then press **Save**.

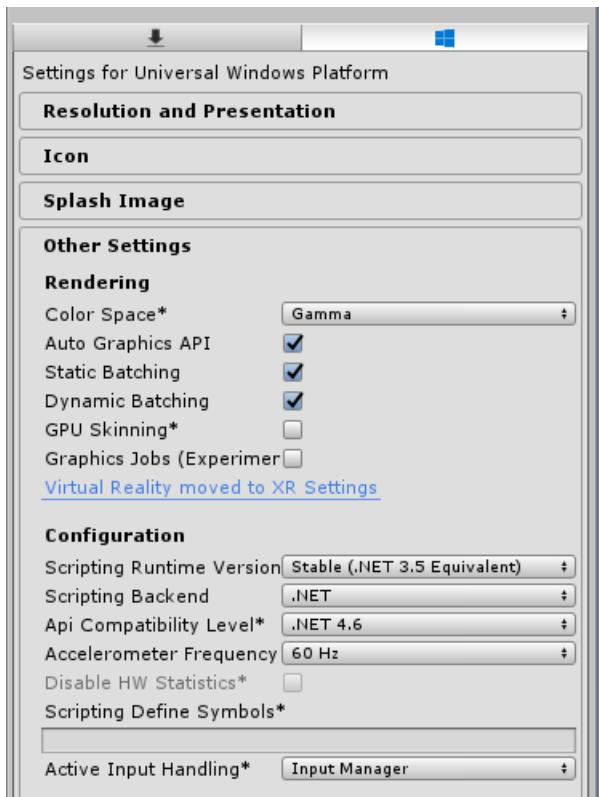


Be aware, you must save your Unity scenes within the *Assets* folder, as they must be associated with the Unity Project. Creating the scenes folder (and other similar folders) is a typical way of structuring a Unity project.

- g. The remaining settings, in *Build Settings*, should be left as default for now.
6. In the *Build Settings* window, click on the **Player Settings** button, this will open the related panel in the space where the *Inspector* is located.

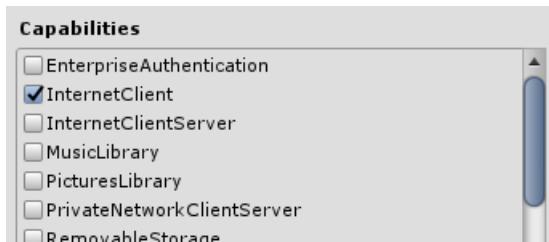


7. In this panel, a few settings need to be verified:
- In the **Other Settings** tab:
 - Scripting Runtime Version** should be **Stable (.NET 3.5 Equivalent)**.
 - Scripting Backend** should be **.NET**
 - API Compatibility Level** should be **.NET 4.6**

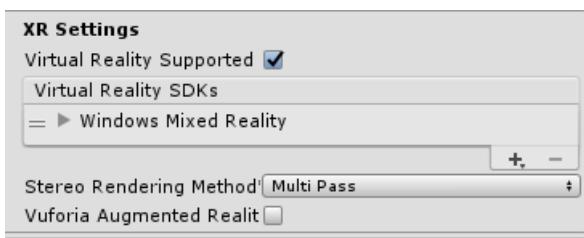


b. Within the **Publishing Settings** tab, under **Capabilities**, check:

- a. **InternetClient**
- b. **Microphone**



c. Further down the panel, in **XR Settings** (found below **Publish Settings**), tick **Virtual Reality Supported**, make sure the **Windows Mixed Reality SDK** is added.



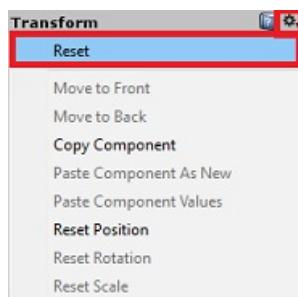
8. Back in **Build Settings**, *Unity C# Projects* is no longer greyed out; tick the checkbox next to this.
9. Close the Build Settings window.
10. Save your Scene and Project (**FILE > SAVE SCENE / FILE > SAVE PROJECT**).

Chapter 3 – Main Camera setup

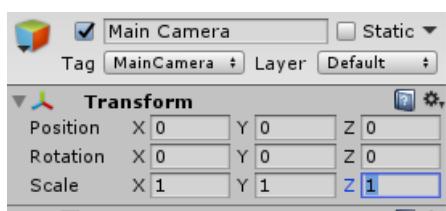
IMPORTANT

If you wish to skip the *Unity Set up* component of this course, and continue straight into code, feel free to [download this .unitypackage](#), import it into your project as a *Custom Package*, and then continue from [Chapter 5](#). You will still need to create a Unity Project.

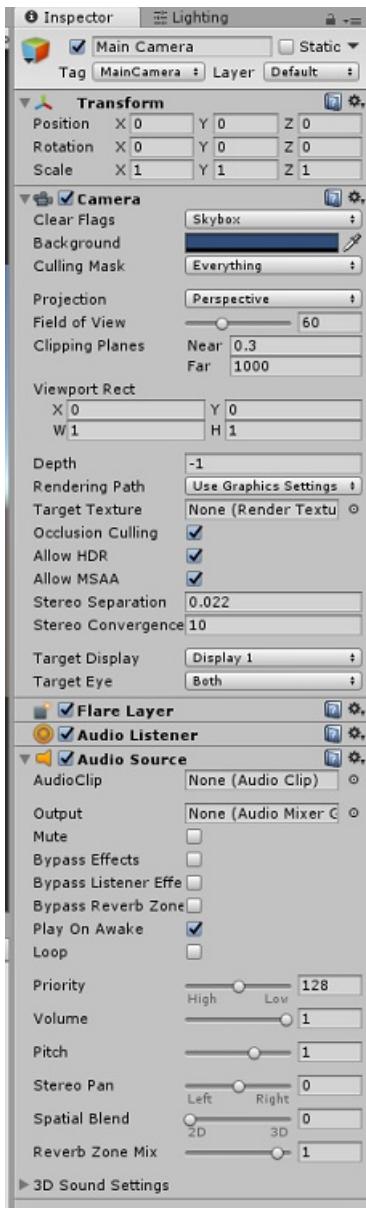
1. In the *Hierarchy Panel*, you will find an object called **Main Camera**, this object represents your "head" point of view once you are "inside" your application.
2. With the Unity Dashboard in front of you, select the **Main Camera GameObject**. You will notice that the *Inspector Panel* (generally found to the right, within the Dashboard) will show the various components of that *GameObject*, with *Transform* at the top, followed by *Camera*, and some other components. You will need to reset the *Transform* of the Main Camera, so it is positioned correctly.
3. To do this, select the **Gear** icon next to the Camera's *Transform* component, and selecting **Reset**.



4. The *Transform* component should then look like:
 - a. The *Position* is set to **0, 0, 0**
 - b. *Rotation* is set to **0, 0, 0**
 - c. And *Scale* is set to **1, 1, 1**



5. Next, with the **Main Camera** object selected, see the **Add Component** button located at the very bottom of the *Inspector Panel*.
6. Select that button, and search (by either typing *Audio Source* into the search field or navigating the sections) for the component called **Audio Source** as shown below and select it (pressing enter on it also works).
7. An *Audio Source* component will be added to the **Main Camera**, as demonstrated below.



NOTE

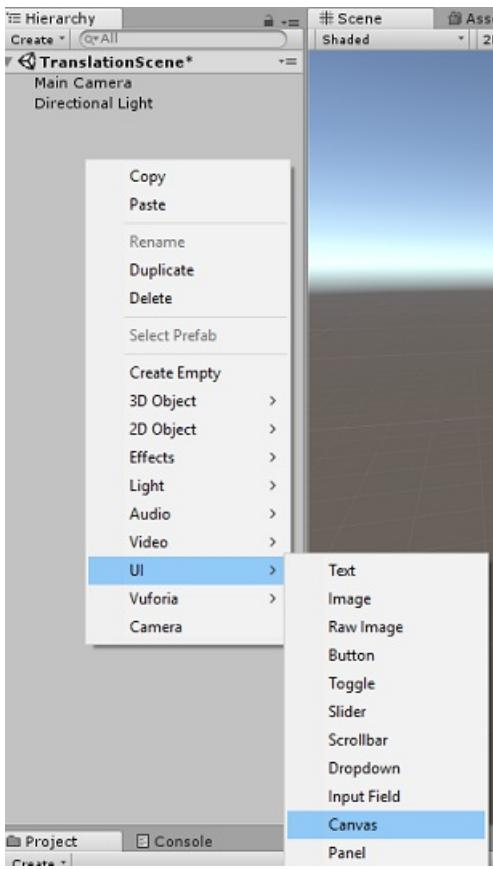
For Microsoft HoloLens, you will need to also change the following, which are part of the **Camera** component on your **Main Camera**:

- **Clear Flags:** Solid Color.
- **Background** 'Black, Alpha 0' – Hex color: #00000000.

Chapter 4 – Setup Debug Canvas

To show the input and output of the translation, a basic UI needs to be created. For this course, you will create a Canvas UI object, with several 'Text' objects to show the data.

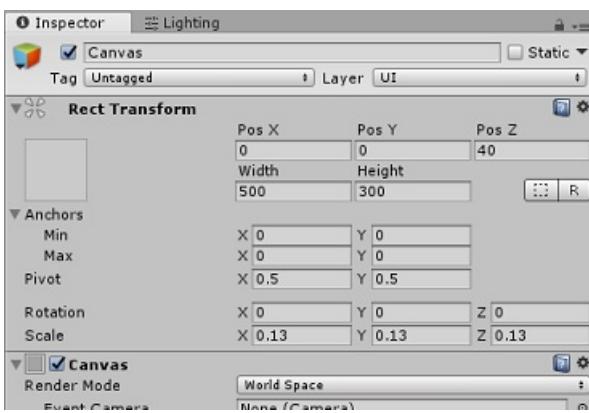
1. Right-click in an empty area of the *Hierarchy Panel*, under **UI**, add a **Canvas**.



2. With the **Canvas** object selected, in the *Inspector Panel* (within the 'Canvas' component), change **Render Mode** to **World Space**.

3. Next, change the following parameters in the *Inspector Panel's Rect Transform*:

- POS - X 0 Y 0 Z 40*
- Width - 500*
- Height - 300*
- Scale - X 0.13 Y 0.13 Z 0.13*

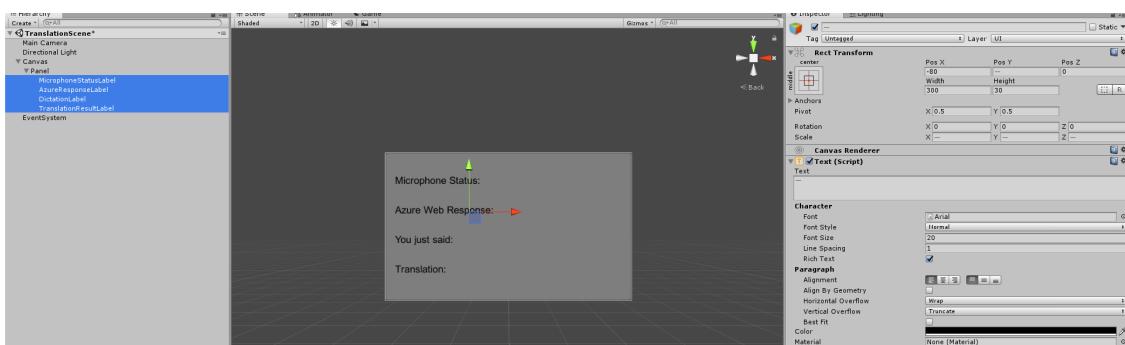


- Right click on the **Canvas** in the *Hierarchy Panel*, under **UI**, and add a **Panel**. This **Panel** will provide a background to the text that you will be displaying in the scene.
- Right click on the **Panel** in the *Hierarchy Panel*, under **UI**, and add a **Text object**. Repeat the same process until you have created four UI Text objects in total (Hint: if you have the first 'Text' object selected, you can simply press '**Ctrl**' + '**D**', to duplicate it, until you have four in total).
- For each **Text Object**, select it and use the below tables to set the parameters in the *Inspector Panel*.
 - For the *Rect Transform* component:

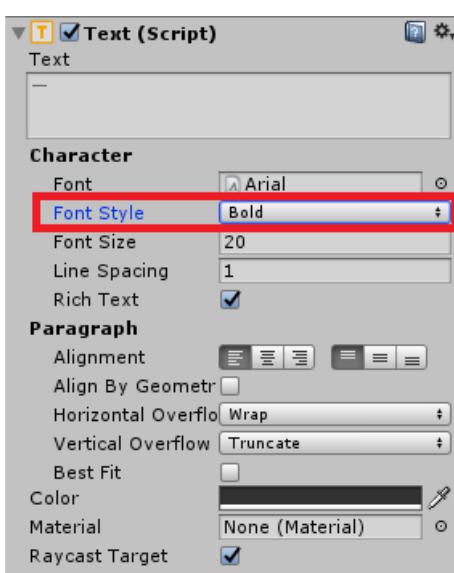
NAME	TRANSFORM - POSITION	WIDTH	HEIGHT
MicrophoneStatusLabel	X -80 Y 90 Z 0	300	30
AzureResponseLabel	X -80 Y 30 Z 0	300	30
DictationLabel	X -80 Y -30 Z 0	300	30
TranslationResultLabel	X -80 Y -90 Z 0	300	30

b. For the **Text (Script)** component:

NAME	TEXT	FONTSIZE
MicrophoneStatusLabel	Microphone Status:	20
AzureResponseLabel	Azure Web Response	20
DictationLabel	You just said:	20
TranslationResultLabel	Translation:	20



c. Also, make the Font Style **Bold**. This will make the text easier to read.



- For each *UI Text object* created in [Chapter 5](#), create a new *child UI Text object*. These children will display the output of the application. Create *child* objects through right-clicking your intended parent (e.g. *MicrophoneStatusLabel*) and then select **UI** and then select **Text**.
- For each of these children, select it and use the below tables to set the parameters in the Inspector Panel.

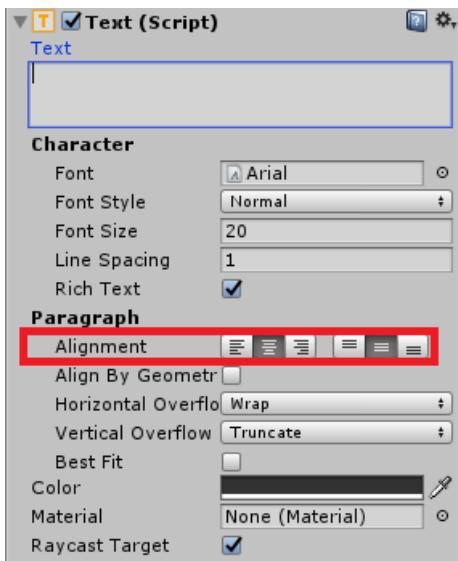
- a. For the **Rect Transform** component:

NAME	TRANSFORM - POSITION	WIDTH	HEIGHT
MicrophoneStatusText	X 0 Y -30 Z 0	300	30
AzureResponseText	X 0 Y -30 Z 0	300	30
DictationText	X 0 Y -30 Z 0	300	30
TranslationResultText	X 0 Y -30 Z 0	300	30

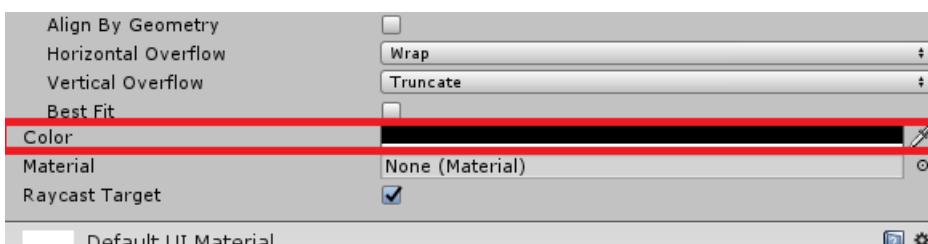
- b. For the **Text (Script)** component:

NAME	TEXT	FONT SIZE
MicrophoneStatusText	??	20
AzureResponseText	??	20
DictationText	??	20
TranslationResultText	??	20

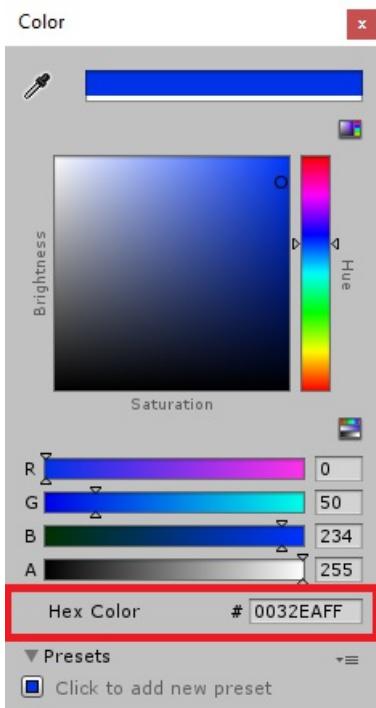
9. Next, select the 'centre' alignment option for each text component:



10. To ensure the **child UI Text** objects are easily readable, change their *Color*. Do this by clicking on the bar (currently 'Black') next to *Color*.

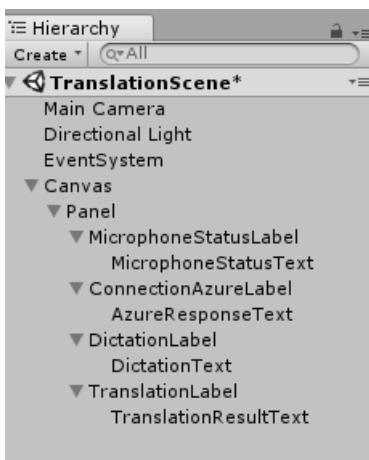


11. Then, in the new, little, *Color* window, change the *Hex Color* to: **0032EAFF**

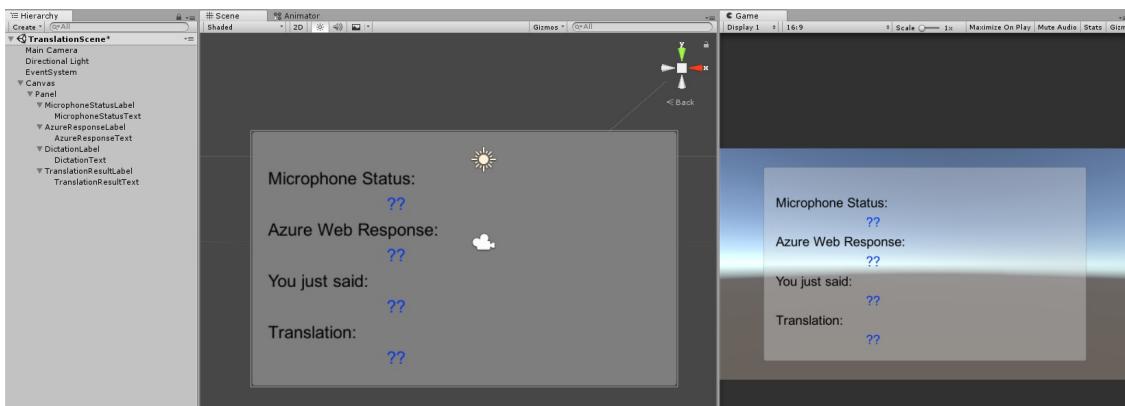


12. Below is how the **UI** should look.

a. In the *Hierarchy Panel*:



b. In the *Scene* and *Game Views*:



Chapter 5 – Create the Results class

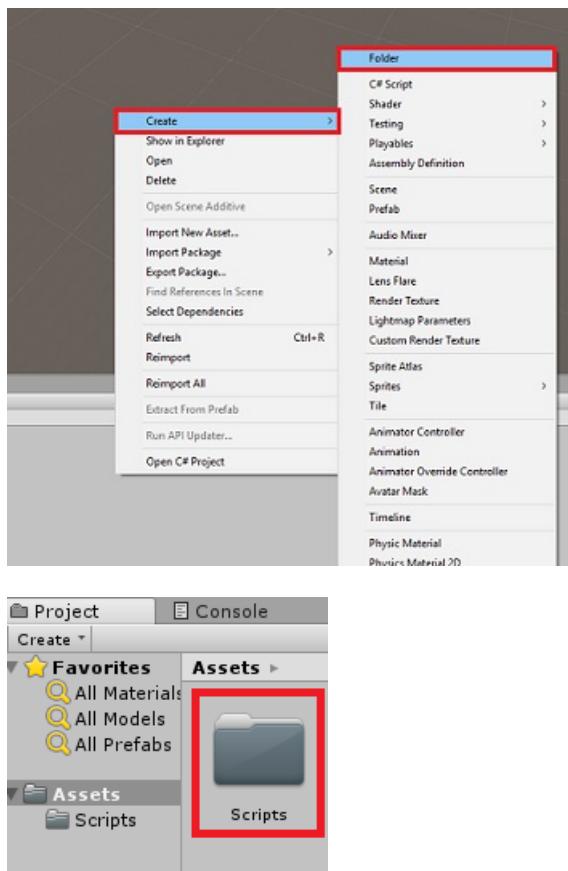
The first script you need to create is the *Results* class, which is responsible for providing a way to see the results of translation. The Class stores and displays the following:

- The response result from Azure.

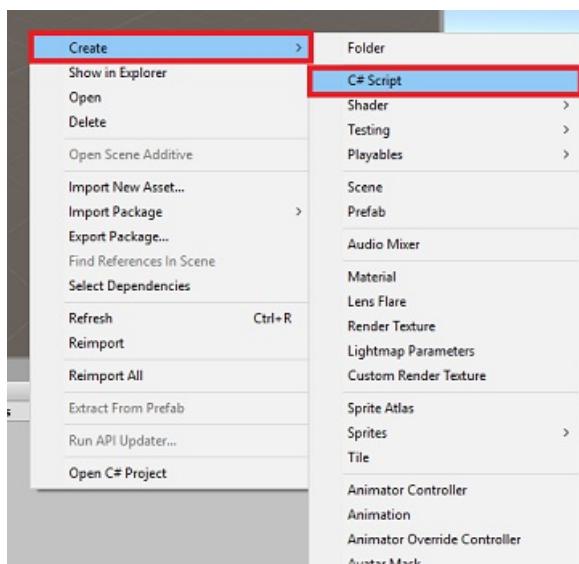
- The microphone status.
- The result of the dictation (voice to text).
- The result of the translation.

To create this class:

1. Right-click in the **Project Panel**, then **Create > Folder**. Name the folder **Scripts**.



2. With the **Scripts** folder created, double click it to open. Then within that folder, right-click, and select **Create > C# Script**. Name the script **Results**.



3. Double click on the new **Results** script to open it with **Visual Studio**.
4. Insert the following namespaces:

```
using UnityEngine;
using UnityEngine.UI;
```

5. Inside the Class insert the following variables:

```
public static Results instance;

[HideInInspector]
public string azureResponseCode;

[HideInInspector]
public string translationResult;

[HideInInspector]
public string dictationResult;

[HideInInspector]
public string micStatus;

public Text microphoneStatusText;

public Text azureResponseText;

public Text dictationText;

public Text translationResultText;
```

6. Then add the *Awake()* method, which will be called when the class initializes.

```
private void Awake()
{
    // Set this class to behave similar to singleton
    instance = this;
}
```

7. Finally, add the methods which are responsible for outputting the various results information to the UI.

```

/// <summary>
/// Stores the Azure response value in the static instance of Result class.
/// </summary>
public void SetAzureResponse(string result)
{
    azureResponseCode = result;
    azureResponseText.text = azureResponseCode;
}

/// <summary>
/// Stores the translated result from dictation in the static instance of Result class.
/// </summary>
public void SetDictationResult(string result)
{
    dictationResult = result;
    dictationText.text = dictationResult;
}

/// <summary>
/// Stores the translated result from Azure Service in the static instance of Result class.
/// </summary>
public void SetTranslatedResult(string result)
{
    translationResult = result;
    translationResultText.text = translationResult;
}

/// <summary>
/// Stores the status of the Microphone in the static instance of Result class.
/// </summary>
public void SetMicrophoneStatus(string result)
{
    micStatus = result;
    microphoneStatusText.text = micStatus;
}

```

- Be sure to save your changes in *Visual Studio* before returning to *Unity*.

Chapter 6 – Create the *MicrophoneManager* class

The second class you are going to create is the *MicrophoneManager*.

This class is responsible for:

- Detecting the recording device attached to the headset or machine (whichever is the default).
- Capture the audio (voice) and use dictation to store it as a string.
- Once the voice has paused, submit the dictation to the Translator class.
- Host a method that can stop the voice capture if desired.

To create this class:

1. Double click on the **Scripts** folder, to open it.
2. Right-click inside the **Scripts** folder, click **Create > C# Script**. Name the script *MicrophoneManager*.
3. Double click on the new script to open it with Visual Studio.
4. Update the namespaces to be the same as the following, at the top of the *MicrophoneManager* class:

```

using UnityEngine;
using UnityEngine.Windows.Speech;

```

5. Then, add the following variables inside the *MicrophoneManager* class:

```
// Help to access instance of this object
public static MicrophoneManager instance;

// AudioSource component, provides access to mic
private AudioSource audioSource;

// Flag indicating mic detection
private bool microphoneDetected;

// Component converting speech to text
private DictationRecognizer dictationRecognizer;
```

6. Code for the *Awake()* and *Start()* methods now needs to be added. These will be called when the class initializes:

```
private void Awake()
{
    // Set this class to behave similar to singleton
    instance = this;
}

void Start()
{
    //Use Unity Microphone class to detect devices and setup AudioSource
    if(Microphone.devices.Length > 0)
    {
        Results.instance.SetMicrophoneStatus("Initialising...");
        audioSource = GetComponent<AudioSource>();
        microphoneDetected = true;
    }
    else
    {
        Results.instance.SetMicrophoneStatus("No Microphone detected");
    }
}
```

7. You can *delete* the *Update()* method since this class will not use it.
8. Now you need the methods that the App uses to start and stop the voice capture, and pass it to the *Translator* class, that you will build soon. Copy the following code and paste it beneath the *Start()* method.

```

/// <summary>
/// Start microphone capture. Debugging message is delivered to the Results class.
/// </summary>
public void StartCapturingAudio()
{
    if(microphoneDetected)
    {
        // Start dictation
        dictationRecognizer = new DictationRecognizer();
        dictationRecognizer.DictationResult += DictationRecognizer_DictationResult;
        dictationRecognizer.Start();

        // Update UI with mic status
        Results.instance.SetMicrophoneStatus("Capturing...");
    }
}

/// <summary>
/// Stop microphone capture. Debugging message is delivered to the Results class.
/// </summary>
public void StopCapturingAudio()
{
    Results.instance.SetMicrophoneStatus("Mic sleeping");
    Microphone.End(null);
    dictationRecognizer.DictationResult -= DictationRecognizer_DictationResult;
    dictationRecognizer.Dispose();
}

```

TIP

Though this application will not make use of it, the *StopCapturingAudio()* method has also been provided here, should you want to implement the ability to stop capturing audio in your application.

9. You now need to add a Dictation Handler that will be invoked when the voice stops. This method will then pass the dictated text to the *Translator* class.

```

/// <summary>
/// This handler is called every time the Dictation detects a pause in the speech.
/// Debugging message is delivered to the Results class.
/// </summary>
private void DictationRecognizer_DictationResult(string text, ConfidenceLevel confidence)
{
    // Update UI with dictation captured
    Results.instance.SetDictationResult(text);

    // Start the coroutine that process the dictation through Azure
    StartCoroutine(Translator.instance.TranslateWithUnityNetworking(text));
}

```

10. Be sure to save your changes in Visual Studio before returning to Unity.

WARNING

At this point you will notice an error appearing in the *Unity Editor Console Panel* ("The name 'Translator' does not exist..."). This is because the code references the *Translator* class, which you will create in the next chapter.

Chapter 7 – Call to Azure and translator service

The last script you need to create is the *Translator* class.

This class is responsible for:

- Authenticating the App with Azure, in exchange for an **Auth Token**.
- Use the **Auth Token** to submit text (received from the *MicrophoneManager Class*) to be translated.
- Receive the translated result and pass it to the *Results Class* to be visualized in the UI.

To create this Class:

1. Go to the **Scripts** folder you created previously.
2. Right-click in the **Project Panel**, **Create > C# Script**. Call the script *Translator*.
3. Double click on the new *Translator* script to open it **with Visual Studio**.
4. Add the following namespaces to the top of the file:

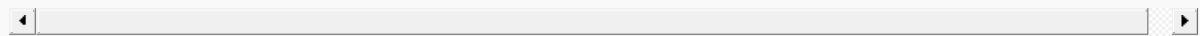
```
using System;
using System.Collections;
using System.Xml.Linq;
using UnityEngine;
using UnityEngine.Networking;
```

5. Then add the following variables inside the *Translator* class:

```
public static Translator instance;
private string translationTokenEndpoint = "https://api.cognitive.microsoft.com/sts/v1.0/issueToken";
private string translationTextEndpoint =
"https://api.microsofttranslator.com/v2/http.svc/Translate?";
private const string ocpApimSubscriptionKeyHeader = "Ocp-Apim-Subscription-Key";

//Substitute the value of authorizationKey with your own Key
private const string authorizationKey = "-InsertYourAuthKeyHere-";
private string authorizationToken;

// languages set below are:
// English
// French
// Italian
// Japanese
// Korean
public enum Languages { en, fr, it, ja, ko };
public Languages from = Languages.en;
public Languages to = Languages.it;
```



NOTE

- The languages inserted into the languages **enum** are just examples. Feel free to add more if you wish; the [API supports over 60 languages](#) (including Klingon)!
- There is a [more interactive page covering available languages](#), though be aware the page only appears to work when the site language is set to 'en-us' (and the Microsoft site will likely redirect to your native language). You can change site language at the bottom of the page or by altering the URL.
- The **authorizationKey** value, in the above code snippet, must be the **Key** you received when you subscribed to the *Azure Translator Text API*. This was covered in [Chapter 1](#).

6. Code for the *Awake()* and *Start()* methods now needs to be added.
7. In this case, the code will make a call to *Azure* using the authorization Key, to get a *Token*.

```

private void Awake()
{
    // Set this class to behave similar to singleton
    instance = this;
}

// Use this for initialization
void Start()
{
    // When the application starts, request an auth token
    StartCoroutine("GetTokenCoroutine", authorizationKey);
}

```

NOTE

The token will expire after 10 minutes. Depending on the scenario for your app, you might have to make the same coroutine call multiple times.

8. The coroutine to obtain the Token is the following:

```

/// <summary>
/// Request a Token from Azure Translation Service by providing the access key.
/// Debugging result is delivered to the Results class.
/// </summary>
private IEnumerator GetTokenCoroutine(string key)
{
    if (string.IsNullOrEmpty(key))
    {
        throw new InvalidOperationException("Authorization key not set.");
    }

    using (UnityWebRequest unityWebRequest = UnityWebRequest.Post(translationTokenEndpoint,
string.Empty))
    {
        unityWebRequest.SetRequestHeader("Ocp-Apim-Subscription-Key", key);
        yield return unityWebRequest.SendWebRequest();

        long responseCode = unityWebRequest.responseCode;

        // Update the UI with the response code
        Results.instance.SetAzureResponse(responseCode.ToString());

        if (unityWebRequest.isNetworkError || unityWebRequest.isHttpError)
        {
            Results.instance.azureResponseText.text = unityWebRequest.error;
            yield return null;
        }
        else
        {
            authorizationToken = unityWebRequest.downloadHandler.text;
        }
    }

    // After receiving the token, begin capturing Audio with the MicrophoneManager Class
    MicrophoneManager.instance.StartCapturingAudio();
}

```

WARNING

If you edit the name of the `IEnumerator` method `GetTokenCoroutine()`, you need to update the `StartCoroutine` and `StopCoroutine` call string values in the above code. As per [Unity documentation](#), to Stop a specific Coroutine, you need to use the string value method.

9. Next, add the coroutine (with a “support” stream method right below it) to obtain the translation of the text received by the `MicrophoneManager` class. This code creates a query string to send to the *Azure Translator Text API*, and then uses the internal Unity `UnityWebRequest` class to make a ‘Get’ call to the endpoint with the query string. The result is then used to set the translation in your `Results` object. The code below shows the implementation:

```
/// <summary>
/// Request a translation from Azure Translation Service by providing a string.
/// Debugging result is delivered to the Results class.
/// </summary>
public IEnumerator TranslateWithUnityNetworking(string text)
{
    // This query string will contain the parameters for the translation
    string queryString = string.Concat("text=", Uri.EscapeDataString(text), "&from=", from, "&to=",
    to);

    using (UnityWebRequest unityWebRequest = UnityWebRequest.Get(translationTextEndpoint +
    queryString))
    {
        unityWebRequest.SetRequestHeader("Authorization", "Bearer " + authorizationToken);
        unityWebRequest.SetRequestHeader("Accept", "application/xml");
        yield return unityWebRequest.SendWebRequest();

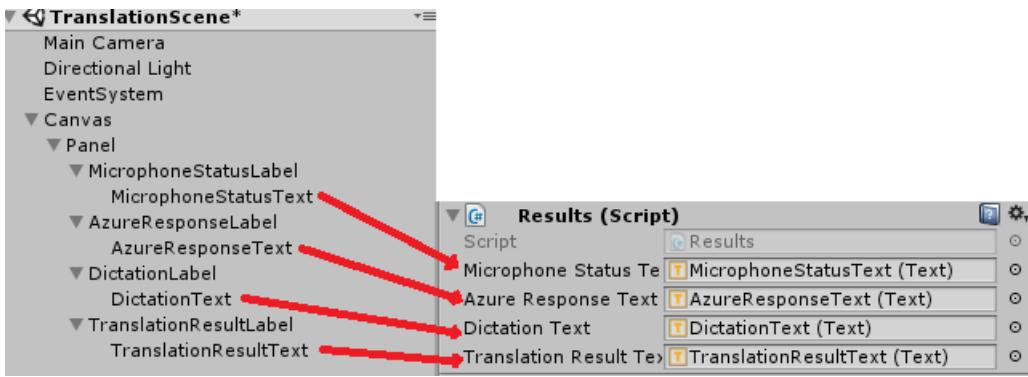
        if (unityWebRequest.isNetworkError || unityWebRequest.isHttpError)
        {
            Debug.Log(unityWebRequest.error);
            yield return null;
        }

        // Parse out the response text from the returned Xml
        string result = XElement.Parse(unityWebRequest.downloadHandler.text).Value;
        Results.instance.SetTranslatedResult(result);
    }
}
```

10. Be sure to save your changes in *Visual Studio* before returning to *Unity*.

Chapter 8 – Configure the Unity Scene

1. Back in the Unity Editor, click and drag the `Results` class from the **Scripts** folder to the **Main Camera** object in the *Hierarchy Panel*.
2. Click on the **Main Camera** and look at the *Inspector Panel*. You will notice that within the newly added `Script` component, there are four fields with empty values. These are the output references to the properties in the code.
3. Drag the appropriate **Text** objects from the *Hierarchy Panel* to those four slots, as shown in the image below.



4. Next, click and drag the *Translator* class from the **Scripts** folder to the **Main Camera** object in the *Hierarchy Panel*.
5. Then, click and drag the *MicrophoneManager* class from the **Scripts** folder to the **Main Camera** object in the *Hierarchy Panel*.
6. Lastly, click on the **Main Camera** and look at the *Inspector Panel*. You will notice that in the script you dragged on, there are two drop down boxes that will allow you to set the languages.



Chapter 9 – Test in mixed reality

At this point you need to test that the Scene has been properly implemented.

Ensure that:

- All the settings mentioned in [Chapter 1](#) are set correctly.
- The *Results*, *Translator*, and *MicrophoneManager*, scripts are attached to the **Main Camera** object.
- You have placed your *Azure Translator Text API Service Key* within the **authorizationKey** variable within the *Translator Script*.
- All the fields in the *Main Camera Inspector Panel* are assigned properly.
- Your microphone is working when running your scene (if not, check that your attached microphone is the *default* device, and that you have [set it up correctly within Windows](#)).

You can test the immersive headset by pressing the **Play** button in the *Unity Editor*. The App should be functioning through the attached immersive headset.

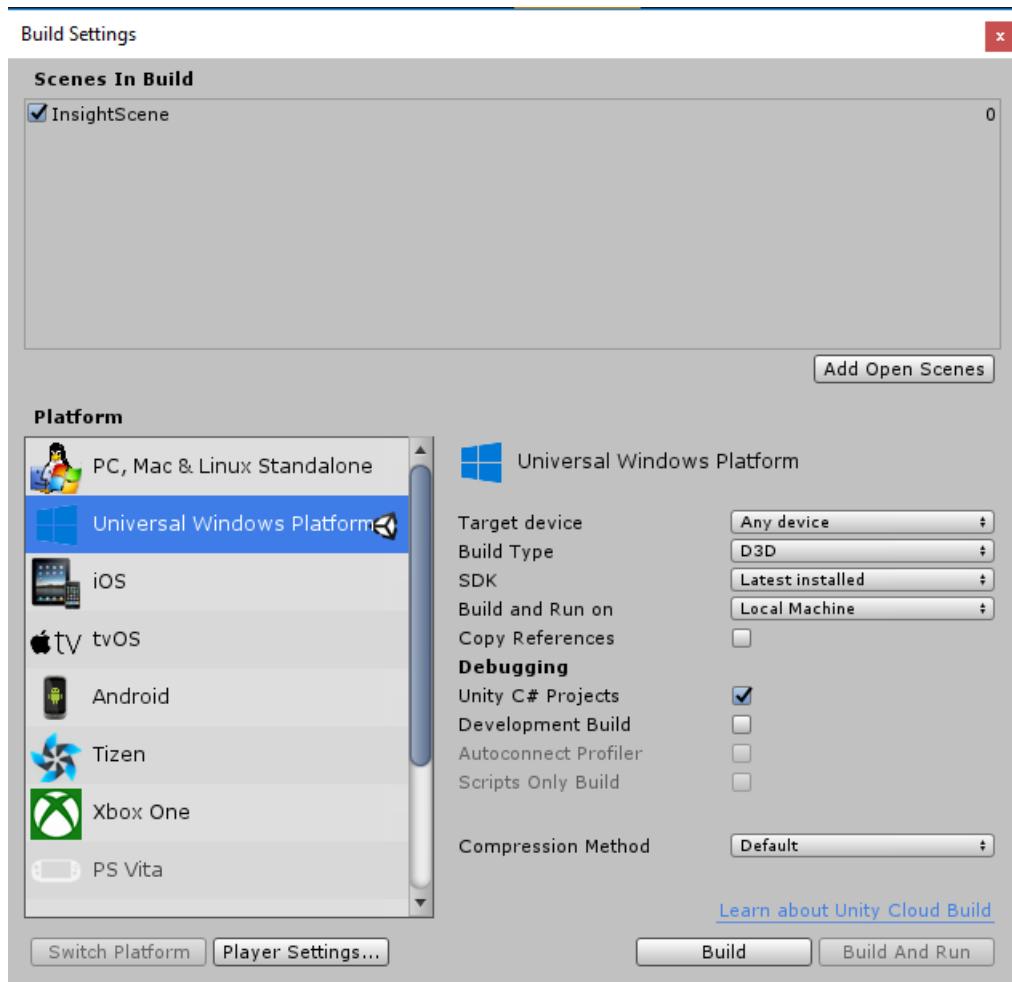
WARNING

If you see an error in the Unity console about the default audio device changing, the scene may not function as expected. This is due to the way the mixed reality portal deals with built-in microphones for headsets that have them. If you see this error, simply stop the scene and start it again and things should work as expected.

Chapter 10 – Build the UWP solution and sideload on local machine

Everything needed for the Unity section of this project has now been completed, so it is time to build it from Unity.

1. Navigate to **Build Settings: File > Build Settings...**
2. From the **Build Settings** window, click **Build**.



3. If not already, tick **Unity C# Projects**.
4. Click **Build**. Unity will launch a *File Explorer* window, where you need to create and then select a folder to build the app into. Create that folder now, and name it *App*. Then with the *App* folder selected, press **Select Folder**.
5. Unity will begin building your project to the *App* folder.
6. Once Unity has finished building (it might take some time), it will open a *File Explorer* window at the location of your build (check your task bar, as it may not always appear above your windows, but will notify you of the addition of a new window).

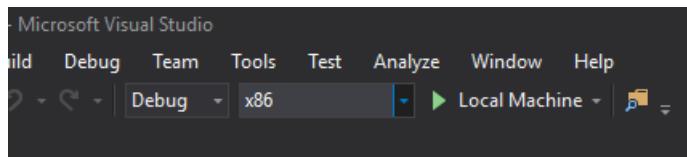
Chapter 11 – Deploy your application

To deploy your application:

1. Navigate to your new Unity build (the *App* folder) and open the solution file with *Visual Studio*.
2. In the Solution Configuration select **Debug**.
3. In the Solution Platform, select **x86, Local Machine**.

For the Microsoft HoloLens, you may find it easier to set this to *Remote Machine*, so that you are not tethered to your computer. Though, you will need to also do the following:

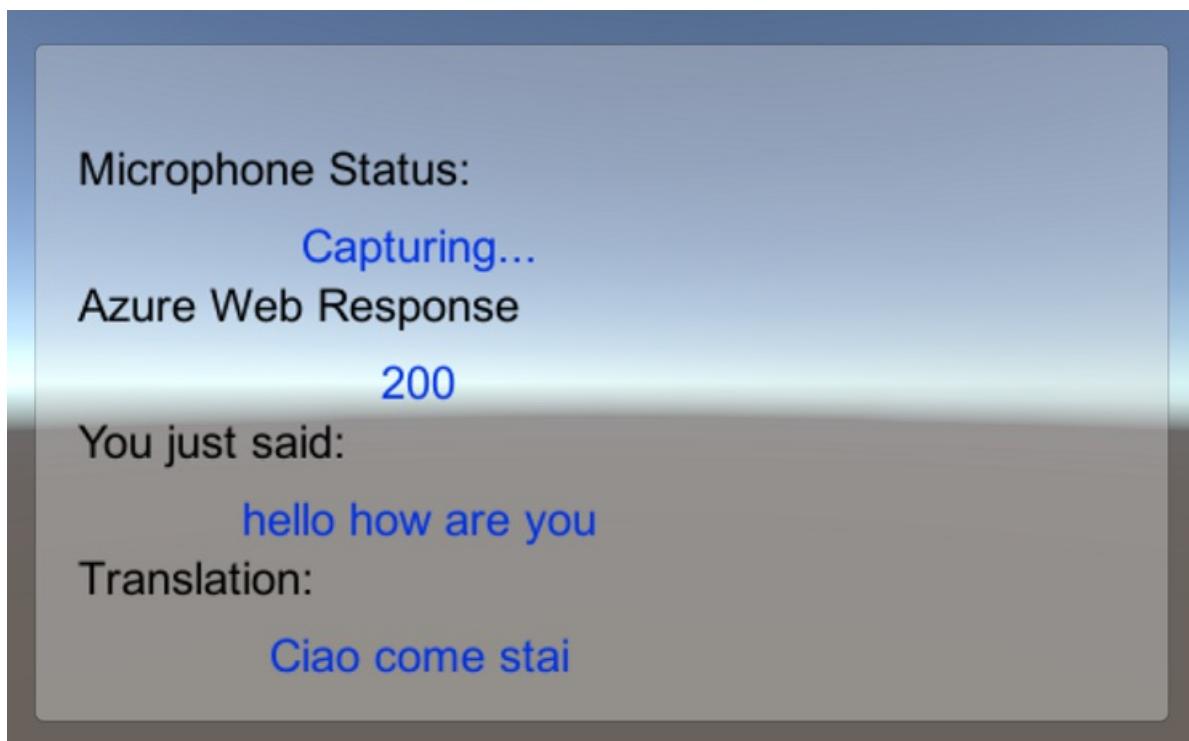
- Know the **IP Address** of your HoloLens, which can be found within the *Settings > Network & Internet > Wi-Fi > Advanced Options*; the IPv4 is the address you should use.
- Ensure *Developer Mode* is **On**; found in *Settings > Update & Security > For developers*.



4. Go to **Build menu** and click on **Deploy Solution** to sideload the application to your PC.
5. Your App should now appear in the list of installed apps, ready to be launched.
6. Once launched, the App will prompt you to authorize access to the Microphone. Make sure to click the **YES** button.
7. You are now ready to start translating!

Your finished Translation Text API application

Congratulations, you built a mixed reality app that leverages the Azure Translation Text API to convert speech to translated text.



Bonus exercises

Exercise 1

Can you add text-to-speech functionality to the app, so that the returned text is spoken?

Exercise 2

Make it possible for the user to change the source and output languages ('from' and 'to') within the app itself, so the app does not need to be rebuilt every time you want to change languages.

MR and Azure 302: Computer vision

11/6/2018 • 21 minutes to read • [Edit Online](#)

In this course, you will learn how to recognize visual content within a provided image, using Azure Computer Vision capabilities in a mixed reality application.

Recognition results will be displayed as descriptive tags. You can use this service without needing to train a machine learning model. If your implementation requires training a machine learning model, see [MR and Azure 302b](#).



The Microsoft Computer Vision is a set of APIs designed to provide developers with image processing and analysis (with return information), using advanced algorithms, all from the cloud. Developers upload an image or image URL, and the Microsoft Computer Vision API algorithms analyze the visual content, based upon inputs chosen by the user, which then can return information, including, identifying the type and quality of an image, detect human faces (returning their coordinates), and tagging, or categorizing images. For more information, visit the [Azure Computer Vision API page](#).

Having completed this course, you will have a mixed reality HoloLens application, which will be able to do the following:

1. Using the Tap gesture, the camera of the HoloLens will capture an image.
2. The image will be sent to the Azure Computer Vision API Service.
3. The objects recognized will be listed in a simple UI group positioned in the Unity Scene.

In your application, it is up to you as to how you will integrate the results with your design. This course is designed to teach you how to integrate an Azure Service with your Unity project. It is your job to use the knowledge you gain from this course to enhance your mixed reality application.

Device support

COURSE	HOLOLENS	IMMERSIVE HEADSETS
MR and Azure 302: Computer vision	✓ <input type="checkbox"/>	✓ <input type="checkbox"/>

NOTE

While this course primarily focuses on HoloLens, you can also apply what you learn in this course to Windows Mixed Reality immersive (VR) headsets. Because immersive (VR) headsets do not have accessible cameras, you will need an external camera connected to your PC. As you follow along with the course, you will see notes on any changes you might need to employ to support immersive (VR) headsets.

Prerequisites

NOTE

This tutorial is designed for developers who have basic experience with Unity and C#. Please also be aware that the prerequisites and written instructions within this document represent what has been tested and verified at the time of writing (May 2018). You are free to use the latest software, as listed within the [install the tools](#) article, though it should not be assumed that the information in this course will perfectly match what you'll find in newer software than what's listed below.

We recommend the following hardware and software for this course:

- A development PC, [compatible with Windows Mixed Reality](#) for immersive (VR) headset development
- [Windows 10 Fall Creators Update \(or later\)](#) with Developer mode enabled
- [The latest Windows 10 SDK](#)
- [Unity 2017.4](#)
- [Visual Studio 2017](#)
- A [Windows Mixed Reality immersive \(VR\) headset](#) or [Microsoft HoloLens](#) with Developer mode enabled
- A camera connected to your PC (for immersive headset development)
- Internet access for Azure setup and Computer Vision API retrieval

Before you start

1. To avoid encountering issues building this project, it is strongly suggested that you create the project mentioned in this tutorial in a root or near-root folder (long folder paths can cause issues at build-time).
2. Set up and test your HoloLens. If you need support setting up your HoloLens, [make sure to visit the HoloLens setup article](#).
3. It is a good idea to perform Calibration and Sensor Tuning when beginning developing a new HoloLens App (sometimes it can help to perform those tasks for each user).

For help on Calibration, please follow this [link to the HoloLens Calibration article](#).

For help on Sensor Tuning, please follow this [link to the HoloLens Sensor Tuning article](#).

Chapter 1 – The Azure Portal

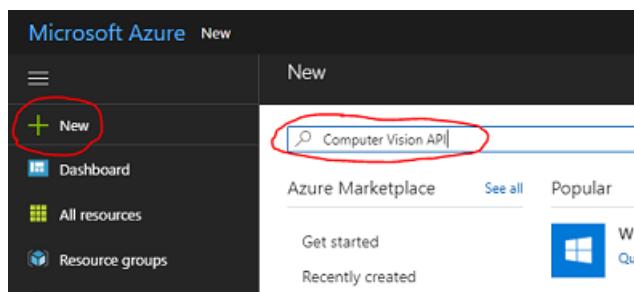
To use the *Computer Vision API* service in Azure, you will need to configure an instance of the service to be made available to your application.

1. First, log in to the [Azure Portal](#).

NOTE

If you do not already have an Azure account, you will need to create one. If you are following this tutorial in a classroom or lab situation, ask your instructor or one of the proctors for help setting up your new account.

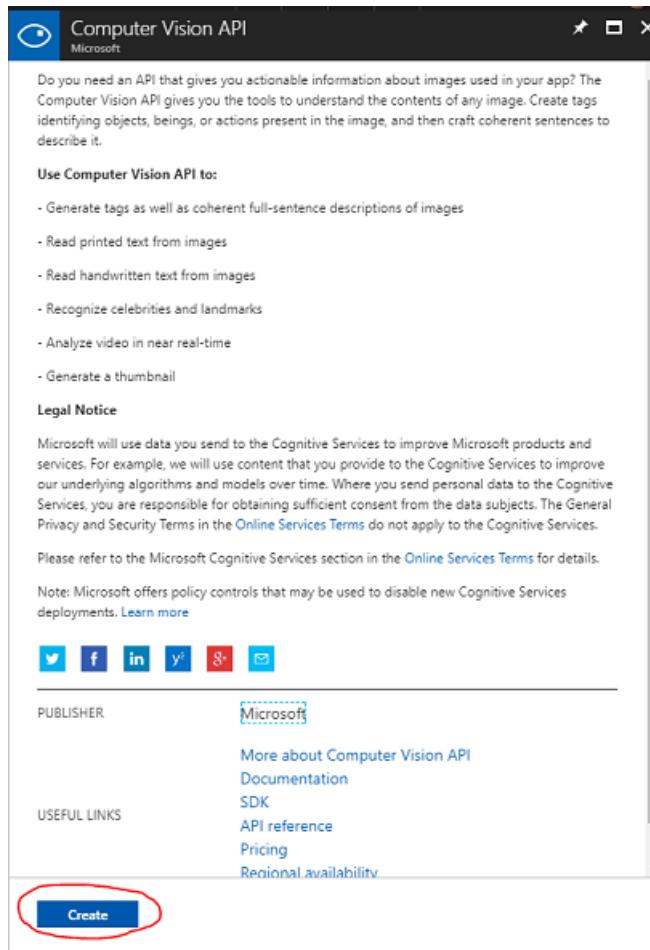
2. Once you are logged in, click on **New** in the top left corner, and search for *Computer Vision API*, and click **Enter**.



NOTE

The word **New** may have been replaced with **Create a resource**, in newer portals.

3. The new page will provide a description of the *Computer Vision API* service. At the bottom left of this page, select the **Create** button, to create an association with this service.



4. Once you have clicked on **Create**:

- Insert your desired **Name** for this service instance.
- Select a **Subscription**.
- Select the **Pricing Tier** appropriate for you, if this is the first time creating a *Computer Vision API* Service, a free tier (named F0) should be available to you.
- Choose a **Resource Group** or create a new one. A resource group provides a way to monitor, control access, provision and manage billing for a collection of Azure assets. It is recommended to keep all

the Azure services associated with a single project (e.g. such as these labs) under a common resource group).

If you wish to read more about Azure Resource Groups, please [visit the resource group article](#).

- e. Determine the Location for your resource group (if you are creating a new Resource Group). The location would ideally be in the region where the application would run. Some Azure assets are only available in certain regions.
- f. You will also need to confirm that you have understood the Terms and Conditions applied to this Service.
- g. Click Create.

The screenshot shows the 'Create a new service' dialog for Computer Vision. The fields filled are:

- Name: MyNewComputerVision
- Subscription: Main Subscription
- Location: West US
- Pricing tier: S1 (10 Calls per second)
- Resource group: AI_For_MR
- A checkbox is checked: "I confirm I have read and understood the notice below."

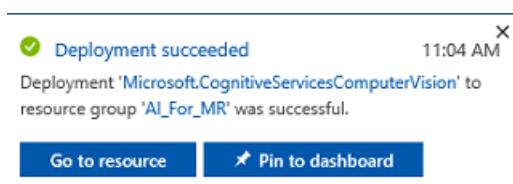
The notice below states: "Microsoft will use data you send to the Cognitive Services to improve Microsoft products and services. Where you send personal data to the Cognitive Services, you are responsible for obtaining sufficient consent from the data subjects. The General Privacy and Security Terms in the Online Services Terms do not apply to the Cognitive Services. Please refer to the Microsoft Cognitive Services section in the [Online Services Terms](#) for details. Microsoft offers policy controls that may be used to [disable new Cognitive Services deployments](#)".

At the bottom, there are buttons for "Pin to dashboard" (unchecked), "Create" (highlighted in blue), and "Automation options".

5. Once you have clicked on **Create**, you will have to wait for the service to be created, this might take a minute.
6. A notification will appear in the portal once the Service instance is created.



7. Click on the notification to explore your new Service instance.



8. Click the **Go to resource** button in the notification to explore your new Service instance. You will be taken to your new Computer Vision API service instance.

The screenshot shows the Azure portal interface for a service named 'MyNewComputerVision'. The left sidebar contains navigation links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, RESOURCE MANAGEMENT (with 'Keys' highlighted by a red box), MONITORING (Metrics, Alert rules), and SUPPORT + TROUBLESHOOTING. The main content area displays a 'Quick start' guide with three steps:

- 1 Grab your keys**: A note states that every call to the Computer Vision API requires a subscription key. It links to the 'Keys' section of the service overview, which is also highlighted with a red box.
- 2 Make an API call**: Provides information about API properties and methods, linking to the 'Computer Vision API reference'.
- 3 Enjoy coding**: Offers documentation and SDK links.

At the bottom of the main content, there are additional resources like Region availability, Support options, and Provide feedback.

9. Within this tutorial, your application will need to make calls to your service, which is done through using your service's Subscription Key.
10. From the *Quick start* page, of your *Computer Vision API* service, navigate to the first step, *Grab your keys*, and click **Keys** (you can also achieve this by clicking the blue hyperlink Keys, located in the services navigation menu, denoted by the key icon). This will reveal your service *Keys*.
11. Take a copy of one of the displayed keys, as you will need this later in your project.
12. Go back to the *Quick start* page, and from there, fetch your endpoint. Be aware yours may be different, depending on your region (which if it is, you will need to make a change to your code later). Take a copy of this endpoint for use later:

Search (Ctrl+ /) <<

- Overview
- Activity log
- Access control (IAM)
- Tags
- Diagnose and solve problems

RESOURCE MANAGEMENT

- Keys
- Quick start**
- Pricing tier
- Billing By Subscription
- Properties
- Locks
- Automation script

MONITORING

- Metrics
- Alerts (classic)

SUPPORT + TROUBLESHOOTING

Congratulations! Your keys are ready.

Now explore the Quickstart guidance to get up and running with Computer Vision API.

1 Grab your keys
Every call to the Computer Vision API requires a subscription key. This key needs to be either passed through a query string parameter or menu.
[Keys](#)

2 Make an API call to endpoint https://westus.api.cognitive.microsoft.com/vision/v1.0
Get in-depth information about each properties and methods of the API. Test your keys with the built-in testing console without writing Azure portal in your API 'Overview'.
[Computer Vision API reference](#)
[Realtime API usage](#)
[API metrics alert](#)
[Billing by subscription](#)
[Resource health status](#)

3 Enjoy coding
Learn more about the features, tutorials, developer tools, examples and how-to guidance to speed up.
[Documentation](#)
[SDK](#)

Additional resources
[Region availability](#)
[Support options](#)
[Provide feedback](#)

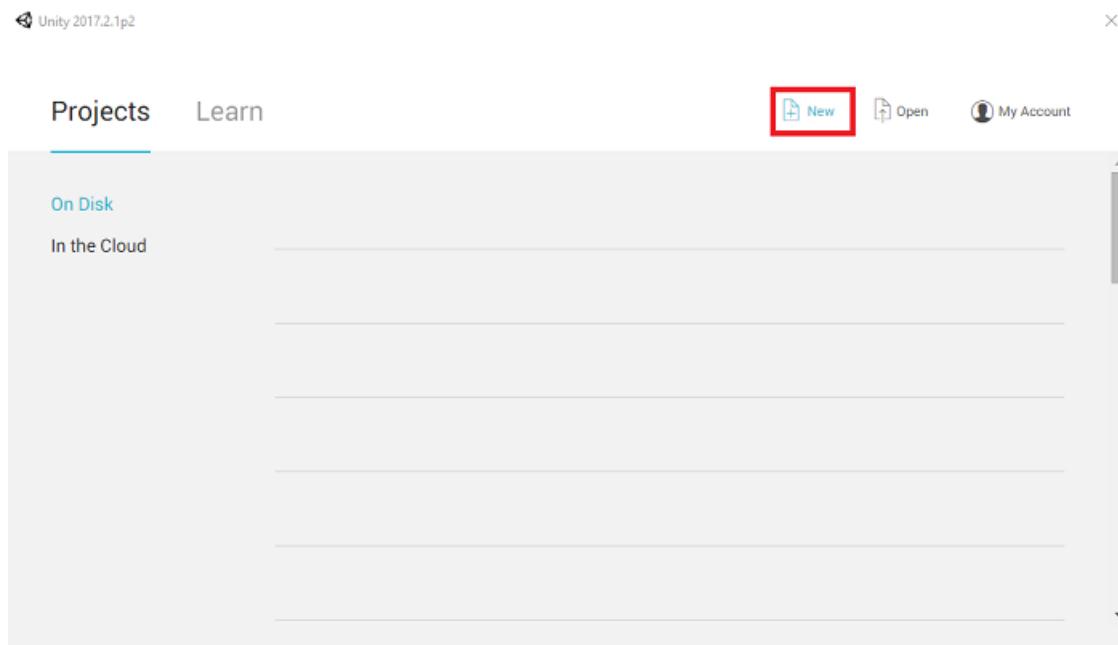
TIP

You can check what the various endpoints are [HERE](#).

Chapter 2 – Set up the Unity project

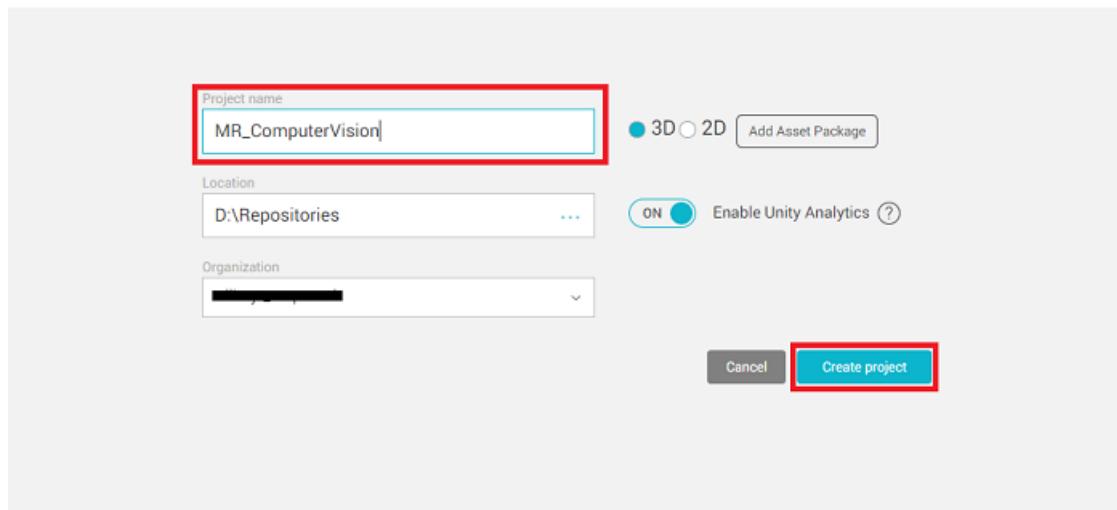
The following is a typical set up for developing with mixed reality, and as such, is a good template for other projects.

1. Open *Unity* and click **New**.

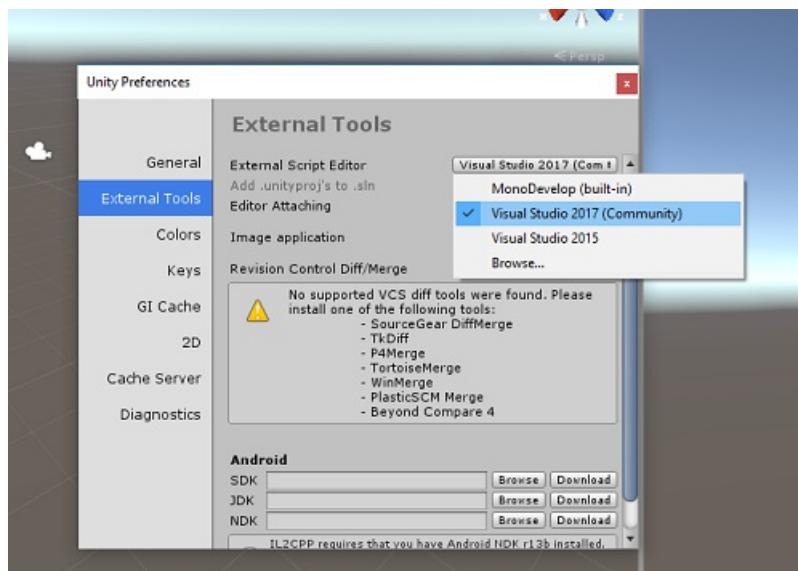


2. You will now need to provide a Unity Project name. Insert **MR_ComputerVision**. Make sure the project type is set to **3D**. Set the **Location** to somewhere appropriate for you (remember, closer to root directories is better). Then, click **Create project**.

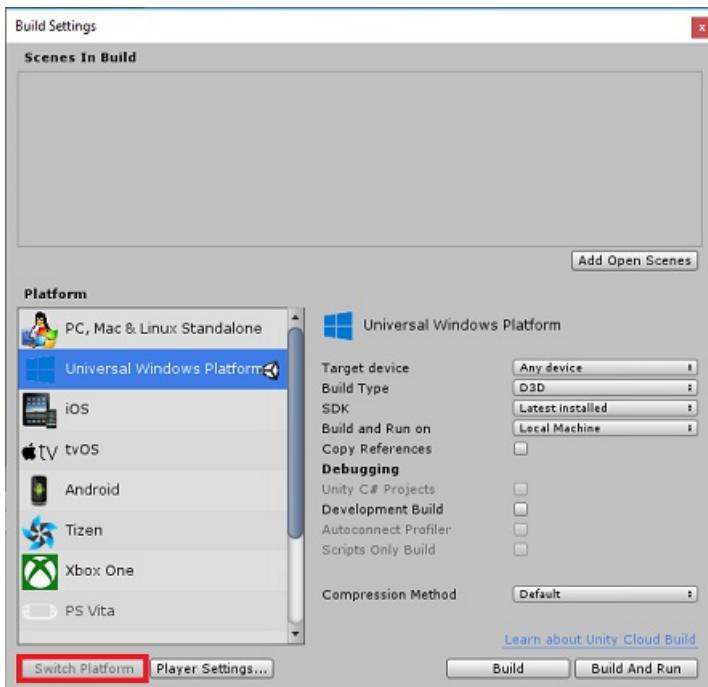
Projects Learn

[New](#) [Open](#) [My Account](#)

- With Unity open, it is worth checking the default **Script Editor** is set to **Visual Studio**. Go to **Edit > Preferences** and then from the new window, navigate to **External Tools**. Change **External Script Editor** to **Visual Studio 2017**. Close the **Preferences** window.



- Next, go to **File > Build Settings** and select **Universal Windows Platform**, then click on the **Switch Platform** button to apply your selection.



5. While still in **File > Build Settings** and make sure that:

a. **Target Device** is set to **HoloLens**

For the immersive headsets, set **Target Device** to *Any Device*.

b. **Build Type** is set to **D3D**

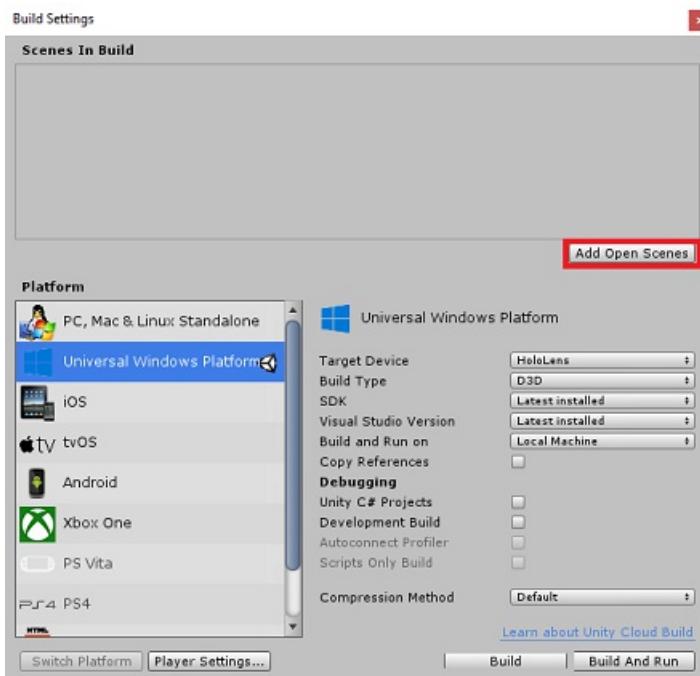
c. **SDK** is set to **Latest installed**

d. **Visual Studio Version** is set to **Latest installed**

e. **Build and Run** is set to **Local Machine**

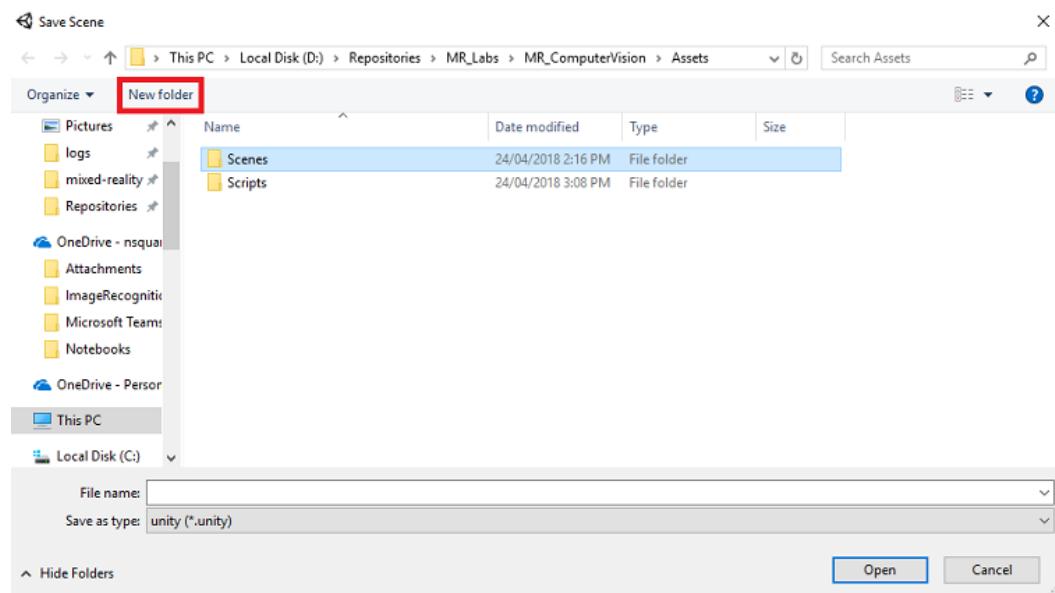
f. Save the scene and add it to the build.

a. Do this by selecting **Add Open Scenes**. A save window will appear.

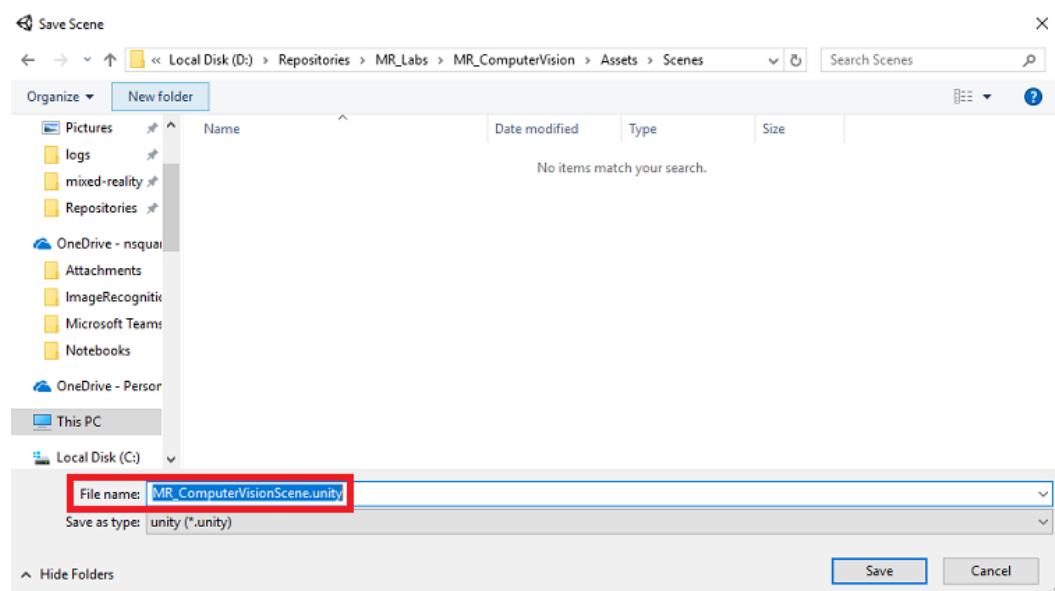


b. Create a new folder for this, and any future, scene, then select the **New folder** button, to

create a new folder, name it **Scenes**.

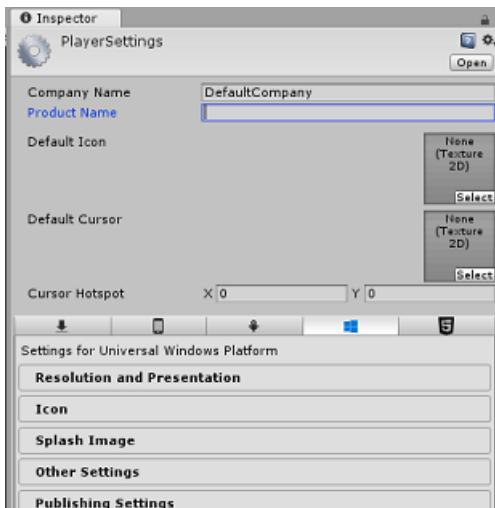


- c. Open your newly created **Scenes** folder, and then in the *File name:* text field, type **MR_ComputerVisionScene**, then click **Save**.



Be aware, you must save your Unity scenes within the *Assets* folder, as they must be associated with the Unity Project. Creating the scenes folder (and other similar folders) is a typical way of structuring a Unity project.

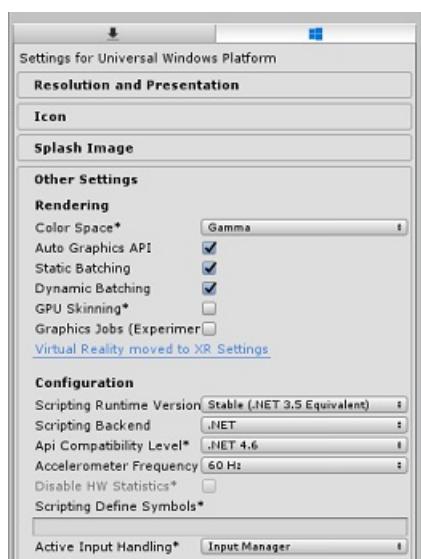
- g. The remaining settings, in *Build Settings*, should be left as default for now.
6. In the *Build Settings* window, click on the **Player Settings** button, this will open the related panel in the space where the *Inspector* is located.



7. In this panel, a few settings need to be verified:

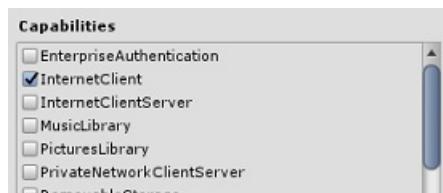
a. In the **Other Settings** tab:

- a. **Scripting Runtime Version** should be **Stable (.NET 3.5 Equivalent)**.
- b. **Scripting Backend** should be **.NET**
- c. **API Compatibility Level** should be **.NET 4.6**

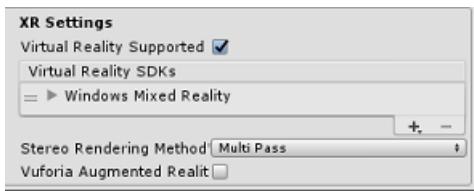


b. Within the **Publishing Settings** tab, under **Capabilities**, check:

- a. **InternetClient**
- b. **Webcam**



c. Further down the panel, in **XR Settings** (found below **Publish Settings**), tick **Virtual Reality Supported**, make sure the **Windows Mixed Reality SDK** is added.



8. Back in *Build Settings Unity C# Projects* is no longer greyed out; tick the checkbox next to this.

9. Close the Build Settings window.

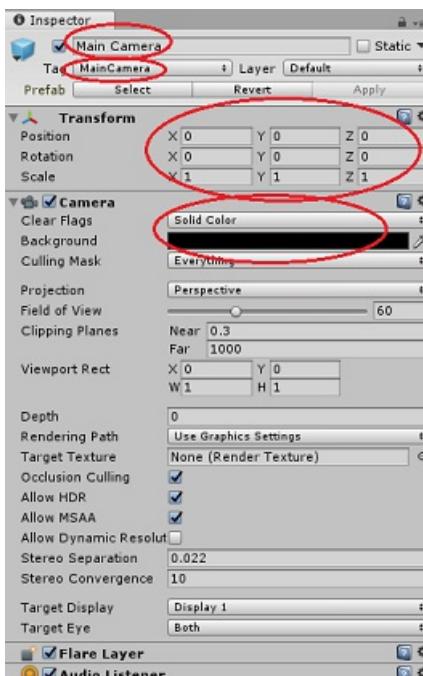
10. Save your Scene and Project (**FILE > SAVE SCENE / FILE > SAVE PROJECT**).

Chapter 3 – Main Camera setup

IMPORTANT

If you wish to skip the *Unity Set up* component of this course, and continue straight into code, feel free to download this [.unitypackage](#), import it into your project as a [Custom Package](#), and then continue from [Chapter 5](#).

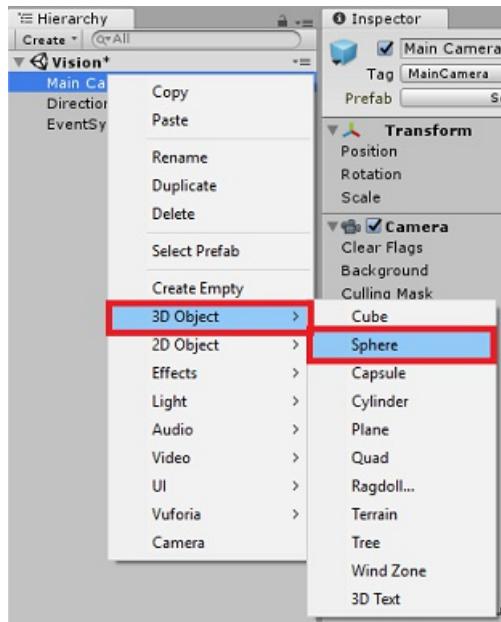
1. In the *Hierarchy Panel*, select the **Main Camera**.
2. Once selected, you will be able to see all the components of the **Main Camera** in the *Inspector Panel*.
 - a. The **Camera object** must be named **Main Camera** (note the spelling!)
 - b. The Main Camera **Tag** must be set to **MainCamera** (note the spelling!)
 - c. Make sure the **Transform Position** is set to **0, 0, 0**
 - d. Set **Clear Flags** to **Solid Color** (ignore this for immersive headset).
 - e. Set the **Background** Color of the Camera Component to **Black, Alpha 0 (Hex Code: #00000000)** (ignore this for immersive headset).



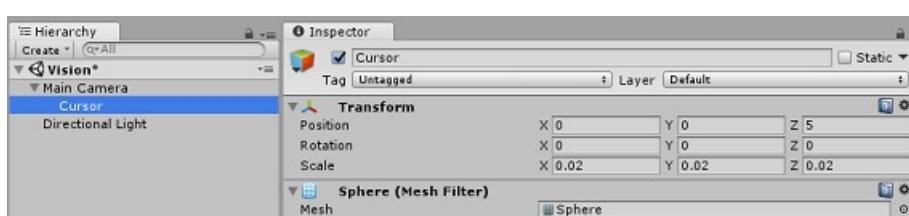
3. Next, you will have to create a simple "Cursor" object attached to the **Main Camera**, which will help you position the image analysis output when the application is running. This Cursor will determine the center point of the camera focus.

To create the Cursor:

1. In the **Hierarchy Panel**, right-click on the **Main Camera**. Under **3D Object**, click on **Sphere**.



2. Rename the **Sphere** to **Cursor** (double click the Cursor object or press the 'F2' keyboard button with the object selected), and make sure it is located as child of the **Main Camera**.
3. In the **Hierarchy Panel**, left click on the **Cursor**. With the Cursor selected, adjust the following variables in the **Inspector Panel**:
 - a. Set the *Transform Position* to **0, 0, 5**
 - b. Set the *Scale* to **0.02, 0.02, 0.02**



Chapter 4 – Setup the Label system

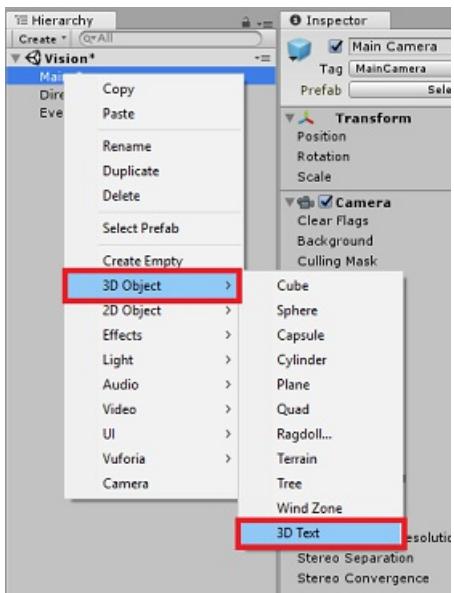
Once you have captured an image with the HoloLens' camera, that image will be sent to your *Azure Computer Vision API* Service instance for analysis.

The results of that analysis will be a list of recognized objects called **Tags**.

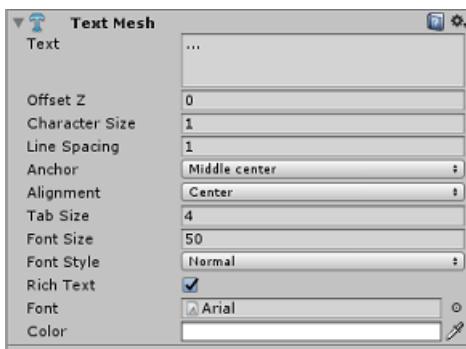
You will use Labels (as a 3D text in world space) to display these Tags at the location the photo was taken.

The following steps will show how to setup the **Label** object.

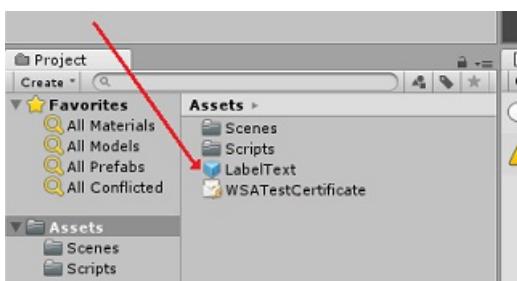
1. Right-click anywhere in the Hierarchy Panel (the location does not matter at this point), under **3D Object**, add a **3D Text**. Name it **LabelText**.



2. In the **Hierarchy Panel**, left click on the **LabelText**. With the **LabelText** selected, adjust the following variables in the **Inspector Panel**:
 - a. Set the **Position** to **0,0,0**
 - b. Set the **Scale** to **0.01, 0.01, 0.01**
 - c. In the component **Text Mesh**:
 - d. Replace all the text within **Text**, with "..."
 - e. Set the **Anchor** to **Middle Center**
 - f. Set the **Alignment** to **Center**
 - g. Set the **Tab Size** to **4**
 - h. Set the **Font Size** to **50**
 - i. Set the **Color** to **#FFFFFF**

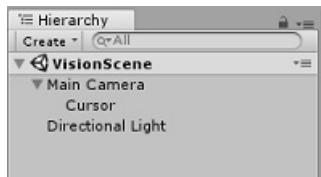


3. Drag the **LabelText** from the **Hierarchy Panel**, into the **Asset Folder**, within in the **Project Panel**. This will make the **LabelText** a Prefab, so that it can be instantiated in code.



4. You should delete the **LabelText** from the **Hierarchy Panel**, so that it will not be displayed in the opening scene. As it is now a prefab, which you will call on for individual instances from your Assets folder, there is no need to keep it within the scene.

5. The final object structure in the *Hierarchy Panel* should be like the one shown in the image below:



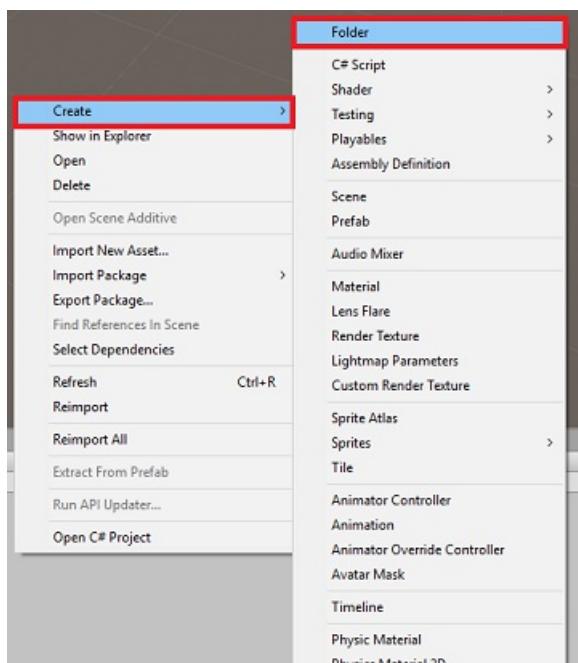
Chapter 5 – Create the ResultsLabel class

The first script you need to create is the *ResultsLabel* class, which is responsible for the following:

- Creating the Labels in the appropriate world space, relative to the position of the Camera.
- Displaying the Tags from the Image Analysis.

To create this class:

1. Right-click in the *Project Panel*, then **Create > Folder**. Name the folder **Scripts**.



2. With the **Scripts** folder created, double click it to open. Then within that folder, right-click, and select **Create > C# Script**. Name the script *ResultsLabel*.
3. Double click on the new *ResultsLabel* script to open it with **Visual Studio**.
4. Inside the Class insert the following code in the *ResultsLabel* class:

```

using System.Collections.Generic;
using UnityEngine;

public class ResultsLabel : MonoBehaviour
{
    public static ResultsLabel instance;

    public GameObject cursor;

    public Transform labelPrefab;

    [HideInInspector]
    public Transform lastLabelPlaced;

    [HideInInspector]
    public TextMesh lastLabelPlacedText;

    private void Awake()
    {
        // allows this instance to behave like a singleton
        instance = this;
    }

    /// <summary>
    /// Instantiate a Label in the appropriate location relative to the Main Camera.
    /// </summary>
    public void CreateLabel()
    {
        lastLabelPlaced = Instantiate(labelPrefab, cursor.transform.position, transform.rotation);

        lastLabelPlacedText = lastLabelPlaced.GetComponent<TextMesh>();

        // Change the text of the label to show that has been placed
        // The final text will be set at a later stage
        lastLabelPlacedText.text = "Analysing...";
    }

    /// <summary>
    /// Set the Tags as Text of the last Label created.
    /// </summary>
    public void SetTagsToLastLabel(Dictionary<string, float> tagsDictionary)
    {
        lastLabelPlacedText = lastLabelPlaced.GetComponent<TextMesh>();

        // At this point we go through all the tags received and set them as text of the label
        lastLabelPlacedText.text = "I see: \n";

        foreach (KeyValuePair<string, float> tag in tagsDictionary)
        {
            lastLabelPlacedText.text += tag.Key + ", Confidence: " + tag.Value.ToString("0.00 \n");
        }
    }
}

```

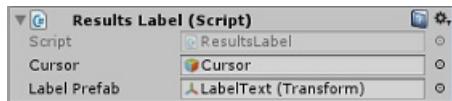
5. Be sure to save your changes in *Visual Studio* before returning to *Unity*.
6. Back in the *Unity Editor*, click and drag the *ResultsLabel* class from the **Scripts** folder to the **Main Camera** object in the *Hierarchy Panel*.
7. Click on the **Main Camera** and look at the *Inspector Panel*.

You will notice that from the script you just dragged into the Camera, there are two fields: **Cursor** and **Label Prefab**.

9. Drag the object called **Cursor** from the *Hierarchy Panel* to the slot named **Cursor**, as shown in the image

below.

10. Drag the object called **LabelText** from the *Assets Folder* in the *Project Panel* to the slot named **Label Prefab**, as shown in the image below.



Chapter 6 – Create the ImageCapture class

The next class you are going to create is the *ImageCapture* class. This class is responsible for:

- Capturing an Image using the HoloLens Camera and storing it in the App Folder.
- Capturing Tap gestures from the user.

To create this class:

1. Go to the **Scripts** folder you created previously.
2. Right-click inside the folder, **Create > C# Script**. Call the script *ImageCapture*.
3. Double click on the new *ImageCapture* script to open it with **Visual Studio**.
4. Add the following namespaces to the top of the file:

```
using System.IO;
using System.Linq;
using UnityEngine;
using UnityEngine.XR.WSA.Input;
using UnityEngine.XR.WSA.WebCam;
```

5. Then add the following variables inside the *ImageCapture* class, above the *Start()* method:

```
public static ImageCapture instance;
public int tapsCount;
private PhotoCapture photoCaptureObject = null;
private GestureRecognizer recognizer;
private bool currentlyCapturing = false;
```

The **tapsCount** variable will store the number of tap gestures captured from the user. This number is used in the naming of the images captured.

6. Code for *Awake()* and *Start()* methods now needs to be added. These will be called when the class initializes:

```
private void Awake()
{
    // Allows this instance to behave like a singleton
    instance = this;
}

void Start()
{
    // subscribing to the Hololens API gesture recognizer to track user gestures
    recognizer = new GestureRecognizer();
    recognizer.SetRecognizableGestures(GestureSettings.Tap);
    recognizer.Tapped += TapHandler;
    recognizer.StartCapturingGestures();
}
```

7. Implement a handler that will be called when a Tap gesture occurs.

```
/// <summary>
/// Respond to Tap Input.
/// </summary>
private void TapHandler(TappedEventArgs obj)
{
    // Only allow capturing, if not currently processing a request.
    if(currentlyCapturing == false)
    {
        currentlyCapturing = true;

        // increment taps count, used to name images when saving
        tapsCount++;

        // Create a label in world space using the ResultsLabel class
        ResultsLabel.instance.CreateLabel();

        // Begins the image capture and analysis procedure
        ExecuteImageCaptureAndAnalysis();
    }
}
```

The *TapHandler()* method increments the number of taps captured from the user and uses the current Cursor position to determine where to position a new Label.

This method then calls the *ExecuteImageCaptureAndAnalysis()* method to begin the core functionality of this application.

8. Once an Image has been captured and stored, the following handlers will be called. If the process is successful, the result is passed to the *VisionManager* (which you are yet to create) for analysis.

```
/// <summary>
/// Register the full execution of the Photo Capture. If successful, it will begin
/// the Image Analysis process.
/// </summary>
void OnCapturedPhotoToDisk(PhotoCapture.PhotoCaptureResult result)
{
    // Call StopPhotoMode once the image has successfully captured
    photoCaptureObject.StopPhotoModeAsync(OnStoppedPhotoMode);
}

void OnStoppedPhotoMode(PhotoCapture.PhotoCaptureResult result)
{
    // Dispose from the object in memory and request the image analysis
    // to the VisionManager class
    photoCaptureObject.Dispose();
    photoCaptureObject = null;
    StartCoroutine(VisionManager.instance.AnalyseLastImageCaptured());
}
```

9. Then add the method that the application uses to start the Image capture process and store the image.

```

/// <summary>
/// Begin process of Image Capturing and send To Azure
/// Computer Vision service.
/// </summary>
private void ExecuteImageCaptureAndAnalysis()
{
    // Set the camera resolution to be the highest possible
    Resolution cameraResolution = PhotoCapture.SupportedResolutions.OrderByDescending((res) =>
    res.width * res.height).First();

    Texture2D targetTexture = new Texture2D(cameraResolution.width, cameraResolution.height);

    // Begin capture process, set the image format
    PhotoCapture.CreateAsync(false, delegate (PhotoCapture captureObject)
    {
        photoCaptureObject = captureObject;
        CameraParameters camParameters = new CameraParameters();
        camParameters.hologramOpacity = 0.0f;
        camParameters.cameraResolutionWidth = targetTexture.width;
        camParameters.cameraResolutionHeight = targetTexture.height;
        camParameters.pixelFormat = CapturePixelFormat.BGRA32;

        // Capture the image from the camera and save it in the App internal folder
        captureObject.StartPhotoModeAsync(camParameters, delegate (PhotoCapture.PhotoCaptureResult
result)
        {
            string filename = string.Format(@"CapturedImage{0}.jpg", tapsCount);

            string filePath = Path.Combine(Application.persistentDataPath, filename);

            VisionManager.instance.imagePath = filePath;

            photoCaptureObject.TakePhotoAsync(filePath, PhotoCaptureFileOutputFormat.JPG,
OnCapturedPhotoToDisk);

            currentlyCapturing = false;
        });
    });
}

```

WARNING

At this point you will notice an error appearing in the *Unity Editor Console Panel*. This is because the code references the *VisionManager* class which you will create in the next Chapter.

Chapter 7 – Call to Azure and Image Analysis

The last script you need to create is the *VisionManager* class.

This class is responsible for:

- Loading the latest image captured as an array of bytes.
- Sending the byte array to your *Azure Computer Vision API* Service instance for analysis.
- Receiving the response as a JSON string.
- Deserializing the response and passing the resulting Tags to the *ResultsLabel* class.

To create this class:

1. Double click on the **Scripts** folder, to open it.
2. Right-click inside the **Scripts** folder, click **Create > C# Script**. Name the script *VisionManager*.

3. Double click on the new script to open it with Visual Studio.
4. Update the namespaces to be the same as the following, at the top of the *VisionManager* class:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.IO;
using UnityEngine;
using UnityEngine.Networking;
```

5. At the top of your script, *inside the VisionManager class* (above the *Start()* method), you now need to create two *Classes* that will represent the deserialized JSON response from Azure:

```
[System.Serializable]
public class TagData
{
    public string name;
    public float confidence;
}

[System.Serializable]
public class AnalysedObject
{
    public TagData[] tags;
    public string requestId;
    public object metadata;
}
```

NOTE

The *TagData* and *AnalysedObject* classes need to have the *[System.Serializable]* attribute added before the declaration to be able to be deserialized with the Unity libraries.

6. In the *VisionManager* class, you should add the following variables:

```
public static VisionManager instance;

// you must insert your service key here!
private string authorizationKey = "- Insert your key here -";
private const string ocpApimSubscriptionKeyHeader = "Ocp-Apim-Subscription-Key";
private string visionAnalysisEndpoint =
"https://westus.api.cognitive.microsoft.com/vision/v1.0/analyze?visualFeatures=Tags"; // This is where
you need to update your endpoint, if you set your location to something other than west-us.

internal byte[] imageBytes;

internal string imagePath;
```

WARNING

Make sure you insert your **Auth Key** into the **authorizationKey** variable. You will have noted your **Auth Key** at the beginning of this course, [Chapter 1](#).

WARNING

The **visionAnalysisEndpoint** variable might differ from the one specified in this example. The **west-us** strictly refers to Service instances created for the West US region. Update this with your [endpoint URL](#); here are some examples of what that might look like:

- West Europe:

```
https://westeurope.api.cognitive.microsoft.com/vision/v1.0/analyze?visualFeatures=Tags
```

- Southeast Asia:

```
https://southeastasia.api.cognitive.microsoft.com/vision/v1.0/analyze?visualFeatures=Tags
```

- Australia East:

```
https://australiaeast.api.cognitive.microsoft.com/vision/v1.0/analyze?visualFeatures=Tags
```

7. Code for Awake now needs to be added.

```
private void Awake()
{
    // allows this instance to behave like a singleton
    instance = this;
}
```

8. Next, add the coroutine (with the static stream method below it), which will obtain the results of the analysis of the image captured by the *ImageCapture* Class.

```

/// <summary>
/// Call the Computer Vision Service to submit the image.
/// </summary>
public IEnumerator AnalyseLastImageCaptured()
{
    WWWForm webForm = new WWWForm();
    using (UnityWebRequest unityWebRequest = UnityWebRequest.Post(visionAnalysisEndpoint, webForm))
    {
        // gets a byte array out of the saved image
        imageBytes = GetImageAsByteArray(imagePath);
        unityWebRequest.SetRequestHeader("Content-Type", "application/octet-stream");
        unityWebRequest.SetRequestHeader(ocpApimSubscriptionKeyHeader, authorizationKey);

        // the download handler will help receiving the analysis from Azure
        unityWebRequest.downloadHandler = new DownloadHandlerBuffer();

        // the upload handler will help uploading the byte array with the request
        unityWebRequest.uploadHandler = new UploadHandlerRaw(imageBytes);
        unityWebRequest.uploadHandler.contentType = "application/octet-stream";

        yield return unityWebRequest.SendWebRequest();

        long responseCode = unityWebRequest.responseCode;

        try
        {
            string jsonResponse = null;
            jsonResponse = unityWebRequest.downloadHandler.text;

            // The response will be in Json format
            // therefore it needs to be deserialized into the classes AnalysedObject and TagData
            AnalysedObject analysedObject = new AnalysedObject();
            analysedObject = JsonUtility.FromJson<AnalysedObject>(jsonResponse);

            if (analysedObject.tags == null)
            {
                Debug.Log("analysedObject.tagData is null");
            }
            else
            {
                Dictionary<string, float> tagsDictionary = new Dictionary<string, float>();

                foreach (TagData td in analysedObject.tags)
                {
                    TagData tag = td as TagData;
                    tagsDictionary.Add(tag.name, tag.confidence);
                }

                ResultsLabel.instance.SetTagsToLastLabel(tagsDictionary);
            }
        }
        catch (Exception exception)
        {
            Debug.Log("Json exception.Message: " + exception.Message);
        }

        yield return null;
    }
}

```

```

/// <summary>
/// Returns the contents of the specified file as a byte array.
/// </summary>
private static byte[] GetImageAsByteArray(string imagePath)
{
    FileStream fileStream = new FileStream(imagePath, FileMode.Open, FileAccess.Read);
    BinaryReader binaryReader = new BinaryReader(fileStream);
    return binaryReader.ReadBytes((int)fileStream.Length);
}

```

9. Be sure to save your changes in *Visual Studio* before returning to *Unity*.
10. Back in the *Unity Editor*, click and drag the *VisionManager* and *ImageCapture* classes from the **Scripts** folder to the **Main Camera** object in the *Hierarchy Panel*.

Chapter 8 – Before building

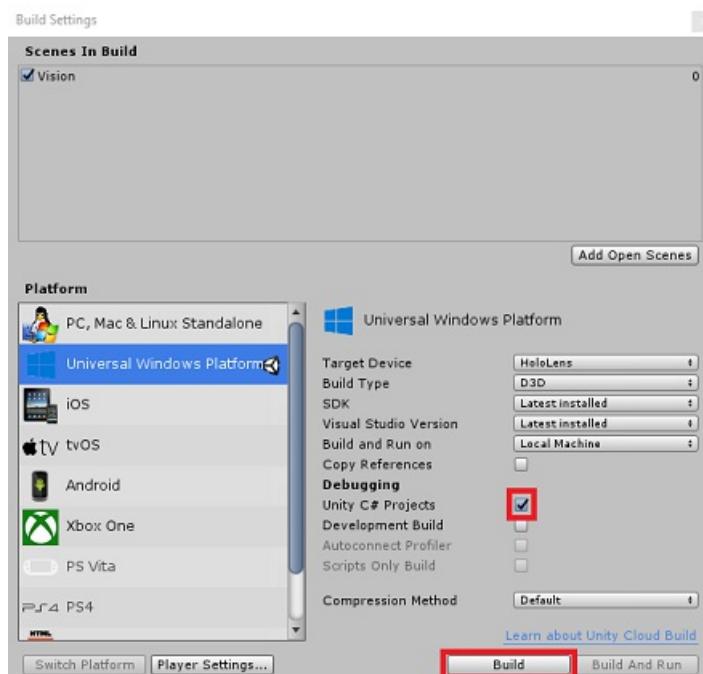
To perform a thorough test of your application you will need to sideload it onto your HoloLens. Before you do, ensure that:

- All the settings mentioned in [Chapter 2](#) are set correctly.
- All the scripts are attached to the **Main Camera** object.
- All the fields in the *Main Camera Inspector Panel* are assigned properly.
- Make sure you insert your **Auth Key** into the **authorizationKey** variable.
- Ensure that you have also checked your endpoint in your *VisionManager* script, and that it aligns to *your* region (this document uses *west-us* by default).

Chapter 9 – Build the UWP Solution and sideload the application

Everything needed for the *Unity* section of this project has now been completed, so it is time to build it from *Unity*.

1. Navigate to *Build Settings* - **File > Build Settings...**
2. From the *Build Settings* window, click **Build**.



3. If not already, tick **Unity C# Projects**.
4. Click **Build**. *Unity* will launch a *File Explorer* window, where you need to create and then select a folder to

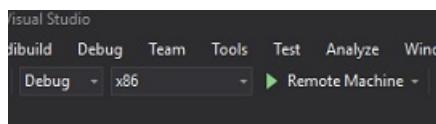
build the app into. Create that folder now, and name it *App*. Then with the *App* folder selected, press **Select Folder**.

5. Unity will begin building your project to the *App* folder.
6. Once Unity has finished building (it might take some time), it will open a *File Explorer* window at the location of your build (check your task bar, as it may not always appear above your windows, but will notify you of the addition of a new window).

Chapter 10 – Deploy to HoloLens

To deploy on HoloLens:

1. You will need the IP Address of your HoloLens (for Remote Deploy), and to ensure your HoloLens is in **Developer Mode**. To do this:
 - a. Whilst wearing your HoloLens, open the **Settings**.
 - b. Go to **Network & Internet > Wi-Fi > Advanced Options**
 - c. Note the **IPv4** address.
 - d. Next, navigate back to **Settings**, and then to **Update & Security > For Developers**
 - e. Set Developer Mode On.
2. Navigate to your new Unity build (the *App* folder) and open the solution file with *Visual Studio*.
3. In the Solution Configuration select **Debug**.
4. In the Solution Platform, select **x86, Remote Machine**.



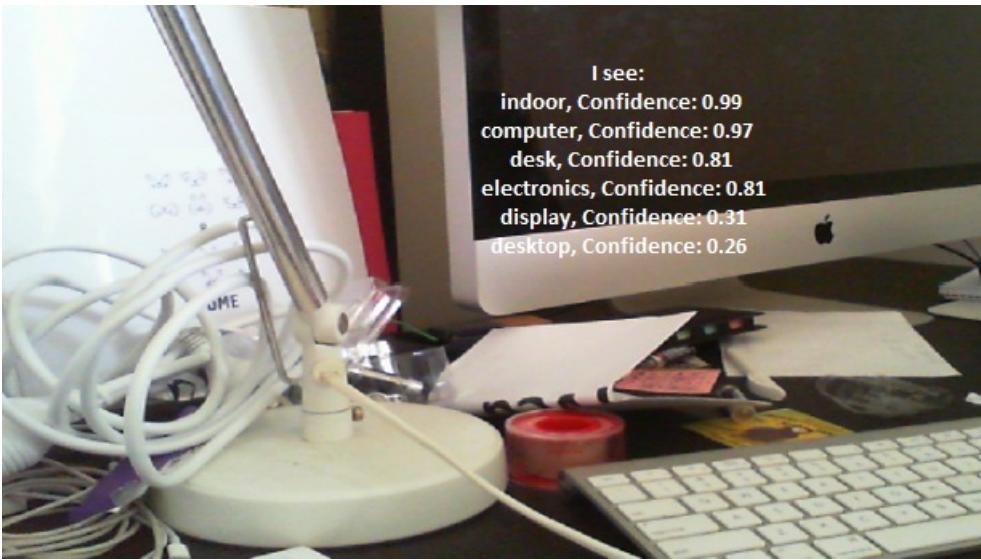
5. Go to the **Build menu** and click on **Deploy Solution**, to sideload the application to your HoloLens.
6. Your App should now appear in the list of installed apps on your HoloLens, ready to be launched!

NOTE

To deploy to immersive headset, set the **Solution Platform** to *Local Machine*, and set the **Configuration** to *Debug*, with *x86* as the **Platform**. Then deploy to the local machine, using the **Build menu**, selecting *Deploy Solution*.

Your finished Computer Vision API application

Congratulations, you built a mixed reality app that leverages the Azure Computer Vision API to recognize real world objects, and display confidence of what has been seen.



Bonus exercises

Exercise 1

Just as you have used the *Tags* parameter (as evidenced within the *endpoint* used within the *VisionManager*), extend the app to detect other information; have a look at what other parameters you have access to [HERE](#).

Exercise 2

Display the returned Azure data, in a more conversational, and readable way, perhaps hiding the numbers. As though a bot might be speaking to the user.

MR and Azure 302b: Custom vision

11/6/2018 • 41 minutes to read • [Edit Online](#)

In this course, you will learn how to recognize custom visual content within a provided image, using Azure Custom Vision capabilities in a mixed reality application.

This service will allow you to train a machine learning model using object images. You will then use the trained model to recognize similar objects, as provided by the camera capture of Microsoft HoloLens or a camera connected to your PC for immersive (VR) headsets.



Azure Custom Vision is a Microsoft Cognitive Service which allows developers to build custom image classifiers. These classifiers can then be used with new images to recognize, or classify, objects within that new image. The Service provides a simple, easy to use, online portal to streamline the process. For more information, visit the [Azure Custom Vision Service page](#).

Upon completion of this course, you will have a mixed reality application which will be able to work in two modes:

- **Analysis Mode:** setting up the Custom Vision Service manually by uploading images, creating tags, and training the Service to recognize different objects (in this case mouse and keyboard). You will then create a HoloLens app that will capture images using the camera, and try to recognize those objects in the real world.
- **Training Mode:** you will implement code that will enable a "Training Mode" in your app. The training mode will allow you to capture images using the HoloLens' camera, upload the captured images to the Service, and train the custom vision model.

This course will teach you how to get the results from the Custom Vision Service into a Unity-based sample application. It will be up to you to apply these concepts to a custom application you might be building.

Device support

COURSE	HOLOLENS	IMMERSIVE HEADSETS
MR and Azure 302b: Custom vision	✓ <input type="checkbox"/>	✓ <input type="checkbox"/>

NOTE

While this course primarily focuses on HoloLens, you can also apply what you learn in this course to Windows Mixed Reality immersive (VR) headsets. Because immersive (VR) headsets do not have accessible cameras, you will need an external camera connected to your PC. As you follow along with the course, you will see notes on any changes you might need to employ to support immersive (VR) headsets.

Prerequisites

NOTE

This tutorial is designed for developers who have basic experience with Unity and C#. Please also be aware that the prerequisites and written instructions within this document represent what has been tested and verified at the time of writing (July 2018). You are free to use the latest software, as listed within the [install the tools](#) article, though it should not be assumed that the information in this course will perfectly match what you will find in newer software than what is listed below.

We recommend the following hardware and software for this course:

- A development PC, [compatible with Windows Mixed Reality](#) for immersive (VR) headset development
- [Windows 10 Fall Creators Update \(or later\)](#) with Developer mode enabled
- [The latest Windows 10 SDK](#)
- [Unity 2017.4](#)
- [Visual Studio 2017](#)
- A [Windows Mixed Reality immersive \(VR\) headset](#) or [Microsoft HoloLens](#) with Developer mode enabled
- A camera connected to your PC (for immersive headset development)
- Internet access for Azure setup and Custom Vision API retrieval
- A series of at least five (5) images (ten (10) recommended) for each object that you would like the Custom Vision Service to recognize. If you wish, you can use [the images already provided with this course \(a computer mouse and a keyboard\)](#).

Before you start

1. To avoid encountering issues building this project, it is strongly suggested that you create the project mentioned in this tutorial in a root or near-root folder (long folder paths can cause issues at build-time).
2. Set up and test your HoloLens. If you need support setting up your HoloLens, [make sure to visit the HoloLens setup article](#).
3. It is a good idea to perform Calibration and Sensor Tuning when beginning developing a new HoloLens app (sometimes it can help to perform those tasks for each user).

For help on Calibration, please follow this [link to the HoloLens Calibration article](#).

For help on Sensor Tuning, please follow this [link to the HoloLens Sensor Tuning article](#).

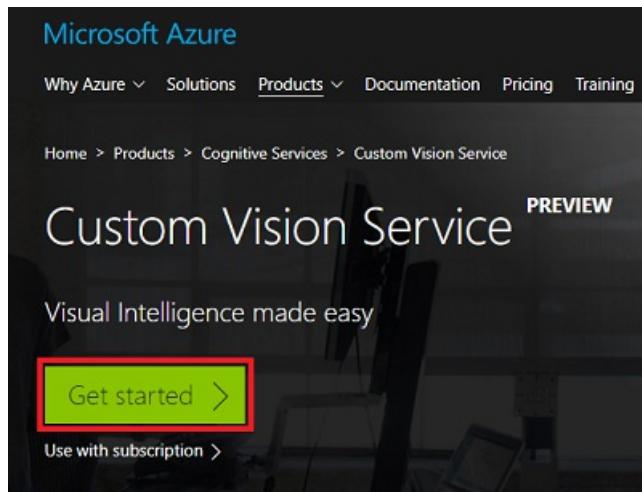
Chapter 1 - The Custom Vision Service Portal

To use the *Custom Vision Service* in Azure, you will need to configure an instance of the Service to be made

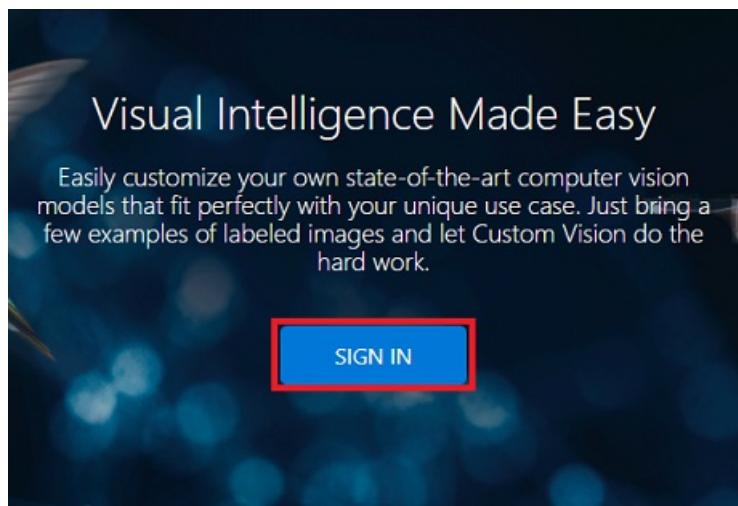
available to your application.

1. First, [navigate to the Custom Vision Service main page](#).

2. Click on the **Get Started** button.



3. Sign in to the *Custom Vision Service Portal*.



NOTE

If you do not already have an Azure account, you will need to create one. If you are following this tutorial in a classroom or lab situation, ask your instructor or one of the proctors for help setting up your new account.

4. Once you are logged in for the first time, you will be prompted with the *Terms of Service* panel. Click on the checkbox to agree to the terms. Then click on **I agree**.

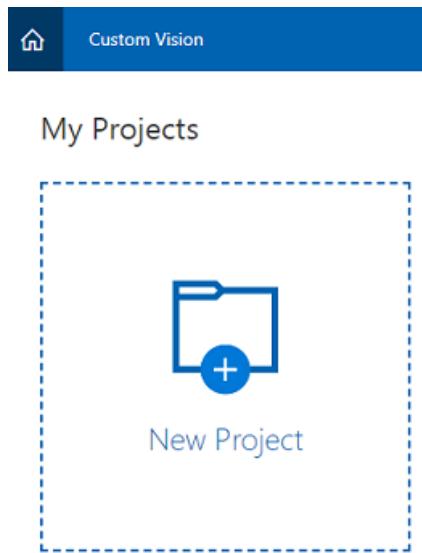
Terms of Service

Please note that Microsoft may retain copies of images uploaded for service improvement purposes. We won't publish your images or let other people use them.

I agree that my use of this service is governed by the [Microsoft Online Subscription Agreement](#), which incorporates the [Online Services Terms](#).

I agree

5. Having agreed to the Terms, you will be navigated to the *Projects* section of the Portal. Click on **New Project**.



6. A tab will appear on the right-hand side, which will prompt you to specify some fields for the project.
 - a. Insert a *Name* for your project.
 - b. Insert a *Description* for your project (*optional*).
 - c. Choose a *Resource Group* or create a new one. A resource group provides a way to monitor, control access, provision and manage billing for a collection of Azure assets. It is recommended to keep all the Azure Services associated with a single project (e.g. such as these courses) under a common resource group.

d. Set the *Project Types* to **Classification**

e. Set the *Domains* as **General**.

New project

Name*
MyCustomVision

Description
for the lab

Resource Group*
create new
Limited trial

Project Types ⓘ
 Classification
 Object Detection (preview)

Domains ⓘ
 General
 Food
 Landmarks
 Retail
 Adult
 General (compact)
 Landmarks (compact)
 Retail (compact)

[Cancel](#) [Create project](#)

If you wish to read more about Azure Resource Groups, please [visit the resource group article](#).

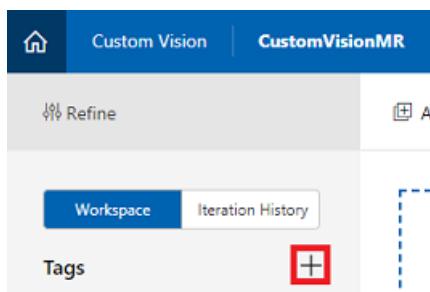
7. Once you are finished, click on **Create project**, you will be redirected to the Custom Vision Service, project page.

Chapter 2 - Training your Custom Vision project

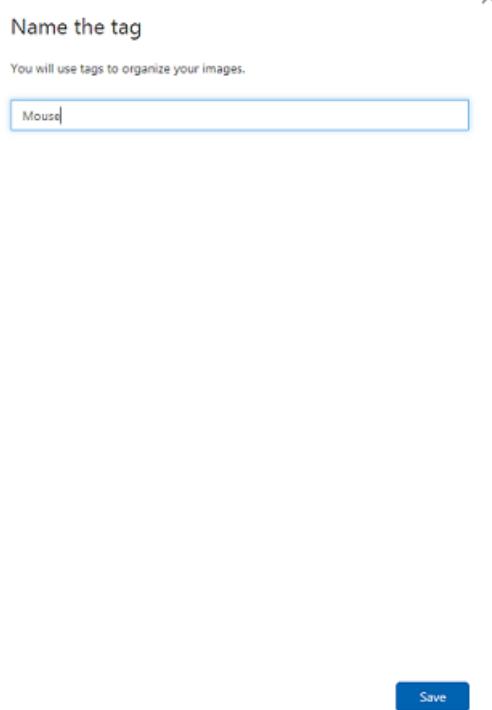
Once in the Custom Vision portal, your primary objective is to train your project to recognize specific objects in images. You need at least five (5) images, though ten (10) is preferred, for each object that you would like your application to recognize. [You can use the images provided with this course \(a computer mouse and a keyboard\).](#)

To train your Custom Vision Service project:

1. Click on the + button next to **Tags**.



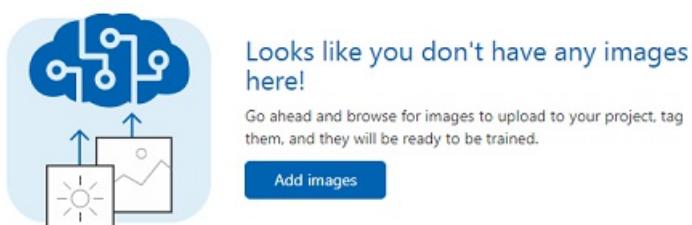
2. Add the **name** of the object you would like to recognize. Click on **Save**.



3. You will notice your **Tag** has been added (you may need to reload your page for it to appear). Click the checkbox alongside your new tag, if it is not already checked.

The screenshot shows the 'MyCustomVision' project interface. On the left, there's a sidebar with a 'Refine' button and a 'Workspace' dropdown. The main area is titled 'Iteration'. It shows a 'Tags' section with a 'tagged' tab selected, displaying the tag 'Mouse'. A blue dashed line highlights the 'Mouse' entry. Below the tags, there's a search bar and a checkbox for 'Mouse 0'.

4. Click on **Add Images** in the center of the page.

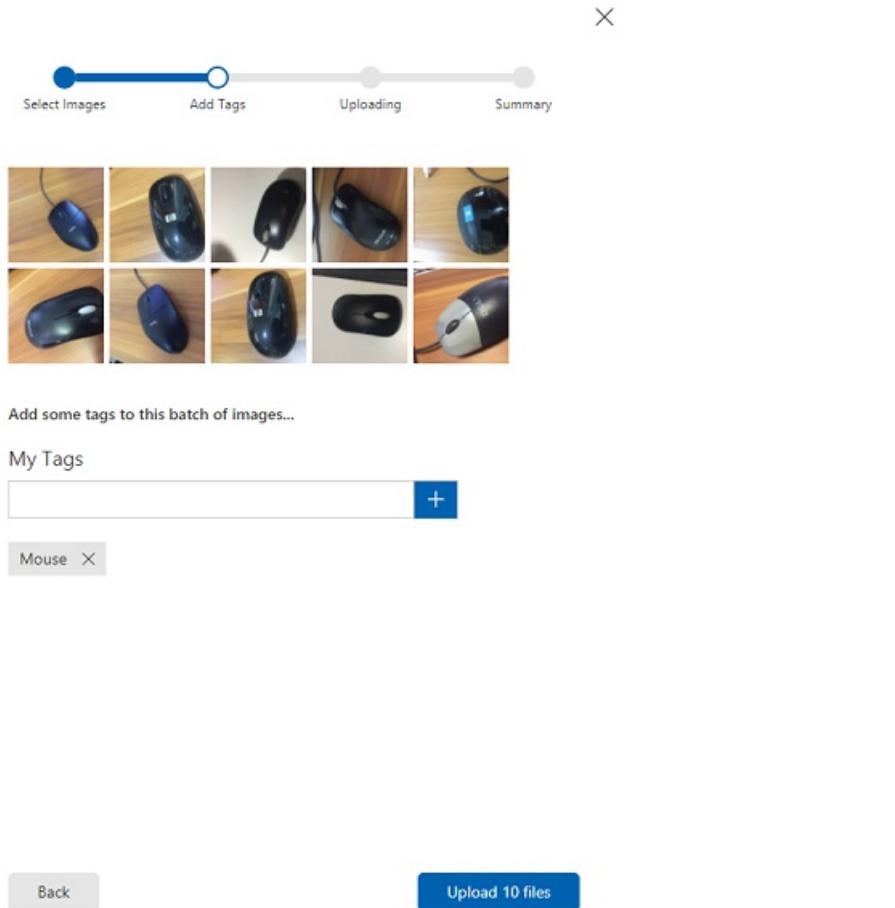


5. Click on **Browse local files**, and search, then select, the images you would like to upload, with the minimum being five (5). Remember all of these images should contain the object which you are training.

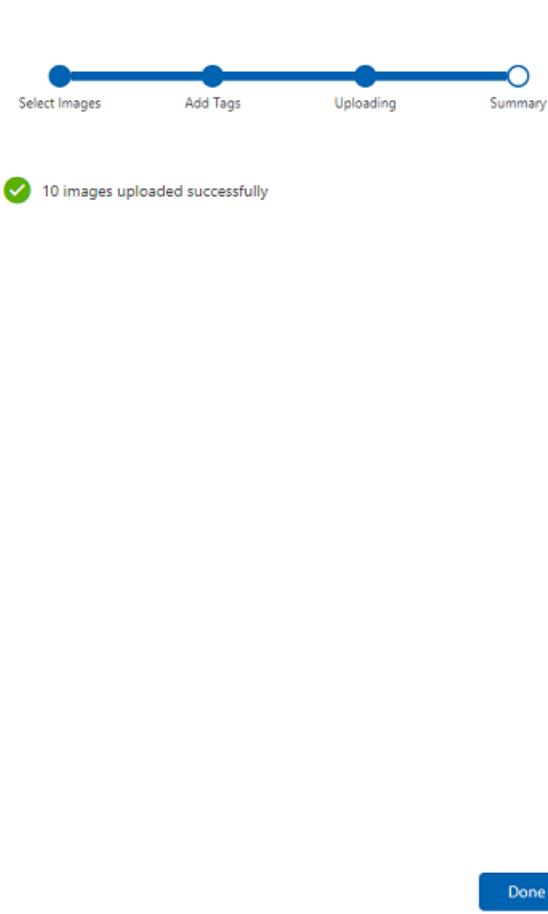
NOTE

You can select several images at a time, to upload.

6. Once you can see the images in the tab, select the appropriate tag in the **My Tags** box.



7. Click on **Upload files**. The files will begin uploading. Once you have confirmation of the upload, click **Done**.



8. Repeat the same process to create a new **Tag** named **Keyboard** and upload the appropriate photos for it. Make sure to **uncheck** *Mouse* once you have created the new tags, so to show the *Add images* window.
9. Once you have both Tags set up, click on **Train**, and the first training iteration will start building.

A screenshot of the Microsoft Custom Vision interface. At the top, there is a navigation bar with icons for Home, Custom Vision, and the current project "CustomVisionMR". To the right of the project name are three tabs: "TRAINING IMAGES" (which is highlighted in blue), "PERFORMANCE", and "PREDICTIONS". Below the navigation bar, there is a toolbar with icons for "Iterations" (with a dropdown arrow), "Delete", and "Export". A "Probability Threshold: 90%" slider is also present. The main area is titled "Iteration 1" and shows a large blue progress bar that is mostly filled. On the left side of the main area, there is a dark blue sidebar with the text "Iteration 1" and "Training...".

10. Once it is built, you will be able to see two buttons called **Make default** and **Prediction URL**. Click on **Make default** first, then click on **Prediction URL**.

Iterations

Prediction URL Make default Delete Export

Probability Threshold: 90%

Iteration 1

Trained : 2 minutes ago with General domain

Iteration 1

Finished training on 13/03/2018, 13:36:44 using General domain

Precision 100.0% Recall 100.0%

Performance Per Tag

Tag	Precision	Recall
Keyboard	100.0%	100.0%
Mouse	100.0%	100.0%

NOTE

The endpoint URL which is provided from this, is set to whichever *Iteration* has been marked as default. As such, if you later make a new *Iteration* and update it as default, you will not need to change your code.

- Once you have clicked on *Prediction URL*, open *Notepad*, and copy and paste the **URL** and the **Prediction-Key**, so that you can retrieve it when you need it later in the code.



How to use the Prediction API

If you have an image URL:

```
https://southcentralus.api.cognitive.microsoft.com/customvision/v1.1/Predictic
Set Prediction-Key Header to : 1cf740dc97d8[REDACTED]
Set Content-Type Header to : application/json
Set Body to : {"Url": "<image url>"}
```

If you have an image file:

```
https://southcentralus.api.cognitive.microsoft.com/customvision/v1.1/Predictic
Set Prediction-Key Header to : 1cf740dc97d8[REDACTED]
Set Content-Type Header to : application/octet-stream
Set Body to : <image file>
```

This iteration is marked as Default. If you mark another iteration as Default, the urls shown above will point to that iteration instead.

- Click on the **Cog** at the top right of the screen.



- Copy the **Training Key** and paste it into a *Notepad*, for later use.



Accounts

Limited trial

Training Key: [REDACTED]6d71b8bb

Prediction Key: [REDACTED]a20322b

1 projects created: 1 remain

67 predictions made: 9933 remain today

A screenshot of the 'Accounts' section. It shows a 'Limited trial' status. Under 'Training Key', a value is shown with the last few characters redacted. Under 'Prediction Key', another value is shown with the last few characters redacted. Below each key, it displays the number of projects created and predictions made, with a link to see more details.

14. Also copy your **Project Id**, and also paste it into your *Notepad* file, for later use.

Project Settings

General

Project Name*

MyCustomVision

Project Id

[REDACTED]e7b

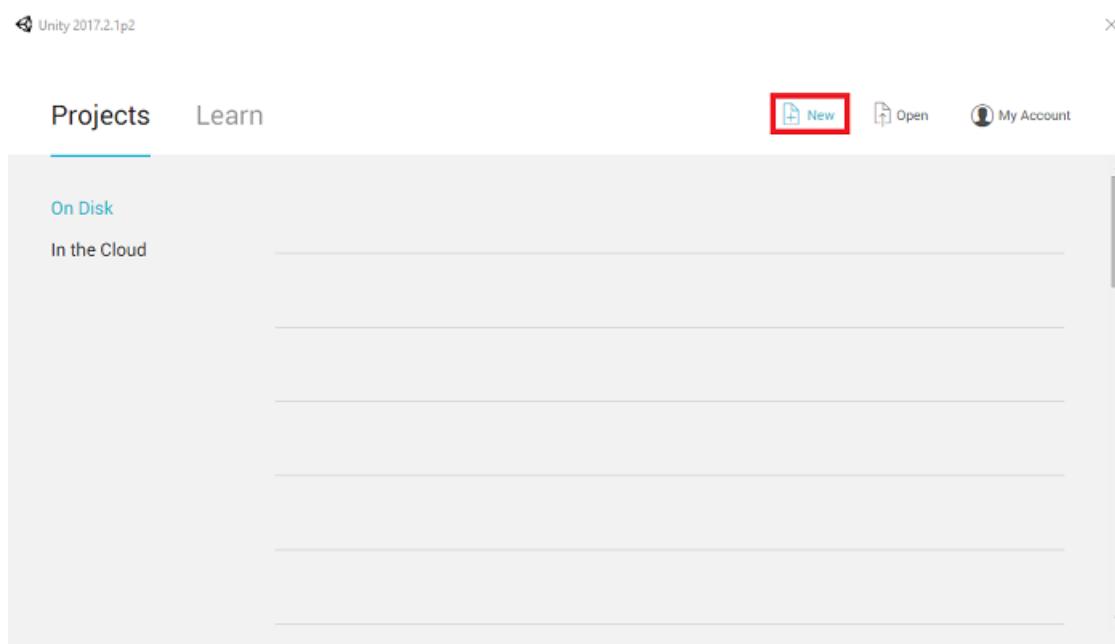
Description

for the lab

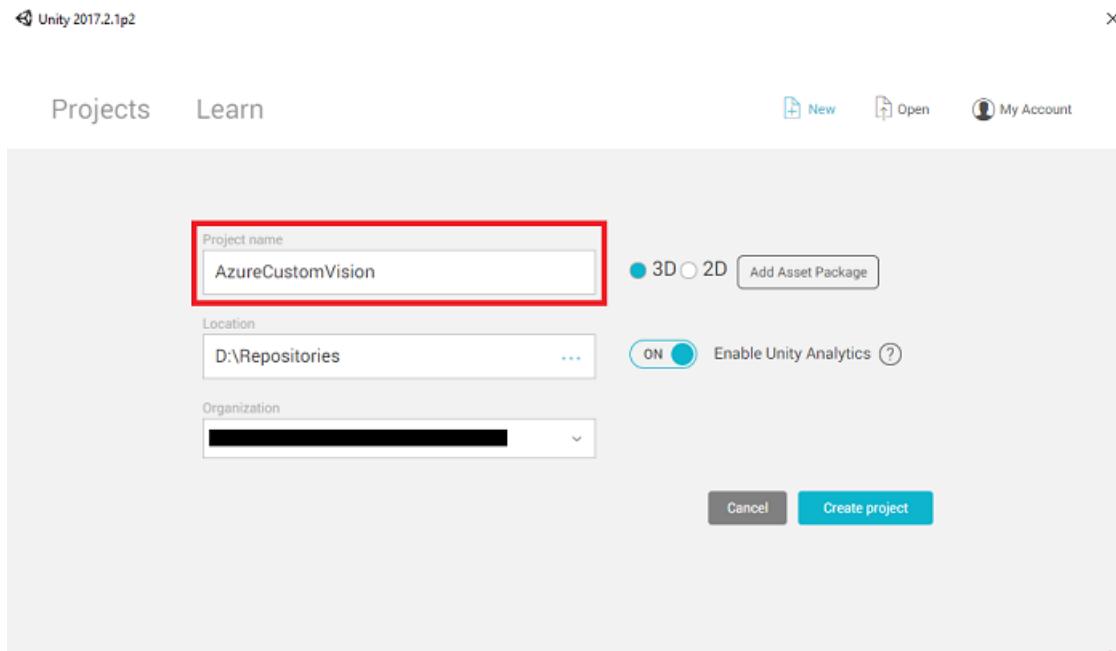
Chapter 3 - Set up the Unity project

The following is a typical set up for developing with mixed reality, and as such, is a good template for other projects.

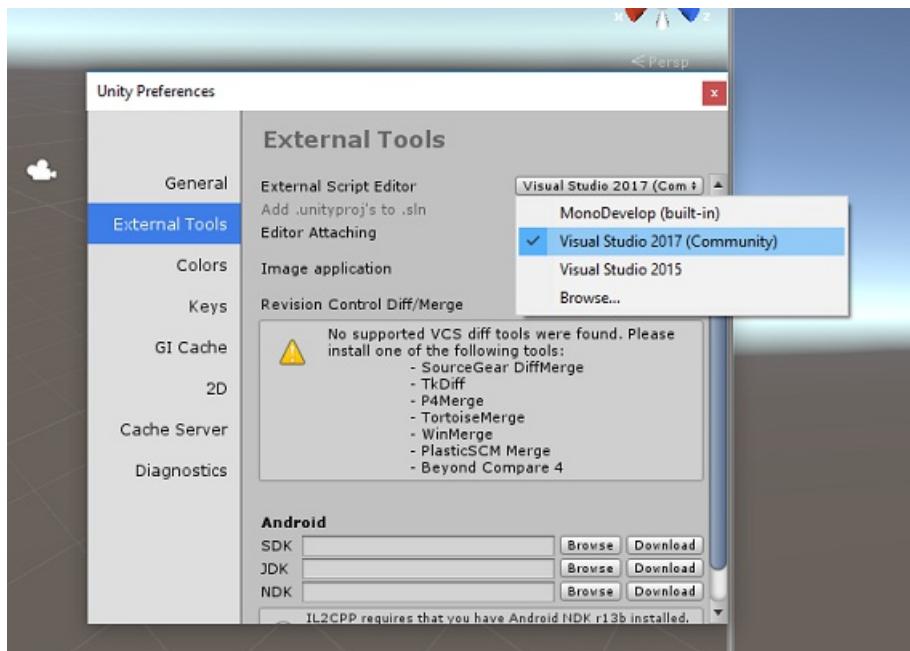
1. Open *Unity* and click **New**.



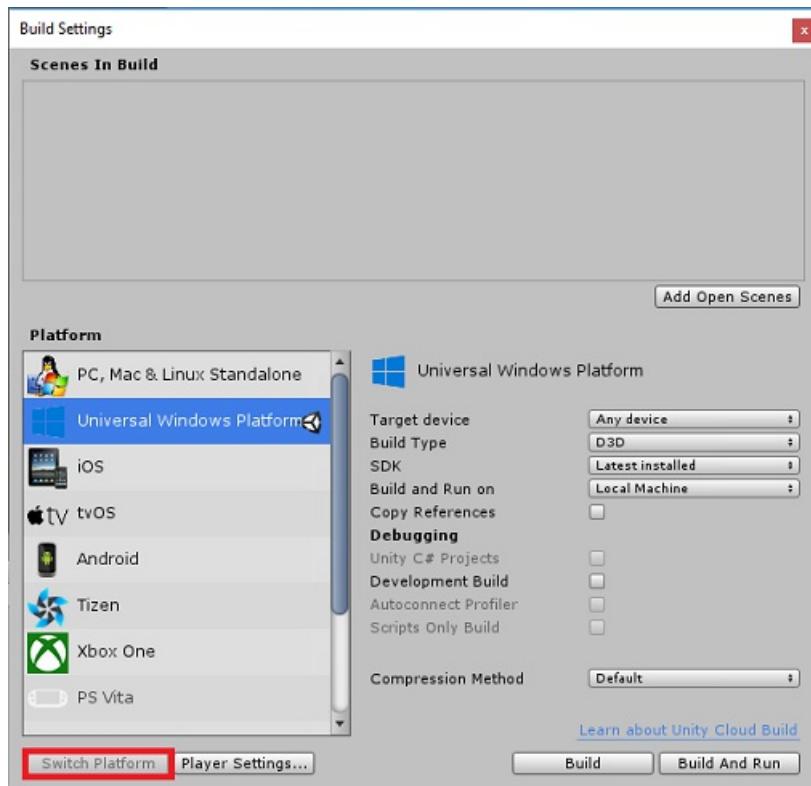
2. You will now need to provide a Unity project name. Insert **AzureCustomVision**. Make sure the project template is set to **3D**. Set the **Location** to somewhere appropriate for you (remember, closer to root directories is better). Then, click **Create project**.



3. With Unity open, it is worth checking the default **Script Editor** is set to **Visual Studio**. Go to *Edit > Preferences* and then from the new window, navigate to **External Tools**. Change **External Script Editor** to **Visual Studio 2017**. Close the **Preferences** window.



4. Next, go to **File > Build Settings** and select **Universal Windows Platform**, then click on the **Switch Platform** button to apply your selection.



5. While still in **File > Build Settings** and make sure that:

a. **Target Device** is set to **Hololens**

For the immersive headsets, set **Target Device** to **Any Device**.

b. **Build Type** is set to **D3D**

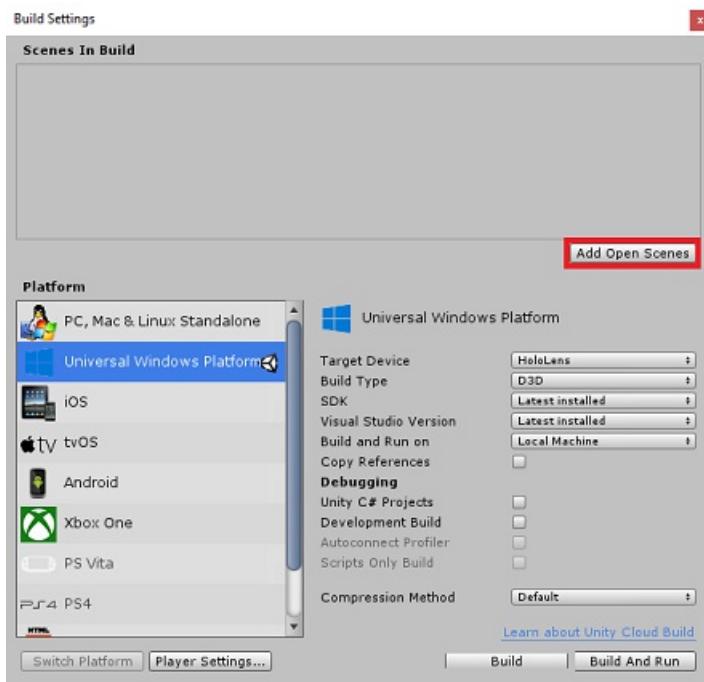
c. **SDK** is set to **Latest installed**

d. **Visual Studio Version** is set to **Latest installed**

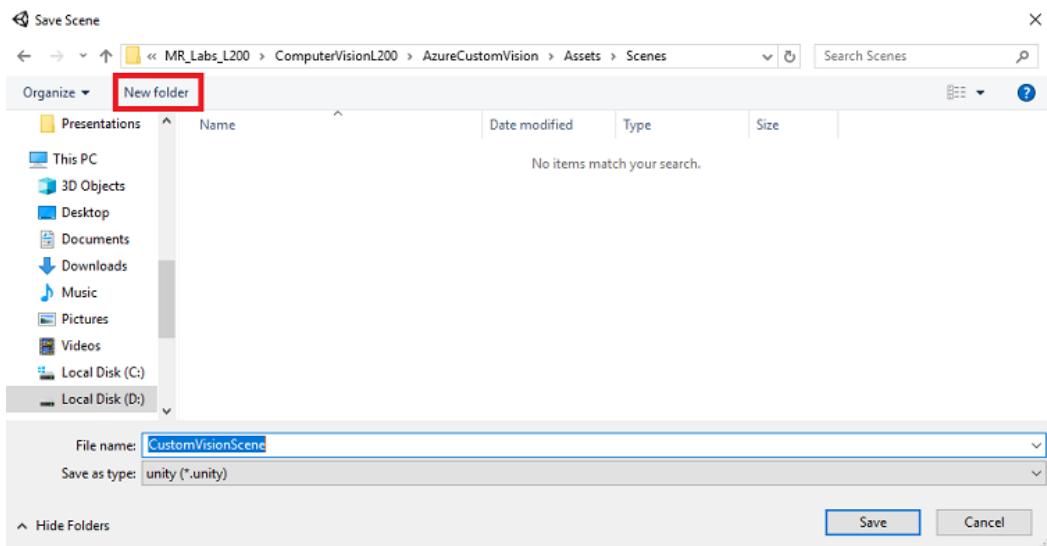
e. **Build and Run** is set to **Local Machine**

f. Save the scene and add it to the build.

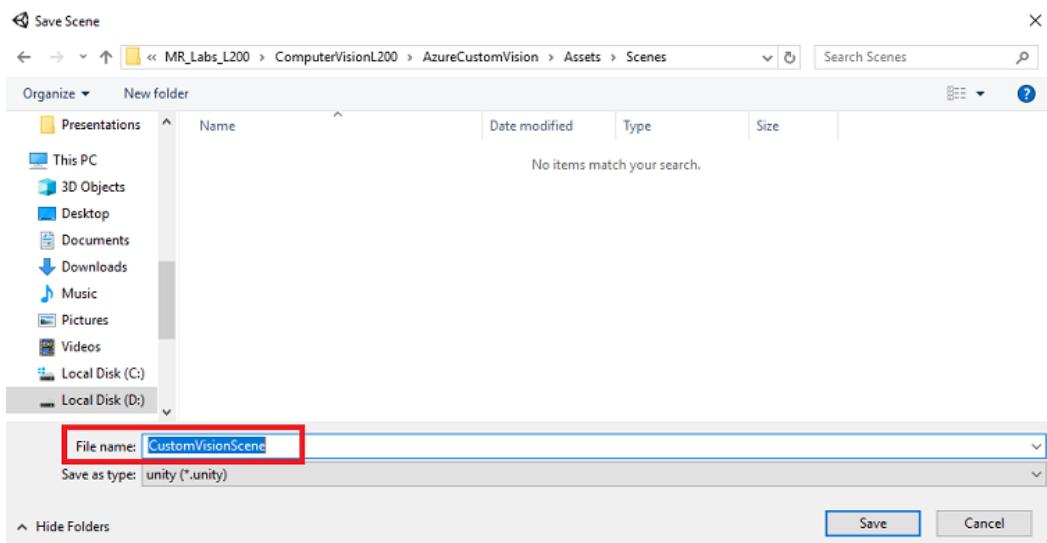
a. Do this by selecting **Add Open Scenes**. A save window will appear.



- b. Create a new folder for this, and any future, scene, then select the **New folder** button, to create a new folder, name it **Scenes**.

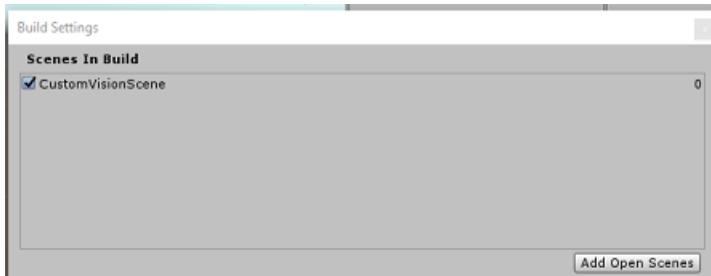


- c. Open your newly created **Scenes** folder, and then in the *File name:* text field, type **CustomVisionScene**, then click on **Save**.



Be aware, you must save your Unity scenes within the Assets folder, as they must be associated with the Unity project. Creating the scenes folder (and other similar folders) is a typical way of structuring a Unity project.

- g. The remaining settings, in *Build Settings*, should be left as default for now.



6. In the *Build Settings* window, click on the **Player Settings** button, this will open the related panel in the space where the *Inspector* is located.
7. In this panel, a few settings need to be verified:

- a. In the **Other Settings** tab:

- Scripting Runtime Version** should be **Experimental (.NET 4.6 Equivalent)**, which will trigger a need to restart the Editor.
- Scripting Backend** should be **.NET**
- API Compatibility Level** should be **.NET 4.6**

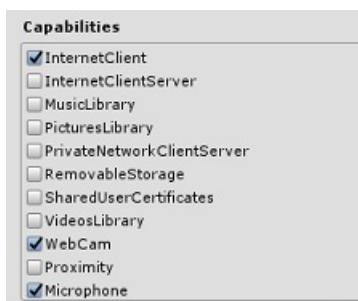


- b. Within the **Publishing Settings** tab, under **Capabilities**, check:

- InternetClient**

b. **Webcam**

c. **Microphone**



c. Further down the panel, in **XR Settings** (found below **Publish Settings**), tick **Virtual Reality Supported**, make sure the **Windows Mixed Reality SDK** is added.



8. Back in *Build Settings Unity C# Projects* is no longer greyed out; tick the checkbox next to this.

9. Close the Build Settings window.

10. Save your Scene and project (**FILE > SAVE SCENE / FILE > SAVE PROJECT**).

Chapter 4 - Importing the Newtonsoft DLL in Unity

IMPORTANT

If you wish to skip the *Unity Set up* component of this course, and continue straight into code, feel free to download this [Azure-MR-302b.unitypackage](#), import it into your project as a **Custom Package**, and then continue from [Chapter 6](#).

This course requires the use of the **Newtonsoft** library, which you can add as a DLL to your assets. The package containing [this library can be downloaded from this Link](#). To import the Newtonsoft library into your project, use the Unity Package which came with this course.

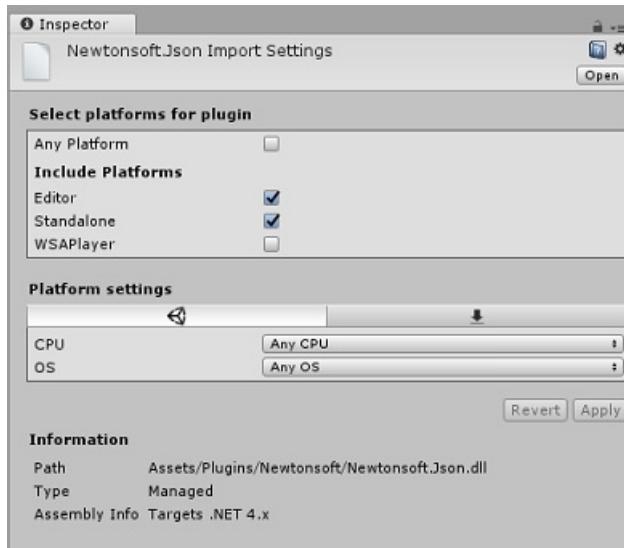
1. Add the *.unitypackage* to Unity by using the *Assets > Import Package > Custom Package* menu option.
2. In the **Import Unity Package** box that pops up, ensure everything under (and including) **Plugins** is selected.



3. Click the **Import** button to add the items to your project.
4. Go to the **Newtonsoft** folder under **Plugins** in the project view and select the *Newtonsoft.Json plugin*.



5. With the *Newtonsoft.Json plugin* selected, ensure that **Any Platform** is **unchecked**, then ensure that **WSAPlayer** is also **unchecked**, then click **Apply**. This is just to confirm that the files are configured correctly.

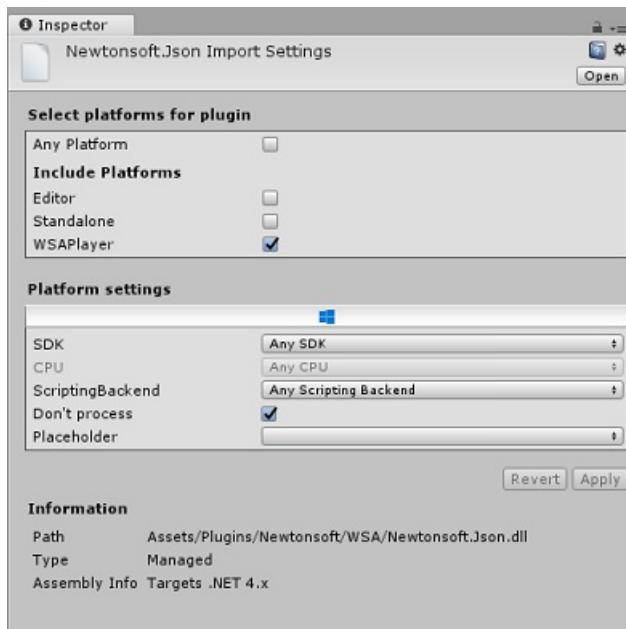


NOTE

Marking these plugins configures them to only be used in the Unity Editor. There are a different set of them in the WSA folder which will be used after the project is exported from Unity.

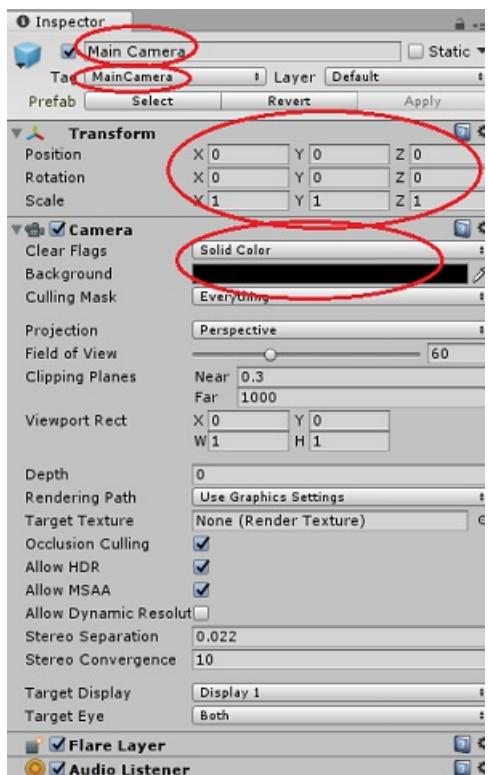
6. Next, you need to open the **WSA** folder, within the **Newtonsoft** folder. You will see a copy of the same file you just configured. Select the file, and then in the inspector, ensure that

- **Any Platform** is **unchecked**
- **only WSAPlayer** is **checked**
- **Dont process** is **checked**



Chapter 5 - Camera setup

1. In the Hierarchy Panel, select the *Main Camera*.
2. Once selected, you will be able to see all the components of the *Main Camera* in the *Inspector Panel*.
 - a. The *camera* object must be named **Main Camera** (note the spelling!)
 - b. The Main Camera **Tag** must be set to **MainCamera** (note the spelling!)
 - c. Make sure the **Transform Position** is set to **0, 0, 0**
 - d. Set **Clear Flags** to **Solid Color** (ignore this for immersive headset).
 - e. Set the **Background** Color of the camera Component to **Black, Alpha 0 (Hex Code: #00000000)** (ignore this for immersive headset).



Chapter 6 - Create the CustomVisionAnalyser class.

At this point you are ready to write some code.

You will begin with the *CustomVisionAnalyser* class.

NOTE

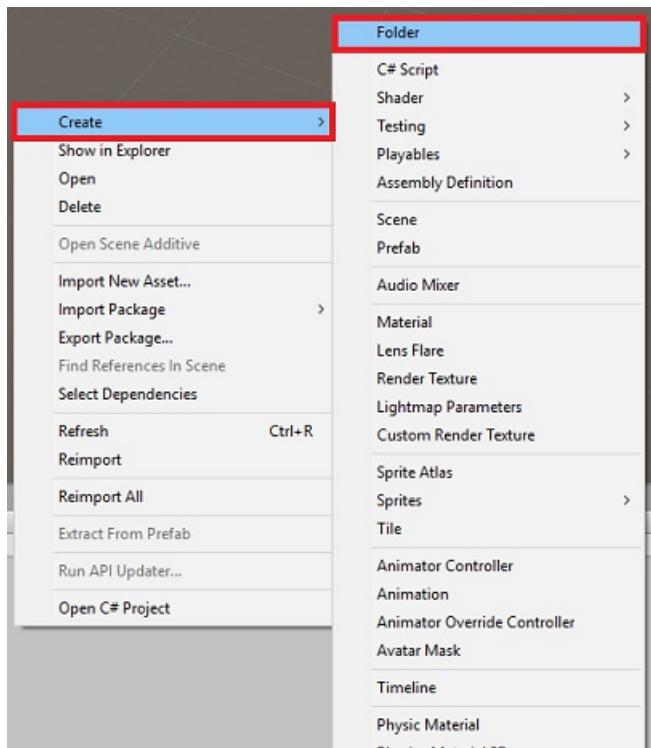
The calls to the **Custom Vision Service** made in the code shown below are made using the **Custom Vision REST API**. Through using this, you will see how to implement and make use of this API (useful for understanding how to implement something similar on your own). Be aware, that Microsoft offers a **Custom Vision Service SDK** that can also be used to make calls to the Service. For more information visit the [Custom Vision Service SDK](#) article.

This class is responsible for:

- Loading the latest image captured as an array of bytes.
- Sending the byte array to your Azure *Custom Vision Service* instance for analysis.
- Receiving the response as a JSON string.
- Deserializing the response and passing the resulting *Prediction* to the *SceneOrganiser* class, which will take care of how the response should be displayed.

To create this class:

1. Right-click in the *Asset Folder* located in the *Project Panel*, then click **Create > Folder**. Call the folder **Scripts**.



2. Double-click on the folder just created, to open it.
3. Right-click inside the folder, then click **Create > C# Script**. Name the script *CustomVisionAnalyser*.
4. Double-click on the new *CustomVisionAnalyser* script to open it with **Visual Studio**.
5. Update the namespaces at the top of your file to match the following:

```
using System.Collections;
using System.IO;
using UnityEngine;
using UnityEngine.Networking;
using Newtonsoft.Json;
```

6. In the *CustomVisionAnalyser* class, add the following variables:

```
/// <summary>
/// Unique instance of this class
/// </summary>
public static CustomVisionAnalyser Instance;

/// <summary>
/// Insert your Prediction Key here
/// </summary>
private string predictionKey = "- Insert your key here -";

/// <summary>
/// Insert your prediction endpoint here
/// </summary>
private string predictionEndpoint = "Insert your prediction endpoint here";

/// <summary>
/// Byte array of the image to submit for analysis
/// </summary>
[HideInInspector] public byte[] imageBytes;
```

NOTE

Make sure you insert your **Prediction Key** into the **predictionKey** variable and your **Prediction Endpoint** into the **predictionEndpoint** variable. You copied these to *Notepad* earlier in the course.

7. Code for **Awake()** now needs to be added to initialize the **Instance** variable:

```
/// <summary>
/// Initialises this class
/// </summary>
private void Awake()
{
    // Allows this instance to behave like a singleton
    Instance = this;
}
```

8. Delete the methods **Start()** and **Update()**.

9. Next, add the coroutine (with the static **GetImageAsByteArray()** method below it), which will obtain the results of the analysis of the image captured by the *ImageCapture* class.

NOTE

In the **AnalyseImageCapture** coroutine, there is a call to the *SceneOrganiser* class that you are yet to create. Therefore, **leave those lines commented for now**.

```

/// <summary>
/// Call the Computer Vision Service to submit the image.
/// </summary>
public IEnumerator AnalyseLastImageCaptured(string imagePath)
{
    WWWForm webForm = new WWWForm();
    using (UnityWebRequest unityWebRequest = UnityWebRequest.Post(predictionEndpoint, webForm))
    {
        // Gets a byte array out of the saved image
        imageBytes = GetImageAsByteArray(imagePath);

        unityWebRequest.SetRequestHeader("Content-Type", "application/octet-stream");
        unityWebRequest.SetRequestHeader("Prediction-Key", predictionKey);

        // The upload handler will help uploading the byte array with the request
        unityWebRequest.uploadHandler = new UploadHandlerRaw(imageBytes);
        unityWebRequest.uploadHandler.contentType = "application/octet-stream";

        // The download handler will help receiving the analysis from Azure
        unityWebRequest.downloadHandler = new DownloadHandlerBuffer();

        // Send the request
        yield return unityWebRequest.SendWebRequest();

        string jsonResponse = unityWebRequest.downloadHandler.text;

        // The response will be in JSON format, therefore it needs to be deserialized

        // The following lines refers to a class that you will build in later Chapters
        // Wait until then to uncomment these lines

        //AnalysisObject analysisObject = new AnalysisObject();
        //analysisObject = JsonConvert.DeserializeObject<AnalysisObject>(jsonResponse);
        //SceneOrganiser.Instance.SetTagsToLastLabel(analysisObject);
    }
}

/// <summary>
/// Returns the contents of the specified image file as a byte array.
/// </summary>
static byte[] GetImageAsByteArray(string imagePath)
{
    FileStream fileStream = new FileStream(imagePath, FileMode.Open, FileAccess.Read);

    BinaryReader binaryReader = new BinaryReader(fileStream);

    return binaryReader.ReadBytes((int)fileStream.Length);
}

```

10. Be sure to save your changes in **Visual Studio** before returning to **Unity**.

Chapter 7 - Create the CustomVisionObjects class

The class you will create now is the *CustomVisionObjects* class.

This script contains a number of objects used by other classes to serialize and deserialize the calls made to the *Custom Vision Service*.

WARNING

It is important that you take note of the endpoint that the Custom Vision Service provides you, as the below JSON structure has been set up to work with *Custom Vision Prediction v2.0*. If you have a different version, you may need to update the below structure.

To create this class:

1. Right-click inside the **Scripts** folder, then click **Create** > **C# Script**. Call the script *CustomVisionObjects*.
2. Double-click on the new **CustomVisionObjects** script to open it with **Visual Studio**.
3. Add the following namespaces to the top of the file:

```
using System;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Networking;
```

4. Delete the **Start()** and **Update()** methods inside the *CustomVisionObjects* class; this class should now be empty.
5. Add the following classes **outside** the *CustomVisionObjects* class. These objects are used by the *Newtonsoft* library to serialize and deserialize the response data:

```
// The objects contained in this script represent the deserialized version
// of the objects used by this application

/// <summary>
/// Web request object for image data
/// </summary>
class MultipartObject : IMultipartFormSection
{
    public string sectionName { get; set; }

    public byte[] sectionData { get; set; }

    public string fileName { get; set; }

    public string contentType { get; set; }
}

/// <summary>
/// JSON of all Tags existing within the project
/// contains the list of Tags
/// </summary>
public class Tags_RootObject
{
    public List<TagOfProject> Tags { get; set; }
    public int TotalTaggedImages { get; set; }
    public int TotalUntaggedImages { get; set; }
}

public class TagOfProject
{
    public string Id { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public int ImageCount { get; set; }
}

/// <summary>
/// JSON of Tag to associate to an image
/// Contains a list of hosting the tags,
/// since multiple tags can be associated with one image
/// </summary>
public class Tag_RootObject
{
    public List<Tag> Tags { get; set; }
}
```

```

public class Tag
{
    public string ImageId { get; set; }
    public string TagId { get; set; }
}

/// <summary>
/// JSON of Images submitted
/// Contains objects that host detailed information about one or more images
/// </summary>
public class ImageRootObject
{
    public bool IsBatchSuccessful { get; set; }
    public List<SubmittedImage> Images { get; set; }
}

public class SubmittedImage
{
    public string SourceUrl { get; set; }
    public string Status { get; set; }
    public ImageObject Image { get; set; }
}

public class ImageObject
{
    public string Id { get; set; }
    public DateTime Created { get; set; }
    public int Width { get; set; }
    public int Height { get; set; }
    public string ImageUri { get; set; }
    public string ThumbnailUri { get; set; }
}

/// <summary>
/// JSON of Service Iteration
/// </summary>
public class Iteration
{
    public string Id { get; set; }
    public string Name { get; set; }
    public bool IsDefault { get; set; }
    public string Status { get; set; }
    public string Created { get; set; }
    public string LastModified { get; set; }
    public string TrainedAt { get; set; }
    public string ProjectId { get; set; }
    public bool Exportable { get; set; }
    public string DomainId { get; set; }
}

/// <summary>
/// Predictions received by the Service after submitting an image for analysis
/// </summary>
[Serializable]
public class AnalysisObject
{
    public List<Prediction> Predictions { get; set; }
}

[Serializable]
public class Prediction
{
    public string TagName { get; set; }
    public double Probability { get; set; }
}

```

Chapter 8 - Create the VoiceRecognizer class

This class will recognize the voice input from the user.

To create this class:

1. Right-click inside the **Scripts** folder, then click **Create** > **C# Script**. Call the script *VoiceRecognizer*.
2. Double-click on the new **VoiceRecognizer** script to open it with **Visual Studio**.
3. Add the following namespaces above the *VoiceRecognizer* class:

```
using System;
using System.Collections.Generic;
using System.Linq;
using UnityEngine;
using UnityEngine.Windows.Speech;
```

4. Then add the following variables inside the *VoiceRecognizer* class, above the *Start()* method:

```
/// <summary>
/// Allows this class to behave like a singleton
/// </summary>
public static VoiceRecognizer Instance;

/// <summary>
/// Recognizer class for voice recognition
/// </summary>
internal KeywordRecognizer keywordRecognizer;

/// <summary>
/// List of Keywords registered
/// </summary>
private Dictionary<string, Action> _keywords = new Dictionary<string, Action>();
```

5. Add the **Awake()** and **Start()** methods, the latter of which will set up the user *keywords* to be recognized when associating a tag to an image:

```

/// <summary>
/// Called on initialization
/// </summary>
private void Awake()
{
    Instance = this;
}

/// <summary>
/// Runs at initialization right after Awake method
/// </summary>
void Start ()
{

    Array tagsArray = Enum.GetValues(typeof(CustomVisionTrainer.Tags));

    foreach (object tagWord in tagsArray)
    {
        _keywords.Add(tagWord.ToString(), () =>
        {
            // When a word is recognized, the following line will be called
            CustomVisionTrainer.Instance.VerifyTag(tagWord.ToString());
        });
    }

    _keywords.Add("Discard", () =>
    {
        // When a word is recognized, the following line will be called
        // The user does not want to submit the image
        // therefore ignore and discard the process
        ImageCapture.Instance.ResetImageCapture();
        keywordRecognizer.Stop();
    });
}

//Create the keyword recognizer
keywordRecognizer = new KeywordRecognizer(_keywords.Keys.ToArray());

// Register for the OnPhraseRecognized event
keywordRecognizer.OnPhraseRecognized += KeywordRecognizer_OnPhraseRecognized;
}

```

6. Delete the **Update()** method.

7. Add the following handler, which is called whenever voice input is recognized:

```

/// <summary>
/// Handler called when a word is recognized
/// </summary>
private void KeywordRecognizer_OnPhraseRecognized(PhraseRecognizedEventArgs args)
{
    Action keywordAction;
    // if the keyword recognized is in our dictionary, call that Action.
    if (_keywords.TryGetValue(args.text, out keywordAction))
    {
        keywordAction.Invoke();
    }
}

```

8. Be sure to save your changes in **Visual Studio** before returning to **Unity**.

NOTE

Do not worry about code which might appear to have an error, as you will provide further classes soon, which will fix these.

Chapter 9 - Create the CustomVisionTrainer class

This class will chain a series of web calls to train the *Custom Vision Service*. Each call will be explained in detail right above the code.

To create this class:

1. Right-click inside the **Scripts** folder, then click **Create** > **C# Script**. Call the script *CustomVisionTrainer*.
2. Double-click on the new *CustomVisionTrainer* script to open it with **Visual Studio**.
3. Add the following namespaces above the *CustomVisionTrainer* class:

```
using Newtonsoft.Json;
using System.Collections;
using System.Collections.Generic;
using System.IO;
using System.Text;
using UnityEngine;
using UnityEngine.Networking;
```

4. Then add the following variables inside the *CustomVisionTrainer* class, above the **Start()** method.

NOTE

The training URL used here is provided within the *Custom Vision Training 1.2* documentation, and has a structure of: <https://southcentralus.api.cognitive.microsoft.com/customvision/v1.2/Training/projects/{projectId}/>
For more information, visit the [Custom Vision Training v1.2 reference API](#).

WARNING

It is important that you take note of the endpoint that the Custom Vision Service provides you for the training mode, as the JSON structure used (within the **CustomVisionObjects** class) has been set up to work with *Custom Vision Training v1.2*. If you have a different version, you may need to update the *Objects* structure.

```

/// <summary>
/// Allows this class to behave like a singleton
/// </summary>
public static CustomVisionTrainer Instance;

/// <summary>
/// Custom Vision Service URL root
/// </summary>
private string url =
"https://southcentralus.api.cognitive.microsoft.com/customvision/v1.2/Training/projects/";

/// <summary>
/// Insert your prediction key here
/// </summary>
private string trainingKey = "- Insert your key here -";

/// <summary>
/// Insert your Project Id here
/// </summary>
private string projectId = "- Insert your Project Id here -";

/// <summary>
/// Byte array of the image to submit for analysis
/// </summary>
internal byte[] imageBytes;

/// <summary>
/// The Tags accepted
/// </summary>
internal enum Tags {Mouse, Keyboard}

/// <summary>
/// The UI displaying the training Chapters
/// </summary>
private TextMesh trainingUI_TextMesh;

```

IMPORTANT

Ensure that you add your **Service Key** (Training Key) value and **Project Id** value, which you noted down previously; these are the values you [collected from the portal earlier in the course \(Chapter 2, step 10 onwards\)](#).

- Add the following **Start()** and **Awake()** methods. Those methods are called on initialization and contain the call to set up the UI:

```

/// <summary>
/// Called on initialization
/// </summary>
private void Awake()
{
    Instance = this;
}

/// <summary>
/// Runs at initialization right after Awake method
/// </summary>
private void Start()
{
    trainingUI_TextMesh = SceneOrganiser.Instance.CreateTrainingUI("TrainingUI", 0.04f, 0, 4,
false);
}

```

- Delete the **Update()** method. This class will not need it.

7. Add the **RequestTagSelection()** method. This method is the first to be called when an image has been captured and stored in the device and is now ready to be submitted to the *Custom Vision Service*, to train it. This method displays, in the training UI, a set of keywords the user can use to tag the image that has been captured. It also alerts the *VoiceRecognizer* class to begin listening to the user for voice input.

```
internal void RequestTagSelection()
{
    trainingUI_TextMesh.gameObject.SetActive(true);
    trainingUI_TextMesh.text = $" \nUse voice command \nto choose between the following tags:
\nMouse\nKeyboard \nor say Discard";

    VoiceRecognizer.Instance.keywordRecognizer.Start();
}
```

8. Add the **VerifyTag()** method. This method will receive the voice input recognized by the **VoiceRecognizer** class and verify its validity, and then begin the training process.

```
/// <summary>
/// Verify voice input against stored tags.
/// If positive, it will begin the Service training process.
/// </summary>
internal void VerifyTag(string spokenTag)
{
    if (spokenTag == Tags.Mouse.ToString() || spokenTag == Tags.Keyboard.ToString())
    {
        trainingUI_TextMesh.text = $"Tag chosen: {spokenTag}";
        VoiceRecognizer.Instance.keywordRecognizer.Stop();
        StartCoroutine(SubmitImageForTraining(ImageCapture.Instance.filePath, spokenTag));
    }
}
```

9. Add the **SubmitImageForTraining()** method. This method will begin the Custom Vision Service training process. The first step is to retrieve the **Tag Id** from the Service which is associated with the validated speech input from the user. The **Tag Id** will then be uploaded along with the image.

```
/// <summary>
/// Call the Custom Vision Service to submit the image.
/// </summary>
public IEnumerator SubmitImageForTraining(string imagePath, string tag)
{
    yield return new WaitForSeconds(2);
    trainingUI_TextMesh.text = $"Submitting Image \nwith tag: {tag} \nto Custom Vision Service";
    string imageId = string.Empty;
    string tagId = string.Empty;

    // Retrieving the Tag Id relative to the voice input
    string getTagIdEndpoint = string.Format("{0}{1}/tags", url, projectId);
    using (UnityWebRequest www = UnityWebRequest.Get(getTagIdEndpoint))
    {
        www.SetRequestHeader("Training-Key", trainingKey);
        www.downloadHandler = new DownloadHandlerBuffer();
        yield return www.SendWebRequest();
        string jsonResponse = www.downloadHandler.text;

        Tags_RootObject tagRootObject = JsonConvert.DeserializeObject<Tags_RootObject>(jsonResponse);

        foreach (TagOfProject tOP in tagRootObject.Tags)
        {
            if (tOP.Name == tag)
            {
                tagId = tOP.Id;
            }
        }
    }
}
```

```

        }

    }

    // Creating the image object to send for training
    List<IMultipartFormSection> multipartList = new List<IMultipartFormSection>();
    MultipartObject multipartObject = new MultipartObject();
    multipartObject.contentType = "application/octet-stream";
    multipartObject.fileName = "";
    multipartObject.sectionData = GetImageAsByteArray(imagePath);
    multipartList.Add(multipartObject);

    string createImageFromDataEndpoint = string.Format("{0}{1}/images?tagIds={2}", url, projectId,
    tagId);

    using (UnityWebRequest www = UnityWebRequest.Post(createImageFromDataEndpoint, multipartList))
    {
        // Gets a byte array out of the saved image
        imageBytes = GetImageAsByteArray(imagePath);

        //unityWebRequest.SetRequestHeader("Content-Type", "application/octet-stream");
        www.SetRequestHeader("Training-Key", trainingKey);

        // The upload handler will help uploading the byte array with the request
        www.uploadHandler = new UploadHandlerRaw(imageBytes);

        // The download handler will help receiving the analysis from Azure
        www.downloadHandler = new DownloadHandlerBuffer();

        // Send the request
        yield return www.SendWebRequest();

        string jsonResponse = www.downloadHandler.text;

        ImageRootObject m = JsonConvert.DeserializeObject<ImageRootObject>(jsonResponse);
        imageId = m.Images[0].Image.Id;
    }
    trainingUI_TextMesh.text = "Image uploaded";
    StartCoroutine(TrainCustomVisionProject());
}

```

10. Add the **TrainCustomVisionProject()** method. Once the image has been submitted and tagged, this method will be called. It will create a new Iteration that will be trained with all the previous images submitted to the Service plus the image just uploaded. Once the training has been completed, this method will call a method to set the newly created **Iteration** as **Default**, so that the endpoint you are using for analysis is the latest trained iteration.

```

/// <summary>
/// Call the Custom Vision Service to train the Service.
/// It will generate a new Iteration in the Service
/// </summary>
public IEnumerator TrainCustomVisionProject()
{
    yield return new WaitForSeconds(2);

    trainingUI_TextMesh.text = "Training Custom Vision Service";

    WWWForm webForm = new WWWForm();

    string trainProjectEndpoint = string.Format("{0}{1}/train", url, projectId);

    using (UnityWebRequest www = UnityWebRequest.Post(trainProjectEndpoint, webForm))
    {
        www.SetRequestHeader("Training-Key", trainingKey);
        www.downloadHandler = new DownloadHandlerBuffer();
        yield return www.SendWebRequest();
        string jsonResponse = www.downloadHandler.text;
        Debug.Log($"Training - JSON Response: {jsonResponse}");

        // A new iteration that has just been created and trained
        Iteration iteration = new Iteration();
        iteration = JsonConvert.DeserializeObject<Iteration>(jsonResponse);

        if (www.isDone)
        {
            trainingUI_TextMesh.text = "Custom Vision Trained";

            // Since the Service has a limited number of iterations available,
            // we need to set the last trained iteration as default
            // and delete all the iterations you dont need anymore
            StartCoroutine(SetDefaultIteration(iteration));
        }
    }
}

```

11. Add the **SetDefaultIteration()** method. This method will set the previously created and trained iteration as *Default*. Once completed, this method will have to delete the previous iteration existing in the Service. At the time of writing this course, there is a limit of a maximum ten (10) iterations allowed to exist at the same time in the Service.

```

/// <summary>
/// Set the newly created iteration as Default
/// </summary>
private IEnumerator SetDefaultIteration(Iteration iteration)
{
    yield return new WaitForSeconds(5);
    trainingUI_TextMesh.text = "Setting default iteration";

    // Set the last trained iteration to default
    iteration.IsDefault = true;

    // Convert the iteration object as JSON
    string iterationAsJson = JsonConvert.SerializeObject(iteration);
    byte[] bytes = Encoding.UTF8.GetBytes(iterationAsJson);

    string setDefaultIterationEndpoint = string.Format("{0}{1}/iterations/{2}",
        url, projectId, iteration.Id);

    using (UnityWebRequest www = UnityWebRequest.Put(setDefaultIterationEndpoint, bytes))
    {
        www.method = "PATCH";
        www.SetRequestHeader("Training-Key", trainingKey);
        www.SetRequestHeader("Content-Type", "application/json");
        www.downloadHandler = new DownloadHandlerBuffer();

        yield return www.SendWebRequest();

        string jsonResponse = www.downloadHandler.text;

        if (www.isDone)
        {
            trainingUI_TextMesh.text = "Default iteration is set \nDeleting Unused Iteration";
            StartCoroutine>DeletePreviousIteration(iteration));
        }
    }
}

```

12. Add the **DeletePreviousIteration()** method. This method will find and delete the previous non-default iteration:

```

/// <summary>
/// Delete the previous non-default iteration.
/// </summary>
public IEnumerator DeletePreviousIteration(Iteration iteration)
{
    yield return new WaitForSeconds(5);

    trainingUI_TextMesh.text = "Deleting Unused \nIteration";

    string iterationToDeleteId = string.Empty;

    string findAllIterationsEndpoint = string.Format("{0}{1}/iterations", url, projectId);

    using (UnityWebRequest www = UnityWebRequest.Get(findAllIterationsEndpoint))
    {
        www.SetRequestHeader("Training-Key", trainingKey);
        www.downloadHandler = new DownloadHandlerBuffer();
        yield return www.SendWebRequest();

        string jsonResponse = www.downloadHandler.text;

        // The iteration that has just been trained
        List<Iteration> iterationsList = new List<Iteration>();
        iterationsList = JsonConvert.DeserializeObject<List<Iteration>>(jsonResponse);

        foreach (Iteration i in iterationsList)
        {
            if (i.IsDefault != true)
            {
                Debug.Log($"Cleaning - Deleting iteration: {i.Name}, {i.Id}");
                iterationToDeleteId = i.Id;
                break;
            }
        }
    }

    string deleteEndpoint = string.Format("{0}{1}/iterations/{2}", url, projectId,
iterationToDeleteId);

    using (UnityWebRequest www2 = UnityWebRequest.Delete(deleteEndpoint))
    {
        www2.SetRequestHeader("Training-Key", trainingKey);
        www2.downloadHandler = new DownloadHandlerBuffer();
        yield return www2.SendWebRequest();
        string jsonResponse = www2.downloadHandler.text;

        trainingUI_TextMesh.text = "Iteration Deleted";
        yield return new WaitForSeconds(2);
        trainingUI_TextMesh.text = "Ready for next \ncapture";

        yield return new WaitForSeconds(2);
        trainingUI_TextMesh.text = "";
        ImageCapture.Instance.ResetImageCapture();
    }
}

```

13. The last method to add in this class is the **GetImageAsByteArray()** method, used on the web calls to convert the image captured into a byte array.

```

/// <summary>
/// Returns the contents of the specified image file as a byte array.
/// </summary>
static byte[] GetImageAsByteArray(string imagePath)
{
    FileStream fileStream = new FileStream(imagePath, FileMode.Open, FileAccess.Read);
    BinaryReader binaryReader = new BinaryReader(fileStream);
    return binaryReader.ReadBytes((int)fileStream.Length);
}

```

14. Be sure to save your changes in **Visual Studio** before returning to **Unity**.

Chapter 10 - Create the SceneOrganiser class

This class will:

- Create a **Cursor** object to attach to the Main Camera.
- Create a **Label** object that will appear when the Service recognizes the real-world objects.
- Set up the Main Camera by attaching the appropriate components to it.
- When in **Analysis Mode**, spawn the Labels at runtime, in the appropriate world space relative to the position of the Main Camera, and display the data received from the Custom Vision Service.
- When in **Training Mode**, spawn the UI that will display the different stages of the training process.

To create this class:

1. Right-click inside the **Scripts** folder, then click **Create** > **C# Script**. Name the script *SceneOrganiser*.
2. Double-click on the new *SceneOrganiser* script to open it with **Visual Studio**.
3. You will only need one namespace, remove the others from above the *SceneOrganiser* class:

```
using UnityEngine;
```

4. Then add the following variables inside the *SceneOrganiser* class, above the **Start()** method:

```

/// <summary>
/// Allows this class to behave like a singleton
/// </summary>
public static SceneOrganiser Instance;

/// <summary>
/// The cursor object attached to the camera
/// </summary>
internal GameObject cursor;

/// <summary>
/// The label used to display the analysis on the objects in the real world
/// </summary>
internal GameObject label;

/// <summary>
/// Object providing the current status of the camera.
/// </summary>
internal TextMesh cameraStatusIndicator;

/// <summary>
/// Reference to the last label positioned
/// </summary>
internal Transform lastLabelPlaced;

/// <summary>
/// Reference to the last label positioned
/// </summary>
internal TextMesh lastLabelPlacedText;

/// <summary>
/// Current threshold accepted for displaying the label
/// Reduce this value to display the recognition more often
/// </summary>
internal float probabilityThreshold = 0.5f;

```

5. Delete the **Start()** and **Update()** methods.
6. Right underneath the variables, add the **Awake()** method, which will initialize the class and set up the scene.

```

/// <summary>
/// Called on initialization
/// </summary>
private void Awake()
{
    // Use this class instance as singleton
    Instance = this;

    // Add the ImageCapture class to this GameObject
    gameObject.AddComponent<ImageCapture>();

    // Add the CustomVisionAnalyser class to this GameObject
    gameObject.AddComponent<CustomVisionAnalyser>();

    // Add the CustomVisionTrainer class to this GameObject
    gameObject.AddComponent<CustomVisionTrainer>();

    // Add the VoiceRecogniser class to this GameObject
    gameObject.AddComponent<VoiceRecognizer>();

    // Add the CustomVisionObjects class to this GameObject
    gameObject.AddComponent<CustomVisionObjects>();

    // Create the camera Cursor
    cursor = CreateCameraCursor();

    // Load the label prefab as reference
    label = CreateLabel();

    // Create the camera status indicator label, and place it above where predictions
    // and training UI will appear.
    cameraStatusIndicator = CreateTrainingUI("Status Indicator", 0.02f, 0.2f, 3, true);

    // Set camera status indicator to loading.
    SetCameraStatus("Loading");
}

```

7. Now add the **CreateCameraCursor()** method that creates and positions the Main Camera cursor, and the **CreateLabel()** method, which creates the **Analysis Label** object.

```

/// <summary>
/// Spawns cursor for the Main Camera
/// </summary>
private GameObject CreateCameraCursor()
{
    // Create a sphere as new cursor
    GameObject newCursor = GameObject.CreatePrimitive(PrimitiveType.Sphere);

    // Attach it to the camera
    newCursor.transform.parent = gameObject.transform;

    // Resize the new cursor
    newCursor.transform.localScale = new Vector3(0.02f, 0.02f, 0.02f);

    // Move it to the correct position
    newCursor.transform.localPosition = new Vector3(0, 0, 4);

    // Set the cursor color to red
    newCursor.GetComponent<Renderer>().material = new Material(Shader.Find("Diffuse"));
    newCursor.GetComponent<Renderer>().material.color = Color.green;

    return newCursor;
}

/// <summary>
/// Create the analysis label object
/// </summary>
private GameObject CreateLabel()
{
    // Create a sphere as new cursor
    GameObject newLabel = new GameObject();

    // Resize the new cursor
    newLabel.transform.localScale = new Vector3(0.01f, 0.01f, 0.01f);

    // Creating the text of the label
    TextMesh t = newLabel.AddComponent<TextMesh>();
    t.anchor = TextAnchor.MiddleCenter;
    t.alignment = TextAlignment.Center;
    t.fontSize = 50;
    t.text = "";

    return newLabel;
}

```

8. Add the **SetCameraStatus()** method, which will handle messages intended for the text mesh providing the status of the camera.

```

///<summary>
/// Set the camera status to a provided string. Will be coloured if it matches a keyword.
///</summary>
///<param name="statusText">Input string</param>
public void SetCameraStatus(string statusText)
{
    if (string.IsNullOrEmpty(statusText) == false)
    {
        string message = "white";

        switch (statusText.ToLower())
        {
            case "loading":
                message = "yellow";
                break;

            case "ready":
                message = "green";
                break;

            case "uploading image":
                message = "red";
                break;

            case "looping capture":
                message = "yellow";
                break;

            case "analysis":
                message = "red";
                break;
        }

        cameraStatusIndicator.GetComponent<TextMesh>().text = $"Camera Status:\n<color={message}>
{statusText}..</color>";
    }
}

```

9. Add the **PlaceAnalysisLabel()** and **SetTagsToLastLabel()** methods, which will spawn and display the data from the Custom Vision Service into the scene.

```

/// <summary>
/// Instantiate a label in the appropriate location relative to the Main Camera.
/// </summary>
public void PlaceAnalysisLabel()
{
    lastLabelPlaced = Instantiate(label.transform, cursor.transform.position, transform.rotation);
    lastLabelPlacedText = lastLabelPlaced.GetComponent<TextMesh>();
}

/// <summary>
/// Set the Tags as Text of the last label created.
/// </summary>
public void SetTagsToLastLabel(AnalysisObject analysisObject)
{
    lastLabelPlacedText = lastLabelPlaced.GetComponent<TextMesh>();

    if (analysisObject.Predictions != null)
    {
        foreach (Prediction p in analysisObject.Predictions)
        {
            if (p.Probability > 0.02)
            {
                lastLabelPlacedText.text += $"Detected: {p.TagName} {p.Probability.ToString("0.00
\n")}";
                Debug.Log($"Detected: {p.TagName} {p.Probability.ToString("0.00 \n")}");
            }
        }
    }
}

```

10. Lastly, add the **CreateTrainingUI()** method, which will spawn the UI displaying the multiple stages of the training process when the application is in Training Mode. This method will also be harnessed to create the camera status object.

```

/// <summary>
/// Create a 3D Text Mesh in scene, with various parameters.
/// </summary>
/// <param name="name">name of object</param>
/// <param name="scale">scale of object (i.e. 0.04f)</param>
/// <param name="yPos">height above the cursor (i.e. 0.3f)</param>
/// <param name="zPos">distance from the camera</param>
/// <param name="setActive">whether the text mesh should be visible when it has been
created</param>
/// <returns>Returns a 3D text mesh within the scene</returns>
internal TextMesh CreateTrainingUI(string name, float scale, float yPos, float zPos, bool
setActive)
{
    GameObject display = new GameObject(name, typeof(TextMesh));
    display.transform.parent = Camera.main.transform;
    display.transform.localPosition = new Vector3(0, yPos, zPos);
    display.SetActive(setActive);
    display.transform.localScale = new Vector3(scale, scale, scale);
    display.transform.rotation = new Quaternion();
    TextMesh textMesh = display.GetComponent<TextMesh>();
    textMesh.anchor = TextAnchor.MiddleCenter;
    textMesh.alignment = TextAlignment.Center;
    return textMesh;
}

```

11. Be sure to save your changes in **Visual Studio** before returning to **Unity**.

IMPORTANT

Before you continue, open the **CustomVisionAnalyser** class, and within the **AnalyseLastImageCaptured()** method, *uncomment* the following lines:

```
AnalysisObject analysisObject = new AnalysisObject();
analysisObject = JsonConvert.DeserializeObject<AnalysisObject>(jsonResponse);
SceneOrganiser.Instance.SetTagsToLastLabel(analysisObject);
```

Chapter 11 - Create the ImageCapture class

The next class you are going to create is the *ImageCapture* class.

This class is responsible for:

- Capturing an image using the HoloLens camera and storing it in the *App* Folder.
- Handling Tap gestures from the user.
- Maintaining the *Enum* value that determines if the application will run in *Analysis* mode or *Training* Mode.

To create this class:

1. Go to the **Scripts** folder you created previously.
2. Right-click inside the folder, then click **Create > C# Script**. Name the script *ImageCapture*.
3. Double-click on the new **ImageCapture** script to open it with **Visual Studio**.
4. Replace the namespaces at the top of the file with the following:

```
using System;
using System.IO;
using System.Linq;
using UnityEngine;
using UnityEngine.XR.WSA.Input;
using UnityEngine.XR.WSA.WebCam;
```

5. Then add the following variables inside the *ImageCapture* class, above the **Start()** method:

```

/// <summary>
/// Allows this class to behave like a singleton
/// </summary>
public static ImageCapture Instance;

/// <summary>
/// Keep counts of the taps for image renaming
/// </summary>
private int captureCount = 0;

/// <summary>
/// Photo Capture object
/// </summary>
private PhotoCapture photoCaptureObject = null;

/// <summary>
/// Allows gestures recognition in HoloLens
/// </summary>
private GestureRecognizer recognizer;

/// <summary>
/// Loop timer
/// </summary>
private float secondsBetweenCaptures = 10f;

/// <summary>
/// Application main functionalities switch
/// </summary>
internal enum AppModes {Analysis, Training }

/// <summary>
/// Local variable for current AppMode
/// </summary>
internal AppModes AppMode { get; private set; }

/// <summary>
/// Flagging if the capture loop is running
/// </summary>
internal bool captureIsActive;

/// <summary>
/// File path of current analysed photo
/// </summary>
internal string filePath = string.Empty;

```

6. Code for **Awake()** and **Start()** methods now needs to be added:

```

/// <summary>
/// Called on initialization
/// </summary>
private void Awake()
{
    Instance = this;

    // Change this flag to switch between Analysis Mode and Training Mode
    AppMode = AppModes.Training;
}

/// <summary>
/// Runs at initialization right after Awake method
/// </summary>
void Start()
{
    // Clean up the LocalState folder of this application from all photos stored
    DirectoryInfo info = new DirectoryInfo(Application.persistentDataPath);
    var fileInfo = info.GetFiles();
    foreach (var file in fileInfo)
    {
        try
        {
            file.Delete();
        }
        catch (Exception)
        {
            Debug.LogFormat("Cannot delete file: ", file.Name);
        }
    }

    // Subscribing to the Hololens API gesture recognizer to track user gestures
    recognizer = new GestureRecognizer();
    recognizer.SetRecognizableGestures(GestureSettings.Tap);
    recognizer.Tapped += TapHandler;
    recognizer.StartCapturingGestures();

    SceneOrganiser.Instance.SetCameraStatus("Ready");
}

```

7. Implement a handler that will be called when a Tap gesture occurs.

```

/// <summary>
/// Respond to Tap Input.
/// </summary>
private void TapHandler(TappedEventArgs obj)
{
    switch (AppMode)
    {
        case AppModes.Analysis:
            if (!captureIsActive)
            {
                captureIsActive = true;

                // Set the cursor color to red
                SceneOrganiser.Instance.cursor.GetComponent<Renderer>().material.color = Color.red;

                // Update camera status to looping capture.
                SceneOrganiser.Instance.SetCameraStatus("Looping Capture");

                // Begin the capture loop
                InvokeRepeating("ExecuteImageCaptureAndAnalysis", 0, secondsBetweenCaptures);
            }
            else
            {
                // The user tapped while the app was analyzing
                // therefore stop the analysis process
                ResetImageCapture();
            }
            break;

        case AppModes.Training:
            if (!captureIsActive)
            {
                captureIsActive = true;

                // Call the image capture
                ExecuteImageCaptureAndAnalysis();

                // Set the cursor color to red
                SceneOrganiser.Instance.cursor.GetComponent<Renderer>().material.color = Color.red;

                // Update camera status to uploading image.
                SceneOrganiser.Instance.SetCameraStatus("Uploading Image");
            }
            break;
    }
}

```

NOTE

In *Analysis* mode, the **TapHandler** method acts as a switch to start or stop the photo capture loop.

In *Training* mode, it will capture an image from the camera.

When the cursor is green, it means the camera is available to take the image.

When the cursor is red, it means the camera is busy.

8. Add the method that the application uses to start the image capture process and store the image.

```

/// <summary>
/// Begin process of Image Capturing and send To Azure Custom Vision Service.
/// </summary>
private void ExecuteImageCaptureAndAnalysis()
{
    // Update camera status to analysis.
    SceneOrganiser.Instance.SetCameraStatus("Analysis");

    // Create a label in world space using the SceneOrganiser class
    // Invisible at this point but correctly positioned where the image was taken
    SceneOrganiser.Instance.PlaceAnalysisLabel();

    // Set the camera resolution to be the highest possible
    Resolution cameraResolution = PhotoCapture.SupportedResolutions.OrderByDescending((res) =>
res.width * res.height).First();

    Texture2D targetTexture = new Texture2D(cameraResolution.width, cameraResolution.height);

    // Begin capture process, set the image format
    PhotoCapture.CreateAsync(false, delegate (PhotoCapture captureObject)
    {
        photoCaptureObject = captureObject;

        CameraParameters camParameters = new CameraParameters
        {
            hologramOpacity = 0.0f,
            cameraResolutionWidth = targetTexture.width,
            cameraResolutionHeight = targetTexture.height,
            pixelFormat = CapturePixelFormat.BGRA32
        };

        // Capture the image from the camera and save it in the App internal folder
        captureObject.StartPhotoModeAsync(camParameters, delegate (PhotoCapture.PhotoCaptureResult
result)
        {
            string filename = string.Format(@"CapturedImage{0}.jpg", captureCount);
            filePath = Path.Combine(Application.persistentDataPath, filename);
            captureCount++;
            photoCaptureObject.TakePhotoAsync(filePath, PhotoCaptureFileOutputFormat.JPG,
OnCapturedPhotoToDisk);
        });
    });
}

```

9. Add the handlers that will be called when the photo has been captured and for when it is ready to be analyzed. The result is then passed to the *CustomVisionAnalyser* or the *CustomVisionTrainer* depending on which mode the code is set on.

```

/// <summary>
/// Register the full execution of the Photo Capture.
/// </summary>
void OnCapturedPhotoToDisk(PhotoCapture.PhotoCaptureResult result)
{
    // Call StopPhotoMode once the image has successfully captured
    photoCaptureObject.StopPhotoModeAsync(OnStoppedPhotoMode);
}

/// <summary>
/// The camera photo mode has stopped after the capture.
/// Begin the Image Analysis process.
/// </summary>
void OnStoppedPhotoMode(PhotoCapture.PhotoCaptureResult result)
{
    Debug.LogFormat("Stopped Photo Mode");

    // Dispose from the object in memory and request the image analysis
    photoCaptureObject.Dispose();
    photoCaptureObject = null;

    switch (AppMode)
    {
        case AppModes.Analysis:
            // Call the image analysis
            StartCoroutine(CustomVisionAnalyser.Instance.AnalyseLastImageCaptured(filePath));
            break;

        case AppModes.Training:
            // Call training using captured image
            CustomVisionTrainer.Instance.RequestTagSelection();
            break;
    }
}

/// <summary>
/// Stops all capture pending actions
/// </summary>
internal void ResetImageCapture()
{
    captureIsActive = false;

    // Set the cursor color to green
    SceneOrganiser.Instance.cursor.GetComponent<Renderer>().material.color = Color.green;

    // Update camera status to ready.
    SceneOrganiser.Instance.SetCameraStatus("Ready");

    // Stop the capture loop if active
    CancelInvoke();
}

```

10. Be sure to save your changes in **Visual Studio** before returning to **Unity**.

11. Now that all the scripts have been completed, go back in the Unity Editor, then click and drag the **SceneOrganiser** class from the **Scripts** folder to the **Main Camera** object in the *Hierarchy Panel*.

Chapter 12 - Before building

To perform a thorough test of your application you will need to sideload it onto your HoloLens.

Before you do, ensure that:

- All the settings mentioned in the [Chapter 2](#) are set correctly.

- All the fields in the **Main Camera**, Inspector Panel, are assigned properly.
- The script **SceneOrganiser** is attached to the **Main Camera** object.
- Make sure you insert your **Prediction Key** into the **predictionKey** variable.
- You have inserted your **Prediction Endpoint** into the **predictionEndpoint** variable.
- You have inserted your **Training Key** into the **trainingKey** variable, of the *CustomVisionTrainer* class.
- You have inserted your **Project ID** into the **projectId** variable, of the *CustomVisionTrainer* class.

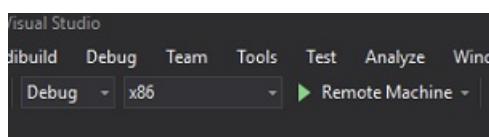
Chapter 13 - Build and sideload your application

To begin the *Build* process:

1. Go to **File > Build Settings**.
2. Tick **Unity C# Projects**.
3. Click **Build**. Unity will launch a **File Explorer** window, where you need to create and then select a folder to build the app into. Create that folder now, and name it **App**. Then with the **App** folder selected, click on **Select Folder**.
4. Unity will begin building your project to the **App** folder.
5. Once Unity has finished building (it might take some time), it will open a **File Explorer** window at the location of your build (check your task bar, as it may not always appear above your windows, but will notify you of the addition of a new window).

To deploy on HoloLens:

1. You will need the IP Address of your HoloLens (for Remote Deploy), and to ensure your HoloLens is in **Developer Mode**. To do this:
 - a. Whilst wearing your HoloLens, open the **Settings**.
 - b. Go to **Network & Internet > Wi-Fi > Advanced Options**
 - c. Note the **IPv4** address.
 - d. Next, navigate back to **Settings**, and then to **Update & Security > For Developers**
 - e. Set **Developer Mode On**.
2. Navigate to your new Unity build (the **App** folder) and open the solution file with **Visual Studio**.
3. In the *Solution Configuration* select **Debug**.
4. In the *Solution Platform*, select **x86, Remote Machine**. You will be prompted to insert the **IP address** of a remote device (the HoloLens, in this case, which you noted).



5. Go to **Build** menu and click on **Deploy Solution** to sideload the application to your HoloLens.
6. Your app should now appear in the list of installed apps on your HoloLens, ready to be launched!

NOTE

To deploy to immersive headset, set the **Solution Platform** to *Local Machine*, and set the **Configuration** to *Debug*, with *x86* as the **Platform**. Then deploy to the local machine, using the **Build** menu item, selecting *Deploy Solution*.

To use the application:

To switch the app functionality between *Training* mode and *Prediction* mode you need to update the **AppMode** variable, located in the **Awake()** method located within the *ImageCapture* class.

```
// Change this flag to switch between Analysis mode and Training mode  
AppMode = AppModes.Training;
```

or

```
// Change this flag to switch between Analysis mode and Training mode  
AppMode = AppModes.Analysis;
```

In *Training* mode:

- Look at **Mouse** or **Keyboard** and use the **Tap gesture**.
- Next, text will appear asking you to provide a tag.
- Say either **Mouse** or **Keyboard**.

In *Prediction* mode:

- Look at an object and use the **Tap gesture**.
- Text will appear providing the object detected, with the highest probability (this is normalized).

Chapter 14 - Evaluate and improve your Custom Vision model

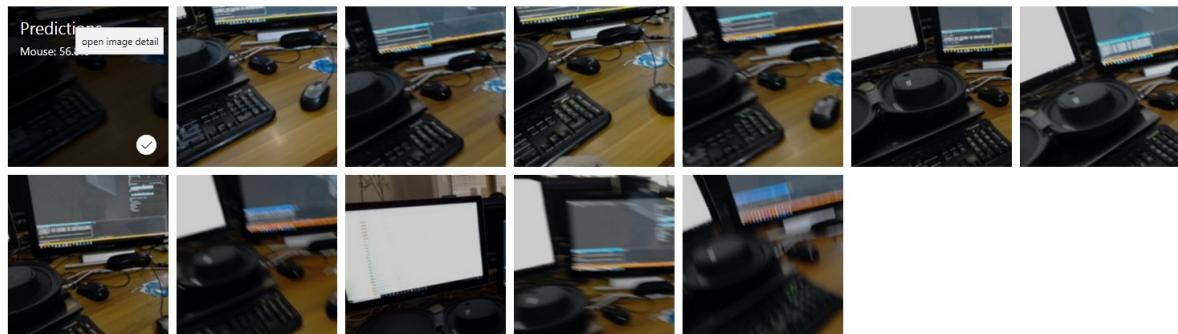
To make your Service more accurate, you will need to continue to train the model used for prediction. This is accomplished through using your new application, with both the *training* and *prediction* modes, with the latter requiring you to visit the portal, which is what is covered in this Chapter. Be prepared to revisit your portal many times, to continually improve your model.

1. Head to your Azure Custom Vision Portal again, and once you are in your project, select the *Predictions* tab (from the top center of the page):



2. You will see all the images which were sent to your Service whilst your application was running. If you hover over the images, they will provide you with the predictions that were made for that image:

 Delete  Tag images



3. Select one of your images to open it. Once open, you will see the predictions made for that image to the right. If you the predictions were correct, and you wish to add this image to your Service's training model, click the *My Tags* input box, and select the tag you wish to associate. When you are finished, click the *Save and close* button to the bottom right, and continue on to the next image.

My Tags
Add a tag and press enter
Mouse X

Predictions

Tag	Probability
Mouse	56.8%

Save and close

4. Once you are back to the grid of images, you will notice the images which you have added tags to (and saved), will be removed. If you find any images which you think do not have your tagged item within them, you can delete them, by clicking the tick on that image (can do this for several images) and then clicking *Delete* in the upper right corner of the grid page. On the popup that follows, you can click either *Yes*, *delete* or *No*, to confirm the deletion or cancel it, respectively.

Delete image?
Are you sure you want to delete these images?

Yes, delete No

5. When you are ready to proceed, click the green *Train* button in the top right. Your Service model will be trained with all the images which you have now provided (which will make it more accurate). Once the training is complete, make sure to click the *Make default* button once more, so that your *Prediction URL* continues to use the most up-to-date iteration of your Service.

The screenshot shows the 'Iteration 2' page of the Azure Custom Vision Service. At the top, there are two buttons: 'Train' (green) and 'Quick Test' (blue). Below them is a row of icons: 'Prediction URL' (globe), 'Make default' (checkmark), 'Delete' (trash can), and 'Export' (down arrow). The 'Make default' button is highlighted with a red box. The main content area displays the text 'Your finished Custom Vision API application' and a message congratulating the user on building a mixed reality app that leverages the Azure Custom Vision API to recognize real world objects, train the Service model, and display confidence of what has been seen. Below this text is a photograph of a Microsoft Mouse 1.00 on a wooden desk, with a keyboard partially visible to the left.

Your finished Custom Vision API application

Congratulations, you built a mixed reality app that leverages the Azure Custom Vision API to recognize real world objects, train the Service model, and display confidence of what has been seen.



Bonus exercises

Exercise 1

Train your **Custom Vision Service** to recognize more objects.

Exercise 2

As a way to expand on what you have learned, complete the following exercises:

Play a sound when an object is recognized.

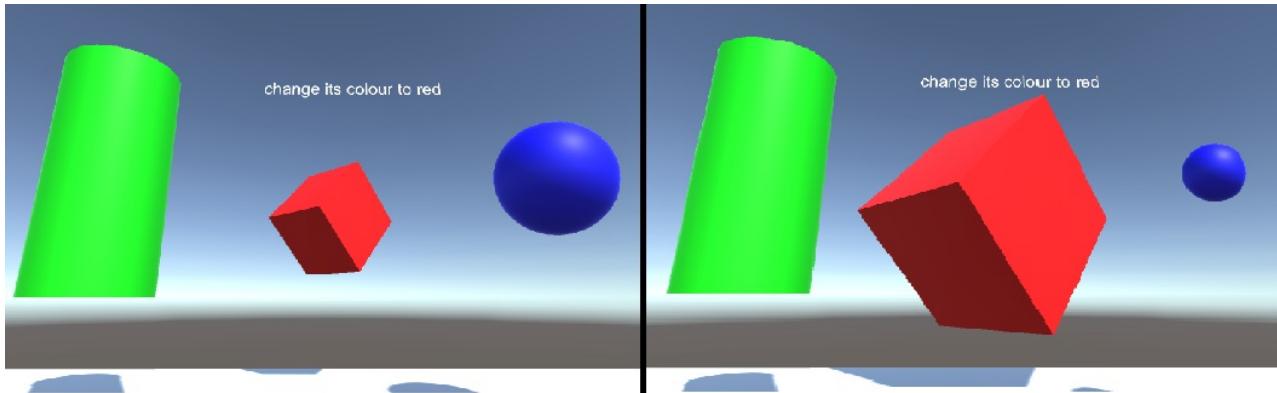
Exercise 3

Use the API to re-train your Service with the same images your app is analyzing, so to make the Service more accurate (do both prediction and training simultaneously).

MR and Azure 303: Natural language understanding (LUIS)

11/6/2018 • 30 minutes to read • [Edit Online](#)

In this course, you will learn how to integrate Language Understanding into a mixed reality application using Azure Cognitive Services, with the Language Understanding API.



Language Understanding (LUIS) is a Microsoft Azure service, which provides applications with the ability to make meaning out of user input, such as through extracting what a person might want, in their own words. This is achieved through machine learning, which understands and learns the input information, and then can reply with detailed, relevant, information. For more information, visit the [Azure Language Understanding \(LUIS\) page](#).

Having completed this course, you will have a mixed reality immersive headset application which will be able to do the following:

1. Capture user input speech, using the Microphone attached to the immersive headset.
2. Send the captured dictation the *Azure Language Understanding Intelligent Service (LUIS)*.
3. Have LUIS extract meaning from the send information, which will be analyzed, and attempt to determine the intent of the user's request will be made.

Development will include the creation of an app where the user will be able to use voice and/or gaze to change the size and the color of the objects in the scene. The use of motion controllers will not be covered.

In your application, it is up to you as to how you will integrate the results with your design. This course is designed to teach you how to integrate an Azure Service with your Unity Project. It is your job to use the knowledge you gain from this course to enhance your mixed reality application.

Be prepared to Train LUIS several times, which is covered in [Chapter 12](#). You will get better results the more times LUIS has been trained.

Device support

COURSE	HOLOLENS	IMMERSIVE HEADSETS
MR and Azure 303: Natural language understanding (LUIS)	✓ <input type="checkbox"/>	✓ <input type="checkbox"/>

NOTE

While this course primarily focuses on Windows Mixed Reality immersive (VR) headsets, you can also apply what you learn in this course to Microsoft HoloLens. As you follow along with the course, you will see notes on any changes you might need to employ to support HoloLens. When using HoloLens, you may notice some echo during voice capture.

Prerequisites

NOTE

This tutorial is designed for developers who have basic experience with Unity and C#. Please also be aware that the prerequisites and written instructions within this document represent what has been tested and verified at the time of writing (May 2018). You are free to use the latest software, as listed within the [install the tools](#) article, though it should not be assumed that the information in this course will perfectly match what you'll find in newer software than what's listed below.

We recommend the following hardware and software for this course:

- A development PC, [compatible with Windows Mixed Reality](#) for immersive (VR) headset development
- [Windows 10 Fall Creators Update \(or later\)](#) with Developer mode enabled
- [The latest Windows 10 SDK](#)
- [Unity 2017.4](#)
- [Visual Studio 2017](#)
- A [Windows Mixed Reality immersive \(VR\) headset](#) or [Microsoft HoloLens](#) with Developer mode enabled
- A set of headphones with a built-in microphone (if the headset doesn't have a built-in mic and speakers)
- Internet access for Azure setup and LUIS retrieval

Before you start

1. To avoid encountering issues building this project, it is strongly suggested that you create the project mentioned in this tutorial in a root or near-root folder (long folder paths can cause issues at build-time).
2. To allow your machine to enable Dictation, go to **Windows Settings > Privacy > Speech, Inking & Typing** and press on the button **Turn On speech services and typing suggestions**.
3. The code in this tutorial will allow you to record from the **Default Microphone Device** set on your machine. Make sure the Default Microphone Device is set as the one you wish to use to capture your voice.
4. If your headset has a built-in microphone, make sure the option "*When I wear my headset, switch to headset mic*" is turned on in the *Mixed Reality Portal* settings.

Home

Find a setting

Mixed reality

Audio and speech

Environment

Headset display

Uninstall

Audio and speech

Audio

Windows Mixed Reality works best with your device's built-in microphone and speakers, or with a headset connected to its audio port.

[Learn more](#)

When I wear my headset, switch to headset audio

On

When I wear my headset, switch to headset mic

On

Speech

Use speech recognition in Windows Mixed Reality. Speech recognition will always listen when Mixed Reality is running.

[Learn more](#)

Chapter 1 – Setup Azure Portal

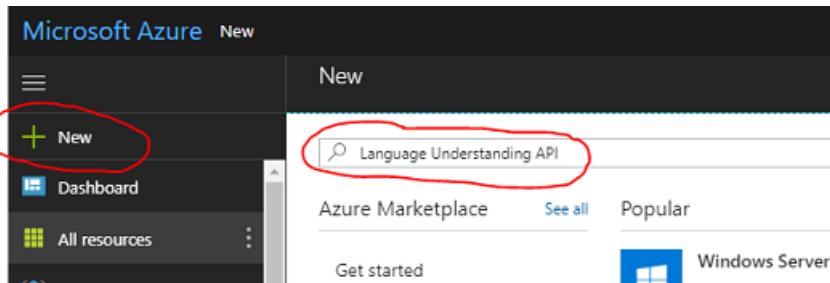
To use the *Language Understanding* service in Azure, you will need to configure an instance of the service to be made available to your application.

1. Log in to the [Azure Portal](#).

NOTE

If you do not already have an Azure account, you will need to create one. If you are following this tutorial in a classroom or lab situation, ask your instructor or one of the proctors for help setting up your new account.

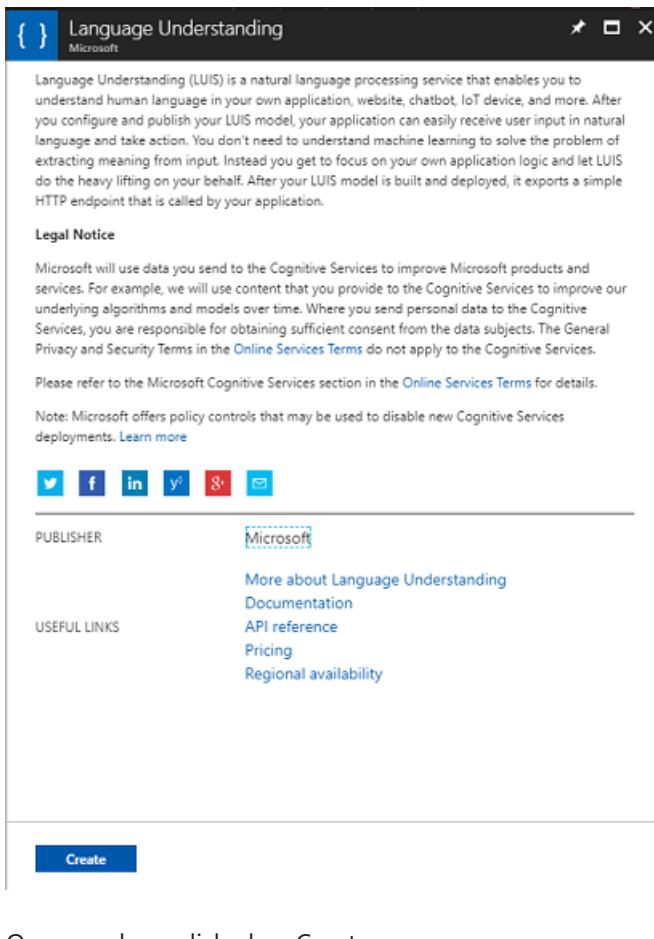
2. Once you are logged in, click on **New** in the top left corner, and search for *Language Understanding*, and click **Enter**.



NOTE

The word **New** may have been replaced with **Create a resource**, in newer portals.

3. The new page to the right will provide a description of the Language Understanding service. At the bottom left of this page, select the **Create** button, to create an instance of this service.



4. Once you have clicked on Create:

- a. Insert your desired **Name** for this service instance.
 - b. Select a **Subscription**.
 - c. Select the **Pricing Tier** appropriate for you, if this is the first time creating a *LUIS Service*, a free tier (named F0) should be available to you. The free allocation should be more than sufficient for this course.
 - d. Choose a **Resource Group** or create a new one. A resource group provides a way to monitor, control access, provision and manage billing for a collection of Azure assets. It is recommended to keep all the Azure services associated with a single project (e.g. such as these courses) under a common resource group.
- If you wish to read more about Azure Resource Groups, please [visit the resource group article](#).
- e. Determine the **Location** for your resource group (if you are creating a new Resource Group). The location would ideally be in the region where the application would run. Some Azure assets are only available in certain regions.
 - f. You will also need to confirm that you have understood the Terms and Conditions applied to this Service.
 - g. Select **Create**.

Create

Language Understanding (LUIS)

* Name
MyNewLUIS

* Subscription
Main Subscription

* Location
West US

* Pricing tier (View full pricing details)
S0 (50 Calls per second)

* Resource group
 Create new Use existing
AI_For_MR

* I confirm I have read and understood the notice below.

Microsoft will use data you send to the Cognitive Services to improve Microsoft products and services. Where you send personal data to the Cognitive Services, you are responsible for obtaining sufficient consent from the data subjects. The General Privacy and Security Terms in the Online Services Terms do not apply to the Cognitive Services. Please refer to the Microsoft Cognitive Services section in the [Online Services Terms](#) for details. Microsoft offers policy controls that may be used to disable new Cognitive Services deployments.

Pin to dashboard

Create Automation options

5. Once you have clicked on **Create**, you will have to wait for the service to be created, this might take a minute.
6. A notification will appear in the portal once the Service instance is created.



7. Click on the notification to explore your new Service instance.



8. Click the **Go to resource** button in the notification to explore your new Service instance. You will be taken to your new LUIS service instance.



Congratulations! Your keys are ready.

Now explore the Quickstart guidance to get up and running with Language Understanding.

1 Grab your keys

Every call to Language Understanding requires a subscription key. This key needs to be either passed through a menu.

[Keys](#)

2 Make an API call

Get in-depth information about each properties and methods of the API. Test your keys with the built-in testing Azure portal in your API 'Overview'. Go to the Language Understanding Portal to start to use the service.

[API reference](#)

[Language Understanding Portal](#)

3 Enjoy coding

Learn more about the features, tutorials, developer tools, examples and how-to guidance to speed up.

[Documentation](#)

[SDKs for Android](#)

[SDKs for Windows](#)

[SDKs for Node.js](#)

[SDKs for Python](#)

Additional resources

[Region availability](#)

[Support options](#)

[Provide feedback](#)

9. Within this tutorial, your application will need to make calls to your service, which is done through using your service's Subscription Key.
10. From the *Quick start* page, of your *Luis API* service, navigate to the first step, *Grab your keys*, and click **Keys** (you can also achieve this by clicking the blue hyperlink Keys, located in the services navigation menu, denoted by the key icon). This will reveal your service *Keys*.
11. Take a copy of one of the displayed keys, as you will need this later in your project.
12. In the *Service* page, click on *Language Understanding Portal* to be redirected to the webpage which you will use to create your new Service, within the LUIS App.

Chapter 2 – The Language Understanding Portal

In this section you will learn how to make a LUIS App on the LUIS Portal.

IMPORTANT

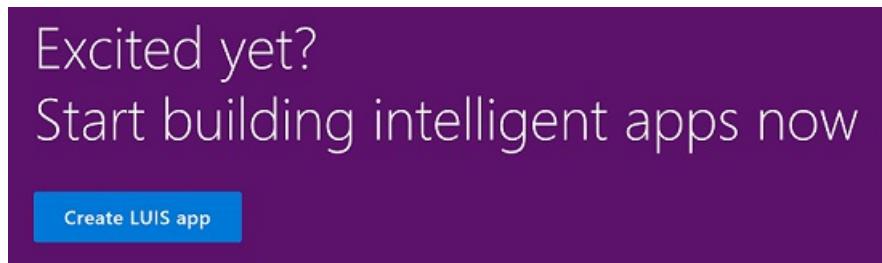
Please be aware, that setting up the *Entities*, *Intents*, and *Utterances* within this chapter is only the first step in building your LUIS service: you will also need to retrain the service, several times, so to make it more accurate. Retraining your service is covered in the [last Chapter](#) of this course, so ensure that you complete it.

1. Upon reaching the *Language Understanding Portal*, you may need to login, if you are not already, with the same credentials as your Azure portal.



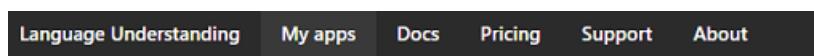
The image shows the Microsoft Language Understanding (LUIS) welcome page. It features a purple header with the LUIS logo and navigation links. Below the header, there's a main content area with a heading "Language Understanding (LUIS)" and a brief description: "A machine learning-based service to build natural language into apps, bots, and IoT devices. Quickly create enterprise-ready, custom models that continuously improve." A "Login / Sign up" button is located at the bottom left of the main content area.

2. If this is your first time using LUIS, you will need to scroll down to the bottom of the welcome page, to find and click on the **Create LUIS app** button.



The image shows a large blue button with white text that reads "Create LUIS app". Above the button, there is a promotional message: "Excited yet? Start building intelligent apps now".

3. Once logged in, click **My apps** (if you are not in that section currently). You can then click on **Create new app**.

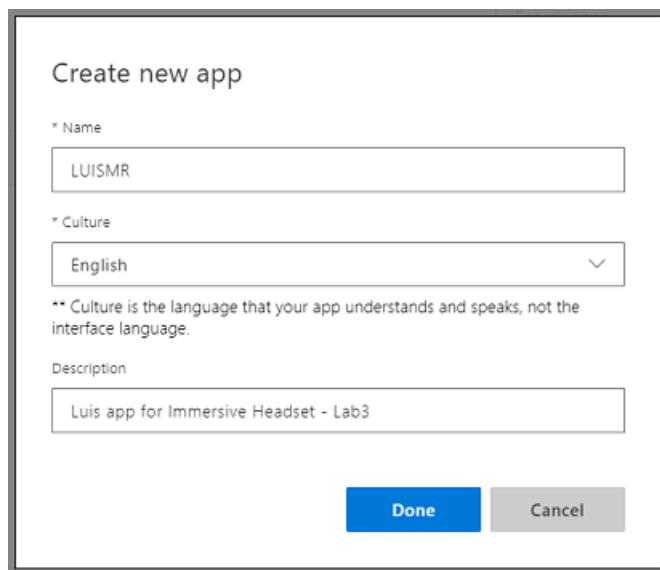


The image shows a dark navigation bar with white text. The items listed are: Language Understanding, My apps (which is highlighted in blue), Docs, Pricing, Support, and About.

My Apps ?

[Create new app](#) [Import new app](#)

4. Give your app a *Name*.
5. If your app is supposed to understand a language different from English, you should change the *Culture* to the appropriate language.
6. Here you can also add a *Description* of your new LUIS app.



The image shows a modal dialog box titled "Create new app". It contains the following fields:

- * Name: A text input field containing "LUISMR".
- * Culture: A dropdown menu set to "English".

** Culture is the language that your app understands and speaks, not the interface language.
- Description: A text input field containing "Luis app for Immersive Headset - Lab3".

At the bottom of the dialog are two buttons: "Done" (in blue) and "Cancel".

7. Once you press **Done**, you will enter the *Build* page of your new *LUIS* application.

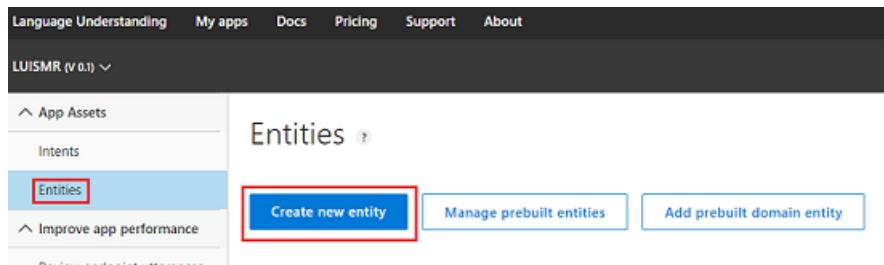
8. There are a few important concepts to understand here:

- *Intent*, represents the method that will be called following a query from the user. An *INTENT* may have one or more *ENTITIES*.
- *Entity*, is a component of the query that describes information relevant to the *INTENT*.
- *Utterances*, are examples of queries provided by the developer, that LUIS will use to train itself.

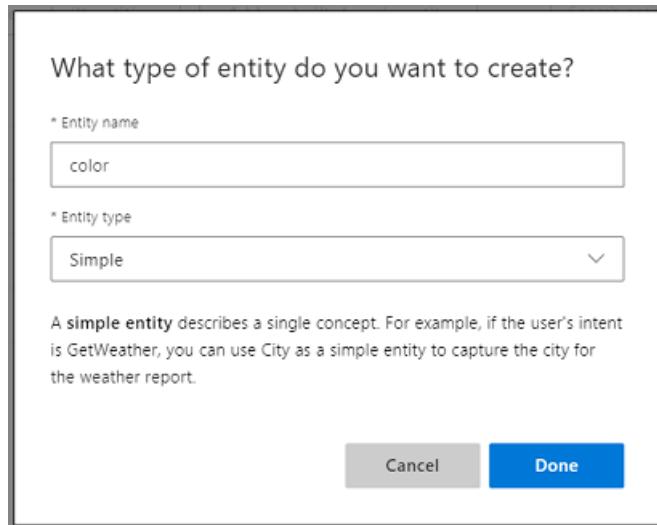
If these concepts are not perfectly clear, do not worry, as this course will clarify them further in this chapter.

You will begin by creating the *Entities* needed to build this course.

9. On the left side of the page, click on *Entities*, then click on **Create new entity**.



10. Call the new Entity *color*, set its type to *Simple*, then press **Done**.



11. Repeat this process to create three (3) more Simple Entities named:

- *upsize*
- *downsize*
- *target*

The result should look like the image below:

Entities

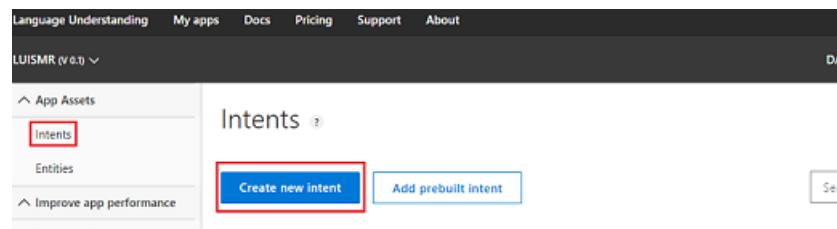
Name	Type
upsize	Simple
target	Simple
downsize	Simple
color	Simple

At this point you can begin creating *Intents*.

WARNING

Do not delete the **None** intent.

12. On the left side of the page, click on **Intents**, then click on **Create new intent**.

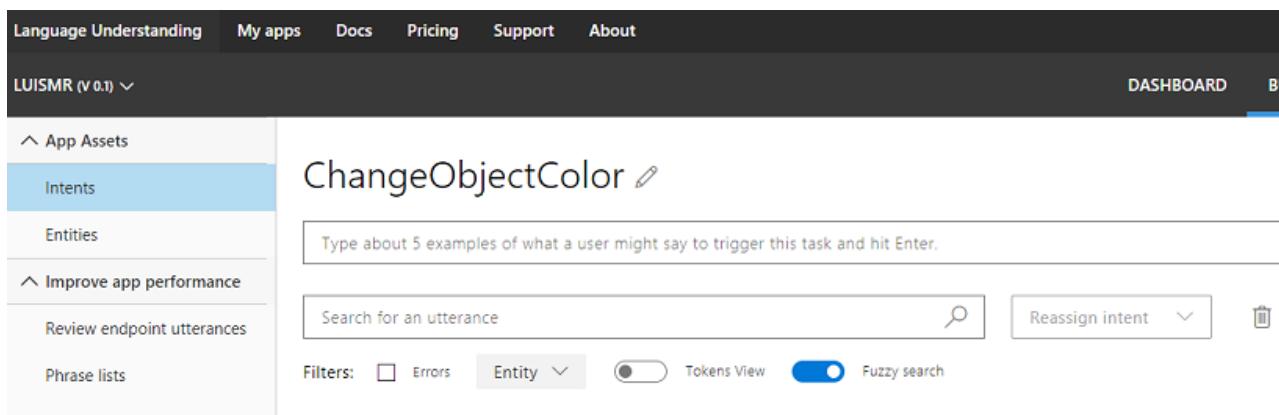


13. Call the new *Intent* **ChangeObjectColor**.

IMPORTANT

This *Intent* name is used within the code later in this course, so for best results, use this name exactly as provided.

Once you confirm the name you will be directed to the Intents Page.



You will notice that there is a textbox asking you to type 5 or more different *Utterances*.

NOTE

Luis converts all Utterances to lower case.

14. Insert the following *Utterance* in the top textbox (currently with the text *Type about 5 examples...*), and press **Enter**:

The color of the cylinder must be red

You will notice that the new *Utterance* will appear in a list underneath.

Following the same process, insert the following six (6) Utterances:

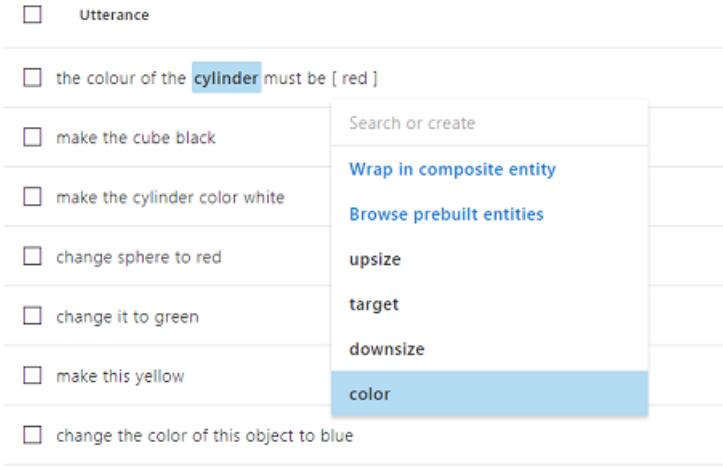
make the cube black
make the cylinder color white
change the sphere to red
change it to green
make this yellow
change the color of this object to blue

For each Utterance you have created, you must identify which words should be used by LUIS as Entities. In this example you need to label all the colors as a *color* Entity, and all the possible reference to a target as a *target* Entity.

15. To do so, try clicking on the word *cylinder* in the first Utterance and select *target*.

The screenshot shows the LUIS builder interface. A context menu is open over the word "cylinder" in the first item of a list. The menu items are: Utterance, the colour of the [cylinder] must be red, make the cube bl, make the cylinder, change sphere to, change it to gree, make this yellow, and change the color of this object to blue. The "target" option is highlighted with a blue background and a white border, indicating it is selected.

16. Now click on the word *red* in the first Utterance and select *color*.



17. Label the next line also, where *cube* should be a *target*, and *black* should be a *color*. Notice also the use of the words '*this*', '*it*', and '*this object*', which we are providing, so to have non-specific target types available also.
18. Repeat the process above until all the Utterances have the Entities labelled. See the below image if you need help.

TIP

When selecting words to label them as entities:

- For single words just click them.
- For a set of two or more words, click at the beginning and then at the end of the set.

NOTE

You can use the *Tokens View* toggle button to switch between **Entities / Tokens View!**

19. The results should be as seen in the images below, showing the **Entities / Tokens View**:

ChangeObjectColor ↴

Type about 5 examples of what a user might say to trigger th

Search for an utterance

Filters: Errors Entity Tokens View

Utterance

the colour of the cylinder must be red

make the cube black

make the cylinder color white

change sphere to red

change it to green

make this yellow

change the color of this object to blue

ChangeObjectColor ↴

Type about 5 examples of what a user might say to trigger th

Search for an utterance

Filters: Errors Entity Entities view

Utterance

the colour of the target must be color

make the target color

make the target color color

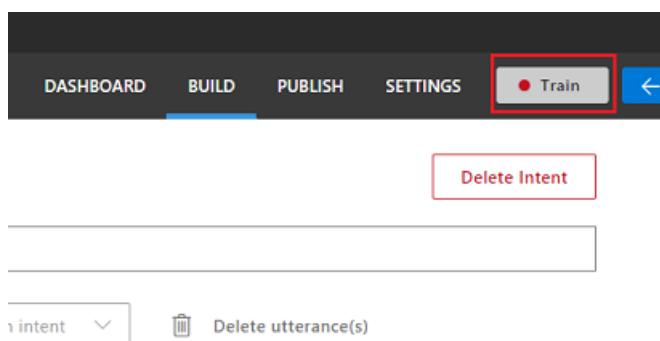
change target to color

change target to color

make target color

change the color of target to color

20. At this point press the **Train** button at the top-right of the page and wait for the small round indicator on it to turn green. This indicates that LUIS has been successfully trained to recognize this Intent.



21. As an exercise for you, create a new Intent called **ChangeObjectSize**, using the Entities *target*, *upscale*, and *downsize*.

22. Following the same process as the previous Intent, insert the following eight (8) Utterances for Size change:

```
increase the dimensions of that
reduce the size of this
i want the sphere smaller
make the cylinder bigger
size down the sphere
size up the cube
decrease the size of that object
increase the size of this object
```

23. The result should be like the one in the image below:

ChangeObjectSize ↕

The screenshot shows the LUIS Intent builder interface. At the top, there is a search bar labeled "Search for an utterance". Below it, a "Filters" section includes checkboxes for "Errors" and "Entity" (with a dropdown menu), and a "Tokens View" toggle switch. A list of utterances is displayed, each with a checkbox and a link to edit:

- Utterance
- increase the dimensions of that
- reduce the size of this
- i want the sphere smaller
- make the cylinder bigger
- size down the sphere
- size up the cube
- decrease the size of that object
- increase the size of this object

24. Once both Intents, **ChangeObjectColor** and **ChangeObjectSize**, have been created and trained, click on the **PUBLISH** button on top of the page.

The screenshot shows the LUIS Publish page. At the top, there are tabs for "DASHBOARD", "BUILD", "PUBLISH" (which is highlighted with a red box), and "SETTINGS". Below the tabs, there is a "Train" button with a red dot and a back arrow. In the center, there is a "Delete Intent" button with a red box around it. Below the button is a large empty text area. At the bottom, there is a dropdown menu labeled "intent" with a red box around it, a "Delete utterance(s)" button, and a "Delete" icon.

25. On the *Publish* page you will finalize and publish your LUIS App so that it can be accessed by your code.

- Set the drop down *Publish To* as **Production**.
- Set the *Timezone* to your time zone.
- Check the box **Include all predicted intent scores**.
- Click on **Publish to Production Slot**.

The screenshot shows the LUIS Publish page with the "PUBLISH" tab selected. At the top, there is a navigation bar with links for "Language Understanding", "My apps", "Docs", "Pricing", "Support", and "About". Below the navigation bar, it says "LUISMR (v0.1) ~". In the center, it displays "Published version: 0.1" and "Published date: Jan 25, 2018, 10:19:00 AM (2 hour(s) ago)". There are four numbered steps outlined with red boxes:

1. **Publish to**: A dropdown menu set to "Production".
2. **Timezone:** A dropdown menu set to "(GMT +10:00) Eastern Australia, Guam, Vladivostok".
3. **Include all predicted intent scores**: A checked checkbox with a question mark icon.
4. **Publish to production slot**: A blue "Publish" button.

26. In the section *Resources and Keys*:

- a. Select the region you set for service instance in the Azure Portal.
- b. You will notice a **Starter_Key** element below, ignore it.
- c. Click on **Add Key** and insert the *Key* that you obtained in the Azure Portal when you created your Service instance. If your Azure and the LUIS portal are logged into the same user, you will be provided drop-down menus for *Tenant name*, *Subscription Name*, and the *Key* you wish to use (will have the same name as you provided previously in the Azure Portal).

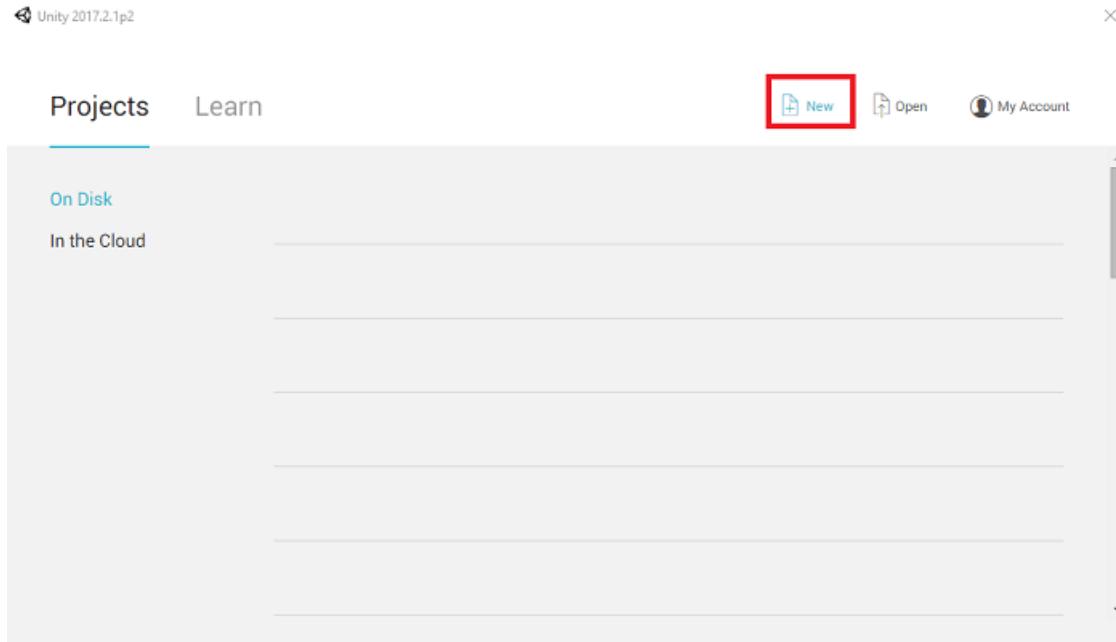
IMPORTANT

Underneath *Endpoint*, take a copy of the endpoint corresponding to the *Key* you have inserted, you will soon use it in your code.

Chapter 3 – Set up the Unity project

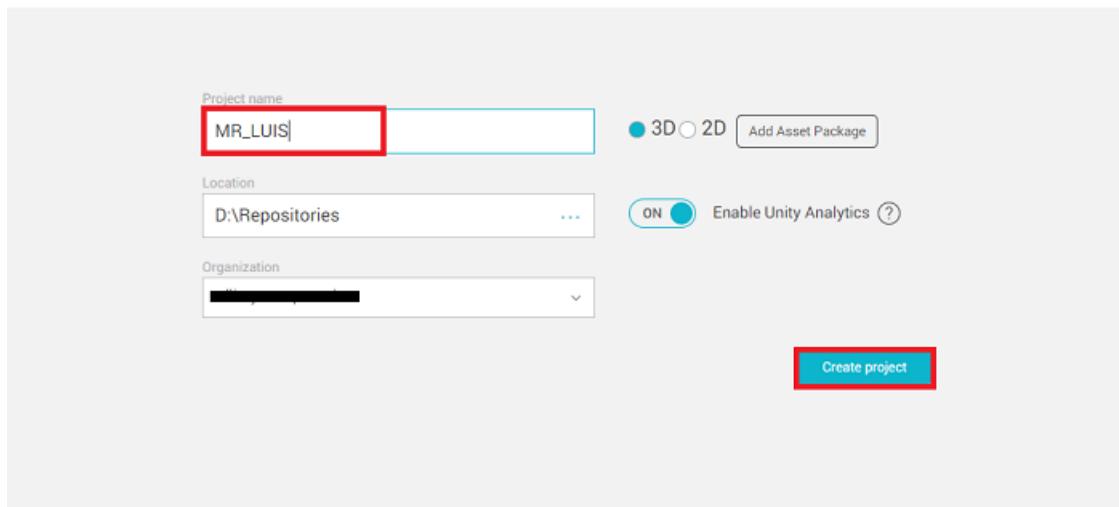
The following is a typical set up for developing with the mixed reality, and as such, is a good template for other projects.

1. Open *Unity* and click **New**.

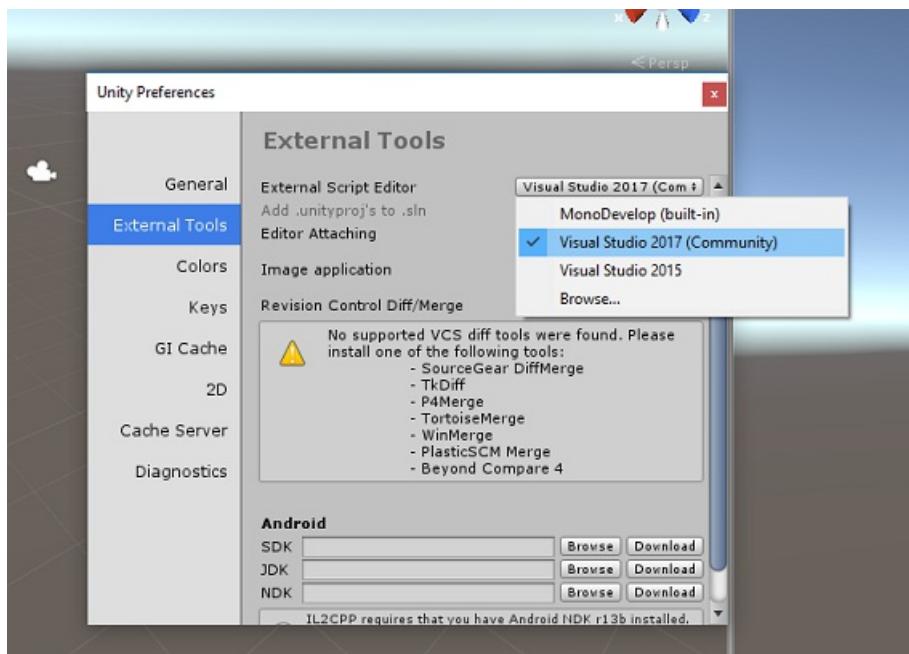


2. You will now need to provide a Unity Project name, insert **MR_LUIS**. Make sure the project type is set to **3D**. Set the **Location** to somewhere appropriate for you (remember, closer to root directories is better). Then, click **Create project**.

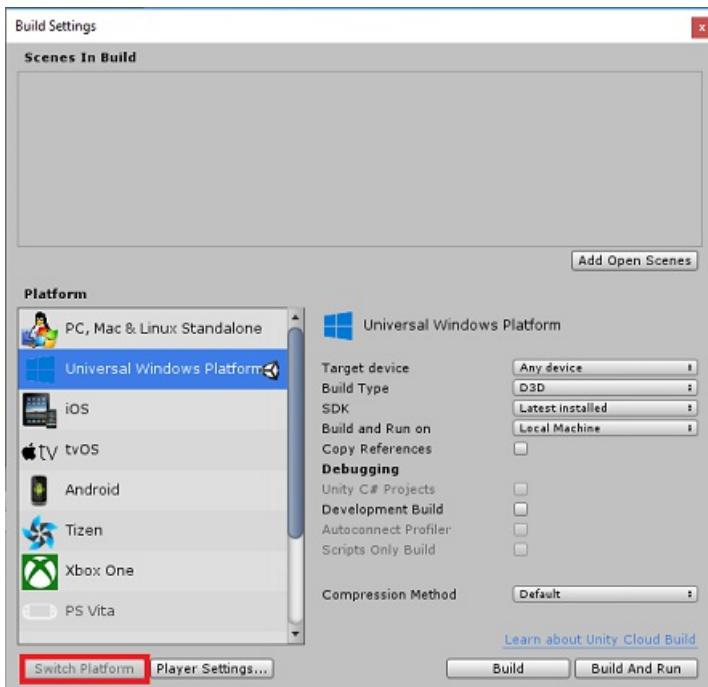
Projects Learn

[New](#) [Open](#) [My Account](#)

- With Unity open, it is worth checking the default **Script Editor** is set to **Visual Studio**. Go to Edit > Preferences and then from the new window, navigate to **External Tools**. Change **External Script Editor** to **Visual Studio 2017**. Close the **Preferences** window.



- Next, go to **File > Build Settings** and switch the platform to **Universal Windows Platform**, by clicking on the **Switch Platform** button.



5. Go to **File > Build Settings** and make sure that:

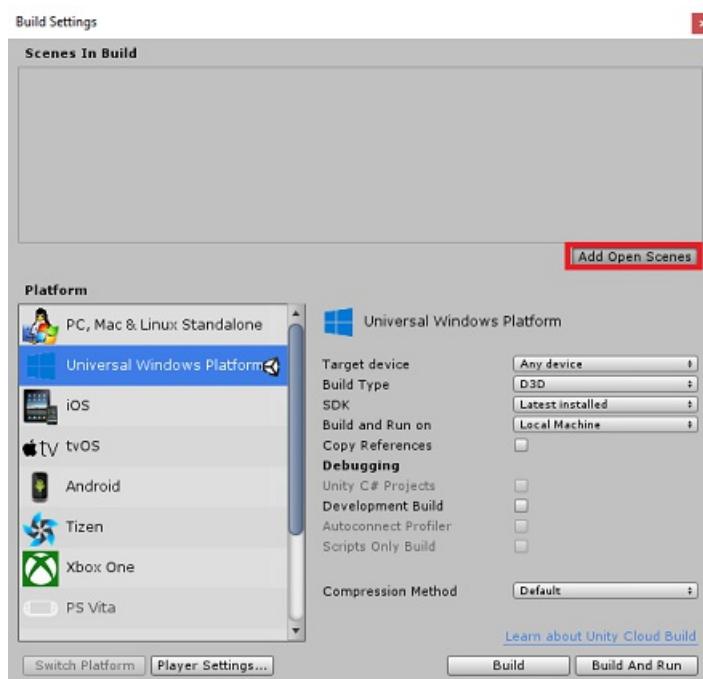
- Target Device** is set to **Any Device**

For the Microsoft HoloLens, set **Target Device** to *HoloLens*.

- Build Type** is set to **D3D**
- SDK** is set to **Latest installed**
- Visual Studio Version** is set to **Latest installed**
- Build and Run** is set to **Local Machine**

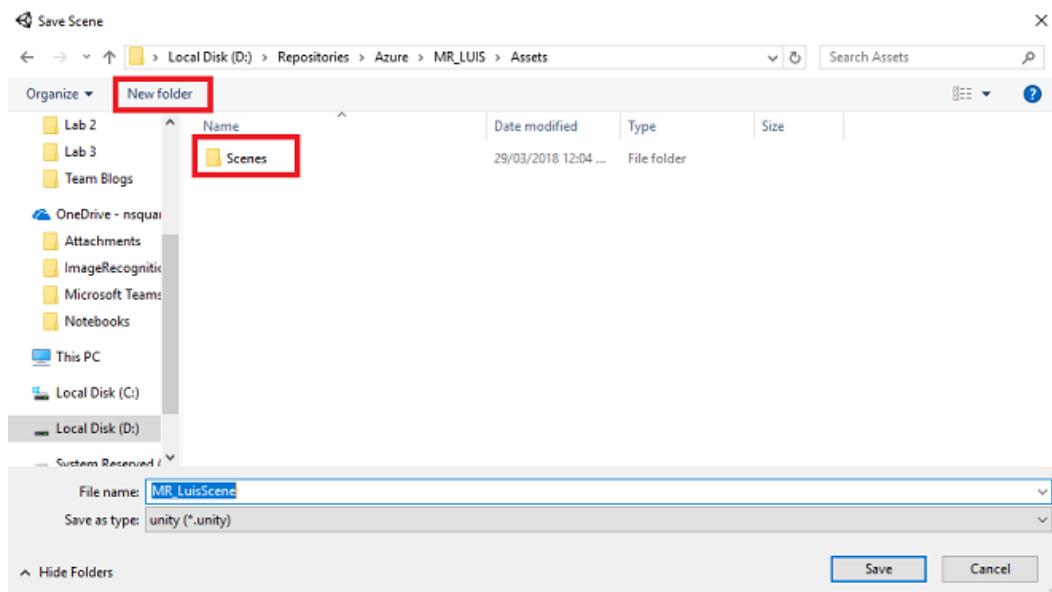
f. Save the scene and add it to the build.

- a. Do this by selecting **Add Open Scenes**. A save window will appear.

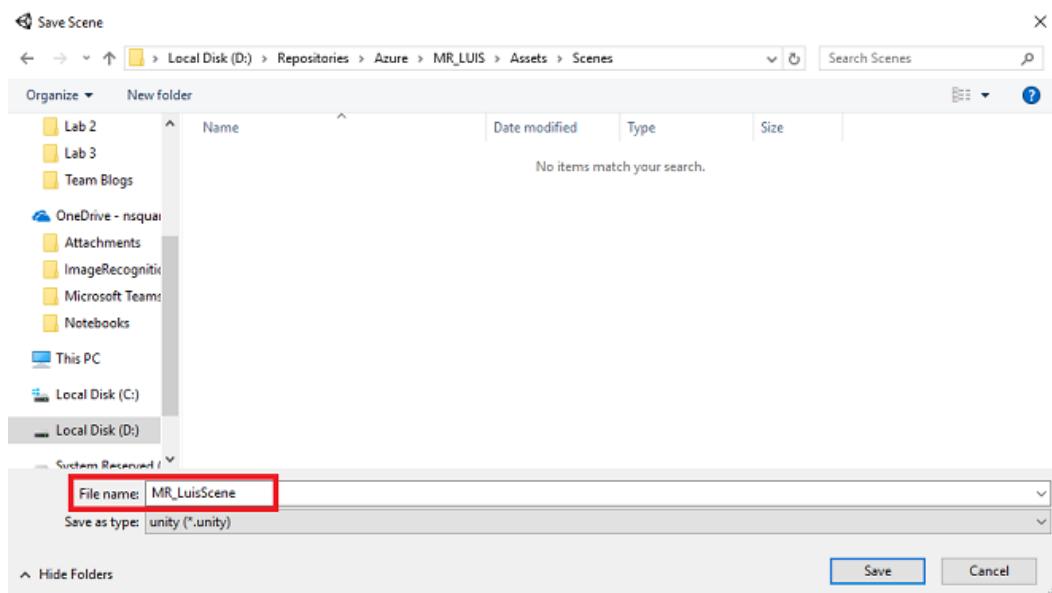


- b. Create a new folder for this, and any future, scene, then select the **New folder** button, to

create a new folder, name it **Scenes**.

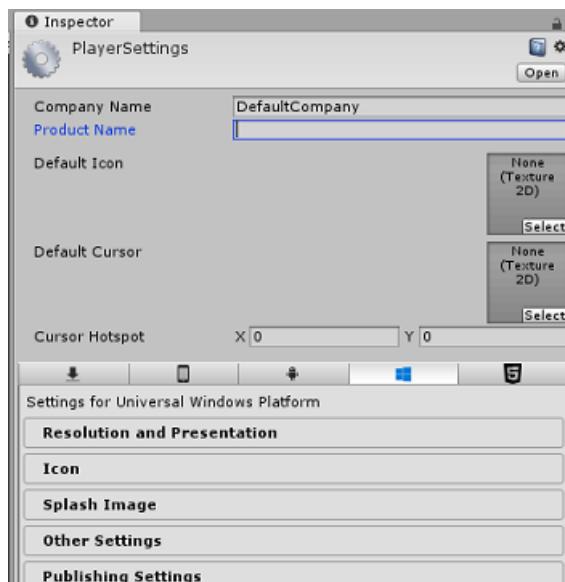


- c. Open your newly created **Scenes** folder, and then in the *File name:* text field, type **MR_LuisScene**, then press **Save**.



- g. The remaining settings, in *Build Settings*, should be left as default for now.

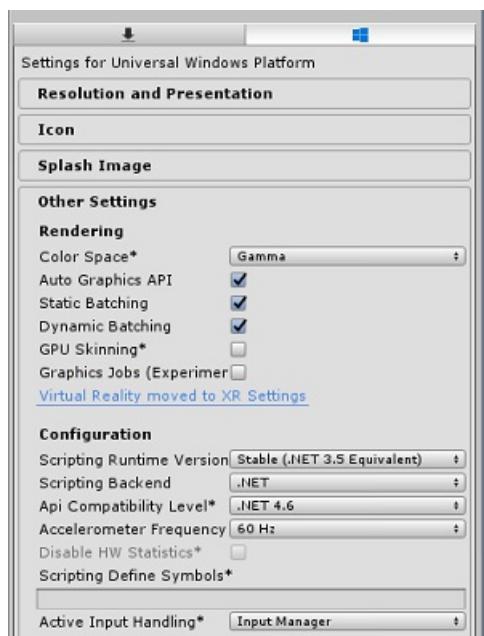
6. In the *Build Settings* window, click on the **Player Settings** button, this will open the related panel in the space where the *Inspector* is located.



7. In this panel, a few settings need to be verified:

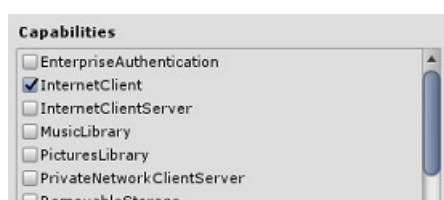
a. In the **Other Settings** tab:

- a. **Scripting Runtime Version** should be **Stable (.NET 3.5 Equivalent)**.
- b. **Scripting Backend** should be **.NET**
- c. **API Compatibility Level** should be **.NET 4.6**

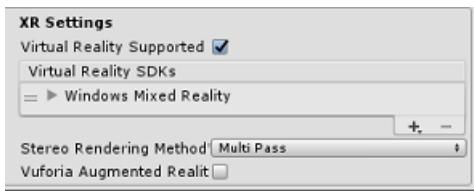


b. Within the **Publishing Settings** tab, under **Capabilities**, check:

- a. **InternetClient**
- b. **Microphone**



c. Further down the panel, in **XR Settings** (found below **Publish Settings**), tick **Virtual Reality Supported**, make sure the **Windows Mixed Reality SDK** is added.



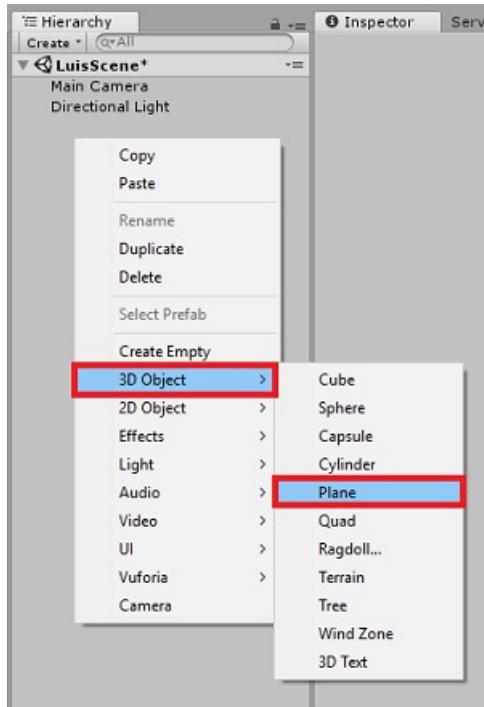
8. Back in *Build Settings Unity C# Projects* is no longer greyed out; tick the checkbox next to this.
9. Close the Build Settings window.
10. Save your Scene and Project (**FILE > SAVE SCENE / FILE > SAVE PROJECT**).

Chapter 4 – Create the scene

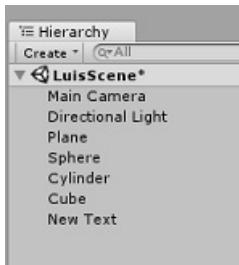
IMPORTANT

If you wish to skip the *Unity Set up* component of this course, and continue straight into code, feel free to download this [.unitypackage](#), import it into your project as a [Custom Package](#), and then continue from [Chapter 5](#).

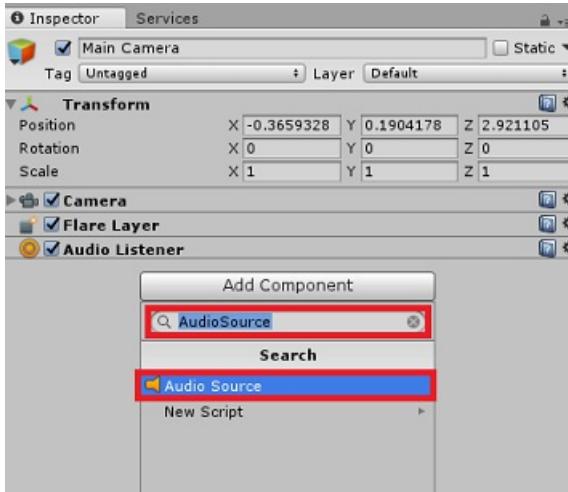
1. Right-click in an empty area of the *Hierarchy Panel*, under **3D Object**, add a **Plane**.



2. Be aware that when you right-click within the *Hierarchy* again to create more objects, if you still have the last object selected, the selected object will be the parent of your new object. Avoid this left-clicking in an empty space within the *Hierarchy*, and then right-clicking.
3. Repeat the above procedure to add the following objects:
 - a. *Sphere*
 - b. *Cylinder*
 - c. *Cube*
 - d. *3D Text*
4. The resulting scene *Hierarchy* should be like the one in the image below:



5. Left click on the **Main Camera** to select it, look at the *Inspector Panel* you will see the Camera object with all the its components.
6. Click on the **Add Component** button located at the very bottom of the *Inspector Panel*.



7. Search for the component called *Audio Source*, as shown above.
8. Also make sure that the *Transform* component of the Main Camera is set to (0,0,0), this can be done by pressing the **Gear** icon next to the Camera's *Transform* component and selecting **Reset**. The *Transform* component should then look like:
 - a. *Position* is set to **0, 0, 0**.
 - b. *Rotation* is set to **0, 0, 0**.

NOTE

For the Microsoft HoloLens, you will need to also change the following, which are part of the **Camera** component, which is on your **Main Camera**:

- **Clear Flags:** Solid Color.
- **Background** 'Black, Alpha 0' – Hex color: #00000000.

9. Left click on the **Plane** to select it. In the *Inspector Panel* set the *Transform* component with the following values:

TRANSFORM - POSITION		
X	Y	Z
0	-1	0

10. Left click on the **Sphere** to select it. In the *Inspector Panel* set the *Transform* component with the following values:

TRANSFORM - POSITION		
X	Y	Z
2	1	2

11. Left click on the **Cylinder** to select it. In the *Inspector Panel* set the *Transform* component with the following values:

TRANSFORM - POSITION		
X	Y	Z
-2	1	2

12. Left click on the **Cube** to select it. In the *Inspector Panel* set the *Transform* component with the following values:

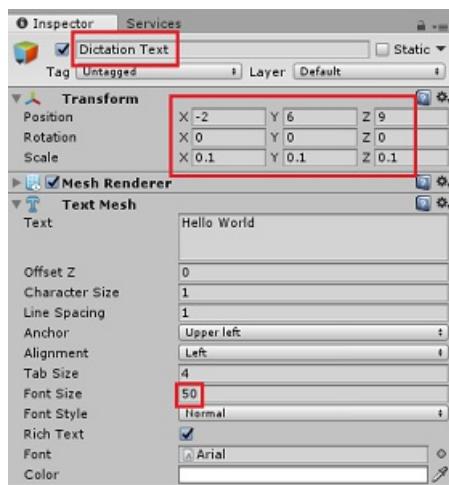
TRANSFORM - POSITION				TRANSFORM - ROTATION		
X	Y	Z		X	Y	Z
0	1	4		45	45	0

13. Left click on the **New Text** object to select it. In the *Inspector Panel* set the *Transform* component with the following values:

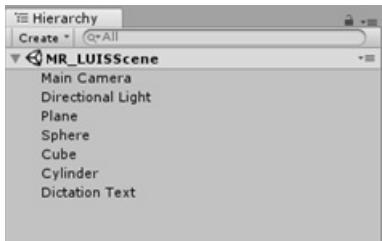
TRANSFORM - POSITION				TRANSFORM - SCALE		
X	Y	Z		X	Y	Z
-2	6	9		0.1	0.1	0.1

14. Change **Font Size** in the **Text Mesh** component to **50**.

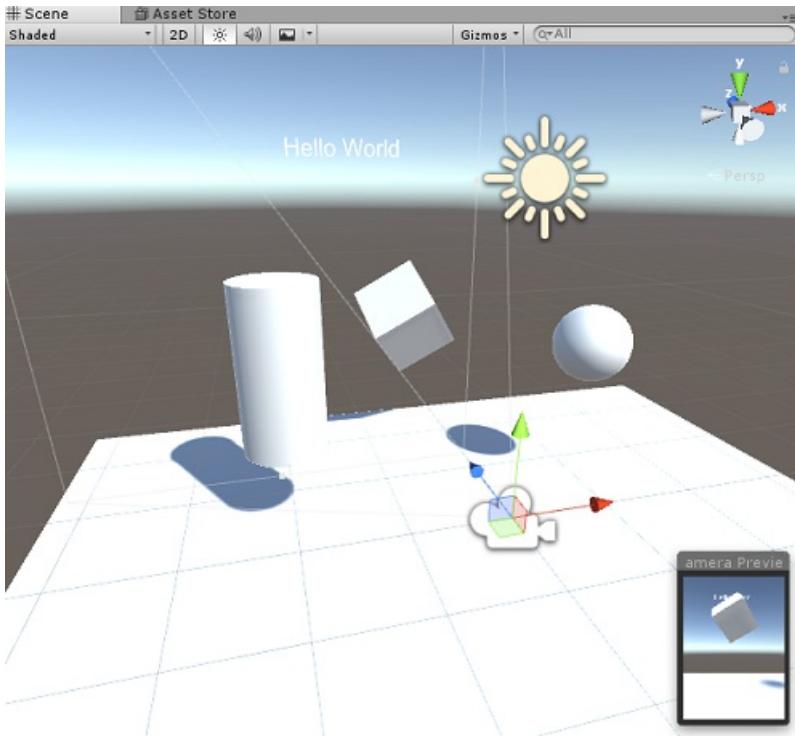
15. Change the *name* of the **Text Mesh** object to **Dictation Text**.



16. Your Hierarchy Panel structure should now look like this:



17. The final scene should look like the image below:



Chapter 5 – Create the MicrophoneManager class

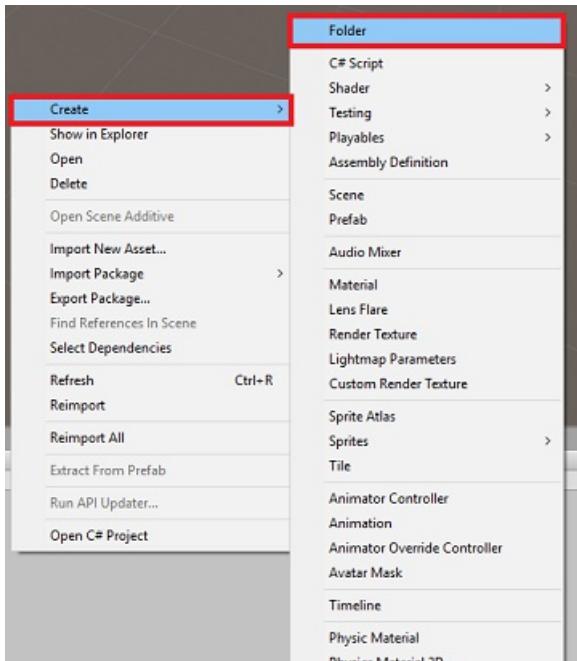
The first Script you are going to create is the *MicrophoneManager* class. Following this, you will create the *LuisManager*, the *Behaviours* class, and lastly the *Gaze* class (feel free to create all these now, though it will be covered as you reach each Chapter).

The *MicrophoneManager* class is responsible for:

- Detecting the recording device attached to the headset or machine (whichever is the default one).
- Capture the audio (voice) and use dictation to store it as a string.
- Once the voice has paused, submit the dictation to the *LuisManager* class.

To create this class:

1. Right-click in the *Project Panel*, **Create > Folder**. Call the folder **Scripts**.



2. With the **Scripts** folder created, double click it, to open. Then, within that folder, right-click, **Create > C# Script**. Name the script *MicrophoneManager*.
3. Double click on *MicrophoneManager* to open it with *Visual Studio*.
4. Add the following namespaces to the top of the file:

```
using UnityEngine;
using UnityEngine.Windows.Speech;
```

5. Then add the following variables inside the *MicrophoneManager* class:

```
public static MicrophoneManager instance; //help to access instance of this object
private DictationRecognizer dictationRecognizer; //Component converting speech to text
public TextMesh dictationText; //a UI object used to debug dictation result
```

6. Code for *Awake()* and *Start()* methods now needs to be added. These will be called when the class initializes:

```
private void Awake()
{
    // allows this class instance to behave like a singleton
    instance = this;
}

void Start()
{
    if (Microphone.devices.Length > 0)
    {
        StartCapturingAudio();
        Debug.Log("Mic Detected");
    }
}
```

7. Now you need the method that the App uses to start and stop the voice capture, and pass it to the *LuisManager* class, that you will build soon.

```

/// <summary>
/// Start microphone capture, by providing the microphone as a continual audio source (looping),
/// then initialise the DictationRecognizer, which will capture spoken words
/// </summary>
public void StartCapturingAudio()
{
    if (dictationRecognizer == null)
    {
        dictationRecognizer = new DictationRecognizer
        {
            InitialSilenceTimeoutSeconds = 60,
            AutoSilenceTimeoutSeconds = 5
        };

        dictationRecognizer.DictationResult += DictationRecognizer_DictationResult;
        dictationRecognizer.DictationError += DictationRecognizer_DictationError;
    }
    dictationRecognizer.Start();
    Debug.Log("Capturing Audio...");
}

/// <summary>
/// Stop microphone capture
/// </summary>
public void StopCapturingAudio()
{
    dictationRecognizer.Stop();
    Debug.Log("Stop Capturing Audio...");
}

```

- Add a *Dictation Handler* that will be invoked when the voice pauses. This method will pass the dictation text to the *LuisManager* class.

```

/// <summary>
/// This handler is called every time the Dictation detects a pause in the speech.
/// This method will stop listening for audio, send a request to the LUIS service
/// and then start listening again.
/// </summary>
private void DictationRecognizer_DictationResult(string dictationCaptured, ConfidenceLevel
confidence)
{
    StopCapturingAudio();
    StartCoroutine(LuisManager.instance.SubmitRequestToLuis(dictationCaptured,
StartCapturingAudio));
    Debug.Log("Dictation: " + dictationCaptured);
    dictationText.text = dictationCaptured;
}

private void DictationRecognizer_DictationError(string error, int hresult)
{
    Debug.Log("Dictation exception: " + error);
}

```

IMPORTANT

Delete the *Update()* method since this class will not use it.

- Be sure to save your changes in *Visual Studio* before returning to *Unity*.

NOTE

At this point you will notice an error appearing in the *Unity Editor Console Panel*. This is because the code references the *LuisManager* class which you will create in the next Chapter.

Chapter 6 – Create the LUISManager class

It is time for you to create the *LuisManager* class, which will make the call to the Azure LUIS service.

The purpose of this class is to receive the dictation text from the *MicrophoneManager* class and send it to the *Azure Language Understanding API* to be analyzed.

This class will deserialize the *JSON* response and call the appropriate methods of the *Behaviours* class to trigger an action.

To create this class:

1. Double click on the **Scripts** folder, to open it.
2. Right-click inside the **Scripts** folder, click **Create > C# Script**. Name the script *LuisManager*.
3. Double click on the script to open it with Visual Studio.
4. Add the following namespaces to the top of the file:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.IO;
using UnityEngine;
using UnityEngine.Networking;
```

5. You will begin by creating three classes **inside** the *LuisManager* class (within the same script file, above the *Start()* method) that will represent the deserialized JSON response from Azure.

```

[Serializable] //this class represents the LUIS response
public class AnalysedQuery
{
    public TopScoringIntentData topScoringIntent;
    public EntityData[] entities;
    public string query;
}

// This class contains the Intent LUIS determines
// to be the most likely
[Serializable]
public class TopScoringIntentData
{
    public string intent;
    public float score;
}

// This class contains data for an Entity
[Serializable]
public class EntityData
{
    public string entity;
    public string type;
    public int startIndex;
    public int endIndex;
    public float score;
}

```

6. Next, add the following variables inside the *LuisManager* class:

```

public static LuisManager instance;

//Substitute the value of luis Endpoint with your own End Point
string luisEndpoint = "https://westus.api.cognitive... add your endpoint from the Luis Portal";

```

7. Make sure to place your LUIS endpoint in now (which you will have from your LUIS portal).
8. Code for the *Awake()* method now needs to be added. This method will be called when the class initializes:

```

private void Awake()
{
    // allows this class instance to behave like a singleton
    instance = this;
}

```

9. Now you need the methods this application uses to send the dictation received from the *MicrophoneManager* class to *Luis*, and then receive and deserialize the response.
10. Once the value of the Intent, and associated Entities, have been determined, they are passed to the instance of the *Behaviours* class to trigger the intended action.

```

/// <summary>
/// Call LUIS to submit a dictation result.
/// The done Action is called at the completion of the method.
/// </summary>
public IEnumerator SubmitRequestToLuis(string dictationResult, Action done)
{
    string queryString = string.Concat(Uri.EscapeDataString(dictationResult));

    using (UnityWebRequest unityWebRequest = UnityWebRequest.Get(luisEndpoint + queryString))
    {
        yield return unityWebRequest.SendWebRequest();

        if (unityWebRequest.isNetworkError || unityWebRequest.isHttpError)
        {
            Debug.Log(unityWebRequest.error);
        }
        else
        {
            try
            {
                AnalysedQuery analysedQuery = JsonUtility.FromJson<AnalysedQuery>
                (unityWebRequest.downloadHandler.text);

                //analyse the elements of the response
                AnalyseResponseElements(analysedQuery);
            }
            catch (Exception exception)
            {
                Debug.Log("Luis Request Exception Message: " + exception.Message);
            }
        }
    }

    done();
    yield return null;
}
}

```

11. Create a new method called *AnalyseResponseElements()* that will read the resulting *AnalysedQuery* and determine the Entities. Once those Entities are determined, they will be passed to the instance of the *Behaviours* class to use in the actions.

```

private void AnalyseResponseElements(AnalysedQuery aQuery)
{
    string topIntent = aQuery.topScoringIntent.intent;

    // Create a dictionary of entities associated with their type
    Dictionary<string, string> entityDic = new Dictionary<string, string>();

    foreach (EntityData ed in aQuery.entities)
    {
        entityDic.Add(ed.type, ed.entity);
    }

    // Depending on the topmost recognised intent, read the entities name
    switch (aQuery.topScoringIntent.intent)
    {
        case "ChangeObjectColor":
            string targetForColor = null;
            string color = null;

            foreach (var pair in entityDic)
            {
                if (pair.Key == "target")
                {
                    targetForColor = pair.Value;
                }
                else if (pair.Key == "color")
                {
                    color = pair.Value;
                }
            }

            Behaviours.instance.ChangeTargetColor(targetForColor, color);
            break;

        case "ChangeObjectSize":
            string targetForSize = null;
            foreach (var pair in entityDic)
            {
                if (pair.Key == "target")
                {
                    targetForSize = pair.Value;
                }
            }

            if (entityDic.ContainsKey("upsize") == true)
            {
                Behaviours.instance.UpSizeTarget(targetForSize);
            }
            else if (entityDic.ContainsKey("downsize") == true)
            {
                Behaviours.instance.DownSizeTarget(targetForSize);
            }
            break;
    }
}

```

IMPORTANT

Delete the *Start()* and *Update()* methods since this class will not use them.

12. Be sure to save your changes in *Visual Studio* before returning to *Unity*.

NOTE

At this point you will notice several errors appearing in the *Unity Editor Console Panel*. This is because the code references the *Behaviours* class which you will create in the next Chapter.

Chapter 7 – Create the Behaviours class

The *Behaviours* class will trigger the actions using the Entities provided by the *LuisManager* class.

To create this class:

1. Double click on the **Scripts** folder, to open it.
2. Right-click inside the **Scripts** folder, click **Create > C# Script**. Name the script *Behaviours*.
3. Double click on the script to open it with *Visual Studio*.
4. Then add the following variables inside the *Behaviours* class:

```
public static Behaviours instance;

// the following variables are references to possible targets
public GameObject sphere;
public GameObject cylinder;
public GameObject cube;
internal GameObject gazedTarget;
```

5. Add the *Awake()* method code. This method will be called when the class initializes:

```
void Awake()
{
    // allows this class instance to behave like a singleton
    instance = this;
}
```

6. The following methods are called by the *LuisManager* class (which you have created previously) to determine which object is the target of the query and then trigger the appropriate action.

```

/// <summary>
/// Changes the color of the target GameObject by providing the name of the object
/// and the name of the color
/// </summary>
public void ChangeTargetColor(string targetName, string colorName)
{
    GameObject foundTarget = FindTarget(targetName);
    if (foundTarget != null)
    {
        Debug.Log("Changing color " + colorName + " to target: " + foundTarget.name);

        switch (colorName)
        {
            case "blue":
                foundTarget.GetComponent<Renderer>().material.color = Color.blue;
                break;

            case "red":
                foundTarget.GetComponent<Renderer>().material.color = Color.red;
                break;

            case "yellow":
                foundTarget.GetComponent<Renderer>().material.color = Color.yellow;
                break;

            case "green":
                foundTarget.GetComponent<Renderer>().material.color = Color.green;
                break;

            case "white":
                foundTarget.GetComponent<Renderer>().material.color = Color.white;
                break;

            case "black":
                foundTarget.GetComponent<Renderer>().material.color = Color.black;
                break;
        }
    }
}

/// <summary>
/// Reduces the size of the target GameObject by providing its name
/// </summary>
public void DownSizeTarget(string targetName)
{
    GameObject foundTarget = FindTarget(targetName);
    foundTarget.transform.localScale -= new Vector3(0.5F, 0.5F, 0.5F);
}

/// <summary>
/// Increases the size of the target GameObject by providing its name
/// </summary>
public void UpSizeTarget(string targetName)
{
    GameObject foundTarget = FindTarget(targetName);
    foundTarget.transform.localScale += new Vector3(0.5F, 0.5F, 0.5F);
}

```

7. Add the *FindTarget()* method to determine which of the *GameObjects* is the target of the current Intent. This method defaults the target to the *GameObject* being “gazed” if no explicit target is defined in the Entities.

```

/// <summary>
/// Determines which obejct reference is the target GameObject by providing its name
/// </summary>
private GameObject FindTarget(string name)
{
    GameObject targetAsGO = null;

    switch (name)
    {
        case "sphere":
            targetAsGO = sphere;
            break;

        case "cylinder":
            targetAsGO = cylinder;
            break;

        case "cube":
            targetAsGO = cube;
            break;

        case "this": // as an example of target words that the user may use when looking at an
object
        case "it": // as this is the default, these are not actually needed in this example
        case "that":
        default: // if the target name is none of those above, check if the user is looking at
something
            if (gazedTarget != null)
            {
                targetAsGO = gazedTarget;
            }
            break;
    }
    return targetAsGO;
}

```

IMPORTANT

Delete the *Start()* and *Update()* methods since this class will not use them.

8. Be sure to save your changes in *Visual Studio* before returning to *Unity*.

Chapter 8 – Create the Gaze Class

The last class that you will need to complete this app is the *Gaze* class. This class updates the reference to the *GameObject* currently in the user's visual focus.

To create this Class:

1. Double click on the **Scripts** folder, to open it.
2. Right-click inside the **Scripts** folder, click **Create > C# Script**. Name the script *Gaze*.
3. Double click on the script to open it with *Visual Studio*.
4. Insert the following code for this class:

```

using UnityEngine;

public class Gaze : MonoBehaviour
{
    internal GameObject gazedObject;
    public float gazeMaxDistance = 300;

    void Update()
    {
        // Uses a raycast from the Main Camera to determine which object is gazed upon.
        Vector3 fwd = gameObject.transform.TransformDirection(Vector3.forward);
        Ray ray = new Ray(Camera.main.transform.position, fwd);
        RaycastHit hit;
        Debug.DrawRay(Camera.main.transform.position, fwd);

        if (Physics.Raycast(ray, out hit, gazeMaxDistance) && hit.collider != null)
        {
            if (gazedObject == null)
            {
                gazedObject = hit.transform.gameObject;

                // Set the gazedTarget in the Behaviours class
                Behaviours.instance.gazedTarget = gazedObject;
            }
        }
        else
        {
            ResetGaze();
        }
    }

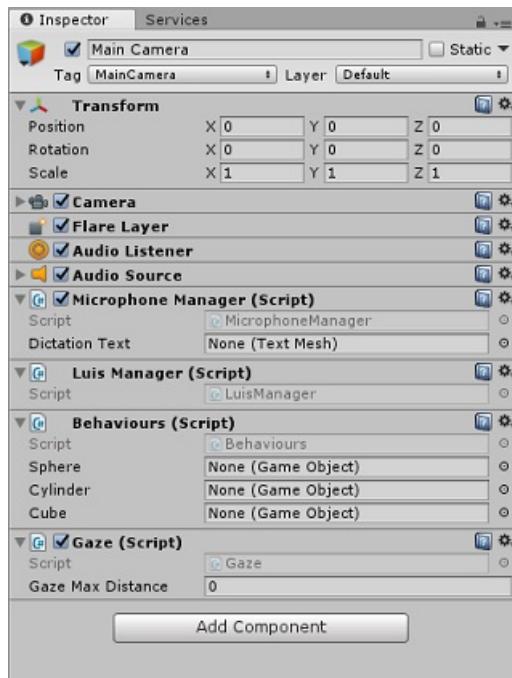
    // Turn the gaze off, reset the gazeObject in the Behaviours class.
    public void ResetGaze()
    {
        if (gazedObject != null)
        {
            Behaviours.instance.gazedTarget = null;
            gazedObject = null;
        }
    }
}

```

5. Be sure to save your changes in *Visual Studio* before returning to *Unity*.

Chapter 9 – Completing the scene setup

1. To complete the setup of the scene, drag each script that you have created from the *Scripts Folder* to the **Main Camera** object in the *Hierarchy Panel*.
2. Select the **Main Camera** and look at the *Inspector Panel*, you should be able to see each script that you have attached, and you will notice that there are parameters on each script that are yet to be set.



3. To set these parameters correctly, follow these instructions:

a. *MicrophoneManager*:

- From the *Hierarchy Panel*, drag the **Dictation Text** object into the **Dictation Text** parameter value box.

b. *Behaviours*, from the *Hierarchy Panel*:

- Drag the **Sphere** object into the *Sphere* reference target box.
- Drag the **Cylinder** into the *Cylinder* reference target box.
- Drag the **Cube** into the *Cube* reference target box.

c. *Gaze*:

- Set the *Gaze Max Distance* to **300** (if it is not already).

4. The result should look like the image below:



Chapter 10 – Test in the Unity Editor

Test that the Scene setup is properly implemented.

Ensure that:

- All the scripts are attached to the **Main Camera** object.
 - All the fields in the *Main Camera Inspector Panel* are assigned properly.
1. Press the **Play** button in the *Unity Editor*. The App should be running within the attached immersive headset.
 2. Try a few utterances, such as:

```
make the cylinder red  
change the cube to yellow  
I want the sphere blue  
make this to green  
change it to white
```

NOTE

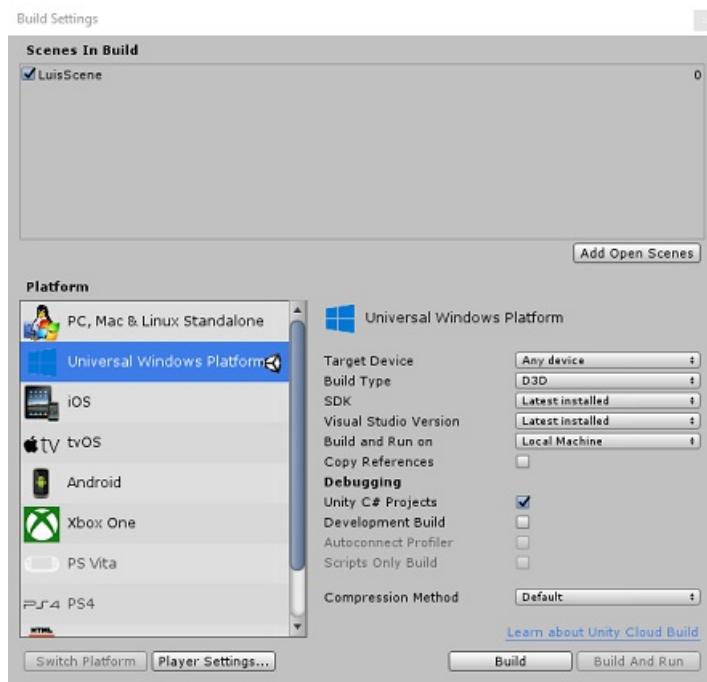
If you see an error in the Unity console about the default audio device changing, the scene may not function as expected. This is due to the way the mixed reality portal deals with built-in microphones for headsets that have them. If you see this error, simply stop the scene and start it again and things should work as expected.

Chapter 11 – Build and sideload the UWP Solution

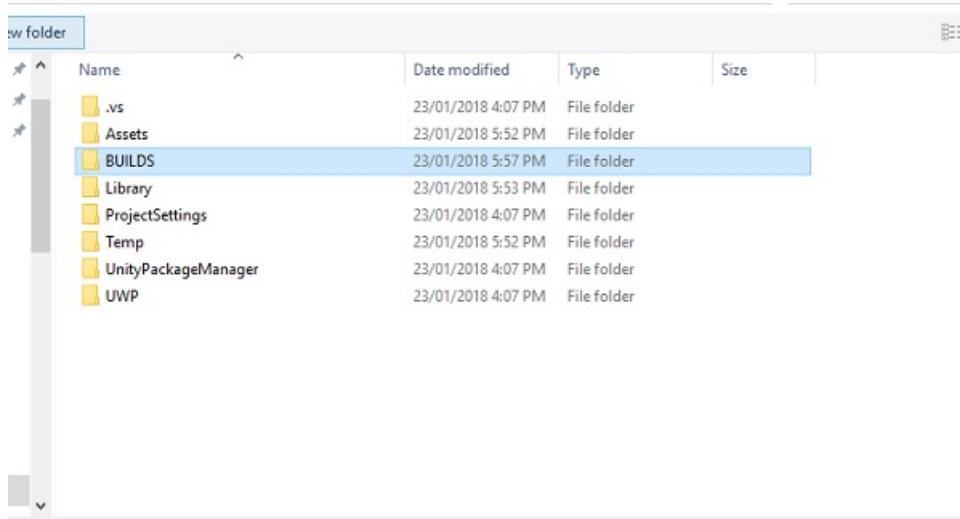
Once you have ensured that the application is working in the Unity Editor, you are ready to Build and Deploy.

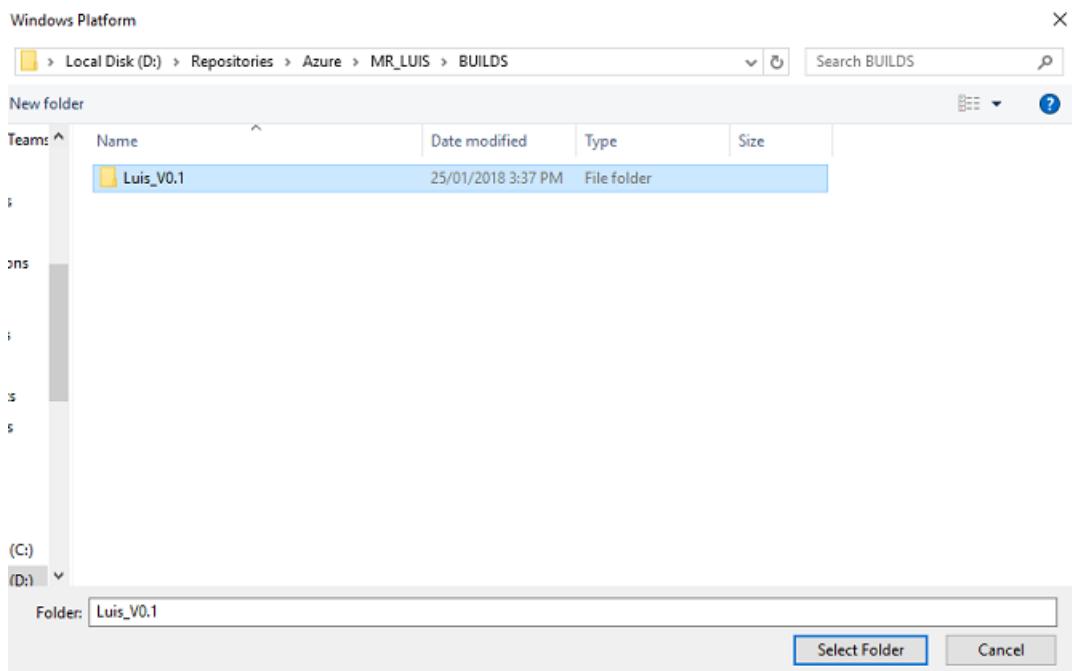
To Build:

1. Save the current scene by clicking on **File > Save**.
2. Go to **File > Build Settings**.
3. Tick the box called **Unity C# Projects** (useful for seeing and debugging your code once the UWP project is created).
4. Click on **Add Open Scenes**, then click **Build**.



5. You will be prompted to select the folder where you want to build the Solution.
6. Create a *BUILDS* folder and within that folder create another folder with an appropriate name of your choice.
7. Click **Select Folder** to begin the build at that location.





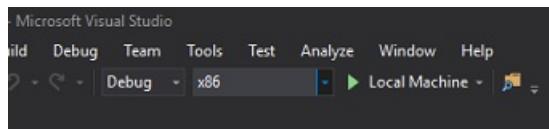
- Once Unity has finished building (it might take some time), it should open a **File Explorer** window at the location of your build.

To Deploy on Local Machine:

- In *Visual Studio*, open the solution file that has been created in the [previous Chapter](#).
- In the **Solution Platform**, select **x86, Local Machine**.
- In the **Solution Configuration** select **Debug**.

For the Microsoft HoloLens, you may find it easier to set this to *Remote Machine*, so that you are not tethered to your computer. Though, you will need to also do the following:

- Know the **IP Address** of your HoloLens, which can be found within the *Settings > Network & Internet > Wi-Fi > Advanced Options*; the IPv4 is the address you should use.
- Ensure **Developer Mode** is **On**; found in *Settings > Update & Security > For developers*.



- Go to the **Build menu** and click on **Deploy Solution** to sideload the application to your machine.
- Your App should now appear in the list of installed apps, ready to be launched!
- Once launched, the App will prompt you to authorize access to the *Microphone*. Use the *Motion Controllers*, or *Voice Input*, or the *Keyboard* to press the **YES** button.

Chapter 12 – Improving your LUIS service

IMPORTANT

This chapter is incredibly important, and may need to be iterated upon several times, as it will help improve the accuracy of your LUIS service: ensure you complete this.

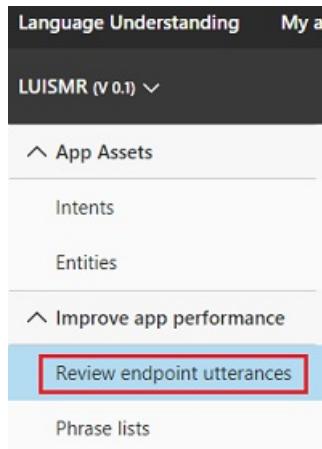
To improve the level of understanding provided by LUIS you need to capture new utterances and use them to re-

train your LUIS App.

For example, you might have trained LUIS to understand "Increase" and "Upsize", but wouldn't you want your app to also understand words like "Enlarge"?

Once you have used your application a few times, everything you have said will be collected by LUIS and available in the LUIS PORTAL.

1. Go to your portal application following this [LINK](#), and Log In.
2. Once you are logged in with your MS Credentials, click on your *App name*.
3. Click the **Review endpoint utterances** button on the left of the page.



4. You will be shown a list of the Utterances that have been sent to LUIS by your mixed reality Application.

A screenshot of the 'Review endpoint utterances' page in the LUIS Portal. The top navigation bar shows 'DASHBOARD', 'BUILD' (selected), 'PUBLISH', 'SETTINGS', 'Train', and 'Test'. The sidebar on the left shows 'App Assets' (Intents, Entities), 'Improve app performance' (Review endpoint utterances, highlighted with a blue box), and 'Phrase lists'. The main area has a title 'Review endpoint utterances' with a help icon. It features a search bar 'Filter list by intent or entity' with 'ChangeObjectColor' typed in, and a 'Labels view (Ctrl+E)' dropdown. A button 'Add all selected utterances to their aligned intents' is present. The main table lists utterances with checkboxes, aligned intents, and actions to add or delete them. Utterances include: 'change target colour to color', 'how old is and green white blue color yellow', 'yellow', 'make the star color', 'change target colour color', 'change the target to color', 'so i don't need to color i need to make that change anything seems to be', 'change keyboard size up', 'change the target color to color', and 'change target color to color'. Each row has a 'Delete' button.

You will notice some highlighted *Entities*.

By hovering over each highlighted word, you can review each Utterance and determine which Entity has been recognized correctly, which Entities are wrong and which Entities are missed.

In the example above, it was found that the word "spear" had been highlighted as a target, so it necessary to correct the mistake, which is done by hovering over the word with the mouse and clicking **Remove Label**.

- change target colour color
- change the target to color
- so i don't need to color i need to make that change anything seems to be

target target
is a Remove label
Wrap in composite entity
Manage entity
e sf. color
target downsize
upsize
the target to color

5. If you find Utterances that are completely wrong, you can delete them using the **Delete** button on the right side of the screen.



6. Or if you feel that LUIS has interpreted the Utterance correctly, you can validate its understanding by using the **Add To Aligned Intent** button.

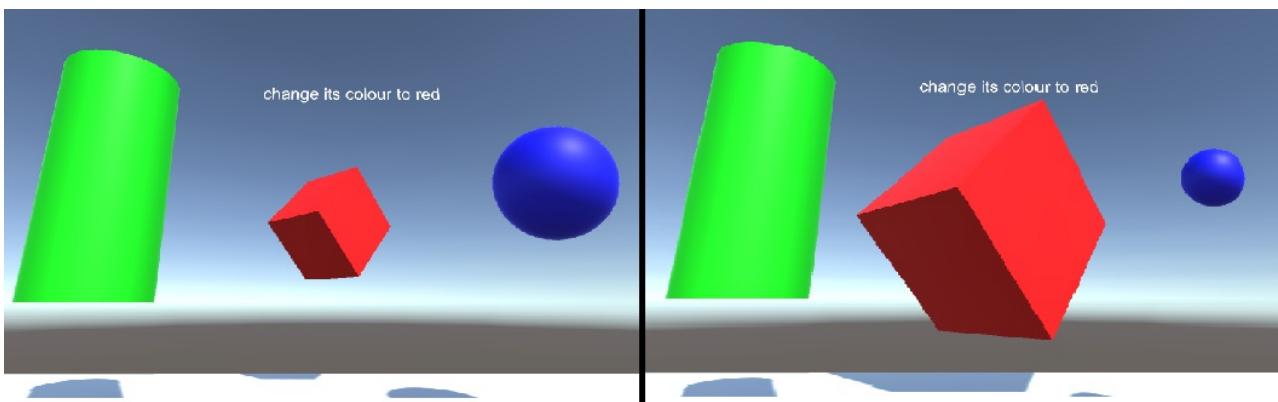


- Once you have sorted all the displayed Utterances, try and reload the page to see if more are available.
 - It is very important to repeat this process as many times as possible to improve your application understanding.

Have fun!

Your finished LUIS Integrated application

Congratulations, you built a mixed reality app that leverages the Azure Language Understanding Intelligence Service, to understand what a user says, and act on that information.



Bonus exercises

Exercise 1

While using this application you might notice that if you gaze at the Floor object and ask to change its color, it will do so. Can you work out how to stop your application from changing the Floor color?

Exercise 2

Try extending the LUIS and App capabilities, adding additional functionality for objects in scene; as an example, create new objects at the Gaze hit point, depending on what the user says, and then be able to use those objects alongside current scene objects, with the existing commands.

MR and Azure 304: Face recognition

11/6/2018 • 24 minutes to read • [Edit Online](#)



In this course you will learn how to add face recognition capabilities to a mixed reality application, using Azure Cognitive Services, with the Microsoft Face API.

Azure Face API is a Microsoft service, which provides developers with the most advanced face algorithms, all in the cloud. The *Face API* has two main functions: face detection with attributes, and face recognition. This allows developers to simply set a set of groups for faces, and then, send query images to the service later, to determine to whom a face belongs. For more information, visit the [Azure Face Recognition page](#).

Having completed this course, you will have a mixed reality HoloLens application, which will be able to do the following:

1. Use a **Tap Gesture** to initiate the capture of an image using the on-board HoloLens camera.
2. Send the captured image to the *Azure Face API* service.
3. Receive the results of the *Face API* algorithm.
4. Use a simple User Interface, to display the name of matched people.

This will teach you how to get the results from the Face API Service into your Unity-based mixed reality application.

In your application, it is up to you as to how you will integrate the results with your design. This course is designed to teach you how to integrate an Azure Service with your Unity Project. It is your job to use the knowledge you gain from this course to enhance your mixed reality application.

Device support

COURSE	HOLOLENS	IMMERSIVE HEADSETS
--------	----------	--------------------

NOTE

While this course primarily focuses on HoloLens, you can also apply what you learn in this course to Windows Mixed Reality immersive (VR) headsets. Because immersive (VR) headsets do not have accessible cameras, you will need an external camera connected to your PC. As you follow along with the course, you will see notes on any changes you might need to employ to support immersive (VR) headsets.

Prerequisites

NOTE

This tutorial is designed for developers who have basic experience with Unity and C#. Please also be aware that the prerequisites and written instructions within this document represent what has been tested and verified at the time of writing (May 2018). You are free to use the latest software, as listed within the [install the tools](#) article, though it should not be assumed that the information in this course will perfectly match what you'll find in newer software than what's listed below.

We recommend the following hardware and software for this course:

- A development PC, [compatible with Windows Mixed Reality](#) for immersive (VR) headset development
- [Windows 10 Fall Creators Update \(or later\) with Developer mode enabled](#)
- [The latest Windows 10 SDK](#)
- [Unity 2017.4](#)
- [Visual Studio 2017](#)
- A [Windows Mixed Reality immersive \(VR\) headset](#) or [Microsoft HoloLens](#) with Developer mode enabled
- A camera connected to your PC (for immersive headset development)
- Internet access for Azure setup and Face API retrieval

Before you start

1. To avoid encountering issues building this project, it is strongly suggested that you create the project mentioned in this tutorial in a root or near-root folder (long folder paths can cause issues at build-time).
2. Set up and test your HoloLens. If you need support setting up your HoloLens, [make sure to visit the HoloLens setup article](#).
3. It is a good idea to perform Calibration and Sensor Tuning when beginning developing a new HoloLens App (sometimes it can help to perform those tasks for each user).

For help on Calibration, please follow this [link to the HoloLens Calibration article](#).

For help on Sensor Tuning, please follow this [link to the HoloLens Sensor Tuning article](#).

Chapter 1 - The Azure Portal

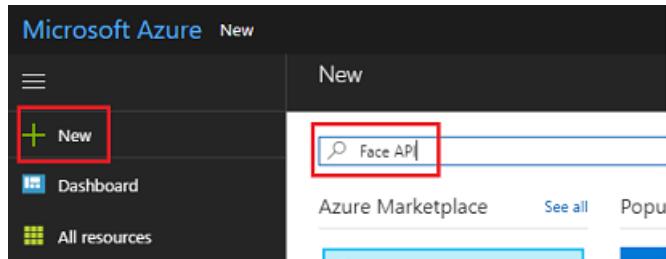
To use the *Face API* service in Azure, you will need to configure an instance of the service to be made available to your application.

1. First, log in to the [Azure Portal](#).

NOTE

If you do not already have an Azure account, you will need to create one. If you are following this tutorial in a classroom or lab situation, ask your instructor or one of the proctors for help setting up your new account.

- Once you are logged in, click on **New** in the top left corner, and search for *Face API*, press **Enter**.



NOTE

The word **New** may have been replaced with **Create a resource**, in newer portals.

- The new page will provide a description of the *Face API* service. At the bottom left of this prompt, select the **Create** button, to create an association with this service.

A screenshot of the Microsoft Azure Marketplace page for the 'Face' service. It shows a brief description of the service, social sharing icons, publisher information (Microsoft), and useful links. At the bottom left, there is a prominent blue 'Create' button.

- Once you have clicked on **Create**:

- Insert your desired name for this service instance.

- b. Select a subscription.
- c. Select the pricing tier appropriate for you, if this is the first time creating a *Face API Service*, a free tier (named F0) should be available to you.
- d. Choose a **Resource Group** or create a new one. A resource group provides a way to monitor, control access, provision and manage billing for a collection of Azure assets. It is recommended to keep all the Azure services associated with a single project (e.g. such as these labs) under a common resource group).

If you wish to read more about Azure Resource Groups, please [visit the resource group article](#).

- e. The UWP app, **Person Maker**, which you use later, requires the use of 'West US' for location.
- f. You will also need to confirm that you have understood the Terms and Conditions applied to this Service.
- g. Select *Create*.*

The screenshot shows the 'Create Face' dialog box. The fields are as follows:

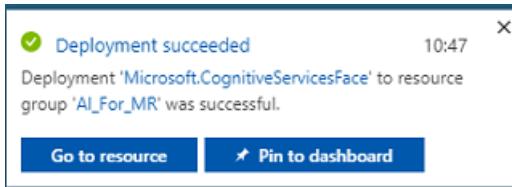
- Name:** MyNewFaceAPI
- Subscription:** Main Subscription
- Location:** West US
- Pricing tier:** S0 (10 Calls per second)
- Resource group:** AI_For_MR

At the bottom, there is a 'Create' button highlighted in blue, and an 'Automation options' link.

- 5. Once you have clicked on *Create*,* you will have to wait for the service to be created, this might take a minute.
- 6. A notification will appear in the portal once the Service instance is created.



- Click on the notifications to explore your new Service instance.



- When you are ready, click **Go to resource** button in the notification to explore your new Service instance.

- Within this tutorial, your application will need to make calls to your service, which is done through using your service's subscription 'key'. From the *Quick start* page, of your *Face API* service, the first point is number 1, to *Grab your keys*.
- On the *Service* page select either the blue **Keys** hyperlink (if on the *Quick start* page), or the **Keys** link in the services navigation menu (to the left, denoted by the 'key' icon), to reveal your keys.

NOTE

Take note of either one of the keys and safeguard it, as you will need it later.

Chapter 2 - Using the 'Person Maker' UWP application

Make sure to download the prebuilt UWP Application called [Person Maker](#). This app is not the end product for this course, just a tool to help you create your Azure entries, which the later project will rely upon.

Person Maker allows you to create Azure entries, which are associated with people, and groups of people. The application will place all the needed information in a format which can then later be used by the FaceAPI, in order to recognize the faces of people whom you have added.

[IMPORTANT] **Person Maker** uses some basic throttling, to help ensure that you do not exceed the number of service calls per minute for the **free subscription tier**. The green text at the top will change to red and update as 'ACTIVE' when throttling is happening; if this is the case, simply wait for the application (it will wait until it can next continue accessing the face service, updating as 'IN-ACTIVE' when you can use it again).

This application uses the *Microsoft.ProjectOxford.Face* libraries, which will allow you to make full use of the Face API. This library is available for free as a NuGet Package. For more information about this, and similar, APIs [make sure to visit the API reference article](#).

NOTE

These are just the steps required, instructions for how to do these things is further down the document. The **Person Maker** app will allow you to:

- Create a *Person Group*, which is a group composed of several people which you want to associate with it. With your Azure account you can host multiple Person Groups.
- Create a *Person*, which is a member of a Person Group. Each person has a number of *Face* images associated with it.
- Assign *face images* to a *Person*, to allow your Azure Face API Service to recognize a *Person* by the corresponding *face*.
- *Train* your Azure Face API Service.

Be aware, so to train this app to recognize people, you will need ten (10) close-up photos of each person which you would like to add to your Person Group. The Windows 10 Cam App can help you to take these. You must ensure that each photo is clear (avoid blurring, obscuring, or being too far, from the subject), have the photo in jpg or png file format, with the image file size being no larger **4 MB**, and no less than **1 KB**.

NOTE

If you are following this tutorial, do not use your own face for training, as when you put the HoloLens on, you cannot look at yourself. Use the face of a colleague or fellow student.

Running **Person Maker**:

1. Open the **PersonMaker** folder and double click on the *PersonMaker solution* to open it with *Visual Studio*.
2. Once the *PersonMaker solution* is open, make sure that:
 - a. The *Solution Configuration* is set to **Debug**.
 - b. The *Solution Platform* is set to **x86**
 - c. The *Target Platform* is **Local Machine**.
 - d. You also may need to *Restore NuGet Packages* (right-click the *Solution* and select **Restore NuGet Packages**).
3. Click *Local Machine* and the application will start. Be aware, on smaller screens, all content may not be visible, though you can scroll further down to view it.

PERSON MAKER

Throttling Status: IN-ACTIVE

To create a new Person with Azure Face Recognition API:

1. Insert your Face Recognition Key here (it will be deleted when you close this app):

2. To create or fetch a Person Group, insert the Person Group Id here:

Create a Person Group

Fetch a Known Group

- Person Group status -

3. Create a person, by inserting the Name here and pressing the Create Person button.

Create a Person

Delete a Person

- Person status -

4. Using Windows 10 Cam App, capture some photos of a single person (min 10)

5. Click the button below to create and open the person folder. Drop the photos in it.

Create and Open Folder

6. If there are at least 6 valid photos, you can click below to submit to azure.

Submit To Azure

Submission Status:

7. Use this button to train the Face API.

Train

Submission Status:

4. Insert your **Azure Authentication Key**, which you should have, from your *Face API* service within Azure.

5. Insert:

- The *ID* you want to assign to the *Person Group*. The ID must be lowercase, with no spaces. Make note of this ID, as it will be required later in your Unity project.
- The *Name* you want to assign to the *Person Group* (can have spaces).

6. Press **Create Person Group** button. A confirmation message should appear underneath the button.

NOTE

If you have an 'Access Denied' error, check the location you set for your Azure service. As stated above, this app is designed for 'West US'.

IMPORTANT

You will notice that you can also click the **Fetch a Known Group** button: this is for if you have already created a person group, and wish to use that, rather than create a new one. Be aware, if you click *Create a Person Group* with a known group, this will also fetch a group.

7. Insert the *Name* of the *Person* you want to create.

- Click the **Create Person** button.
- A confirmation message should appear underneath the button.
- If you wish to delete a person you have previously created, you can write the name into the textbox

and press **Delete Person**

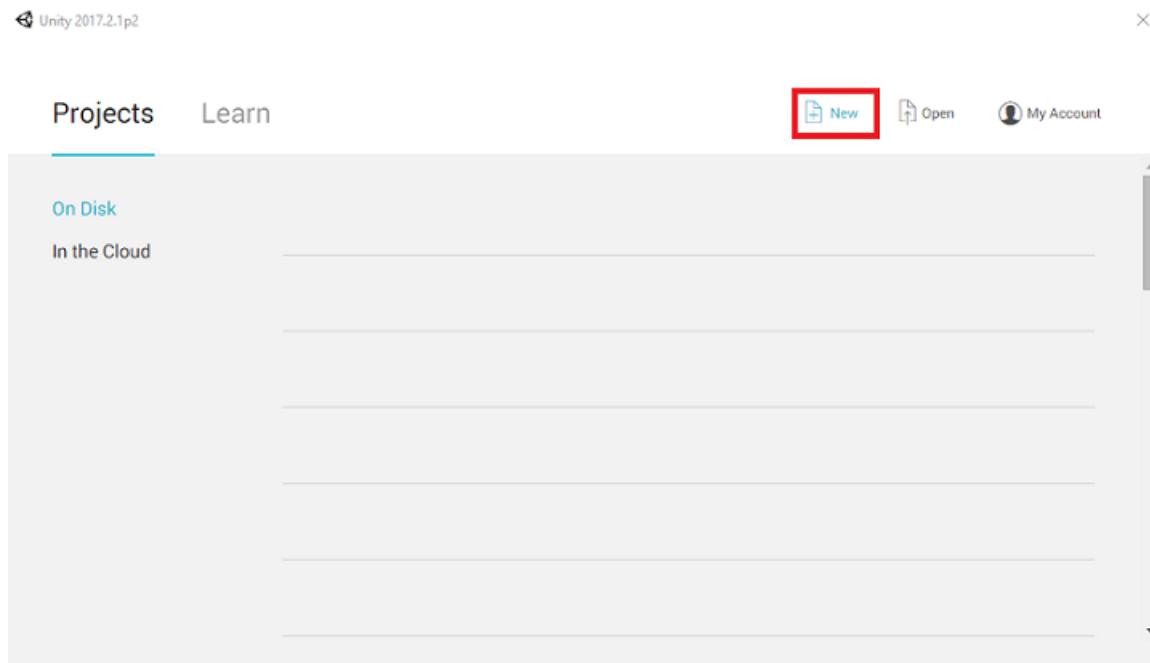
8. Make sure you know the location of ten (10) photos of the person you would like to add to your group.
9. Press **Create and Open Folder** to open Windows Explorer to the folder associated to the person. Add the ten (10) images in the folder. These must be of *JPG* or *PNG* file format.
10. Click on **Submit To Azure**. A counter will show you the state of the submission, followed by a message when it has completed.
11. Once the counter has finished and a confirmation message has been displayed click on **Train** to train your Service.

Once the process has completed, you are ready to move into Unity.

Chapter 3 - Set up the Unity project

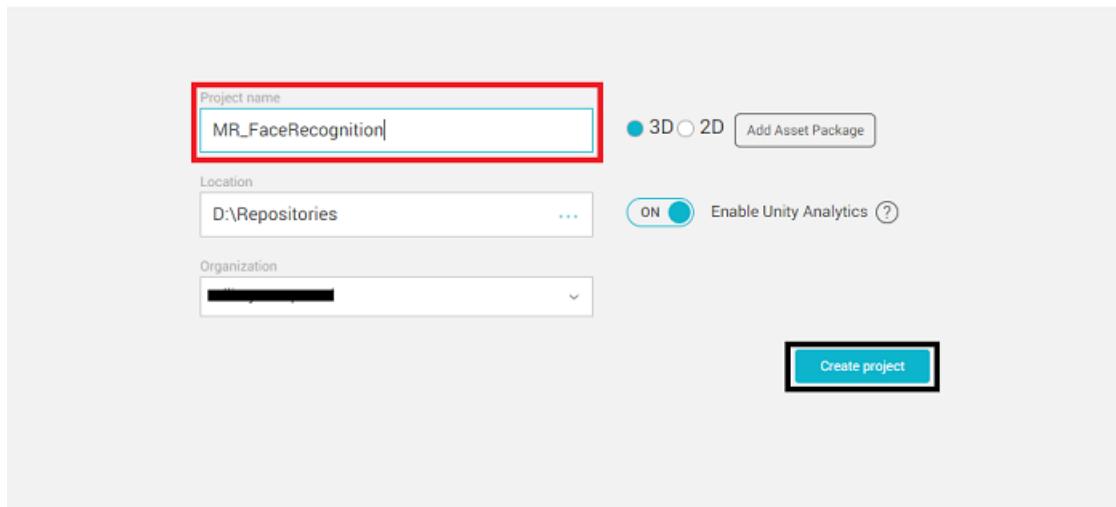
The following is a typical set up for developing with mixed reality, and as such, is a good template for other projects.

1. Open *Unity* and click **New**.

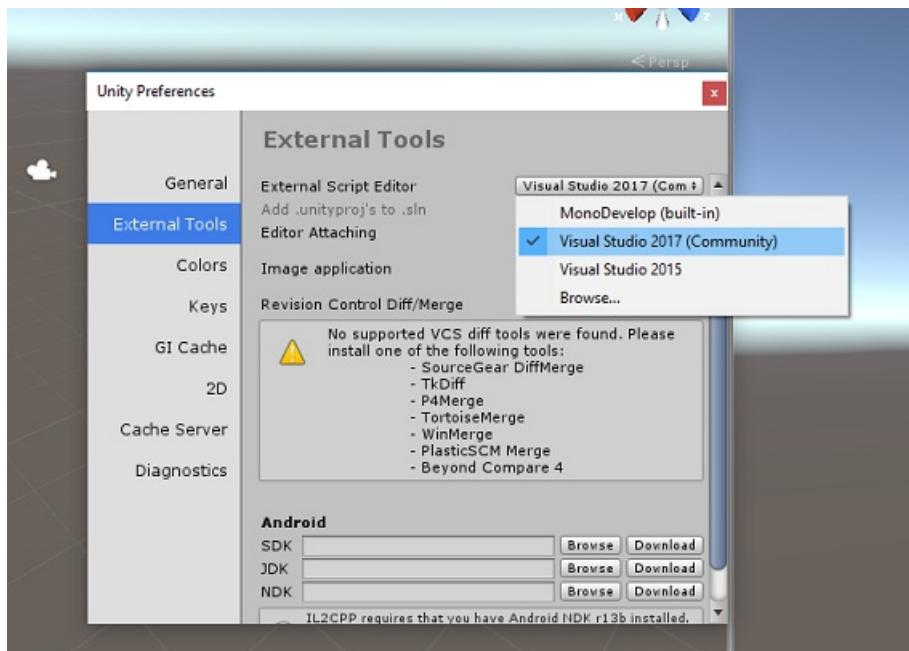


2. You will now need to provide a Unity Project name. Insert **MR_FaceRecognition**. Make sure the project type is set to **3D**. Set the **Location** to somewhere appropriate for you (remember, closer to root directories is better). Then, click **Create project**.

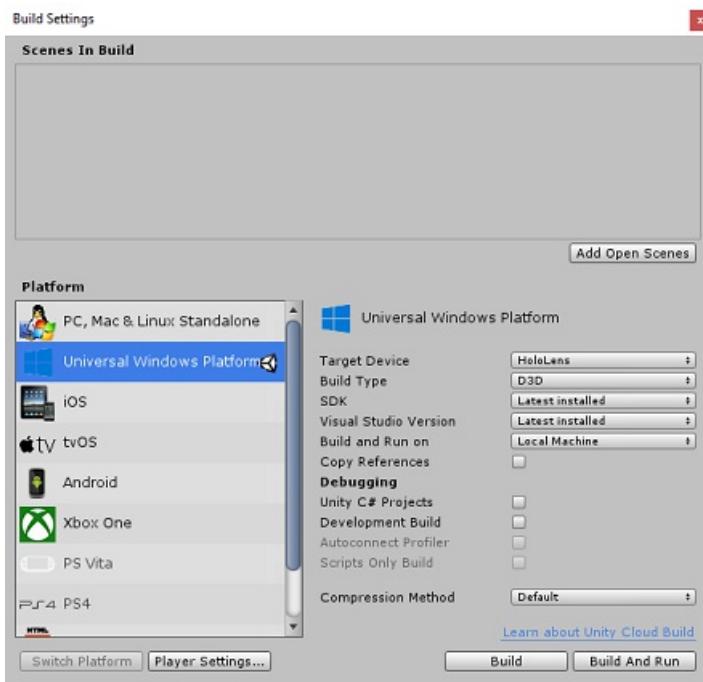
Projects Learn

[New](#) [Open](#) [My Account](#)

3. With Unity open, it is worth checking the default **Script Editor** is set to **Visual Studio**. Go to **Edit > Preferences** and then from the new window, navigate to **External Tools**. Change **External Script Editor** to **Visual Studio 2017**. Close the **Preferences** window.



4. Next, go to **File > Build Settings** and switch the platform to **Universal Windows Platform**, by clicking on the **Switch Platform** button.



5. Go to **File > Build Settings** and make sure that:

a. **Target Device** is set to **HoloLens**

For the immersive headsets, set **Target Device** to *Any Device*.

b. **Build Type** is set to **D3D**

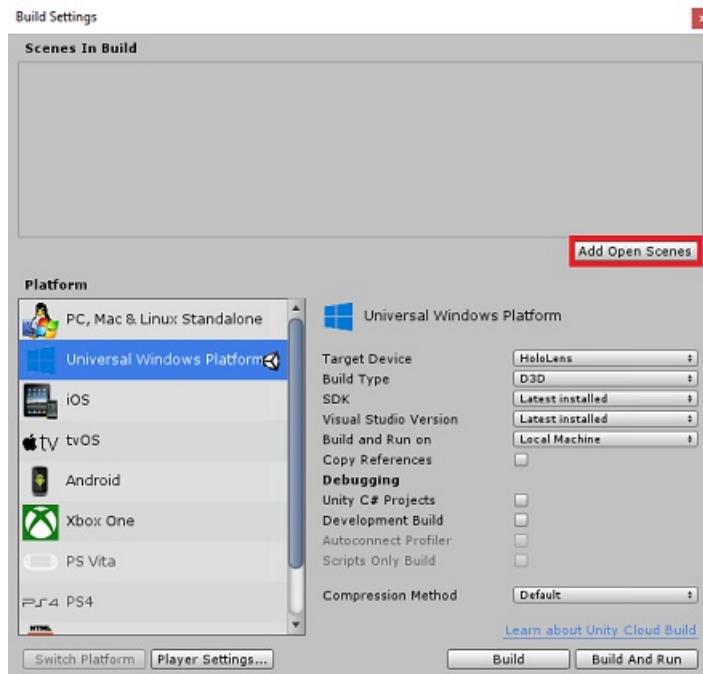
c. **SDK** is set to **Latest installed**

d. **Visual Studio Version** is set to **Latest installed**

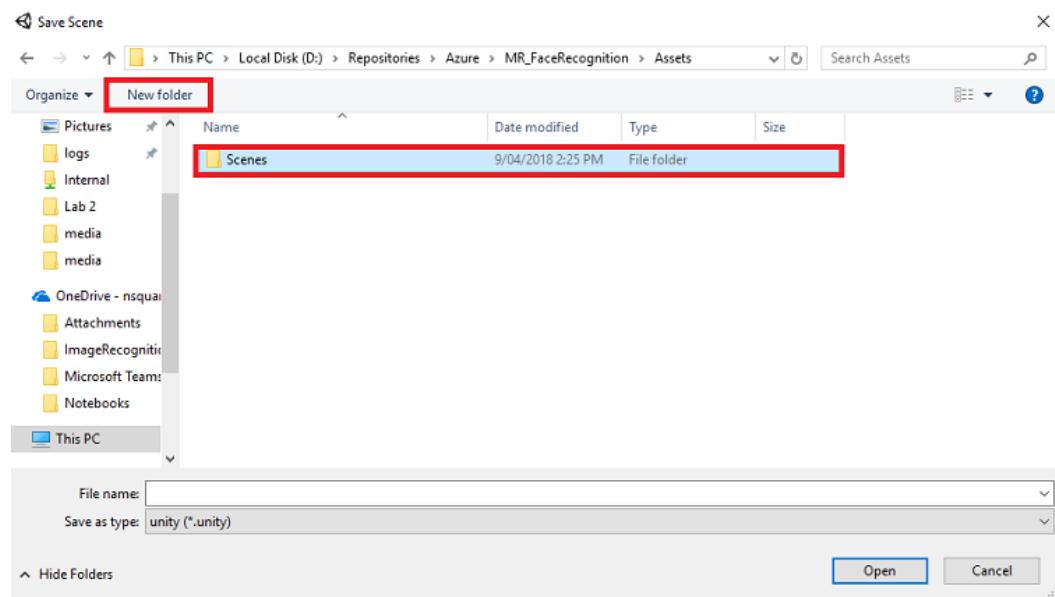
e. **Build and Run** is set to **Local Machine**

f. Save the scene and add it to the build.

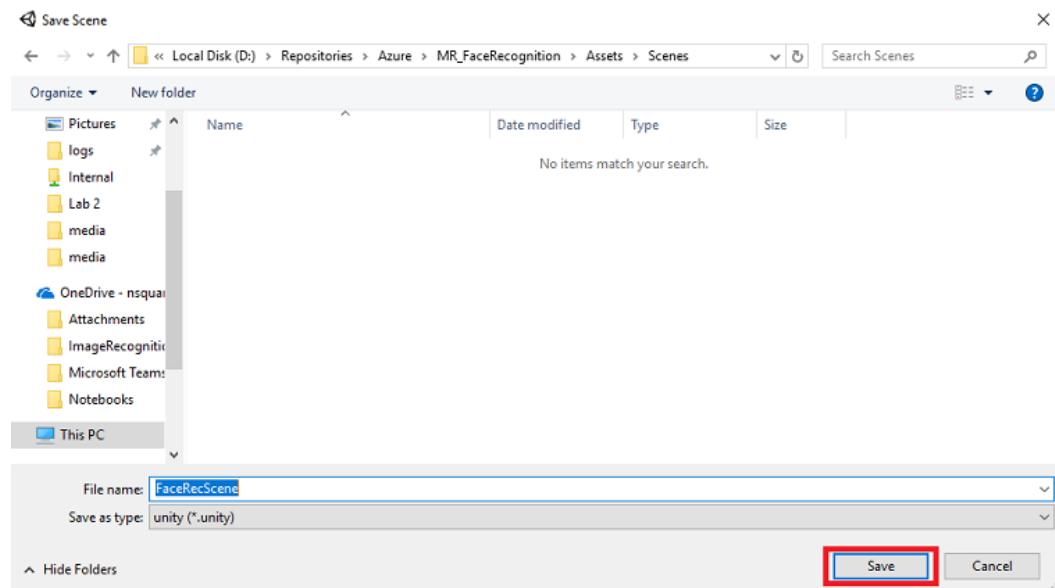
a. Do this by selecting **Add Open Scenes**. A save window will appear.



b. Select the **New folder** button, to create a new folder, name it **Scenes**.

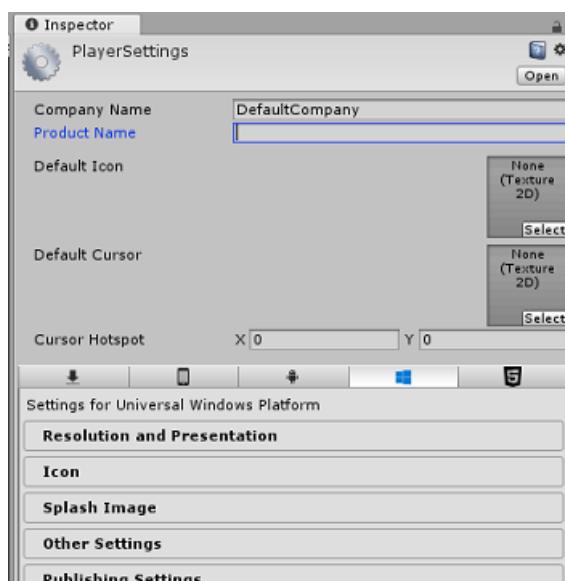


- c. Open your newly created **Scenes** folder, and then in the **File name:** text field, type **FaceRecScene**, then press **Save**.



g. The remaining settings, in *Build Settings*, should be left as default for now.

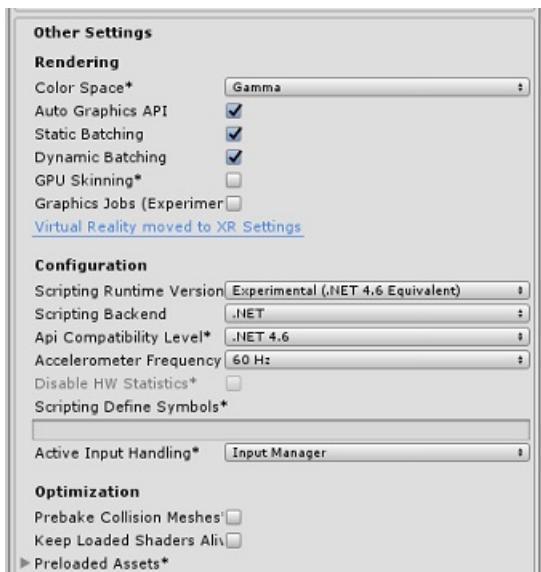
6. In the *Build Settings* window, click on the **Player Settings** button, this will open the related panel in the space where the *Inspector* is located.



7. In this panel, a few settings need to be verified:

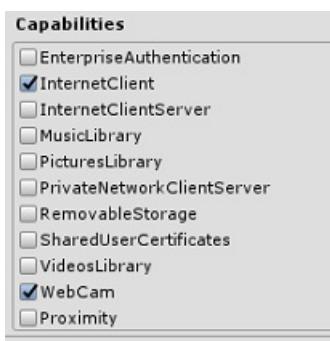
a. In the **Other Settings** tab:

- a. **Scripting Runtime Version** should be **Experimental (.NET 4.6 Equivalent)**. Changing this will trigger a need to restart the Editor.
- b. **Scripting Backend** should be **.NET**
- c. **API Compatibility Level** should be **.NET 4.6**

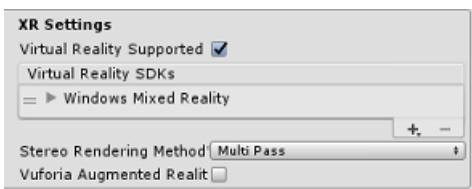


b. Within the **Publishing Settings** tab, under **Capabilities**, check:

- **InternetClient**
- **Webcam**



c. Further down the panel, in **XR Settings** (found below **Publish Settings**), tick **Virtual Reality Supported**, make sure the **Windows Mixed Reality SDK** is added.



8. Back in **Build Settings**, **Unity C# Projects** is no longer greyed out; tick the checkbox next to this.

9. Close the Build Settings window.

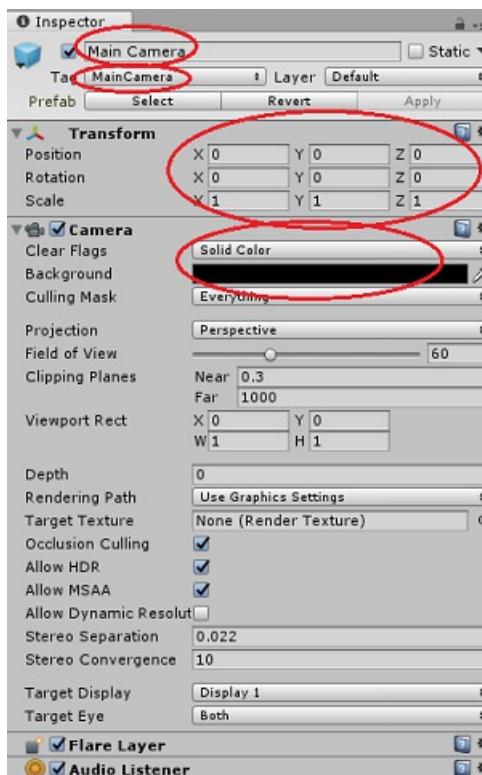
10. Save your Scene and Project (**FILE > SAVE SCENE / FILE > SAVE PROJECT**).

Chapter 4 - Main Camera setup

IMPORTANT

If you wish to skip the *Unity Set up* component of this course, and continue straight into code, feel free to [download this .unitypackage](#), and import it into your project as a **Custom Package**. Be aware that this package also includes the import of the *Newtonsoft DLL*, covered in [Chapter 5](#). With this imported, you can continue from [Chapter 6](#).

1. In the *Hierarchy Panel*, select the **Main Camera**.
2. Once selected, you will be able to see all the components of the **Main Camera** in the *Inspector Panel*.
 - a. The **Camera object** must be named **Main Camera** (note the spelling!)
 - b. The Main Camera **Tag** must be set to **MainCamera** (note the spelling!)
 - c. Make sure the **Transform Position** is set to **0, 0, 0**
 - d. Set **Clear Flags** to **Solid Color**
 - e. Set the **Background** Color of the Camera Component to **Black, Alpha 0 (Hex Code: #00000000)**



Chapter 5 – Import the Newtonsoft.Json library

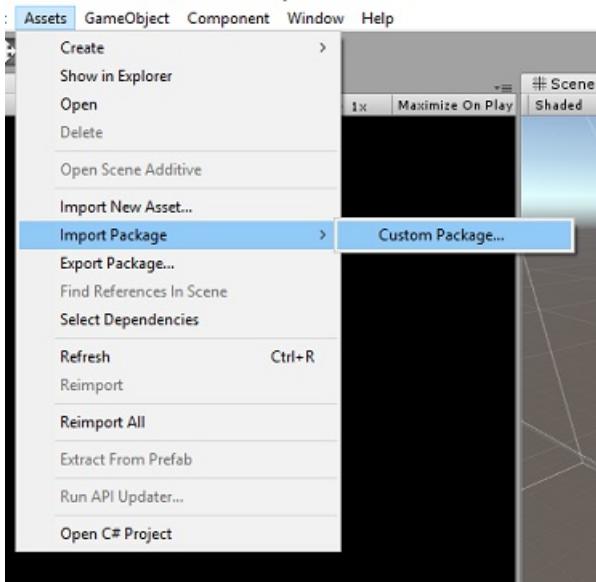
IMPORTANT

If you imported the '.unitypackage' in the [last Chapter](#), you can skip this Chapter.

To help you deserialize and serialize objects received and sent to the Bot Service you need to download the *Newtonsoft.Json* library. You will find a compatible version already organized with the correct Unity folder structure in this [Unity package file](#).

To import the library:

1. Download the Unity Package.
2. Click on **Assets, Import Package, Custom Package**.



3. Look for the Unity Package you have downloaded, and click Open.
4. Make sure all the components of the package are ticked and click **Import**.



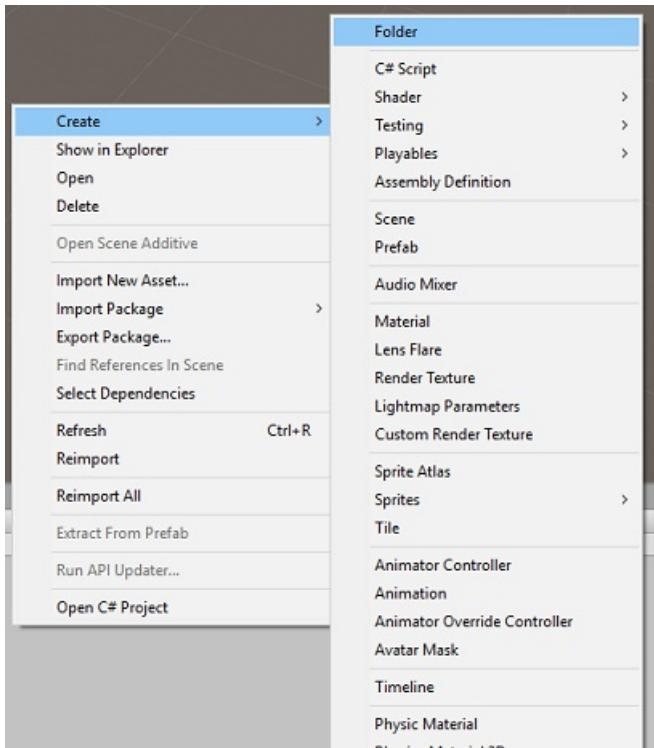
Chapter 6 - Create the FaceAnalysis class

The purpose of the *FaceAnalysis* class is to host the methods necessary to communicate with your Azure Face Recognition Service.

- After sending the service a capture image, it will analyse it and identify the faces within, and determine if any belong to a known person.
- If a known person is found, this class will display its name as UI text in the scene.

To create the *FaceAnalysis* class:

1. Right-click in the *Assets Folder* located in the Project Panel, then click on **Create > Folder**. Call the folder **Scripts**.



2. Double click on the folder just created, to open it.
3. Right-click inside the folder, then click on **Create > C# Script**. Call the script *FaceAnalysis*.
4. Double click on the new *FaceAnalysis* script to open it with Visual Studio 2017.
5. Enter the following namespaces above the *FaceAnalysis* class:

```
using Newtonsoft.Json;
using System.Collections;
using System.Collections.Generic;
using System.IO;
using System.Text;
using UnityEngine;
using UnityEngine.Networking;
```

6. You now need to add all of the objects which are used for deserialising. These objects need to be added **outside** of the *FaceAnalysis* script (beneath the bottom curly bracket).

```

/// <summary>
/// The Person Group object
/// </summary>
public class Group_RootObject
{
    public string personGroupId { get; set; }
    public string name { get; set; }
    public object userData { get; set; }
}

/// <summary>
/// The Person Face object
/// </summary>
public class Face_RootObject
{
    public string faceId { get; set; }
}

/// <summary>
/// Collection of faces that needs to be identified
/// </summary>
public class FacesToIdentify_RootObject
{
    public string personGroupId { get; set; }
    public List<string> faceIds { get; set; }
    public int maxNumOfCandidatesReturned { get; set; }
    public double confidenceThreshold { get; set; }
}

/// <summary>
/// Collection of Candidates for the face
/// </summary>
public class Candidate_RootObject
{
    public string faceId { get; set; }
    public List<Candidate> candidates { get; set; }
}

public class Candidate
{
    public string personId { get; set; }
    public double confidence { get; set; }
}

/// <summary>
/// Name and Id of the identified Person
/// </summary>
public class IdentifiedPerson_RootObject
{
    public string personId { get; set; }
    public string name { get; set; }
}

```

7. The *Start()* and *Update()* methods will not be used, so delete them now.

8. Inside the *FaceAnalysis* class, add the following variables:

```

/// <summary>
/// Allows this class to behave like a singleton
/// </summary>
public static FaceAnalysis Instance;

/// <summary>
/// The analysis result text
/// </summary>
private TextMesh labelText;

/// <summary>
/// Bytes of the image captured with camera
/// </summary>
internal byte[] imageBytes;

/// <summary>
/// Path of the image captured with camera
/// </summary>
internal string imagePath;

/// <summary>
/// Base endpoint of Face Recognition Service
/// </summary>
const string baseEndpoint = "https://westus.api.cognitive.microsoft.com/face/v1.0/";

/// <summary>
/// Auth key of Face Recognition Service
/// </summary>
private const string key = "- Insert your key here -";

/// <summary>
/// Id (name) of the created person group
/// </summary>
private const string personGroupId = "- Insert your group Id here -";

```

NOTE

Replace the **key** and the **personGroupId** with your Service Key and the Id of the group that you created previously.

- Add the *Awake()* method, which initialises the class, adding the *ImageCapture* class to the Main Camera and calls the Label creation method:

```

/// <summary>
/// Initialises this class
/// </summary>
private void Awake()
{
    // Allows this instance to behave like a singleton
    Instance = this;

    // Add the ImageCapture Class to this Game Object
    gameObject.AddComponent<ImageCapture>();

    // Create the text label in the scene
    CreateLabel();
}

```

- Add the *CreateLabel()* method, which creates the *Label* object to display the analysis result:

```
/// <summary>
/// Spawns cursor for the Main Camera
/// </summary>
private void CreateLabel()
{
    // Create a sphere as new cursor
    GameObject newLabel = new GameObject();

    // Attach the label to the Main Camera
    newLabel.transform.parent = gameObject.transform;

    // Resize and position the new cursor
    newLabel.transform.localScale = new Vector3(0.4f, 0.4f, 0.4f);
    newLabel.transform.position = new Vector3(0f, 3f, 60f);

    // Creating the text of the Label
    labelText = newLabel.AddComponent<TextMesh>();
    labelText.anchor = TextAnchor.MiddleCenter;
    labelText.alignment = TextAlignment.Center;
    labelText.tabSize = 4;
    labelText.fontSize = 50;
    labelText.text = ".";
}
```

11. Add the *DetectFacesFromImage()* and *GetImageAsByteArray()* method. The former will request the Face Recognition Service to detect any possible face in the submitted image, while the latter is necessary to convert the captured image into a bytes array:

```

/// <summary>
/// Detect faces from a submitted image
/// </summary>
internal IEnumerator DetectFacesFromImage()
{
    WWWForm webForm = new WWWForm();
    string detectFacesEndpoint = $"{baseEndpoint}detect";

    // Change the image into a bytes array
    imageBytes = GetImageAsByteArray(imagePath);

    using (UnityWebRequest www =
        UnityWebRequest.Post(detectFacesEndpoint, webForm))
    {
        www.SetRequestHeader("Ocp-Apim-Subscription-Key", key);
        www.SetRequestHeader("Content-Type", "application/octet-stream");
        www.uploadHandler.contentType = "application/octet-stream";
        www.uploadHandler = new UploadHandlerRaw(imageBytes);
        www.downloadHandler = new DownloadHandlerBuffer();

        yield return www.SendWebRequest();
        string jsonResponse = www.downloadHandler.text;
        Face_RootObject[] face_RootObject =
            JsonConvert.DeserializeObject<Face_RootObject[]>(jsonResponse);

        List<string> facesIdList = new List<string>();
        // Create a list with the face Ids of faces detected in image
        foreach (Face_RootObject faceRO in face_RootObject)
        {
            facesIdList.Add(faceRO.faceId);
            Debug.Log($"Detected face - Id: {faceRO.faceId}");
        }

        StartCoroutine(IdentifyFaces(facesIdList));
    }
}

/// <summary>
/// Returns the contents of the specified file as a byte array.
/// </summary>
static byte[] GetImageAsByteArray(string imagePath)
{
    FileStream fileStream = new FileStream(imagePath, FileMode.Open, FileAccess.Read);
    BinaryReader binaryReader = new BinaryReader(fileStream);
    return binaryReader.ReadBytes((int)fileStream.Length);
}

```

12. Add the *IdentifyFaces()* method, which requests the *Face Recognition Service* to identify any known face previously detected in the submitted image. The request will return an id of the identified person but not the name:

```

/// <summary>
/// Identify the faces found in the image within the person group
/// </summary>
internal IEnumerator IdentifyFaces(List<string> listOffacesIdToIdentify)
{
    // Create the object hosting the faces to identify
    FacesToIdentify_RootObject facesToIdentify = new FacesToIdentify_RootObject();
    facesToIdentify.faceIds = new List<string>();
    facesToIdentify.personGroupId = personGroupId;
    foreach (string facesId in listOffacesIdToIdentify)
    {
        facesToIdentify.faceIds.Add(facesId);
    }
    facesToIdentify.maxNumOfCandidatesReturned = 1;
    facesToIdentify.confidenceThreshold = 0.5;

    // Serialise to Json format
    string facesToIdentifyJson = JsonConvert.SerializeObject(facesToIdentify);
    // Change the object into a bytes array
    byte[] facesData = Encoding.UTF8.GetBytes(facesToIdentifyJson);

    WWWForm webForm = new WWWForm();
    string detectFacesEndpoint = $"{baseEndpoint}identify";

    using (UnityWebRequest www = UnityWebRequest.Post(detectFacesEndpoint, webForm))
    {
        www.SetRequestHeader("Ocp-Apim-Subscription-Key", key);
        www.SetRequestHeader("Content-Type", "application/json");
        www.uploadHandler.contentType = "application/json";
        www.uploadHandler = new UploadHandlerRaw(facesData);
        www.downloadHandler = new DownloadHandlerBuffer();

        yield return www.SendWebRequest();
        string jsonResponse = www.downloadHandler.text;
        Debug.Log($"Get Person - jsonResponse: {jsonResponse}");
        Candidate_RootObject [] candidate_RootObject =
        JsonConvert.DeserializeObject<Candidate_RootObject[]>(jsonResponse);

        // For each face to identify that has been submitted, display its candidate
        foreach (Candidate_RootObject candidateRO in candidate_RootObject)
        {
            StartCoroutine(GetPerson(candidateRO.candidates[0].personId));

            // Delay the next "GetPerson" call, so all faces candidate are displayed properly
            yield return new WaitForSeconds(3);
        }
    }
}

```

13. Add the *GetPerson()* method. By providing the person id, this method then requests for the *Face Recognition Service* to return the name of the identified person:

```

/// <summary>
/// Provided a personId, retrieve the person name associated with it
/// </summary>
internal IEnumerator GetPerson(string personId)
{
    string getGroupEndpoint = $"{baseEndpoint}persons/groups/{personGroupId}/persons/{personId}?";
    WWWForm webForm = new WWWForm();

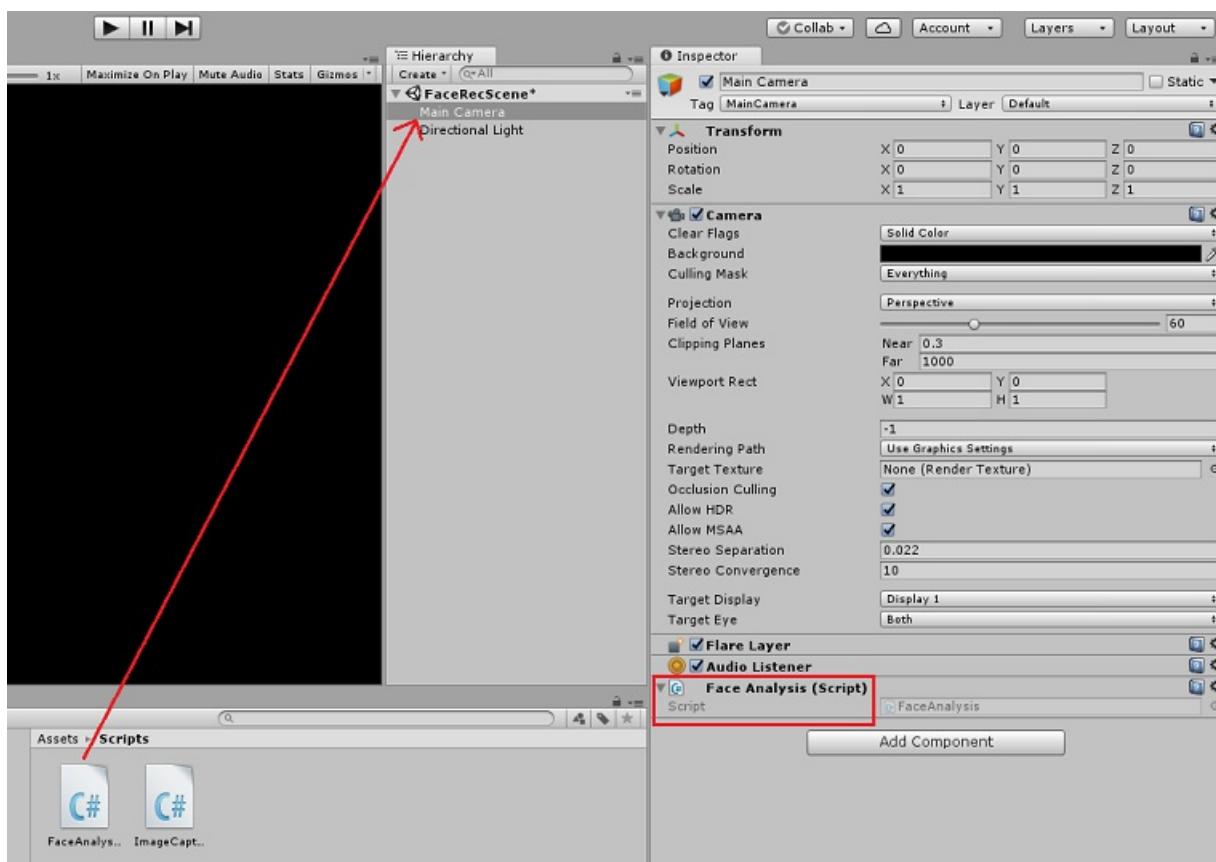
    using (UnityWebRequest www = UnityWebRequest.Get(getGroupEndpoint))
    {
        www.SetRequestHeader("Ocp-Apim-Subscription-Key", key);
        www.downloadHandler = new DownloadHandlerBuffer();
        yield return www.SendWebRequest();
        string jsonResponse = www.downloadHandler.text;

        Debug.Log($"Get Person - jsonResponse: {jsonResponse}");
        IdentifiedPerson_RootObject identifiedPerson_RootObject =
        JsonConvert.DeserializeObject<IdentifiedPerson_RootObject>(jsonResponse);

        // Display the name of the person in the UI
        labelText.text = identifiedPerson_RootObject.name;
    }
}

```

14. Remember to **Save** the changes before going back to the Unity Editor.
15. In the Unity Editor, drag the FaceAnalysis script from the Scripts folder in Project panel to the Main Camera object in the *Hierarchy panel*. The new script component will be so added to the Main Camera.



Chapter 7 - Create the ImageCapture class

The purpose of the *ImageCapture* class is to host the methods necessary to communicate with your *Azure Face Recognition Service* to analyse the image you will capture, identifying faces in it and determining if it belongs to a known person. If a known person is found, this class will display its name as UI text in the scene.

To create the *ImageCapture* class:

1. Right-click inside the **Scripts** folder you have created previously, then click on **Create, C# Script**. Call the script *ImageCapture*.
2. Double click on the new *ImageCapture* script to open it with Visual Studio 2017.
3. Enter the following namespaces above the *ImageCapture* class:

```
using System.IO;
using System.Linq;
using UnityEngine;
using UnityEngine.XR.WSA.Input;
using UnityEngine.XR.WSA.WebCam;
```

4. Inside the *ImageCapture* class, add the following variables:

```
/// <summary>
/// Allows this class to behave like a singleton
/// </summary>
public static ImageCapture instance;

/// <summary>
/// Keeps track of tapCounts to name the captured images
/// </summary>
private int tapsCount;

/// <summary>
/// PhotoCapture object used to capture images on HoloLens
/// </summary>
private PhotoCapture photoCaptureObject = null;

/// <summary>
/// HoloLens class to capture user gestures
/// </summary>
private GestureRecognizer recognizer;
```

5. Add the *Awake()* and *Start()* methods necessary to initialise the class and allow the HoloLens to capture the user's gestures:

```
/// <summary>
/// Initialises this class
/// </summary>
private void Awake()
{
    instance = this;
}

/// <summary>
/// Called right after Awake
/// </summary>
void Start()
{
    // Initialises user gestures capture
    recognizer = new GestureRecognizer();
    recognizer.SetRecognizableGestures(GestureSettings.Tap);
    recognizer.Tapped += TapHandler;
    recognizer.StartCapturingGestures();
}
```

6. Add the *TapHandler()* which is called when the user performs a *Tap* gesture:

```

/// <summary>
/// Respond to Tap Input.
/// </summary>
private void TapHandler(TappedEventArgs obj)
{
    tapsCount++;
    ExecuteImageCaptureAndAnalysis();
}

```

7. Add the *ExecuteImageCaptureAndAnalysis()* method, which will begin the process of Image Capturing:

```

/// <summary>
/// Begin process of Image Capturing and send To Azure Computer Vision service.
/// </summary>
private void ExecuteImageCaptureAndAnalysis()
{
    Resolution cameraResolution = PhotoCapture.SupportedResolutions.OrderByDescending
        ((res) => res.width * res.height).First();
    Texture2D targetTexture = new Texture2D(cameraResolution.width, cameraResolution.height);

    PhotoCapture.CreateAsync(false, delegate (PhotoCapture captureObject)
    {
        photoCaptureObject = captureObject;

        CameraParameters c = new CameraParameters();
        c.hologramOpacity = 0.0f;
        c.cameraResolutionWidth = targetTexture.width;
        c.cameraResolutionHeight = targetTexture.height;
        c.pixelFormat = CapturePixelFormat.BGRA32;

        captureObject.StartPhotoModeAsync(c, delegate (PhotoCapture.PhotoCaptureResult result)
        {
            string filename = string.Format(@"CapturedImage{0}.jpg", tapsCount);
            string filePath = Path.Combine(Application.persistentDataPath, filename);

            // Set the image path on the FaceAnalysis class
            FaceAnalysis.Instance.imagePath = filePath;

            photoCaptureObject.TakePhotoAsync
                (filePath, PhotoCaptureFileOutputFormat.JPG, OnCapturedPhotoToDisk);
        });
    });
}

```

8. Add the handlers that are called when the photo capture process has been completed:

```

/// <summary>
/// Called right after the photo capture process has concluded
/// </summary>
void OnCapturedPhotoToDisk(PhotoCapture.PhotoCaptureResult result)
{
    photoCaptureObject.StopPhotoModeAsync(OnStoppedPhotoMode);
}

/// <summary>
/// Register the full execution of the Photo Capture. If successfull, it will begin the Image
Analysis process.
/// </summary>
void OnStoppedPhotoMode(PhotoCapture.PhotoCaptureResult result)
{
    photoCaptureObject.Dispose();
    photoCaptureObject = null;

    // Request image caputer analysis
    StartCoroutine(FaceAnalysis.Instance.DetectFacesFromImage());
}

```

9. Remember to **Save** the changes before going back to the Unity Editor.

Chapter 8 - Building the solution

To perform a thorough test of your application you will need to sideload it onto your HoloLens.

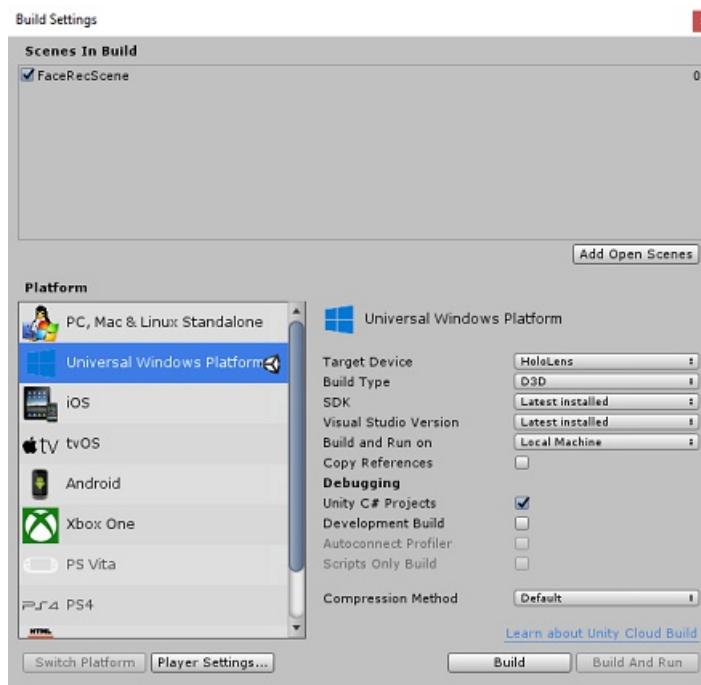
Before you do, ensure that:

- All the settings mentioned in the Chapter 3 are set correctly.
- The script *FaceAnalysis* is attached to the Main Camera object.
- Both the **Auth Key** and **Group Id** have been set within the *FaceAnalysis* script.

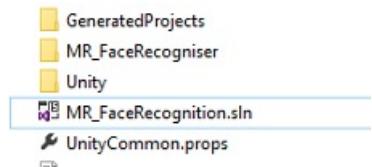
At this point you are ready to build the Solution. Once the Solution has been built, you will be ready to deploy your application.

To begin the Build process:

1. Save the current scene by clicking on File, Save.
2. Go to File, Build Settings, click on Add Open Scenes.
3. Make sure to tick Unity C# Projects.



4. Press Build. Upon doing so, Unity will launch a File Explorer window, where you need to create and then select a folder to build the app into. Create that folder now, within the Unity project, and call it App. Then with the App folder selected, press Select Folder.
5. Unity will begin building your project, out to the App folder.
6. Once Unity has finished building (it might take some time), it will open a File Explorer window at the location of your build.

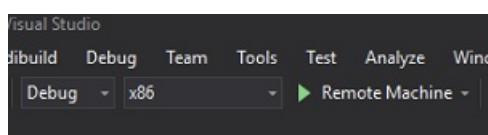


7. Open your App folder, and then open the new Project Solution (as seen above, MR_FaceRecognition.sln).

Chapter 9 - Deploying your application

To deploy on HoloLens:

1. You will need the IP Address of your HoloLens (for Remote Deploy), and to ensure your HoloLens is in **Developer Mode**. To do this:
 - a. Whilst wearing your HoloLens, open the **Settings**.
 - b. Go to **Network & Internet > Wi-Fi > Advanced Options**
 - c. Note the **IPv4** address.
 - d. Next, navigate back to **Settings**, and then to **Update & Security > For Developers**
 - e. Set Developer Mode On.
2. Navigate to your new Unity build (the App folder) and open the solution file with *Visual Studio*.
3. In the Solution Configuration select **Debug**.
4. In the Solution Platform, select **x86, Remote Machine**.



5. Go to the **Build menu** and click on **Deploy Solution**, to sideload the application to your HoloLens.
6. Your App should now appear in the list of installed apps on your HoloLens, ready to be launched!

NOTE

To deploy to immersive headset, set the **Solution Platform** to *Local Machine*, and set the **Configuration** to *Debug*, with *x86* as the **Platform**. Then deploy to the local machine, using the **Build menu**, selecting *Deploy Solution*.

Chapter 10 - Using the application

1. Wearing the HoloLens, launch the app.
2. Look at the person that you have registered with the *Face API*. Make sure that:
 - The person's face is not too distant and clearly visible
 - The environment lighting is not too dark
3. Use the tap gesture to capture the person's picture.
4. Wait for the App to send the analysis request and receive a response.
5. If the person has been successfully recognized, the person's name will appear as UI text.
6. You can repeat the capture process using the tap gesture every few seconds.

Your finished Azure Face API Application

Congratulations, you built a mixed reality app that leverages the Azure Face Recognition service to detect faces within an image, and identify any known faces.



Bonus exercises

Exercise 1

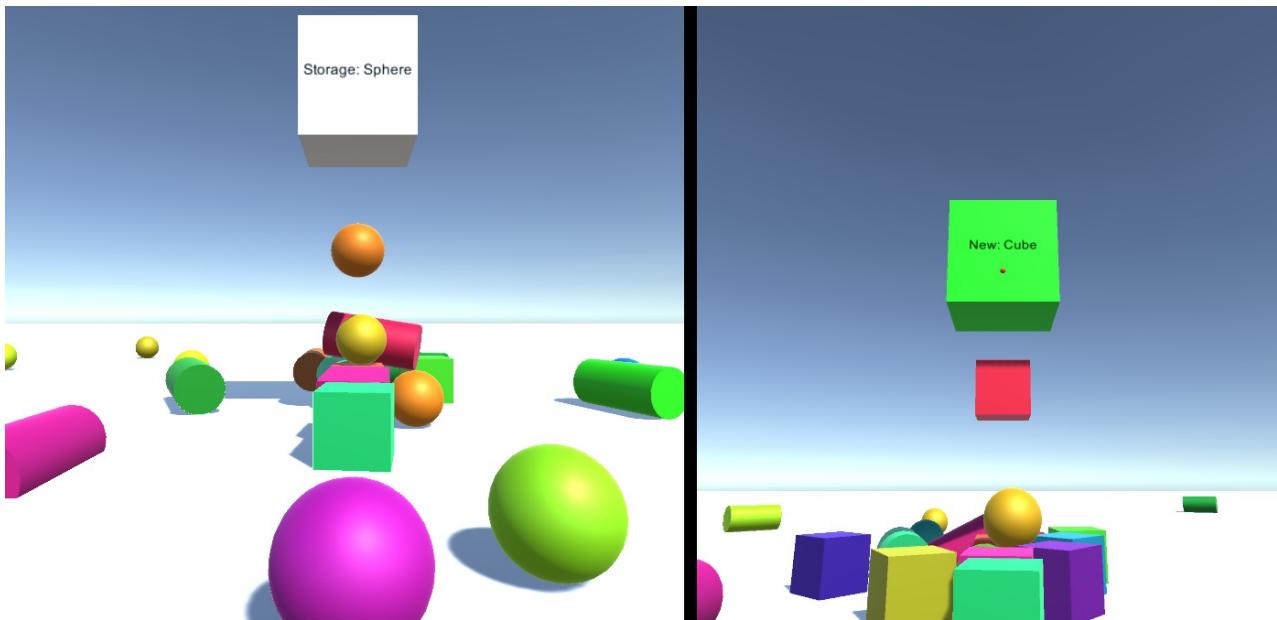
The **Azure Face API** is powerful enough to detect up to 64 faces in a single image. Extend the application, so that it could recognize two or three faces, amongst many other people.

Exercise 2

The **Azure Face API** is also able to provide back all kinds of attribute information. Integrate this into the application. This could be even more interesting, when combined with the [Emotion API](#).

MR and Azure 305: Functions and storage

11/13/2018 • 29 minutes to read • [Edit Online](#)



In this course, you will learn how to create and use Azure Functions and store data with an Azure Storage resource, within a mixed reality application.

Azure Functions is a Microsoft service, which allows developers to run small pieces of code, 'functions', in Azure. This provides a way to delegate work to the cloud, rather than your local application, which can have many benefits. *Azure Functions* supports several development languages, including C#, F#, Node.js, Java, and PHP. For more information, visit the [Azure Functions article](#).

Azure Storage is a Microsoft cloud service, which allows developers to store data, with the insurance that it will be highly available, secure, durable, scalable, and redundant. This means Microsoft will handle all maintenance, and critical problems for you. For more information, visit the [Azure Storage article](#).

Having completed this course, you will have a mixed reality immersive headset application which will be able to do the following:

1. Allow the user to gaze around a scene.
2. Trigger the spawning of objects when the user gazes at a 3D 'button'.
3. The spawned objects will be chosen by an Azure Function.
4. As each object is spawned, the application will store the object type in an *Azure File*, located in *Azure Storage*.
5. Upon loading a second time, the *Azure File* data will be retrieved, and used to replay the spawning actions from the previous instance of the application.

In your application, it is up to you as to how you will integrate the results with your design. This course is designed to teach you how to integrate an Azure Service with your Unity Project. It is your job to use the knowledge you gain from this course to enhance your mixed reality Application.

Device support

COURSE	HOLOLENS	IMMERSIVE HEADSETS
--------	----------	--------------------

NOTE

While this course primarily focuses on Windows Mixed Reality immersive (VR) headsets, you can also apply what you learn in this course to Microsoft HoloLens. As you follow along with the course, you will see notes on any changes you might need to employ to support HoloLens.

Prerequisites

NOTE

This tutorial is designed for developers who have basic experience with Unity and C#. Please also be aware that the prerequisites and written instructions within this document represent what has been tested and verified at the time of writing (May 2018). You are free to use the latest software, as listed within the [install the tools](#) article, though it should not be assumed that the information in this course will perfectly match what you'll find in newer software than what's listed below.

We recommend the following hardware and software for this course:

- A development PC, [compatible with Windows Mixed Reality](#) for immersive (VR) headset development
- [Windows 10 Fall Creators Update \(or later\)](#) with Developer mode enabled
- [The latest Windows 10 SDK](#)
- [Unity 2017.4](#)
- [Visual Studio 2017](#)
- A [Windows Mixed Reality immersive \(VR\) headset](#) or [Microsoft HoloLens](#) with Developer mode enabled
- A subscription to an Azure account for creating Azure resources
- Internet access for Azure setup and data retrieval

Before you start

To avoid encountering issues building this project, it is strongly suggested that you create the project mentioned in this tutorial in a root or near-root folder (long folder paths can cause issues at build-time).

Chapter 1 - The Azure Portal

To use the **Azure Storage Service**, you will need to create and configure a **Storage Account** in the Azure portal.

1. Log in to the [Azure Portal](#).

NOTE

If you do not already have an Azure account, you will need to create one. If you are following this tutorial in a classroom or lab situation, ask your instructor or one of the proctors for help setting up your new account.

2. Once you are logged in, click on **New** in the top left corner, and search for *Storage account*, and click **Enter**.

Home > New > Marketplace > Everything > Storage account - blob, file, table, queue

Everything

Filter

storage accounts

Results

NAME	PUBLISHER	CATEGORY
Storage account - blob, file, table, queue	Microsoft	Storage
Data Lake Store	Microsoft	Storage
SafeNet ProtectV Service Gateway, 200 Nodes	Gemalto	Compute

NOTE

The word **New** may have been replaced with **Create a resource**, in newer portals.

3. The new page will provide a description of the *Azure Storage account* service. At the bottom left of this prompt, select the **Create** button, to create an association with this service.

Storage account - blob, file, table, queue

Microsoft

Microsoft Azure provides scalable, durable cloud storage, backup, and recovery solutions for any data, big or small. It works with the infrastructure you already have to cost-effectively enhance your existing applications and business continuity strategy, and provide the storage required by your cloud applications, including unstructured text or binary data such as video, audio, and images.

[Twitter](#) [Facebook](#) [LinkedIn](#) [YouTube](#) [Blog](#) [Email](#)

PUBLISHER Microsoft

USEFUL LINKS [Documentation](#) [Service overview](#) [Pricing](#)

Create

4. Once you have clicked on **Create**:

- a. Insert a *Name* for your account, be aware this field only accepts numbers, and lowercase letters.
- b. For *Deployment model*, select **Resource manager**.
- c. For *Account kind*, select **Storage (general purpose v1)**.
- d. Determine the *Location* for your resource group (if you are creating a new Resource Group). The location would ideally be in the region where the application would run. Some Azure assets are only

available in certain regions.

- e. For *Replication* select **Read-access-geo-redundant storage (RA-GRS)**.
- f. For *Performance*, select **Standard**.
- g. Leave *Secure transfer required* as **Disabled**.
- h. Select a *Subscription*.
- i. Choose a *Resource Group* or create a new one. A resource group provides a way to monitor, control access, provision and manage billing for a collection of Azure assets. It is recommended to keep all the Azure services associated with a single project (e.g. such as these labs) under a common resource group).

If you wish to read more about Azure Resource Groups, please [visit the resource group article](#).

- j. You will also need to confirm that you have understood the Terms and Conditions applied to this Service.

- k. Select **Create**.

The screenshot shows the 'Create storage account' wizard. The top bar has a close button (X). Below it, a note says: 'The cost of your storage account depends on the usage and the options you choose below.' with a 'Learn more' link. The main form fields are:

- Name**: mynewazurestorage (highlighted with a purple border)
- Deployment model**: Resource manager (selected)
- Account kind**: Storage (general purpose v1)
- Location**: West US
- Replication**: Read-access geo-redundant storage (RA-GRS) (highlighted with a purple border)
- Performance**: Standard (selected)
- Secure transfer required**: Disabled (selected)
- Subscription**: Main Subscription
- Resource group**: Create new (radio button)
- Virtual networks**: Configure virtual networks (radio button): Enabled (selected)

At the bottom, there's a 'Pin to dashboard' checkbox, a 'Create' button (highlighted with a red border), and an 'Automation options' link.

5. Once you have clicked on **Create**, you will have to wait for the service to be created, this might take a minute.
6. A notification will appear in the portal once the Service instance is created.



7. Click on the notifications to explore your new Service instance.



8. Click the **Go to resource** button in the notification to explore your new Service instance. You will be taken to your new *Storage account* service instance.

9. Click *Access keys*, to reveal the endpoints for this cloud service. Use *Notepad* or similar, to copy one of your keys for use later. Also, note the *Connection string* value, as it will be used in the *AzureServices* class, which you will create later.

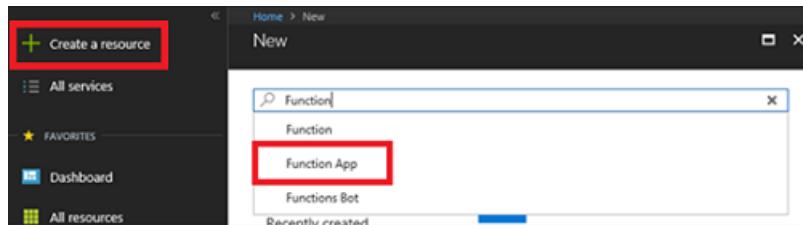
Chapter 2 - Setting up an Azure Function

You will now write an **Azure Function** in the Azure Service.

You can use an **Azure Function** to do nearly anything that you would do with a classic function in your code, the difference being that this function can be accessed by any application that has credentials to access your Azure Account.

To create an Azure Function:

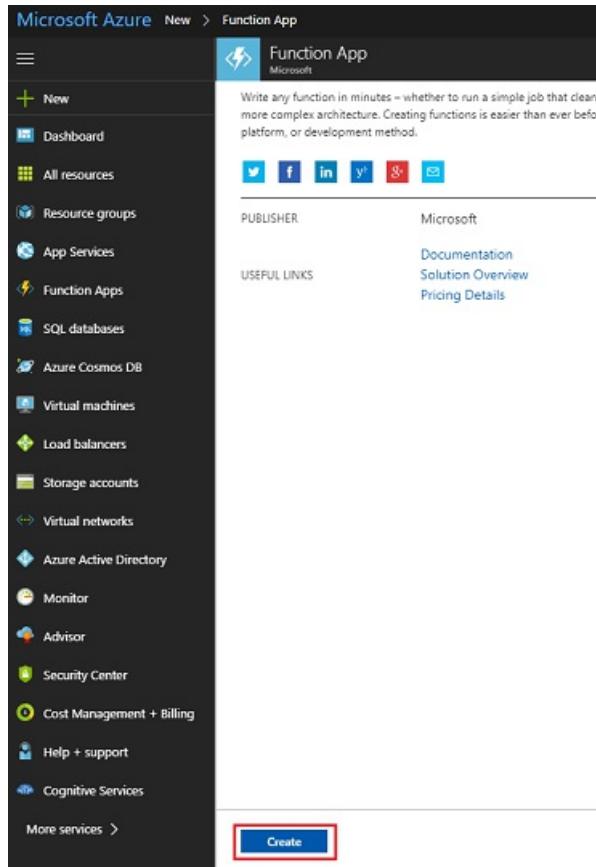
1. From your *Azure Portal*, click on **New** in the top left corner, and search for *Function App*, and click **Enter**.



NOTE

The word **New** may have been replaced with **Create a resource**, in newer portals.

2. The new page will provide a description of the *Azure Function App* service. At the bottom left of this prompt, select the **Create** button, to create an association with this service.

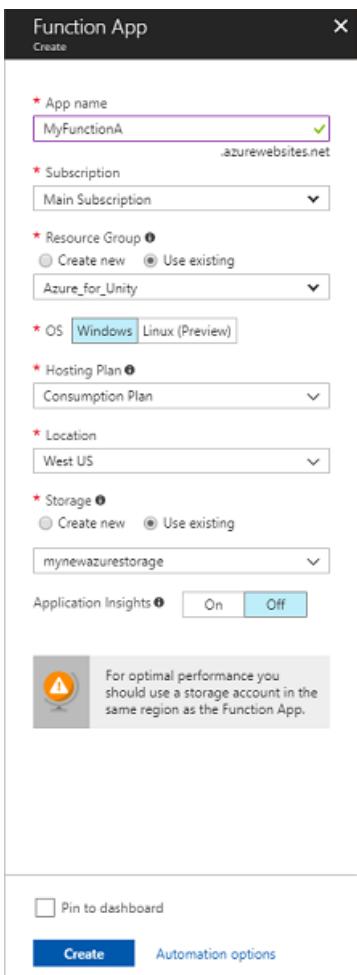


3. Once you have clicked on **Create**:

- a. Provide an *App name*. Only letters and numbers can be used here (either upper or lower case is allowed).
- b. Select your preferred *Subscription*.
- c. Choose a *Resource Group* or create a new one. A resource group provides a way to monitor, control access, provision and manage billing for a collection of Azure assets. It is recommended to keep all the Azure services associated with a single project (e.g. such as these labs) under a common resource group).

If you wish to read more about Azure Resource Groups, please [visit the resource group article](#).

- d. For this exercise, select **Windows** as the chosen **OS**.
- e. Select **Consumption Plan** for the **Hosting Plan**.
- f. Determine the **Location** for your resource group (if you are creating a new Resource Group). The location would ideally be in the region where the application would run. Some Azure assets are only available in certain regions. For optimal performance, select the same region as the storage account.
- g. For **Storage**, select **Use existing**, and then using the dropdown menu, find your previously created storage.
- h. Leave **Application Insights** off for this exercise.



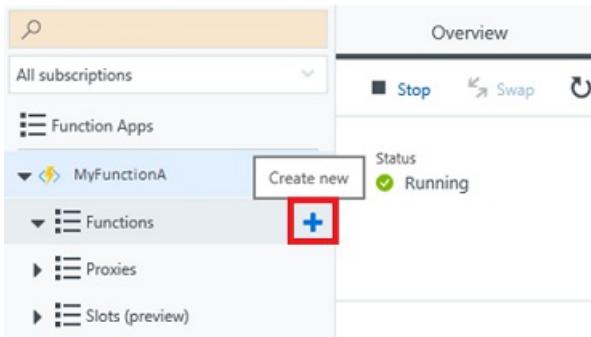
4. Click the **Create** button.
5. Once you have clicked on **Create**, you will have to wait for the service to be created, this might take a minute.
6. A notification will appear in the portal once the Service instance is created.



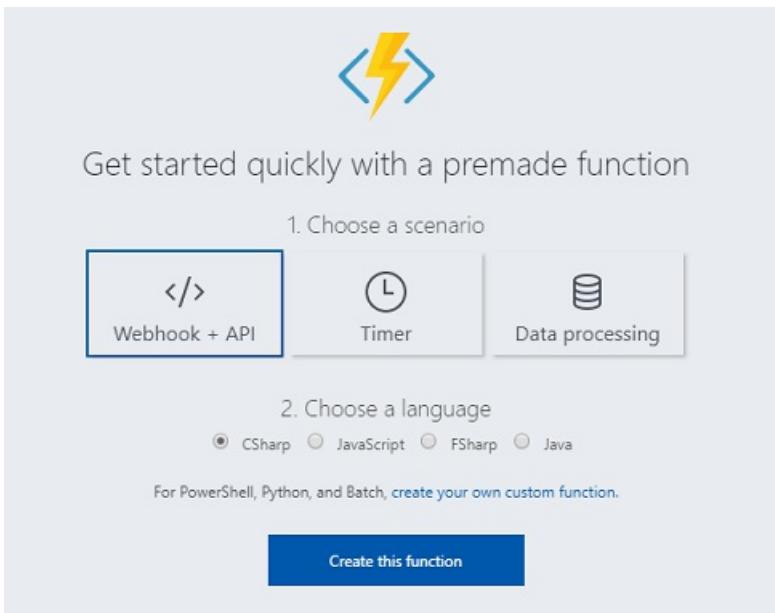
7. Click on the notifications to explore your new Service instance.



8. Click the **Go to resource** button in the notification to explore your new Service instance. You will be taken to your new *Function App* service instance.
9. On the *Function App* dashboard, hover your mouse over *Functions*, found within the panel on the left, and then click the + (plus) symbol.



10. On the next page, ensure **Webhook + API** is selected, and for *Choose a language*, select **CSharp**, as this will be the language used for this tutorial. Lastly, click the **Create this function** button.



11. You should be taken to the code page (run.csx), if not though, click on the newly created Function in the Functions list within the panel on the left.

The screenshot shows the Azure portal's 'Function Apps' blade. At the top, there's a search bar and a dropdown for 'All subscriptions'. Below that, a tree view shows 'Function Apps' expanded, with 'FuncEx1' and 'MRAzureFunction' expanded. Under 'MRAzureFunction', the 'Functions' node is expanded, and 'HttpTriggerCSharp1' is selected and highlighted with a red box. To the right of the tree view, there are several actions: 'Integrate', 'Manage', and 'Monitor' (all in blue), followed by 'Proxies', 'Slots (preview)', and 'testfunctionappnpsqu'.

12. Copy the following code into your function. This function will simply return a random integer between 0 and 2 when called. Do not worry about the existing code, feel free to paste over the top of it.

```
using System.Net;
using System.Threading.Tasks;

public static int Run(CustomObject req, TraceWriter log)
{
    Random rnd = new Random();
    int randomInt = rnd.Next(0, 3);
    return randomInt;
}

public class CustomObject
{
    public String name {get; set;}
}
```

13. Select **Save**.
14. The result should look like the image below.
15. Click on **Get function URL** and take note of the *endpoint* displayed. You will need to insert it into the *AzureServices* class that you will create later in this course.

```

1 | using System.Net;
2 | using System.Threading.Tasks;
3 |
4 | public static int Run(CustomObject req, TraceWriter log)
5 | {
6 |     Random rnd = new Random();
7 |     int randomInt = rnd.Next(0, 3);
8 |     return randomInt;
9 | }
10 |
11 | public class CustomObject
12 | {
13 |     public String name {get; set;}
14 | }
15 |

```



Chapter 3 - Setting up the Unity project

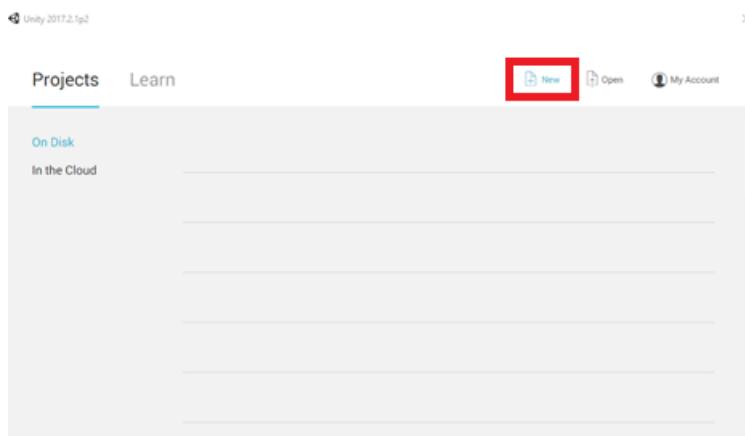
The following is a typical set up for developing with Mixed Reality, and as such, is a good template for other projects.

Set up and test your mixed reality immersive headset.

NOTE

You will **not** require Motion Controllers for this course. If you need support setting up the immersive headset, please [visit the mixed reality set up article](#).

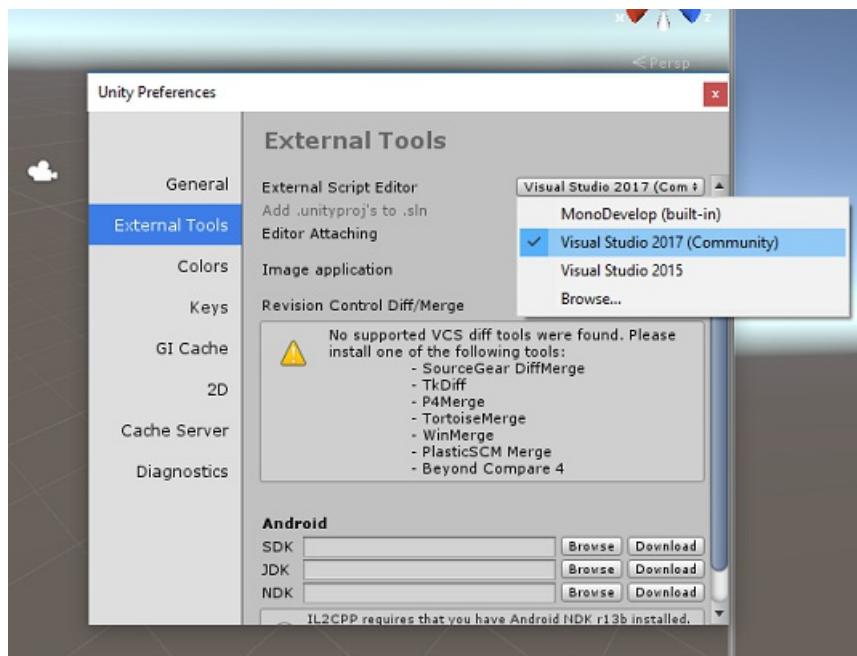
1. Open Unity and click **New**.



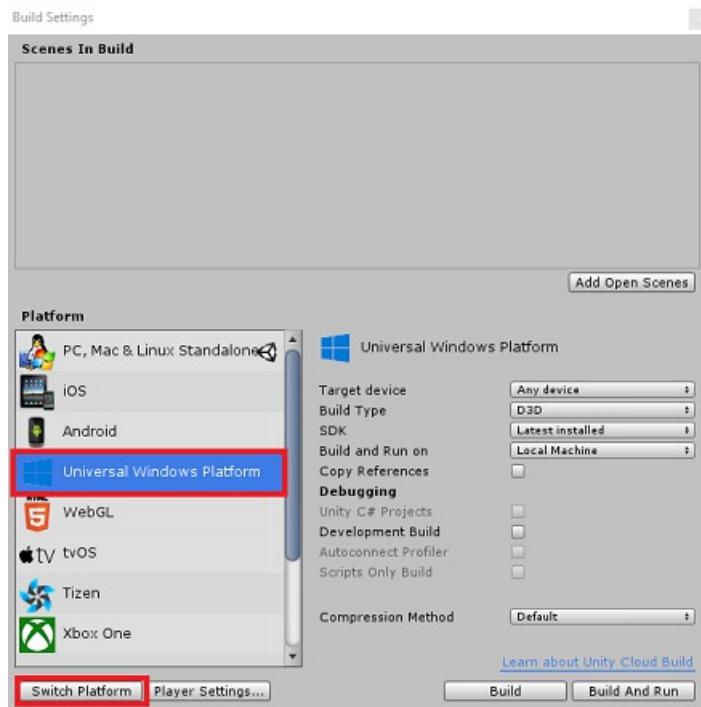
2. You will now need to provide a Unity Project name. Insert **MR_Azure_Functions**. Make sure the project type is set to **3D**. Set the *Location* to somewhere appropriate for you (remember, closer to root directories is better). Then, click **Create project**.



3. With Unity open, it is worth checking the default **Script Editor** is set to **Visual Studio**. Go to **Edit > Preferences** and then from the new window, navigate to **External Tools**. Change **External Script Editor** to **Visual Studio 2017**. Close the **Preferences** window.



4. Next, go to **File > Build Settings** and switch the platform to **Universal Windows Platform**, by clicking on the **Switch Platform** button.



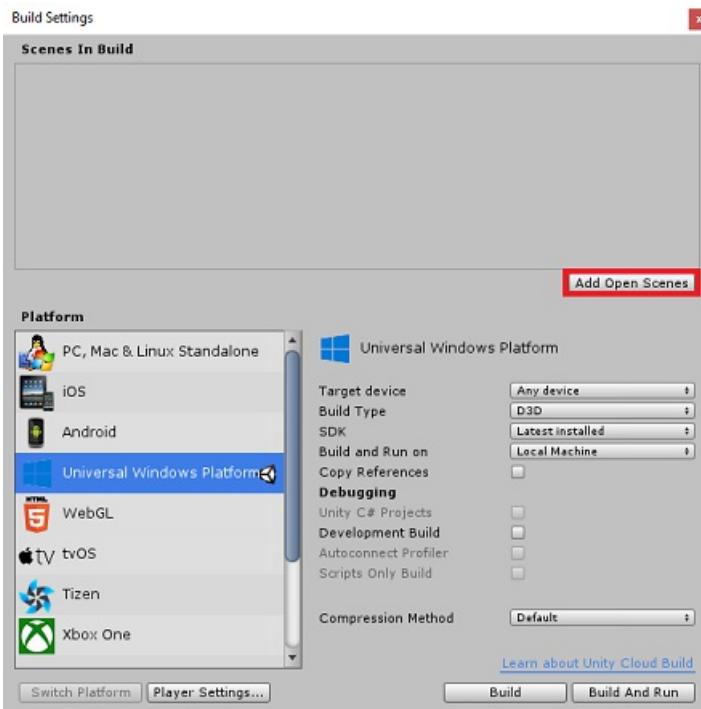
5. Go to **File > Build Settings** and make sure that:

- Target Device** is set to **Any Device**.

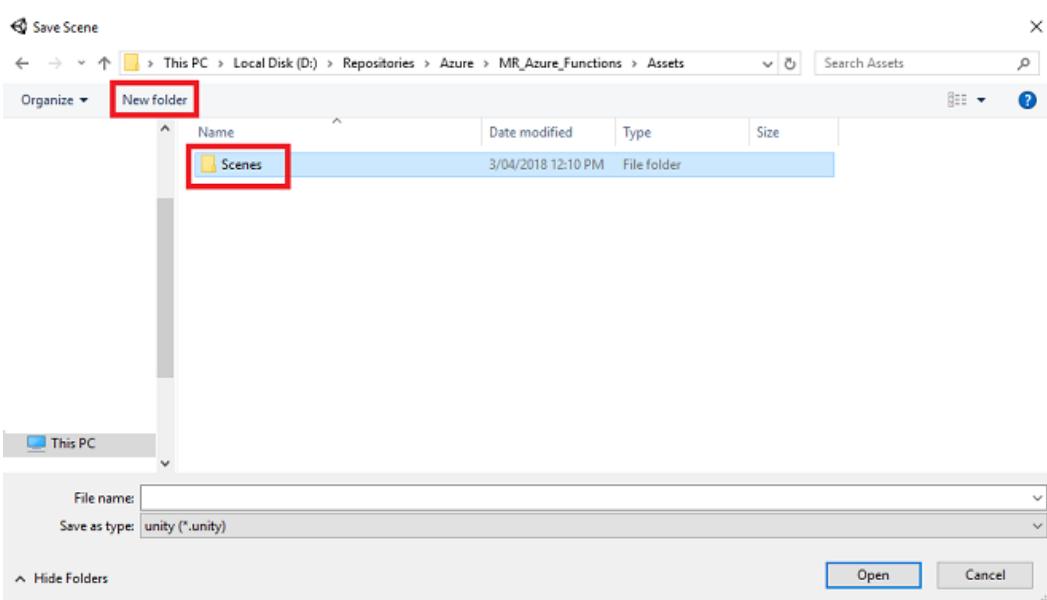
For Microsoft HoloLens, set **Target Device** to *HoloLens*.

- b. **Build Type** is set to **D3D**
- c. **SDK** is set to **Latest installed**
- d. **Visual Studio Version** is set to **Latest installed**
- e. **Build and Run** is set to **Local Machine**
- f. Save the scene and add it to the build.

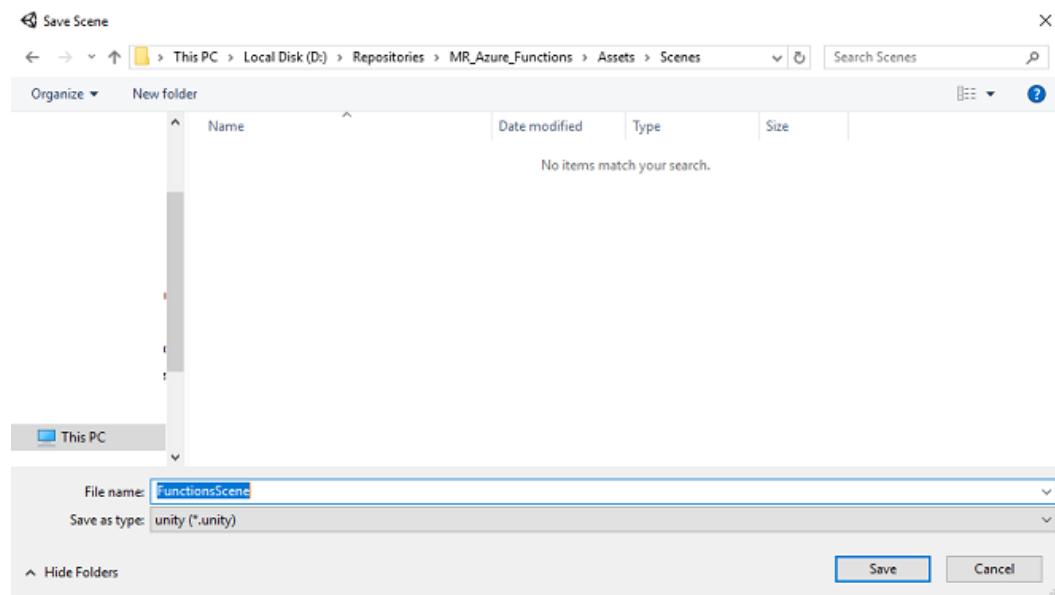
a. Do this by selecting **Add Open Scenes**. A save window will appear.



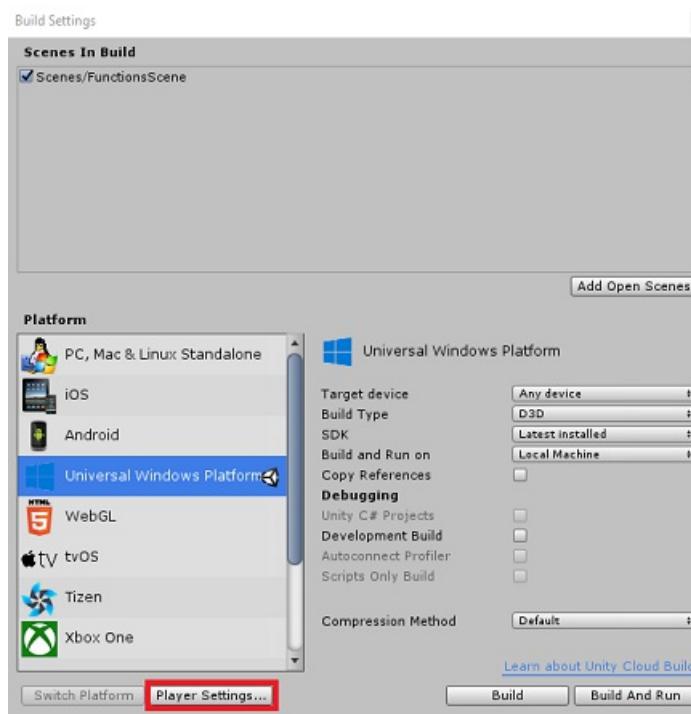
- b. Create a new folder for this, and any future, scene, then select the **New folder** button, to create a new folder, name it **Scenes**.



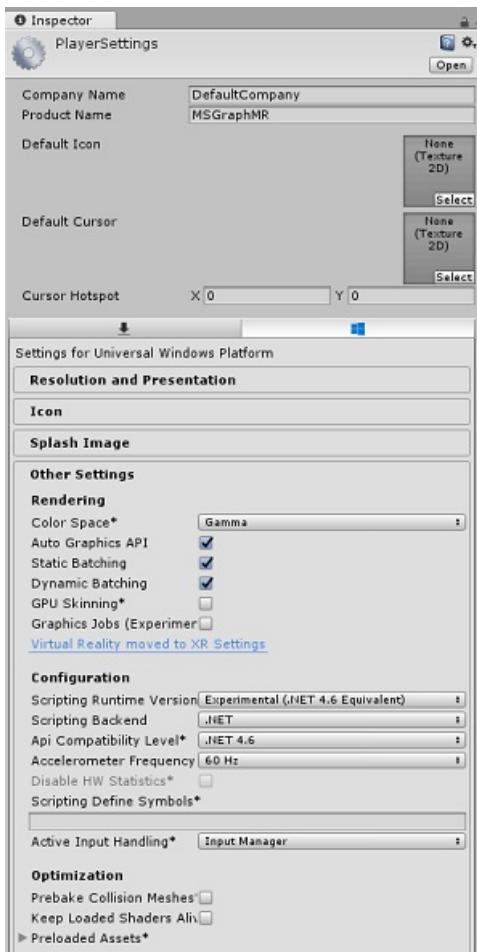
- c. Open your newly created **Scenes** folder, and then in the **File name:** text field, type **FunctionsScene**, then press **Save**.



6. The remaining settings, in **Build Settings**, should be left as default for now.



7. In the *Build Settings* window, click on the **Player Settings** button, this will open the related panel in the space where the *Inspector* is located.



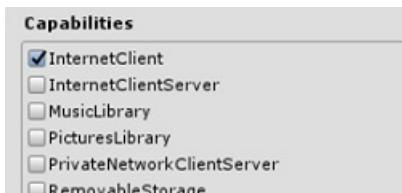
8. In this panel, a few settings need to be verified:

a. In the **Other Settings** tab:

- a. **Scripting Runtime Version** should be **Experimental (.NET 4.6 Equivalent)**, which will trigger a need to restart the Editor.
- b. **Scripting Backend** should be **.NET**
- c. **API Compatibility Level** should be **.NET 4.6**

b. Within the **Publishing Settings** tab, under **Capabilities**, check:

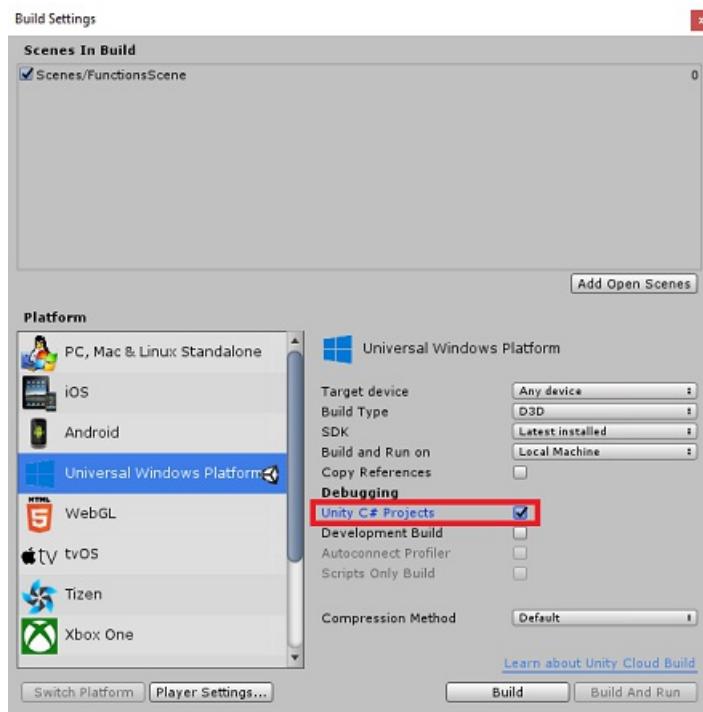
- **InternetClient**



c. Further down the panel, in **XR Settings** (found below **Publishing Settings**), tick **Virtual Reality Supported**, make sure the **Windows Mixed Reality SDK** is added.



9. Back in *Build Settings Unity C# Projects* is no longer greyed out; tick the checkbox next to this.



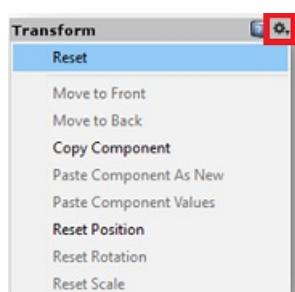
10. Close the Build Settings window.
11. Save your Scene and Project (**FILE > SAVE SCENE / FILE > SAVE PROJECT**).

Chapter 4 - Setup Main Camera

IMPORTANT

If you wish to skip the *Unity Set up* components of this course, and continue straight into code, feel free to [download this .unitypackage](#), and import it into your project as a [Custom Package](#). This will also contain the DLLs from the next Chapter. After import, continue from [Chapter 7](#).

1. In the *Hierarchy Panel*, you will find an object called **Main Camera**, this object represents your "head" point of view once you are "inside" your application.
2. With the Unity Dashboard in front of you, select the **Main Camera GameObject**. You will notice that the *Inspector Panel* (generally found to the right, within the Dashboard) will show the various components of that *GameObject*, with *Transform* at the top, followed by *Camera*, and some other components. You will need to reset the *Transform* of the Main Camera, so it is positioned correctly.
3. To do this, select the **Gear** icon next to the Camera's *Transform* component, and select **Reset**.



4. Then update the **Transform** component to look like:

TRANSFORM - POSITION		
X	Y	Z

TRANSFORM - POSITION

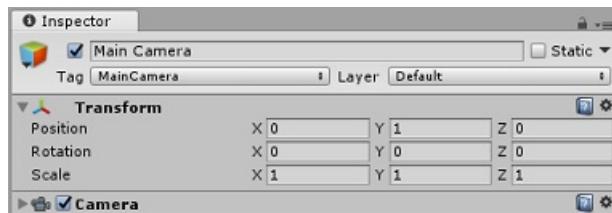
0	1	0
---	---	---

TRANSFORM - ROTATION

X	Y	Z
0	0	0

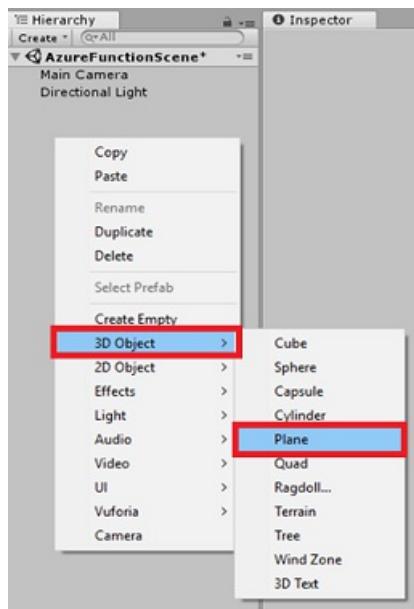
TRANSFORM - SCALE

X	Y	Z
1	1	1



Chapter 5 - Setting up the Unity scene

1. Right-click in an empty area of the *Hierarchy Panel*, under **3D Object**, add a **Plane**.

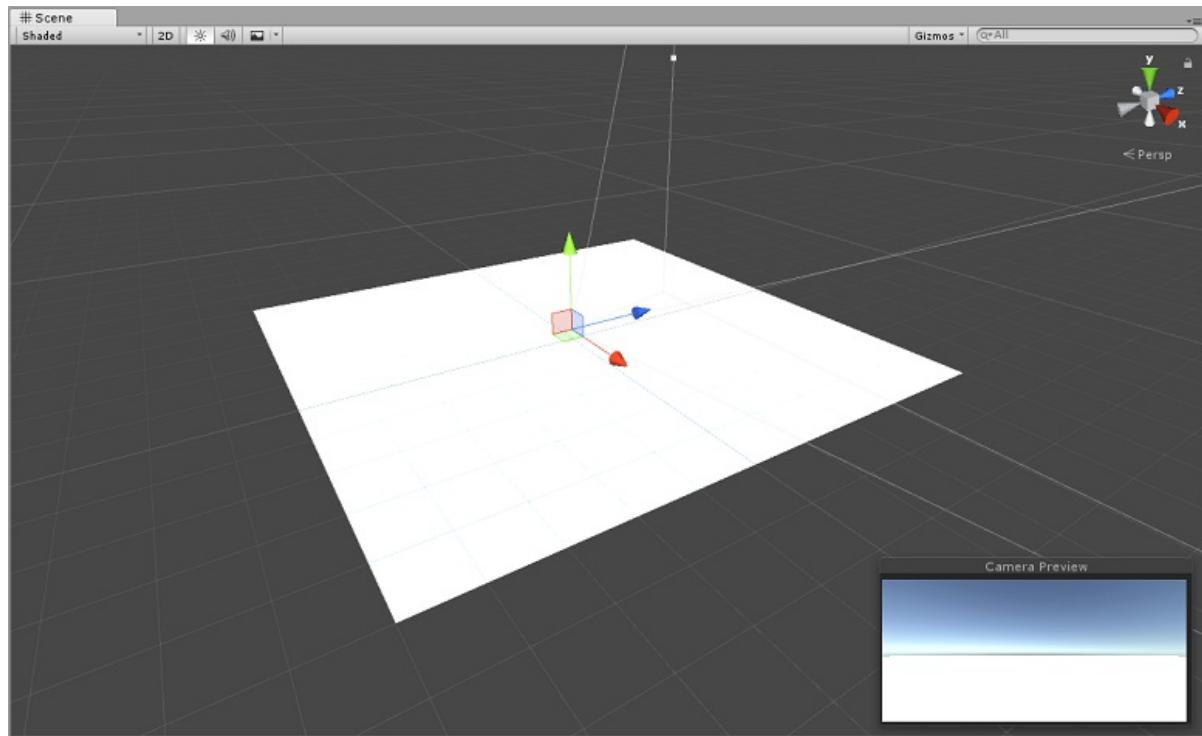
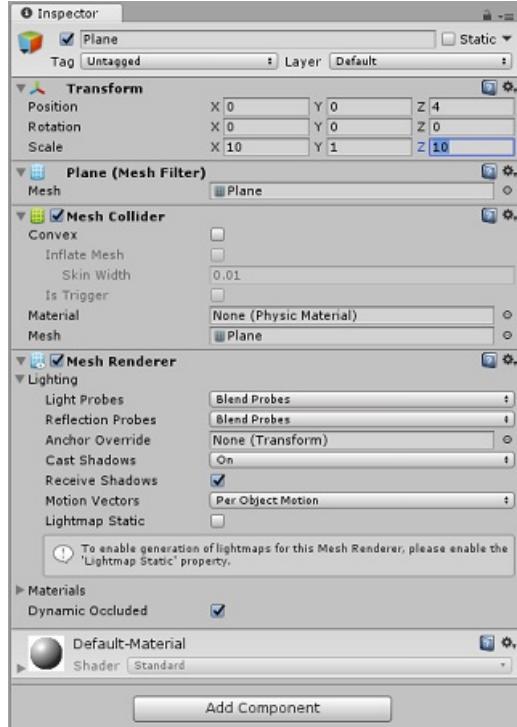


2. With the **Plane** object selected, change the following parameters in the *Inspector Panel*:

TRANSFORM - POSITION

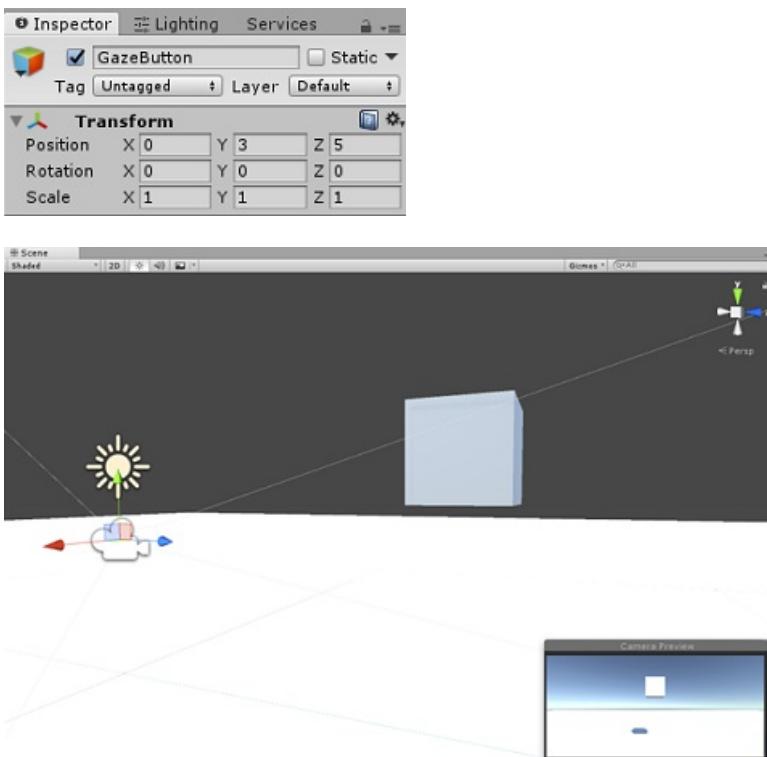
X	Y	Z
0	0	4

TRANSFORM - SCALE		
X	Y	Z
10	1	10

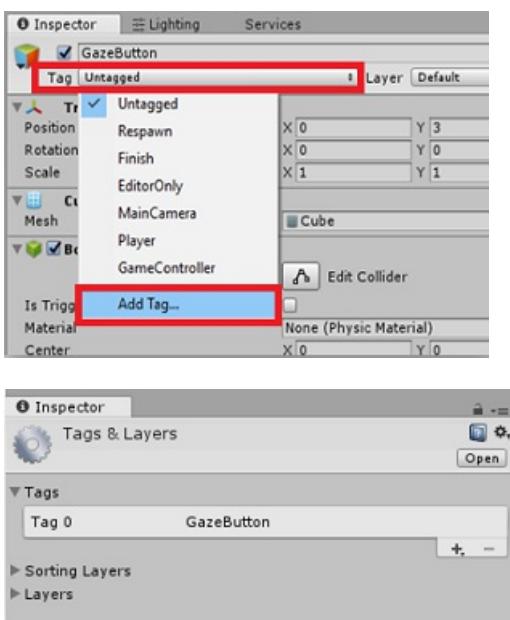


3. Right-click in an empty area of the **Hierarchy Panel**, under **3D Object**, add a **Cube**.
 - a. Rename the Cube to **GazeButton** (with the Cube selected, press 'F2').
 - b. Change the following parameters in the **Inspector Panel**:

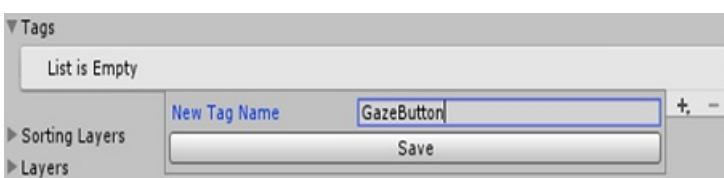
TRANSFORM - POSITION		
X	Y	Z
0	3	5



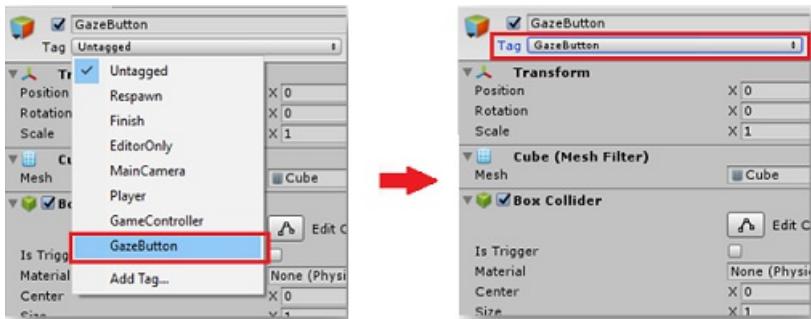
c. Click on the **Tag** drop-down button and click on **Add Tag** to open the *Tags & Layers* pane.



d. Select the **+** (plus) button, and in the *New Tag Name* field, enter **GazeButton**, and press **Save**.

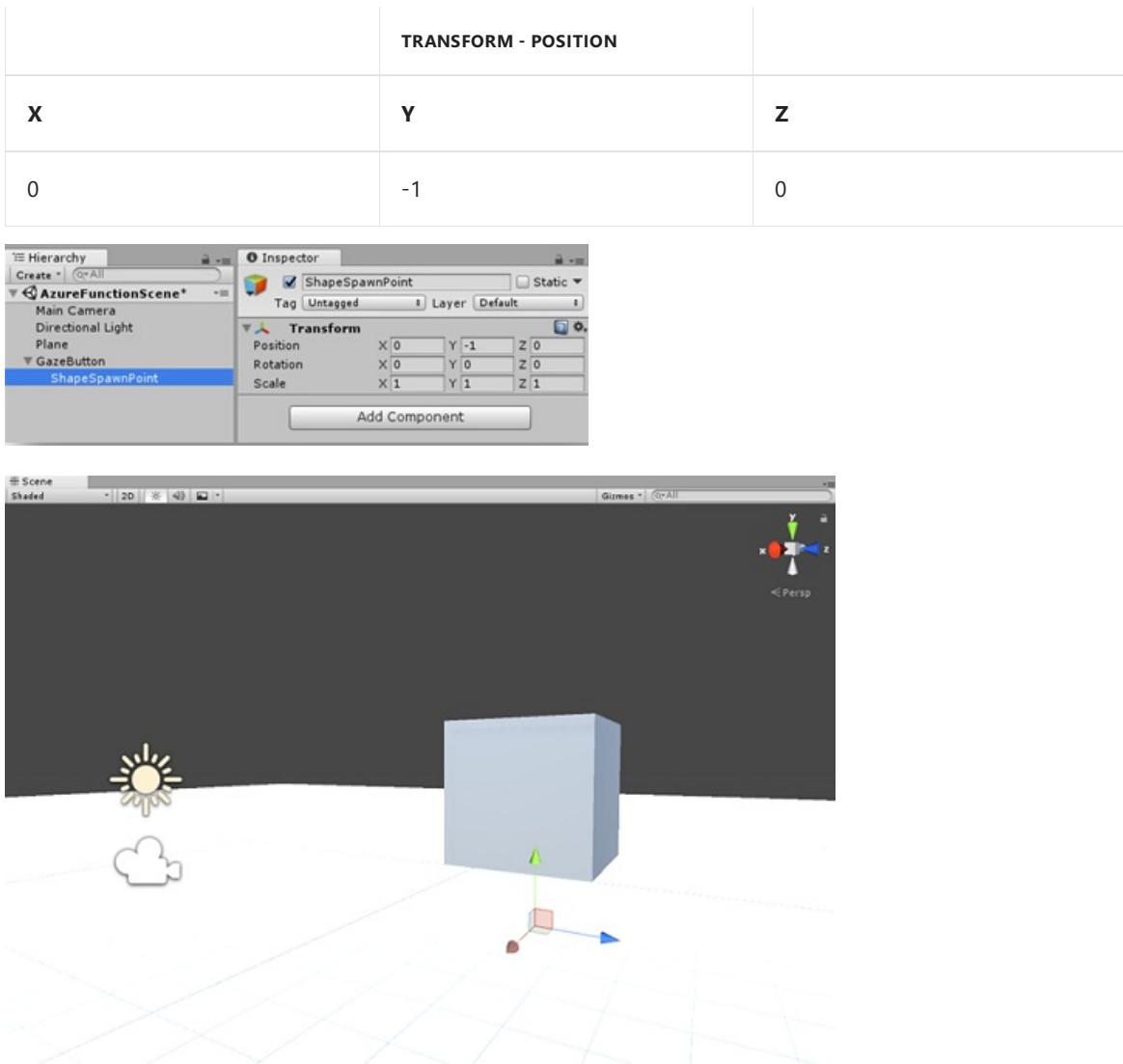


e. Click on the **GazeButton** object in the *Hierarchy Panel*, and in the *Inspector Panel*, assign the newly created **GazeButton** tag.



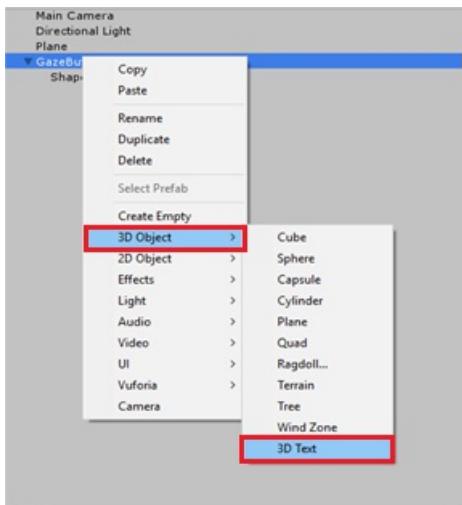
4. Right-click on the **GazeButton** object, in the *Hierarchy Panel*, and add an **Empty GameObject** (which will be added as a *child* object).
5. Select the new object and rename it **ShapeSpawnPoint**.

a. Change the following parameters in the *Inspector Panel*:



6. Next you will create a **3D Text** object to provide feedback on the status of the Azure service.

Right click on the **GazeButton** in the Hierarchy Panel again and add a **3D Object > 3D Text** object as a *child*.



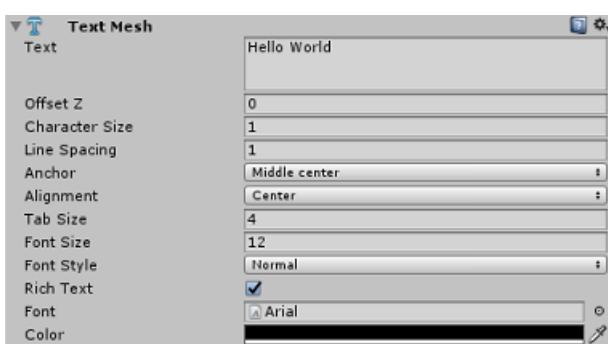
7. Rename the **3D Text** object to **AzureStatusText**.
8. Change the **AzureStatusText** object Transform as follows:

TRANSFORM - POSITION		
X	Y	Z
0	0	-0.6
TRANSFORM - SCALE		
X	Y	Z
0.1	0.1	0.1

NOTE

Do not worry if it appears to be off-centre, as this will be fixed when the below Text Mesh component is updated.

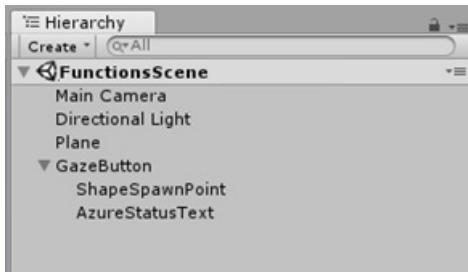
9. Change the **Text Mesh** component to match the below:



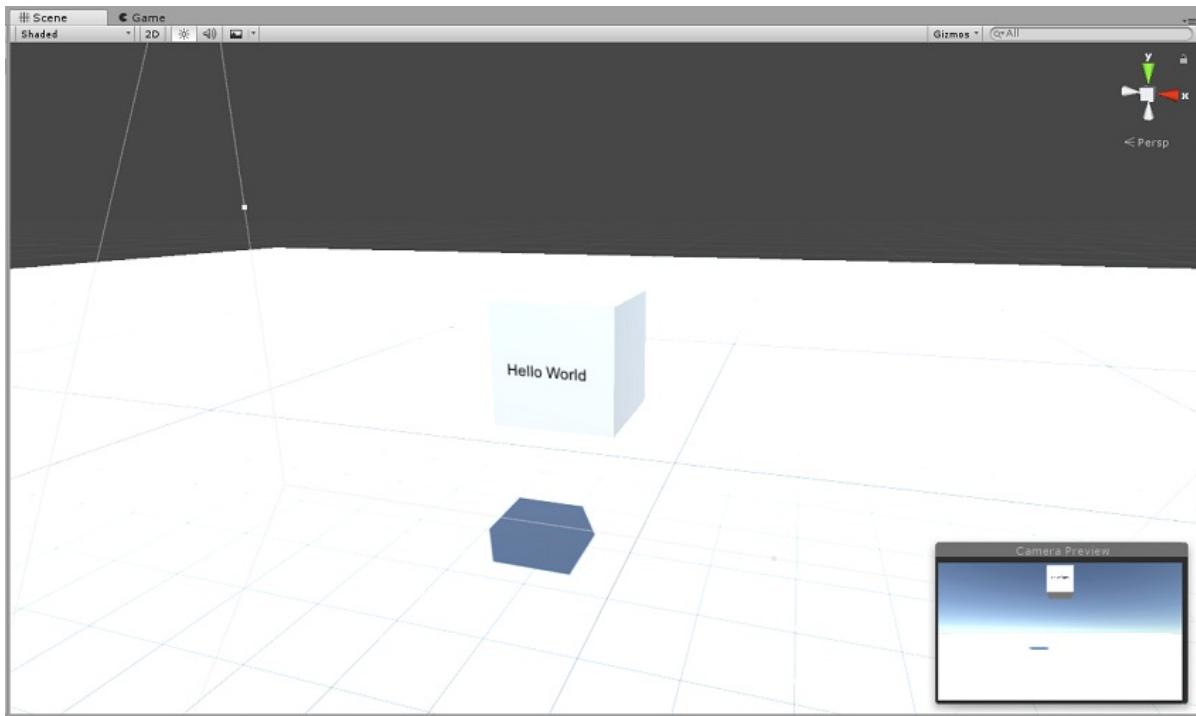
TIP

The selected color here is Hex color: **000000FF**, though feel free to choose your own, just ensure it is readable.

10. Your Hierarchy Panel structure should now look like this:



11. Your scene should now look like this:



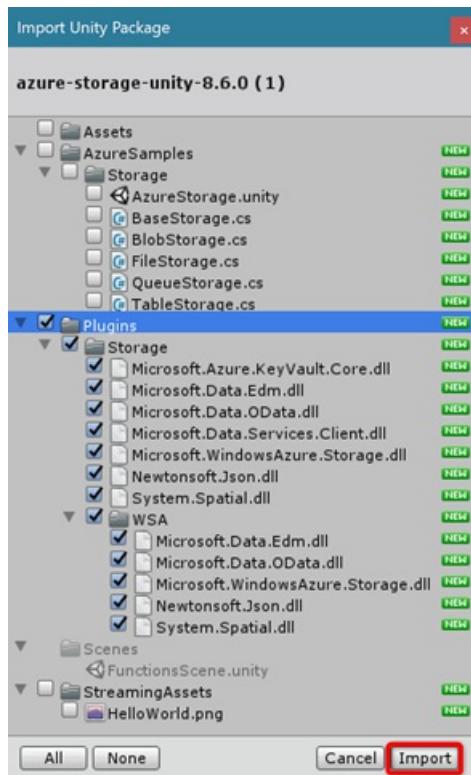
Chapter 6 - Import Azure Storage for Unity

You will be using Azure Storage for Unity (which itself leverages the .Net SDK for Azure). You can read more about this at the [Azure Storage for Unity article](#).

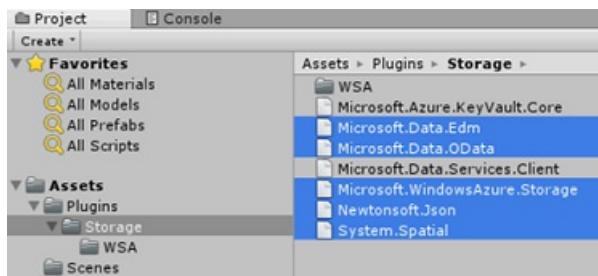
There is currently a known issue in Unity which requires plugins to be reconfigured after import. These steps (4 - 7 in this section) will no longer be required after the bug has been resolved.

To import the SDK into your own project, make sure you have downloaded the latest '[.unitypackage](#)' from [GitHub](#). Then, do the following:

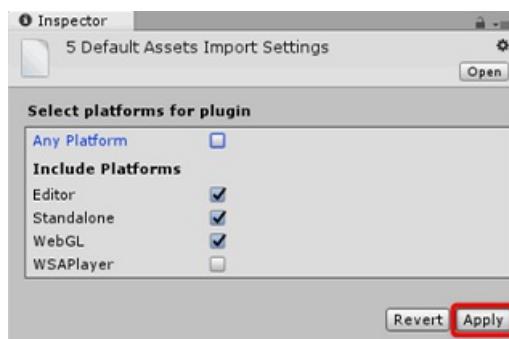
1. Add the **.unitypackage** file to Unity by using the **Assets > Import Package > Custom Package** menu option.
2. In the **Import Unity Package** box that pops up, you can select everything under *Plugin > Storage*. Uncheck everything else, as it is not needed for this course.



3. Click the **Import** button to add the items to your project.
4. Go to the *Storage* folder under *Plugins*, in the Project view, and select the following plugins *only*:
 - Microsoft.Data.Edm
 - Microsoft.Data.OData
 - Microsoft.WindowsAzure.Storage
 - Newtonsoft.Json
 - System.Spatial



5. With *these specific plugins selected*, **uncheck Any Platform** and **uncheck WSAPlayer** then click **Apply**.

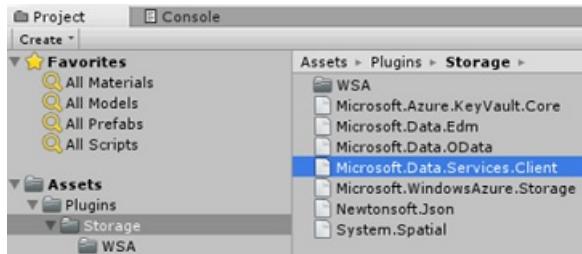


NOTE

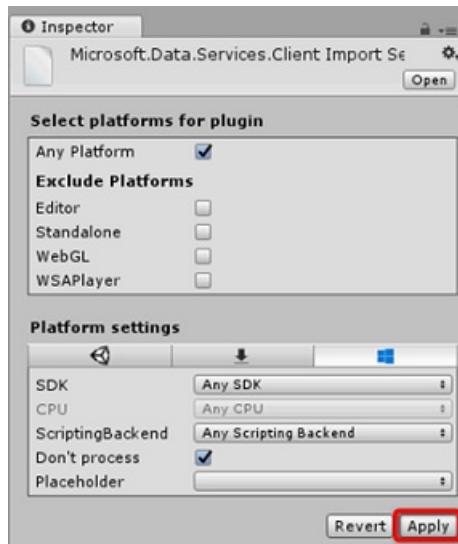
We are marking these particular plugins to only be used in the Unity Editor. This is because there are different versions of the same plugins in the WSA folder that will be used after the project is exported from Unity.

6. In the *Storage* plugin folder, select only:

- Microsoft.Data.Services.Client



7. Check the **Don't Process** box under *Platform Settings* and click **Apply**.



NOTE

We are marking this plugin "Don't process" because the Unity assembly patcher has difficulty processing this plugin. The plugin will still work even though it is not processed.

Chapter 7 - Create the AzureServices class

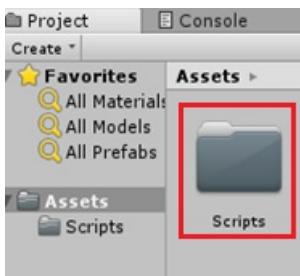
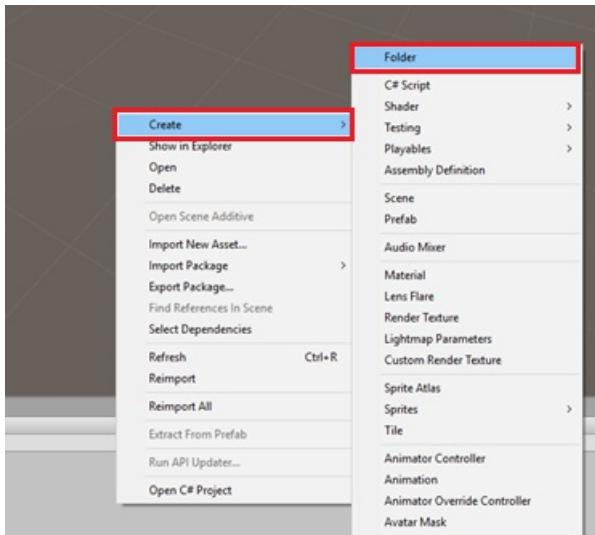
The first class you are going to create is the *AzureServices* class.

The *AzureServices* class will be responsible for:

- Storing Azure Account credentials.
- Calling your Azure App Function.
- The upload and download of the data file in your Azure Cloud Storage.

To create this Class:

1. Right-click in the Asset Folder, located in the Project Panel, **Create > Folder**. Name the folder **Scripts**.



2. Double click on the folder just created, to open it.
3. Right-click inside the folder, **Create > C# Script**. Call the script *AzureServices*.
4. Double click on the new *AzureServices* class to open it with *Visual Studio*.
5. Add the following namespaces to the top of the *AzureServices*:

```
using System;
using System.Threading.Tasks;
using UnityEngine;
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.File;
using System.IO;
using System.Net;
```

6. Add the following Inspector Fields inside the *AzureServices* class:

```
/// <summary>
/// Provides Singleton-like behavior to this class.
/// </summary>
public static AzureServices instance;

/// <summary>
/// Reference Target for AzureStatusText Text Mesh object
/// </summary>
public TextMesh azureStatusText;
```

7. Then add the following member variables inside the *AzureServices* class:

```

/// <summary>
/// Holds the Azure Function endpoint - Insert your Azure Function
/// Connection String here.
/// </summary>

private readonly string azureFunctionEndpoint = "--Insert here you AzureFunction Endpoint--";

/// <summary>
/// Holds the Storage Connection String - Insert your Azure Storage
/// Connection String here.
/// </summary>
private readonly string storageConnectionString = "--Insert here you AzureStorage Connection String--";

/// <summary>
/// Name of the Cloud Share - Hosts directories.
/// </summary>
private const string fileShare = "fileshare";

/// <summary>
/// Name of a Directory within the Share
/// </summary>
private const string storageDirectory = "storagedirectory";

/// <summary>
/// The Cloud File
/// </summary>
private CloudFile shapeIndexCloudFile;

/// <summary>
/// The Linked Storage Account
/// </summary>
private CloudStorageAccount storageAccount;

/// <summary>
/// The Cloud Client
/// </summary>
private CloudFileClient fileClient;

/// <summary>
/// The Cloud Share - Hosts Directories
/// </summary>
private CloudFileShare share;

/// <summary>
/// The Directory in the share that will host the Cloud file
/// </summary>
private CloudFileDirectory dir;

```

IMPORTANT

Make sure you replace the *endpoint* and *connection string* values with the values from your Azure storage, found in the Azure Portal

8. Code for *Awake()* and *Start()* methods now needs to be added. These methods will be called when the class initializes:

```

private void Awake()
{
    instance = this;
}

// Use this for initialization
private void Start()
{
    // Set the Status text to loading, whilst attempting connection to Azure.
    azureStatusText.text = "Loading...";
}

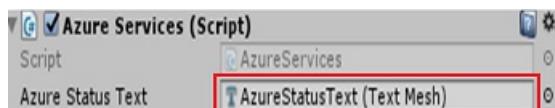
/// <summary>
/// Call to the Azure Function App to request a Shape.
/// </summary>
public async void CallAzureFunctionForNextShape()
{
}

```

IMPORTANT

We will fill in the code for `CallAzureFunctionForNextShape()` in a [future Chapter](#).

9. Delete the `Update()` method since this class will not use it.
10. Save your changes in Visual Studio, and then return to Unity.
11. Click and drag the `AzureServices` class from the Scripts folder to the Main Camera object in the *Hierarchy Panel*.
12. Select the Main Camera, then grab the **AzureStatusText** child object from beneath the **GazeButton** object, and place it within the **AzureStatusText** reference target field, in the *Inspector*, to provide the reference to the `AzureServices` script.



Chapter 8 - Create the ShapeFactory class

The next script to create, is the `ShapeFactory` class. The role of this class is to create a new shape, when requested, and keep a history of the shapes created in a *Shape History List*. Every time a shape is created, the *Shape History list* is updated in the `AzureService` class, and then stored in your *Azure Storage*. When the application starts, if a stored file is found in your *Azure Storage*, the *Shape History list* is retrieved and replayed, with the **3D Text** object providing whether the generated shape is from storage, or new.

To create this class:

1. Go to the **Scripts** folder you created previously.
2. Right-click inside the folder, **Create > C# Script**. Call the script `ShapeFactory`.
3. Double click on the new `ShapeFactory` script to open it with *Visual Studio*.
4. Ensure the `ShapeFactory` class includes the following namespaces:

```
using System.Collections.Generic;
using UnityEngine;
```

5. Add the variables shown below to the *ShapeFactory* class, and replace the *Start()* and *Awake()* functions with those below:

```
/// <summary>
/// Provide this class Singleton-like behaviour
/// </summary>
[HideInInspector]
public static ShapeFactory instance;

/// <summary>
/// Provides an Inspector exposed reference to ShapeSpawnPoint
/// </summary>
[SerializeField]
public Transform spawnPoint;

/// <summary>
/// Shape History Index
/// </summary>
[HideInInspector]
public List<int> shapeHistoryList;

/// <summary>
/// Shapes Enum for selecting required shape
/// </summary>
private enum Shapes { Cube, Sphere, Cylinder }

private void Awake()
{
    instance = this;
}

private void Start()
{
    shapeHistoryList = new List<int>();
}
```

6. The *CreateShape()* method generates the primitive shapes, based upon the provided *integer* parameter. The Boolean parameter is used to specify whether the currently created shape is from storage, or new. Place the following code in your *ShapeFactory* class, below the previous methods:

```

/// <summary>
/// Use the Shape Enum to spawn a new Primitive object in the scene
/// </summary>
/// <param name="shape">Enumerator Number for Shape</param>
/// <param name="storageShape">Provides whether this is new or old</param>
internal void CreateShape(int shape, bool storageSpace)
{
    Shapes primitive = (Shapes)shape;
    GameObject newObjet = null;
    string shapeText = storageSpace == true ? "Storage: " : "New: ";

    AzureServices.instance.azureStatusText.text = string.Format("{0}{1}", shapeText,
primitive.ToString());

    switch (primitive)
    {
        case Shapes.Cube:
            newObjet = GameObject.CreatePrimitive(PrimitiveType.Cube);
            break;

        case Shapes.Sphere:
            newObjet = GameObject.CreatePrimitive(PrimitiveType.Sphere);
            break;

        case Shapes.Cylinder:
            newObjet = GameObject.CreatePrimitive(PrimitiveType.Cylinder);
            break;
    }

    if (newObjet != null)
    {
        newObjet.transform.position = spawnPoint.position;

        newObjet.transform.localScale = new Vector3(0.5f, 0.5f, 0.5f);

        newObjet.AddComponent<Rigidbody>().useGravity = true;

        newObjet.GetComponent<Renderer>().material.color = UnityEngine.Random.ColorHSV(0f, 1f, 1f,
1f, 0.5f, 1f);
    }
}

```

7. Be sure to save your changes in Visual Studio before returning to Unity.
8. Back in the Unity Editor, click and drag the *ShapeFactory* class from the **Scripts** folder to the **Main Camera** object in the *Hierarchy Panel*.
9. With the Main Camera selected you will notice the *ShapeFactory* script component is missing the *Spawn Point* reference. To fix it, drag the **ShapeSpawnPoint** object from the *Hierarchy Panel* to the **Spawn Point** reference target.



Chapter 9 - Create the Gaze class

The last script you need to create is the *Gaze* class.

This class is responsible for creating a **Raycast** that will be projected forward from the Main Camera, to detect which object the user is looking at. In this case, the Raycast will need to identify if the user is looking at the **GazeButton** object in the scene and trigger a behavior.

To create this Class:

1. Go to the **Scripts** folder you created previously.
2. Right-click in the Project Panel, **Create > C# Script**. Call the script *Gaze*.
3. Double click on the new *Gaze* script to open it with *Visual Studio*.
4. Ensure the following namespace is included at the top of the script:

```
using UnityEngine;
```

5. Then add the following variables inside the *Gaze* class:

```
/// <summary>
/// Provides Singleton-like behavior to this class.
/// </summary>
public static Gaze instance;

/// <summary>
/// The Tag which the Gaze will use to interact with objects. Can also be set in editor.
/// </summary>
public string InteractableTag = "GazeButton";

/// <summary>
/// The layer which will be detected by the Gaze ('~0' equals everything).
/// </summary>
public LayerMask LayerMask = ~0;

/// <summary>
/// The Max Distance the gaze should travel, if it has not hit anything.
/// </summary>
public float GazeMaxDistance = 300;

/// <summary>
/// The size of the cursor, which will be created.
/// </summary>
public Vector3 CursorSize = new Vector3(0.05f, 0.05f, 0.05f);

/// <summary>
/// The color of the cursor - can be set in editor.
/// </summary>
public Color CursorColour = Color.HSVToRGB(0.0223f, 0.7922f, 1.000f);

/// <summary>
/// Provides when the gaze is ready to start working (based upon whether
/// Azure connects successfully).
/// </summary>
internal bool GazeEnabled = false;

/// <summary>
/// The currently focused object.
/// </summary>
internal GameObject FocusedObject { get; private set; }

/// <summary>
/// The object which was last focused on.
/// </summary>
internal GameObject _oldFocusedObject { get; private set; }

/// <summary>
/// The info taken from the last hit.
/// </summary>
internal RaycastHit HitInfo { get; private set; }

/// <summary>
```

```

    ///> 
    /// The cursor object.
    /// </summary>
    internal GameObject Cursor { get; private set; }

    /// <summary>
    /// Provides whether the raycast has hit something.
    /// </summary>
    internal bool Hit { get; private set; }

    /// <summary>
    /// This will store the position which the ray last hit.
    /// </summary>
    internal Vector3 Position { get; private set; }

    /// <summary>
    /// This will store the normal, of the ray from its last hit.
    /// </summary>
    internal Vector3 Normal { get; private set; }

    /// <summary>
    /// The start point of the gaze ray cast.
    /// </summary>
    private Vector3 _gazeOrigin;

    /// <summary>
    /// The direction in which the gaze should be.
    /// </summary>
    private Vector3 _gazeDirection;

```

IMPORTANT

Some of these variables will be able to be edited in the *Editor*.

6. Code for the *Awake()* and *Start()* methods now needs to be added.

```

    /// <summary>
    /// The method used after initialization of the scene, though before Start().
    /// </summary>
    private void Awake()
    {
        // Set this class to behave similar to singleton
        instance = this;
    }

    /// <summary>
    /// Start method used upon initialization.
    /// </summary>
    private void Start()
    {
        FocusedObject = null;
        Cursor = CreateCursor();
    }

```

7. Add the following code, which will create a cursor object at start, along with the *Update()* method, which will run the Raycast method, along with being where the GazeEnabled boolean is toggled:

```

/// <summary>
/// Method to create a cursor object.
/// </summary>
/// <returns></returns>
private GameObject CreateCursor()
{
    GameObject newCursor = GameObject.CreatePrimitive(PrimitiveType.Sphere);
    newCursor.SetActive(false);

    // Remove the collider, so it doesn't block raycast.
    Destroy(newCursor.GetComponent<SphereCollider>());
    newCursor.transform.localScale = CursorSize;

    newCursor.GetComponent<MeshRenderer>().material = new Material(Shader.Find("Diffuse"))
    {
        color = CursorColour
    };

    newCursor.name = "Cursor";

    newCursor.SetActive(true);

    return newCursor;
}

/// <summary>
/// Called every frame
/// </summary>
private void Update()
{
    if(GazeEnabled == true)
    {
        _gazeOrigin = Camera.main.transform.position;

        _gazeDirection = Camera.main.transform.forward;

        UpdateRaycast();
    }
}

```

8. Next add the *UpdateRaycast()* method, which will project a Raycast and detect the hit target.

```

private void UpdateRaycast()
{
    // Set the old focused gameobject.
    _oldFocusedObject = FocusedObject;

    RaycastHit hitInfo;

    // Initialise Raycasting.
    Hit = Physics.Raycast(_gazeOrigin,
        _gazeDirection,
        out hitInfo,
        GazeMaxDistance, LayerMask);

    HitInfo = hitInfo;

    // Check whether raycast has hit.
    if (Hit == true)
    {
        Position = hitInfo.point;

        Normal = hitInfo.normal;

        // Check whether the hit has a collider.
        if (hitInfo.collider != null)

```

```

    {
        // Set the focused object with what the user just looked at.
        FocusedObject = hitInfo.collider.gameObject;
    }
    else
    {
        // Object looked on is not valid, set focused gameobject to null.
        FocusedObject = null;
    }
}
else
{
    // No object looked upon, set focused gameobject to null.
    FocusedObject = null;

    // Provide default position for cursor.
    Position = _gazeOrigin + (_gazeDirection * GazeMaxDistance);

    // Provide a default normal.
    Normal = _gazeDirection;
}

// Lerp the cursor to the given position, which helps to stabilize the gaze.
Cursor.transform.position = Vector3.Lerp(Cursor.transform.position, Position, 0.6f);

// Check whether the previous focused object is this same
// object. If so, reset the focused object.
if (FocusedObject != _oldFocusedObject)
{
    ResetFocusedObject();

    if (FocusedObject != null)
    {
        if (FocusedObject.CompareTag(InteractableTag.ToString()))
        {
            // Set the Focused object to green - success!
            FocusedObject.GetComponent<Renderer>().material.color = Color.green;

            // Start the Azure Function, to provide the next shape!
            AzureServices.instance.CallAzureFunctionForNextShape();
        }
    }
}
}
}

```

9. Lastly, add the *ResetFocusedObject()* method, which will toggle the GazeButton objects current color, indicating whether it is creating a new shape or not.

```

/// <summary>
/// Reset the old focused object, stop the gaze timer, and send data if it
/// is greater than one.
/// </summary>
private void ResetFocusedObject()
{
    // Ensure the old focused object is not null.
    if (_oldFocusedObject != null)
    {
        if (_oldFocusedObject.CompareTag(InteractableTag.ToString()))
        {
            // Set the old focused object to red - its original state.
            _oldFocusedObject.GetComponent<Renderer>().material.color = Color.red;
        }
    }
}

```

10. Save your changes in Visual Studio before returning to Unity.
11. Click and drag the *Gaze* class from the Scripts folder to the **Main Camera** object in the *Hierarchy Panel*.

Chapter 10 - Completing the AzureServices class

With the other scripts in place, it is now possible to *complete* the *AzureServices* class. This will be achieved through:

1. Adding a new method named *CreateCloudIdentityAsync()*, to set up the authentication variables needed for communicating with Azure.

This method will also check for the existence of a previously stored File containing the Shape List.

If the file is found, it will disable the user *Gaze*, and trigger Shape creation, according to the pattern of shapes, as stored in the **Azure Storage file**. The user can see this, as the **Text Mesh** will provide display 'Storage' or 'New', depending on the shapes origin.

If no file is found, it will enable the *Gaze*, enabling the user to create shapes when looking at the **GazeButton** object in the scene.

```

/// <summary>
/// Create the references necessary to log into Azure
/// </summary>
private async void CreateCloudIdentityAsync()
{
    // Retrieve storage account information from connection string
    storageAccount = CloudStorageAccount.Parse(storageConnectionString);

    // Create a file client for interacting with the file service.
    fileClient = storageAccount.CreateCloudFileClient();

    // Create a share for organizing files and directories within the storage account.
    share = fileClient.GetShareReference(fileShare);

    await share.CreateIfNotExistsAsync();

    // Get a reference to the root directory of the share.
    CloudFileDirectory root = share.GetRootDirectoryReference();

    // Create a directory under the root directory
    dir = root.GetDirectoryReference(storageDirectory);

    await dir.CreateIfNotExistsAsync();

    //Check if there is a stored text file containing the list
    shapeIndexCloudFile = dir.GetFileReference("TextShapeFile");

    if (!await shapeIndexCloudFile.ExistsAsync())
    {
        // File not found, enable gaze for shapes creation
        Gaze.instance.GazeEnabled = true;

        azureStatusText.text = "No Shape\nFile!";
    }
    else
    {
        // The file has been found, disable gaze and get the list from the file
        Gaze.instance.GazeEnabled = false;

        azureStatusText.text = "Shape File\nFound!";

        await ReplicateListFromAzureAsync();
    }
}

```

2. The next code snippet is from within the *Start()* method; wherein a call will be made to the *CreateCloudIdentityAsync()* method. Feel free to copy over your current *Start()* method, with the below:

```

private void Start()
{
    // Disable TLS cert checks only while in Unity Editor (until Unity adds support for TLS)
#if UNITY_EDITOR
    ServicePointManager.ServerCertificateValidationCallback = delegate { return true; };
#endif

    // Set the Status text to loading, whilst attempting connection to Azure.
    azureStatusText.text = "Loading...";

    //Creating the references necessary to log into Azure and check if the Storage Directory is
    empty
    CreateCloudIdentityAsync();
}

```

3. Fill in the code for the method *CallAzureFunctionForNextShape()*. You will use the previously created *Azure*

Function App to request a shape index. Once the new shape is received, this method will send the shape to the *ShapeFactory* class to create the new shape in the scene. Use the code below to complete the body of *CallAzureFunctionForNextShape()*.

```
/// <summary>
/// Call to the Azure Function App to request a Shape.
/// </summary>
public async void CallAzureFunctionForNextShape()
{
    int azureRandomInt = 0;

    // Call Azure function
    HttpWebRequest webRequest = WebRequest.CreateHttp(azureFunctionEndpoint);

    WebResponse response = await webRequest.GetResponseAsync();

    // Read response as string
    using (Stream stream = response.GetResponseStream())
    {
        StreamReader reader = new StreamReader(stream);

        String responseString = reader.ReadToEnd();

        //parse result as integer
        Int32.TryParse(responseString, out azureRandomInt);
    }

    //add random int from Azure to the ShapeIndexList
    ShapeFactory.instance.shapeHistoryList.Add(azureRandomInt);

    ShapeFactory.instance.CreateShape(azureRandomInt, false);

    //Save to Azure storage
    await UploadListToAzureAsync();
}
```

4. Add a method to create a string, by concatenating the integers stored in the shape history list, and saving it in your *Azure Storage File*.

```
/// <summary>
/// Upload the locally stored List to Azure
/// </summary>
private async Task UploadListToAzureAsync()
{
    // Uploading a local file to the directory created above
    string listToString = string.Join(", ", ShapeFactory.instance.shapeHistoryList.ToArray());

    await shapeIndexCloudFile.UploadTextAsync(listToString);
}
```

5. Add a method to retrieve the text stored in the file located in your *Azure Storage File* and *deserialize* it into a list.
6. Once this process is completed, the method re-enables the gaze so that the user can add more shapes to the scene.

```

///<summary>
/// Get the List stored in Azure and use the data retrieved to replicate
/// a Shape creation pattern
///</summary>
private async Task ReplicateListFromAzureAsync()
{
    string azureTextFileContent = await shapeIndexCloudFile.DownloadTextAsync();

    string[] shapes = azureTextFileContent.Split(new char[] { ',' });

    foreach (string shape in shapes)
    {
        int i;

        Int32.TryParse(shape.ToString(), out i);

        ShapeFactory.instance.shapeHistoryList.Add(i);

        ShapeFactory.instance.CreateShape(i, true);

        await Task.Delay(500);
    }

    Gaze.instance.GazeEnabled = true;

    azureStatusText.text = "Load Complete!";
}

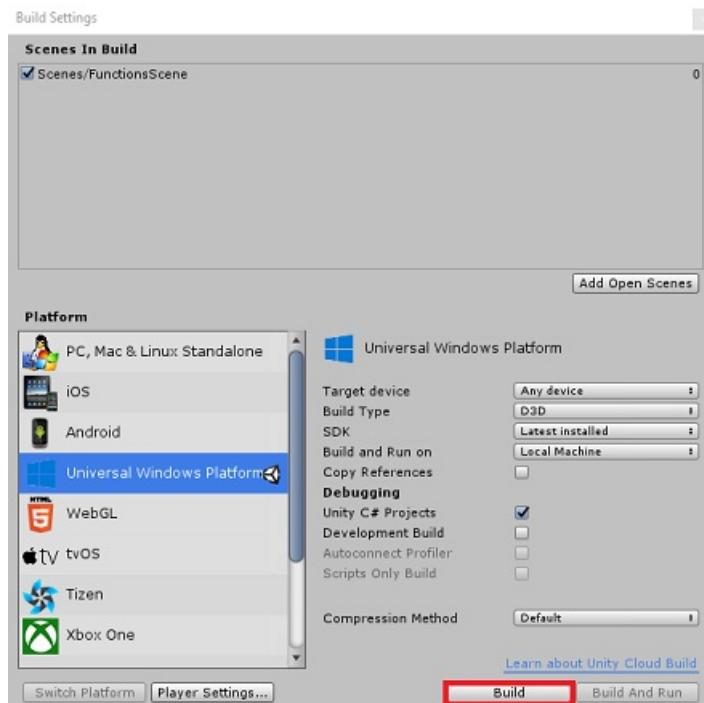
```

- Save your changes in Visual Studio before returning to Unity.

Chapter 11 - Build the UWP Solution

To begin the Build process:

- Go to **File > Build Settings**.



- Click **Build**. Unity will launch a *File Explorer* window, where you need to create and then select a folder to build the app into. Create that folder now, and name it *App*. Then with the *App* folder selected, press **Select Folder**.

3. Unity will begin building your project to the *App* folder.
4. Once Unity has finished building (it might take some time), it will open a *File Explorer* window at the location of your build (check your task bar, as it may not always appear above your windows, but will notify you of the addition of a new window).

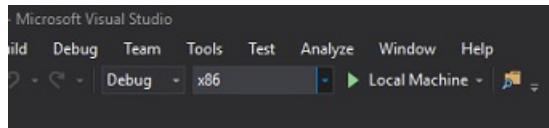
Chapter 12 - Deploying your application

To deploy your application:

1. Navigate to the *App* folder which was created in the [last Chapter](#). You will see a file with your apps name, with the '.sln' extension, which you should double-click, so to open it within *Visual Studio*.
2. In the **Solution Platform**, select **x86, Local Machine**.
3. In the **Solution Configuration** select **Debug**.

For the Microsoft HoloLens, you may find it easier to set this to *Remote Machine*, so that you are not tethered to your computer. Though, you will need to also do the following:

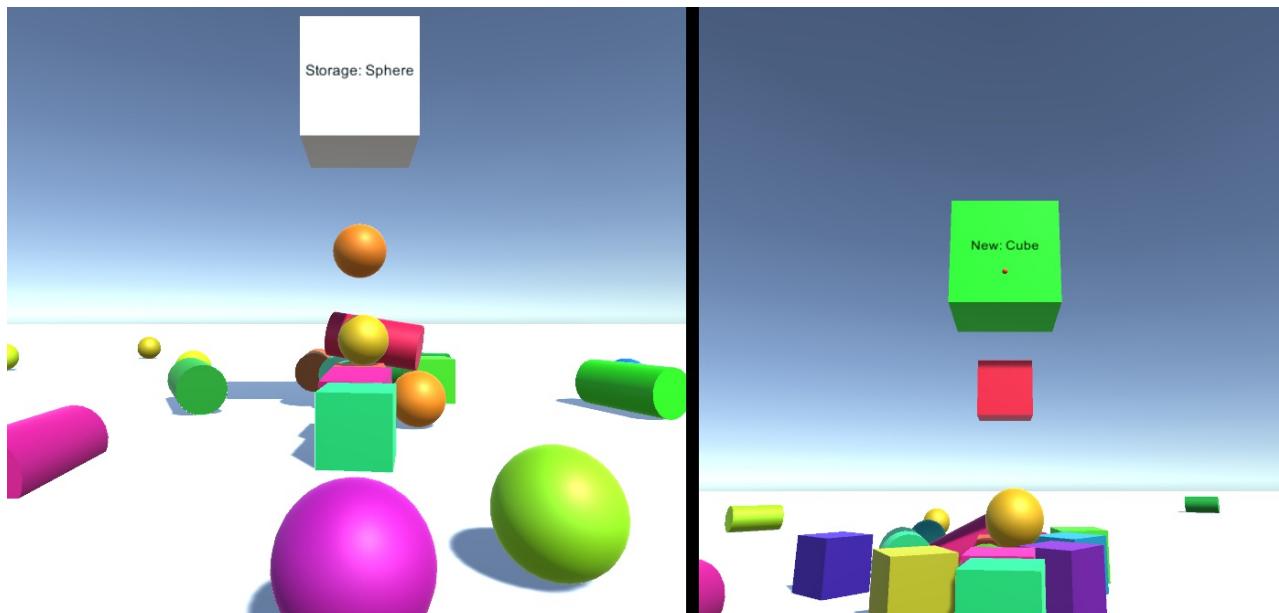
- Know the **IP Address** of your HoloLens, which can be found within the *Settings > Network & Internet > Wi-Fi > Advanced Options*; the IPv4 is the address you should use.
- Ensure **Developer Mode** is **On**; found in *Settings > Update & Security > For developers*.



4. Go to the **Build** menu and click on **Deploy Solution** to sideload the application to your machine.
5. Your App should now appear in the list of installed apps, ready to be launched and tested!

Your finished Azure Functions and Storage Application

Congratulations, you built a mixed reality app that leverages both the Azure Functions and Azure Storage services. Your app will be able to draw on stored data, and provide an action based on that data.



Bonus exercises

Exercise 1

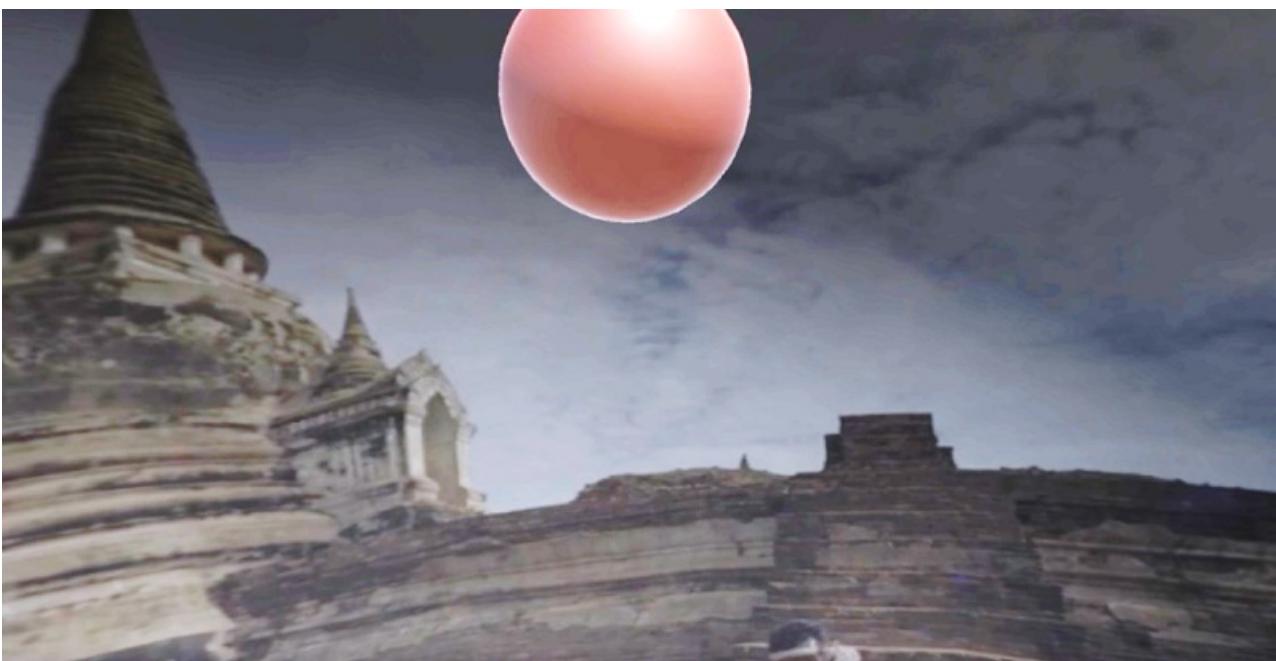
Create a second spawn point and record which spawn point an object was created from. When you load the data file, replay the shapes being spawned from the location they originally were created.

Exercise 2

Create a way to restart the app, rather than having to re-open it each time. **Loading Scenes** is a good spot to start. After doing that, create a way to clear the stored list in *Azure Storage*, so that it can be easily reset from your app.

MR and Azure 306: Streaming video

11/6/2018 • 23 minutes to read • [Edit Online](#)



In this course you will learn how to connect your Azure Media Services to a Windows Mixed Reality VR experience to allow streaming 360 degree video playback on immersive headsets.

Azure Media Services are a collection of services that give you broadcast-quality video streaming services to reach larger audiences on today's most popular mobile devices. For more information, visit the [Azure Media Services page](#).

Having completed this course, you will have a mixed reality immersive headset application, which will be able to do the following:

1. Retrieve a 360 degree video from an **Azure Storage**, through the **Azure Media Service**.
2. Display the retrieved 360 degree video within a Unity scene.

3. Navigate between two scenes, with two different videos.

In your application, it is up to you as to how you will integrate the results with your design. This course is designed to teach you how to integrate an Azure Service with your Unity Project. It is your job to use the knowledge you gain from this course to enhance your mixed reality application.

Device support

COURSE	HOLOLENS	IMMERSIVE HEADSETS
MR and Azure 306: Streaming video		✓ <input type="checkbox"/>

Prerequisites

NOTE

This tutorial is designed for developers who have basic experience with Unity and C#. Please also be aware that the prerequisites and written instructions within this document represent what has been tested and verified at the time of writing (May 2018). You are free to use the latest software, as listed within the [install the tools article](#), though it should not be assumed that the information in this course will perfectly match what you'll find in newer software than what's listed below.

We recommend the following hardware and software for this course:

- A development PC, [compatible with Windows Mixed Reality](#) for immersive (VR) headset development
- [Windows 10 Fall Creators Update \(or later\)](#) with [Developer mode enabled](#)
- [The latest Windows 10 SDK](#)
- [Unity 2017.4](#)
- [Visual Studio 2017](#)
- A [Windows Mixed Reality immersive \(VR\) headset](#)
- Internet access for Azure setup and data retrieval
- Two 360-degree videos in mp4 format (you can find some royalty-free videos [at this download page](#))

Before you start

1. To avoid encountering issues building this project, it is strongly suggested that you create the project mentioned in this tutorial in a root or near-root folder (long folder paths can cause issues at build-time).
2. Set up and test your Mixed Reality Immersive Headset.

NOTE

You will **not** require Motion Controllers for this course. If you need support setting up the Immersive Headset, please click [link on how to set up Windows Mixed Reality](#).

Chapter 1 - The Azure Portal: creating the Azure Storage Account

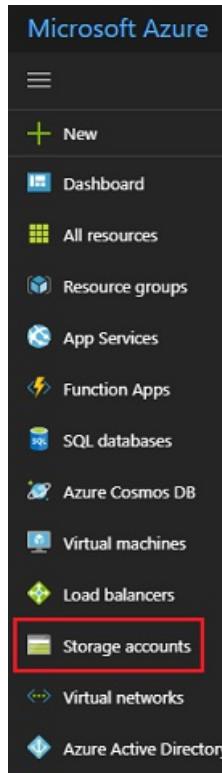
To use the **Azure Storage Service**, you will need to create and configure a **Storage Account** in the Azure portal.

1. Log in to the [Azure Portal](#).

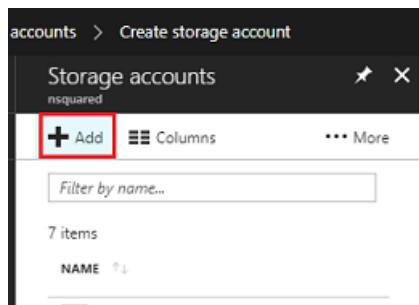
NOTE

If you do not already have an Azure account, you will need to create one. If you are following this tutorial in a classroom or lab situation, ask your instructor or one of the proctors for help setting up your new account.

- Once you are logged in, click on **Storage accounts** in the left menu.



- On the **Storage Accounts** tab, click on **Add**.



- In the **Create storage account** tab:

- Insert a **Name** for your account, be aware this field only accepts numbers, and lowercase letters.
- For **Deployment model**, select **Resource manager**.
- For **Account kind**, select **Storage (general purpose v1)**.
- For **Performance**, select **Standard.***
- For **Replication** select **Locally-redundant storage (LRS)**.
- Leave **Secure transfer required** as **Disabled**.
- Select a **Subscription**.
- Choose a **Resource group** or create a new one. A resource group provides a way to monitor, control access, provision and manage billing for a collection of Azure assets.

- i. Determine the **Location** for your resource group (if you are creating a new Resource Group). The location would ideally be in the region where the application would run. Some Azure assets are only available in certain regions.
5. You will need to confirm that you have understood the Terms and Conditions applied to this Service.

The cost of your storage account depends on the usage and the options you choose below.
[Learn more](#)

Name .core.windows.net

Deployment model Resource manager Classic

Account kind

Performance Standard Premium

Replication

Secure transfer required Enabled Disabled

Subscription

Resource group Create new Use existing
Azure_for_Unity

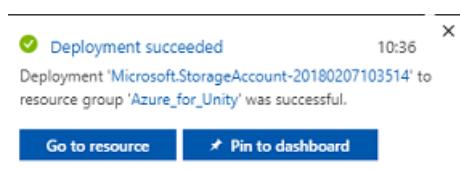
Location

Virtual networks
Configure virtual networks Enabled

Pin to dashboard

Create [Automation options](#)

6. Once you have clicked on **Create**, you will have to wait for the service to be created, this might take a minute.
7. A notification will appear in the portal once the Service instance is created.



8. At this point you do not need to follow the resource, simply move to the next Chapter.

Chapter 2 - The Azure Portal: creating the Media Service

To use the Azure Media Service, you will need to configure an instance of the service to be made available to your application (wherein the account holder needs to be an Admin).

- In the Azure Portal, click on **Create a resource** in the top left corner, and search for **Media Service**, press **Enter**. The resource you want currently has a pink icon; click this, to show a new page.

The screenshot shows the Azure Marketplace search results for 'Media Service'. On the left, there's a sidebar with various service categories like All services, Favorites, Dashboard, etc. The main area shows a list of results with columns for Name, Publisher, and Category. The 'Media Services' item by Microsoft is highlighted with a red box. To the right, there's a detailed description of Media Services, including its purpose (building workflows for creation, management, and distribution of media) and its benefits (flexibility, scalability, reliability). At the bottom right of this panel is a prominent blue 'Create' button, also highlighted with a red box.

- The new page will provide a description of the **Media Service**. At the bottom left of this prompt, click the **Create** button, to create an association with this service.

This screenshot is identical to the one above, showing the Azure Marketplace search results for 'Media Service'. The 'Media Services' item by Microsoft is highlighted with a red box. The right pane contains the same detailed description of Media Services and features a blue 'Create' button at the bottom right, which is also highlighted with a red box.

- Once you have clicked on **Create** a panel will appear where you need to provide some details about your new Media Service:

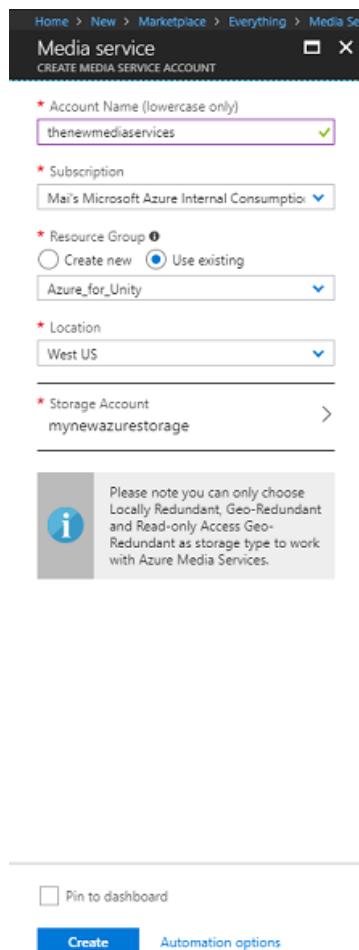
- Insert your desired **Account Name** for this service instance.
- Select a **Subscription**.
- Choose a **Resource Group** or create a new one. A resource group provides a way to monitor, control access, provision and manage billing for a collection of Azure assets. It is recommended to keep all the Azure services associated with a single project (e.g. such as these labs) under a common resource group).

If you wish to read more about Azure Resource Groups, please follow this [link on how to manage Azure Resource Groups](#).

- Determine the **Location** for your resource group (if you are creating a new Resource Group). The

location would ideally be in the region where the application would run. Some Azure assets are only available in certain regions.

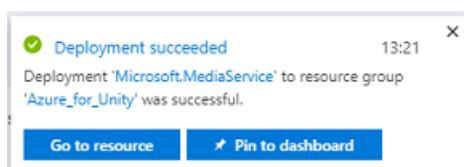
- e. For the **Storage Account** section, click the **Please select...** section, then click the **Storage Account** you created in the last Chapter.
- f. You will also need to confirm that you have understood the Terms and Conditions applied to this Service.
- g. Click **Create**.



4. Once you have clicked on **Create**, you will have to wait for the service to be created, this might take a minute.
5. A notification will appear in the portal once the Service instance is created.



6. Click on the notification to explore your new Service instance.



7. Click the **Go to resource** button in the notification to explore your new Service instance.
8. Within the new Media service page, within the panel on the left, click on the **Assets** link, which is about halfway down.

9. On the next page, in the top-left corner of the page, click **Upload**.

The screenshot shows the 'Assets' blade in the Azure Media Services portal. The left sidebar contains navigation links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings (Locks, Automation script), Media Services (Properties, API access, Assets, Content protection, Jobs, Live streaming, Media Reserved Units, Streaming endpoints, Storage accounts), and Support + Troubleshooting (New support request). The 'Assets' link is highlighted with a red box. The main area shows a search bar and a table with columns 'NAME' and 'ASSET TYPE'. A message 'No results' is displayed.

10. Click on the **Folder** icon to browse your files and select the first 360 Video that you would like to stream.

You can follow this [link to download a sample video](#).

The screenshot shows the 'Upload a video asset' dialog. It displays a file named 'Video1.mp4' with a delete button. Below the file list, there are three informational messages: 1. 'Check the list of supported input formats.' 2. 'Check file size limitations for encoding.' 3. 'Check the list of supported characters for file names.'

WARNING

Long filenames may cause an issue with the encoder: so to ensure videos do not have issues, consider shortening the length of your video file names.

11. The progress bar will turn green when the video has finished uploading.

The screenshot shows a web-based interface for uploading a video asset. At the top, it says 'Home > thenewmediaservices - Assets > Upload a video asset'. Below that, there's a 'File upload' section with a progress bar for 'Video1.mp4'. The progress bar is nearly full, indicating the upload is almost complete.

12. Click on the text above (**your servicename - Assets**) to return to the **Assets** page.

13. You will notice that your video has been successfully uploaded. Click on it.

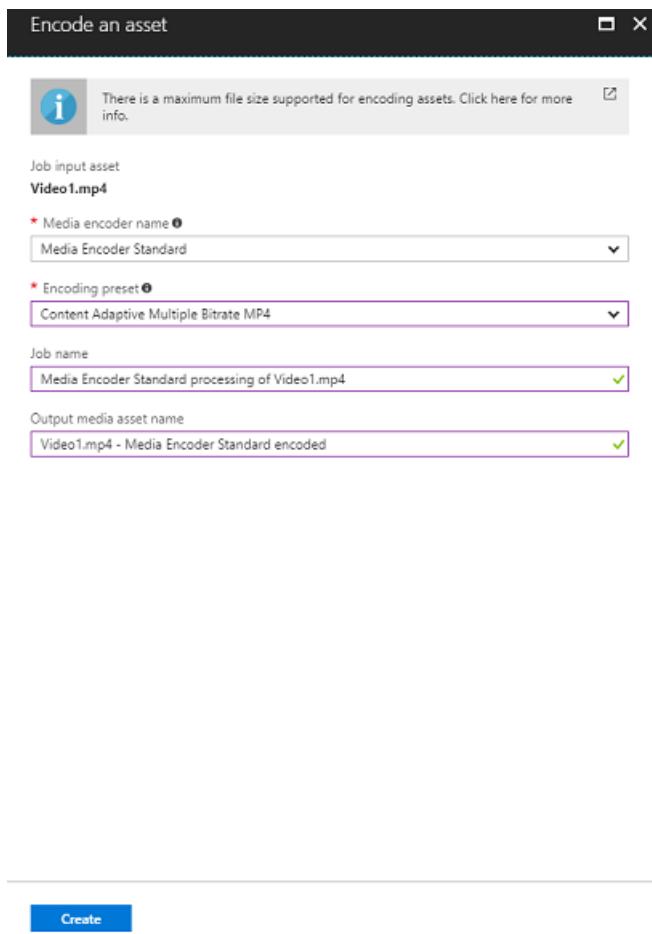
The screenshot shows the 'Assets' page with a list of uploaded files. On the left is a sidebar with links like Overview, Activity log, Access control (IAM), Tags, and Diagnosis and solve problems. The main area shows a table with columns 'NAME' and 'ASSET TYPE'. A row for 'Video1.mp4' is selected, highlighted with a red box around its name. The asset type is listed as 'Single MP4'.

14. The page you are redirected to will show you detailed information about your video. To be able to use your video you need to encode it, by clicking the **Encode** button at the top-left of the page.

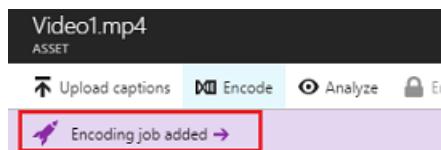
The screenshot shows the detailed view of the 'Video1.mp4' asset. At the top, there are several buttons: Upload captions, **Encode** (which is highlighted with a red box), Analyze, Encrypt, Remove Encryption, Publish, Unpublish, Play, and More. The main content area is divided into sections: Overview, Content Keys, Delivery Policies, Published URLs, and Files. Each section contains various configuration fields and status information. The 'Encode' button is located in the top navigation bar.

15. A new panel will appear to the right, where you will be able to set encoding options for your file. Set the following properties (some will be already set by default):

- a. **Media encoder name Media Encoder Standard**
- b. **Encoding preset Content Adaptive Multiple Bitrate MP4**
- c. **Job name Media Encoder Standard processing of Video1.mp4**
- d. **Output media asset name Video1.mp4 -- Media Encoder Standard encoded**



16. Click the **Create** button.
17. You will notice a bar with **Encoding job added**, click on that bar and a panel will appear with the Encoding progress displayed in it.



The screenshot shows the 'Job Media Encoder Standard processing of Video1.mp4' panel. It includes sections for Overview, Job Tasks, Task Errors, and Output Assets. The Job Tasks section shows a single task in progress with a progress bar at 64%. The Output Assets section shows one asset named 'Video1.mp4 - Media Encoder Standard encoded'.

18. Wait for the Job to be completed. Once it is done, feel free to close the panel with the 'X' at the top right of that panel.

The screenshot shows the 'Job Media Encoder Standard processing of Video01.mp4' panel. The Job Tasks section shows a task labeled 'Finished' with a progress bar at 100%.

The screenshot shows the 'Job Media Encoder Standard processing of Video01.mp4' panel. The close button in the top right corner is highlighted with a red box.

IMPORTANT

The time this takes, depends on the file size of your video. This process can take quite some time.

19. Now that the encoded version of the video has been created, you can publish it to make it accessible. To do so, click the blue link **Assets** to go back to the assets page.

The screenshot shows the Microsoft Azure Assets page for the asset 'Video1.mp4'. The left sidebar shows 'Create a resource', 'All services', 'FAVORITES' (Dashboard, All resources, Resource groups, App Services, Function Apps), and a 'Encoding job added' notification. The main content area shows the asset's Overview with fields like ID, LAST MODIFIED, ASSET TYPE (Single MP4), TOTAL SIZE (13.88 MB), and FILE COUNT (1).

20. You will see your video along with another, which is of **Asset Type Multi-Bitrate MP4**.

ASSET TYPE	LAST MODIFIED	SIZE
Unknown	[REDACTED]	0 bytes
Single MP4	[REDACTED]	333.57 MB

NOTE

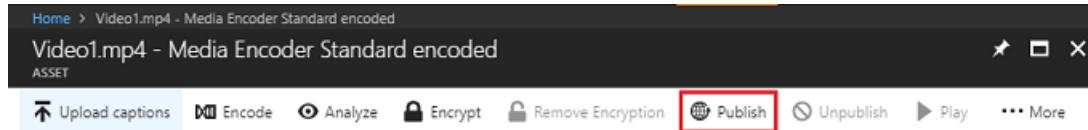
You may notice that the new asset, alongside your initial video, is *Unknown*, and has '0' bytes for it's **Size**, just refresh your window for it to update.

21. Click this new asset.



NAME	ASSET TYPE	LAST MODIFIED	SIZE
Video1.mp4 - Media Encoder Standard encoded	Multi-Bitrate MP4	Fri, Feb 23, 2018 15:29	34.93 MB
Video1.mp4	Single MP4	Fri, Feb 23, 2018 15:23	13.88 MB

22. You will see a similar panel to the one you used before, just this is a different asset. Click the **Publish** button located at the top-center of the page.



23. You will be prompted to set a **Locator**, which is the entry point, to file/s in your Assets. For your scenario set the following properties:

- Locator type > Progressive.**
- The **date** and **time** will be set for you, from your current date, to a time in the future (one hundred years in this case). Leave as is or change it to suit.

NOTE

For more information about Locators, and what you can choose, visit the [Azure Media Services Documentation](#).

24. At the bottom of that panel, click on the **Add** button.

Publish the asset

SETTINGS



By adding a locator to the asset,
you publish the asset

Locator type

Streaming Progressive

* Start date and time

2018-02-22 14:10:49

* End date and time

2118-02-22 14:10:49

Add

25. Your video is now published and can be streamed by using its endpoint. Further down the page is a **Files** section. This is where the different encoded versions of your video will be. Select the highest possible resolution one (in the image below it is the 1920x960 file), and then a panel to the right will appear. There you will find a **Download URL**. Copy this **Endpoint** as you will use it later in your code.

Upload captions			Encode	Analyze	Encrypt	Remove Encryption	Publish	Unpublish	Play	More
NAME	TYPE	AUTHORIZATION POLICY ID								
There are no keys in the asset.										
Delivery Policies										
NAME	TYPE	PROTOCOL								
There are no delivery policies in the asset.										
Published URLs										
LOCATOR TYPE	URL									
Progressive	https://[REDACTED]...									
Files										
NAME	MIME TYPE	SIZE								
Video1.ismc	application/octet-stream	1.00 KB								
Video1.ism	application/octet-stream	2.00 KB								
Video1_540x270_AACAudio_540.mp4	video/mp4	2.16 MB								
Video1_720x360_AACAudio_860.mp4	video/mp4	3.18 MB								
Video1_1080x540_AACAudio_1650.mp4	video/mp4	5.70 MB								
Video1_1440x720_AACAudio_2620.mp4	video/mp4	8.82 MB								
Video1_1920x960_AACAudio_4180.mp4	video/mp4	13.74 MB								
Video1_360x180_AACAudio_280.mp4	video/mp4	1.32 MB								
Video1_manifest.xml	text/xml	4.00 KB								
e1b09228-bd7f-486b-8cf7-ffabcf32051b_meta...	text/xml	2.00 KB								

Home > Video1.mp4 - Media Encoder Standard encoded

ASSET

Video1.mp4 - Media Encoder Standard encoded

Upload captions Encode Analyze Encrypt Remove Encryption Publish Unpublish **Play** More

Overview

ID: nb:cid:UUID:0E551A5C-91E1-42D2-921707625600

LAST MODIFIED: Thu, 22 Feb 2018, 13:52 GMT+11

ASSET TYPE: Multi-Bitrate MP4

TOTAL SIZE: 34.93 MB

FILE COUNT: 10

ENCRYPTION: No Encryption

NOTE

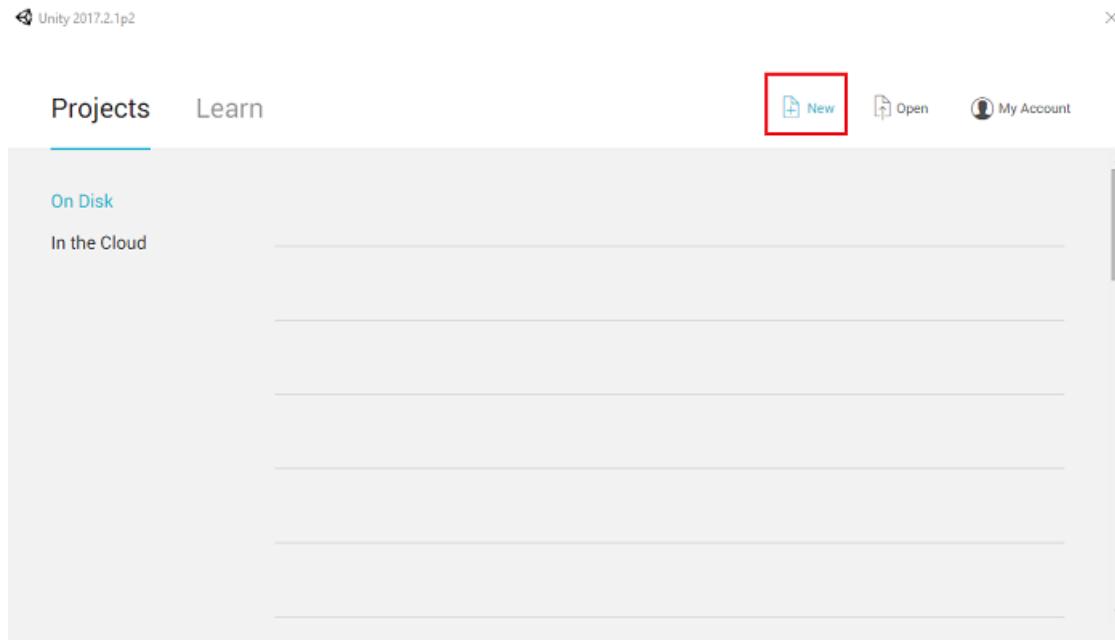
You can also press the **Play** button to play your video and test it.

26. You now need to upload the second video that you will use in this Lab. Follow the steps above, repeating the same process for the second video. Ensure you copy the second **Endpoint** also. Use the following [link to download a second video](#).
27. Once both videos have been published, you are ready to move to the next Chapter.

Chapter 3 - Setting up the Unity Project

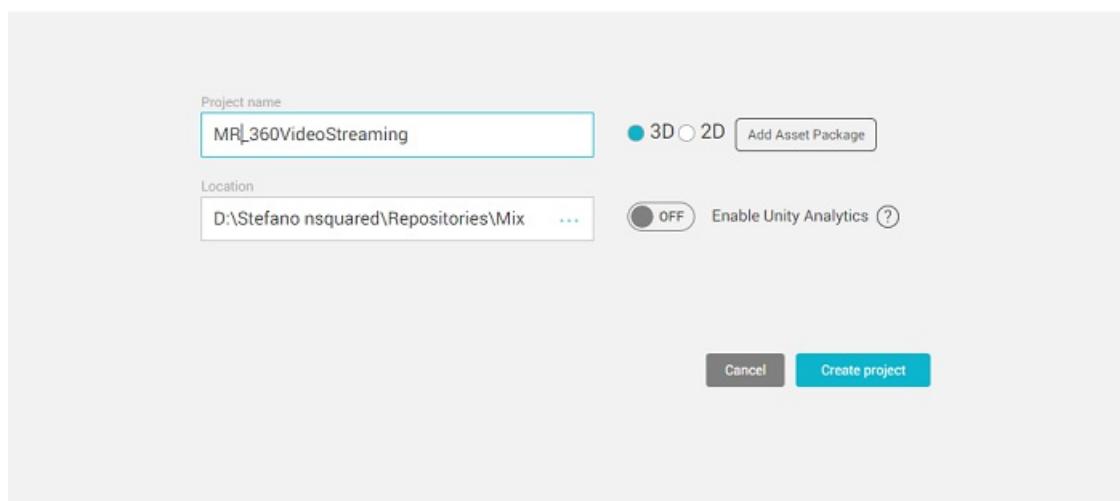
The following is a typical set up for developing with the Mixed Reality, and as such, is a good template for other projects.

1. Open **Unity** and click **New**.

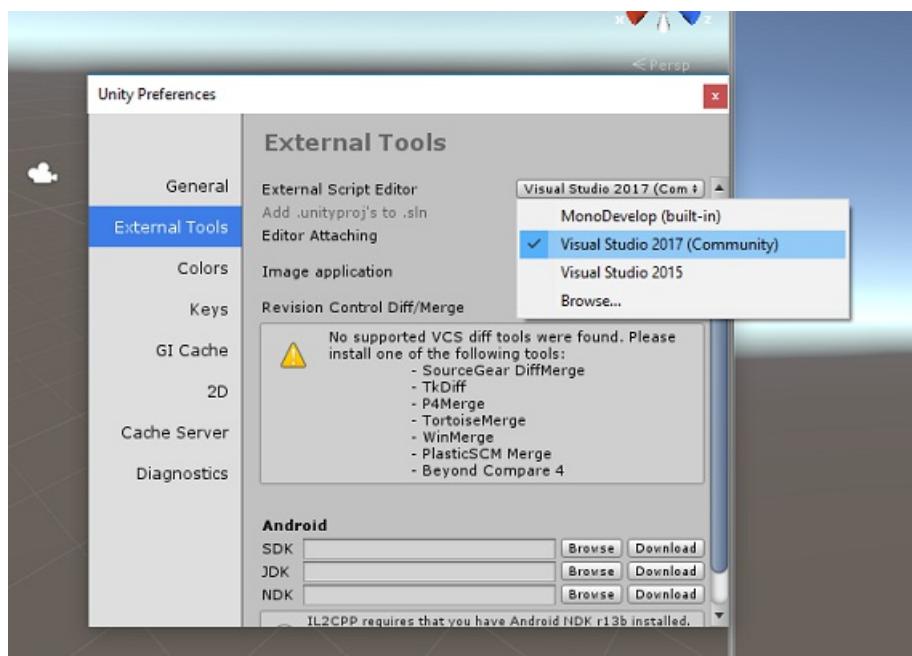


2. You will now need to provide a Unity Project name, insert **MR_360VideoStreaming**. Make sure the project type is set to **3D**. Set the Location to somewhere appropriate for you (remember, closer to root directories is better). Then, click **Create project**.

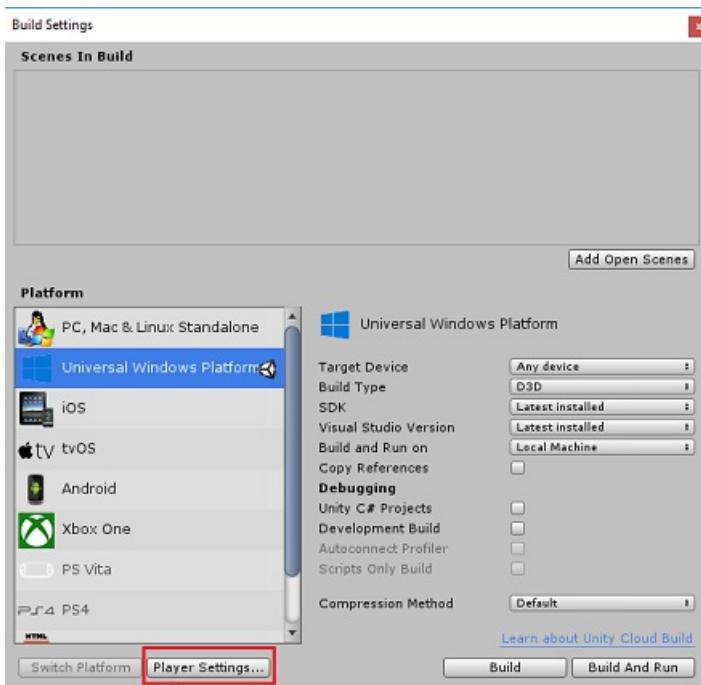
Projects Learn

[New](#) [Open](#) [My Account](#)

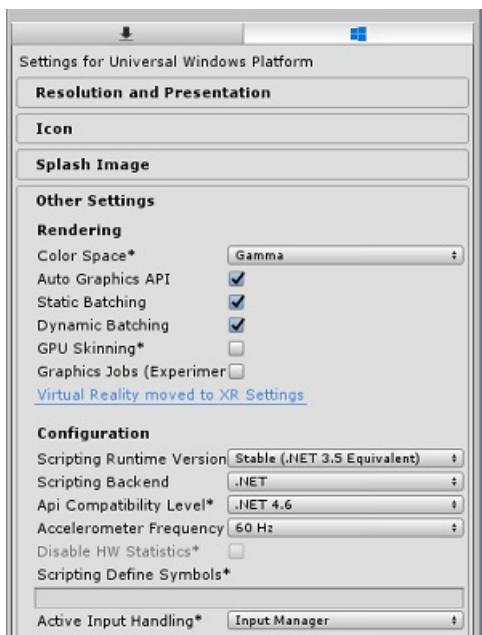
3. With Unity open, it is worth checking the default **Script Editor** is set to **Visual Studio**. Go to **Edit Preferences** and then from the new window, navigate to **External Tools**. Change **External Script Editor** to **Visual Studio 2017**. Close the **Preferences** window.



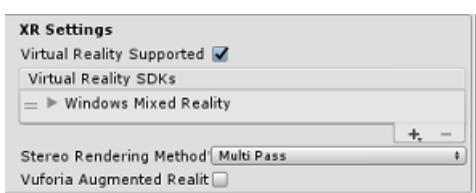
4. Next, go to **File Build Settings** and switch the platform to **Universal Windows Platform**, by clicking on the **Switch Platform** button.
5. Also make sure that:
- Target Device** is set to **Any Device**.
 - Build Type** is set to **D3D**.
 - SDK** is set to **Latest installed**.
 - Visual Studio Version** is set to **Latest installed**.
 - Build and Run** is set to **Local Machine**.
 - Do not worry about setting up **Scenes** right now, as you will set these up later.
 - The remaining settings should be left as default for now.



6. In the **Build Settings** window, click on the **Player Settings** button, this will open the related panel in the space where the **Inspector** is located.
7. In this panel, a few settings need to be verified:
 - a. In the **Other Settings** tab:
 - a. **Scripting Runtime Version** should be **Stable (.NET 3.5 Equivalent)**.
 - b. **Scripting Backend** should be **.NET**.
 - c. **API Compatibility Level** should be **.NET 4.6**.



- b. Further down the panel, in **XR Settings** (found below **Publish Settings**), tick **Virtual Reality Supported**, make sure the **Windows Mixed Reality SDK** is added.



c. Within the **Publishing Settings** tab, under **Capabilities**, check:

- **InternetClient**



8. Once you have made those changes, close the **Build Settings** window.

9. Save your Project *File Save Project*.

Chapter 4 - Importing the InsideOutSphere Unity package

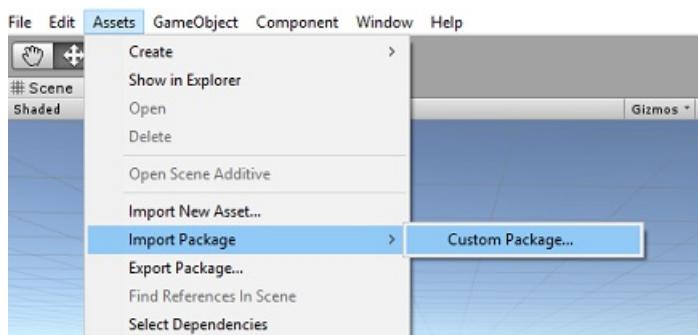
IMPORTANT

If you wish to skip the *Unity Set up* component of this course, and continue straight into code, feel free to download this [.unitypackage](#), import it into your project as a **Custom Package**, and then continue from **Chapter 5**. You will still need to create a Unity Project.

For this course you will need to download a Unity Asset Package called **InsideOutSphere.unitypackage**.

How-to import the **unitypackage**:

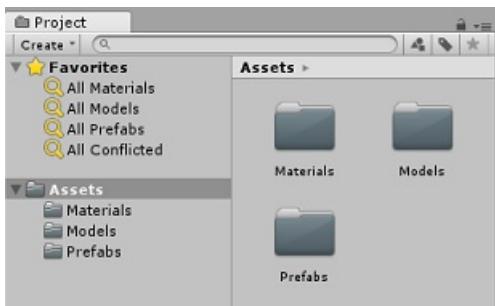
1. With the Unity dashboard in front of you, click on **Assets** in the menu at the top of the screen, then click on **Import Package > Custom Package**.



2. Use the file picker to select the **InsideOutSphere.unitypackage** package and click **Open**. A list of components for this asset will be displayed to you. Confirm the import by clicking **Import**.



3. Once it has finished importing, you will notice three new folders, **Materials**, **Models**, and **Prefabs**, have been added to your **Assets** folder. This kind of folder structure is typical for a Unity project.



- a. Open the **Models** folder, and you will see that the **InsideOutSphere** model has been imported.
- b. Within the **Materials** folder you will find the **InsideOutSpheres** material *lambert1*, along with a material called *ButtonMaterial*, which is used by the GazeButton, which you will see soon.
- c. The **Prefabs** folder contains the **InsideOutSphere** prefab which contains both the **InsideOutSphere** model and the *GazeButton*.
- d. **No code is included**, you will write the code by following this course.

4. Within the **Hierarchy**, select the **Main Camera** object, and update the following components:

a. **Transform**

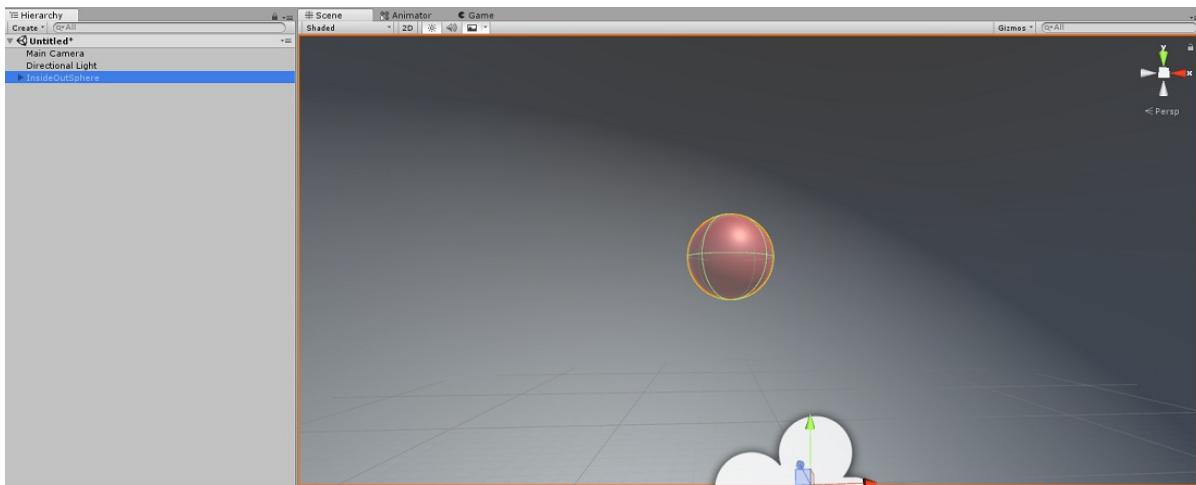
- a. Position = **X: 0, Y: 0, Z: 0**.
- b. Rotation = **X: 0, Y: 0, Z: 0**.
- c. Scale **X: 1, Y: 1, Z: 1**.

b. **Camera**

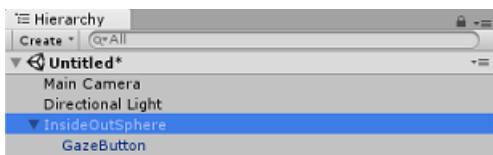
- a. **Clear Flags**: Solid Color.
- b. **Clipping Planes**: Near: 0.1, Far: 6.



5. Navigate to the **Prefab** folder, and then drag the **InsideOutSphere** prefab into the **Hierarchy** Panel.

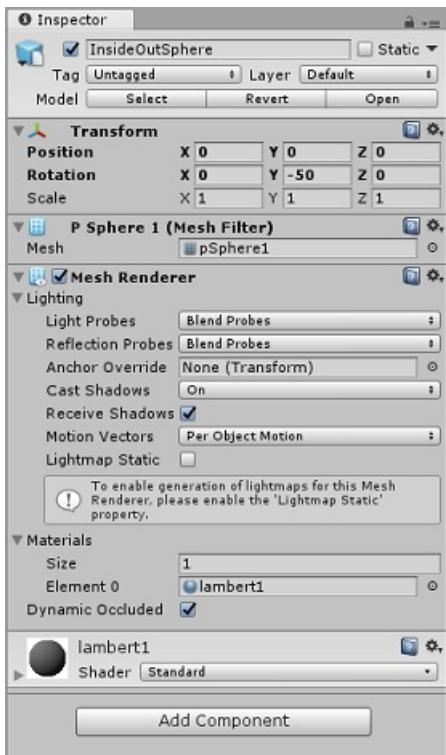


6. Expand the **InsideOutSphere** object within the **Hierarchy** by clicking the little arrow next to it. You will see a **child** object beneath it called **GazeButton**. This will be used to change scenes and thus videos.



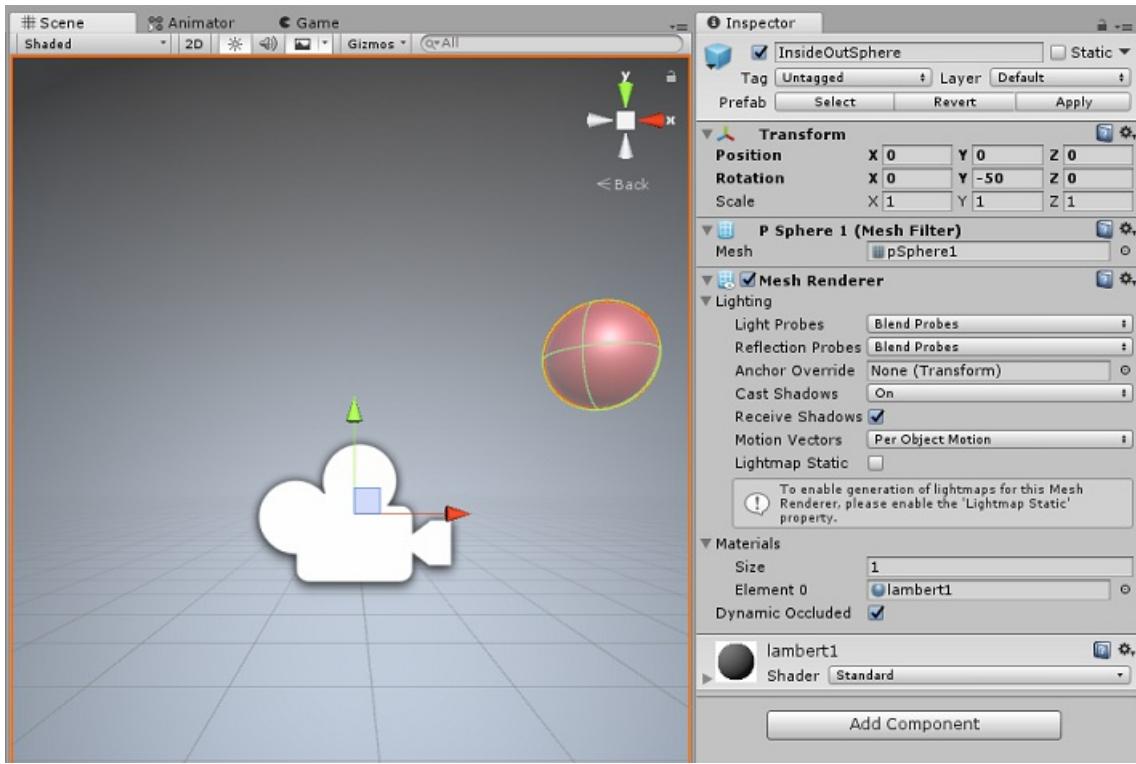
7. In the Inspector Window click on the **InsideOutSphere**'s Transform component, ensure that the following properties are set:

TRANSFORM - POSITION		
X	Y	Z
0	0	0
TRANSFORM - ROTATION		
X	Y	Z
0	-50	0
TRANSFORM - SCALE		
X	Y	Z
1	1	1



8. Click on the **GazeButton** child object, and set its **Transform** as follows:

TRANSFORM - POSITION		
X 3.6	Y 1.3	Z 0
TRANSFORM - ROTATION		
X 0	Y 0	Z 0
TRANSFORM - SCALE		
X 1	Y 1	Z 1

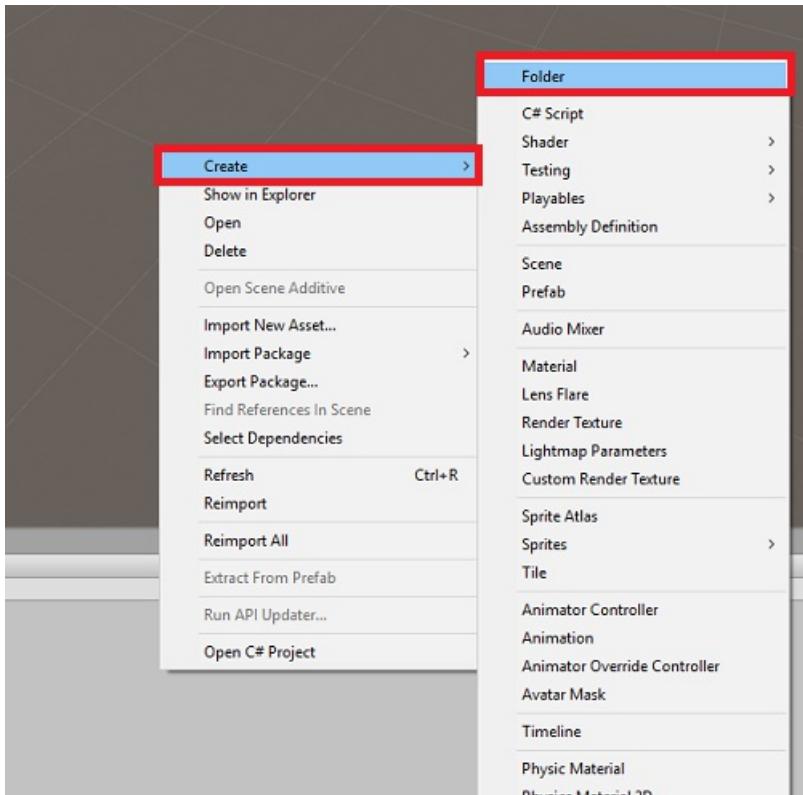


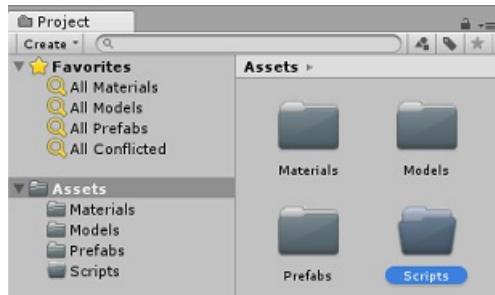
Chapter 5 - Create the VideoController class

The **VideoController** class hosts the two video endpoints that will be used to stream the content from the Azure Media Service.

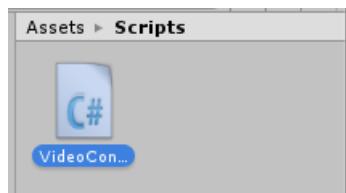
To create this class:

1. Right-click in the **Asset Folder**, located in the **Project Panel**, and click **Create > Folder**. Name the folder **Scripts**.





2. Double click on the **Scripts** folder to open it.
3. Right-click inside the folder, then click **Create > C# Script**. Name the script **VideoController**.



4. Double click on the new **VideoController** script to open it with **Visual Studio 2017**.

```
VideoController.cs  ↗ X
MR_360VideoStreaming
1 1 using System.Collections;
2 2 using System.Collections.Generic;
3 3 using UnityEngine;
4
5 public class VideoController : MonoBehaviour
6 {
7
8     // Use this for initialization
9     void Start()
10    {
11        ...
12    }
13
14     // Update is called once per frame
15     void Update()
16    {
17        ...
18    }
19
20 }
```

The screenshot shows the 'VideoController.cs' file in Visual Studio 2017. The code is a simple MonoBehaviour script with a Start() and Update() method. The 'using' statements at the top are missing some UnityEngine modules.

5. Update the namespaces at the top of the code file as follows:

```
using System.Collections;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.Video;
```

6. Enter the following variables in the **VideoController** class, along with the **Awake()** method:

```

/// <summary>
/// Provides Singleton-like behaviour to this class.
/// </summary>
public static VideoController instance;

/// <summary>
/// Reference to the Camera VideoPlayer Component.
/// </summary>
private VideoPlayer videoPlayer;

/// <summary>
/// Reference to the Camera AudioSource Component.
/// </summary>
private AudioSource audioSource;

/// <summary>
/// Reference to the texture used to project the video streaming
/// </summary>
private RenderTexture videoStreamRenderTexture;

/// <summary>
/// Insert here the first video endpoint
/// </summary>
private string video1endpoint = "-- Insert video 1 Endpoint here --";

/// <summary>
/// Insert here the second video endpoint
/// </summary>
private string video2endpoint = "-- Insert video 2 Endpoint here --";

/// <summary>
/// Reference to the Inside-Out Sphere.
/// </summary>
public GameObject sphere;

void Awake()
{
    instance = this;
}

```

7. Now is the time to enter the endpoints from your Azure Media Service videos:

- The first into the *video1endpoint* variable.
- The second into the *video2endpoint* variable.

WARNING

There is a known issue with using *https* within Unity, with version 2017.4.1f1. If the videos provide an error on play, try using 'http' instead.

8. Next, the **Start()** method needs to be edited. This method will be triggered every time the user switches scene (consequently switching the video) by looking at the Gaze Button.

```

// Use this for initialization
void Start()
{
    Application.runInBackground = true;
    StartCoroutine(PlayVideo());
}

```

9. Following the **Start()** method, insert the **PlayVideo() IEnumerator** method, which will be used to start

videos seamlessly (so no stutter is seen).

```
private IEnumerator PlayVideo()
{
    // create a new render texture to display the video
    videoStreamRenderTexture = new RenderTexture(2160, 1440, 32, RenderTextureFormat.ARGB32);

    videoStreamRenderTexture.Create();

    // assign the render texture to the object material
    Material sphereMaterial = sphere.GetComponent<Renderer>().sharedMaterial;

    //create a VideoPlayer component
    videoPlayer = gameObject.AddComponent<VideoPlayer>();

    // Set the video to loop.
    videoPlayer.isLooping = true;

    // Set the VideoPlayer component to play the video from the texture
    videoPlayer.renderMode = VideoRenderMode.RenderTexture;

    videoPlayer.targetTexture = videoStreamRenderTexture;

    // Add AudioSource
    audioSource = gameObject.AddComponent<
```

```

        while (videoPlayer.isPlaying)
    {
        yield return null;
    }
}

```

10. The last method you need for this class is the **ChangeScene()** method, which will be used to swap between scenes.

```

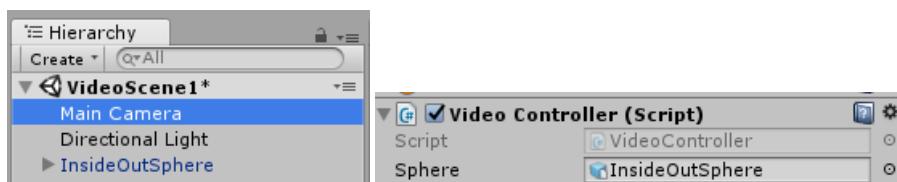
public void ChangeScene()
{
    SceneManager.LoadScene(SceneManager.GetActiveScene().name == "VideoScene1" ? "VideoScene2" :
    "VideoScene1");
}

```

TIP

The **ChangeScene()** method uses a handy C# feature called the *Conditional Operator*. This allows for conditions to be checked, and then values returned based on the outcome of the check, all within a single statement. Follow this [link to learn more about Conditional Operator](#).

11. Save your changes in Visual Studio before returning to Unity.
12. Back in the Unity Editor, click and drag the **VideoController** class [from]{underline} the **Scripts** folder to the **Main Camera** object in the **Hierarchy Panel**.
13. Click on the **Main Camera** and look at the **Inspector Panel**. You will notice that within the newly added Script component, there is a field with an empty value. This is a reference field, which targets the public variables within your code.
14. Drag the **InsideOutSphere** object from the **Hierarchy Panel** to the **Sphere** slot, as shown in the image below.



Chapter 6 - Create the Gaze class

This class is responsible for creating a **Raycast** that will be projected forward from the **Main Camera**, to detect which object the user is looking at. In this case, the **Raycast** will need to identify if the user is looking at the **GazeButton** object in the scene and trigger a behavior.

To create this Class:

1. Go to the **Scripts** folder you created previously.
2. Right-click in the **Project** Panel, *Create C# Script*. Name the script **Gaze**.
3. Double click on the new **Gaze** script to open it with **Visual Studio 2017**.
4. Ensure the following namespace is at the top of the script, and remove any others:

```
using UnityEngine;
```

5. Then add the following variables inside the **Gaze** class:

```
/// <summary>
/// Provides Singleton-like behaviour to this class.
/// </summary>
public static Gaze instance;

/// <summary>
/// Provides a reference to the object the user is currently looking at.
/// </summary>
public GameObject FocusedGameObject { get; private set; }

/// <summary>
/// Provides a reference to compare whether the user is still looking at
/// the same object (and has not looked away).
/// </summary>
private GameObject oldFocusedObject = null;

/// <summary>
/// Max Ray Distance
/// </summary>
float gazeMaxDistance = 300;

/// <summary>
/// Provides whether an object has been successfully hit by the raycast.
/// </summary>
public bool Hit { get; private set; }
```

6. Code for the **Awake()** and **Start()** methods now needs to be added.

```
private void Awake()
{
    // Set this class to behave similar to singleton
    instance = this;
}

void Start()
{
    FocusedGameObject = null;
}
```

7. Add the following code in the **Update()** method to project a Raycast and detect the target hit:

```

void Update()
{
    // Set the old focused gameobject.
    oldFocusedObject = FocusedGameObject;
    RaycastHit hitInfo;

    // Initialise Raycasting.
    Hit = Physics.Raycast(Camera.main.transform.position, Camera.main.transform.forward, out
    hitInfo, gazeMaxDistance);

    // Check whether raycast has hit.
    if (Hit == true)
    {
        // Check whether the hit has a collider.
        if (hitInfo.collider != null)
        {
            // Set the focused object with what the user just looked at.
            FocusedGameObject = hitInfo.collider.gameObject;
        }
        else
        {
            // Object looked on is not valid, set focused gameobject to null.
            FocusedGameObject = null;
        }
    }
    else
    {
        // No object looked upon, set focused gameobject to null.
        FocusedGameObject = null;
    }

    // Check whether the previous focused object is this same
    // object (so to stop spamming of function).
    if (FocusedGameObject != oldFocusedObject)
    {
        // Compare whether the new Focused Object has the desired tag we set previously.
        if (FocusedGameObject.CompareTag("GazeButton"))
        {
            FocusedGameObject.SetActive(false);
            VideoController.instance.ChangeScene();
        }
    }
}

```

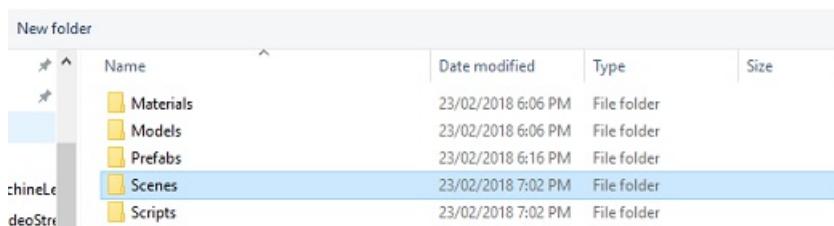
8. Save your changes in Visual Studio before returning to Unity.

9. Click and drag the **Gaze** class from the Scripts folder to the Main Camera object in the **Hierarchy** Panel.

Chapter 7 - Setup the two Unity Scenes

The purpose of this Chapter is to setup the two scenes, each hosting a video to stream. You will duplicate the scene you have already created, so that you do not need to set it up again, though you will then edit the new scene, so that the *GazeButton* object is in a different location and has a different appearance. This is to show how to change between scenes.

1. Do this by going to **File > Save Scene as....** A save window will appear. Click the **New folder** button.



2. Name the folder **Scenes**.
3. The **Save Scene** window will still be open. Open your newly created **Scenes** folder.
4. In the **File name:** text field, type **VideoScene1**, then press **Save**.
5. Back in Unity, open your **Scenes** folder, and left-click your **VideoScene1** file. Use your keyboard, and press **Ctrl + D** you will duplicate that scene

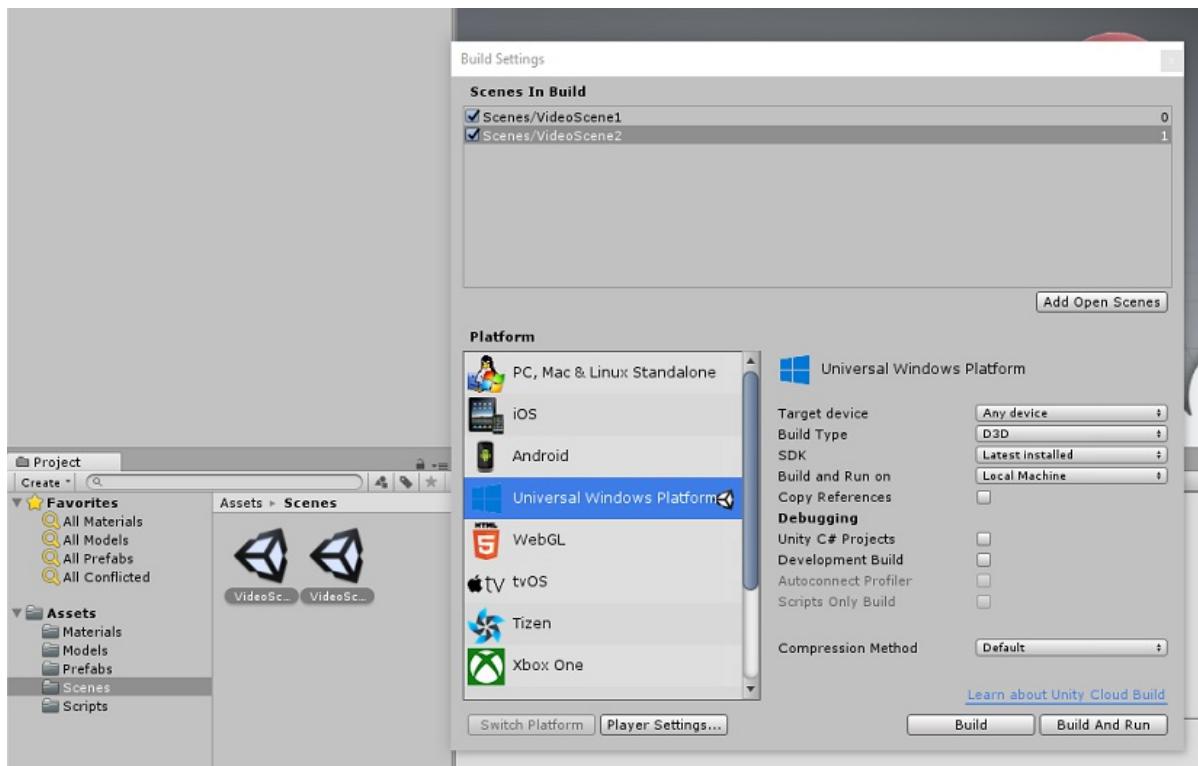
TIP

The **Duplicate** command can also be performed by navigating to **Edit > Duplicate**.

6. Unity will automatically increment the scene names number, but check it anyway, to ensure it matches the previously inserted code.

You should have **VideoScene1** and **VideoScene2**.

7. With your two scenes, go to **File > Build Settings**. With the **Build Settings** window open, drag your scenes to the **Scenes in Build** section.



TIP

You can select both of your scenes from your **Scenes** folder through holding the **Ctrl** button, and then left-clicking each scene, and finally drag both across.

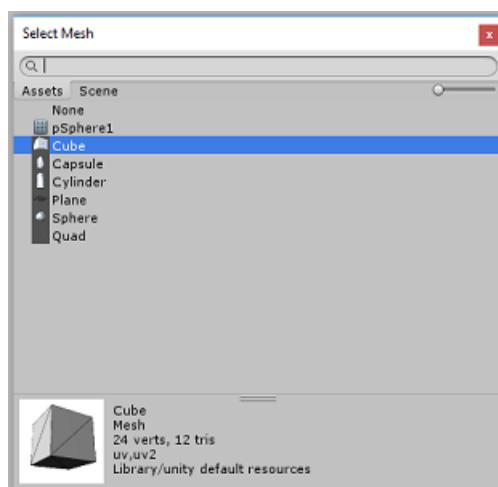
8. Close the **Build Settings** window, and double click on **VideoScene2**.
9. With the second scene open, click on the **GazeButton** child object of the **InsideOutSphere**, and set its Transform as follows:

TRANSFORM - POSITION		
X 0	Y 1.3	Z 3.6
TRANSFORM - ROTATION		
X 0	Y 0	Z 0
TRANSFORM - SCALE		
X 1	Y 1	Z 1

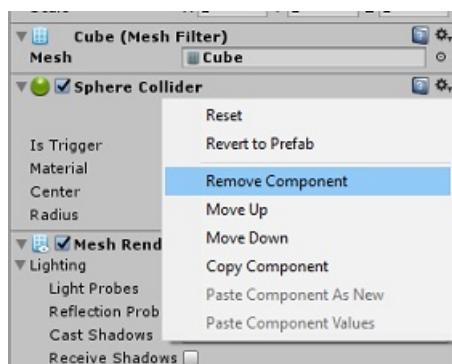
10. With the **GazeButton** child still selected, look at the **Inspector** and at the **Mesh Filter**. Click the little target next to the **Mesh** reference field:



11. A **Select Mesh** popup window will appear. Double click the **Cube** mesh from the list of **Assets**.

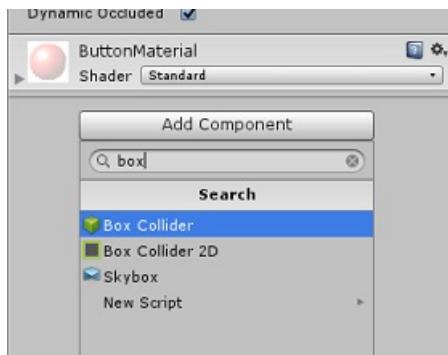


12. The **Mesh Filter** will update, and now be a **Cube**. Now, click the **Gear** icon next to **Sphere Collider** and click **Remove Component**, to delete the collider from this object.

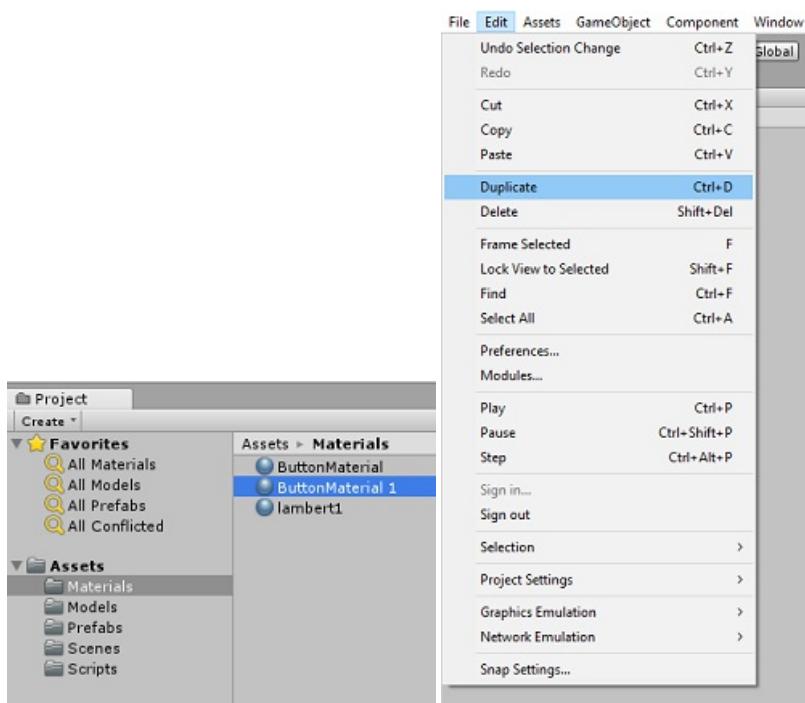


13. With the **GazeButton** still selected, click the **Add Component** button at the bottom of the **Inspector**. In the search field, type **box**, and **Box Collider** will be an option -- click that, to add a **Box Collider** to your

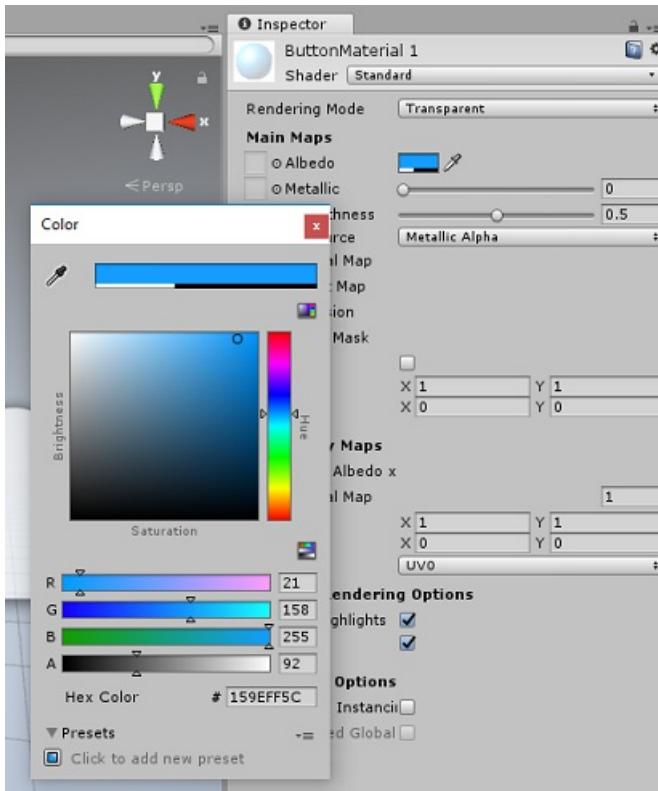
GazeButton object.



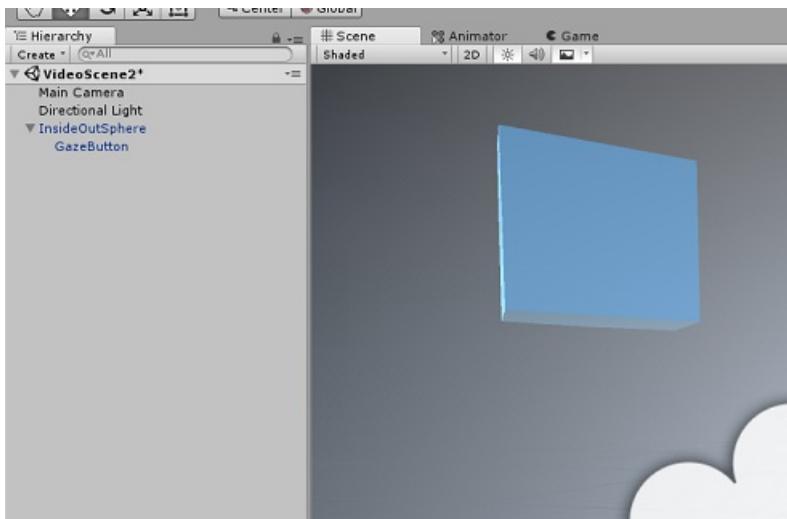
14. The **GazeButton** is now partially updated, to look different, however, you will now create a new **Material**, so that it looks completely different, and is easier to recognize as a different object, than the object in the first scene.
15. Navigate to your **Materials** folder, within the **Project Panel**. Duplicate the **ButtonMaterial** Material (press **Ctrl + D** on the keyboard, or left-click the **Material**, then from the **Edit** file menu option, select **Duplicate**).



16. Select the new **ButtonMaterial** Material (here named **ButtonMaterial 1**), and within the **Inspector**, click the **Albedo** color window. A popup will appear, where you can select another color (choose whichever you like), then close the popup. The Material will be its own instance, and different to the original.



17. Drag the new **Material** onto the **GazeButton** child, to now completely update its look, so that it is easily distinguishable from the first scenes button.



18. At this point you can test the project in the Editor before building the UWP project.

- Press the **Play** button in the **Editor** and wear your headset.



19. Look at the two **GazeButton** objects to switch between the first and second video.

Chapter 8 - Build the UWP Solution

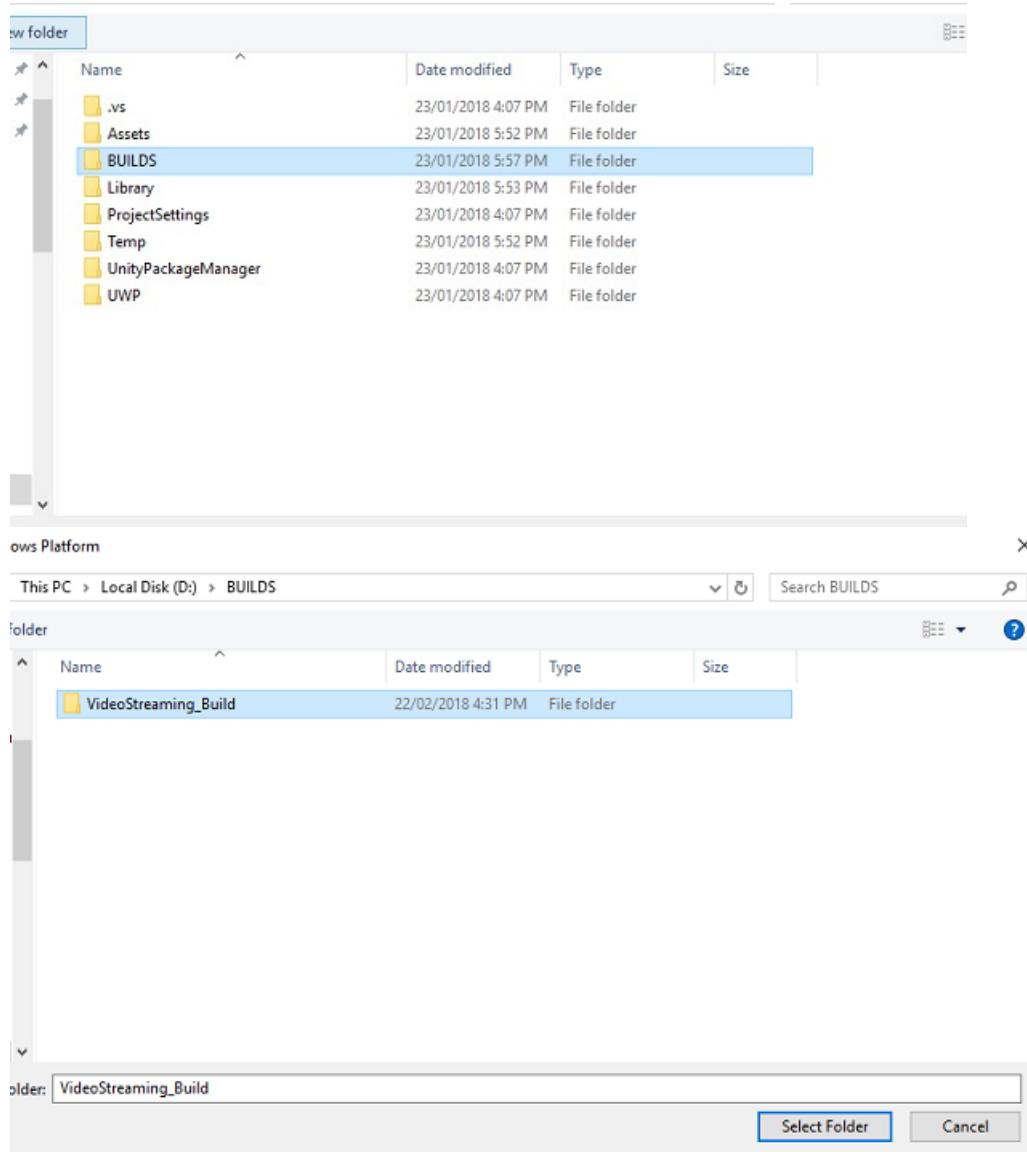
Once you have ensured that the editor has no errors, you are ready to Build.

To Build:

1. Save the current scene by clicking on **File > Save**.
2. Check the box called **Unity C# Projects** (this is important because it will allow you to edit the classes after

build is completed).

3. Go to **File > Build Settings**, click on **Build**.
4. You will be prompted to select the folder where you want to build the Solution.
5. Create a **BUILDS** folder and within that folder create another folder with an appropriate name of your choice.
6. Click your new folder and then click **Select Folder**, so to choose that folder, to begin the build at that location.



7. Once Unity has finished building (it might take some time), it will open a **File Explorer** window at the location of your build.

Chapter 9 - Deploy on Local Machine

Once the build has been completed, a **File Explorer** window will appear at the location of your build. Open the Folder you named and built to, then double click on the solution (.sln) file within that folder, to open your solution with Visual Studio 2017.

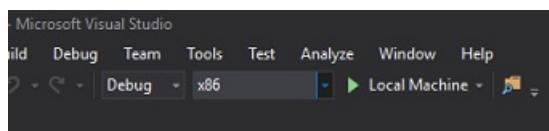
The only thing left to do is deploy your app to your computer (or *Local Machine*).

To deploy to Local Machine:

1. In **Visual Studio 2017**, open the solution file that has just been created.

2. In the **Solution Platform**, select **x86, Local Machine**.

3. In the **Solution Configuration** select **Debug**.



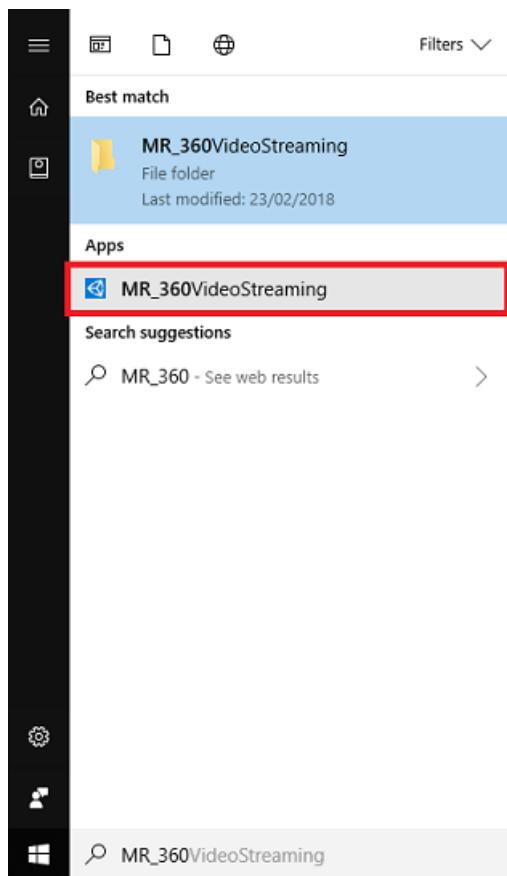
4. You will now need to restore any packages to your solution. Right-click on your **Solution**, and click **Restore NuGet Packages for Solution...**

NOTE

This is done because the packages which Unity built need to be targeted to work with your local machines references.

5. Go to **Build menu** and click on **Deploy Solution** to sideload the application to your machine. Visual Studio will first build and then deploy your application.

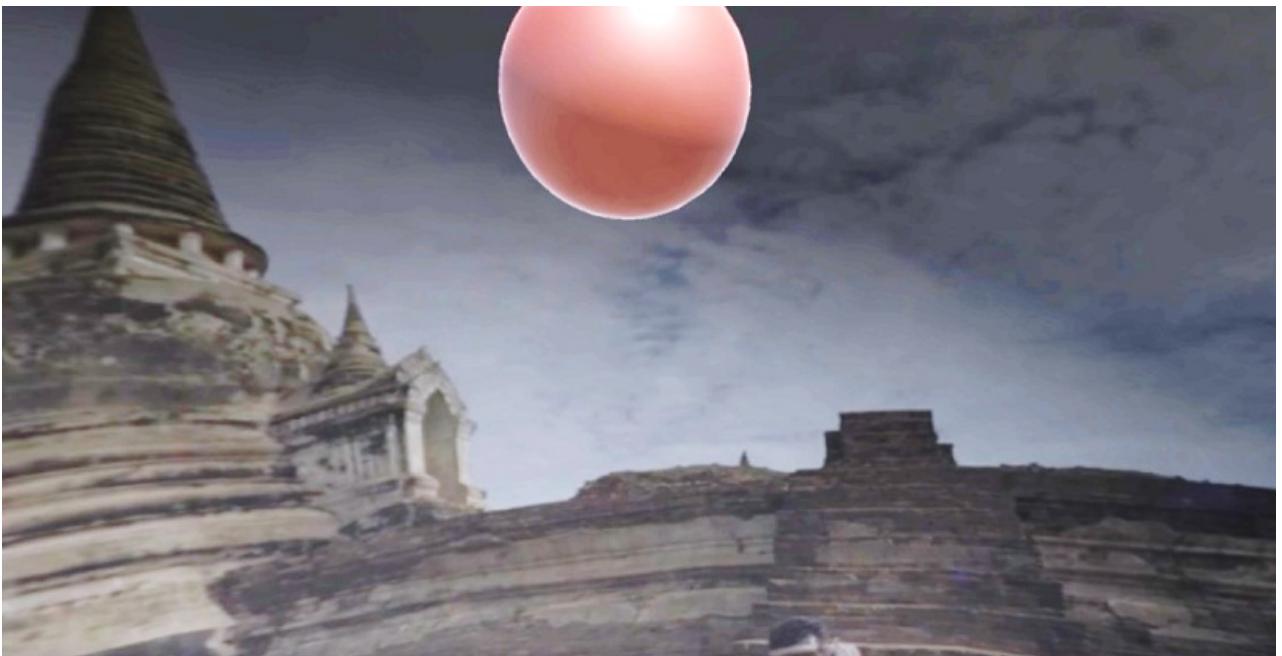
6. Your App should now appear in the list of installed apps, ready to be launched.



When you run the Mixed Reality application, you will be within the **InsideOutSphere** model which you used within your app. This sphere will be where the video will be streamed to, providing a 360-degree view, of the incoming video (which was filmed for this kind of perspective). Do not be surprised if the video takes a couple of seconds to load, your app is subject to your available Internet speed, as the video needs to be fetched and then downloaded, so to stream into your app. When you are ready, change scenes and open your second video, by gazing at the red sphere! Then feel free to go back, using the blue cube in the second scene!

Your finished Azure Media Service application

Congratulations, you built a mixed reality app that leverages the Azure Media Service to stream 360 videos.



Bonus Exercises

Exercise 1

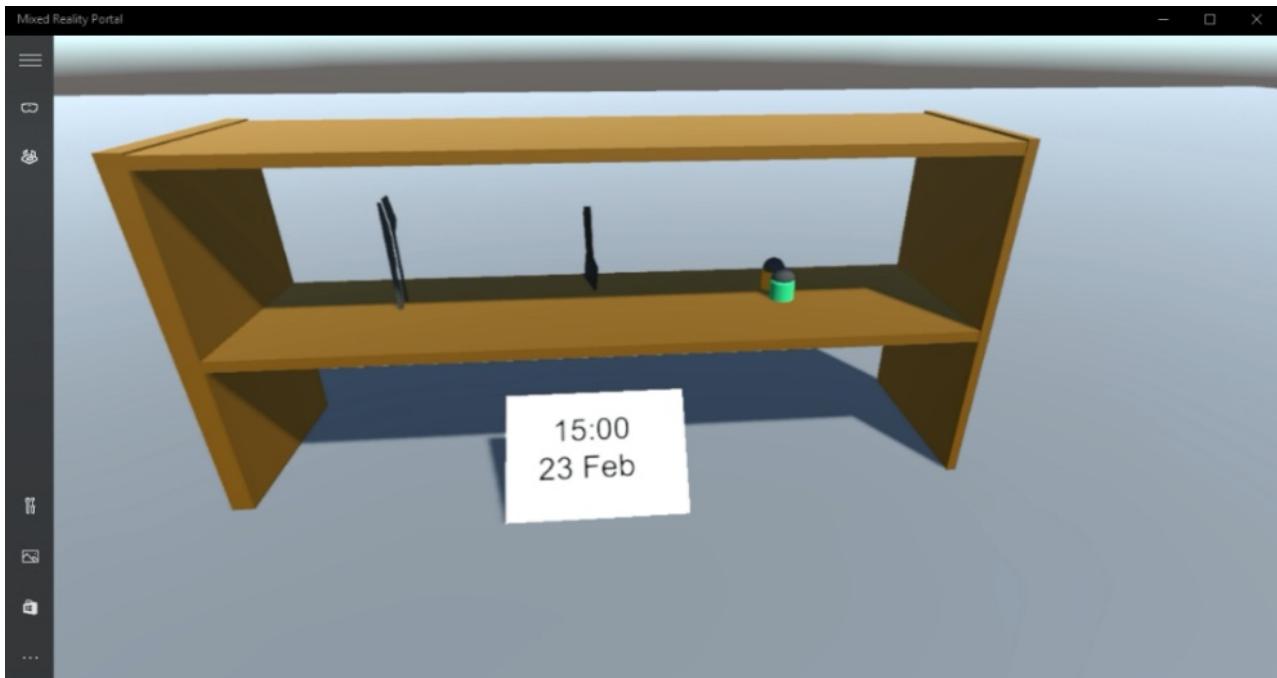
It is entirely possible to only use a single scene to change videos within this tutorial. Experiment with your application and make it into a single scene! Perhaps even add another video to the mix.

Exercise 2

Experiment with Azure and Unity, and attempt to implement the ability for the app to automatically select a video with a different file size, depending on the strength of an Internet connection.

MR and Azure 307: Machine learning

11/6/2018 • 27 minutes to read • [Edit Online](#)



In this course, you will learn how to add Machine Learning (ML) capabilities to a mixed reality application using Azure Machine Learning Studio.

Azure Machine Learning Studio is a Microsoft service, which provides developers with a large number of machine learning algorithms, which can help with data input, output, preparation, and visualization. From these components, it is then possible to develop a predictive analytics experiment, iterate on it, and use it to train your model. Following training, you can make your model operational within the Azure cloud, so that it can then score new data. For more information, visit the [Azure Machine Learning Studio page](#).

Having completed this course, you will have a mixed reality immersive headset application, and will have learned how to do the following:

1. Provide a table of sales data to the *Azure Machine Learning Studio* portal, and design an algorithm to predict future sales of popular items.
2. Create a **Unity Project**, which can receive and interpret prediction data from the ML service.
3. Display the prediction data visually within the **Unity Project**, through providing the most popular sales items, on a shelf.

In your application, it is up to you as to how you will integrate the results with your design. This course is designed to teach you how to integrate an Azure Service with your Unity Project. It is your job to use the knowledge you gain from this course to enhance your mixed reality application.

This course is a self-contained tutorial, which does not directly involve any other Mixed Reality Labs.

Device support

COURSE	HOLOLENS	IMMERSIVE HEADSETS
MR and Azure 307: Machine learning	✓ <input type="checkbox"/>	✓ <input type="checkbox"/>

NOTE

While this course primarily focuses on Windows Mixed Reality immersive (VR) headsets, you can also apply what you learn in this course to Microsoft HoloLens. As you follow along with the course, you will see notes on any changes you might need to employ to support HoloLens. When using HoloLens, you may notice some echo during voice capture.

Prerequisites

NOTE

This tutorial is designed for developers who have basic experience with Unity and C#. Please also be aware that the prerequisites and written instructions within this document represent what has been tested and verified at the time of writing (May 2018). You are free to use the latest software, as listed within the [install the tools article](#), though it should not be assumed that the information in this course will perfectly match what you'll find in newer software than what's listed below.

We recommend the following hardware and software for this course:

- A development PC, [compatible with Windows Mixed Reality](#) for immersive (VR) headset development
- [Windows 10 Fall Creators Update \(or later\)](#) with Developer mode enabled
- [The latest Windows 10 SDK](#)
- [Unity 2017.4](#)
- [Visual Studio 2017](#)
- A [Windows Mixed Reality immersive \(VR\) headset](#) or [Microsoft HoloLens](#) with Developer mode enabled
- Internet access for Azure setup and ML data retrieval

Before you start

To avoid encountering issues building this project, it is strongly suggested that you create the project mentioned in this tutorial in a root or near-root folder (long folder paths can cause issues at build-time).

Chapter 1 - Azure Storage Account setup

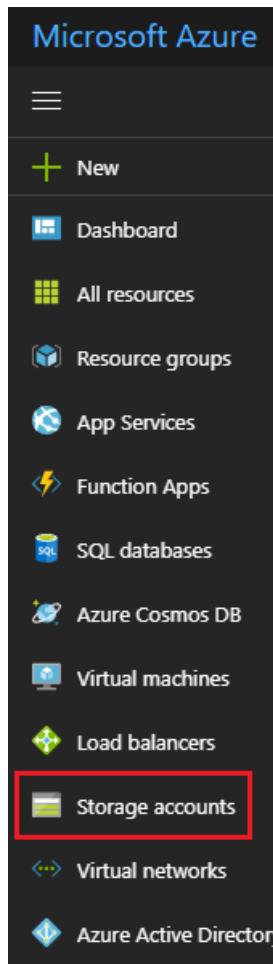
To use the Azure Translator API, you will need to configure an instance of the service to be made available to your application.

1. Log in to the [Azure Portal](#).

NOTE

If you do not already have an Azure account, you will need to create one. If you are following this tutorial in a classroom or lab situation, ask your instructor or one of the proctors for help setting up your new account.

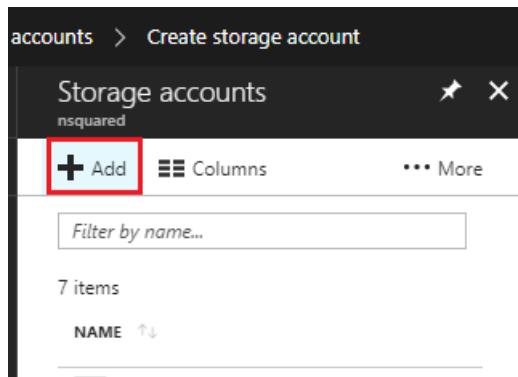
2. Once you are logged in, click on **Storage Accounts** in the left menu.



NOTE

The word **New** may have been replaced with **Create a resource**, in newer portals.

3. On the **Storage Accounts** tab, click on **Add**.



4. In the **Create Storage Account** panel:

- Insert a **Name** for your account, be aware this field only accepts numbers, and lowercase letters.
- For **Deployment model**, select **Resource manager**.
- For **Account kind**, select **Storage (general purpose v1)**.
- For **Performance**, select **Standard**.
- For **Replication** select **Read-access-geo-redundant storage (RA-GRS)**.
- Leave **Secure transfer required** as **Disabled**.

g. Select a **Subscription**.

- h. Choose a **Resource Group** or create a new one. A resource group provides a way to monitor, control access, provision and manage billing for a collection of Azure assets. It is recommended to keep all the Azure services associated with a single project (e.g. such as these labs) under a common resource group).

If you wish to read more about Azure Resource Groups, please [visit the resource group article](#).

- i. Determine the **Location** for your resource group (if you are creating a new Resource Group). The location would ideally be in the region where the application would run. Some Azure assets are only available in certain regions.

5. You will also need to confirm that you have understood the Terms and Conditions applied to this Service.

Create storage account X

The cost of your storage account depends on the usage and the options you choose below.
[Learn more](#)

*** Name** ✓
mynewazurestorage.core.windows.net

Deployment model Resource manager Classic

Account kind Storage (general purpose v1)

Performance Standard Premium

Replication Read-access geo-redundant storage (RA...)

*** Secure transfer required** Disabled Enabled

*** Subscription** Main Subscription

*** Resource group** Create new Use existing

Azure_for_Unity

*** Location** West US

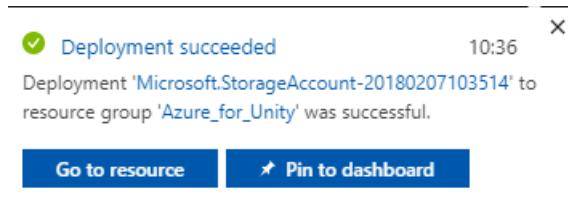
Virtual networks
Configure virtual networks Enabled

Pin to dashboard

Create [Automation options](#)

6. Once you have clicked on **Create**, you will have to wait for the service to be created, this might take a minute.

7. A notification will appear in the portal once the Service instance is created.



Chapter 2 - The Azure Machine Learning Studio

To use the *Azure Machine Learning*, you will need to configure an instance of the Machine Learning service to be made available to your application.

1. In the Azure Portal, click on **New** in the top left corner, and search for **Machine Learning Studio Workspace**, press **Enter**.

Machine Learning Studio Workspace

Azure Machine Learning Studio is a powerful cloud-based predictive analytics service that makes it possible to quickly create and deploy predictive models as analytics solutions.

Use this template to create an Azure Machine Learning Studio Workspace. A Workspace allows you to use Machine Learning Studio to create and manage machine learning experiments and predictive web services. You can create multiple Workspaces, each one containing a set of your experiments, datasets, trained predictive models, web services, and notebooks. As the owner of a Workspace, you can invite other users to share the Workspace so you can collaborate with them on predictive analytics solutions.

Create

2. The new page will provide a description of the **Machine Learning Studio Workspace** service. At the bottom left of this prompt, click the **Create** button, to create an association with this service.
3. Once you have clicked on **Create**, a panel will appear where you need to provide some details about your

new Machine Learning Studio service:

- a. Insert your desired **Workspace name** for this service instance.
- b. Select a **Subscription**.
- c. Choose a **Resource Group** or create a new one. A resource group provides a way to monitor, control access, provision and manage billing for a collection of Azure assets. It is recommended to keep all the Azure services associated with a single project (e.g. such as these labs) under a common resource group).

If you wish to read more about Azure Resource Groups, please [visit the resource group article](#).
- d. Determine the **Location** for your resource group (if you are creating a new Resource Group). The location would ideally be in the region where the application would run. Some Azure assets are only available in certain regions. You should use the same resource group that you used for creating the Azure Storage in the previous Chapter.
- e. For the **Storage account** section, click **Use existing**, then click the dropdown menu, and from there, click the **Storage Account** you created in the last Chapter.
- f. Select the appropriate **Workspace pricing tier** for you, from the dropdown menu.
- g. Within the **Web service plan** section, click **Create new**, then insert a name for it in the text field.
- h. From the **Web service plan pricing tier** section, select the price tier of your choice. A development testing tier called **DEVTEST Standard** should be available to you at no charge.
- i. You will also need to confirm that you have understood the Terms and Conditions applied to this Service.
- j. Click **Create**.

Machine Learning Studio workspace

* Workspace name
MyNewMLSW ✓

* Subscription
Main Subscription

* Resource group
Create new Use existing
Azure_for_Unity

* Location
West Central US

* Storage account ⓘ
Create new
mynewmlswstorage ✓

Workspace pricing tier ⓘ
Standard

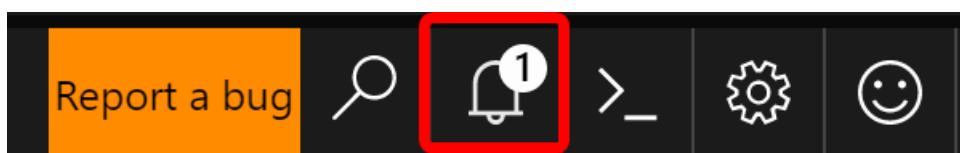
* Web service plan ⓘ
Create new
MyNewMLSWPlan

* Web service plan pricing tier ⓘ
DevTest Standard

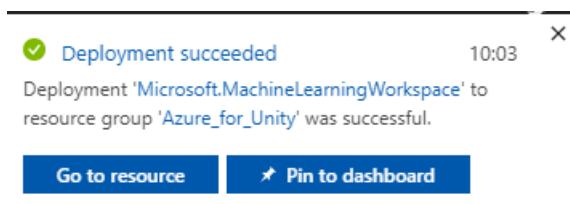
Pin to dashboard

Create Automation options

- Once you have clicked on **Create**, you will have to wait for the service to be created, this might take a minute.
- A notification will appear in the portal once the Service instance is created.



- Click on the notification to explore your new Service instance.



- Click the **Go to resource** button in the notification to explore your new Service instance.

8. In the page displayed, under the **Additional Links** section, click **Launch Machine Learning Studio**, which will direct your browser to the **Machine Learning Studio** portal.

The screenshot shows the 'MyNewMLSW' workspace settings in the Azure portal. On the left, there's a sidebar with links like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Locks, Automation script, Properties, Resync Storage Keys, and New support request. The main area shows workspace details: Resource group (Azure_for_Unity), Status (Enabled), Location (West Central US), Subscription name (Main Subscription), and Subscription ID (733f8637-3113-4e02-a647-b8d0406f309b). Below this is the 'Additional Links' section, which contains three items: 'Launch Machine Learning Studio' (with a red box around it), 'Launch Machine Learning Gallery', and 'Launch Machine Learning Studio Web Service Management'.

9. Use the **Sign In** button, at the top right or in the center, to log into your Machine Learning Studio.

The screenshot shows the 'Azure Machine Learning Workbench PREVIEW' landing page. It features a large blue 'Try it today!' button. To its right, there's a 'Welcome to Azure Machine Learning' message, a 'Try it for free' section, and a 'Sign In' button (which is highlighted with a red box). Below the sign-in button is a link for 'Not an Azure ML user? Sign up here'. At the bottom, there's a 'Pricing & FAQ' section and a note about terms of use.

Chapter 3 - The Machine Learning Studio: Dataset setup

One of the ways Machine Learning algorithms work is by analyzing existing data and then attempting to predict future results based on the existing data set. This generally means that the more existing data you have, the better the algorithm will be at predicting future results.

A sample table is provided to you, for this course, called **ProductsTableCSV** and can be downloaded [here](#).

IMPORTANT

The above .zip file contains both the **ProductsTableCSV** and the **.unitypackage**, which you will need in [Chapter 6](#). This package is also provided within that Chapter, though separate to the csv file.

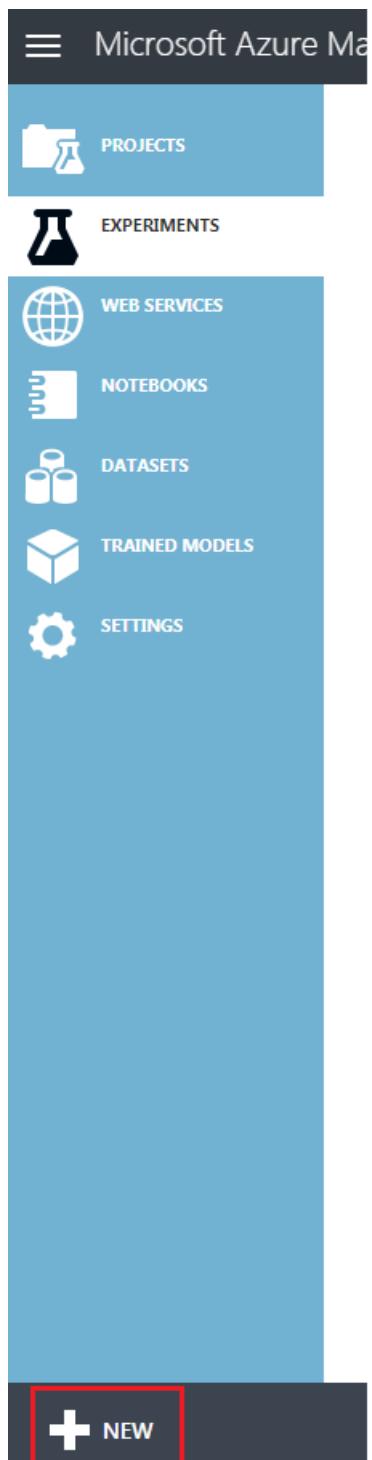
This sample data set contains a record of the best-selling objects at every hour of each day of the year 2017.

	A	B	C	D
1	day	hour	product	
2	1	9	cutlery	
3	1	9	cutlery	
4	1	9	ladle	
5	1	10	knife	
6	1	10	spatula	
7	1	11	choppingboard	
8	1	11	ladle	
9	1	12	cup	
10	1	12	spatula	
11	1	13	saltpepper	
12	1	14	cup	
13	1	14	knife	
14	1	14	cutlery	
15	1	14	cutlery	
16	1	14	knife	
17	1	14	spatula	
18	1	15	plate	
19	1	16	plate	
20	1	17	knife	
21	1	17	cutlery	

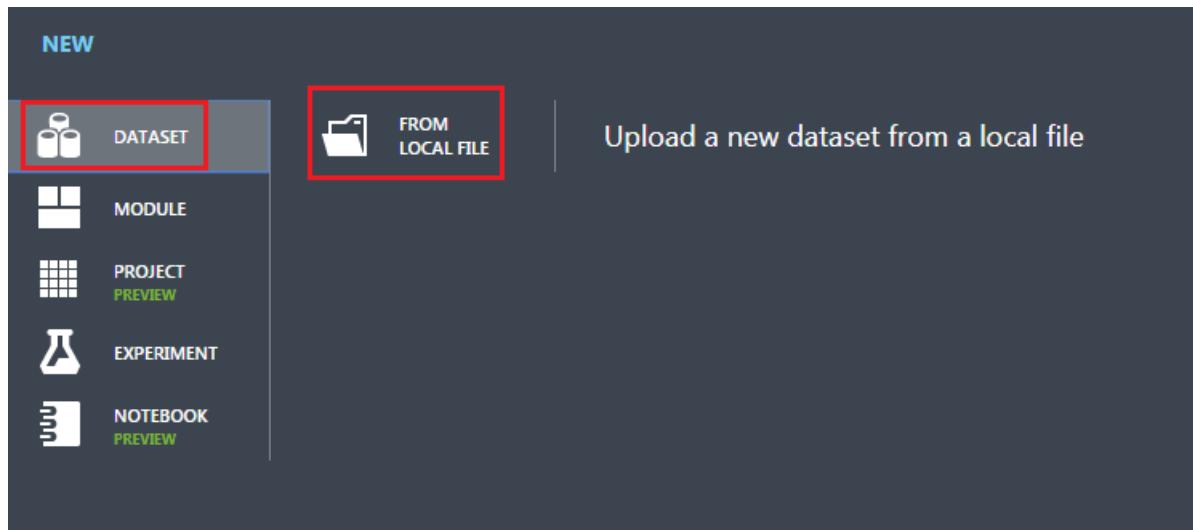
For example, on day 1 of 2017, at 1pm (hour 13), the best-selling item was salt and pepper.

This sample table contains 9998 entries.

1. Head back to the **Machine Learning Studio** portal, and add this table as a **Dataset** for your ML. Do this by clicking the **+ New** button in the bottom left corner of the screen.

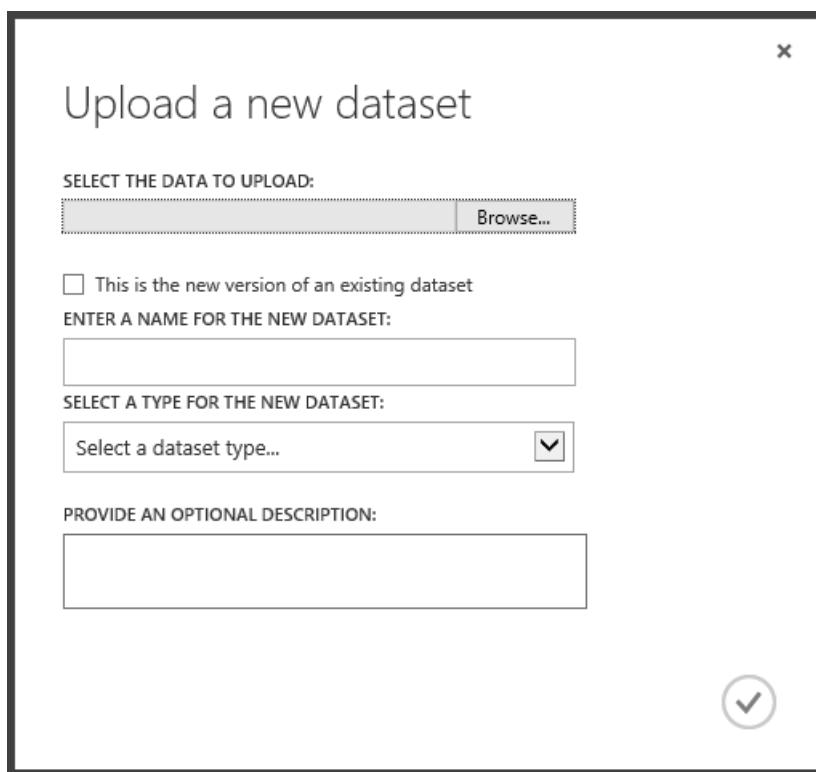


2. A section will come up from the bottom, and within that there is navigation panel on the left. Click **Dataset**, then to the right of that, **From Local File**.



3. Upload the new **Dataset** by following these steps:

- The upload window will appear, where you can **Browse** your hard drive for the new dataset.



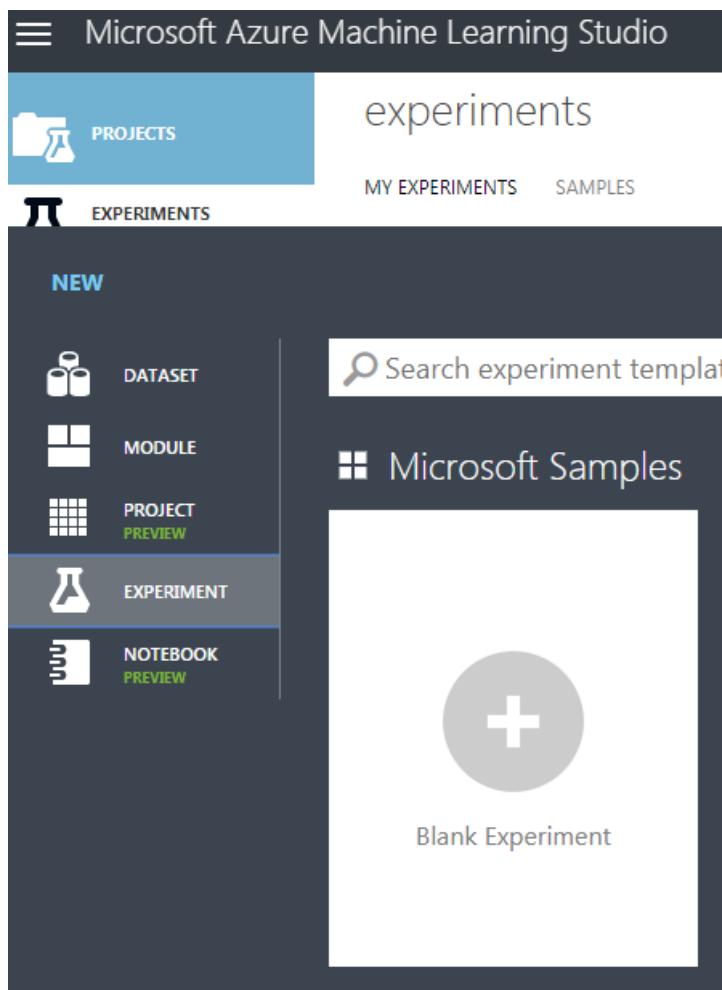
- Once selected, and back in the upload window, leave the checkbox unticked.
- In the text field below, enter **ProductsTableCSV.csv** as the name for the dataset (though should automatically be added).
- Using the dropdown menu for **Type**, select **Generic CSV File with a header (.csv)**.
- Press the tick in the bottom right of the upload window, and your **Dataset** will be uploaded.

Chapter 4 - The Machine Learning Studio: The Experiment

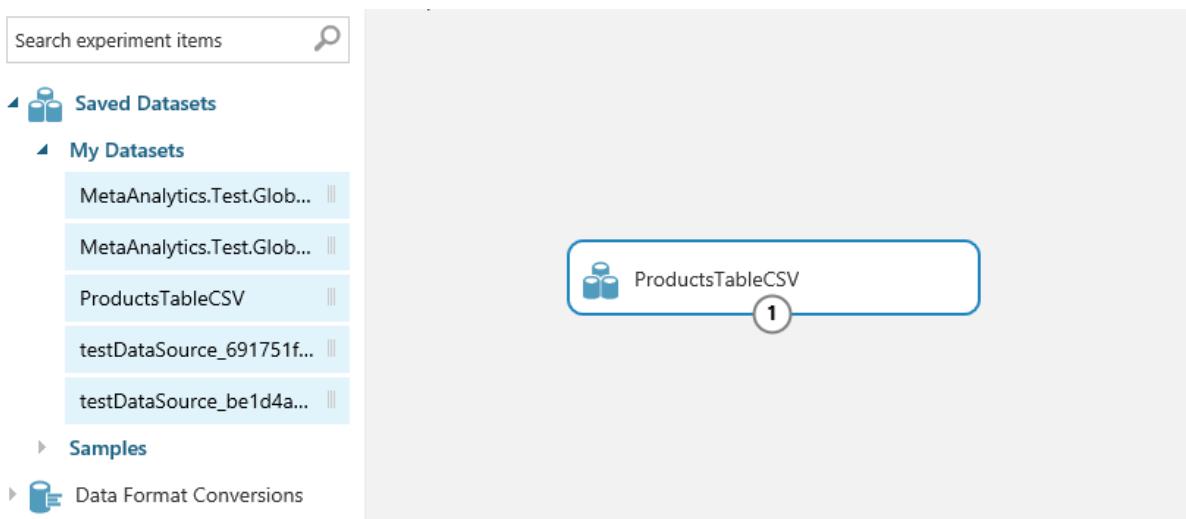
Before you can build your machine learning system, you will need to build an experiment, to validate your theory about your data. With the results, you will know whether you need more data, or if there is no correlation between the data and a possible outcome.

To start creating an experiment:

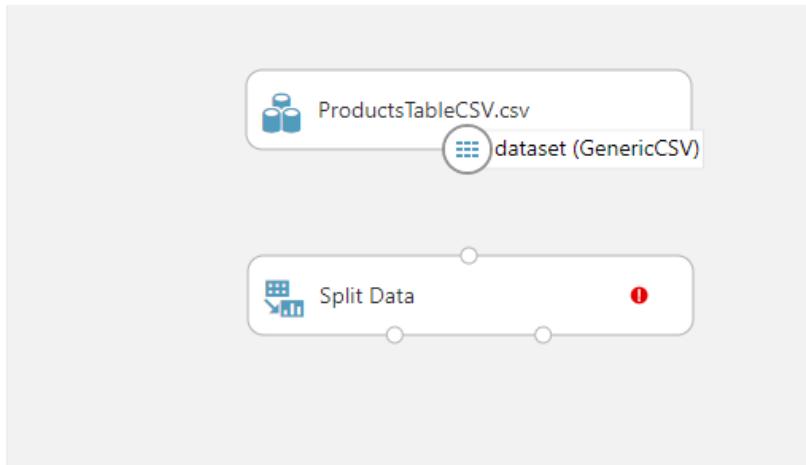
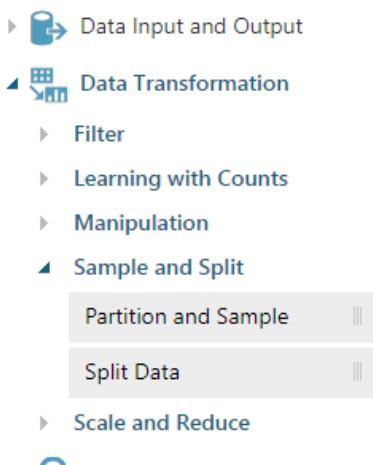
1. Click again on the **+ New** button on the bottom left of the page, then click on **Experiment > Blank Experiment**.



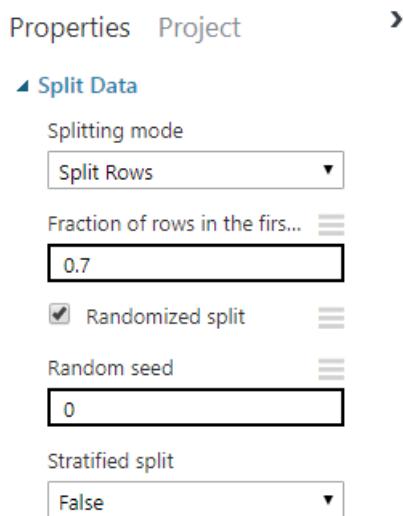
2. A new page will be displayed with a blank Experiment:
3. From the panel on the left expand *Saved Datasets > My Datasets* and drag the **ProductsTableCSV** on to the **Experiment Canvas**.



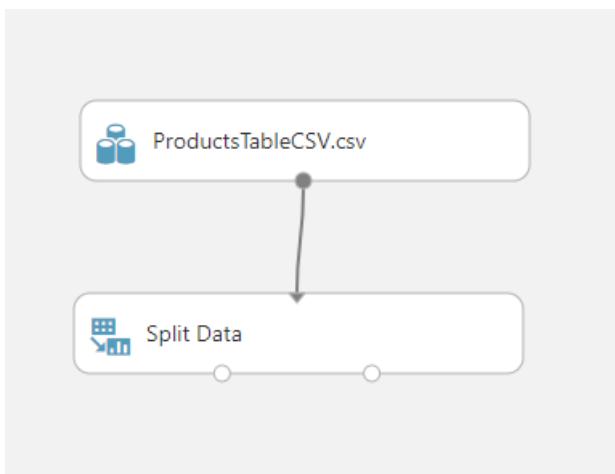
4. In the panel on the left, expand **Data Transformation > Sample and Split**. Then drag the **Split Data** item in to the **Experiment Canvas**. The Split Data item will split the data set into two parts. One part you will use for training the machine learning algorithm. The second part will be used to evaluate the accuracy of the algorithm generated.



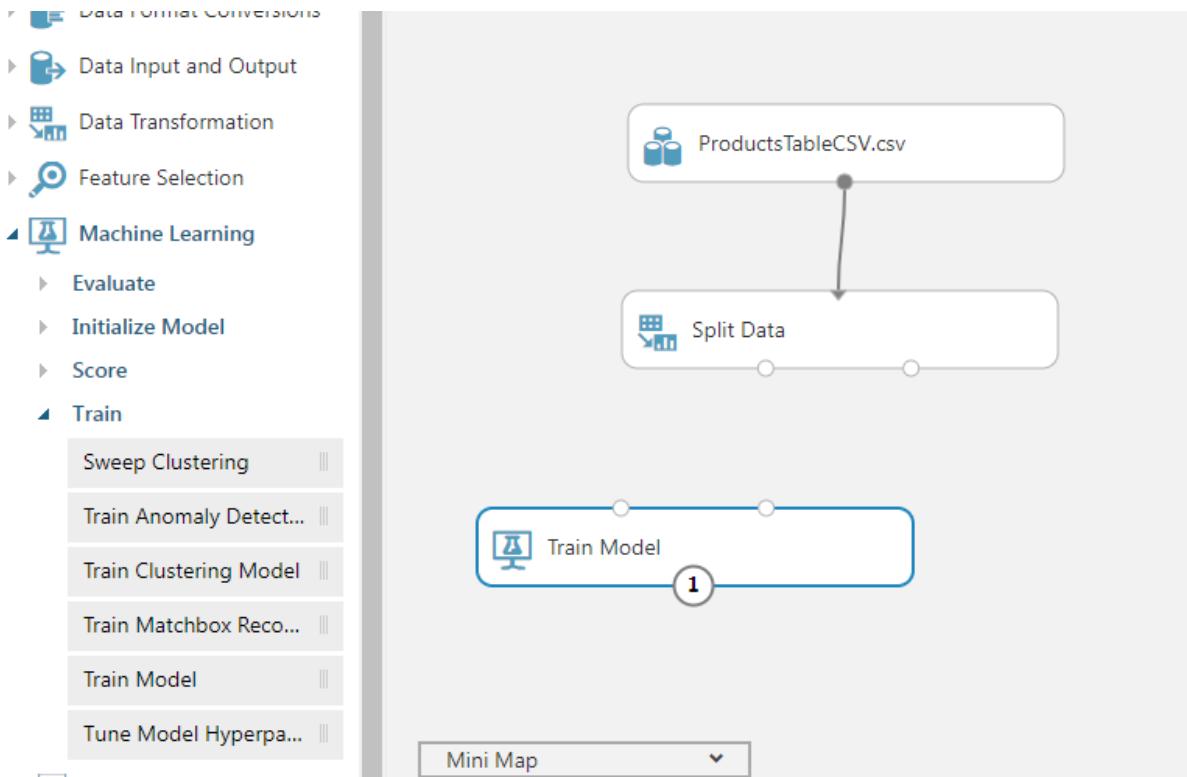
5. In the right panel (while the Split Data item on the canvas is selected), edit the **Fraction of rows in the first output dataset** to **0.7**. This will split the data into two parts, the first part will be 70% of the data, and the second part will be the remaining 30%. To ensure that the data is split randomly, make sure the **Randomized split** checkbox remains checked.



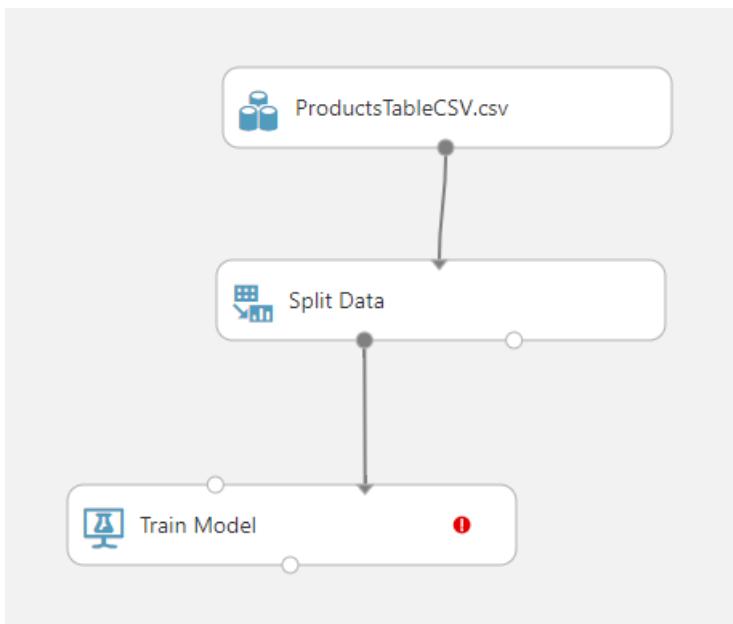
6. Drag a connection from the base of the **ProductsTableCSV** item on the canvas to the top of the Split Data item. This will connect the items and send the **ProductsTableCSV** dataset output (the data) to the Split Data input.



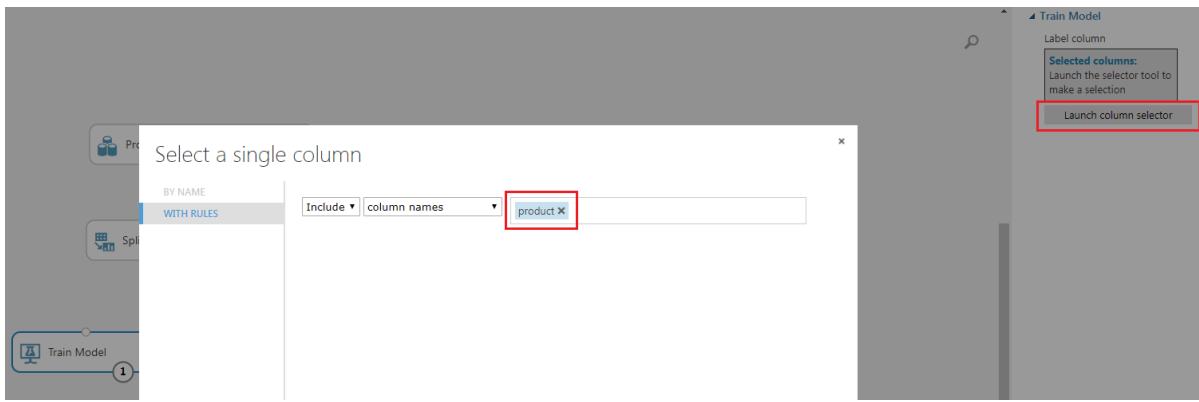
7. In the **Experiments** panel on the left side, expand *Machine Learning > Train*. Drag the **Train Model** item out in to the Experiment canvas. Your canvas should look the same as the below.



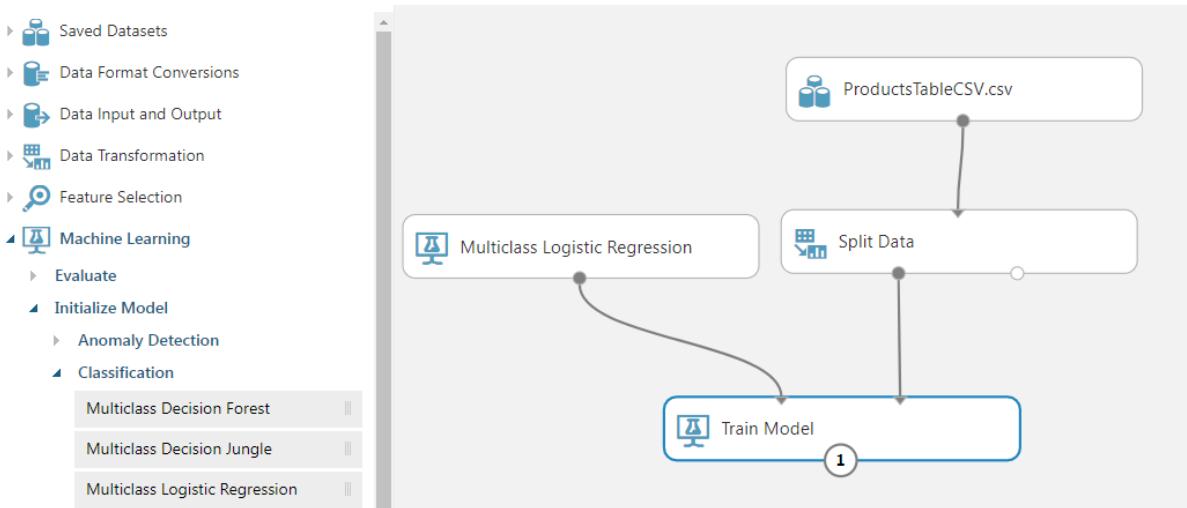
8. From the **bottom left** of the **Split Data** item drag a connection to the **top right** of the **Train Model** item.
The first 70% split from the dataset will be used by the Train Model to train the algorithm.



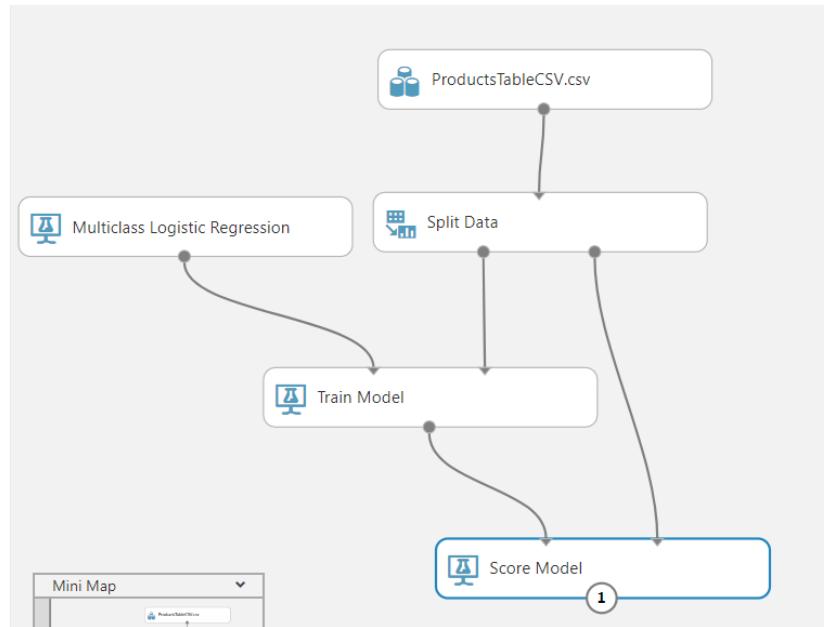
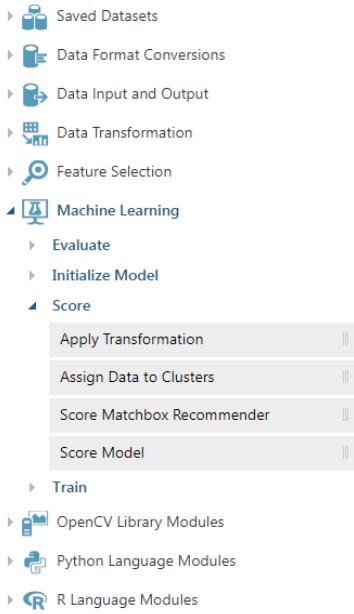
9. Select the **Train Model** item on the canvas, and in the **Properties** panel (on the right-hand side of your browser window) click the **Launch column selector** button.
10. In the text box type **product** and then press **Enter**, **product** will be set as a column to train predictions.
Following this, click on the **tick** in the bottom-right corner to close the selection dialog.



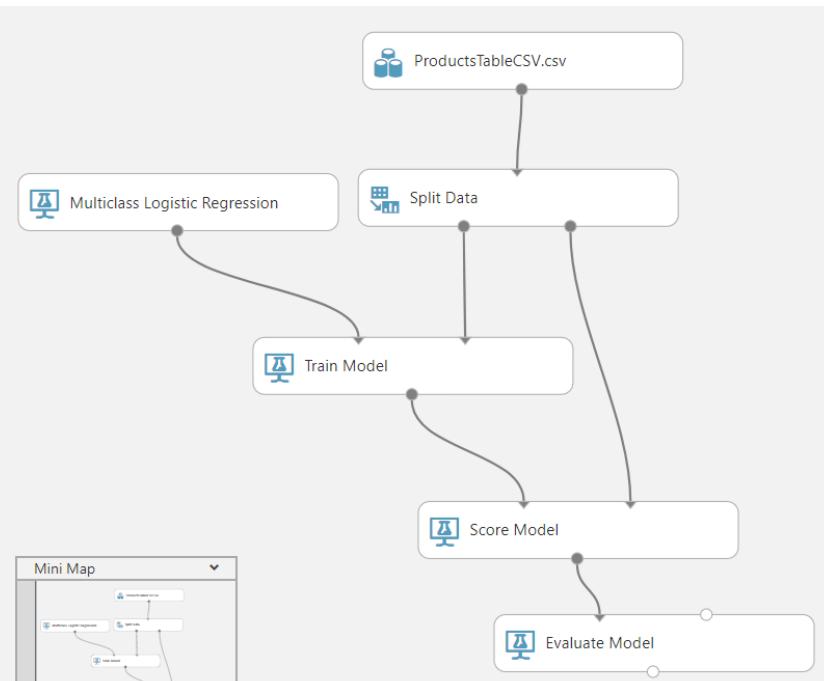
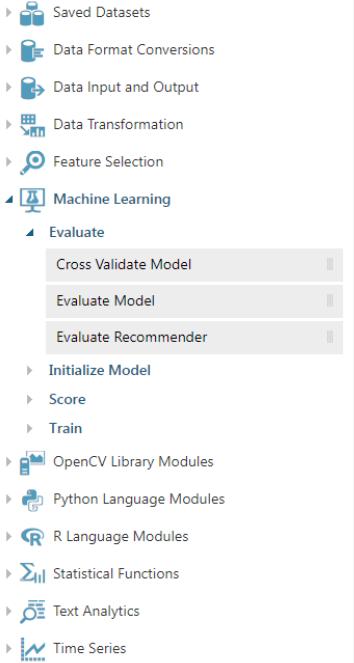
11. You are going to train a **Multiclass Logistic Regression** algorithm to predict the most sold **product** based on the hour of the day and the date. It is beyond the scope of this document to explain the details of the different algorithms provided by the Azure Machine Learning studio, though, you can find out more from the [Machine Learning Algorithm Cheat Sheet](#)
12. From the experiment items panel on the left, expand **Machine Learning > Initialize Model > Classification**, and drag the **Multiclass Logistic Regression** item on to the experiment canvas.
13. Connect the output, from the bottom of the **Multiclass Logistic Regression**, to the top-left input of the **Train Model** item.



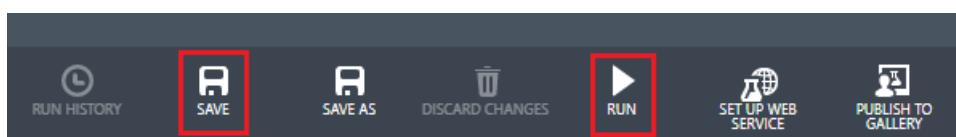
14. In list of experiment items in the panel on the left, expand **Machine Learning > Score**, and drag the **Score Model** item on to the canvas.
15. Connect the output, from the bottom of the **Train Model**, to the top-left input of the **Score Model**.
16. Connect the bottom-right output from **Split Data**, to the top-right input of the **Score Model** item.



17. In the list of **Experiment** items in the panel on the left, expand **Machine Learning > Evaluate**, and drag the **Evaluate Model** item onto the canvas.
18. Connect the output from the **Score Model** to the top-left input of the **Evaluate Model**.



19. You have built your first Machine Learning Experiment. You can now save and run the experiment. In the menu at the bottom of the page, click on the **Save** button to save your experiment and then click **Run** to the start the experiment.

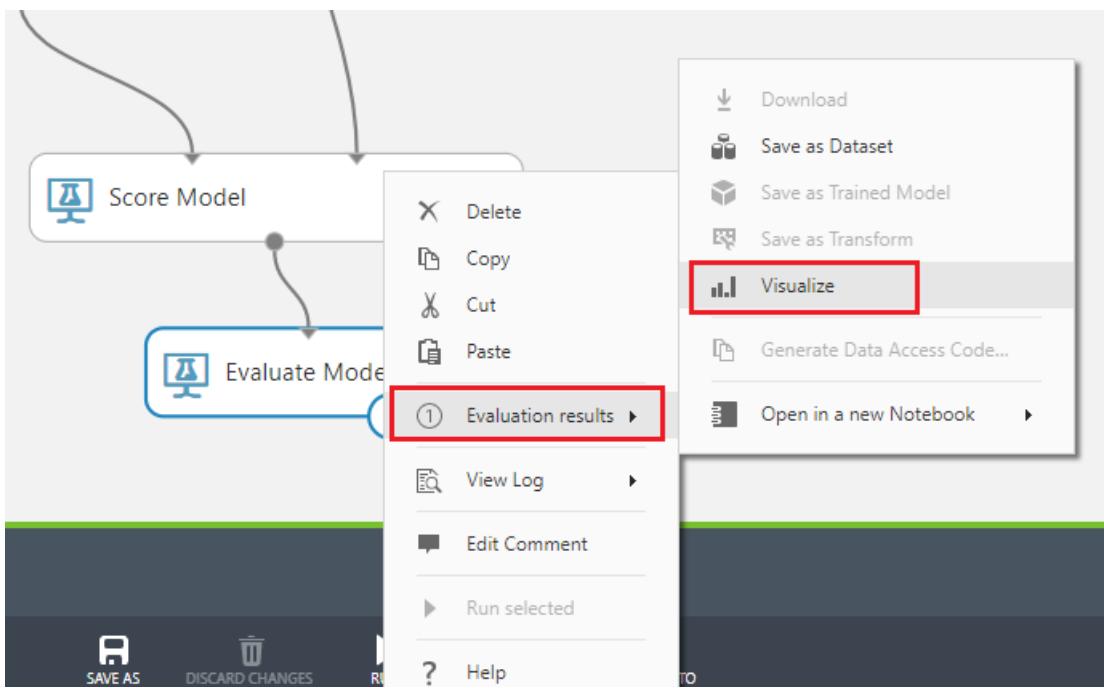


20. You can see the **status** of the experiment in the top-right of the canvas. Wait a few moments for the experiment to finish.

If you have a big (real world) dataset it is likely that the experiment could take hours to run.

The screenshot shows the Microsoft Azure Machine Learning Studio interface. At the top, there's a header bar with the title 'Microsoft Azure Machine Learning Studio', the experiment name 'MyNewMLSW', and status information 'Running (0:00:14)'. Below the header is the main canvas where a 'Score Model' and an 'Evaluate Model' component are connected. A context menu is open over the 'Evaluate Model' component, with the 'Evaluation results' option highlighted with a red box. The 'Visualize' option within this submenu is also highlighted with a red box.

21. Right click on the **Evaluate Model** item in the canvas and from the context menu hover the mouse over **Evaluation Results**, then select **Visualize**.

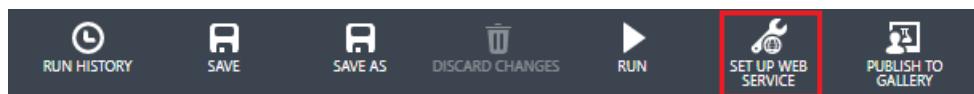


22. The evaluation results will be displayed showing the predicted outcomes versus the actual outcomes. This uses the 30% of the original dataset, that was split earlier, for evaluating the model. You can see the results are not great, ideally you would have the highest number in each row be the highlighted item in the columns.

	chopping...	cup	cutlery	knife	ladle	plate	saltpepper	spatula	woodensp...
Actual Class	chopping...		22.3%	9.5%	9.5%		16.8%	41.9%	
	cup	0.6%	23.8%	10.5%	7.2%		13.3%	44.6%	
	cutlery	0.6%	22.5%	9.2%	8.3%		14.2%	45.2%	
	knife	1.2%	28.2%	9.3%	7.6%		17.7%	36.0%	
	ladle	0.6%	24.6%	7.4%	9.2%		12.5%	45.7%	
	plate		21.5%	7.9%	6.5%		14.4%	49.7%	
	saltpepper		24.6%	7.8%	10.6%		14.7%	42.3%	
	spatula	0.6%	18.5%	10.6%	5.2%		19.4%	45.8%	
	woodensp...	0.9%	19.0%	9.9%	8.0%		13.9%	48.3%	

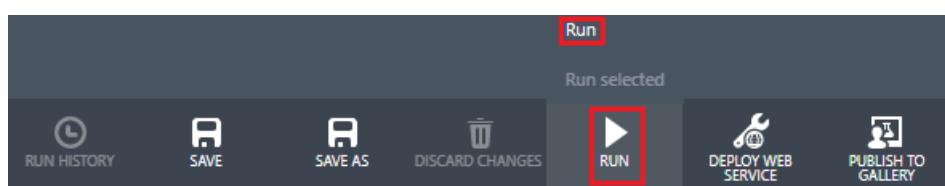
23. Close the **Results**.

24. To use your newly trained Machine Learning model you need to expose it as a **Web Service**. To do this, click on the **Set Up Web Service** menu item in the menu at the bottom of the page, and click on **Predictive Web Service**.

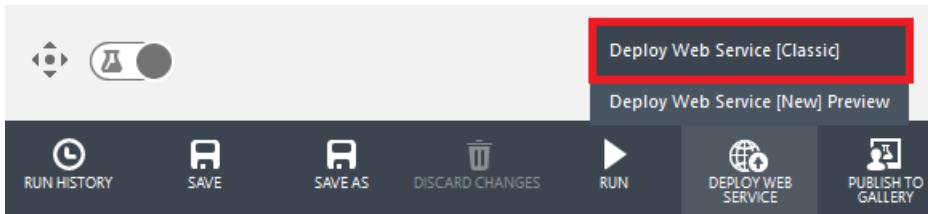


25. A new tab will be created, and the train model merged to create the new web service.

26. In the menu at the bottom of the page click **Save**, then click **Run**. You will see the status updated in the top-right corner of the experiment canvas.



27. Once it has finished running, a **Deploy Web Service** button will appear at the bottom of the page. You are ready to deploy the web service. Click **Deploy Web Service (Classic)** in the menu at the bottom of the page.



Your browser may prompt to allow a pop-up, which you should **allow**, though you may need to press **Deploy Web Service** again, if the deploy page does not show.

28. Once the Experiment has been created you will be redirected to a **Dashboard** page where you will have your **API Key** displayed. Copy it into a notepad for the moment, you will need it in your code very soon. Once you have noted your API Key, click on the **REQUEST/RESPONSE** button in the **Default Endpoint** section underneath the Key.

A screenshot of the Microsoft Azure Machine Learning Studio Dashboard page. On the left is a sidebar with icons for experiment, general, published experiment, description, API key, default endpoint, and batch execution. The main area shows an experiment created on 07/03/2018 [predictive exp.]. It includes sections for General (with a New Web Services Experience preview link), Published experiment (with View snapshot and View latest links), Description (with a note about no description provided), API key (with a redacted value), Default Endpoint (with API HELP PAGE and TEST tabs, and a REQUEST/RESPONSE button highlighted with a red box), and Batch Execution (with Test and Test preview links).

experiment created on 07/03/2018 [predictive exp.]

DASHBOARD CONFIGURATION

General New Web Services Experience [preview](#)

Published experiment

[View snapshot](#) [View latest](#)

Description

No description provided for this web service.

API key

+x5n5zXd5CK1fFUsFlu[REDACTED]

Default Endpoint

API HELP PAGE TEST

[REQUEST/RESPONSE](#) [Test](#) [Test preview](#)

BATCH EXECUTION [Test preview](#)

NOTE

If you click Test in this page, you will be able to enter input data and view the output. Enter the **day** and **hour**. Leave the **product** entry blank. Then click the **Confirm** button. The output on the bottom of the page will show the JSON representing the likelihood of each product being the choice.

29. A new web page will open up, displaying the instructions and some examples about the Request structure required by the Machine Learning Studio. Copy the **Request URI** displayed in this page, into your notepad.

Request Response API Documentation for Experiment created on 07/03/2018 [Predictive Exp.]

Updated: 03/07/2018 23:11

No description provided for this web service.

- [Previous version of this API](#)
- [Submit a request](#)
- [Input Parameters](#)
- [Output Parameters](#)
- [Web App Template for RRS](#)
- [Sample Code](#)
- [API Swagger Document](#) ⓘ
- [Endpoint Management Swagger Document](#) ⓘ

Request

Method	Request URI	HTTP Version
POST	<code>https://ussouthcentral.services.azureml.net/workspaces/7ac211111111111111111111111111149a09542f/services/bc75411111111111111111111111111e7397/exec?api-version=2.0&details=true</code>	HTTP/1.1

Note: You may omit the **details** parameter from the query string. This would cause **ColumnTypes** to be omitted from the output

Request Headers

Request Header	Description
<code>Authorization:Bearer abc123</code>	Required. Pass the API Key here. Obtain this key from the publisher of the API.
<code>Content-Length</code>	Required. The length of the content body.
<code>Content-Type:application/json</code>	Required if the request body is sent in JSON format

You have now built a machine learning system that provides the most likely product to be sold based on historical purchasing data, correlated with the time of the day and day of the year.

To call the web service, you will need the URL for the service endpoint and an API Key for the service. Click on the **Consume** tab, from the top menu.

The **Consumption** Info page will display the information you will need to call the web service from your code. Take a copy of the **Primary Key** and the **Request-Response** URL. You will need these in the next Chapter.

Chapter 5 - Setting up the Unity Project

Set up and test your Mixed Reality Immersive Headset.

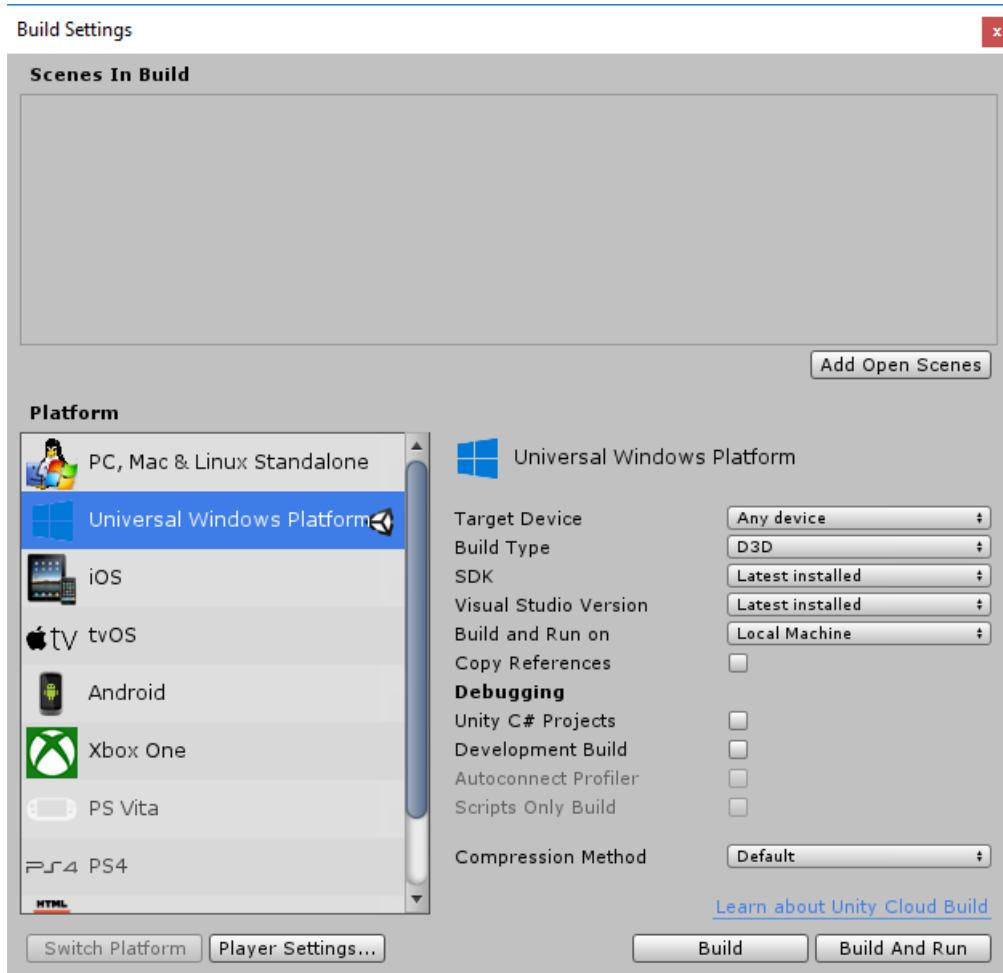
NOTE

You will **not** require Motion Controllers for this course. If you need support setting up the Immersive Headset, please click [HERE](#).

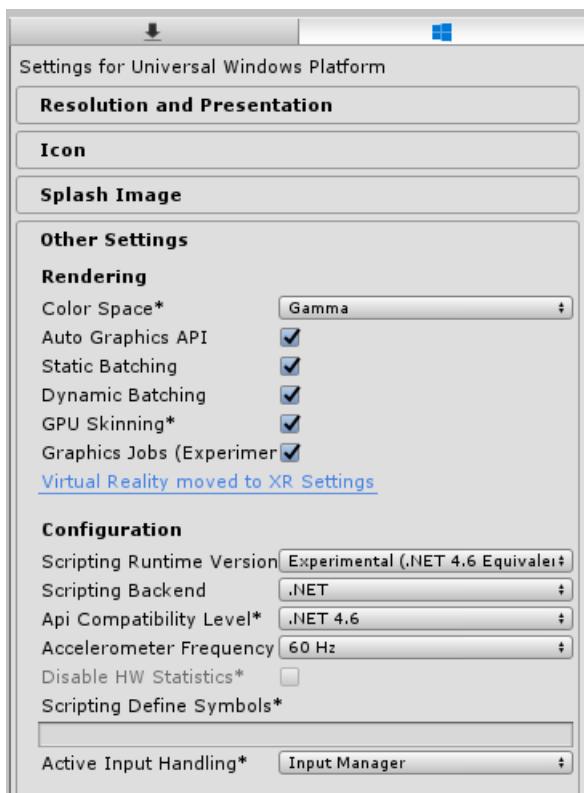
1. Open **Unity** and create a new Unity Project called **MR_MachineLearning**. Make sure the project type is set to **3D**.
2. With Unity open, it is worth checking the default **Script Editor** is set to **Visual Studio**. Go to **Edit > Preferences** and then from the new window, navigate to **External Tools**. Change **External Script Editor** to **Visual Studio 2017**. Close the **Preferences** window.
3. Next, go to **File > Build Settings** and switch the platform to **Universal Windows Platform**, by clicking on the **Switch Platform** button.
4. Also make sure that:
 - a. **Target Device** is set to **Any Device**.

For the Microsoft HoloLens, set **Target Device** to **HoloLens**.

- b. **Build Type** is set to **D3D**.
- c. **SDK** is set to **Latest installed**.
- d. **Visual Studio Version** is set to **Latest installed**.
- e. **Build and Run** is set to **Local Machine**.
- f. Do not worry about setting up **Scenes** right now, as these are provided later.
- g. The remaining settings should be left as default for now.

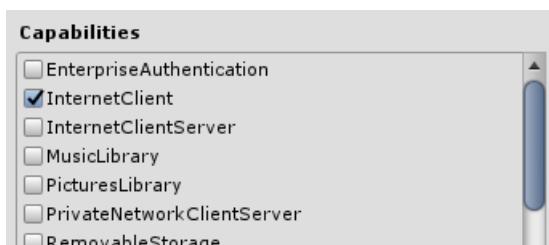


5. In the **Build Settings** window, click on the **Player Settings** button, this will open the related panel in the space where the **Inspector** is located.
6. In this panel, a few settings need to be verified:
 - a. In the **Other Settings** tab:
 - a. **Scripting Runtime Version** should be **Experimental (.NET 4.6 Equivalent)**
 - b. **Scripting Backend** should be **.NET**
 - c. **API Compatibility Level** should be **.NET 4.6**

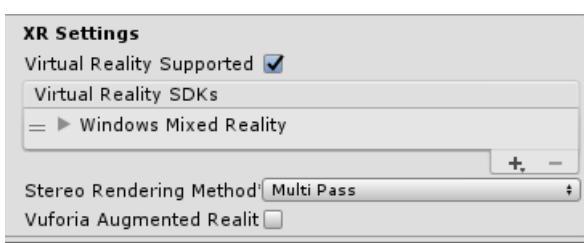


b. Within the **Publishing Settings** tab, under **Capabilities**, check:

- **InternetClient**



c. Further down the panel, in **XR Settings** (found below **Publish Settings**), tick **Virtual Reality Supported**, make sure the **Windows Mixed Reality SDK** is added



7. Back in **Build Settings** Unity C# Projects is no longer greyed out; tick the checkbox next to this.

8. Close the Build Settings window.

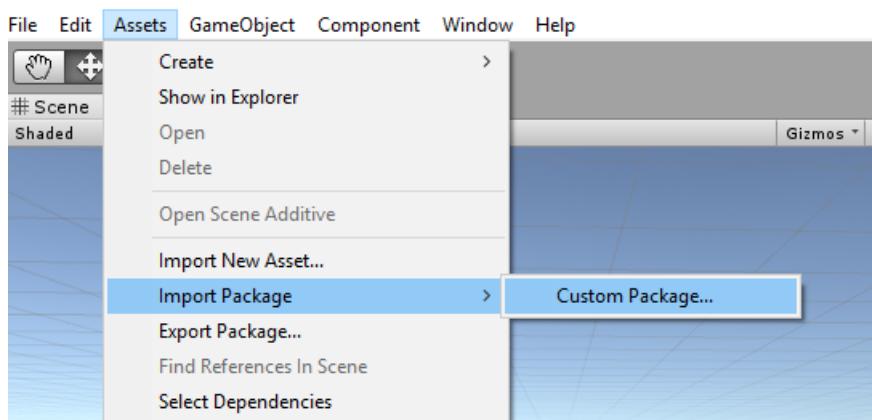
9. Save your Project (**FILE > SAVE PROJECT**).

Chapter 6 - Importing the MLProducts Unity Package

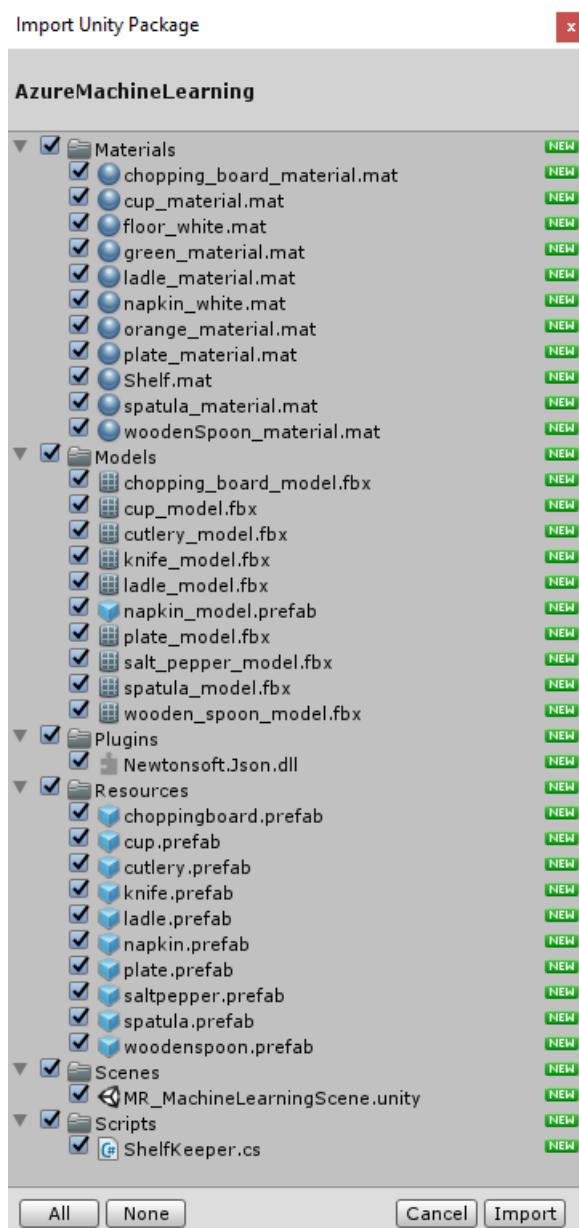
For this course, you will need to download a Unity Asset Package called [Azure-MR-307.unitypackage](#). This package comes complete with a scene, with all objects in that prebuilt, so you can focus on getting it all working. The **ShelfKeeper** script is provided, though only holds the public variables, for the purpose of scene setup structure. You will need to do all other sections.

To import this package:

- With the Unity dashboard in front of you, click on **Assets** in the menu at the top of the screen, then click on **Import Package, Custom Package**.



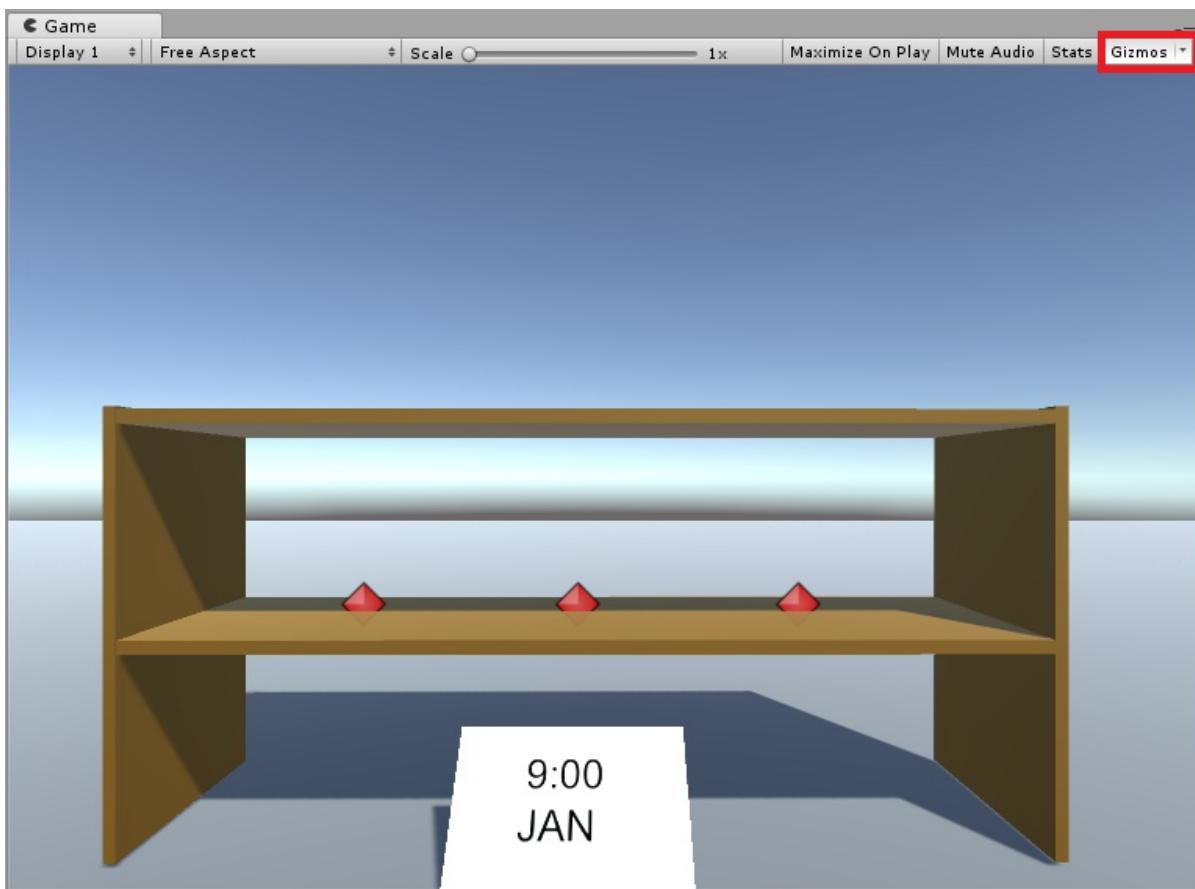
- Use the file picker to select the **Azure-MR-307.unitypackage** package and click **Open**.
- A list of components for this asset will be displayed to you. Confirm the import by clicking **Import**.



- Once it has finished importing, you will notice that some new folders have appeared in your Unity **Project Panel**. Those are the 3D models and the respective materials that are part of the pre-made scene you will work on. You will write the majority of the code in this course.



- Within the **Project Panel** folder, click on the **Scenes** folder and double click on the scene inside (called **MR_MachineLearningScene**). The scene will open (see image below). If the red diamonds are missing, simply click the **Gizmos** button, at the top right of the **Game Panel**.

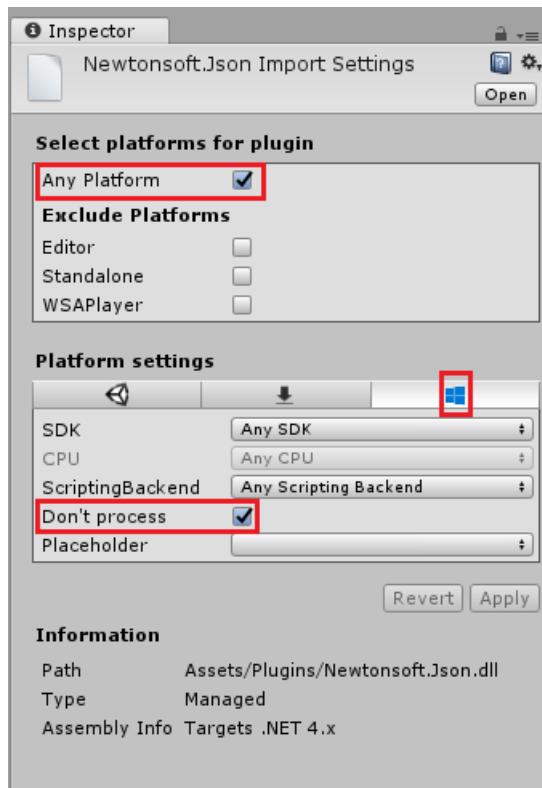


Chapter 7 - Checking the DLLs in Unity

To leverage the use of JSON libraries (used for serializing and deserializing), a Newtonsoft DLL has been implemented with the package you brought in. The library should have the correct configuration, though it is worth checking (particularly if you are having issues with code not working).

To do so:

- Left-click on the Newtonsoft file inside the Plugins folder and look at the **Inspector panel**. Make sure **Any Platform** is ticked. Go to the **UWP tab** and also ensure **Don't process** is ticked.



Chapter 8 - Create the ShelfKeeper class

The **ShelfKeeper** class hosts methods that control the UI and products spawned in the scene.

As part of the imported package, you will have been given this class, though it is incomplete. It is now time to complete that class:

1. Double click on the **ShelfKeeper** script, within the **Scripts** folder, to open it with **Visual Studio 2017**.
2. Replace all the code existing in the script with the following code, which sets the time and date and has a method to show a product.

```

using UnityEngine;

public class ShelfKeeper : MonoBehaviour
{
    /// <summary>
    /// Provides this class Singleton-like behavior
    /// </summary>
    public static ShelfKeeper instance;

    /// <summary>
    /// Unity Inspector accessible Reference to the Text Mesh object needed for data
    /// </summary>
    public TextMesh dateText;

    /// <summary>
    /// Unity Inspector accessible Reference to the Text Mesh object needed for time
    /// </summary>
    public TextMesh timeText;

    /// <summary>
    /// Provides references to the spawn locations for the products prefabs
    /// </summary>
    public Transform[] spawnPoint;

    private void Awake()
    {
        instance = this;
    }

    /// <summary>
    /// Set the text of the date in the scene
    /// </summary>
    public void SetDate(string day, string month)
    {
        dateText.text = day + " " + month;
    }

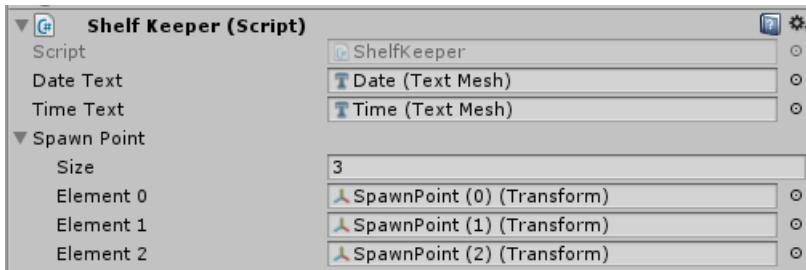
    /// <summary>
    /// Set the text of the time in the scene
    /// </summary>
    public void SetTime(string hour)
    {
        timeText.text = hour + ":00";
    }

    /// <summary>
    /// Spawn a product on the shelf by providing the name and selling grade
    /// </summary>
    /// <param name="name"></param>
    /// <param name="sellingGrade">0 being the best seller</param>
    public void SpawnProduct(string name, int sellingGrade)
    {
        Instantiate(Resources.Load(name),
                    spawnPoint[sellingGrade].transform.position, spawnPoint[sellingGrade].transform.rotation);
    }
}

```

3. Be sure to save your changes in **Visual Studio** before returning to **Unity**.

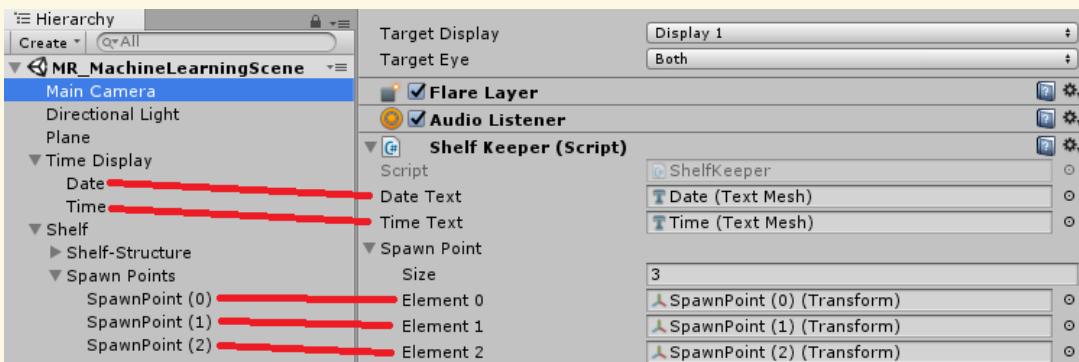
4. Back in the Unity Editor, check that the **ShelfKeeper** class looks like the below:



IMPORTANT

If your script does not have the reference targets (i.e. **Date (Text Mesh)**), simply drag the corresponding objects from the **Hierarchy Panel**, into the target fields. See below for explanation, if needed:

1. Open the **Spawn Point** array within the **ShelfKeeper** component script by left-clicking it. A sub-section will appear called **Size**, which indicates the size of the array. Type **3** into the textbox next to **Size** and press **Enter**, and three slots will be created beneath.
2. Within the **Hierarchy** expand the **Time Display** object (by left-clicking the arrow beside it). Next click the **Main Camera** from within the **Hierarchy**, so that the **Inspector** shows its information.
3. Select the **Main Camera** in the **Hierarchy Panel**. Drag the **Date** and **Time** objects from the **Hierarchy Panel** to the **Date Text** and **Time Text** slots within the **Inspector** of the **Main Camera** in the **ShelfKeeper** component.
4. Drag the **Spawn Points** from the **Hierarchy Panel** (beneath the **Shelf** object) to the **3 Element** reference targets beneath the **Spawn Point** array, as shown in the image.



Chapter 9 - Create the ProductPrediction class

The next class you are going to create is the **ProductPrediction** class.

This class is responsible for:

- Querying the **Machine Learning Service** instance, providing the current date and time.
- Deserializing the JSON response into usable data.
- Interpreting the data, retrieving the 3 recommended products.
- Calling the **ShelfKeeper** class methods to display the data in the Scene.

To create this class:

1. Go to the **Scripts** folder, in the **Project Panel**.
2. Right-click inside the folder, **Create > C# Script**. Call the script **ProductPrediction**.
3. Double click on the new **ProductPrediction** script to open it with **Visual Studio 2017**.

4. If the **File Modification Detected** dialog pops up, click *Reload Solution.

5. Add the following namespaces to the top of the ProductPrediction class:

```
using System;
using System.Collections.Generic;
using UnityEngine;
using System.Linq;
using Newtonsoft.Json;
using UnityEngine.Networking;
using System.Runtime.Serialization;
using System.Collections;
```

6. Inside the **ProductPrediction** class insert the following two objects which are composed of a number of nested classes. These classes are used to serialize and deserialize the JSON for the Machine Learning Service.

```
/// <summary>
/// This object represents the Prediction request
/// It host the day of the year and hour of the day
/// The product must be left blank when serialising
/// </summary>
public class RootObject
{
    public Inputs Inputs { get; set; }
}

public class Inputs
{
    public Input1 input1 { get; set; }
}

public class Input1
{
    public List<string> ColumnNames { get; set; }
    public List<List<string>> Values { get; set; }
}
```

```

/// <summary>
/// This object containing the deserialised Prediction result
/// It host the list of the products
/// and the likelihood of them being sold at current date and time
/// </summary>
public class Prediction
{
    public Results Results { get; set; }
}

public class Results
{
    public Output1 output1;
}

public class Output1
{
    public string type;
    public Value value;
}

public class Value
{
    public List<string> ColumnNames { get; set; }
    public List<List<string>> Values { get; set; }
}

```

7. Then add the following variables above the previous code (so that the JSON related code is at the bottom of the script, below all other code, and out of the way):

```
/// <summary>
/// The 'Primary Key' from your Machine Learning Portal
/// </summary>
private string authKey = "-- Insert your service authentication key here --";

/// <summary>
/// The 'Request-Response' Service Endpoint from your Machine Learning Portal
/// </summary>
private string serviceEndpoint = "-- Insert your service endpoint here --";

/// <summary>
/// The Hour as set in Windows
/// </summary>
private string thisHour;

/// <summary>
/// The Day, as set in Windows
/// </summary>
private string thisDay;

/// <summary>
/// The Month, as set in Windows
/// </summary>
private string thisMonth;

/// <summary>
/// The Numeric Day from current Date Conversion
/// </summary>
private string dayOfTheYear;

/// <summary>
/// Dictionary for holding the first (or default) provided prediction
/// from the Machine Learning Experiment
/// </summary>
private Dictionary<string, string> predictionDictionary;

/// <summary>
/// List for holding product prediction with name and scores
/// </summary>
private List<KeyValuePair<string, double>> keyValueList;
```

IMPORTANT

Make sure to insert the **primary key** and **request-response endpoint**, from the Machine Learning Portal, into the variables here. The below images show where you would have taken the key and endpoint from.

The screenshot shows the Microsoft Azure Machine Learning Studio interface. In the center, there's a message: "experiment created on 07/03/2018 [predictive exp.]". Below it, under "DASHBOARD", there are links for "General", "New Web Services Experience", "Published experiment", "View snapshot", and "View latest". A "Description" section states "No description provided for this web service." Under "API key", a value is shown: "+xNzXG5CK1FU5zFl...". Under "Default Endpoint", there are "TEST" and "BATCH EXECUTION" buttons. A red box highlights the "REQUEST RESPONSE" button, which is located between the TEST and BATCH EXECUTION buttons.

Request Response API Documentation for Experiment created on 07/03/2018 [Predictive Exp.]

Updated: 03/07/2018 23:11

No description provided for this web service.

- [Previous version of this API](#)
- [Submit a request](#)
- [Issue Parameters](#)
- [Output Parameters](#)
- [Web App Template for RBS](#)
- [Sample Code](#)
- [API Swagger Document](#) ⓘ
- [Endpoint Management Swagger Document](#) ⓘ

Request

The screenshot shows the "Request" section of the API documentation. It includes a "Method" dropdown set to "Request URI" and a "HTTP Version" dropdown set to "HTTP/1.1". The "Request URI" field contains the URL: "https://usouthcentral.services.azureml.net/workspaces/7ac.../services/be75a.../execute?api-version=2.0&details=true". A note below says: "Note: You may omit the **details** parameter from the query string. This would cause **ColumnTypes** to be omitted from the output." Below this, there are sections for "Request Headers" and "Request Body".

8. Insert this code within the **Start()** method. The **Start()** method is called when the class initializes:

```
void Start()
{
    // Call to get the current date and time as set in Windows
    GetTodayDateAndTime();

    // Call to set the HOUR in the UI
    ShelfKeeper.instance.SetTime(thisHour);

    // Call to set the DATE in the UI
    ShelfKeeper.instance.SetDate(thisDay, thisMonth);

    // Run the method to Get Predication from Azure Machine Learning
    StartCoroutine(GetPrediction(thisHour, dayOfTheYear));
}
```

9. The following is the method that collects the date and time from Windows and converts it into a format that our Machine Learning Experiment can use to compare with the data stored in the table.

```
/// <summary>
/// Get current date and hour
/// </summary>
private void GetTodayDateAndTime()
{
    // Get today date and time
    DateTime todayDate = DateTime.Now;

    // Extrapolate the HOUR
    thisHour = todayDate.Hour.ToString();

    // Extrapolate the DATE
    thisDay = todayDate.Day.ToString();
    thisMonth = todayDate.ToString("MMM");

    // Extrapolate the day of the year
    dayOfTheYear = todayDate.DayOfYear.ToString();
}
```

10. You can **delete** the **Update()** method since this class will not use it.
11. Add the following method which will communicate the current date and time to the Machine Learning endpoint and receive a response in JSON format.

```

private IEnumerator GetPrediction(string timeOfDay, string dayOfYear)
{
    // Populate the request object
    // Using current day of the year and hour of the day
    RootObject ro = new RootObject
    {
        Inputs = new Inputs
        {
            input1 = new Input1
            {
                ColumnNames = new List<string>
                {
                    "day",
                    "hour",
                    "product"
                },
                Values = new List<List<string>>()
            }
        }
    };

    List<string> l = new List<string>
    {
        dayOfYear,
        timeOfDay,
        ""
    };

    ro.Inputs.input1.Values.Add(l);

    Debug.LogFormat("Score request built");

    // Serialise the request
    string json = JsonConvert.SerializeObject(ro);

    using (UnityWebRequest www = UnityWebRequest.Post(serviceEndpoint, "POST"))
    {
        byte[] jsonToSend = new System.Text.UTF8Encoding().GetBytes(json);
        www.uploadHandler = new UploadHandlerRaw(jsonToSend);

        www.downloadHandler = new DownloadHandlerBuffer();
        www.SetRequestHeader("Authorization", "Bearer " + authKey);
        www.SetRequestHeader("Content-Type", "application/json");
        www.SetRequestHeader("Accept", "application/json");

        yield return www.SendWebRequest();
        string response = www.downloadHandler.text;

        // Deserialize the response
        DataContractSerializer serializer;
        serializer = new DataContractSerializer(typeof(string));
        DeserialiseJsonResponse(response);
    }
}

```

12. Add the following method, which is responsible for deserializing the JSON response, and communicating the result of the deserialization to the **ShelfKeeper** class. This result will be the names of the three items predicted to sell the most at current date and time. Insert the code below into the **ProductPrediction** class, below the previous method.

```

/// <summary>
/// Deserialize the response received from the Machine Learning portal
/// </summary>
public void DeserialiseJsonResponse(string jsonResponse)
{
    // Deserialize JSON
    Prediction prediction = JsonConvert.DeserializeObject<Prediction>(jsonResponse);
    predictionDictionary = new Dictionary<string, string>();

    for (int i = 0; i < prediction.Results.output1.value.ColumnNames.Count; i++)
    {
        if (prediction.Results.output1.value.Values[0][i] != null)
        {
            predictionDictionary.Add(prediction.Results.output1.value.ColumnNames[i],
prediction.Results.output1.value.Values[0][i]);
        }
    }

    keyValueList = new List<KeyValuePair<string, double>>();

    // Strip all non-results, by adding only items of interest to the scoreList
    for (int i = 0; i < predictionDictionary.Count; i++)
    {
        KeyValuePair<string, string> pair = predictionDictionary.ElementAt(i);
        if (pair.Key.StartsWith("Scored Probabilities"))
        {
            // Parse string as double then simplify the string key so to only have the item name
            double scorefloat = 0f;
            double.TryParse(pair.Value, out scorefloat);
            string simplifiedName =
                pair.Key.Replace("\\", "").Replace("Scored Probabilities for Class", "").Trim();
            keyValueList.Add(new KeyValuePair<string, double>(simplifiedName, scorefloat));
        }
    }

    // Sort Predictions (results will be lowest to highest)
    keyValueList.Sort((x, y) => y.Value.CompareTo(x.Value));

    // Spawn the top three items, from the keyValueList, which we have sorted
    for (int i = 0; i < 3; i++)
    {
        ShelfKeeper.instance.SpawnProduct(keyValueList[i].Key, i);
    }

    // Clear lists in case of reuse
    keyValueList.Clear();
    predictionDictionary.Clear();
}

```

13. Save **Visual Studio** and head back to **Unity**.

14. Drag the **ProductPrediction** class script from the **Script** folder, onto the **Main Camera** object.

15. Save your scene and project **File > Save Scene / File > Save Project**.

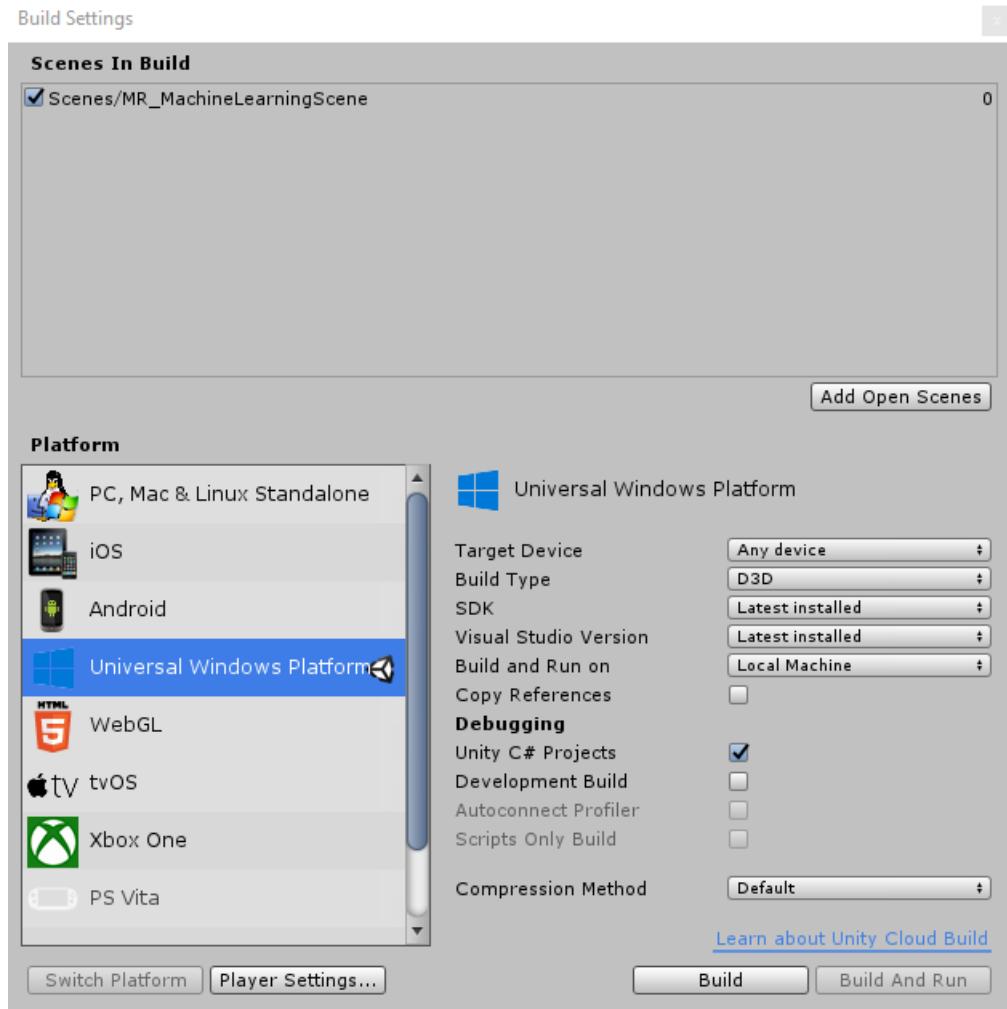
Chapter 10 - Build the UWP Solution

It is now time to build your project as a UWP solution, so that it can run as a standalone application.

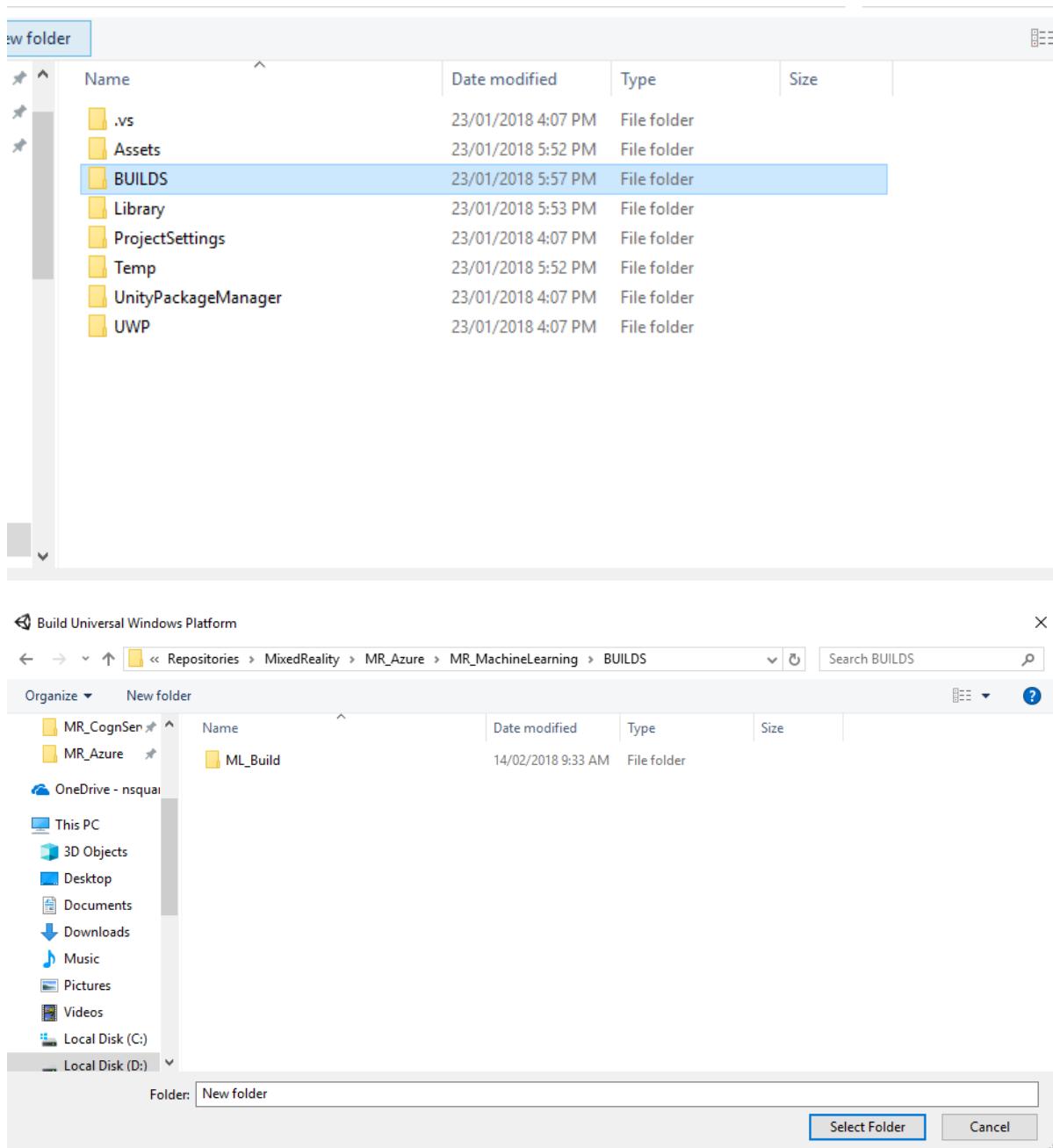
To Build:

1. Save the current scene by clicking on **File Save Scenes**.
2. Go to **File Build Settings**

3. Check the box called **Unity C# Projects** (this is important because it will allow you to edit the classes after build is completed).
4. Click on **Add Open Scenes**,
5. Click **Build**.



6. You will be prompted to select the folder where you want to build the Solution.
7. Create a **BUILDS** folder and within that folder create another folder with an appropriate name of your choice.
8. Click your new folder and then click **Select Folder**, to begin the build at that location.

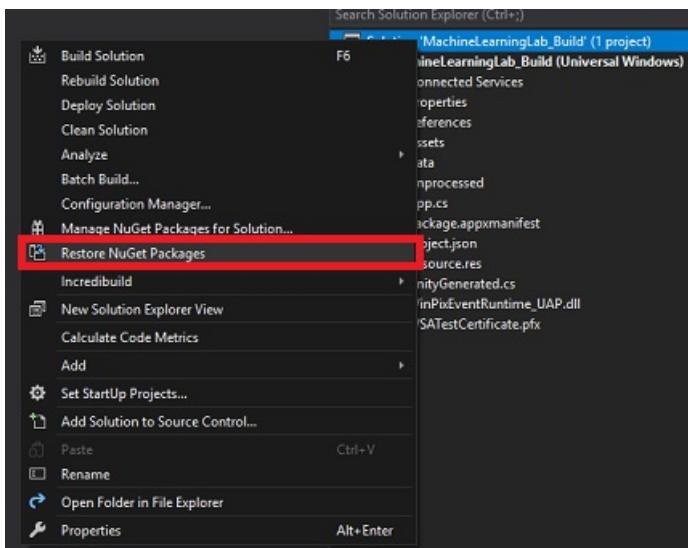


- Once Unity has finished building (it might take some time), it will open a **File Explorer** window at the location of your build (check your task bar, as it may not always appear above your windows, but will notify you of the addition of a new window).

Chapter 11 - Deploy your Application

To deploy your application:

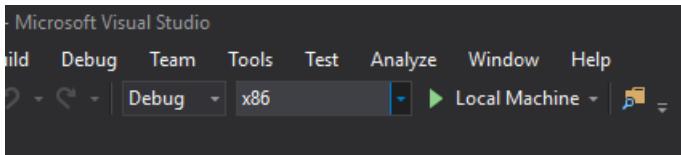
1. Navigate to your new Unity build (the **App** folder) and open the solution file with **Visual Studio**.
2. With Visual Studio open, you need to Restore NuGet Packages, which can be done through right-clicking your MachineLearningLab_Build solution, from the Solution Explorer (found to the right of Visual Studio), and then clicking Restore NuGet Packages:



3. In the Solution Configuration select **Debug**.
4. In the Solution Platform, select **x86, Local Machine**.

For the Microsoft HoloLens, you may find it easier to set this to *Remote Machine*, so that you are not tethered to your computer. Though, you will need to also do the following:

- Know the **IP Address** of your HoloLens, which can be found within the *Settings > Network & Internet > Wi-Fi > Advanced Options*; the IPv4 is the address you should use.
- Ensure **Developer Mode** is **On**; found in *Settings > Update & Security > For developers*.

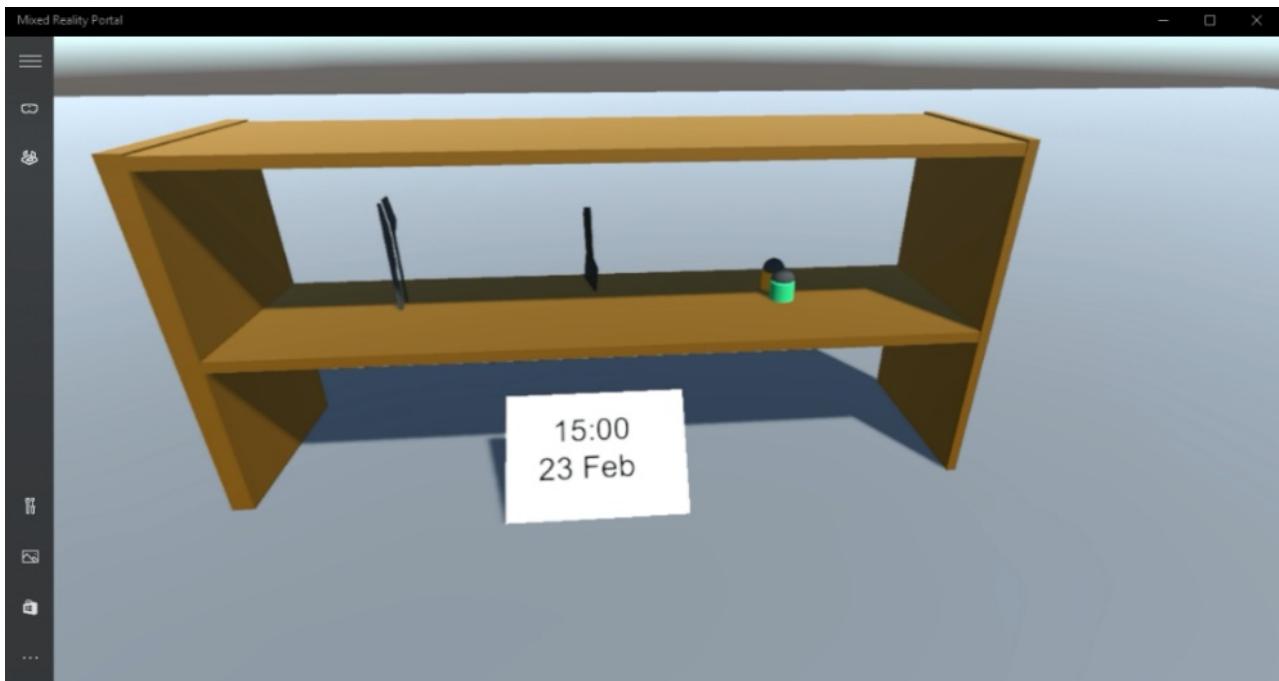


5. Go to **Build menu** and click on **Deploy Solution** to sideload the application to your PC.
6. Your App should now appear in the list of installed apps, ready to be launched.

When you run the Mixed Reality application, you will see the bench that was set up in your Unity scene, and from initialization, the data you set up within Azure will be fetched. The data will be deserialized within your application, and the three top results for your current date and time will be provided visually, as three models on the bench.

Your finished Machine Learning application

Congratulations, you built a mixed reality app that leverages the Azure Machine Learning to make data predictions and display it on your scene.



Exercise

Exercise 1

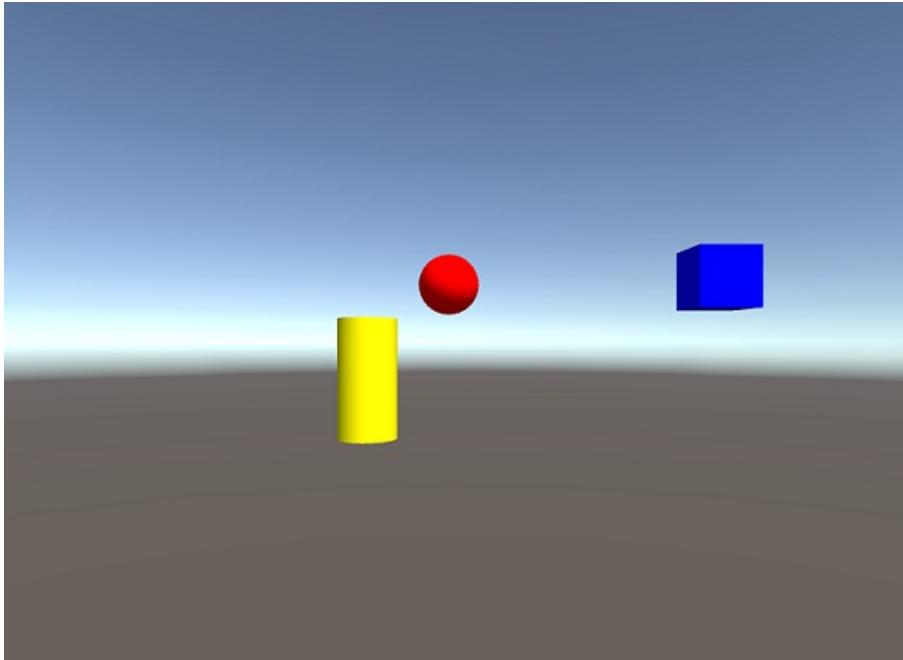
Experiment with the sort order of your application and have the three bottom predictions appear on the shelf, as this data would potentially be useful also.

Exercise 2

Using **Azure Tables** populate a new table with weather information and create a new experiment using the data.

MR and Azure 308: Cross-device notifications

11/13/2018 • 44 minutes to read • [Edit Online](#)



In this course, you will learn how to add Notification Hubs capabilities to a mixed reality application using Azure Notification Hubs, Azure Tables, and Azure Functions.

Azure Notification Hubs is a Microsoft service, which allows developers to send targeted and personalized push notifications to any platform, all powered within the cloud. This can effectively allow developers to communicate with end users, or even communicate between various applications, depending on the scenario. For more information, visit the [Azure Notification Hubs page](#).

Azure Functions is a Microsoft service, which allows developers to run small pieces of code, 'functions', in Azure. This provides a way to delegate work to the cloud, rather than your local application, which can have many benefits. **Azure Functions** supports several development languages, including C#, F#, Node.js, Java, and PHP. For more information, visit the [Azure Functions page](#).

Azure Tables is a Microsoft cloud service, which allows developers to store structured non-SQL data in the cloud, making it easily accessible anywhere. The service boasts a schemaless design, allowing for the evolution of tables as needed, and thus is very flexible. For more information, visit the [Azure Tables page](#)

Having completed this course, you will have a mixed reality immersive headset application, and a Desktop PC application, which will be able to do the following:

1. The Desktop PC app will allow the user to move an object in 2D space (X and Y), using the mouse.
2. The movement of objects within the PC app will be sent to the cloud using JSON, which will be in the form of a string, containing an object ID, type, and transform information (X and Y coordinates).
3. The mixed reality app, which has an identical scene to the desktop app, will receive notifications regarding object movement, from the Notification Hubs service (which has just been updated by the Desktop PC app).
4. Upon receiving a notification, which will contain the object ID, type, and transform information, the mixed reality app will apply the received information to its own scene.

In your application, it is up to you as to how you will integrate the results with your design. This course is designed

to teach you how to integrate an Azure Service with your Unity Project. It is your job to use the knowledge you gain from this course to enhance your mixed reality application. This course is a self-contained tutorial, which does not directly involve any other Mixed Reality Labs.

Device support

COURSE	HOLOLENS	IMMERSIVE HEADSETS
MR and Azure 308: Cross-device notifications	✓ <input type="checkbox"/>	✓ <input type="checkbox"/>

NOTE

While this course primarily focuses on Windows Mixed Reality immersive (VR) headsets, you can also apply what you learn in this course to Microsoft HoloLens. As you follow along with the course, you will see notes on any changes you might need to employ to support HoloLens. When using HoloLens, you may notice some echo during voice capture.

Prerequisites

NOTE

This tutorial is designed for developers who have basic experience with Unity and C#. Please also be aware that the prerequisites and written instructions within this document represent what has been tested and verified at the time of writing (May 2018). You are free to use the latest software, as listed within the [install the tools](#) article, though it should not be assumed that the information in this course will perfectly match what you'll find in newer software than what's listed below.

We recommend the following hardware and software for this course:

- A development PC, [compatible with Windows Mixed Reality](#) for immersive (VR) headset development
- [Windows 10 Fall Creators Update \(or later\)](#) with Developer mode enabled
- [The latest Windows 10 SDK](#)
- [Unity 2017.4](#)
- [Visual Studio 2017](#)
- A [Windows Mixed Reality immersive \(VR\) headset](#) or [Microsoft HoloLens](#) with Developer mode enabled
- Internet access for Azure setup and to access Notification Hubs

Before you start

- To avoid encountering issues building this project, it is strongly suggested that you create the project mentioned in this tutorial in a root or near-root folder (long folder paths can cause issues at build-time).
- You must be the owner of your Microsoft Developer Portal and your Application Registration Portal, otherwise you will not have permission to access the app in [Chapter 2](#).

Chapter 1 - Create an application on the Microsoft Developer Portal

To use the **Azure Notification Hubs** Service, you will need to create an Application on the Microsoft Developer Portal, as your application will need to be registered, so that it can send and receive notifications.

1. Log in to the [Microsoft Developer Portal](#).

You will need to log in to your Microsoft Account.

- From the Dashboard, click **Create a new app**.

The screenshot shows the Microsoft Windows Store Developer Portal. On the left, there's a sidebar with 'Dashboard', 'WINDOWS', 'Overview', and 'Products'. The main area is titled 'Overview' with a 'Create a new app' button. Below it are fields for 'Name' and 'Type'.

- A popup will appear, wherein you need to reserve a name for your new app. In the textbox, insert an appropriate name; if the chosen name is available, a tick will appear to the right of the textbox. Once you have an available name inserted, click the **Reserve product name** button to the bottom left of the popup.

The screenshot shows a dialog box titled 'Create your app by reserving a name'. It contains instructions about provisioning services like push notifications. A note says 'Make sure you have the rights to use any name you reserve. You must submit this app to the Store within one year, or you'll lose your name reservation.' A 'Learn more' link is provided. A text input field contains 'MR_NotHub_App'. To its right is a green checkmark icon and a button labeled 'Check availability'. At the bottom are 'Reserve product name' and 'Cancel' buttons.

- With the app now created, you are ready to move to the next Chapter.

Chapter 2 - Retrieve your new apps credentials

Log into the Application Registration Portal, where your new app will be listed, and retrieve the credentials which will be used to setup the **Notification Hubs Service** within the **Azure Portal**.

- Navigate to the [Application Registration Portal](#).

The screenshot shows the Microsoft Application Registration Portal. The top navigation bar includes 'Microsoft', 'Application Registration Portal', 'Tools', 'Docs', and 'Feedback'. The main area is titled 'My applications' with a 'Learn More' link. A table header row shows 'Name' and 'App ID / Client Id'. Below the table is a red warning box containing text about logging in with a Microsoft Account.

WARNING
You will need to use your Microsoft Account to Login.
This **must** be the Microsoft Account which you used in the previous [Chapter](#), with the Windows Store Developer portal.

- You will find your app under the **My applications** section. Once you have found it, click on it and you will be taken to a new page which has the app name plus **Registration**.

The screenshot shows the 'MR_NotHub_App Registration' page. The top navigation bar is identical to the previous screenshot. The main content area has a heading 'MR_NotHub_App Registration' and a sub-heading 'Click here for help integrating your application with Microsoft.'. Below is a 'Properties' section with a 'Name' field.

- Scroll down the registration page to find your **Application Secrets** section and the **Package SID** for your app. Copy both for use with setting up the **Azure Notification Hubs Service** in the next Chapter.

Application Secrets

The screenshot shows the 'Application Secrets' section of the Azure portal. At the top, there is a 'Generate New Password' button. Below it is a password input field containing 'IGnz3', which is highlighted with a red box. To the right of the password field is the text 'Version 0'. There is also a 'Copy' button next to the password field.

Platforms

The screenshot shows the 'Platforms' section of the Azure portal. It includes sections for 'Web' and 'Windows Store'. In the 'Web' section, there are checkboxes for 'Allow Implicit Flow' and 'Restrict token issuing to this app'. A note below says 'Limits the issuing of JSON Web Tokens (JWT) for your domain to exclusively this application.' Under 'Target Domain', there is a text input field labeled 'Target Domain'. In the 'Redirect URLs' section, there is a list with one item: 'ms-app://s-1-.....0328'. A red box highlights this URL. There is also a 'Link to different app' button.

Chapter 3 - Setup Azure Portal: create Notification Hubs Service

With your apps credentials retrieved, you will need to go to the Azure Portal, where you will create an Azure Notification Hubs Service.

1. Log into the [Azure Portal](#).

NOTE

If you do not already have an Azure account, you will need to create one. If you are following this tutorial in a classroom or lab situation, ask your instructor or one of the proctors for help setting up your new account.

2. Once you are logged in, click on **New** in the top left corner, and search for **Notification Hub**, and click **Enter**.

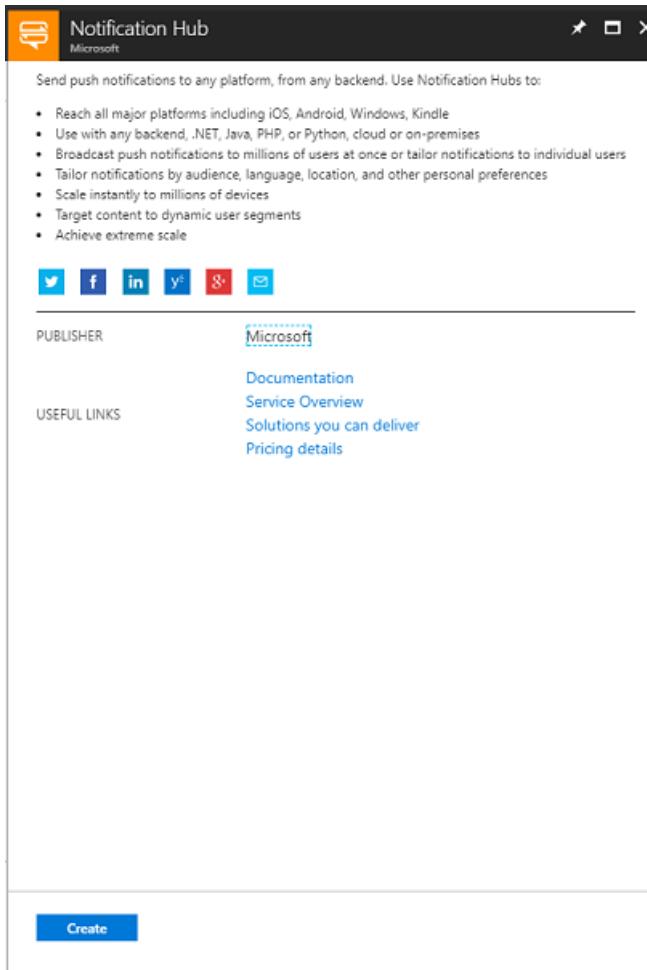
The screenshot shows the Microsoft Azure Marketplace search results. The search bar at the top contains the text 'Notification Hub', which is highlighted with a red box. The results table has columns for NAME, PUBLISHER, and CATEGORY. One result is listed: 'Notification Hub' by Microsoft, categorized under 'Web + Mobile'.

NOTE

The word **New** may have been replaced with **Create a resource**, in newer portals.

3. The new page will provide a description of the *Notification Hubs* service. At the bottom left of this prompt,

select the **Create** button, to create an association with this service.



4. Once you have clicked on **Create**:

- a. Insert your desired name for this service instance.
- b. Provide a **namespace** which you will be able to associate with this app.
- c. Select a **Location**.
- d. Choose a **Resource Group** or create a new one. A resource group provides a way to monitor, control access, provision and manage billing for a collection of Azure assets. It is recommended to keep all the Azure services associated with a single project (e.g. such as these labs) under a common resource group).

If you wish to read more about Azure Resource Groups, please follow this [link on how to manage a Resource Group](#).

- e. Select an appropriate **Subscription**.
- f. You will also need to confirm that you have understood the Terms and Conditions applied to this Service.
- g. Select **Create**.

The screenshot shows the 'New Notification Hub' configuration page in the Azure Marketplace. The form contains the following fields:

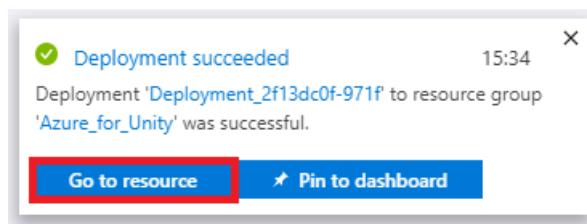
- Notification Hub**: MR_NotHub_ServiceInstance
- Create a new namespace**: MRNotHubsNS
- Select existing**: (button)
- Location**: West US
- Resource Group**: Use existing (radio button selected) Azure_for_Unity
- Subscription**: (dropdown menu)
- Pricing tier**: Free

At the bottom, there is a 'Pin to dashboard' checkbox and a 'Create' button.

- Once you have clicked on **Create**, you will have to wait for the service to be created, this might take a minute.
- A notification will appear in the portal once the Service instance is created.



- Click the **Go to resource** button in the notification to explore your new Service instance. You will be taken to your new **Notification Hub** service instance.



- From the overview page, halfway down the page, click **Windows (WNS)**. The panel on the right will change to show two text fields, which require your **Package SID** and **Security Key**, from the app you set up previously.

Home > MR_Nothub_ServiceInstance

MR_Nothub_ServiceInstance

Notification Hub

Search (Ctrl+ /)

Overview

Activity log

Access control (IAM)

Tags

Diagnose and solve problems

Quick Start

Properties

NOTIFICATION SETTINGS

- Apple (APNS)
- Google (GCM)
- Windows (WNS)
- Windows Phone (MPNS)
- Amazon (ADM)
- Baidu (Android China)

MANAGE

- Access Policies
- Pricing Tier

SETTINGS

- Locks
- Automation script

Test Send Delete

Essentials

Resource group (change) **Azure_for_Unity**

Subscription ID e7eaf587-7a72-47d0-85e1-f97e7b213aa5

Subscription name (change) **Mal's Microsoft Azure Internal Consumption**

Location West US

ACTIVE DEVICE REGISTERED 0

TOTAL PUSHES SINCE 2/01/2018 0

MAX DEVICE ALLOWED IN NAMESPACE 500

INCLUDED PUSHES IN NAMESPACE 1 M

Monitoring

Incoming Messages, APNs Authentication Errors and 3 more metrics today

100
80
60
40
20
0

- Once you have copied the details into the correct fields, click **Save**, and you will receive a notification when the Notification Hub has been successfully updated.

Home > MR_Nothub_ServiceInstance - Windows (WNS)

MR_Nothub_ServiceInstance - Windows (WNS)

Save Discard

Package SID ms-appx://...

Security Key ...Gra3

Notification Hub updated successfully! 15:35

Notification Hub updated successfully!

Overview

Activity log

Access control (IAM)

Tags

Diagnose and solve problems

Quick Start

Properties

NOTIFICATION SETTINGS

- Apple (APNS)
- Google (GCM)
- Windows (WNS)
- Windows Phone (MPNS)

Chapter 4 - Setup Azure Portal: create Table Service

After creating your Notification Hubs Service instance, navigate back to your Azure Portal, where you will create an Azure Tables Service by creating a Storage Resource.

- If not already signed in, log into the [Azure Portal](#).
- Once logged in, click on **New** in the top left corner, and search for **Storage account**, and click **Enter**.

NOTE

The word **New** may have been replaced with **Create a resource**, in newer portals.

3. Select **Storage account - blob, file, table, queue** from the list.

The screenshot shows the Azure Marketplace search interface. At the top, there's a breadcrumb navigation: Home > New > Marketplace > Everything. Below that is a search bar with the placeholder 'storage account'. Under the heading 'Results', there's a table with columns: NAME, PUBLISHER, and CATEGORY. The first row, 'Storage account - blob, file, table, queue' by Microsoft, is highlighted with a red box. Other visible items include 'Data Lake Store' by Microsoft, 'SafeNet ProtectV Service Gateway, 200 Nodes' by Gemalto, 'HPE ArcSight Logger' by Hewlett Packard Enterprise, and 'MailCloud Archiver 250 User Basic' by MailCloud.

NAME	PUBLISHER	CATEGORY
Storage account - blob, file, table, queue	Microsoft	Storage
Data Lake Store	Microsoft	Storage
SafeNet ProtectV Service Gateway, 200 Nodes	Gemalto	Comput
HPE ArcSight Logger	Hewlett Packard Enterprise	Comput
MailCloud Archiver 250 User Basic	MailCloud	Comput

4. The new page will provide a description of the **Storage account** service. At the bottom left of this prompt, select the **Create** button, to create an instance of this service.

The screenshot shows the detailed view of the 'Storage account - blob, file, table, queue' service. At the top, it says 'Storage account - blob, file, table, queue Microsoft'. Below that is a description: 'Microsoft Azure provides scalable, durable cloud storage, backup, and recovery solutions for any data, big or small. It works with the infrastructure you already have to cost-effectively enhance your existing applications and business continuity strategy, and provide the storage required by your cloud applications, including unstructured text or binary data such as video, audio, and images.' There are social sharing icons (Facebook, Twitter, LinkedIn, YouTube, etc.) and useful links for Documentation, Service overview, and Pricing. At the bottom left, there's a prominent blue 'Create' button.

5. Once you have clicked on **Create**, a panel will appear:

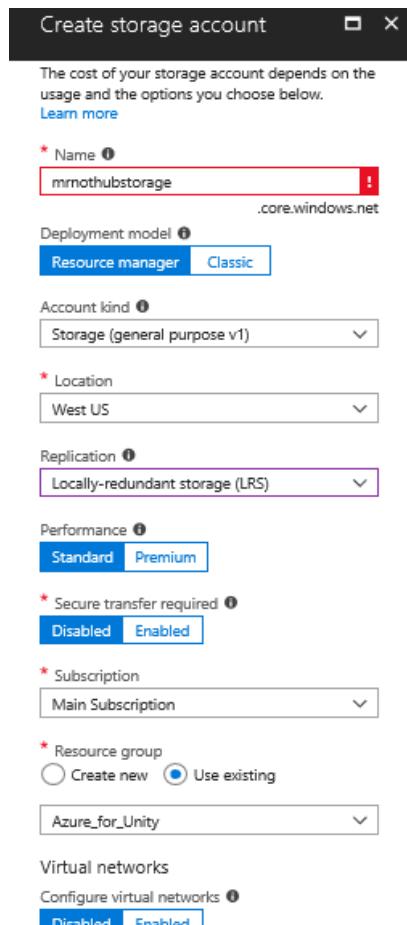
- Insert your desired **Name** for this service instance (must be all lowercase).
- For **Deployment model**, click **Resource manager**.
- For **Account kind**, using the dropdown menu, select **Storage (general purpose v1)**.
- Select an appropriate **Location**.
- For the **Replication** dropdown menu, select **Read-access-geo-redundant storage (RA-GRS)**.
- For **Performance**, click **Standard**.
- Within the **Secure transfer required** section, select **Disabled**.
- From the **Subscription** dropdown menu, select an appropriate subscription.
- Choose a **Resource Group** or create a new one. A resource group provides a way to monitor, control access, provision and manage billing for a collection of Azure assets. It is recommended to keep all

the Azure services associated with a single project (e.g. such as these labs) under a common resource group).

If you wish to read more about Azure Resource Groups, please follow this [link on how to manage a Resource Group](#).

j. Leave **Virtual networks** as **Disabled** if this is an option for you.

k. Click **Create**.

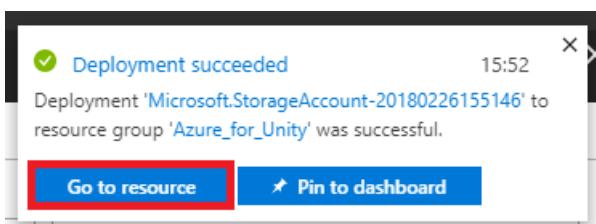


6. Once you have clicked on **Create**, you will have to wait for the service to be created, this might take a minute.

7. A notification will appear in the portal once the Service instance is created. Click on the notifications to explore your new Service instance.



8. Click the **Go to resource** button in the notification to explore your new Service instance. You will be taken to your new Storage Service instance overview page.



9. From the overview page, to the right-hand side, click **Tables**.

The screenshot shows the Azure Storage account overview page for 'mrnothubstorage'. On the left, there's a sidebar with navigation links like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, SETTINGS (Storage Explorer, Access keys, Configuration, Shared access signature, Firewalls and virtual networks, Metrics (preview), Properties, Locks), and a search bar. The main content area displays account details: Resource group (Azure_for_Unity), Status (Primary: Available), Location (West US), Subscription (change), Performance Standard, Replication Locally-redundant storage (LRS), Account kind Storage (general purpose v1). Below this, there are four service cards: Blobs (Object storage for unstructured data), Files (File shares that use SMB 3.0 protocol), Queues (Scale apps depending on traffic), and Tables (Tabular data storage). The 'Tables' card is highlighted with a red box.

10. The panel on the right will change to show the **Table service** information, wherein you need to add a new table. Do this by clicking the **+ Table** button to the top-left corner.

The screenshot shows the 'Table service' page for 'mrnothubstorage'. At the top, there's a header with Home > mrnothubstorage > Table service. Below it, there are buttons for Refresh, + Table (highlighted with a red box), and Delete tables. A message encourages checking out premium Table experience with Azure Cosmos DB. The main area shows account details: Storage account (mrnothubstorage), Status (Primary: Available), Location (West US), Subscription (change) (Mail's Microsoft Azure Internal Consumption), and Subscription ID. It also shows the Table service endpoint URL: https://mrnothubstorage.table.core.windows.net/. Below this, there's a search bar for tables by prefix, a table section with columns for TABLE and URL, and a message stating 'You don't have any tables yet.'

11. A new page will be shown, wherein you need to enter a **Table name**. This is the name you will use to refer to the data in your application in later Chapters. Insert an appropriate name and click **OK**.

The screenshot shows the 'Add table' dialog box. At the top, there's a header with Home > mrnothubstorage > Table service and a Table service section. Below it, there are buttons for Refresh, + Table (highlighted with a red box), and Delete tables. The main area is titled 'Add table' and contains a form with a required field 'Table name' (highlighted with a red box) containing the value 'SceneObjectsTable'. At the bottom, there are 'OK' and 'Cancel' buttons.

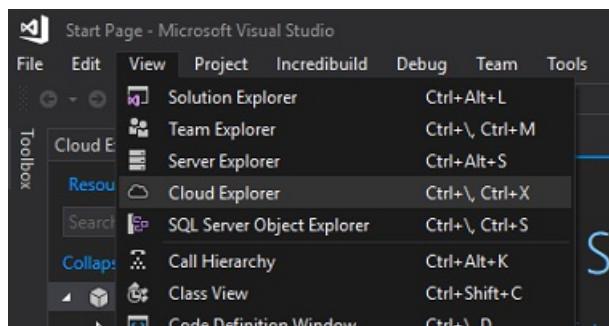
12. Once the new table has been created, you will be able to see it within the **Table service** page (at the bottom).

The screenshot shows the Azure Table service blade. At the top, it displays the storage account 'mrnothubstorage', status 'Available', location 'West US', and subscription 'Ma's Microsoft Azure Internal Consumption'. Below this, there is a search bar labeled 'Search tables by prefix' and a table list. The table 'SceneObjectsTable' is selected and highlighted with a red box. To the right of the table list, there is a URL field containing 'https://mrnothubstorage.table.core.windows.net/'.

Chapter 5 - Completing the Azure Table in Visual Studio

Now that your **Table service** storage account has been setup, it is time to add data to it, which will be used to store and retrieve information. The editing of your Tables can be done through **Visual Studio**.

1. Open **Visual Studio**.
2. From the menu, click **View > Cloud Explorer**.

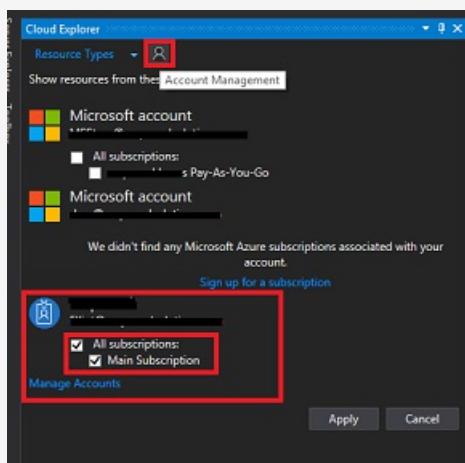


3. The **Cloud Explorer** will open as a docked item (be patient, as loading may take time).

NOTE

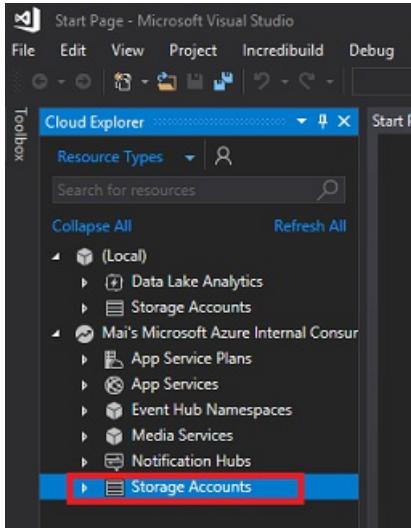
If the Subscription you used to create your *Storage Accounts* is not visible, ensure that you have:

- Logged in to the same account as the one you used for the Azure Portal.
- Selected your Subscription from the Account Management Page (you may need to apply a filter from your account settings):

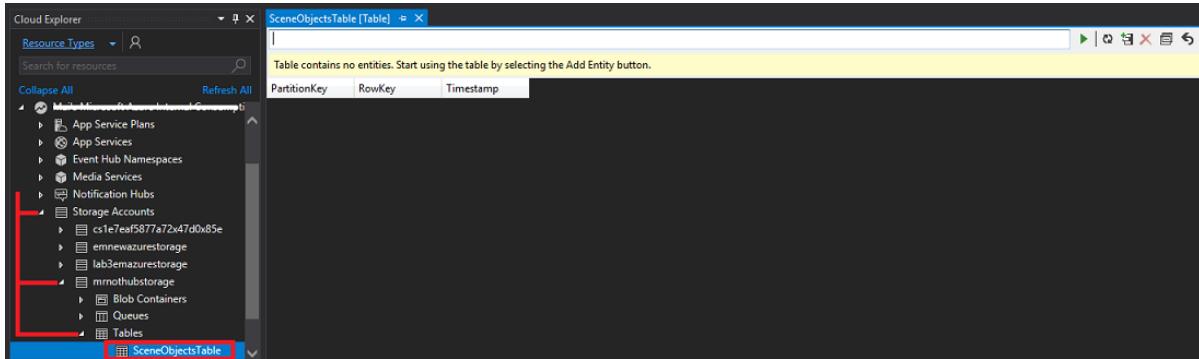


4. Your Azure cloud services will be shown. Find **Storage Accounts** and click the arrow to the left of that to

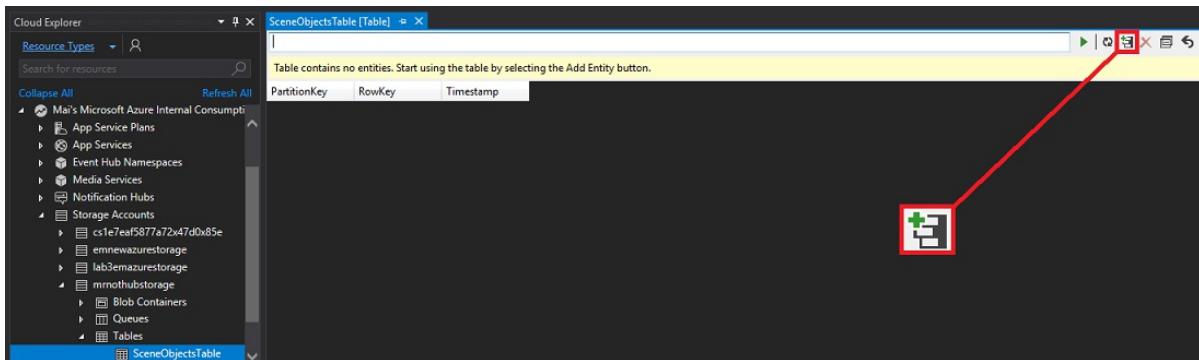
expand your accounts.



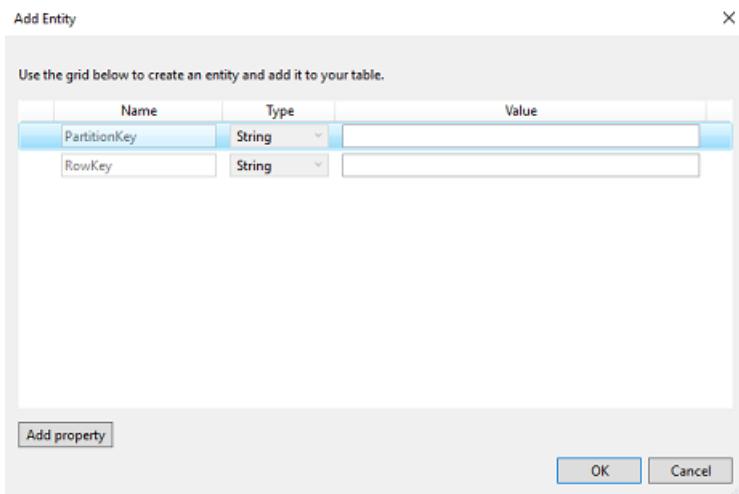
- Once expanded, your newly created **Storage account** should be available. Click the arrow to the left of your storage, and then once that is expanded, find **Tables** and click the arrow next to that, to reveal the **Table** you created in the last Chapter. Double click your **Table**.



- Your table will be opened in the center of your Visual Studio window. Click the table icon with the + (plus) on it.



- A window will appear prompting for you to *Add Entity*. You will create three entities in total, each with several properties. You will notice that *PartitionKey* and *RowKey* are already provided, as these are used by the table to find your data.

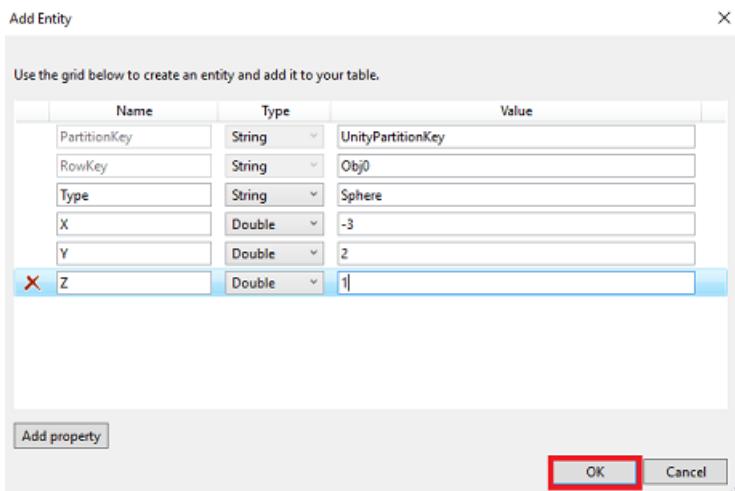


8. Update the **Value** of the **PartitionKey** and **RowKey** as follows (remember to do this for each row property you add, though increment the RowKey each time):

Name	Type	Value
PartitionKey	String	UnityPartitionKey
RowKey	String	Obj0

9. Click **Add property** to add extra rows of data. Make your first empty table match the below table.

10. Click **OK** when you are finished.



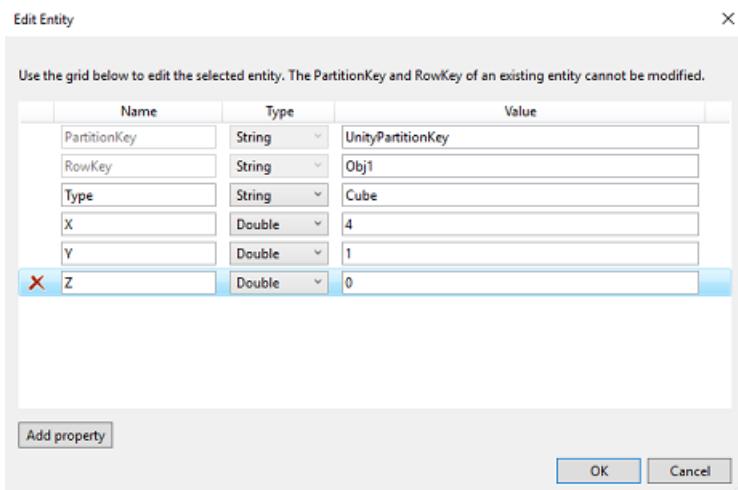
WARNING

Ensure that you have changed the **Type** of the **X**, **Y**, and **Z**, entries to **Double**.

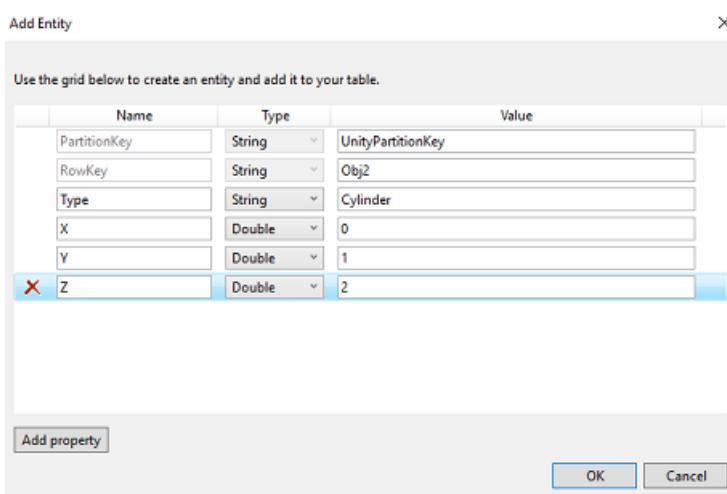
11. You will notice your table now has a row of data. Click the + (plus) icon again to add another entity.

PartitionKey	RowKey	Timestamp	Type	X	Y	Z
UnityPartitionKey	Obj0	26/02/2018 5:17...	Sphere	-3	2	1

12. Create an additional property, and then set the values of the new entity to match those shown below.



13. Repeat the last step to add another entity. Set the values for this entity to those shown below.



14. Your table should now look like the one below.

SceneObjectsTable [Table]						
Enter a WCF Data Services filter to limit the entities returned						
PartitionKey	RowKey	Timestamp	Type	X	Y	Z
UnityPartitionKey	Obj0	26/02/2018 5:17...	Sphere	-3	2	1
UnityPartitionKey	Obj1	26/02/2018 5:23...	Cube	4	1	0
UnityPartitionKey	Obj2	26/02/2018 5:24...	Cylinder	0	1	2

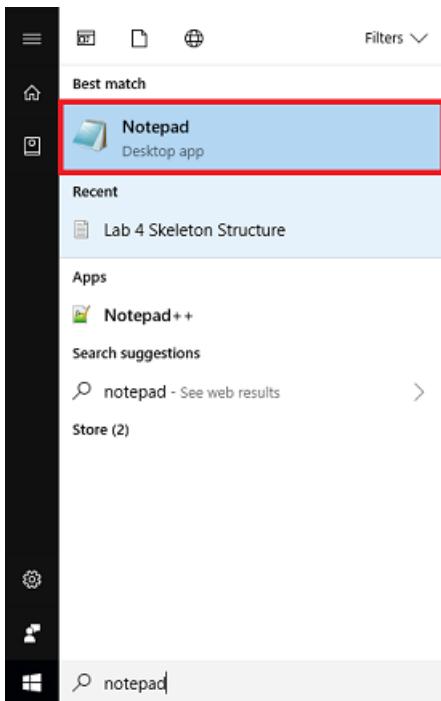
15. You have completed this Chapter. Make sure to save.

Chapter 6 - Create an Azure Function App

Create an Azure Function App, which will be called by the Desktop application to update the **Table** service and send a notification through the **Notification Hub**.

First, you need to create a file that will allow your Azure Function to load the libraries you need.

1. Open **Notepad** (press Windows Key and type notepad).



- With Notepad open, insert the JSON structure below into it. Once you have done that, save it on your desktop as **project.json**. It is important that the naming is correct: ensure it does **NOT have a .txt file extension**. This file defines the libraries your function will use, if you have used NuGet it will look familiar.

```
{  
  "frameworks": {  
    "net46":{  
      "dependencies": {  
        "WindowsAzure.Storage": "7.0.0",  
        "Microsoft.Azure.NotificationHubs" : "1.0.9",  
        "Microsoft.Azure.WebJobs.Extensions.NotificationHubs" :"1.1.0"  
      }  
    }  
  }  
}
```

- Log in to the [Azure Portal](#).

- Once you are logged in, click on **New** in the top left corner, and search for **Function App**, press **Enter**.

A screenshot of the Microsoft Azure Marketplace. On the left, there is a sidebar with various service categories like "Create a resource", "All services", "Favorites", etc. The main area shows a search bar with "function app" typed in. Below the search bar is a "Results" table with three items:

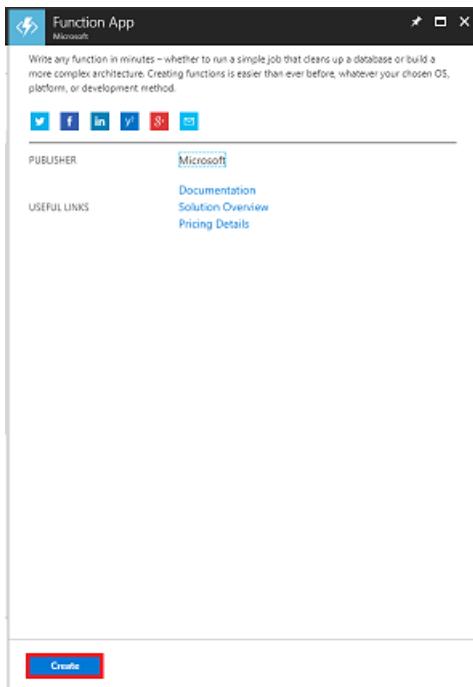
NAME	PUBLISHER	CATEGORY
Function App	Microsoft	Web + Mobile
Functions Bot	Microsoft	AI + Cognitive Services
Logic Apps Custom Connector	Microsoft	Web + Mobile

The "Function App" row is highlighted with a red box.

NOTE

The word **New** may have been replaced with **Create a resource**, in newer portals.

- The new page will provide a description of the **Function App** service. At the bottom left of this prompt, select the **Create** button, to create an association with this service.



6. Once you have clicked on **Create**, fill in the following:

- a. For **App name**, insert your desired name for this service instance.
- b. Select a **Subscription**.
- c. Select the pricing tier appropriate for you, if this is the first time creating a **Function App Service**, a free tier should be available to you.
- d. Choose a **Resource Group** or create a new one. A resource group provides a way to monitor, control access, provision and manage billing for a collection of Azure assets. It is recommended to keep all the Azure services associated with a single project (e.g. such as these labs) under a common resource group).

If you wish to read more about Azure Resource Groups, please follow this [link on how to manage a Resource Group](#).

- e. For **OS**, click Windows, as that is the intended platform.
- f. Select a **Hosting Plan** (this tutorial is using a **Consumption Plan**).
- g. Select a **Location (choose the same location as the storage you have built in the previous step)**
- h. For the **Storage** section, **you must select the Storage Service you created in the previous step**.
 - i. You will not need *Application Insights* in this app, so feel free to leave it **Off**.
 - j. Click **Create**.

Home > New > Marketplace > Everything > Function

Function App

Create

* App name
MRNNotHubFunctionApp ✓

.azurewebsites.net

* Subscription
Mai's Microsoft Azure Internal Consumption

* Resource Group ✓
 Create new Use existing
Azure_for_Unity

* OS Windows Linux (Preview)

* Hosting Plan ✓
Consumption Plan

* Location
West US

* Storage ✓
 Create new Use existing
mrothubstorage

Application Insights ✓ On Off

⚠ For optimal performance you should use a storage account in the same region as the Function App.

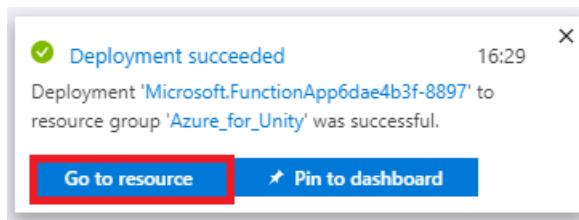
Pin to dashboard

Create Automation options

7. Once you have clicked on **Create** you will have to wait for the service to be created, this might take a minute.
8. A notification will appear in the portal once the Service instance is created.

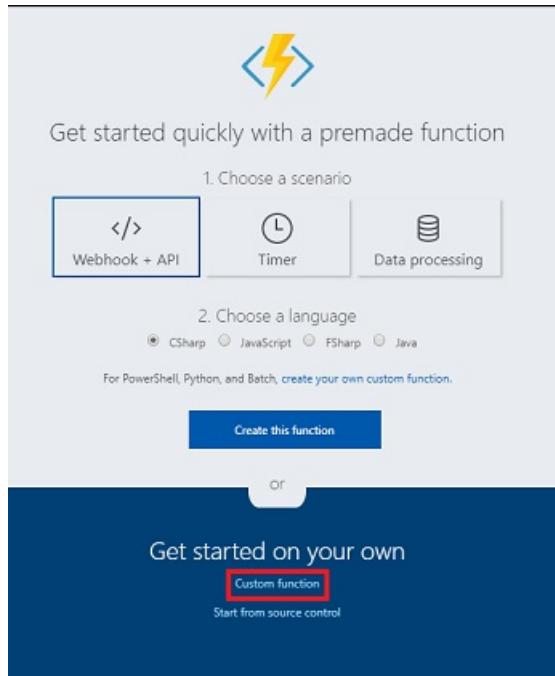


9. Click on the notifications to explore your new Service instance.
10. Click the **Go to resource** button in the notification to explore your new Service instance.



11. Click the + (plus) icon next to *Functions*, to *Create new*.

12. Within the central panel, the **Function** creation window will appear. Ignore the information in the upper half of the panel, and click **Custom function**, which is located near the bottom (in the blue area, as below).



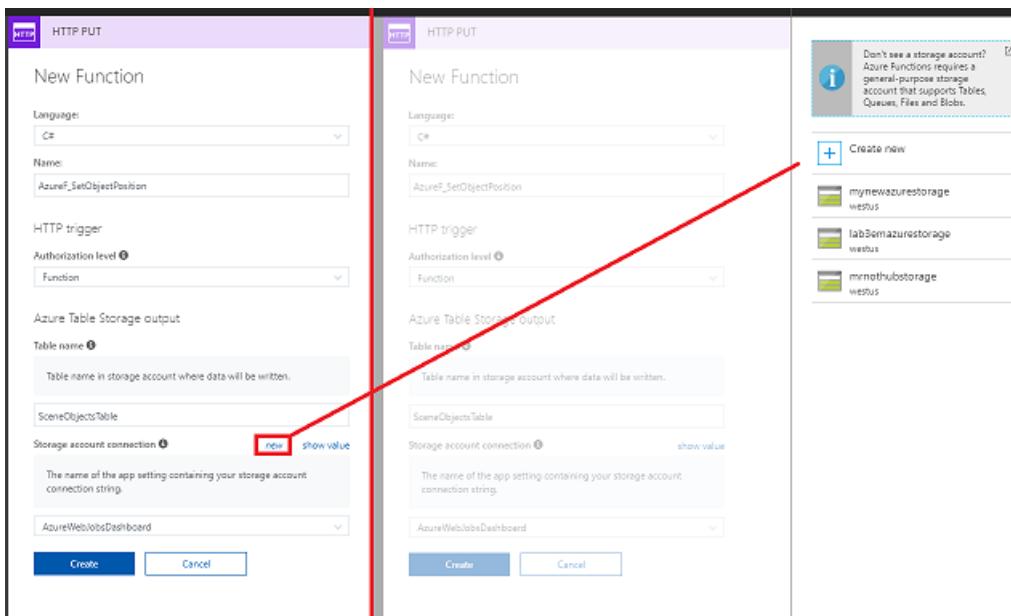
13. The new page within the window will show various function types. Scroll down to view the purple types, and click **HTTP PUT** element.

IMPORTANT

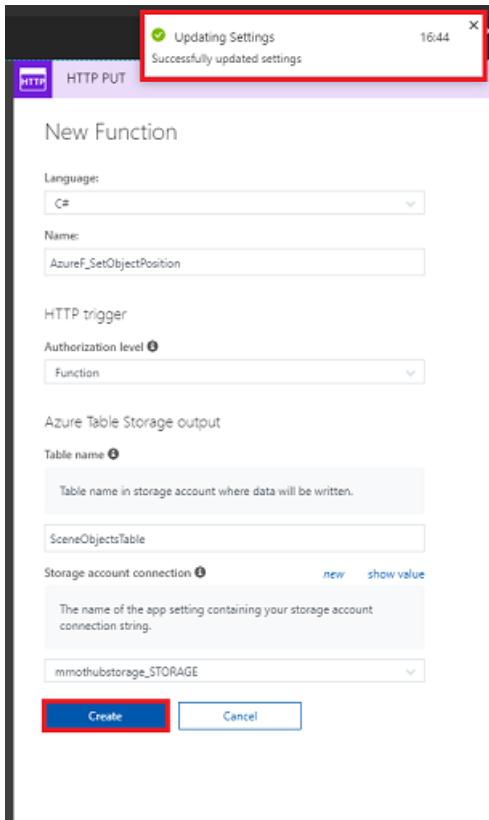
You may have to scroll further down the page (and this image may not look exactly the same, if Azure Portal updates have taken place), however, you are looking for an element called **HTTP PUT**.

14. The **HTTP PUT** window will appear, where you need to configure the function (see below for image).

- a. For **Language**, using the dropdown menu, select C#.
- b. For **Name**, input an appropriate name.
- c. In the **Authentication level** dropdown menu, select **Function**.
- d. For the **Table name** section, you need to use the exact name you used to create your **Table** service previously (including the same letter case).
- e. Within the **Storage account connection** section, use the dropdown menu, and select your storage account from there. If it is not there, click the **New** hyperlink alongside the section title, to show another panel, where your storage account should be listed.



15. Click **Create** and you will receive a notification that your settings have been updated successfully.



16. After clicking **Create**, you will be redirected to the function editor.

```

    1 #r "Microsoft.WindowsAzure.Storage"
    2
    3 using System.Net;
    4 using Microsoft.WindowsAzure.Storage.Table;
    5
    6 public static HttpResponseMessage Run(Person person, CloudTable outTable, TraceWriter log)
    7 {
    8     if (string.IsNullOrEmpty(person.Name))
    9     {
    10         return new HttpResponseMessage(HttpStatusCode.BadRequest);
    11     }
    12     Content = new StringContent("A non-empty Name must be specified.");
    13 }
    14
    15 log.Info($"PersonName={person.Name}");
    16
    17 TableOperation updateOperation = TableOperation.InsertOrReplace(person);
    18 TableResult result = outTable.Execute(updateOperation);
    19 return new HttpResponseMessage(( HttpStatusCode )result.StatusCode);
    20
    21
    22 public class Person : TableEntity
    23 {
    24     public string Name { get; set; }
    25 }
    26
    27
  
```

17. Insert the following code into the function editor (replacing the code in the function):

```

#r "Microsoft.WindowsAzure.Storage"

using System;
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.Table;
using Microsoft.Azure.NotificationHubs;
using Newtonsoft.Json;

public static async Task Run(UnityGameObject gameObj, CloudTable table, IAsyncCollector<Notification>
notification, TraceWriter log)
{
    //RowKey of the table object to be changed
    string rowKey = gameObj.RowKey;

    //Retrieve the table object by its RowKey
    TableOperation operation = TableOperation.Retrieve<UnityGameObject>("UnityPartitionKey", rowKey);

    TableResult result = table.Execute(operation);

    //Create a UnityGameObject so to set its parameters
    UnityGameObject existingGameObj = (UnityGameObject)result.Result;

    existingGameObj.RowKey = rowKey;
    existingGameObj.X = gameObj.X;
    existingGameObj.Y = gameObj.Y;
    existingGameObj.Z = gameObj.Z;

    //Replace the table appropriate table Entity with the value of the UnityGameObject
    operation = TableOperation.Replace(existingGameObj);

    table.Execute(operation);

    log.Verbose($"Updated object position");

    //Serialize the UnityGameObject
    string wnsNotificationPayload = JsonConvert.SerializeObject(existingGameObj);

    log.Info($"{wnsNotificationPayload}");

    var headers = new Dictionary<string, string>();

    headers["X-WNS-Type"] = @"wns/raw";

    //Send the raw notification to subscribed devices
    await notification.AddAsync(new WindowsNotification(wnsNotificationPayload, headers));

    log.Verbose($"Sent notification");
}

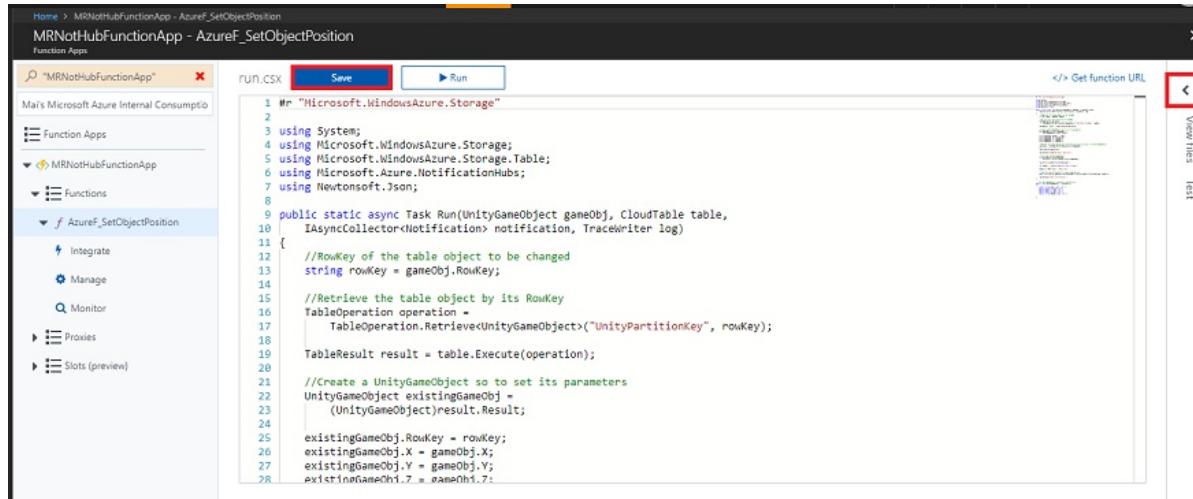
// This UnityGameObject represent a Table Entity
public class UnityGameObject : TableEntity
{
    public string Type { get; set; }
    public double X { get; set; }
    public double Y { get; set; }
    public double Z { get; set; }
    public string RowKey { get; set; }
}

```

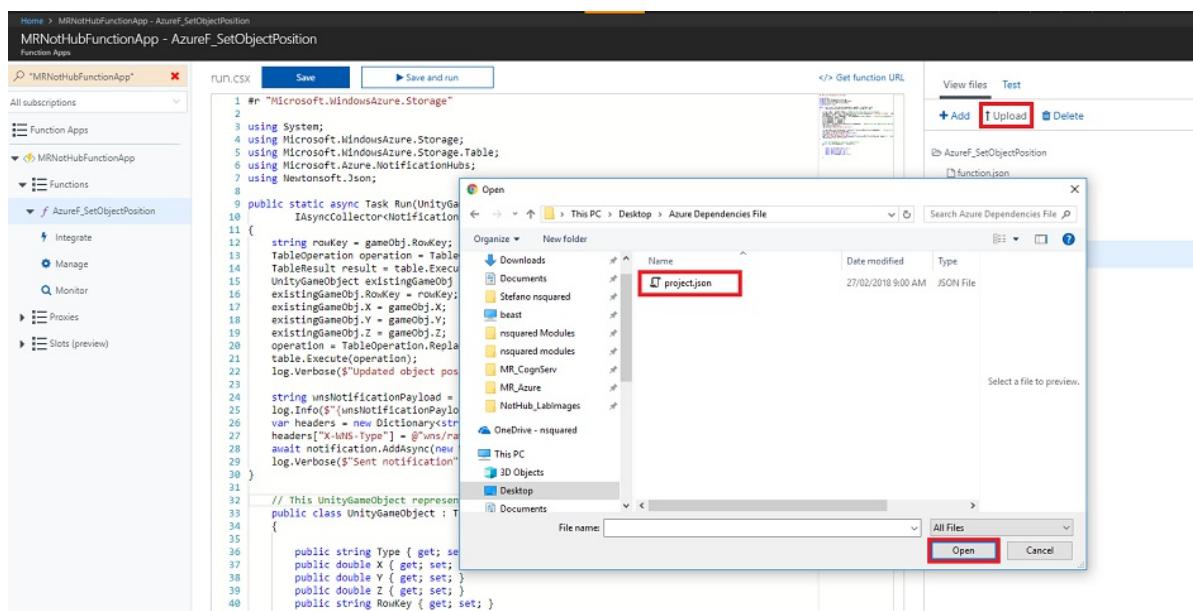
NOTE

Using the included libraries, the function receives the name and location of the object which was moved in the Unity scene (as a C# object, called **UnityGameObject**). This object is then used to update the object parameters within the created table. Following this, the function makes a call to your created Notification Hub service, which notifies all subscribed applications.

18. With the code in place, click **Save**.
19. Next, click the < (arrow) icon, on the right-hand side of the page.



20. A panel will slide in from the right. In that panel, click **Upload**, and a File Browser will appear.
21. Navigate to, and click, the **project.json** file, which you created in **Notepad** previously, and then click the **Open** button. This file defines the libraries that your function will use.



22. When the file has uploaded, it will appear in the panel on the right. Clicking it will open it within the **Function** editor. It must look **exactly** the same as the next image (below step 23).
23. Then, in the panel on the left, beneath **Functions**, click the **Integrate** link.

The screenshot shows the Azure portal interface for a function app named 'MRNotHubFunctionApp'. In the left sidebar, under 'Functions', the 'AzureF_SetObjectPosition' function is selected. A red box highlights the 'Integrate' button. The main content area displays the 'project.json' file's code:

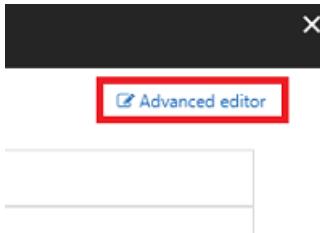
```

1 {
2   "frameworks": {
3     "net46": {
4       "dependencies": {
5         "WindowsAzure.Storage": "7.0.0",
6         "Microsoft.Azure.NotificationHubs": "1.0.9",
7         "Microsoft.Azure.WebJobs.Extensions.NotificationHubs": "1.1.0"
8       }
9     }
10   }
11 }

```

To the right, there is a file browser with files like 'function.json', 'project.json', 'project.lock.json', and 'run.csx'.

24. On the next page, in the top right corner, click **Advanced editor** (as below).



25. A **function.json** file will be opened in the center panel, which needs to be replaced with the following code snippet. This defines the function you are building and the parameters passed into the function.

```
{
  "bindings": [
    {
      "authLevel": "function",
      "type": "httpTrigger",
      "methods": [
        "get",
        "post"
      ],
      "name": "gameObj",
      "direction": "in"
    },
    {
      "type": "table",
      "name": "table",
      "tableName": "SceneObjectsTable",
      "connection": "mrnothubstorage_STORAGE",
      "direction": "in"
    },
    {
      "type": "notificationHub",
      "direction": "out",
      "name": "notification",
      "hubName": "MR_NotHub_ServiceInstance",
      "connection": "MRNotHubNS_DefaultFullSharedAccessSignature_NH",
      "platform": "wns"
    }
  ]
}
```

26. Your editor should now look like the image below:

```

function.json
1  "bindings": [
2    {
3      "authLevel": "function",
4      "type": "httpTrigger",
5      "methods": [
6        "get",
7        "post"
8      ],
9      "name": "gameObj",
10     "direction": "in"
11   },
12   {
13     "type": "table",
14     "name": "table",
15     "tableName": "SceneObjectsTable",
16     "connection": "mrnothubstorage_STORAGE",
17     "direction": "in"
18   },
19   {
20     "type": "notificationHub",
21     "direction": "out",
22     "name": "notification",
23     "hubName": "MR_NotHub_ServiceInstance",
24     "connection": "MRNotHubS_DefaultFullSharedAccessSignature_MH",
25     "platform": "wns"
26   }
27 }
28 ]

```

27. You may notice the input parameters that you have just inserted might not match your table and storage details and therefore will need to be updated with your information. **Do not do this here**, as it is covered next. Simply click the **Standard editor** link, in the top-right corner of the page, to go back.

28. Back in the **Standard editor**, click **Azure Table Storage (table)**, under **Inputs**.

Inputs

Azure Table Storage (table)

Table parameter name: table

Partition key (optional): Partition key (optional)

Maximum number of records to read (optional): 50

Storage account connection: mrnothubstorage_STORAGE (show value, new)

Table name: SceneObjectsTable

Row key (optional): Row key (optional)

Query filter (optional): Query filter (optional)

29. Ensure the following match to **your** information, as they may be different (there is an image below the following steps):

- Table name:** the name of the table you created within your Azure Storage, Tables service.
- Storage account connection:** click **new**, which appears alongside the dropdown menu, and a panel will appear to the right of the window.

The screenshot shows the 'Storage Account' configuration page for an Azure Function. On the right, there's a sidebar with a list of storage accounts: 'mynewazurestorage westus', 'lab3semazurestorage westus', and 'mrnthonhubstorage westus'. On the left, under 'Inputs', there's a 'Table Storage input' section with fields for 'Table name' (set to 'SceneObjectsTable'), 'Row key (optional)', 'Partition key (optional)' (set to 'Waiting for the correct value'), and 'Query filter (optional)'. Below this, the 'Storage account connection' dropdown is set to 'mrnthonhubstorage_STORAGE'. A red arrow points from the sidebar to this dropdown.

- Select your **Storage Account**, which you created previously to host the **Function Apps**.
- You will notice that the **Storage Account** connection value has been created.
- Make sure to press **Save** once you are done.

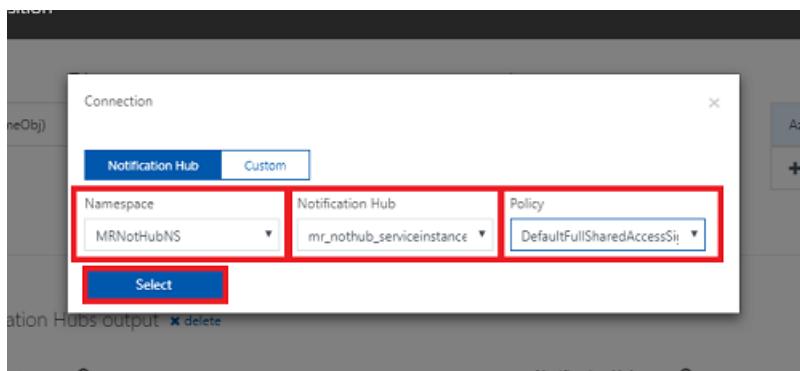
c. The **Inputs** page should now match the below, showing **your** information.

The screenshot shows the 'Inputs' configuration page for the 'Azure Table Storage input' section. It includes fields for 'Table name' (set to 'SceneObjectsTable'), 'Row key (optional)', 'Partition key (optional)', 'Maximum number of records to read (optional)' (set to 50), and 'Storage account connection' (set to 'mrnthonhubstorage_STORAGE'). The 'Storage account connection' dropdown and the 'Table name' field are both highlighted with red boxes. At the bottom, there are 'Save' and 'Cancel' buttons, with 'Save' being highlighted with a red box.

30. Next, click **Azure Notification Hub (notification)** - under **Outputs**. Ensure the following are matched to **your** information, as they may be different (there is an image below the following steps):

- Notification Hub Name:** this is the name of your **Notification Hub** service instance, which you created previously.
- Notification Hubs namespace connection:** click **new**, which appears alongside the dropdown menu.

- The **Connection** popup will appear (see image below), where you need to select the **Namespace** of the **Notification Hub**, which you set up previously.
- Select your **Notification Hub** name from the middle dropdown menu.
- Set the **Policy** dropdown menu to **DefaultFullSharedAccessSignature**.
- Click the **Select** button to go back.



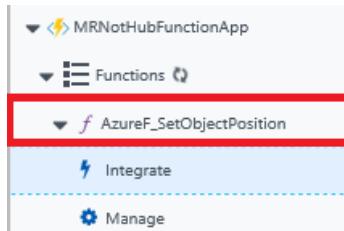
- The **Outputs** page should now match the below, but with **your** information instead. Make sure to press **Save**.

WARNING

Do not edit the Notification Hub name directly (this should all be done using the Advanced Editor, provided you followed the previous steps correctly).

32. At this point, you should test the function, to ensure it is working. To do this:

- Navigate to the function page once more:



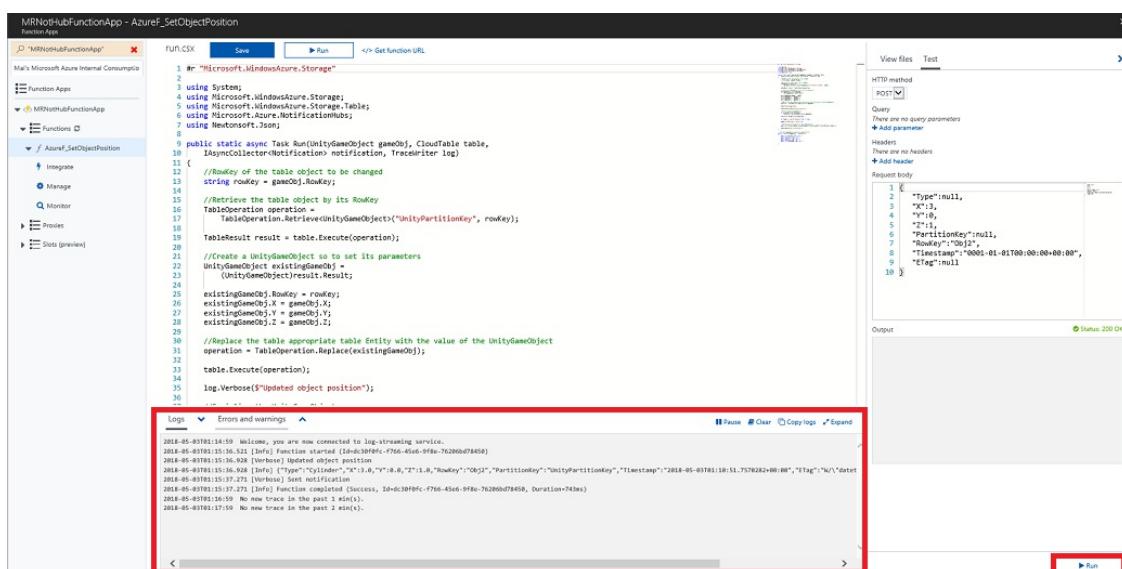
- Back on the function page, click the **Test** tab on the far right side of the page, to open the *Test* blade:



- Within the **Request body** textbox of the blade, paste the below code:

```
{
    "Type":null,
    "X":3,
    "Y":0,
    "Z":1,
    "PartitionKey":null,
    "RowKey":"Obj2",
    "Timestamp":"2001-01-01T00:00:00+00:00",
    "ETag":null
}
```

- With the test code in place, click the **Run** button at the bottom right, and the test will be run. The output logs of the test will appear in the console area, below your function code.



WARNING

If the above test fails, you will need to double check that you have followed the above steps exactly, particularly the settings within the **integrate panel**.

Chapter 7 - Set up Desktop Unity Project

IMPORTANT

The Desktop application which you are now creating, **will not** work in the Unity Editor. It needs to be run outside of the Editor, following the Building of the application, using Visual Studio (or the deployed application).

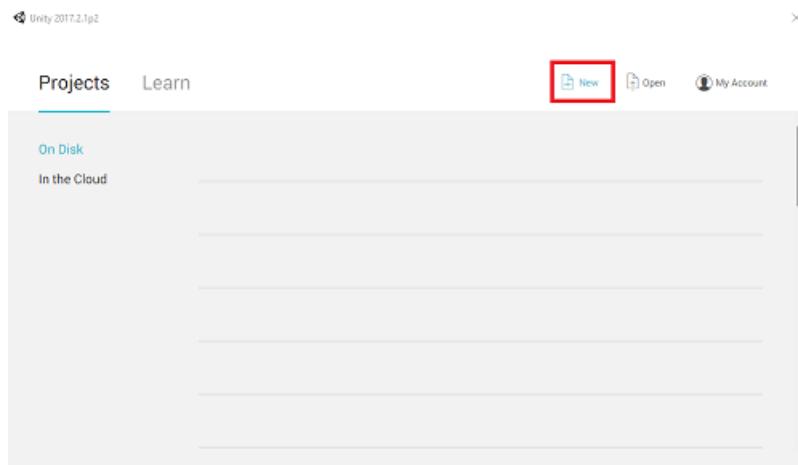
The following is a typical set up for developing with Unity and mixed reality, and as such, is a good template for other projects.

Set up and test your mixed reality immersive headset.

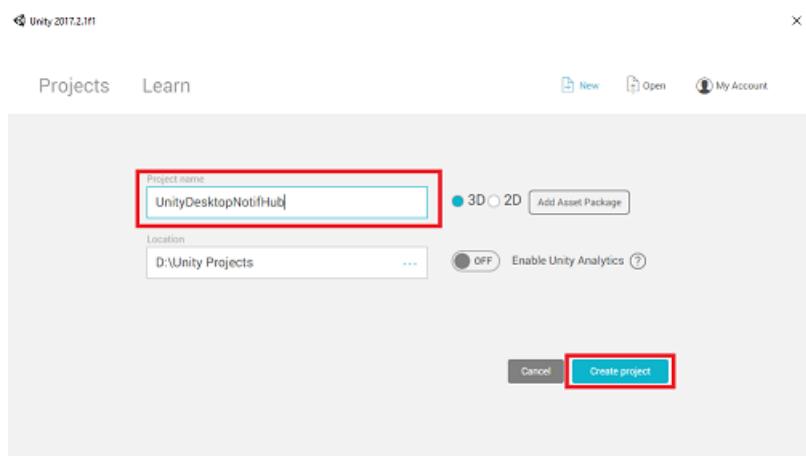
NOTE

You will **not** require Motion Controllers for this course. If you need support setting up the immersive headset, please follow this [link on how to set up Windows Mixed Reality](#).

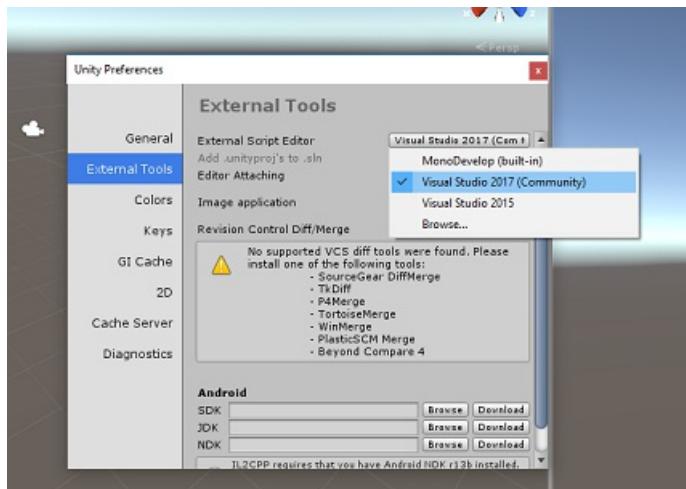
1. Open **Unity** and click **New**.



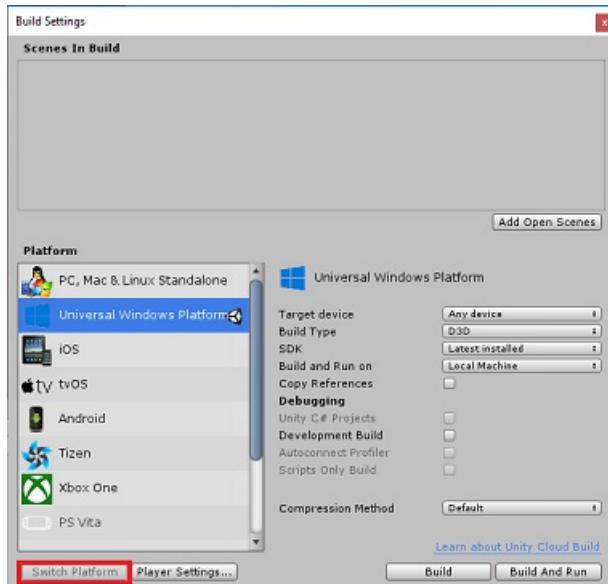
2. You need to provide a Unity Project name, insert **UnityDesktopNotifHub**. Make sure the project type is set to **3D**. Set the **Location** to somewhere appropriate for you (remember, closer to root directories is better). Then, click **Create project**.



3. With Unity open, it is worth checking the default **Script Editor** is set to **Visual Studio**. Go to **Edit > Preferences** and then from the new window, navigate to **External Tools**. Change **External Script Editor** to **Visual Studio 2017**. Close the **Preferences** window.



4. Next, go to **File > Build Settings** and select **Universal Windows Platform**, then click on the **Switch Platform** button to apply your selection.



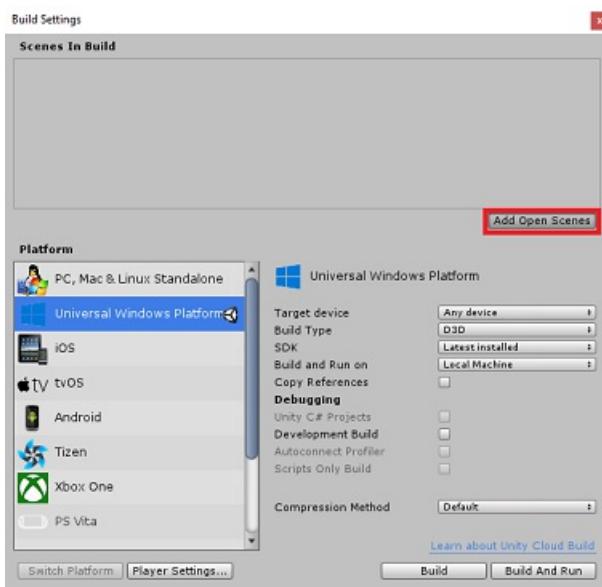
5. While still in **File > Build Settings**, make sure that:

- Target Device** is set to **Any Device**

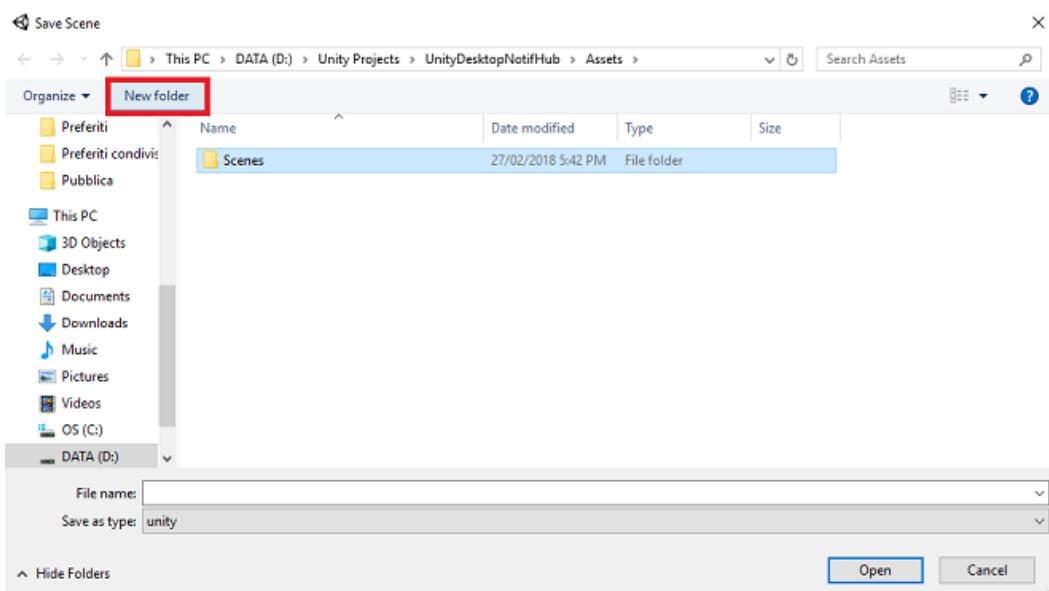
This Application will be for your desktop, so must be **Any Device**

- Build Type** is set to **D3D**
- SDK** is set to **Latest installed**
- Visual Studio Version** is set to **Latest installed**
- Build and Run** is set to **Local Machine**
- While here, it is worth saving the scene, and adding it to the build.

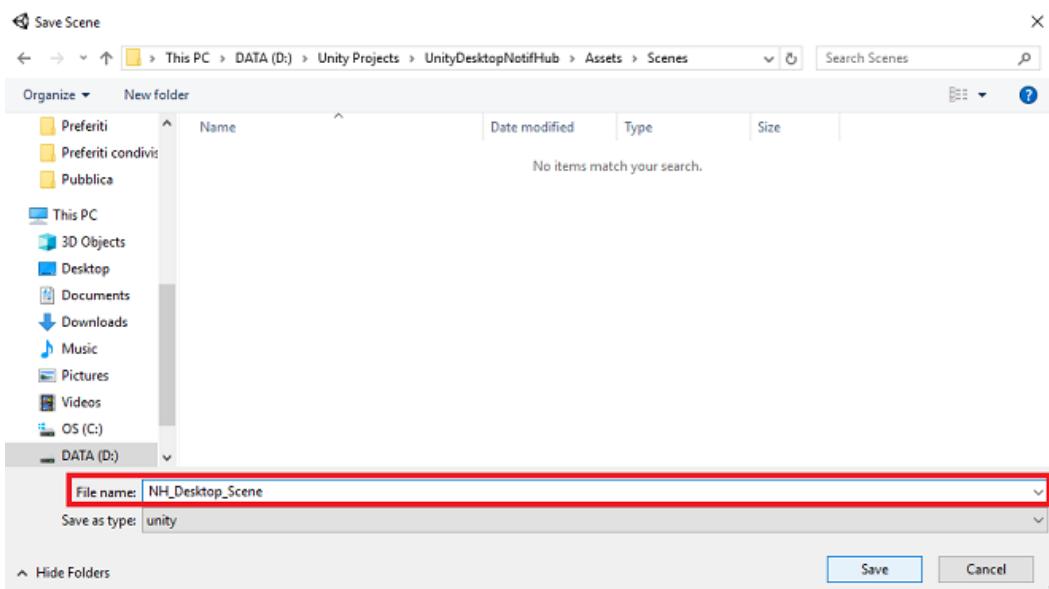
- Do this by selecting **Add Open Scenes**. A save window will appear.



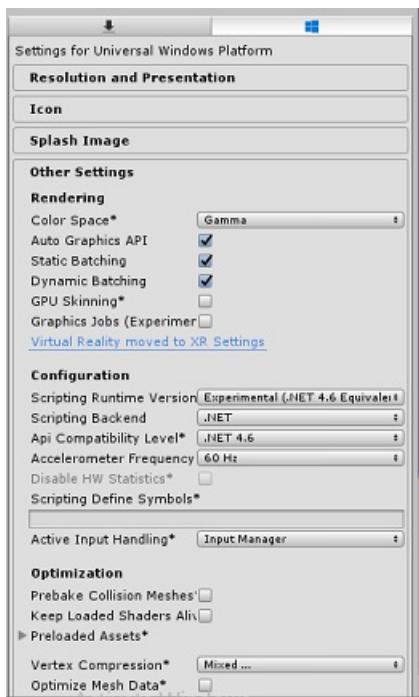
- b. Create a new folder for this, and any future, scene, then select the **New folder** button, to create a new folder, name it **Scenes**.



- c. Open your newly created **Scenes** folder, and then in the **File name:** text field, type **NH/Desktop_Scene**, then press **Save**.

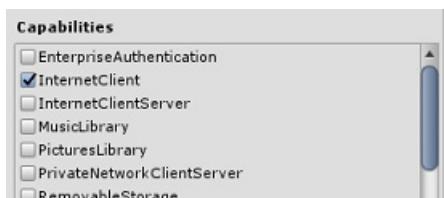


- g. The remaining settings, in **Build Settings**, should be left as default for now.
6. In the same window click on the **Player Settings** button, this will open the related panel in the space where the **Inspector** is located.
7. In this panel, a few settings need to be verified:
- In the **Other Settings** tab:
 - Scripting Runtime Version** should be **Experimental (.NET 4.6 Equivalent)**
 - Scripting Backend** should be **.NET**
 - API Compatibility Level** should be **.NET 4.6**



- b. Within the **Publishing Settings** tab, under **Capabilities**, check:

- **InternetClient**



8. Back in **Build Settings Unity C# Projects** is no longer greyed out; tick the checkbox next to this.
9. Close the **Build Settings** window.
10. Save your Scene and Project *File > Save Scene / File > Save Project*.

IMPORTANT

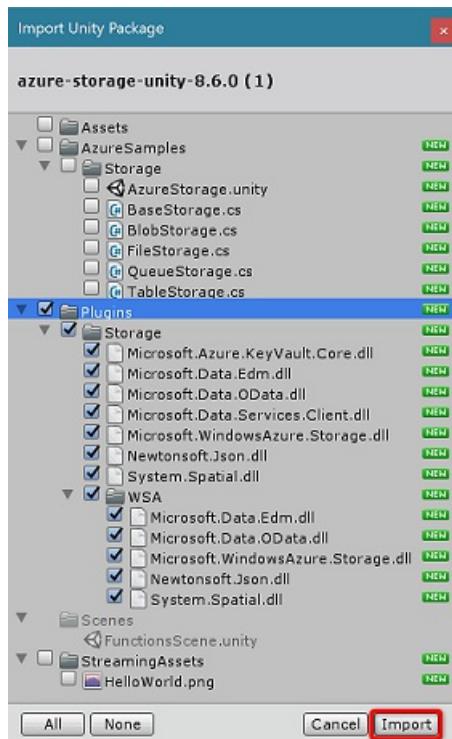
If you wish to skip the *Unity Set up* component for this project (Desktop App), and continue straight into code, feel free to [download this .unitypackage](#), import it into your project as a **Custom Package**, and then continue from [Chapter 9](#). You will still need to add the script components.

You will be using Azure Storage for Unity (which itself leverages the .Net SDK for Azure). For more information follow this [link about Azure Storage for Unity](#).

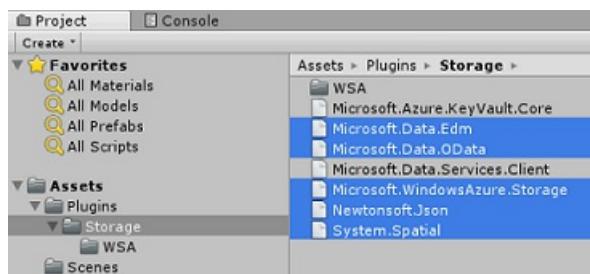
There is currently a known issue in Unity which requires plugins to be reconfigured after import. These steps (4 - 7 in this section) will no longer be required after the bug has been resolved.

To import the SDK into your own project, make sure you have downloaded the latest [.unitypackage](#) from GitHub. Then, do the following:

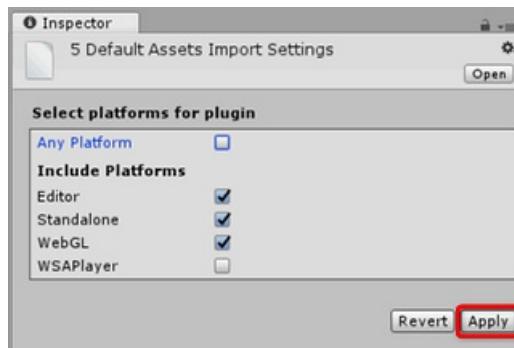
1. Add the **.unitypackage** to Unity by using the **Assets > Import Package > Custom Package** menu option.
2. In the **Import Unity Package** box that pops up, you can select everything under **Plugin > Storage**. Uncheck everything else, as it is not needed for this course.



3. Click the **Import** button to add the items to your project.
4. Go to the **Storage** folder under **Plugins** in the Project view and select the following plugins *only*:
 - Microsoft.Data.Edm
 - Microsoft.Data.OData
 - Microsoft.WindowsAzure.Storage
 - Newtonsoft.Json
 - System.Spatial



5. With *these specific plugins* selected, **uncheck Any Platform** and **uncheck WSAPlayer** then click **Apply**.

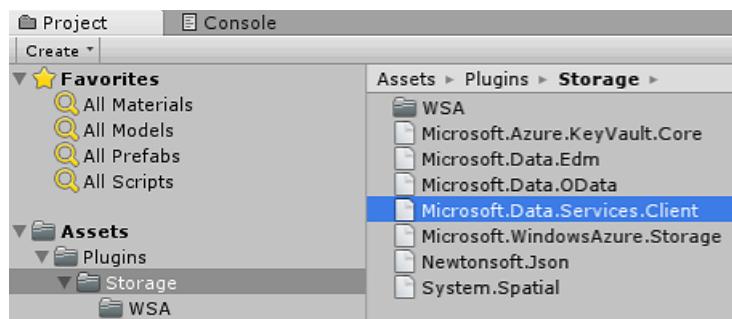


NOTE

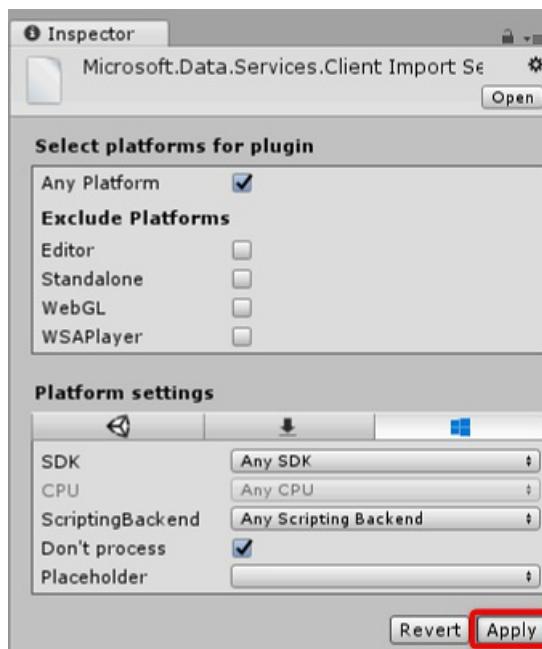
We are marking these particular plugins to only be used in the Unity Editor. This is because there are different versions of the same plugins in the WSA folder that will be used after the project is exported from Unity.

6. In the **Storage** plugin folder, select only:

- Microsoft.Data.Services.Client



7. Check the **Don't Process** box under **Platform Settings** and click **Apply**.



NOTE

We are marking this plugin "Don't process", because the Unity assembly patcher has difficulty processing this plugin. The plugin will still work even though it is not processed.

Chapter 9 - Create the TableToScene class in the Desktop Unity project

You now need to create the scripts containing the code to run this application.

The first script you need to create is **TableToScene**, which is responsible for:

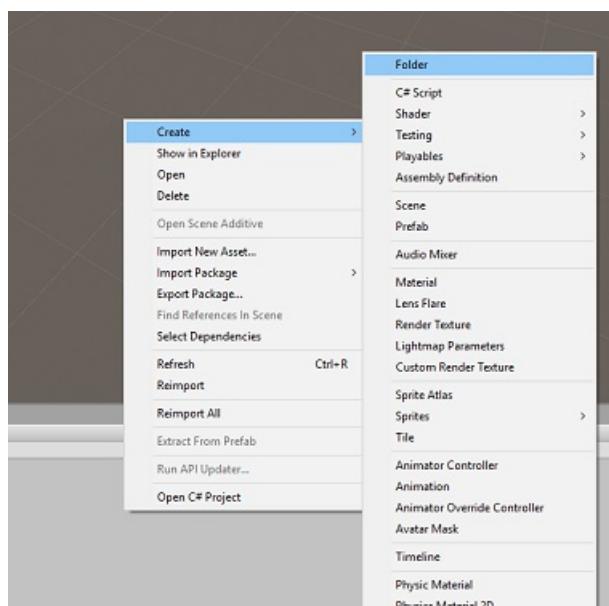
- Reading entities within the Azure Table.
- Using the Table data, determine which objects to spawn, and in which position.

The second script you need to create is **CloudScene**, which is responsible for:

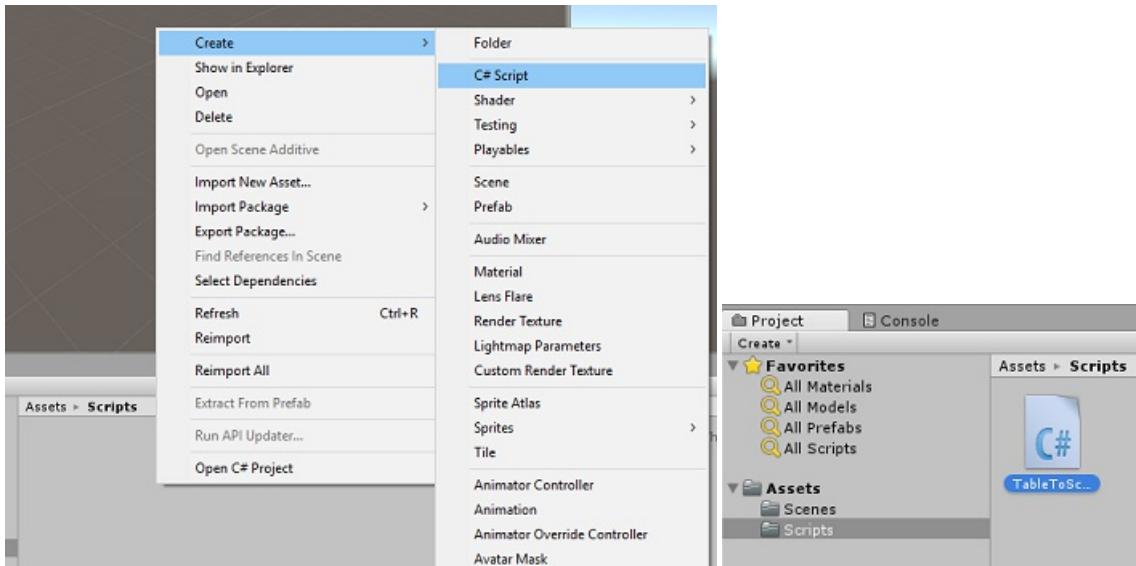
- Registering the left-click event, to allow the user to drag objects around the scene.
- Serializing the object data from this Unity scene, and sending it to the Azure Function App.

To create this class:

1. Right-click in the **Asset** Folder located in the Project Panel, **Create > Folder**. Name the folder **Scripts**.



2. Double click on the folder just created, to open it.
3. Right-click inside the **Scripts** folder, click **Create C# Script**. Name the script **TableToScene**.



4. Double-click on the script to open it in Visual Studio 2017.

5. Add the following namespaces:

```
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.Auth;
using Microsoft.WindowsAzure.Storage.Table;
using UnityEngine;
```

6. Within the class, insert the following variables:

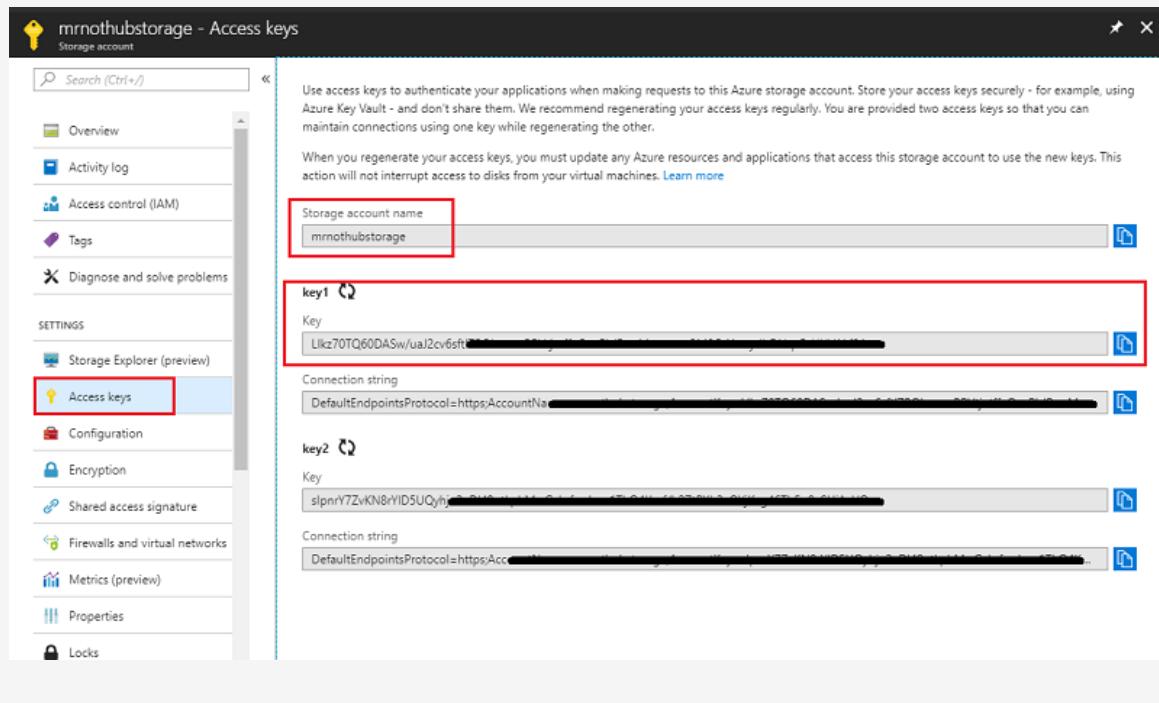
```
/// <summary>
/// allows this class to behave like a singleton
/// </summary>
public static TableToScene instance;

/// <summary>
/// Insert here you Azure Storage name
/// </summary>
private string accountName = " -- Insert your Azure Storage name -- ";

/// <summary>
/// Insert here you Azure Storage key
/// </summary>
private string accountKey = " -- Insert your Azure Storage key -- ";
```

NOTE

Substitute the **accountName** value with your Azure Storage Service name and **accountKey** value with the key value found in the Azure Storage Service, in the Azure Portal (See Image below).



The screenshot shows the 'Access keys' page for an Azure storage account named 'mrnothubstorage'. The left sidebar lists various account settings like Overview, Activity log, Access control (IAM), Tags, and Diagnose and solve problems. Under 'SETTINGS', 'Access keys' is selected and highlighted with a red box. On the right, there are two sets of access keys: 'key1' and 'key2'. Each key has a 'Key' field containing a long, randomly generated string. The 'key1' section is also highlighted with a red box.

7. Now add the **Start()** and **Awake()** methods to initialize the class.

```
/// <summary>
/// Triggers before initialization
/// </summary>
void Awake()
{
    // static instance of this class
    instance = this;
}

/// <summary>
/// Use this for initialization
/// </summary>
void Start()
{
    // Call method to populate the scene with new objects as
    // specified in the Azure Table
    PopulateSceneFromTableAsync();
}
```

8. Within the **TableToScene** class, add the method that will retrieve the values from the Azure Table and use them to spawn the appropriate primitives in the scene.

```
/// <summary>
/// Populate the scene with new objects as specified in the Azure Table
/// </summary>
private async void PopulateSceneFromTableAsync()
{
    // Obtain credentials for the Azure Storage
    StorageCredentials creds = new StorageCredentials(accountName, accountKey);

    // Storage account
    CloudStorageAccount account = new CloudStorageAccount(creds, useHttps: true);

    // Storage client
```

```

CloudTableClient client = account.CreateCloudTableClient();

// Table reference
CloudTable table = client.GetTableReference("SceneObjectsTable");

TableContinuationToken token = null;

// Query the table for every existing Entity
do
{
    // Queries the whole table by breaking it into segments
    // (would happen only if the table had huge number of Entities)
    TableQuerySegment<AzureTableEntity> queryResult = await table.ExecuteQuerySegmentedAsync(new
TableQuery<AzureTableEntity>(), token);

    foreach (AzureTableEntity entity in queryResult.Results)
    {
        GameObject newSceneGameObject = null;
        Color newColor;

        // check for the Entity Type and spawn in the scene the appropriate Primitive
        switch (entity.Type)
        {
            case "Cube":
                // Create a Cube in the scene
                newSceneGameObject = GameObject.CreatePrimitive(PrimitiveType.Cube);
                newColor = Color.blue;
                break;

            case "Sphere":
                // Create a Sphere in the scene
                newSceneGameObject = GameObject.CreatePrimitive(PrimitiveType.Sphere);
                newColor = Color.red;
                break;

            case "Cylinder":
                // Create a Cylinder in the scene
                newSceneGameObject = GameObject.CreatePrimitive(PrimitiveType.Cylinder);
                newColor = Color.yellow;
                break;
            default:
                newColor = Color.white;
                break;
        }

        newSceneGameObject.name = entity.RowKey;

        newSceneGameObject.GetComponent<MeshRenderer>().material = new
Material(Shader.Find("Diffuse"))
{
    color = newColor
};

        //check for the Entity X,Y,Z and move the Primitive at those coordinates
        newSceneGameObject.transform.position = new Vector3((float)entity.X, (float)entity.Y,
(entity.Z));
    }

    // if the token is null, it means there are no more segments left to query
    token = queryResult.ContinuationToken;
}

while (token != null);
}

```

9. Outside the **TableToScene** class, you need to define the class used by the application to serialize and deserialize the **Table Entities**.

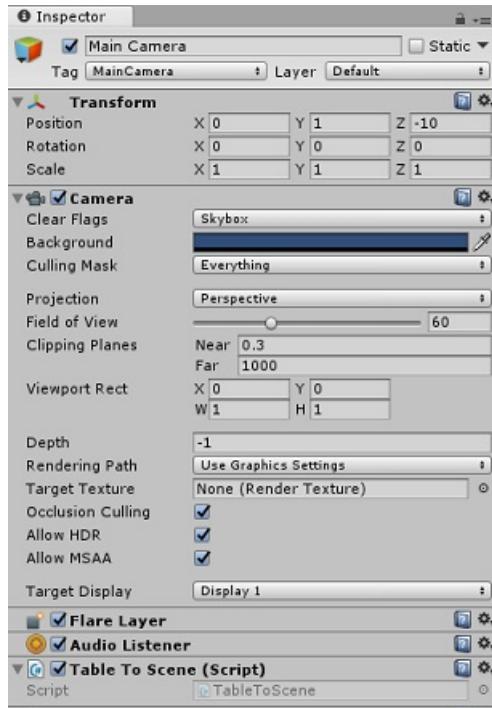
```

/// <summary>
/// This objects is used to serialize and deserialize the Azure Table Entity
/// </summary>
[System.Serializable]
public class AzureTableEntity : TableEntity
{
    public AzureTableEntity(string partitionKey, string rowKey)
        : base(partitionKey, rowKey) { }

    public AzureTableEntity() { }
    public string Type { get; set; }
    public double X { get; set; }
    public double Y { get; set; }
    public double Z { get; set; }
}

```

10. Make sure you **Save** before going back to the Unity Editor.
11. Click the **Main Camera** from the **Hierarchy** panel, so that its properties appear in the **Inspector**.
12. With the **Scripts** folder open, select the script **TableToScene** file and drag it onto the **Main Camera**. The result should be as below:



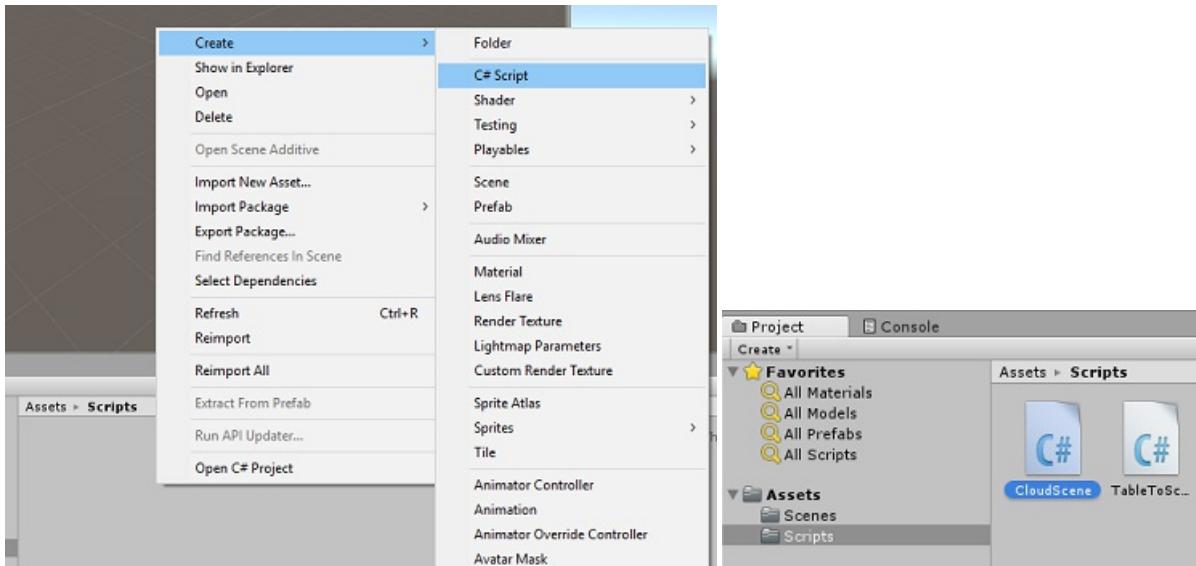
Chapter 10 - Create the CloudScene class in the Desktop Unity Project

The second script you need to create is **CloudScene**, which is responsible for:

- Registering the left-click event, to allow the user to drag objects around the scene.
- Serializing the object data from this Unity scene, and sending it to the Azure Function App.

To create the second script:

1. Right-click inside the **Scripts** folder, click **Create, C# Script**. Name the script **CloudScene**



2. Add the following namespaces:

```
using Newtonsoft.Json;
using System.Collections;
using System.Text;
using System.Threading.Tasks;
using UnityEngine;
using UnityEngine.Networking;
```

3. Insert the following variables:

```
/// <summary>
/// Allows this class to behave like a singleton
/// </summary>
public static CloudScene instance;

/// <summary>
/// Insert here your Azure Function Url
/// </summary>
private string azureFunctionEndpoint = "--Insert here your Azure Function Endpoint--";

/// <summary>
/// Flag for object being moved
/// </summary>
private bool gameObjHasMoved;

/// <summary>
/// Transform of the object being dragged by the mouse
/// </summary>
private Transform gameObjHeld;

/// <summary>
/// Class hosted in the TableToScene script
/// </summary>
private AzureTableEntity azureTableEntity;
```

4. Substitute the **azureFunctionEndpoint** value with your Azure Function App URL found in the Azure Function App Service, in the Azure Portal, as shown in the image below:

```

1 #r "Microsoft.WindowsAzure.Storage"
2
3 using System;
4 using Microsoft.WindowsAzure.Storage;
5 using Microsoft.WindowsAzure.Storage.Table;
6 using Microsoft.Azure.NotificationHubs;
7 using Newtonsoft.Json;
8
9 public static async Task Run(UnityGameObject gameObj, CloudTable table,
10    IAsyncCollector<Notification> notification, TraceWriter log)
11 {
12     //RowKey of the table object to be changed
13     string rowKey = gameObj.RowKey;
14
15     //Retrieve the table object by its RowKey
16     TableOperation operation =
17         TableOperation.Retrieve<UnityGameObject>("UnityPartitionKey", rowKey);
18
19     TableResult result = table.Execute(operation);
20
21     //Create a UnityGameObject so to set its parameters
22     UnityGameObject existingGameObj =

```

5. Now add the **Start()** and **Awake()** methods to initialize the class.

```

/// <summary>
/// Triggers before initialization
/// </summary>
void Awake()
{
    // static instance of this class
    instance = this;
}

/// <summary>
/// Use this for initialization
/// </summary>
void Start()
{
    // initialise an AzureTableEntity
    azureTableEntity = new AzureTableEntity();
}

```

6. Within the **Update()** method, add the following code that will detect the mouse input and drag, which will in turn move GameObjects in the scene. If the user has dragged and dropped an object, it will pass the name and coordinates of the object to the method **UpdateCloudScene()**, which will call the Azure Function App service, which will update the Azure table and trigger the notification.

```

/// <summary>
/// Update is called once per frame
/// </summary>
void Update()
{
    //Enable Drag if button is held down
    if (Input.GetMouseButton(0))
    {
        // Get the mouse position
        Vector3 mousePosition = new Vector3(Input.mousePosition.x, Input.mousePosition.y, 10);

        Vector3 objPos = Camera.main.ScreenToWorldPoint(mousePosition);

        Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);

        RaycastHit hit;

        // Raycast from the current mouse position to the object overlapped by the mouse
        if (Physics.Raycast(ray, out hit))
        {
            // update the position of the object "hit" by the mouse
            hit.transform.position = objPos;

            gameObjHasMoved = true;

            gameObjHeld = hit.transform;
        }
    }

    // check if the left button mouse is released while holding an object
    if (Input.GetMouseButtonUp(0) && gameObjHasMoved)
    {
        gameObjHasMoved = false;

        // Call the Azure Function that will update the appropriate Entity in the Azure Table
        // and send a Notification to all subscribed Apps
        Debug.Log("Calling Azure Function");

        StartCoroutine(UpdateCloudScene(gameObjHeld.name, gameObjHeld.position.x,
        gameObjHeld.position.y, gameObjHeld.position.z));
    }
}

```

7. Now add the **UpdateCloudScene()** method, as below:

```

private IEnumerator UpdateCloudScene(string objName, double xPos, double yPos, double zPos)
{
    WWWForm form = new WWWForm();

    // set the properties of the AzureTableEntity
    azureTableEntity.RowKey = objName;

    azureTableEntity.X = xPos;

    azureTableEntity.Y = yPos;

    azureTableEntity.Z = zPos;

    // Serialize the AzureTableEntity object to be sent to Azure
    string jsonObject = JsonConvert.SerializeObject(azureTableEntity);

    using (UnityWebRequest www = UnityWebRequest.Post(azureFunctionEndpoint, jsonObject))
    {
        byte[] jsonToSend = new System.Text.UTF8Encoding().GetBytes(jsonObject);

        www.uploadHandler = new UploadHandlerRaw(jsonToSend);

        www.uploadHandler.contentType = "application/json";

        www.downloadHandler = new DownloadHandlerBuffer();

        www.SetRequestHeader("Content-Type", "application/json");

        yield return www.SendWebRequest();

        string response = www.responseCode.ToString();
    }
}

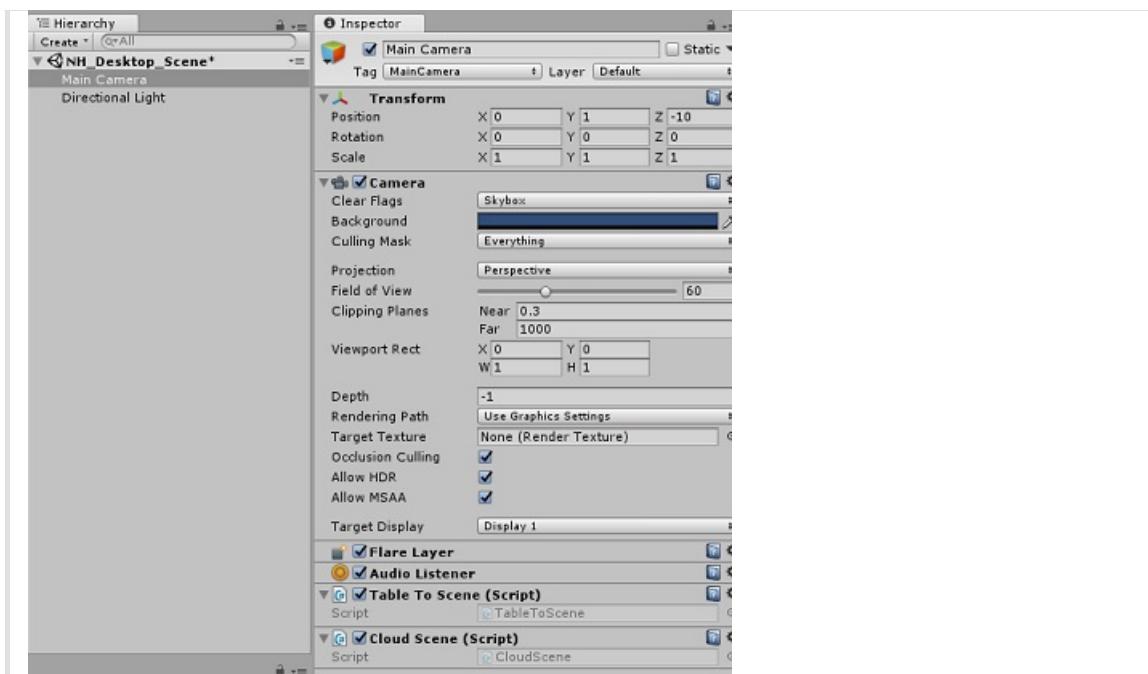
```

8. Save the code and return to Unity

9. Drag the **CloudScene** script onto the **Main Camera**.

a. Click the **Main Camera** from the **Hierarchy** panel, so that its properties appear in the **Inspector**.

b. With the **Scripts** folder open, select the **CloudScene** script and drag it onto the **Main Camera**. The result should be as below:

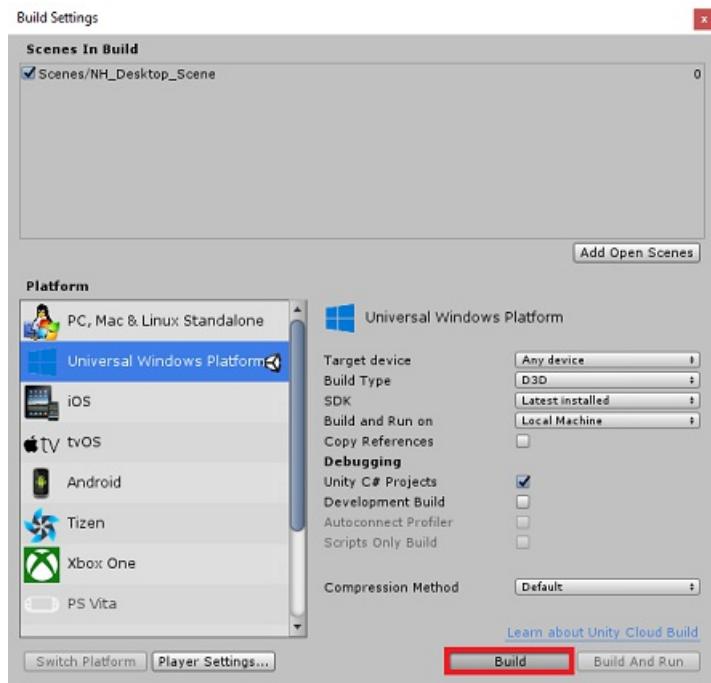


Chapter 11 - Build the Desktop Project to UWP

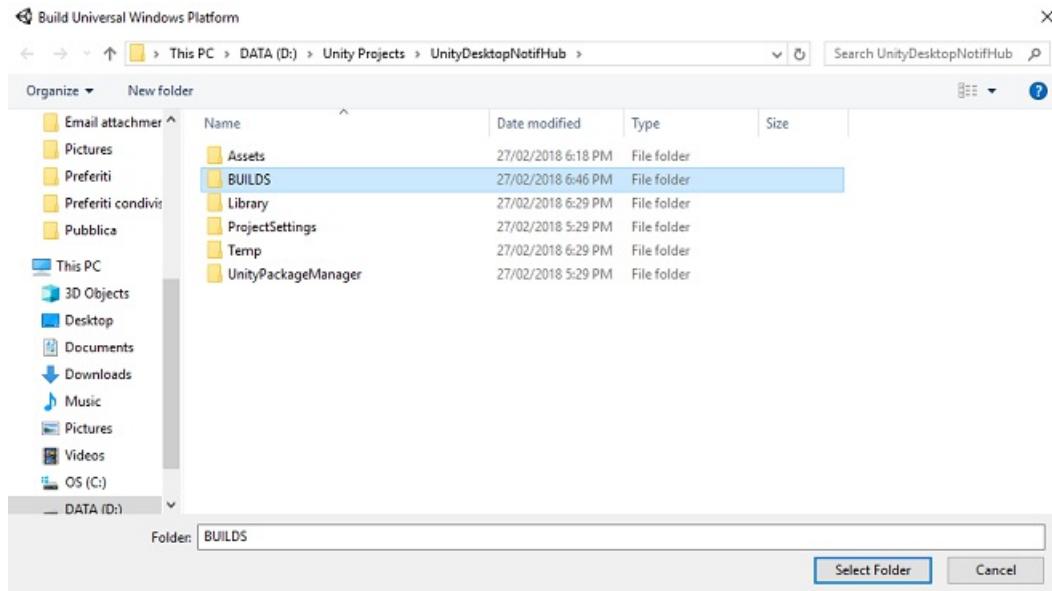
Everything needed for the Unity section of this project has now been completed.

1. Navigate to **Build Settings** (**File > Build Settings**).

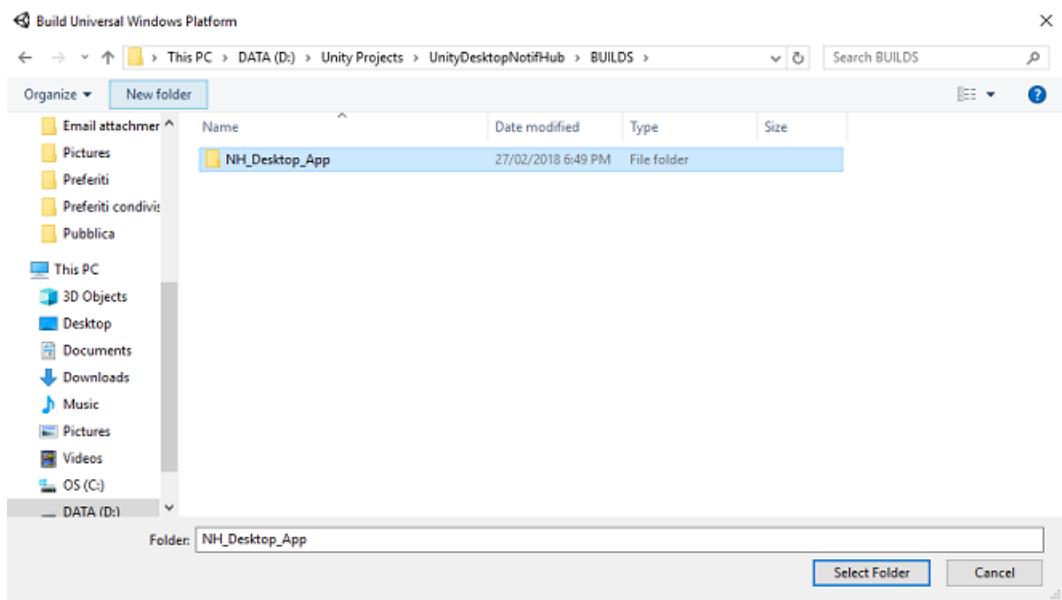
2. From the **Build Settings** window, click **Build**.



3. A **File Explorer** window will popup, prompting you for a location to Build. Create a new folder (by clicking **New Folder** in the top-left corner), and name it **BUILDS**.



a. Open the new **BUILDS** folder, and create another folder (using **New Folder** once more), and name it **NH/Desktop_App**.

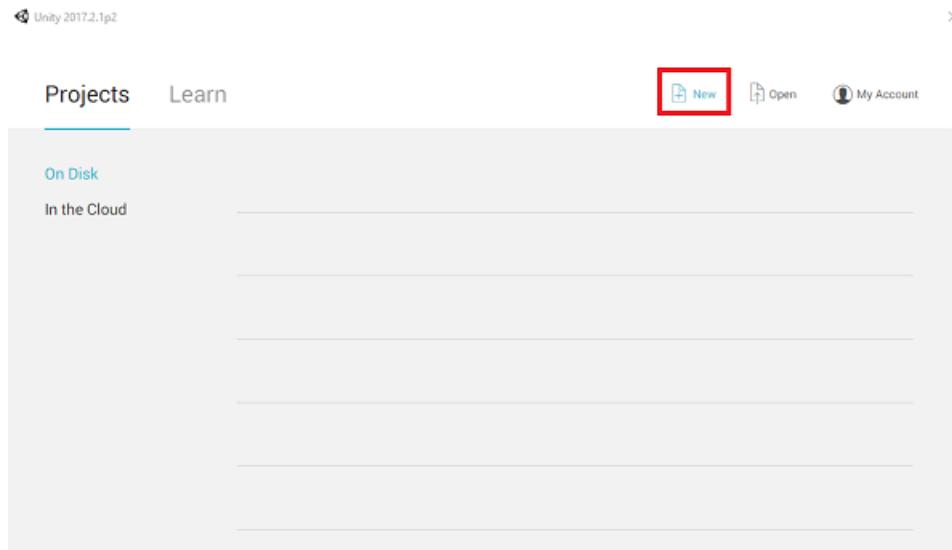


- b. With the **NH/Desktop_App** selected. click **Select Folder**. The project will take a minute or so to build.
4. Following build, **File Explorer** will appear showing you the location of your new project. No need to open it, though, as you need to create the other Unity project first, in the next few Chapters.

Chapter 12 - Set up Mixed Reality Unity Project

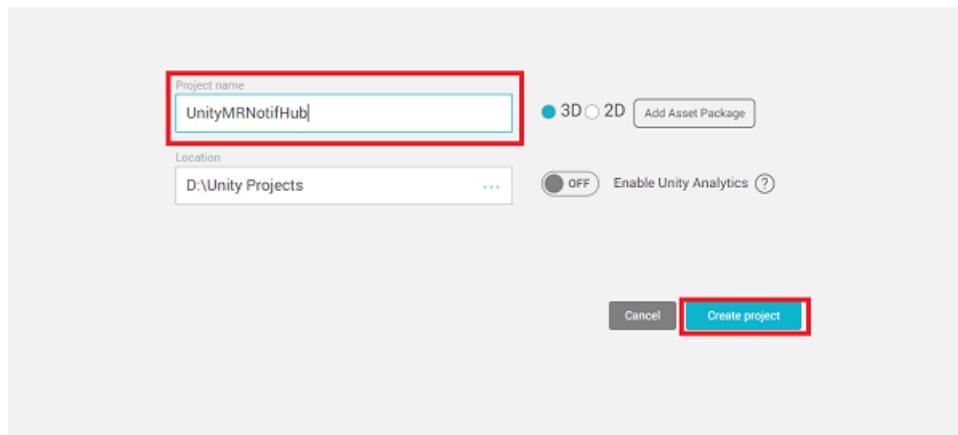
The following is a typical set up for developing with the mixed reality, and as such, is a good template for other projects.

1. Open **Unity** and click **New**.

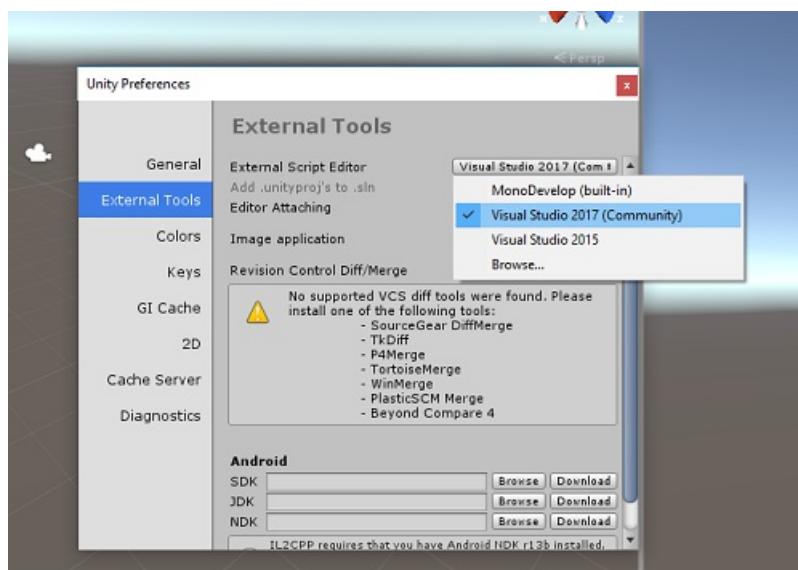


2. You will now need to provide a Unity Project name, insert **UnityMRNotifHub**. Make sure the project type is set to **3D**. Set the **Location** to somewhere appropriate for you (remember, closer to root directories is better). Then, click **Create project**.

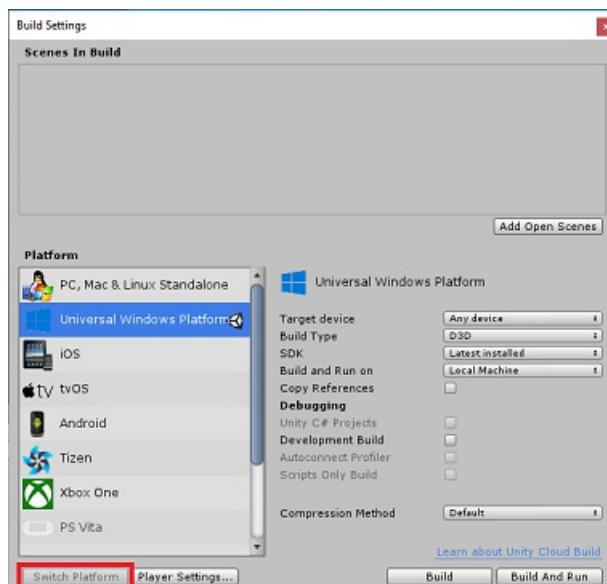
Projects Learn

[New](#) [Open](#) [My Account](#)

3. With Unity open, it is worth checking the default **Script Editor** is set to **Visual Studio**. Go to **Edit > Preferences** and then from the new window, navigate to **External Tools**. Change **External Script Editor** to **Visual Studio 2017**. Close the **Preferences** window.



4. Next, go to **File > Build Settings** and switch the platform to **Universal Windows Platform**, by clicking on the **Switch Platform** button.



5. Go to **File > Build Settings** and make sure that:

a. **Target Device** is set to **Any Device**

For the Microsoft HoloLens, set **Target Device** to *HoloLens*.

b. **Build Type** is set to **D3D**

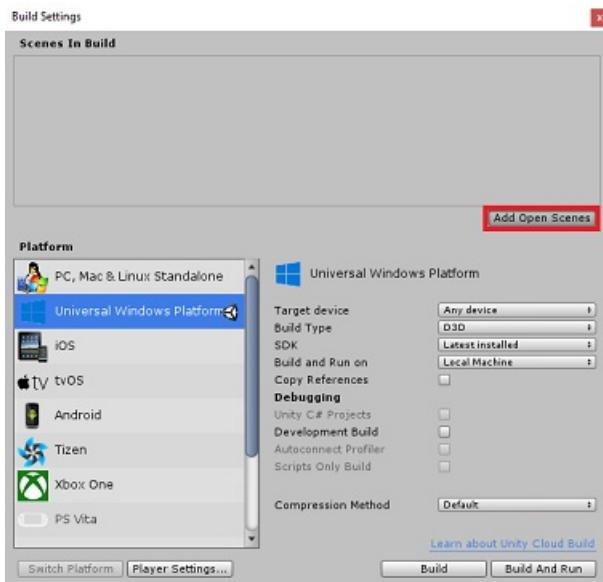
c. **SDK** is set to **Latest installed**

d. **Visual Studio Version** is set to **Latest installed**

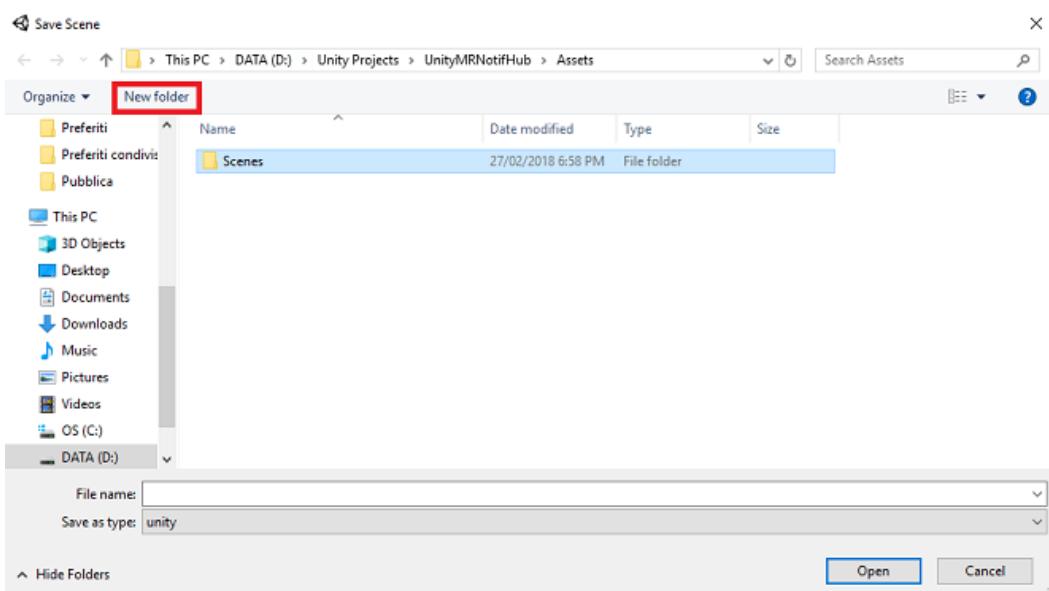
e. **Build and Run** is set to **Local Machine**

f. While here, it is worth saving the scene, and adding it to the build.

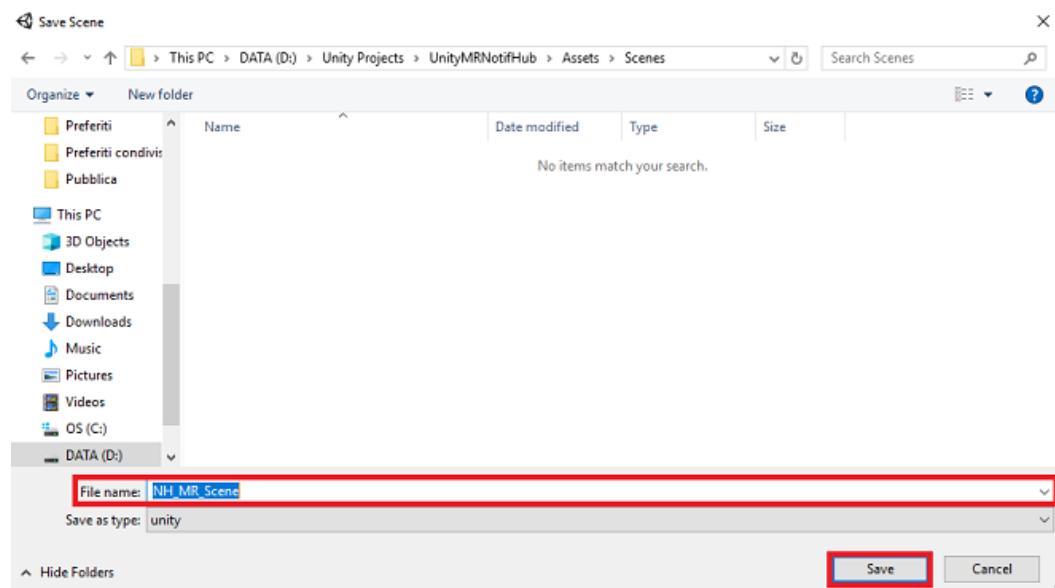
a. Do this by selecting **Add Open Scenes**. A save window will appear.



b. Create a new folder for this, and any future, scene, then select the **New folder** button, to create a new folder, name it **Scenes**.

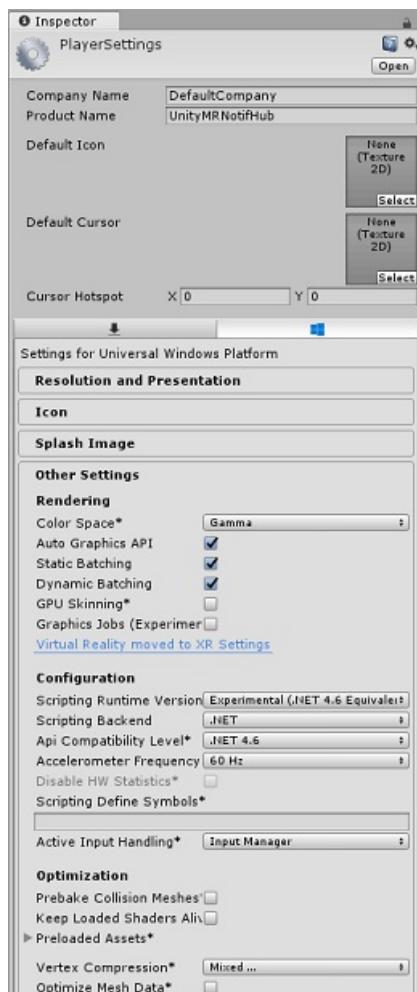


c. Open your newly created **Scenes** folder, and then in the **File name:** text field, type **NH_MR_Scene**, then press **Save**.



g. The remaining settings, in **Build Settings**, should be left as default for now.

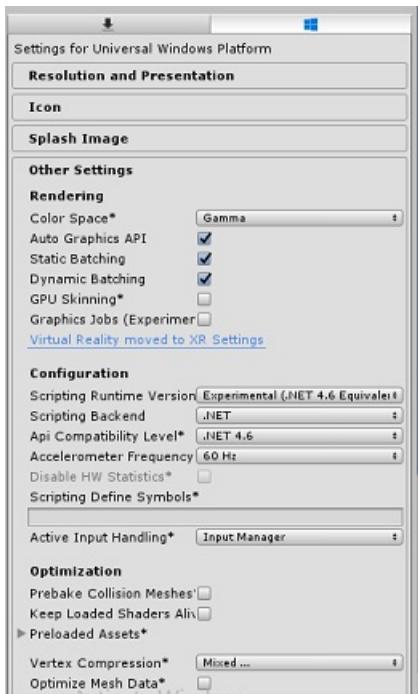
6. In the same window click on the **Player Settings** button, this will open the related panel in the space where the **Inspector** is located.



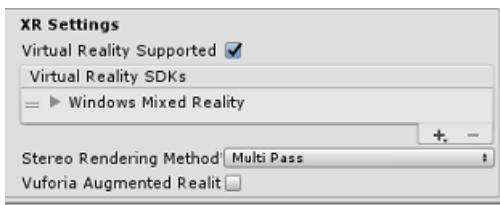
7. In this panel, a few settings need to be verified:

a. In the **Other Settings** tab:

- a. **Scripting Runtime Version** should be **Experimental (.NET 4.6 Equivalent)**
- b. **Scripting Backend** should be **.NET**
- c. **API Compatibility Level** should be **.NET 4.6**

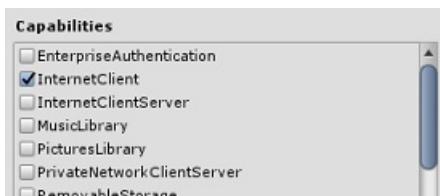


- b. Further down the panel, in **XR Settings** (found below **Publish Settings**), tick **Virtual Reality Supported**, make sure the **Windows Mixed Reality SDK** is added



- c. Within the **Publishing Settings** tab, under **Capabilities**, check:

- **InternetClient**



8. Back in **Build Settings** *Unity C# Projects* is no longer greyed out: tick the checkbox next to this.
9. With these changes done, close the Build Settings window.
10. Save your Scene and Project *File Save Scene/ File Save Project*.

IMPORTANT

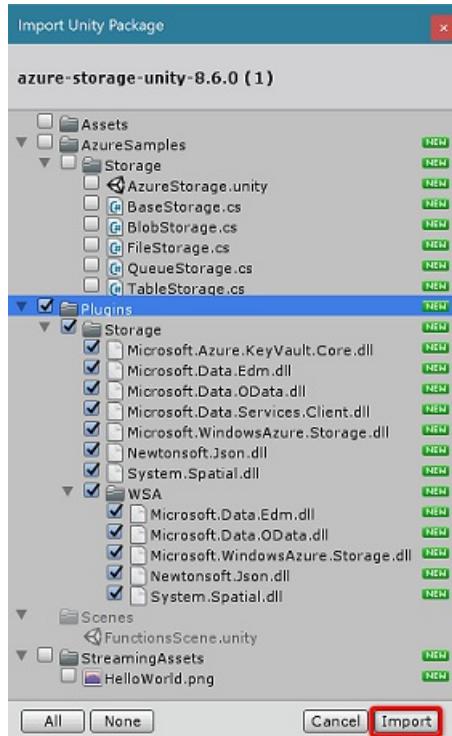
If you wish to skip the *Unity Set up* component for this project (mixed reality App), and continue straight into code, feel free to [download this .unitypackage](#), import it into your project as a **Custom Package**, and then continue from [Chapter 14](#). You will still need to add the script components.

Chapter 13 - Importing the DLLs in the Mixed Reality Unity Project

You will be using Azure Storage for Unity library (which uses the .Net SDK for Azure). Please follow this [link on how to use Azure Storage with Unity](#). There is currently a known issue in Unity which requires plugins to be reconfigured after import. These steps (4 - 7 in this section) will no longer be required after the bug has been resolved.

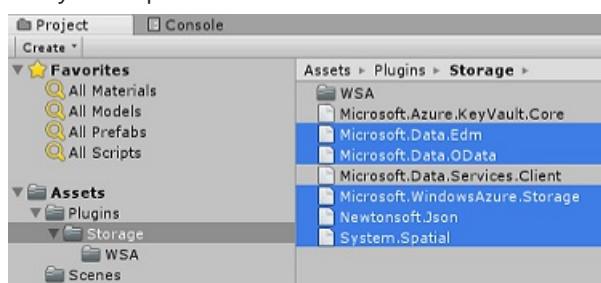
To import the SDK into your own project, make sure you have downloaded the latest [.unitypackage](#). Then, do the following:

1. Add the .unitypackage you downloaded from the above, to Unity by using the **Assets > Import Package > Custom Package** menu option.
2. In the **Import Unity Package** box that pops up, you can select everything under **Plugin > Storage**.

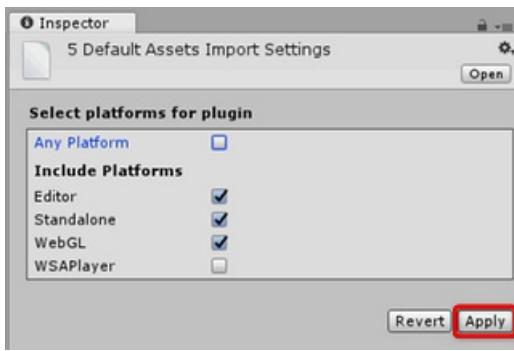


3. Click the **Import** button to add the items to your project.
4. Go to the **Storage** folder under **Plugins** in the Project view and select the following plugins *only*:

- Microsoft.Data.Edm
- Microsoft.Data.OData
- Microsoft.WindowsAzure.Storage
- Newtonsoft.Json
- System.Spatial



5. With *these specific plugins* selected, **uncheck Any Platform** and **uncheck WSAPlayer** then click **Apply**.

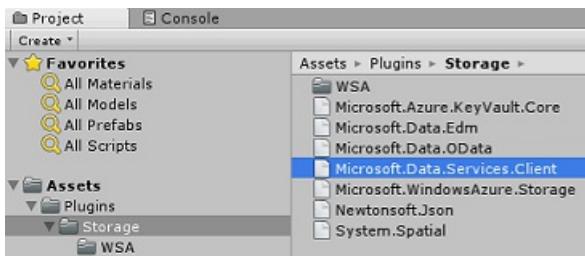


NOTE

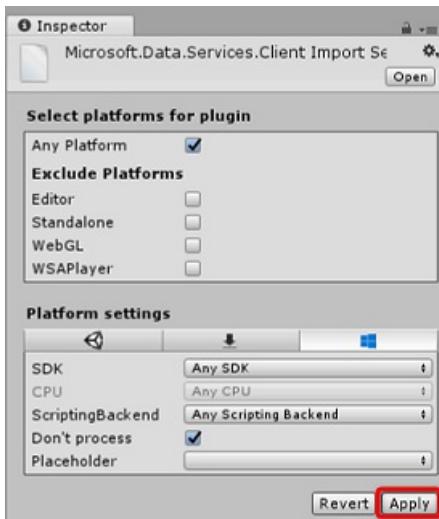
You are marking these particular plugins to only be used in the Unity Editor. This is because there are different versions of the same plugins in the WSA folder that will be used after the project is exported from Unity.

- In the **Storage** plugin folder, select only:

- Microsoft.Data.Services.Client



- Check the **Don't Process** box under **Platform Settings** and click **Apply**.



NOTE

You are marking this plugin "Don't process" because the Unity assembly patcher has difficulty processing this plugin. The plugin will still work even though it isn't processed.

Chapter 14 - Creating the TableToScene class in the mixed reality Unity project

The **TableToScene** class is identical to the one explained in [Chapter 9](#). Create the same class in the mixed reality Unity Project following the same procedure explained in [Chapter 9](#).

Once you have completed this Chapter, both of your **Unity Projects** will have this class set up on the Main Camera.

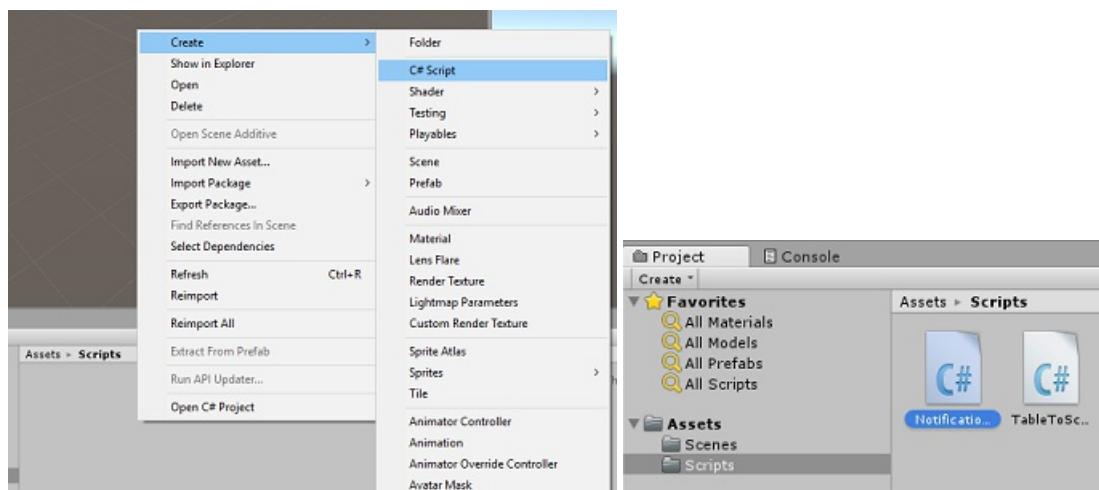
Chapter 15 - Creating the NotificationReceiver class in the Mixed Reality Unity Project

The second script you need to create is **NotificationReceiver**, which is responsible for:

- Registering the app with the Notification Hub at initialization.
- Listening to notifications coming from the Notification Hub.
- Deserializing the object data from received notifications.
- Move the GameObjects in the scene, based on the deserialized data.

To create the **NotificationReceiver** script:

1. Right-click inside the **Scripts** folder, click **Create, C# Script**. Name the script **NotificationReceiver**.



2. Double click on the script to open it.
3. Add the following namespaces:

```
//using Microsoft.WindowsAzure.Messaging;
using Newtonsoft.Json;
using System;
using System.Collections;
using UnityEngine;

#if UNITY_WSA_10_0 && !UNITY_EDITOR
using Windows.Networking.PushNotifications;
#endif
```

4. Insert the following variables:

```

/// <summary>
/// allows this class to behave like a singleton
/// </summary>
public static NotificationReceiver instance;

/// <summary>
/// Value set by the notification, new object position
/// </summary>
Vector3 newObjPosition;

/// <summary>
/// Value set by the notification, object name
/// </summary>
string gameObjectName;

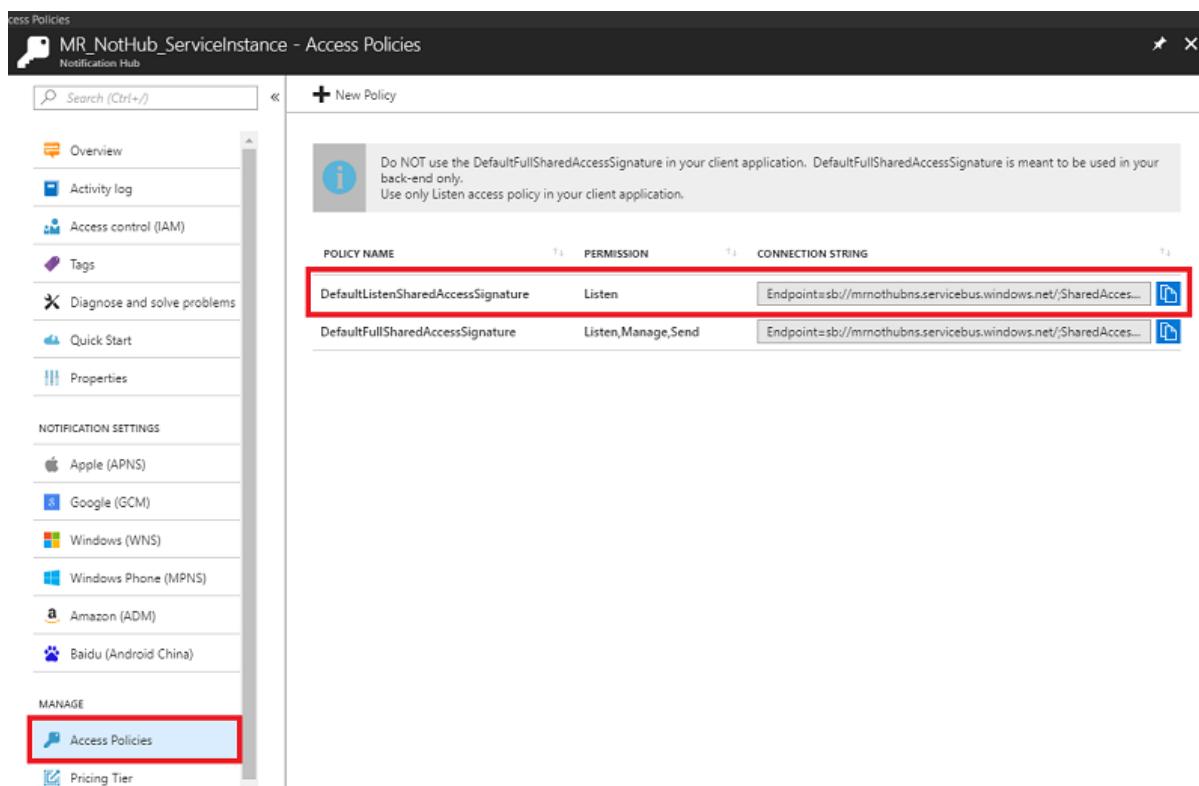
/// <summary>
/// Value set by the notification, new object position
/// </summary>
bool notifReceived;

/// <summary>
/// Insert here your Notification Hub Service name
/// </summary>
private string hubName = " -- Insert the name of your service -- ";

/// <summary>
/// Insert here your Notification Hub Service "Listen endpoint"
/// </summary>
private string hubListenEndpoint = "-Insert your Notification Hub Service Listen endpoint-";

```

5. Substitute the **hubName** value with your Notification Hub Service name, and **hubListenEndpoint** value with the endpoint value found in the Access Policies tab, Azure Notification Hub Service, in the Azure Portal (see image below).



POLICY NAME	PERMISSION	CONNECTION STRING
DefaultListenSharedAccessSignature	Listen	Endpoint=sb://mrnothubns.servicebus.windows.net/SharedAccessSignature
DefaultFullSharedAccessSignature	Listen,Manage,Send	Endpoint=sb://mrnothubns.servicebus.windows.net/SharedAccessSignature

6. Now add the **Start()** and **Awake()** methods to initialize the class.

```

/// <summary>
/// Triggers before initialization
/// </summary>
void Awake()
{
    // static instance of this class
    instance = this;
}

/// <summary>
/// Use this for initialization
/// </summary>
void Start()
{
    // Register the App at launch
    InitNotificationsAsync();

    // Begin listening for notifications
    StartCoroutine(WaitForNotification());
}

```

7. Add the **WaitForNotification** method to allow the app to receive notifications from the Notification Hub Library without clashing with the Main Thread:

```

/// <summary>
/// This notification listener is necessary to avoid clashes
/// between the notification hub and the main thread
/// </summary>
private IEnumerator WaitForNotification()
{
    while (true)
    {
        // Checks for notifications each second
        yield return new WaitForSeconds(1f);

        if (notifReceived)
        {
            // If a notification is arrived, moved the appropriate object to the new position
            GameObject.Find(gameObjectName).transform.position = newObjPosition;

            // Reset the flag
            notifReceived = false;
        }
    }
}

```

8. The following method, **InitNotificationAsync()**, will register the application with the notification Hub Service at initialization. The code is commented out as Unity will not be able to Build the project. You will remove the comments when you import the Azure Messaging Nuget package in Visual Studio.

```

/// <summary>
/// Register this application to the Notification Hub Service
/// </summary>
private async void InitNotificationsAsync()
{
    // PushNotificationChannel channel = await
PushNotificationChannelManager.CreatePushNotificationChannelForApplicationAsync();

    // NotificationHub hub = new NotificationHub(hubName, hubListenEndpoint);

    // Registration result = await hub.RegisterNativeAsync(channel.Uri);

    // If registration was successful, subscribe to Push Notifications
    // if (result.RegistrationId != null)
    //{
    //     Debug.Log($"Registration Successful: {result.RegistrationId}");
    //     channel.PushNotificationReceived += Channel_PushNotificationReceived;
    //}
}

```

9. The following handler, **Channel_PushNotificationReceived()**, will be triggered every time a notification is received. It will deserialize the notification, which will be the Azure Table Entity that has been moved on the Desktop Application, and then move the corresponding GameObject in the MR scene to the same position.

IMPORTANT

The code is commented out because the code references the Azure Messaging library, which you will add after building the Unity project using the Nuget Package Manager, within Visual Studio. As such, the Unity project will not be able to build, unless it is commented out. Be aware, that should you build your project, and then wish to return to Unity, you will need to **re-comment** that code.

```

///// <summary>
///// Handler called when a Push Notification is received
///// </summary>
//private void Channel_PushNotificationReceived(PushNotificationChannel sender,
PushNotificationReceivedEventArgs args)
//{
//    Debug.Log("New Push Notification Received");
//
//    if (args.NotificationType == PushNotificationType.Raw)
//    {
//        // Raw content of the Notification
//        string jsonContent = args.RawNotification.Content;
//
//        // Deserialise the Raw content into an AzureTableEntity object
//        AzureTableEntity ate = JsonConvert.DeserializeObject<AzureTableEntity>(jsonContent);
//
//        // The name of the Game Object to be moved
//        gameObjectName = ate.RowKey;
//
//        // The position where the Game Object has to be moved
//        newObjPosition = new Vector3((float)ate.X, (float)ate.Y, (float)ate.Z);
//
//        // Flag that's a notification has been received
//        notifReceived = true;
//    }
//}

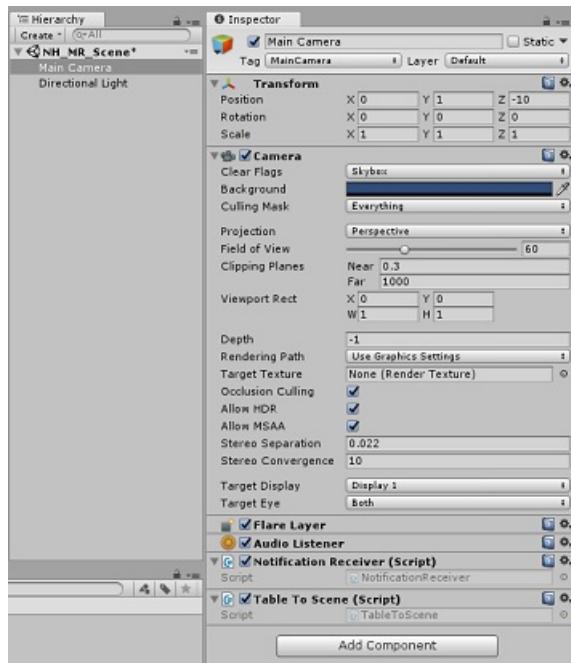
```

10. Remember to save your changes before going back to the Unity Editor.

11. Click the **Main Camera** from the **Hierarchy** panel, so that its properties appear in the **Inspector**.

12. With the **Scripts** folder open, select the **NotificationReceiver** script and drag it onto the **Main Camera**.

The result should be as below:



NOTE

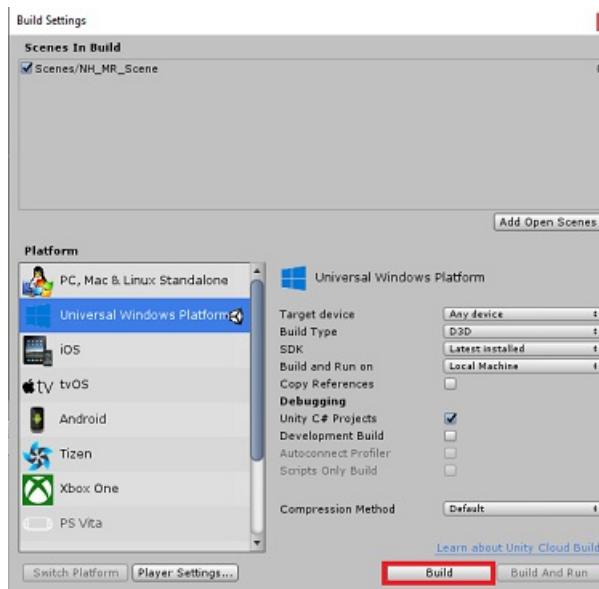
If you are developing this for the Microsoft HoloLens, you will need to update the **Main Camera**'s *Camera* component, so that:

- Clear Flags: Solid Color
- Background: Black

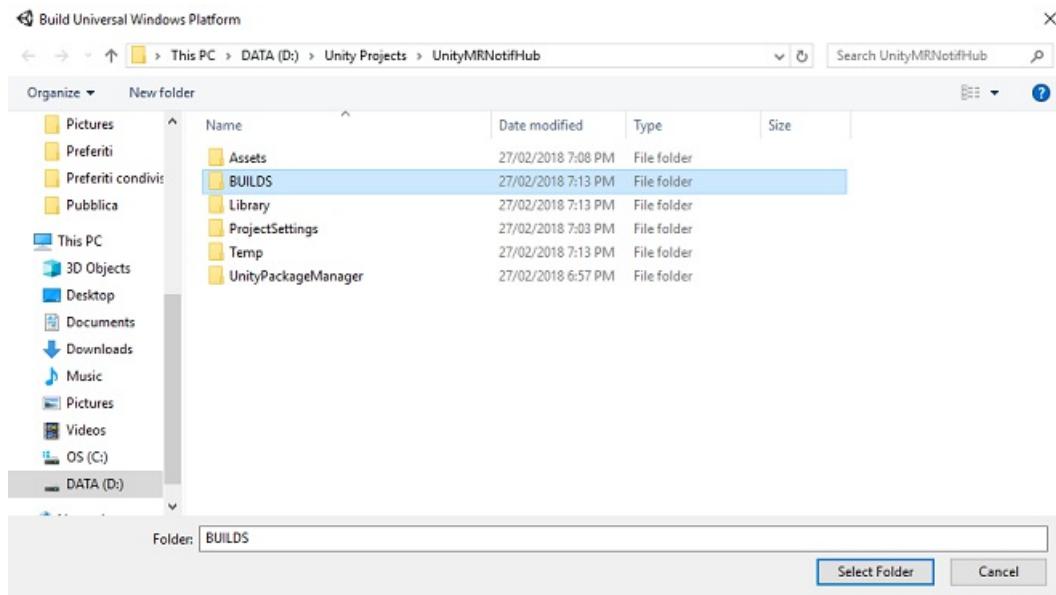
Chapter 16 - Build the Mixed Reality Project to UWP

This Chapter is identical to build process for the previous project. Everything needed for the Unity section of this project has now been completed, so it is time to build it from Unity.

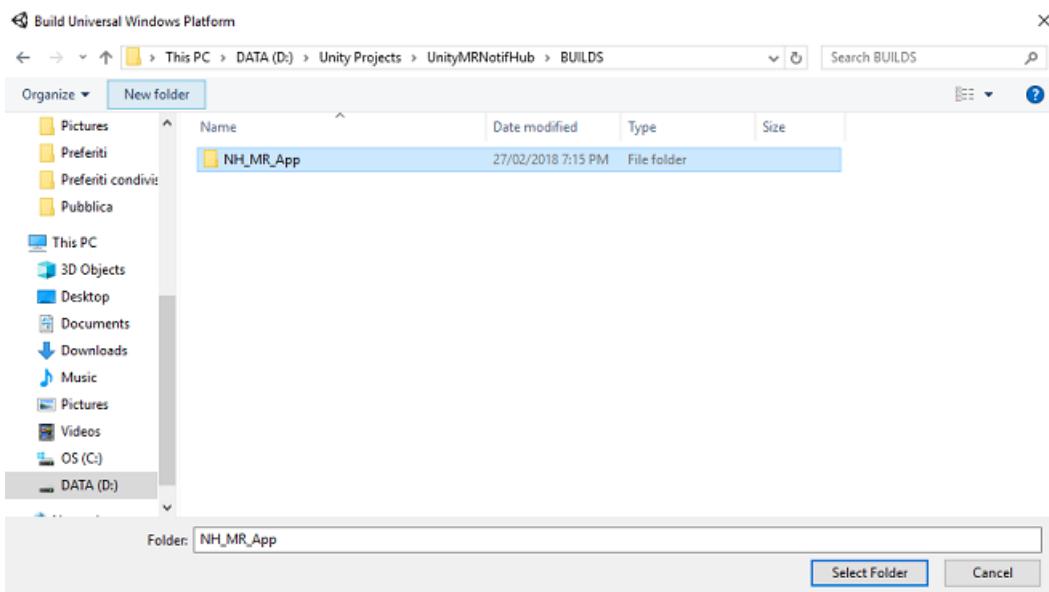
1. Navigate to **Build Settings** (**File > Build Settings...**).
2. From the **Build Settings** menu, ensure **Unity C# Projects*** is ticked (which will allow you to edit the scripts in this project, after build).
3. After this is done, click **Build**.



4. A **File Explorer** window will popup, prompting you for a location to Build. Create a new folder (by clicking **New Folder** in the top-left corner), and name it **BUILDS**.



- a. Open the new **BUILDS** folder, and create another folder (using **New Folder** once more), and name it **NH_MR_App**.



- b. With the **NH_MR_App** selected, click **Select Folder**. The project will take a minute or so to build.
5. Following the build, a **File Explorer** window will open at the location of your new project.

Chapter 17 - Add NuGet packages to the UnityMRNotifHub Solution

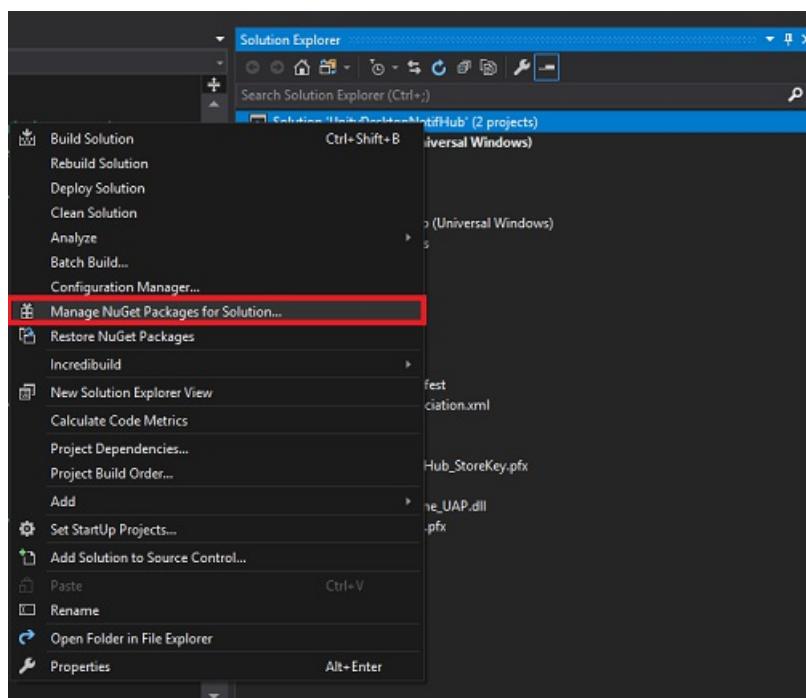
WARNING

Please remember that, once you add the following NuGet Packages (and uncomment the code in the next Chapter), the Code, when reopened within the Unity Project, will present errors. If you wish to go back and continue editing in the Unity Editor, you will need comment that erroneous code, and then uncomment again later, once you are back in Visual Studio.

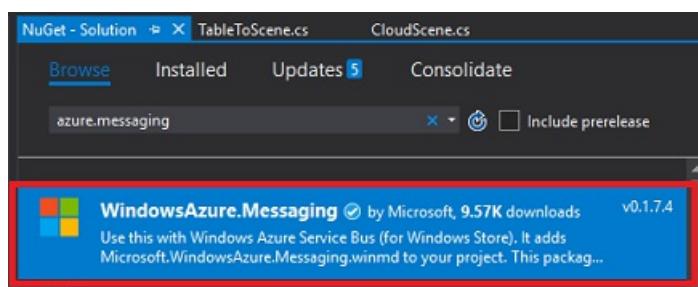
Once the mixed reality build has been completed, navigate to the mixed reality project, which you built, and double click on the solution (.sln) file within that folder, to open your solution with Visual Studio 2017. You will now need to add the **WindowsAzure.Messaging.managed** NuGet package; this is a library that is used to receive Notifications from the Notification Hub.

To import the NuGet package:

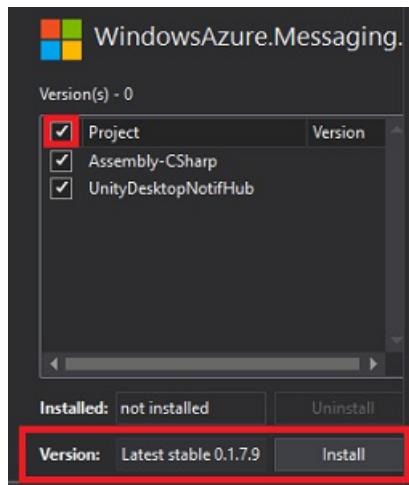
1. In the **Solution Explorer**, right click on your Solution
2. Click on **Manage NuGet Packages**.



3. Select the **Browse** tab and search for **WindowsAzure.Messaging.managed**.



4. Select the result (as shown below), and in the window to the right, select the checkbox next to **Project**. This will place a tick in the checkbox next to **Project**, along with the checkbox next to the **Assembly-CSharp** and **UnityMRNotifHub** project.



5. The version initially provided **may not** be compatible with this project. Therefore, click on the dropdown menu next to **Version**, and click **Version 0.1.7.9**, then click **Install**.
6. You have now finished installing the NuGet package. Find the commented code you entered in the **NotificationReceiver** class and remove the comments..

Chapter 18 - Edit UnityMRNotifHub application, NotificationReceiver class

Following having added the **NuGet Packages**, you will need to *uncomment* some of the code within the **NotificationReceiver** class.

This includes:

1. The namespace at the top:

```
using Microsoft.WindowsAzure.Messaging;
```

2. All the code within the **InitNotificationsAsync()** method:

```
/// <summary>
/// Register this application to the Notification Hub Service
/// </summary>
private async void InitNotificationsAsync()
{
    PushNotificationChannel channel = await
PushNotificationChannelManager.CreatePushNotificationChannelForApplicationAsync();

    NotificationHub hub = new NotificationHub(hubName, hubListenEndpoint);

    Registration result = await hub.RegisterNativeAsync(channel.Uri);

    // If registration was successful, subscribe to Push Notifications
    if (result.RegistrationId != null)
    {
        Debug.Log($"Registration Successful: {result.RegistrationId}");
        channel.PushNotificationReceived += Channel_PushNotificationReceived;
    }
}
```

WARNING

The code above has a comment in it: ensure that you have not accidentally *uncommented* that comment (as the code will not compile if you have!).

3. And, lastly, the **Channel_PushNotificationReceived** event:

```
/// <summary>
/// Handler called when a Push Notification is received
/// </summary>
private void Channel_PushNotificationReceived(PushNotificationChannel sender,
PushNotificationReceivedEventArgs args)
{
    Debug.Log("New Push Notification Received");

    if (args.NotificationType == PushNotificationType.Raw)
    {
        // Raw content of the Notification
        string jsonContent = args.RawNotification.Content;

        // Deserialize the Raw content into an AzureTableEntity object
        AzureTableEntity ate = JsonConvert.DeserializeObject<AzureTableEntity>(jsonContent);

        // The name of the Game Object to be moved
        gameObjectName = ate.RowKey;

        // The position where the Game Object has to be moved
        newObjPosition = new Vector3((float)ate.X, (float)ate.Y, (float)ate.Z);

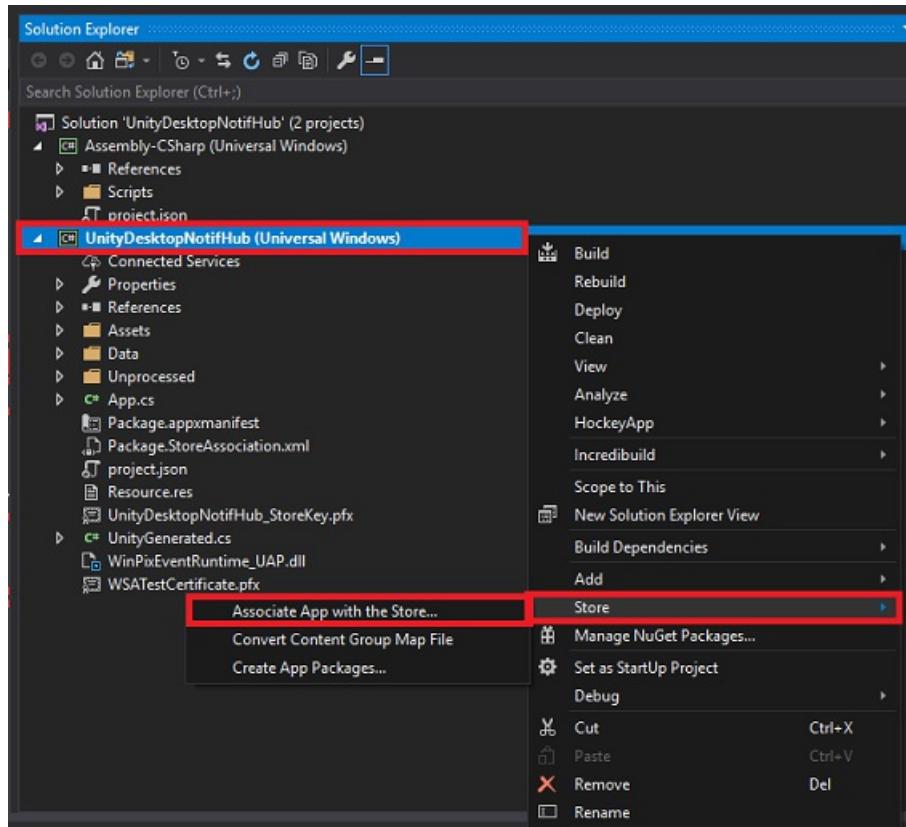
        // Flag thats a notification has been received
        notifReceived = true;
    }
}
```

With these uncommented, ensure that you save, and then proceed to the next Chapter.

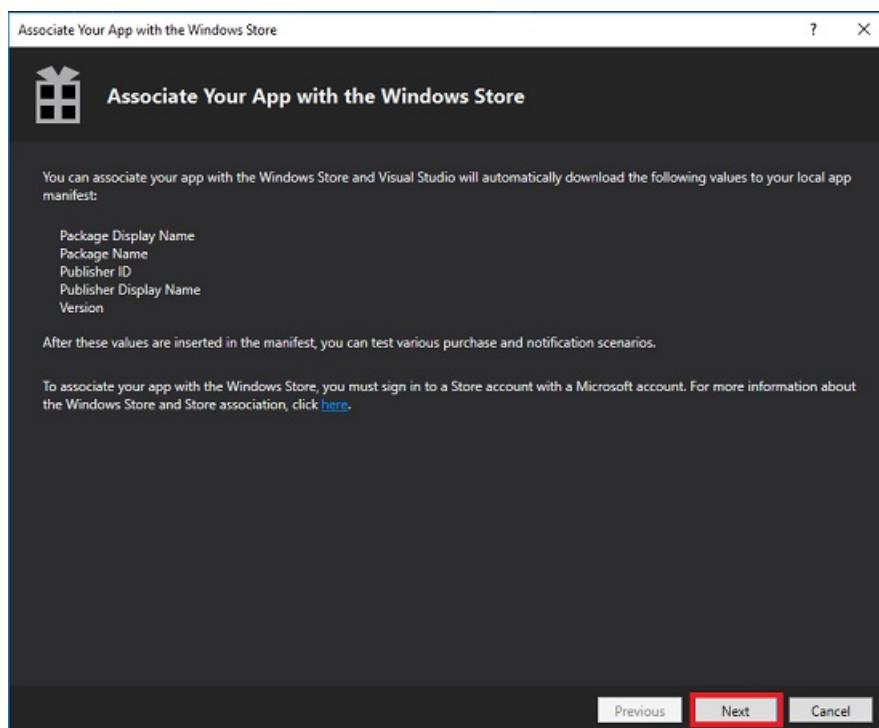
Chapter 19 - Associate the mixed reality project to the Store app

You now need to associate the **mixed reality** project to the Store App you created in at the start of the lab.

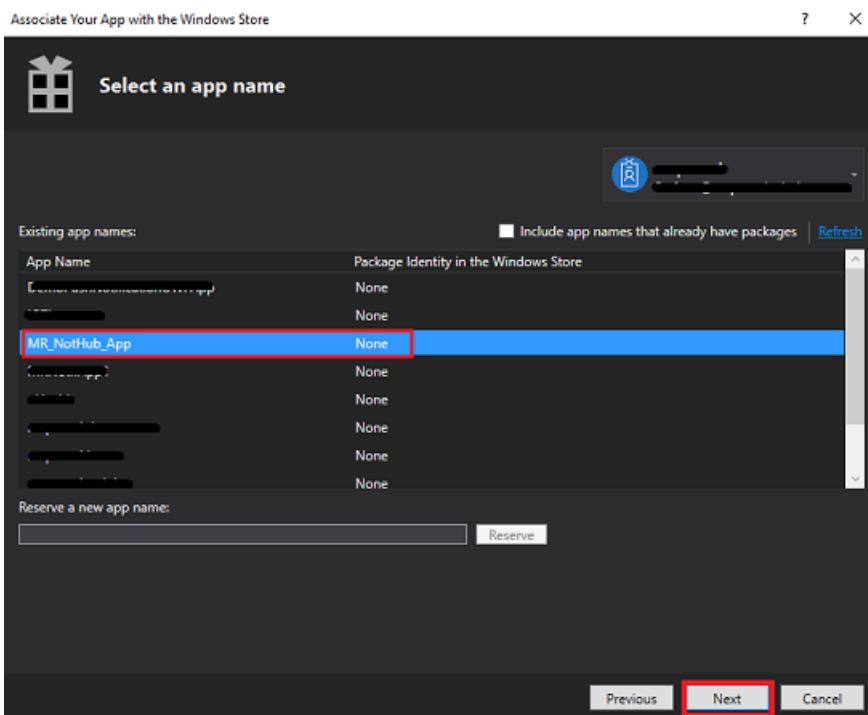
1. Open the solution.
2. Right click on the UWP app Project in the Solution Explorer panel, the go to **Store**, and **Associate App with the Store....**



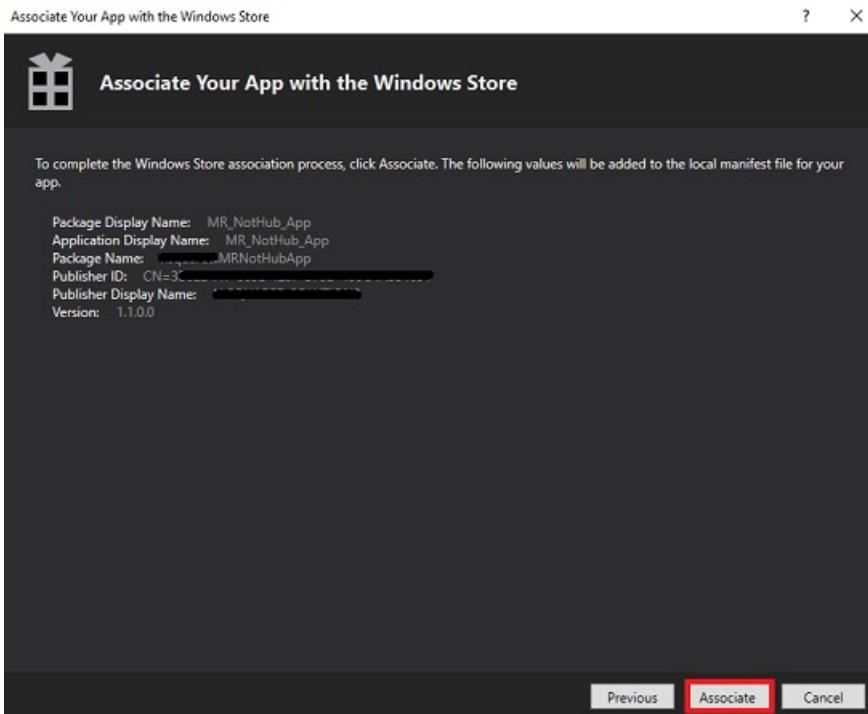
3. A new window will appear called **Associate Your App with the Windows Store**. Click **Next**.



4. It will load up all the Applications associated with the Account which you have logged in. If you are not logged in to your account, you can **Log In** on this page.
5. Find the **Store App name** that you created at the start of this tutorial and select it. Then click **Next**.



6. Click **Associate**.



7. Your App is now **Associated** with the Store App. This is necessary for enabling Notifications.

Chapter 20 - Deploy UnityMRNotifHub and UnityDesktopNotifHub applications

This Chapter may be easier with two people, as the result will include both apps running, one running on your computer Desktop, and the other within your immersive headset.

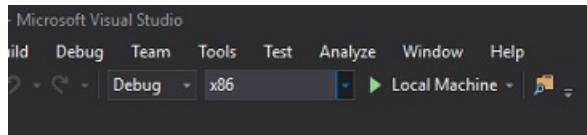
The immersive headset app is waiting to receive changes to the scene (position changes of the local GameObjects), and the Desktop app will be making changes to their local scene (position changes), which will be shared to the MR app. It makes sense to deploy the MR app first, followed by the Desktop app, so that the receiver can begin listening.

To deploy the **UnityMRNotifHub** app on your Local Machine:

1. Open the solution file of your **UnityMRNotifHub** app in **Visual Studio 2017**.

2. In the **Solution Platform**, select **x86, Local Machine**.

3. In the **Solution Configuration** select **Debug**.



4. Go to **Build menu** and click on **Deploy Solution** to sideload the application to your machine.

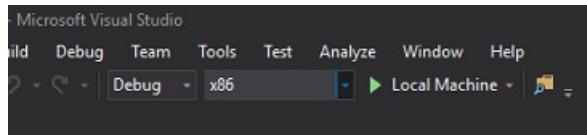
5. Your App should now appear in the list of installed apps, ready to be launched.

To deploy the **UnityDesktopNotifHub** app on Local Machine:

1. Open the solution file of your **UnityDesktopNotifHub** app in **Visual Studio 2017**.

2. In the **Solution Platform**, select **x86, Local Machine**.

3. In the **Solution Configuration** select **Debug**.



4. Go to **Build menu** and click on **Deploy Solution** to sideload the application to your machine.

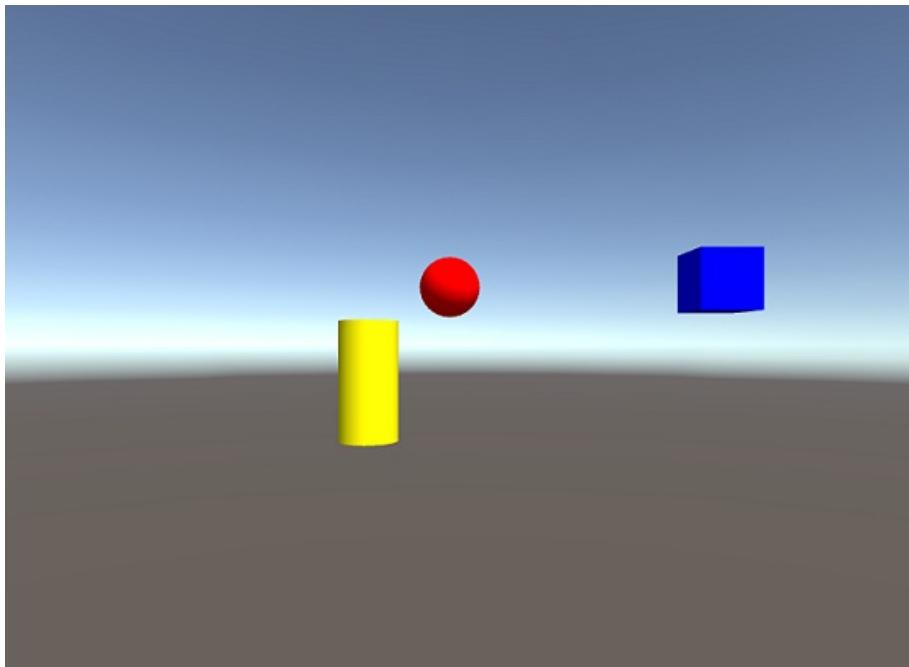
5. Your App should now appear in the list of installed apps, ready to be launched.

6. Launch the mixed reality application, followed by the Desktop application.

With both applications running, move an object in the desktop scene (using the Left Mouse Button). These positional changes will be made locally, serialized, and sent to the Function App service. The Function App service will then update the Table along with the Notification Hub. Having received an update, the Notification Hub will send the updated data directly to all the registered applications (in this case the immersive headset app), which will then deserialize the incoming data, and apply the new positional data to the local objects, moving them in scene.

Your finished your Azure Notification Hubs application

Congratulations, you built a mixed reality app that leverages the Azure Notification Hubs Service and allow communication between apps.



Bonus exercises

Exercise 1

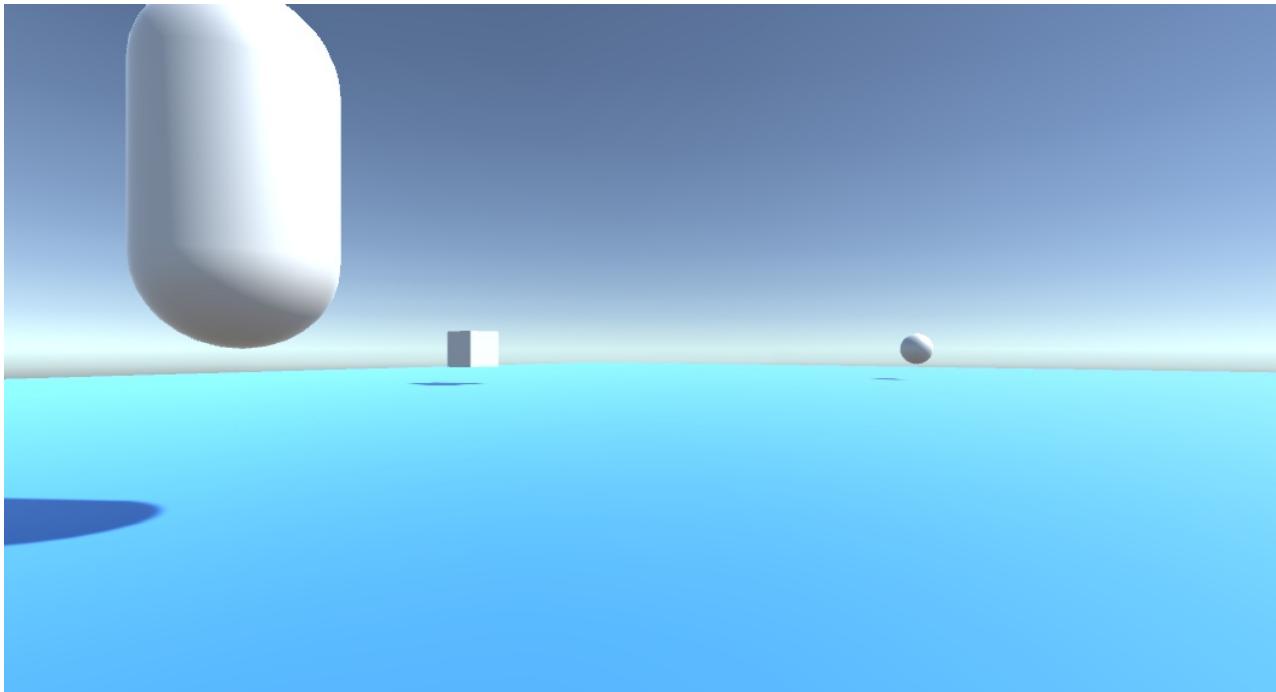
Can you work out how to change the color of the GameObjects and send that notification to other apps viewing the scene?

Exercise 2

Can you add movement of the GameObjects to your MR app and see the updated scene in your desktop app?

MR and Azure 309: Application insights

11/6/2018 • 30 minutes to read • [Edit Online](#)



In this course, you will learn how to add Application Insights capabilities to a mixed reality application, using the Azure Application Insights API to collect analytics regarding user behavior.

Application Insights is a Microsoft service, allowing developers to collect analytics from their applications and manage it from an easy-to-use portal. The analytics can be anything from performance to custom information you would like to collect. For more information, visit the [Application Insights page](#).

Having completed this course, you will have a mixed reality immersive headset application which will be able to do the following:

1. Allow the user to gaze and move around a scene.
2. Trigger the sending of analytics to the *Application Insights Service*, through the use of Gaze and Proximity to in-scene objects.
3. The app will also call upon the Service, fetching information about which object has been approached the most by the user, within the last 24 hours. That object will change its color to green.

This course will teach you how to get the results from the Application Insights Service, into a Unity-based sample application. It will be up to you to apply these concepts to a custom application you might be building.

Device support

COURSE	HOLOLENS	IMMERSIVE HEADSETS
MR and Azure 309: Application insights	✓ <input type="checkbox"/>	✓ <input type="checkbox"/>

NOTE

While this course primarily focuses on Windows Mixed Reality immersive (VR) headsets, you can also apply what you learn in this course to Microsoft HoloLens. As you follow along with the course, you will see notes on any changes you might need to employ to support HoloLens. When using HoloLens, you may notice some echo during voice capture.

Prerequisites

NOTE

This tutorial is designed for developers who have basic experience with Unity and C#. Please also be aware that the prerequisites and written instructions within this document represent what has been tested and verified at the time of writing (July 2018). You are free to use the latest software, as listed within the [Install the tools](#) article, though it should not be assumed that the information in this course will perfectly match what you will find in newer software than what is listed below.

We recommend the following hardware and software for this course:

- A development PC, [compatible with Windows Mixed Reality](#) for immersive (VR) headset development
- [Windows 10 Fall Creators Update \(or later\)](#) with Developer mode enabled
- [The latest Windows 10 SDK](#)
- [Unity 2017.4](#)
- [Visual Studio 2017](#)
- A [Windows Mixed Reality immersive \(VR\) headset](#) or [Microsoft HoloLens](#) with Developer mode enabled
- A set of headphones with a built-in microphone (if the headset does not have a built-in mic and speakers)
- Internet access for Azure setup and Application Insights data retrieval

Before you start

To avoid encountering issues building this project, it is strongly suggested that you create the project mentioned in this tutorial in a root or near-root folder (long folder paths can cause issues at build-time).

WARNING

Be aware, data going to *Application Insights* takes time, so be patient. If you want to check if the Service has received your data, check out [Chapter 14](#), which will show you how to navigate the portal.

Chapter 1 - The Azure Portal

To use *Application Insights*, you will need to create and configure an *Application Insights Service* in the Azure portal.

1. Log in to the [Azure Portal](#).

NOTE

If you do not already have an Azure account, you will need to create one. If you are following this tutorial in a classroom or lab situation, ask your instructor or one of the proctors for help setting up your new account.

2. Once you are logged in, click on **New** in the top left corner, and search for *Application Insights*, and click **Enter**.

NOTE

The word **New** may have been replaced with **Create a resource**, in newer portals.

A screenshot of the Microsoft Azure Marketplace interface. The search bar at the top contains the text 'Insight'. Below the search bar, a table lists a single result: 'Application Insights' by Microsoft, categorized under 'Web + Mobile'. The left sidebar includes links for 'Create a resource', 'All services', 'Favorites', 'Dashboard', 'All resources', 'Resource groups', and 'App Services'.

3. The new page to the right will provide a description of the *Azure Application Insights Service*. At the bottom left of this page, select the **Create** button, to create an association with this Service.

A screenshot of the Azure Application Insights service details page. The top section contains a brief description of what Application Insights does, followed by a bulleted list of features: Availability and performance monitoring, Users and usage insights, Diagnostic logs and crash analytics, Seamless integration with Microsoft Azure and Visual Studio, and Supports ASP.NET websites (Azure or on-premises), Windows Phone, and Windows Store apps. Below this is a social sharing section with icons for Twitter, Facebook, LinkedIn, YouTube, Google+, and Email. The main content area shows a dashboard with various metrics and charts, including Average response time, Request rate, and Failed requests. The bottom of the page shows the publisher information 'Microsoft' and a prominent red 'Create' button.

4. Once you have clicked on **Create**:

- Insert your desired **Name** for this Service instance.
- As **Application Type**, select **General**.
- Select an appropriate **Subscription**.
- Choose a **Resource Group** or create a new one. A resource group provides a way to monitor, control access, provision and manage billing for a collection of Azure assets. It is recommended to keep all the Azure Services associated with a single project (e.g. such as these courses) under a common resource group.

If you wish to read more about Azure Resource Groups, please [visit the resource group article](#).

e. Select a **Location**.

f. You will also need to confirm that you have understood the Terms and Conditions applied to this Service.

g. Select **Create**.

The screenshot shows the 'Create' step of the Application Insights wizard. The form fields are as follows:

- Name: MyNewInsight
- * Application Type: General
- * Subscription: Mai's Microsoft Azure Internal Consumption
- * Resource Group: Azure_for_Unity
- * Location: Southeast Asia

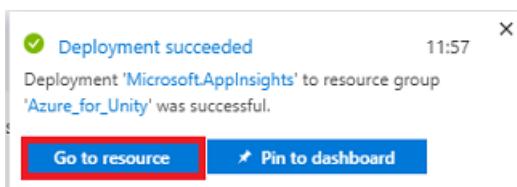
At the bottom, there is a 'Create' button highlighted in blue, and an 'Automation options' link.

5. Once you have clicked on **Create**, you will have to wait for the Service to be created, this might take a minute.

6. A notification will appear in the portal once the Service instance is created.



7. Click on the notifications to explore your new Service instance.



8. Click the **Go to resource** button in the notification to explore your new Service instance. You will be taken to your new *Application Insights Service* instance.

Home > MyNewInsight

MyNewInsight

Application Insights - Last 24 hours (30 minute granularity) - General

Search (Ctrl+ /)

Overview

Activity log

Access control (IAM)

Tags

Diagnose and solve problems

INVESTIGATE

Application map

Smart Detection

Metrics Explorer

Metrics (preview)

Search

Availability

Failures (preview)

Performance (preview)

Servers

Browser

Workbooks (preview)

USAGE (PREVIEW)

Users

Sessions

Search Metrics Explorer Analytics Time range Refresh More

Essentials

0 Alerts Live Stream Click to configure Smart Detection 0 Detections (7d) Availability App map

Health

Overview timeline MYNEWINSIGHT

Learn how to collect server response time data.

Learn how to collect browser page load data.

Learn how to collect server request data.

Learn how to collect failed request data.

Total of Server Requests by Request Performance MYNEWINSIGHT

REQUEST PERFORMANCE	TOTAL	% TOTAL
No results		

NOTE

Keep this web page open and easy to access, you will come back here often to see the data collected.

IMPORTANT

To implement Application Insights, you will need to use three (3) specific values: **Instrumentation Key**, **Application ID**, and **API Key**. Below you will see how to retrieve these values from your Service. Make sure to note these values on a blank *Notepad* page, because you will use them soon in your code.

9. To find the **Instrumentation Key**, you will need to scroll down the list of Service functions, and click on **Properties**, the tab displayed will reveal the **Service Key**.

NAME
StefanoApplicationInsight

TYPE
General

LOCATION
Southeast Asia

INSTRUMENTATION KEY
639a06...

RESOURCE GROUP NAME
Azure_for_Unity

SUBSCRIPTION NAME
Mai's Microsoft Azure Internal Consum...

SUBSCRIPTION ID
e7ea...

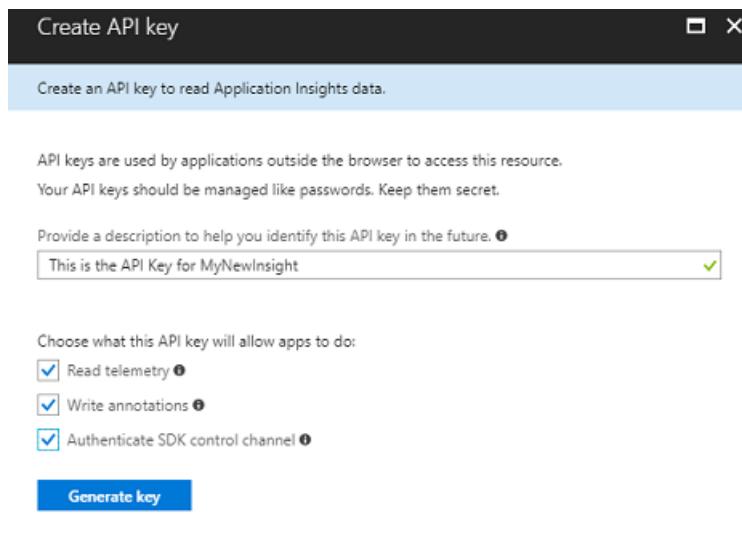
10. A little below **Properties**, you will find **API Access**, which you need to click. The panel to the right will provide the **Application ID** of your app.

MyNewInsight - API Access

Application ID: b3e0...

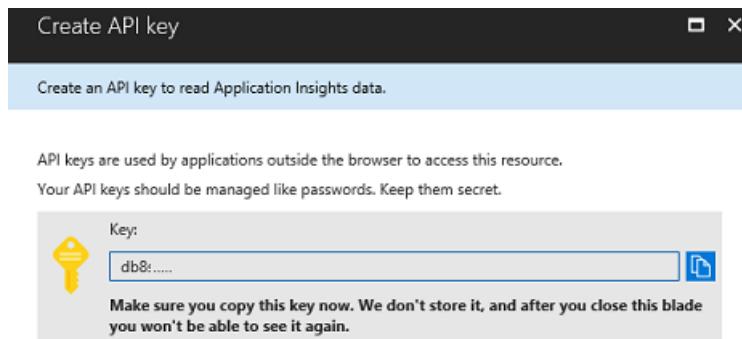
API KEY DESCRIPTION	LAST USED	CREATED ON	PERMISSIONS
Loading...			

11. With the **Application ID** panel still open, click **Create API Key**, which will open the *Create API key* panel.



12. Within the now open *Create API key* panel, type a description, and **tick the three boxes**.

13. Click **Generate Key**. Your **API Key** will be created and displayed.



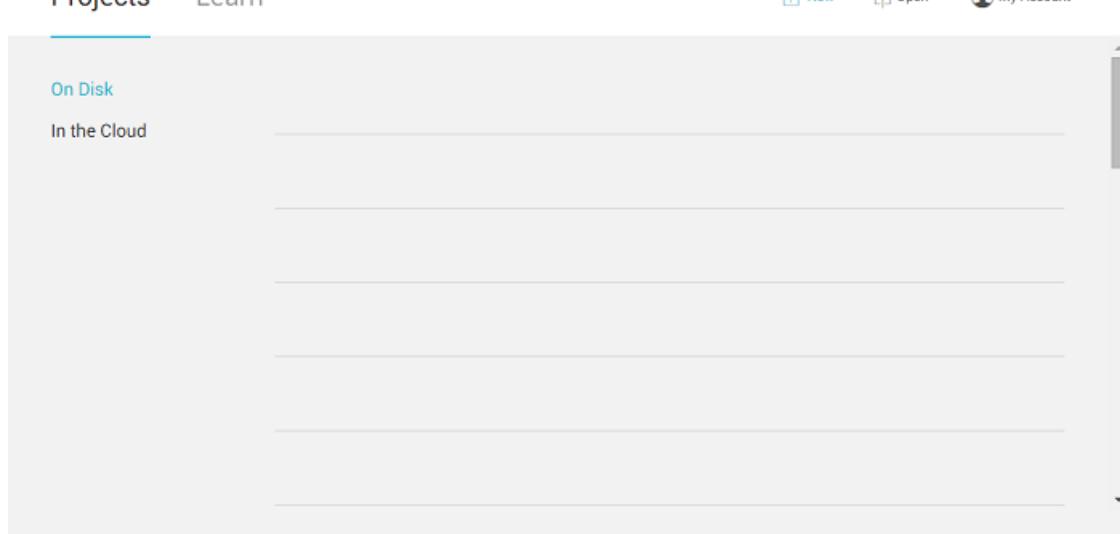
WARNING

This is the only time your **Service Key** will be displayed, so ensure you make a copy of it now.

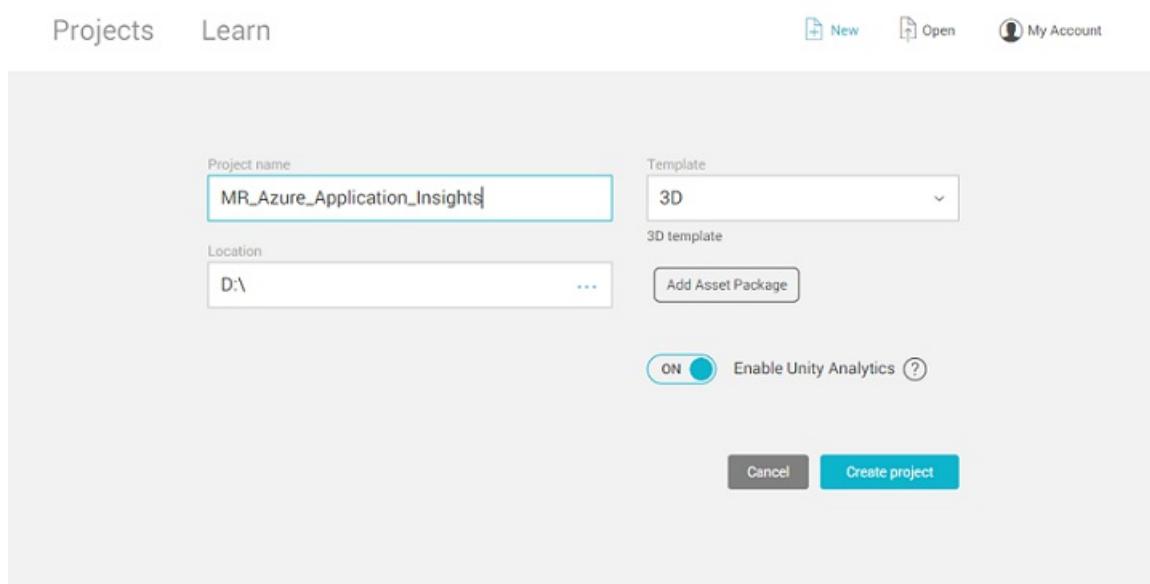
Chapter 2 - Set up the Unity project

The following is a typical set up for developing with the mixed reality, and as such, is a good template for other projects.

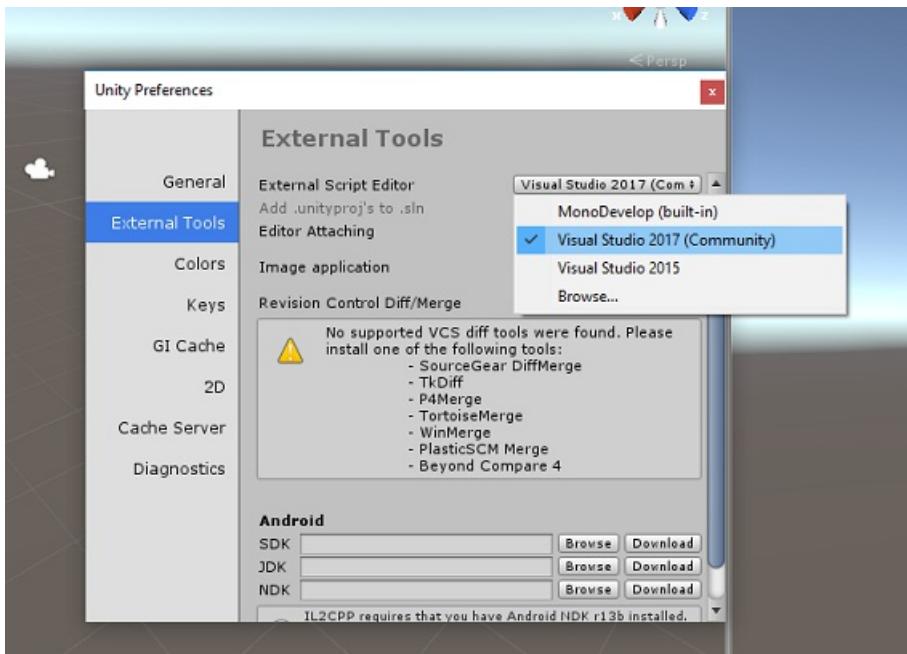
1. Open *Unity* and click **New**.



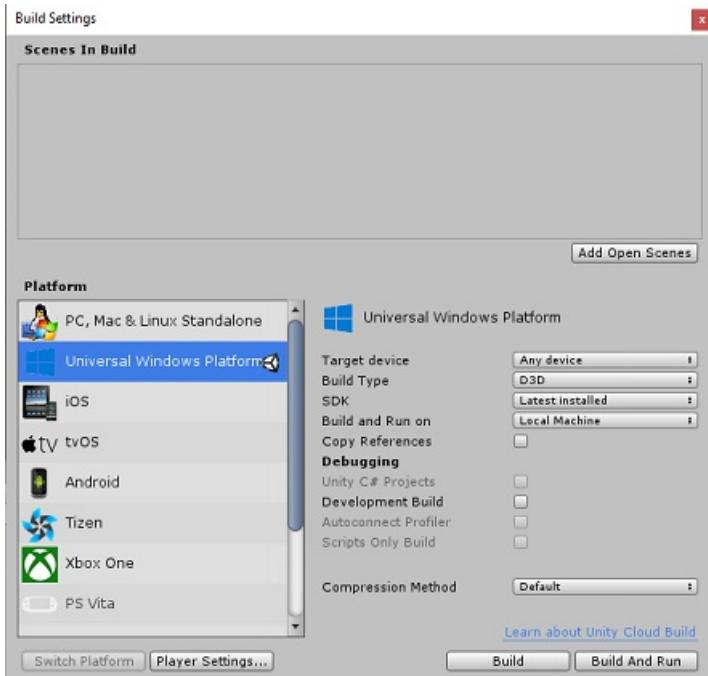
2. You will now need to provide a Unity Project name, insert **MR_Azure_Application_Insights**. Make sure the *Template* is set to **3D**. Set the *Location* to somewhere appropriate for you (remember, closer to root directories is better). Then, click **Create project**.



3. With Unity open, it is worth checking the default **Script Editor** is set to **Visual Studio**. Go to **Edit > Preferences** and then from the new window, navigate to **External Tools**. Change **External Script Editor** to **Visual Studio 2017**. Close the **Preferences** window.



4. Next, go to **File > Build Settings** and switch the platform to **Universal Windows Platform**, by clicking on the **Switch Platform** button.



5. Go to **File > Build Settings** and make sure that:

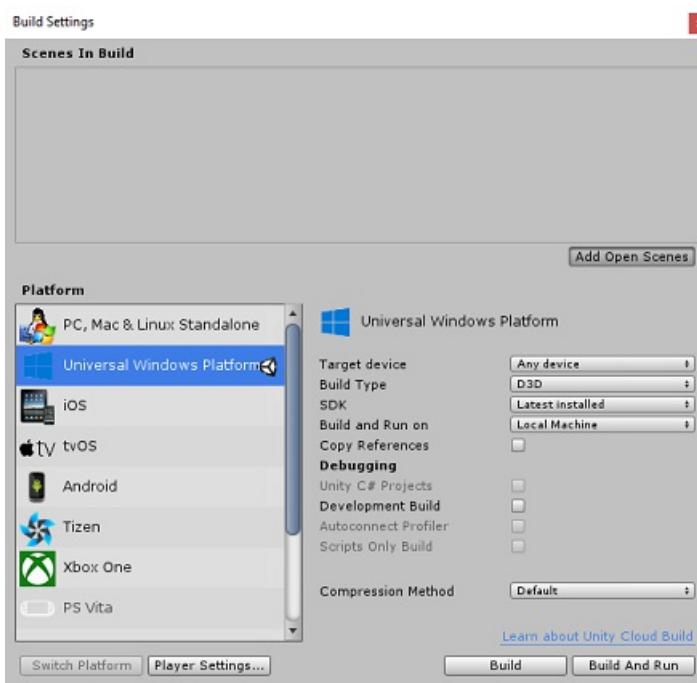
- a. **Target Device** is set to **Any device**

For the Microsoft HoloLens, set **Target Device** to *HoloLens*.

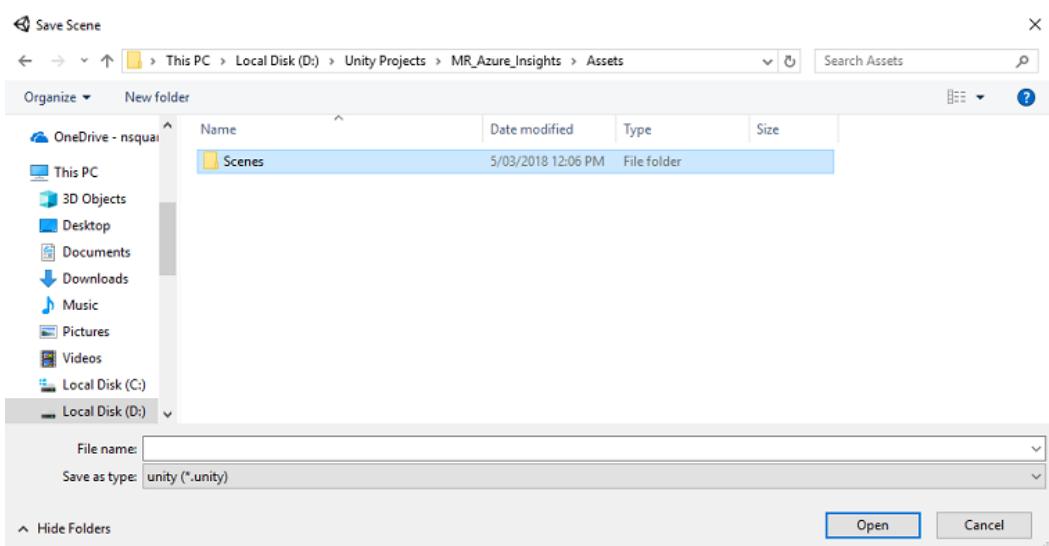
- b. **Build Type** is set to **D3D**
- c. **SDK** is set to **Latest installed**
- d. **Build and Run** is set to **Local Machine**

- e. Save the scene and add it to the build.

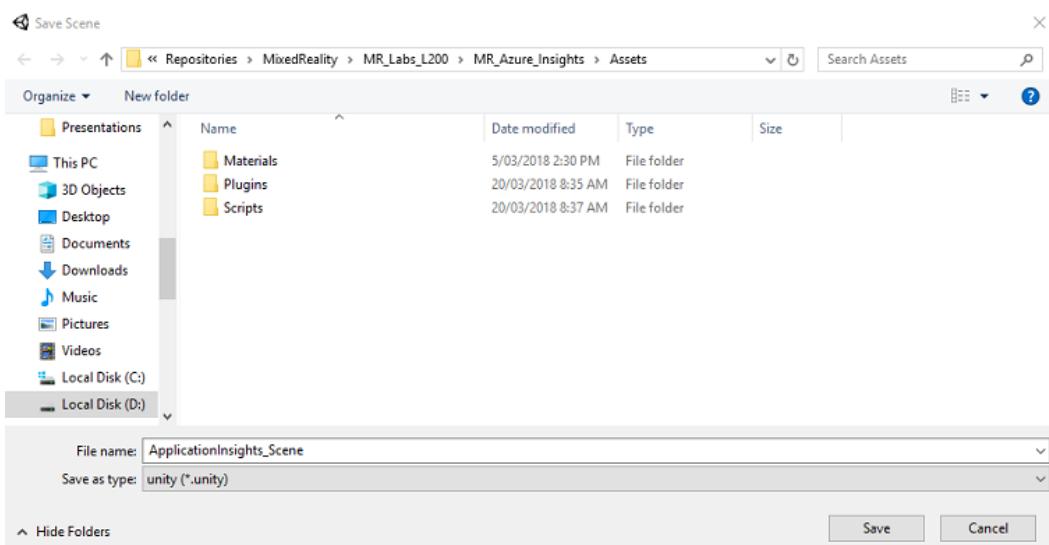
- a. Do this by selecting **Add Open Scenes**. A save window will appear.



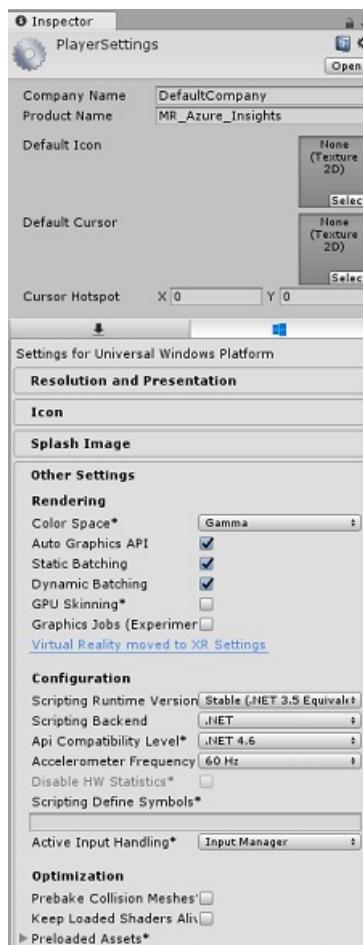
- b. Create a new folder for this, and any future scene, then click the **New folder** button, to create a new folder, name it **Scenes**.



- c. Open your newly created **Scenes** folder, and then in the *File name:* text field, type **ApplicationInsightsScene**, then click **Save**.

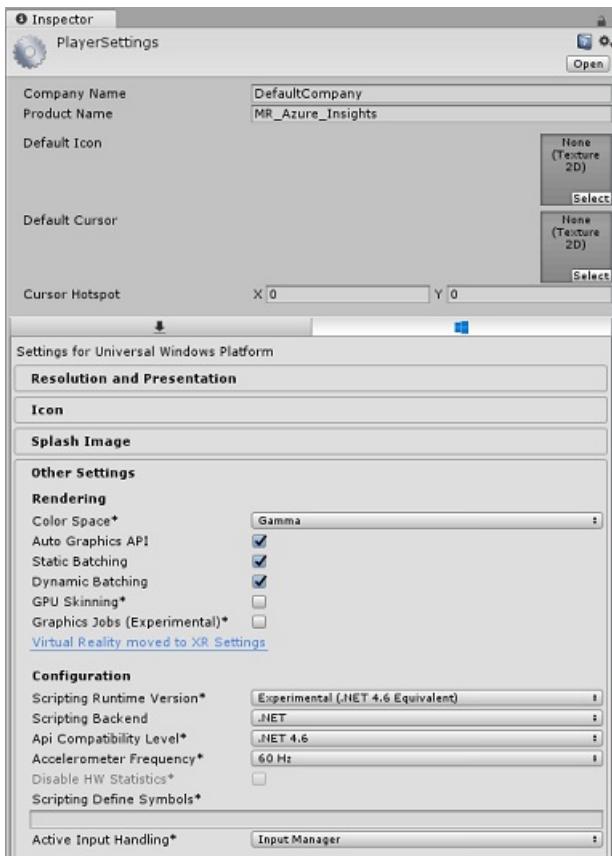


6. The remaining settings, in **Build Settings**, should be left as default for now.
7. In the **Build Settings** window, click on the **Player Settings** button, this will open the related panel in the space where the **Inspector** is located.



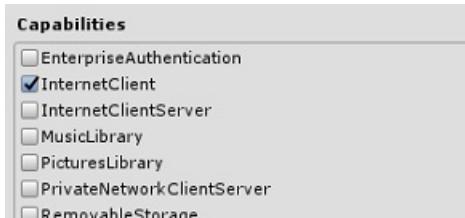
8. In this panel, a few settings need to be verified:

- a. In the **Other Settings** tab:
 - a. **Scripting Runtime Version** should be **Experimental (.NET 4.6 Equivalent)**, which will trigger a need to restart the Editor.
 - b. **Scripting Backend** should be **.NET**
 - c. **API Compatibility Level** should be **.NET 4.6**

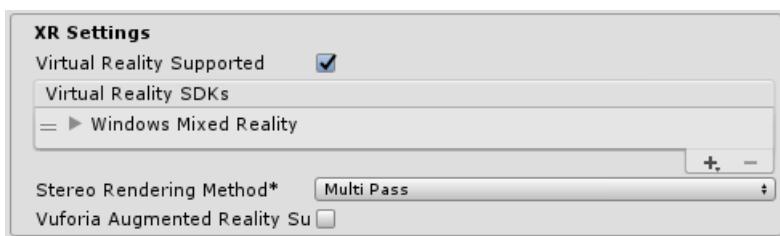


b. Within the **Publishing Settings** tab, under **Capabilities**, check:

- **InternetClient**



c. Further down the panel, in **XR Settings** (found below **Publishing Settings**), tick **Virtual Reality Supported**, make sure the **Windows Mixed Reality SDK** is added.



9. Back in **Build Settings, Unity C# Projects** is no longer greyed out; tick the checkbox next to this.

10. Close the Build Settings window.

11. Save your Scene and Project (**FILE > SAVE SCENE / FILE > SAVE PROJECT**).

Chapter 3 - Import the Unity package

IMPORTANT

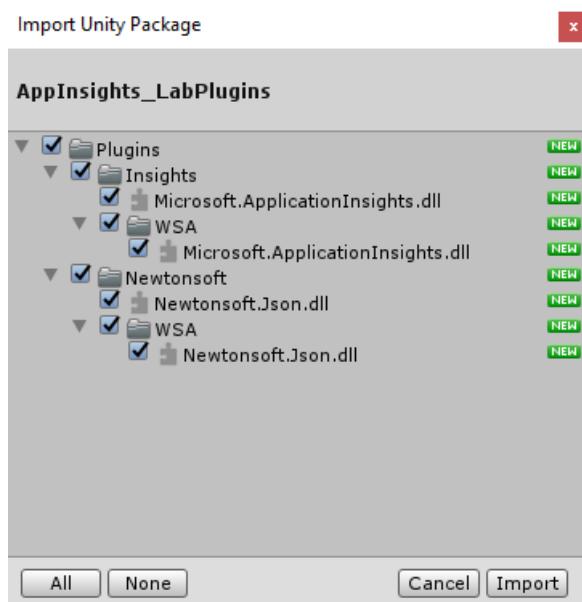
If you wish to skip the *Unity Set up* components of this course, and continue straight into code, feel free to download this [Azure-MR-309.unitypackage](#), import it into your project as a **Custom Package**. This will also contain the DLLs from the next Chapter. After import, continue from [Chapter 6](#).

IMPORTANT

To use Application Insights within Unity, you need to import the DLL for it, along with the Newtonsoft DLL. There is currently a known issue in Unity which requires plugins to be reconfigured after import. These steps (4 - 7 in this section) will no longer be required after the bug has been resolved.

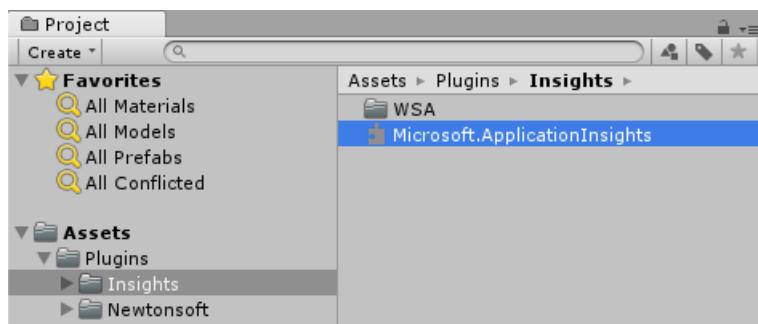
To import Application Insights into your own project, make sure you have [downloaded the '.unitypackage'](#), [containing the plugins](#). Then, do the following:

1. Add the **.unitypackage** to Unity by using the **Assets > Import Package > Custom Package** menu option.
2. In the **Import Unity Package** box that pops up, ensure everything under (and including) **Plugins** is selected.

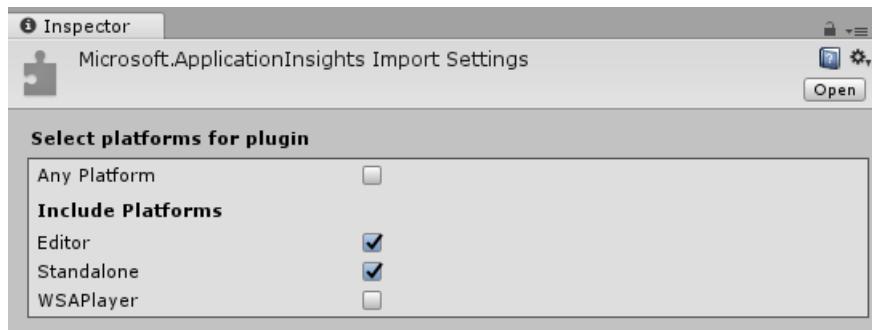


3. Click the **Import** button, to add the items to your project.
4. Go to the **Insights** folder under **Plugins** in the Project view and select the following plugins *only*:

- Microsoft.ApplicationInsights



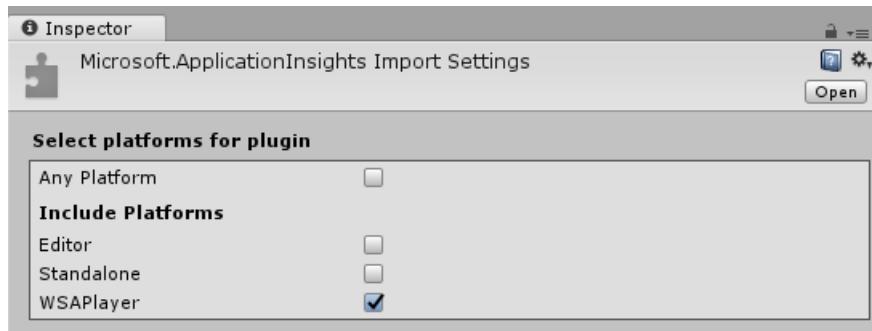
5. With this *plugin* selected, ensure that **Any Platform** is **unchecked**, then ensure that **WSAPlayer** is also **unchecked**, then click **Apply**. Doing this is just to confirm that the files are configured correctly.



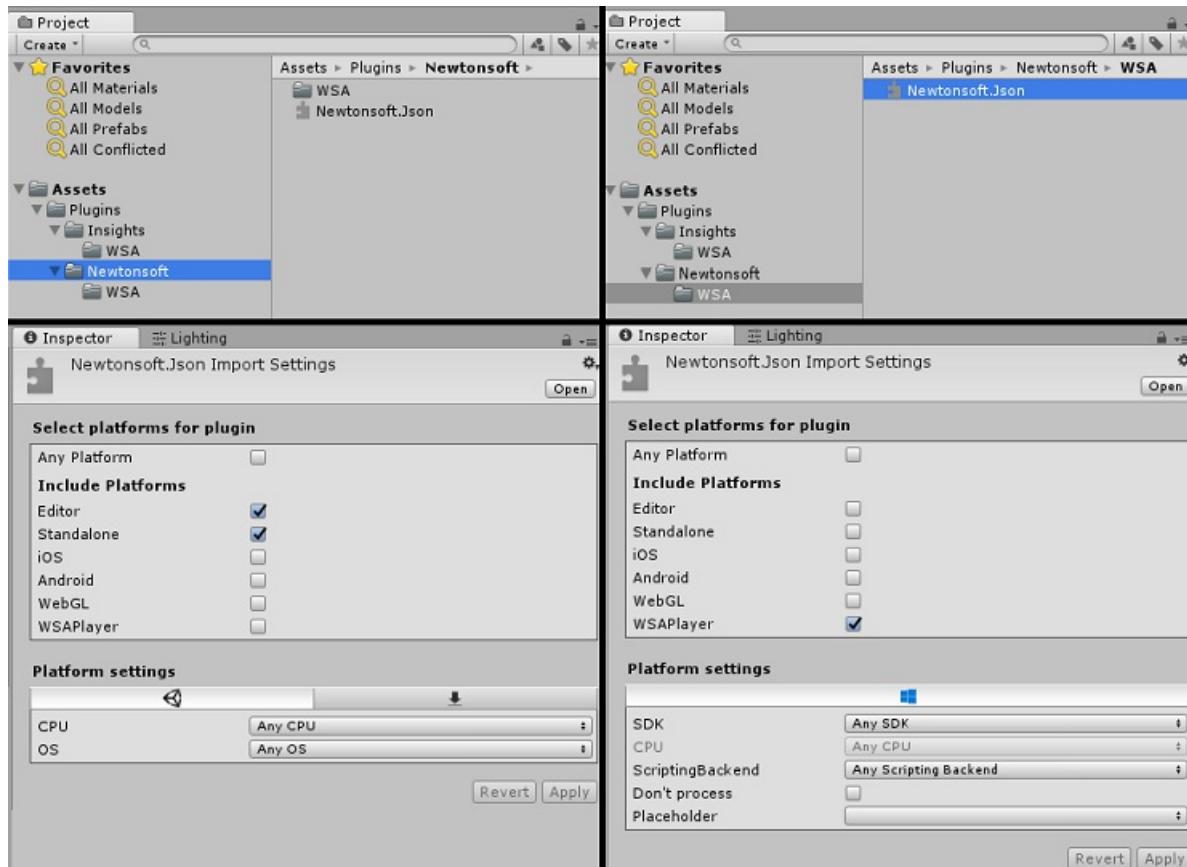
NOTE

Marking the plugins like this, configures them to only be used in the Unity Editor. There are a different set of DLLs in the WSA folder which will be used after the project is exported from Unity.

6. Next, you need to open the **WSA** folder, within the **Insights** folder. You will see a copy of the same file you just configured. Select this file, and then in the inspector, ensure that **Any Platform** is **unchecked**, then ensure that **only WSAPlayer** is **checked**. Click **Apply**.



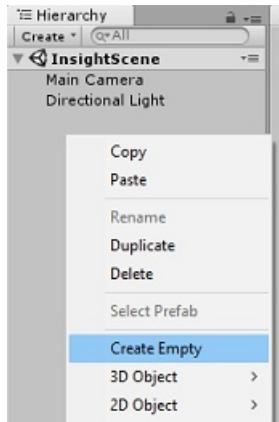
7. You will now need to follow **steps 4-6**, but for the **Newtonsoft** plugins instead. See the below screenshot for what the outcome should look like.



Chapter 4 - Set up the camera and user controls

In this Chapter you will set up the camera and the controls to allow the user to see and move in the scene.

1. Right-click in an empty area in the Hierarchy Panel, then on **Create > Empty**.



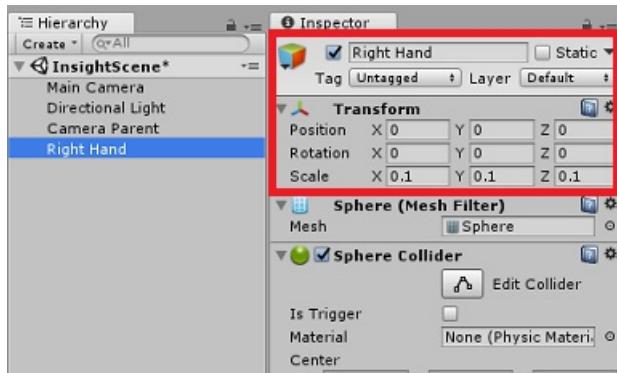
2. Rename the new empty GameObject to **Camera Parent**.



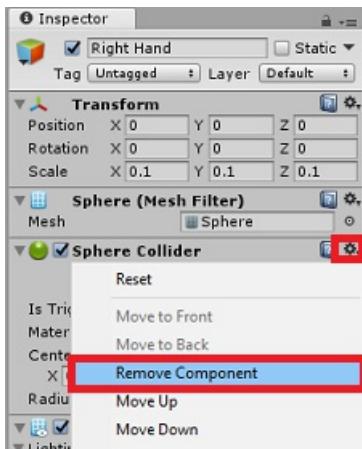
3. Right-click in an empty area in the Hierarchy Panel, then on **3D Object**, then on **Sphere**.

4. Rename the Sphere to **Right Hand**.

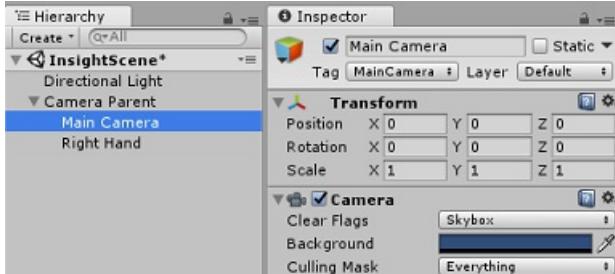
5. Set the **Transform Scale** of the Right Hand to **0.1, 0.1, 0.1**



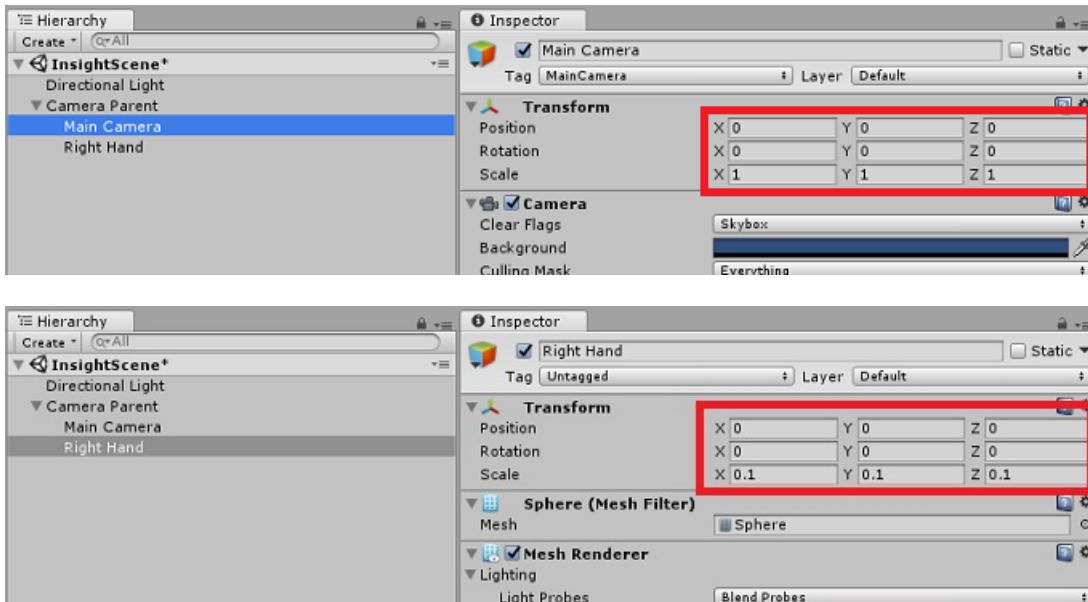
6. Remove the **Sphere Collider** component from the Right Hand by clicking on the **Gear** in the **Sphere Collider** component, and then **Remove Component**.



7. In the Hierarchy Panel drag the **Main Camera** and the **Right Hand** objects onto the **Camera Parent** object.



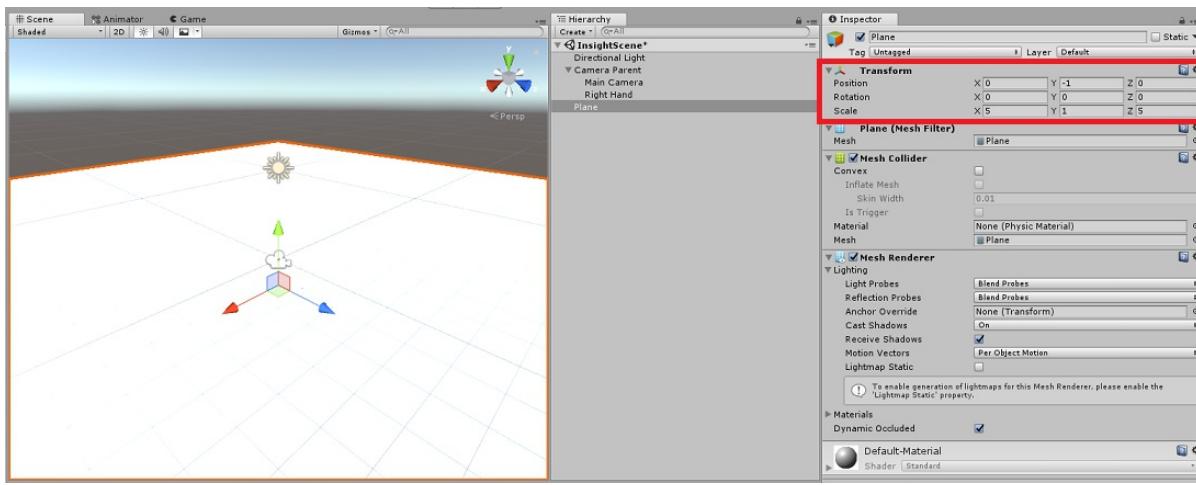
8. Set the **Transform Position** of both the **Main Camera** and the **Right Hand** object to **0, 0, 0**.



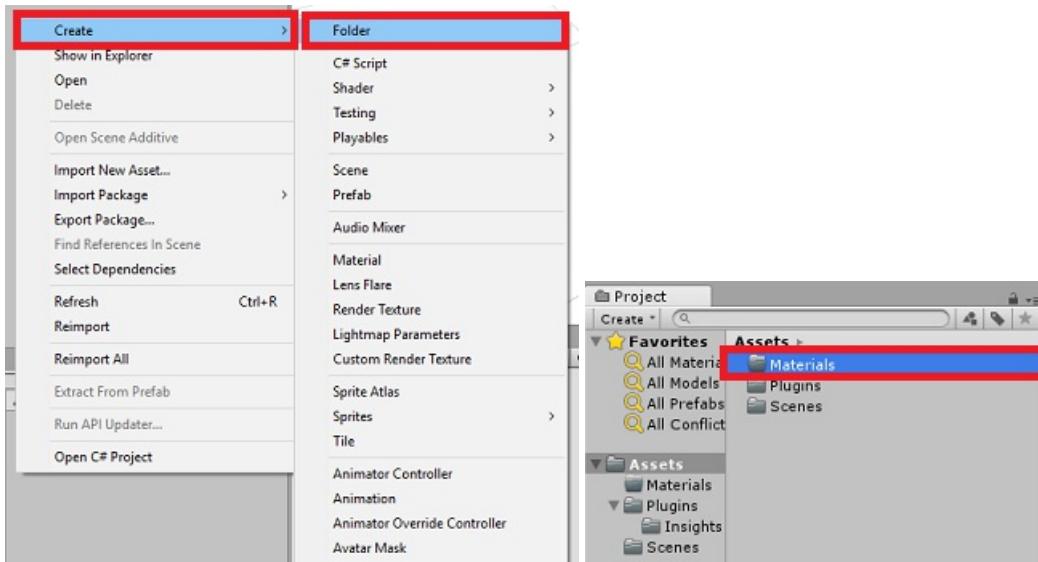
Chapter 5 - Set up the objects in the Unity scene

You will now create some basic shapes for your scene, with which the user can interact.

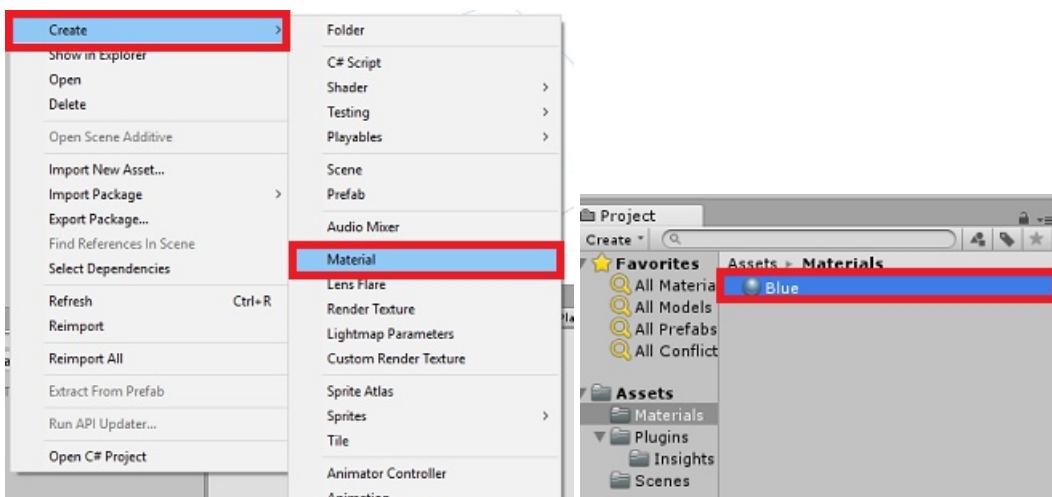
1. Right-click in an empty area in the *Hierarchy Panel*, then on **3D Object**, then select **Plane**.
2. Set the Plane **Transform Position** to **0, -1, 0**.
3. Set the Plane **Transform Scale** to **5, 1, 5**.



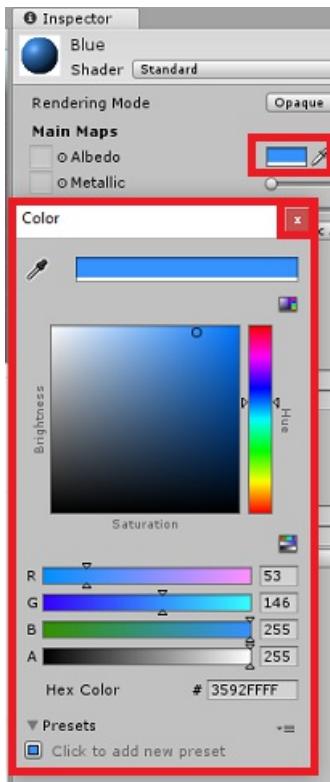
4. Create a basic material to use with your **Plane** object, so that the other shapes are easier to see. Navigate to your *Project Panel*, right-click, then **Create**, followed by **Folder**, to create a new folder. Name it **Materials**.



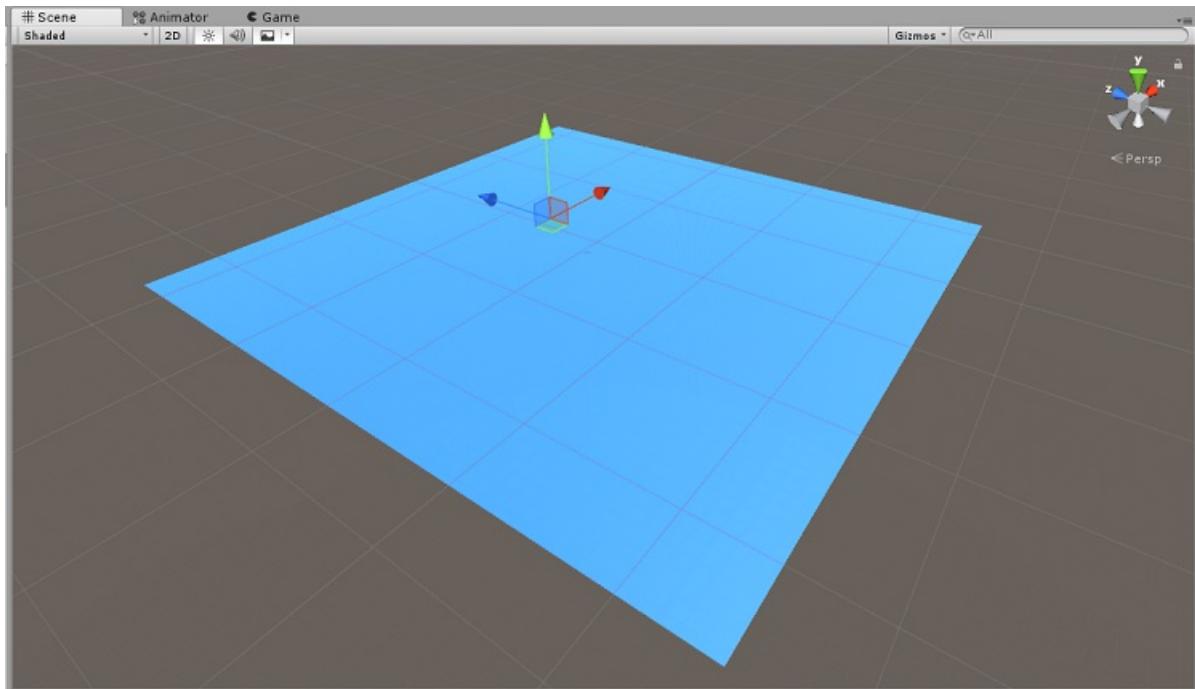
5. Open the **Materials** folder, then right-click, click **Create**, then **Material**, to create a new material. Name it **Blue**.



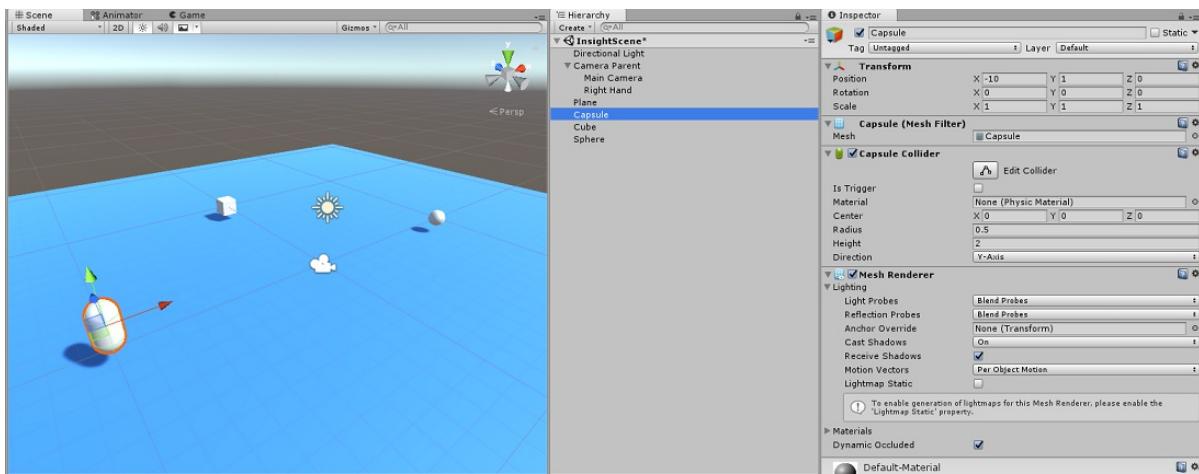
6. With the new **Blue** material selected, look at the *Inspector*, and click the rectangular window alongside **Albedo**. Select a blue color (the one picture below is **Hex Color: #3592FFFF**). Click the close button once you have chosen.



7. Drag your new material from the **Materials** folder, onto your newly created **Plane**, within your scene (or drop it on the **Plane** object within the *Hierarchy*).



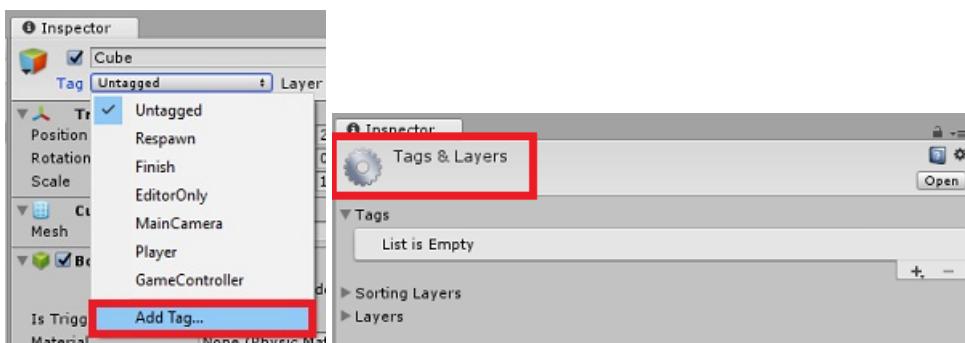
8. Right-click in an empty area in the *Hierarchy Panel*, then on **3D Object, Capsule**.
 - With the **Capsule** selected, change its **Transform Position** to: **-10, 1, 0**.
9. Right-click in an empty area in the *Hierarchy Panel*, then on **3D Object, Cube**.
 - With the **Cube** selected, change its **Transform Position** to: **0, 0, 10**.
10. Right-click in an empty area in the *Hierarchy Panel*, then on **3D Object, Sphere**.
 - With the **Sphere** selected, change its **Transform Position** to: **10, 0, 0**.



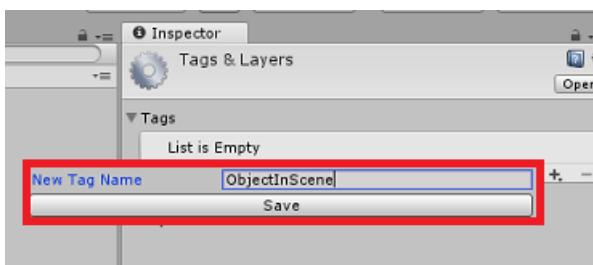
NOTE

These *Position* values are *suggestions*. You are free to set the positions of the objects to whatever you would like, though it is easier for the user of the application if the objects distances are not too far from the camera.

- When your application is running, it needs to be able to identify the objects within the scene, to achieve this, they need to be tagged. Select one of the objects, and in the *Inspector* panel, click **Add Tag...**, which will swap the *Inspector* with the **Tags & Layers** panel.



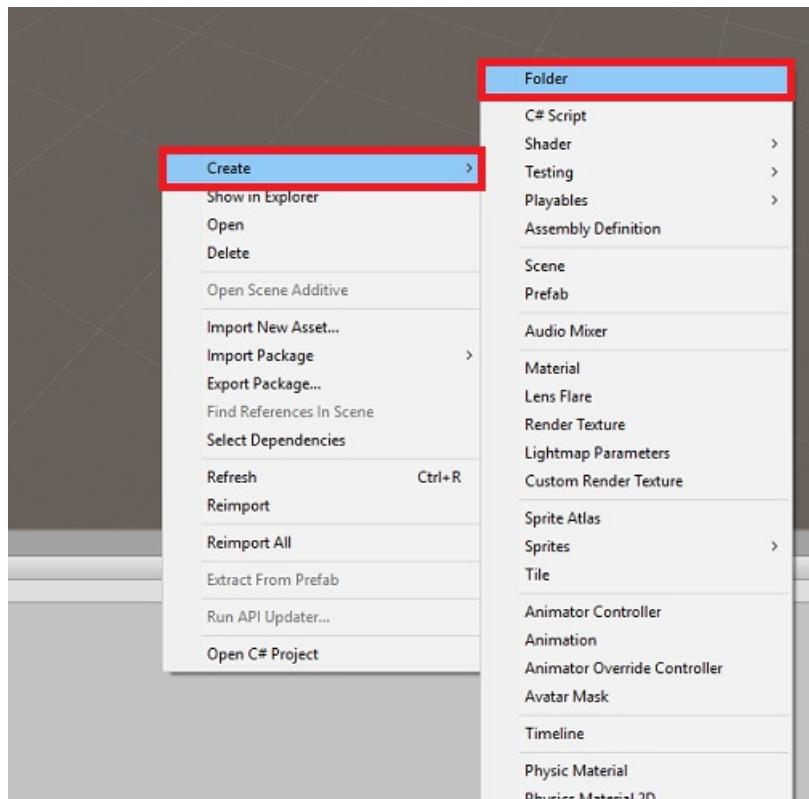
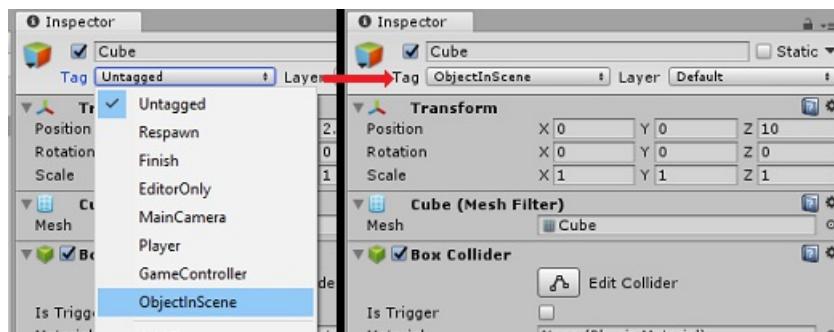
- Click the **+** (plus) symbol, then type the tag name as **ObjectInScene**.



WARNING

If you use a different name for your tag, you will need to ensure this change is also made the *DataFromAnalytics*, *ObjectTrigger*, and *Gaze*, scripts later, so that your objects are found, and detected, within your scene.

- With the tag created, you now need to apply it to all three of your objects. From the *Hierarchy*, hold the **Shift** key, then click the **Capsule**, **Cube**, and **Sphere**, objects, then in the *Inspector*, click the dropdown menu alongside **Tag**, then click the *ObjectInScene* tag you created.



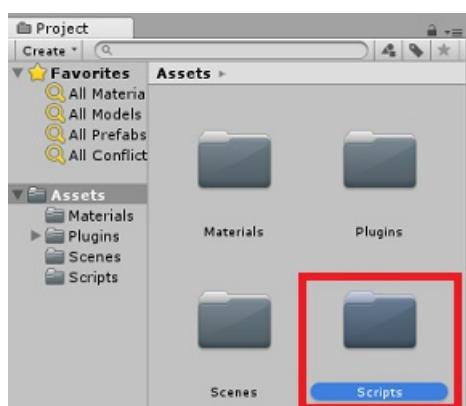
Chapter 6 - Create the ApplicationInsightsTracker class

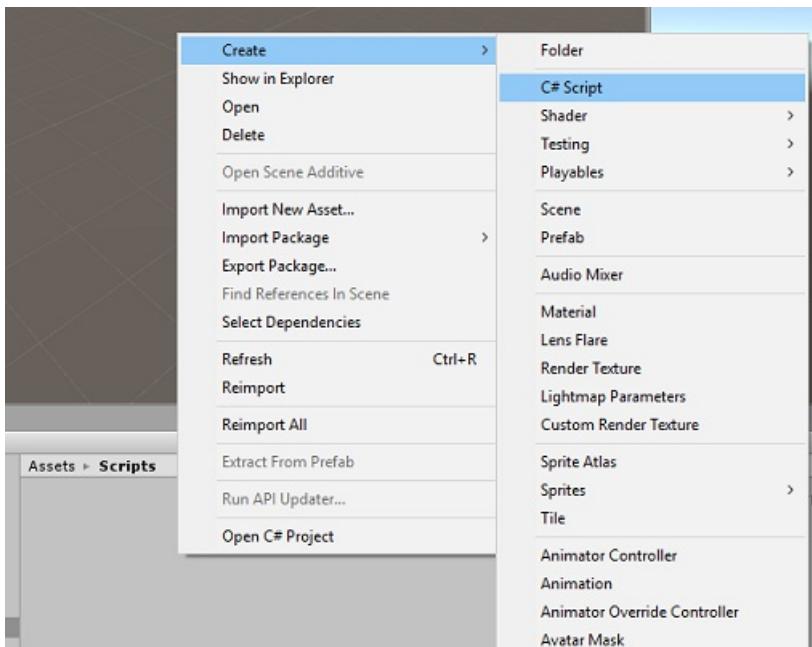
The first script you need to create is **ApplicationInsightsTracker**, which is responsible for:

1. Creating events based on user interactions to submit to Azure Application Insights.
2. Creating appropriate Event names, depending on user interaction.
3. Submitting events to the Application Insights Service instance.

To create this class:

1. Right-click in the *Project Panel*, then **Create > Folder**. Name the folder **Scripts**.





2. With the **Scripts** folder created, double-click it, to open. Then, within that folder, right-click, **Create > C# Script**. Name the script **ApplicationInsightsTracker**.
3. Double-click on the new **ApplicationInsightsTracker** script to open it with **Visual Studio**.
4. Update namespaces at the top of the script to be as below:

```
using Microsoft.ApplicationInsights;
using Microsoft.ApplicationInsights.DataContracts;
using Microsoft.ApplicationInsights.Extensibility;
using UnityEngine;
```

5. Inside the class insert the following variables:

```

/// <summary>
/// Allows this class to behavior like a singleton
/// </summary>
public static ApplicationInsightsTracker Instance;

/// <summary>
/// Insert your Instrumentation Key here
/// </summary>
internal string instrumentationKey = "Insert Instrumentation Key here";

/// <summary>
/// Insert your Application Id here
/// </summary>
internal string applicationId = "Insert Application Id here";

/// <summary>
/// Insert your API Key here
/// </summary>
internal string API_Key = "Insert API Key here";

/// <summary>
/// Represent the Analytic Custom Event object
/// </summary>
private TelemetryClient telemetryClient;

/// <summary>
/// Represent the Analytic object able to host gaze duration
/// </summary>
private MetricTelemetry metric;

```

NOTE

Set the **instrumentationKey**, **applicationId** and **API_Key** values appropriately, using the *Service Keys* from the Azure Portal as mentioned in [Chapter 1](#), step 9 onwards.

6. Then add the **Start()** and **Awake()** methods, which will be called when the class initializes:

```

/// <summary>
/// Sets this class instance as a singleton
/// </summary>
void Awake()
{
    Instance = this;
}

/// <summary>
/// Use this for initialization
/// </summary>
void Start()
{
    // Instantiate telemetry and metric
    telemetryClient = new TelemetryClient();

    metric = new MetricTelemetry();

    // Assign the Instrumentation Key to the Event and Metric objects
    TelemetryConfiguration.Active.InstrumentationKey = instrumentationKey;

    telemetryClient.InstrumentationKey = instrumentationKey;
}

```

7. Add the methods responsible for sending the events and metrics registered by your application:

```

/// <summary>
/// Submit the Event to Azure Analytics using the event trigger object
/// </summary>
public void RecordProximityEvent(string objectName)
{
    telemetryClient.TrackEvent(CreateEventName(objectName));
}

/// <summary>
/// Uses the name of the object involved in the event to create
/// and return an Event Name convention
/// </summary>
public string CreateEventName(string name)
{
    string eventName = $"User near {name}";
    return eventName;
}

/// <summary>
/// Submit a Metric to Azure Analytics using the metric gazed object
/// and the time count of the gaze
/// </summary>
public void RecordGazeMetrics(string objectName, int time)
{
    // Output Console information about gaze.
    Debug.Log($"Finished gazing at {objectName}, which went for <b>{time}</b> second{(time != 1 ? "s" : "")}");

    metric.Name = $"Gazed {objectName}";

    metric.Value = time;

    telemetryClient.TrackMetric(metric);
}

```

8. Be sure to save your changes in *Visual Studio* before returning to *Unity*.

Chapter 7 - Create the Gaze script

The next script to create is the **Gaze** script. This script is responsible for creating a *Raycast* that will be projected forward from the *Main Camera*, to detect which object the user is looking at. In this case, the *Raycast* will need to identify if the user is looking at an object with the **ObjectInScene** tag, and then count how long the user *gazes* at that object.

1. Double-click on the **Scripts** folder, to open it.
2. Right-click inside the **Scripts** folder, click **Create > C# Script**. Name the script **Gaze**.
3. Double-click on the script to open it with Visual Studio.
4. Replace the existing code with the following:

```

using UnityEngine;

public class Gaze : MonoBehaviour
{
    /// <summary>
    /// Provides Singleton-like behavior to this class.
    /// </summary>
    public static Gaze Instance;

    /// <summary>
    /// Provides a reference to the object the user is currently looking at.
    /// </summary>
    public GameObject FocusedGameObject { get; private set; }

    /// <summary>
    /// Provides whether an object has been successfully hit by the raycast.
    /// </summary>
    public bool Hit { get; private set; }

    /// <summary>
    /// Provides a reference to compare whether the user is still looking at
    /// the same object (and has not looked away).
    /// </summary>
    private GameObject _oldFocusedObject = null;

    /// <summary>
    /// Max Ray Distance
    /// </summary>
    private float _gazeMaxDistance = 300;

    /// <summary>
    /// Max Ray Distance
    /// </summary>
    private float _gazeTimeCounter = 0;

    /// <summary>
    /// The cursor object will be created when the app is running,
    /// this will store its values.
    /// </summary>
    private GameObject _cursor;
}

```

5. Code for the **Awake()** and **Start()** methods now needs to be added.

```

private void Awake()
{
    // Set this class to behave similar to singleton
    Instance = this;
    _cursor = CreateCursor();
}

void Start()
{
    FocusedGameObject = null;
}

/// <summary>
/// Create a cursor object, to provide what the user
/// is looking at.
/// </summary>
/// <returns></returns>
private GameObject CreateCursor()
{
    GameObject newCursor = GameObject.CreatePrimitive(PrimitiveType.Sphere);

    // Remove the collider, so it does not block raycast.
    Destroy(newCursor.GetComponent<SphereCollider>());

    newCursor.transform.localScale = new Vector3(0.1f, 0.1f, 0.1f);

    newCursor.GetComponent<MeshRenderer>().material.color =
    Color.HSVToRGB(0.0223f, 0.7922f, 1.000f);

    newCursor.SetActive(false);
    return newCursor;
}

```

6. Inside the **Gaze** class, add the following code in the **Update()** method to project a *Raycast* and detect the target hit:

```

/// <summary>
/// Called every frame
/// </summary>
void Update()
{
    // Set the old focused gameobject.
    _oldFocusedObject = FocusedGameObject;

    RaycastHit hitInfo;

    // Initialize Raycasting.
    Hit = Physics.Raycast(Camera.main.transform.position, Camera.main.transform.forward, out
hitInfo, _gazeMaxDistance);

    // Check whether raycast has hit.
    if (Hit == true)
    {
        // Check whether the hit has a collider.
        if (hitInfo.collider != null)
        {
            // Set the focused object with what the user just looked at.
            FocusedGameObject = hitInfo.collider.gameObject;

            // Lerp the cursor to the hit point, which helps to stabilize the gaze.
            _cursor.transform.position = Vector3.Lerp(_cursor.transform.position, hitInfo.point,
0.6f);

            _cursor.SetActive(true);
        }
        else
        {
            // Object looked on is not valid, set focused gameobject to null.
            FocusedGameObject = null;

            _cursor.SetActive(false);
        }
    }
    else
    {
        // No object looked upon, set focused gameobject to null.
        FocusedGameObject = null;

        _cursor.SetActive(false);
    }

    // Check whether the previous focused object is this same object. If so, reset the focused
object.
    if (FocusedGameObject != _oldFocusedObject)
    {
        ResetFocusedObject();
    }
    // If they are the same, but are null, reset the counter.
    else if (FocusedGameObject == null && _oldFocusedObject == null)
    {
        _gazeTimeCounter = 0;
    }
    // Count whilst the user continues looking at the same object.
    else
    {
        _gazeTimeCounter += Time.deltaTime;
    }
}

```

7. Add the **ResetFocusedObject()** method, to send data to **Application Insights** when the user has looked at an object.

```

/// <summary>
/// Reset the old focused object, stop the gaze timer, and send data if it
/// is greater than one.
/// </summary>
public void ResetFocusedObject()
{
    // Ensure the old focused object is not null.
    if (_oldFocusedObject != null)
    {
        // Only looking for objects with the correct tag.
        if (_oldFocusedObject.CompareTag("ObjectInScene"))
        {
            // Turn the timer into an int, and ensure that more than zero time has passed.
            int gazeAsInt = (int)_gazeTimeCounter;

            if (gazeAsInt > 0)
            {
                //Record the object gazed and duration of gaze for Analytics
                ApplicationInsightsTracker.Instance.RecordGazeMetrics(_oldFocusedObject.name,
gazeAsInt);
            }
            //Reset timer
            _gazeTimeCounter = 0;
        }
    }
}

```

8. You have now completed the **Gaze** script. Save your changes in *Visual Studio* before returning to *Unity*.

Chapter 8 - Create the ObjectTrigger class

The next script you need to create is **ObjectTrigger**, which is responsible for:

- Adding components needed for collision to the Main Camera.
- Detecting if the camera is near an object tagged as **ObjectInScene**.

To create the script:

1. Double-click on the **Scripts** folder, to open it.
2. Right-click inside the **Scripts** folder, click **Create C# > Script**. Name the script **ObjectTrigger**.
3. Double-click on the script to open it with Visual Studio. Replace the existing code with the following:

```

using UnityEngine;

public class ObjectTrigger : MonoBehaviour
{
    private void Start()
    {
        // Add the Collider and Rigidbody components,
        // and set their respective settings. This allows for collision.
        gameObject.AddComponent<SphereCollider>().radius = 1.5f;

        gameObject.AddComponent<Rigidbody>().useGravity = false;
    }

    /// <summary>
    /// Triggered when an object with a collider enters this objects trigger collider.
    /// </summary>
    /// <param name="collision">Collided object</param>
    private void OnCollisionEnter(Collision collision)
    {
        CompareTriggerEvent(collision, true);
    }

    /// <summary>
    /// Triggered when an object with a collider exits this objects trigger collider.
    /// </summary>
    /// <param name="collision">Collided object</param>
    private void OnCollisionExit(Collision collision)
    {
        CompareTriggerEvent(collision, false);
    }

    /// <summary>
    /// Method for providing debug message, and sending event information to InsightsTracker.
    /// </summary>
    /// <param name="other">Collided object</param>
    /// <param name="enter">Enter = true, Exit = False</param>
    private void CompareTriggerEvent(Collision other, bool enter)
    {
        if (other.collider.CompareTag("ObjectInScene"))
        {
            string message = $"User is{(enter == true ? " " : " no longer ")}near <b>{other.gameObject.name}</b>";

            if (enter == true)
            {
                ApplicationInsightsTracker.Instance.RecordProximityEvent(other.gameObject.name);
            }
            Debug.Log(message);
        }
    }
}

```

4. Be sure to save your changes in *Visual Studio* before returning to *Unity*.

Chapter 9 - Create the DataFromAnalytics class

You will now need to create the **DataFromAnalytics** script, which is responsible for:

- Fetching analytics data about which object has been approached by the camera the most.
- Using the *Service Keys*, that allow communication with your Azure Application Insights Service instance.
- Sorting the objects in scene, according to which has the highest event count.
- Changing the material color, of the most approached object, to *green*.

To create the script:

1. Double-click on the **Scripts** folder, to open it.
2. Right-click inside the **Scripts** folder, click **Create C# > Script**. Name the script **DataFromAnalytics**.
3. Double-click on the script to open it with Visual Studio.
4. Insert the following namespaces:

```
using Newtonsoft.Json;
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using UnityEngine;
using UnityEngine.Networking;
```

5. Inside the script, insert the following:

```
/// <summary>
/// Number of most recent events to be queried
/// </summary>
private int _quantityOfEventsQueried = 10;

/// <summary>
/// The timespan with which to query. Needs to be in hours.
/// </summary>
private int _timespanAsHours = 24;

/// <summary>
/// A list of the objects in the scene
/// </summary>
private List<GameObject> _listOfGameObjectsInScene;

/// <summary>
/// Number of queries which have returned, after being sent.
/// </summary>
private int _queriesReturned = 0;

/// <summary>
/// List of GameObjects, as the Key, with their event count, as the Value.
/// </summary>
private List<KeyValuePair<GameObject, int>> _pairedObjectsWithEventCount = new
List<KeyValuePair<GameObject, int>>();

// Use this for initialization
void Start()
{
    // Find all objects in scene which have the ObjectInScene tag (as there may be other GameObjects
    // in the scene which you do not want).
    _listOfGameObjectsInScene = GameObject.FindGameObjectsWithTag("ObjectInScene").ToList();

    FetchAnalytics();
}
```

6. Within the **DataFromAnalytics** class, right after the **Start()** method, add the following method called **FetchAnalytics()**. This method is responsible for populating the list of key value pairs, with a *GameObject* and a placeholder event count number. It then initializes the **GetWebRequest()** coroutine. The query structure of the call to *Application Insights* can be found within this method also, as the *Query URL* endpoint.

```

private void FetchAnalytics()
{
    // Iterate through the objects in the list
    for (int i = 0; i < _listOfGameObjectsInScene.Count; i++)
    {
        // The current event number is not known, so set it to zero.
        int eventCount = 0;

        // Add new pair to list, as placeholder, until eventCount is known.
        _pairedObjectsWithEventCount.Add(new KeyValuePair<GameObject, int>
        (_listOfGameObjectsInScene[i], eventCount));

        // Set the renderer of the object to the default color, white
        _listOfGameObjectsInScene[i].GetComponent<Renderer>().material.color = Color.white;

        // Create the appropriate object name using Insights structure
        string objectName = _listOfGameObjectsInScene[i].name;

        // Build the queryUrl for this object.
        string queryUrl = Uri.EscapeUriString(string.Format(
            "https://api.applicationinsights.io/v1/apps/{0}/events/$all?timespan=PT{1}H&$search=
            {2}&$select=customMetric/name&$top={3}&$count=true",
            ApplicationInsightsTracker.Instance.applicationId, _timeSpanAsHours, "Gazed " + objectName,
            _quantityOfEventsQueried));

        // Send this object away within the WebRequest Coroutine, to determine it is event count.
        StartCoroutine("GetWebRequest", new KeyValuePair<string, int>(queryUrl, i));
    }
}

```

7. Right below the **FetchAnalytics()** method, add a method called **GetWebRequest()**, which returns an *IEnumerator*. This method is responsible for requesting the number of times an event, corresponding with a specific *GameObject*, has been called within *Application Insights*. When all the sent queries have returned, the **DetermineWinner()** method is called.

```

/// <summary>
/// Requests the data count for number of events, according to the
/// input query URL.
/// </summary>
/// <param name="webQueryPair">Query URL and the list number count.</param>
/// <returns></returns>
private IEnumerator GetWebRequest(KeyValuePair<string, int> webQueryPair)
{
    // Set the URL and count as their own variables (for readability).
    string url = webQueryPair.Key;
    int currentCount = webQueryPair.Value;

    using (UnityWebRequest unityWebRequest = UnityWebRequest.Get(url))
    {
        DownloadHandlerBuffer handlerBuffer = new DownloadHandlerBuffer();

        unityWebRequest.downloadHandler = handlerBuffer;

        unityWebRequest.SetRequestHeader("host", "api.applicationinsights.io");

        unityWebRequest.SetRequestHeader("x-api-key", ApplicationInsightsTracker.Instance.API_Key);

        yield return unityWebRequest.SendWebRequest();

        if (unityWebRequest.isNetworkError)
        {
            // Failure with web request.
            Debug.Log("<color=red>Error Sending:</color> " + unityWebRequest.error);
        }
        else
        {
            // This query has returned, so add to the current count.
            _queriesReturned++;

            // Initialize event count integer.
            int eventCount = 0;

            // Deserialize the response with the custom Analytics class.
            Analytics welcome = JsonConvert.DeserializeObject<Analytics>
(unityWebRequest.downloadHandler.text);

            // Get and return the count for the Event
            if (int.TryParse(welcome.OdataCount, out eventCount) == false)
            {
                // Parsing failed. Can sometimes mean that the Query URL was incorrect.
                Debug.Log("<color=red>Failure to Parse Data Results. Check Query URL for issues.
</color>");
            }
            else
            {
                // Overwrite the current pair, with its actual values, now that the event count is
known.
                _pairedObjectsWithEventCount[currentCount] = new KeyValuePair<GameObject, int>
(_pairedObjectsWithEventCount[currentCount].Key, eventCount);
            }

            // If all queries (compared with the number which was sent away) have
            // returned, then run the determine winner method.
            if (_queriesReturned == _pairedObjectsWithEventCount.Count)
            {
                DetermineWinner();
            }
        }
    }
}

```

8. The next method is **DetermineWinner()**, which sorts the list of *GameObject* and *Int* pairs, according to the highest event count. It then changes the material color of that *GameObject* to *green* (as feedback for it having the highest count). This displays a message with the analytics results.

```

/// <summary>
/// Call to determine the keyValue pair, within the objects list,
/// with the highest event count.
/// </summary>
private void DetermineWinner()
{
    // Sort the values within the list of pairs.
    _pairedObjectsWithEventCount.Sort((x, y) => y.Value.CompareTo(x.Value));

    // Change its colour to green
    _pairedObjectsWithEventCount.First().Key.GetComponent<Renderer>().material.color = Color.green;

    // Provide the winner, and other results, within the console window.
    string message = $"<b>Analytics Results:</b>\n " +
        $"<i>{_pairedObjectsWithEventCount.First().Key.name}</i> has the highest event count, " +
        $"with <i>{_pairedObjectsWithEventCount.First().Value.ToString()}</i>.\nFollowed by: ";

    for (int i = 1; i < _pairedObjectsWithEventCount.Count; i++)
    {
        message += $"{_pairedObjectsWithEventCount[i].Key.name}, " +
            $"with {_pairedObjectsWithEventCount[i].Value.ToString()} events.\n";
    }

    Debug.Log(message);
}

```

9. Add the class structure which will be used to deserialize the JSON object, received from *Application Insights*. Add these classes at the very bottom of your **DataFromAnalytics** class file, **outside** of the class definition.

```

/// <summary>
/// These classes represent the structure of the JSON response from Azure Insight
/// </summary>
[Serializable]
public class Analytics
{
    [JsonProperty("@odata.context")]
    public string OdataContext { get; set; }

    [JsonProperty("@odata.count")]
    public string OdataCount { get; set; }

    [JsonProperty("value")]
    public Value[] Value { get; set; }
}

[Serializable]
public class Value
{
    [JsonProperty("customMetric")]
    public CustomMetric CustomMetric { get; set; }
}

[Serializable]
public class CustomMetric
{
    [JsonProperty("name")]
    public string Name { get; set; }
}

```

10. Be sure to save your changes in *Visual Studio* before returning to *Unity*.

Chapter 10 - Create the Movement class

The **Movement** script is the next script you will need to create. It is responsible for:

- Moving the Main Camera according to the direction the camera is looking towards.
- Adding all other scripts to scene objects.

To create the script:

1. Double-click on the **Scripts** folder, to open it.
2. Right-click inside the **Scripts** folder, click **Create > C# Script**. Name the script **Movement**.
3. Double-click on the script to open it with *Visual Studio*.
4. Replace the existing code with the following:

```

using UnityEngine;
using UnityEngine.XR.WSA.Input;

public class Movement : MonoBehaviour
{
    /// <summary>
    /// The rendered object representing the right controller.
    /// </summary>
    public GameObject Controller;

    /// <summary>
    /// The movement speed of the user.
    /// </summary>
    public float UserSpeed;

    /// <summary>
    /// Provides whether source updates have been registered.
    /// </summary>
    private bool _isAttached = false;

    /// <summary>
    /// The chosen controller hand to use.
    /// </summary>
    private InteractionSourceHandedness _handness = InteractionSourceHandedness.Right;

    /// <summary>
    /// Used to calculate and proposes movement translation.
    /// </summary>
    private Vector3 _playerMovementTranslation;

    private void Start()
    {
        // You are now adding components dynamically
        // to ensure they are existing on the correct object

        // Add all camera related scripts to the camera.
        Camera.main.gameObject.AddComponent<Gaze>();
        Camera.main.gameObject.AddComponent<ObjectTrigger>();

        // Add all other scripts to this object.
        gameObject.AddComponent<ApplicationInsightsTracker>();
        gameObject.AddComponent<DataFromAnalytics>();
    }

    // Update is called once per frame
    void Update()
    {
    }
}

```

5. Within the **Movement** class, *below* the empty **Update()** method, insert the following methods that allow the user to use the hand controller to move in the virtual space:

```

/// <summary>
/// Used for tracking the current position and rotation of the controller.
/// </summary>
private void UpdateControllerState()
{
#if UNITY_WSA && UNITY_2017_2_OR_NEWER
    // Check for current connected controllers, only if WSA.
    string message = string.Empty;

    if (InteractionManager.GetCurrentReading().Length > 0)
    {
        foreach (var sourceState in InteractionManager.GetCurrentReading())

```

```

    {
        if (sourceState.source.kind == InteractionSourceKind.Controller &&
sourceState.source.handedness == _handness)
        {
            // If a controller source is found, which matches the selected handness,
            // check whether interaction source updated events have been registered.
            if (_isAttached == false)
            {
                // Register events, as not yet registered.
                message = "<color=green>Source Found: Registering Controller Source
Events</color>";
                _isAttached = true;

                InteractionManager.InteractionSourceUpdated +=
InteractionManager_InteractionSourceUpdated;
            }

            // Update the position and rotation information for the controller.
            Vector3 newPosition;
            if (sourceState.sourcePose.TryGetPosition(out newPosition,
InteractionSourceNode.Pointer) && ValidPosition(newPosition))
            {
                Controller.transform.localPosition = newPosition;
            }

            Quaternion newRotation;

            if (sourceState.sourcePose.TryGetRotation(out newRotation,
InteractionSourceNode.Pointer) && ValidRotation(newRotation))
            {
                Controller.transform.localRotation = newRotation;
            }
        }
    }
    else
    {
        // Controller source not detected.
        message = "<color=blue>Trying to detect controller source</color>";

        if (_isAttached == true)
        {
            // A source was previously connected, however, has been lost. Disconnected
            // all registered events.

            _isAttached = false;

            InteractionManager.InteractionSourceUpdated -=
InteractionManager_InteractionSourceUpdated;

            message = "<color=red>Source Lost: Detaching Controller Source Events</color>";
        }
    }

    if(message != string.Empty)
    {
        Debug.Log(message);
    }
#endif
}

```

```

/// <summary>
/// This registered event is triggered when a source state has been updated.
/// </summary>
/// <param name="obj"></param>
private void InteractionManager_InteractionSourceUpdated(InteractionSourceUpdatedEventArgs obj)
{
    if (obj.state.source.handedness == _handness)
    {
        if(obj.state.thumbstickPosition.magnitude > 0.2f)
        {
            float thumbstickY = obj.state.thumbstickPosition.y;

            // Vertical Input.
            if (thumbstickY > 0.3f || thumbstickY < -0.3f)
            {
                _playerMovementTranslation = Camera.main.transform.forward;
                _playerMovementTranslation.y = 0;
                transform.Translate(_playerMovementTranslation * UserSpeed * Time.deltaTime *
thumbstickY, Space.World);
            }
        }
    }
}

```

```

/// <summary>
/// Check that controller position is valid.
/// </summary>
/// <param name="inputVector3">The Vector3 to check</param>
/// <returns>The position is valid</returns>
private bool ValidPosition(Vector3 inputVector3)
{
    return !float.IsNaN(inputVector3.x) && !float.IsNaN(inputVector3.y) &&
!float.IsNaN(inputVector3.z) && !float.IsInfinity(inputVector3.x) && !float.IsInfinity(inputVector3.y) &&
!float.IsInfinity(inputVector3.z);
}

/// <summary>
/// Check that controller rotation is valid.
/// </summary>
/// <param name="inputQuaternion">The Quaternion to check</param>
/// <returns>The rotation is valid</returns>
private bool ValidRotation(Quaternion inputQuaternion)
{
    return !float.IsNaN(inputQuaternion.x) && !float.IsNaN(inputQuaternion.y) &&
!float.IsNaN(inputQuaternion.z) && !float.IsNaN(inputQuaternion.w) &&
!float.IsInfinity(inputQuaternion.x) && !float.IsInfinity(inputQuaternion.y) &&
!float.IsInfinity(inputQuaternion.z) && !float.IsInfinity(inputQuaternion.w);
}

```

- Lastly add the method call within the **Update()** method.

```

// Update is called once per frame
void Update()
{
    UpdateControllerState();
}

```

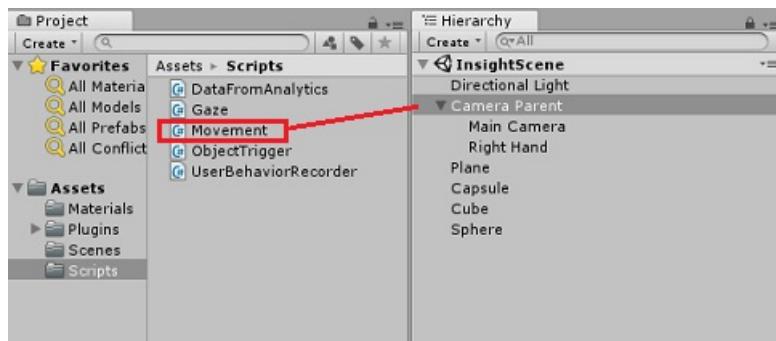
- Be sure to save your changes in *Visual Studio* before returning to *Unity*.

Chapter 11 - Setting up the scripts references

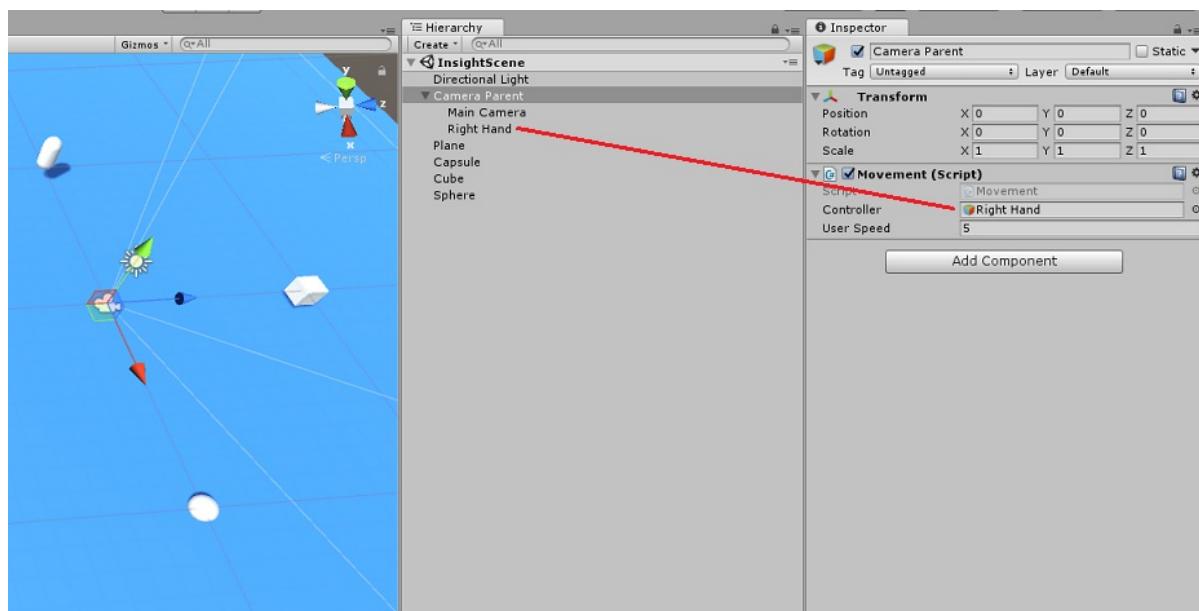
In this Chapter you need to place the **Movement** script onto the **Camera Parent** and set its reference targets.

That script will then handle placing the other scripts where they need to be.

1. From the **Scripts** folder in the *Project Panel*, drag the **Movement** script to the **Camera Parent** object, located in the *Hierarchy Panel*.



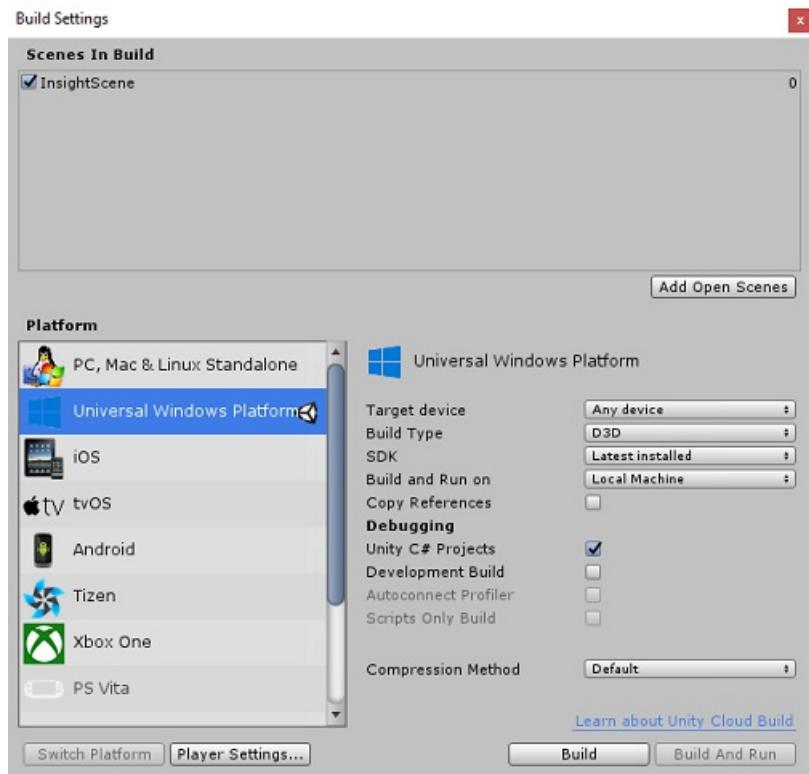
2. Click on the **Camera Parent**. In the *Hierarchy Panel*, drag the **Right Hand** object from the *Hierarchy Panel* to the reference target, **Controller**, in the *Inspector Panel*. Set the **User Speed** to **5**, as shown in the image below.



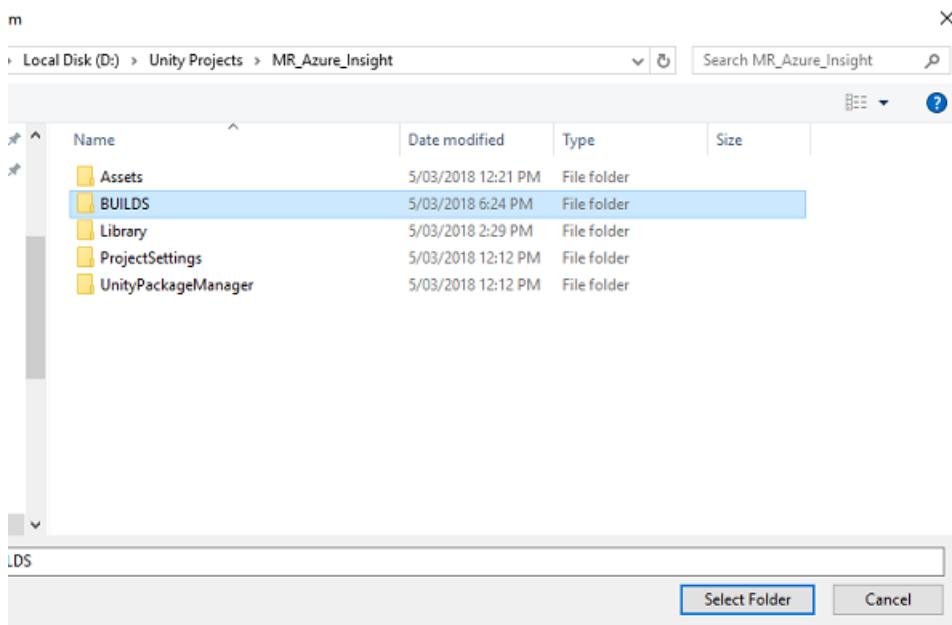
Chapter 12 - Build the Unity project

Everything needed for the Unity section of this project has now been completed, so it is time to build it from Unity.

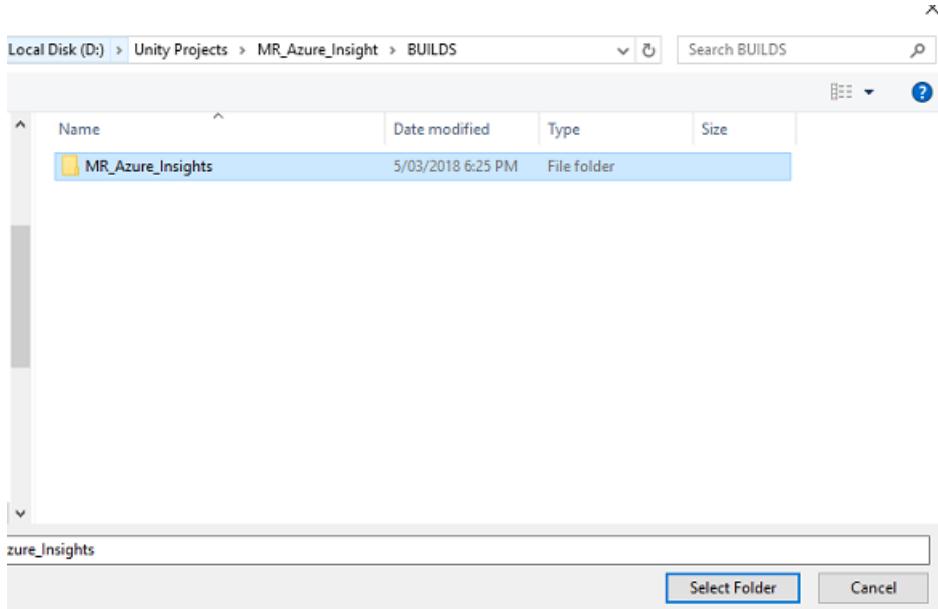
1. Navigate to **Build Settings**, (**File > Build Settings...**).
2. From the **Build Settings** window, click **Build**.



3. A **File Explorer** window will pop-up, prompting you for a location for the build. Create a new folder (by clicking **New Folder** in the top-left corner), and name it **BUILDS**.



- a. Open the new **BUILDS** folder, and create another folder (using **New Folder** once more), and name it **MR_Azure_Application_Insights**.

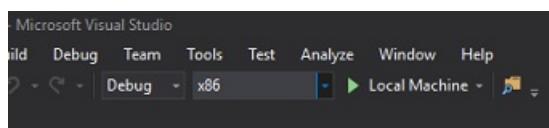


- b. With the **MR_Azure_Application_Insights** folder selected, click **Select Folder**. The project will take a minute or so to build.
4. Following *Build*, **File Explorer** will appear showing you the location of your new project.

Chapter 13 - Deploy MR_Azure_Application_Insights app to your machine

To deploy the **MR_Azure_Application_Insights** app on your Local Machine:

1. Open the solution file of your **MR_Azure_Application_Insights** app in **Visual Studio**.
2. In the **Solution Platform**, select **x86, Local Machine**.
3. In the **Solution Configuration** select **Debug**.



4. Go to **Build menu** and click on **Deploy Solution** to sideload the application to your machine.
5. Your app should now appear in the list of installed apps, ready to be launched.
6. Launch the mixed reality application.
7. Move around the scene, approaching objects and looking at them, when the *Azure Insight Service* has collected enough event data, it will set the object that has been approached the most to green.

IMPORTANT

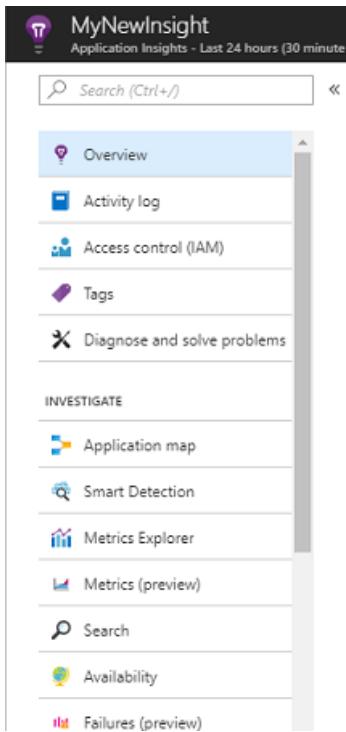
While the average waiting time for the *Events and Metrics* to be collected by the Service takes around 15 min, in some occasions it might take up to 1 hour.

Chapter 14 - The Application Insights Service portal

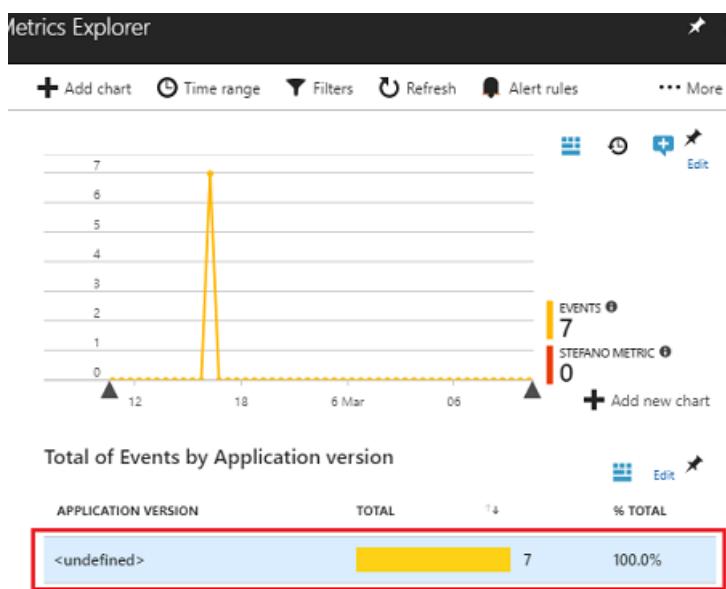
Once you have roamed around the scene and gazed at several objects you can see the data collected in the *Application Insights Service* portal.

1. Go back to your Application Insights Service portal.

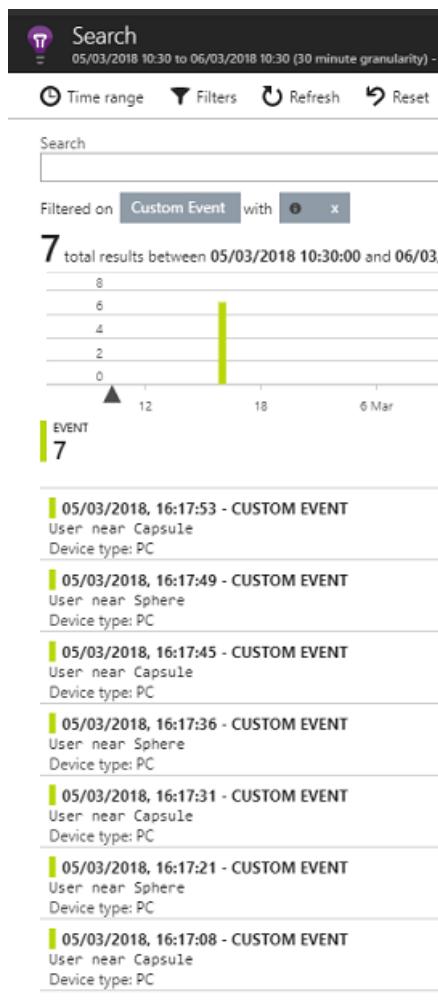
2. Click on *Metrics Explorer*.



3. It will open in a tab containing the graph which represent the *Events and Metrics* related to your application.
As mentioned above, it might take some time (up to 1 hour) for the data to be displayed in the graph

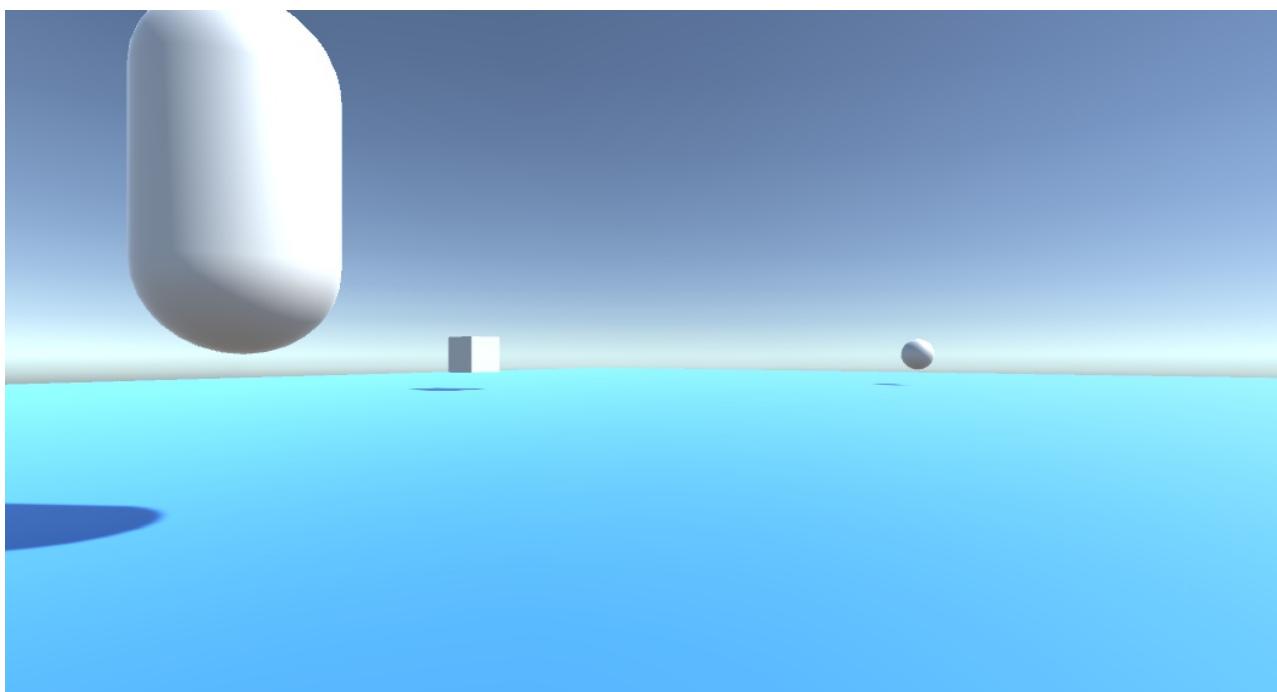


4. Click on the *Events bar* in the *Total of Events by Application Version*, to see a detailed breakdown of the events with their names.



Your finished your Application Insights Service application

Congratulations, you built a mixed reality app that leverages the Application Insights Service to monitor user's activity within your app.



Bonus Exercises

Exercise 1

Try spawning, rather than manually creating, the ObjectInScene objects and set their coordinates on the plane within your scripts. In this way, you could ask Azure what the most popular object was (either from gaze or proximity results) and spawn an *extra* one of those objects.

Exercise 2

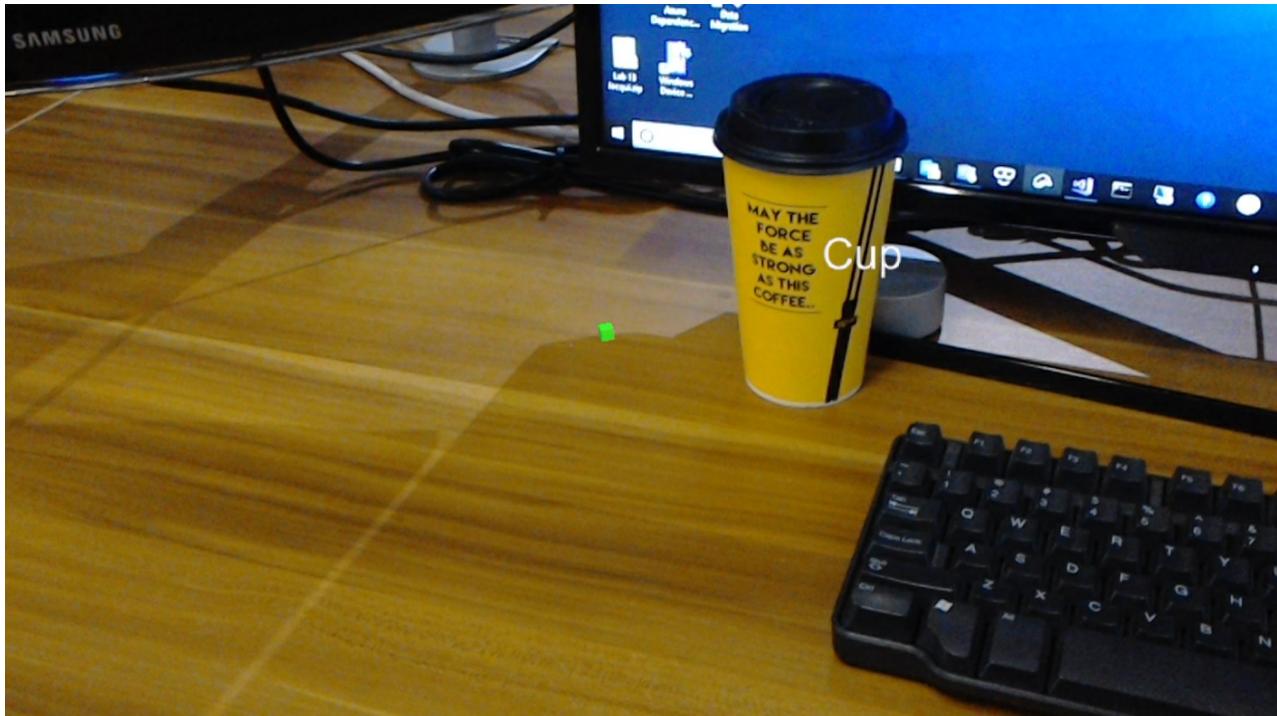
Sort your Application Insights results by time, so that you get the most relevant data, and implement that time sensitive data in your application.

Mr and Azure 310: Object detection

11/6/2018 • 34 minutes to read • [Edit Online](#)

In this course, you will learn how to recognize custom visual content and its spatial position within a provided image, using Azure Custom Vision "Object Detection" capabilities in a mixed reality application.

This service will allow you to train a machine learning model using object images. You will then use the trained model to recognize similar objects and approximate their location in the real world, as provided by the camera capture of Microsoft HoloLens or a camera connect to a PC for immersive (VR) headsets.



Azure Custom Vision, Object Detection is a Microsoft Service which allows developers to build custom image classifiers. These classifiers can then be used with new images to detect objects within that new image, by providing **Box Boundaries** within the image itself. The Service provides a simple, easy to use, online portal to streamline this process. For more information, visit the following links:

- [Azure Custom Vision page](#)
- [Limits and Quotas](#)

Upon completion of this course, you will have a mixed reality application which will be able to do the following:

1. The user will be able to *gaze* at an object, which they have trained using the Azure Custom Vision Service, Object Detection.
2. The user will use the *Tap* gesture to capture an image of what they are looking at.
3. The app will send the image to the Azure Custom Vision Service.
4. There will be a reply from the Service which will display the result of the recognition as world-space text. This will be accomplished through utilizing the Microsoft HoloLens' Spatial Tracking, as a way of understanding the world position of the recognized object, and then using the *Tag* associated with what is detected in the image, to provide the label text.

The course will also cover manually uploading images, creating tags, and training the Service, to recognize different objects (in the provided example, a cup), by setting the *Boundary Box* within the image you submit.

IMPORTANT

Following the creation and use of the app, the developer should navigate back to the Azure Custom Vision Service, and identify the predictions made by the Service, and determine whether they were correct or not (through tagging anything the Service missed, and adjusting the *Bounding Boxes*). The Service can then be re-trained, which will increase the likelihood of it recognizing real world objects.

This course will teach you how to get the results from the Azure Custom Vision Service, Object Detection, into a Unity-based sample application. It will be up to you to apply these concepts to a custom application you might be building.

Device support

COURSE	HOLOLENS	IMMERSIVE HEADSETS
MR and Azure 310: Object detection	✓ <input type="checkbox"/>	

Prerequisites

NOTE

This tutorial is designed for developers who have basic experience with Unity and C#. Please also be aware that the prerequisites and written instructions within this document represent what has been tested and verified at the time of writing (July 2018). You are free to use the latest software, as listed within the [install the tools](#) article, though it should not be assumed that the information in this course will perfectly match what you will find in newer software than what is listed below.

We recommend the following hardware and software for this course:

- A development PC
- [Windows 10 Fall Creators Update \(or later\) with Developer mode enabled](#)
- [The latest Windows 10 SDK](#)
- [Unity 2017.4 LTS](#)
- [Visual Studio 2017](#)
- A [Microsoft HoloLens](#) with Developer mode enabled
- Internet access for Azure setup and Custom Vision Service retrieval
- A series of at least fifteen (15) images are required for each object that you would like the Custom Vision to recognize. If you wish, you can use the images already provided with this course, [a series of cups](#)).

Before you start

1. To avoid encountering issues building this project, it is strongly suggested that you create the project mentioned in this tutorial in a root or near-root folder (long folder paths can cause issues at build-time).
2. Set up and test your HoloLens. If you need support setting up your HoloLens, [make sure to visit the HoloLens setup article](#).
3. It is a good idea to perform Calibration and Sensor Tuning when beginning developing a new HoloLens App (sometimes it can help to perform those tasks for each user).

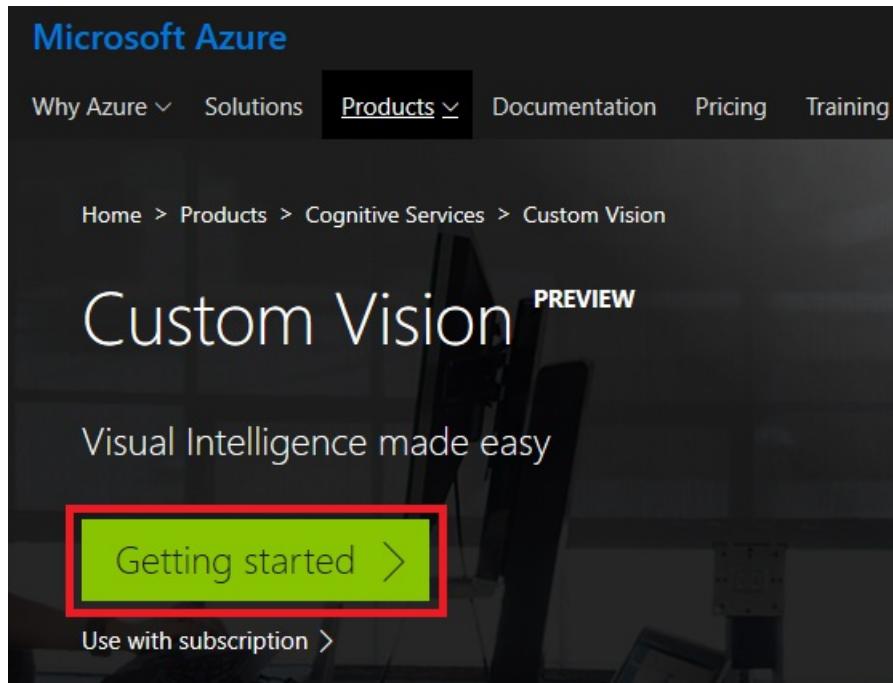
For help on Calibration, please follow this [link to the HoloLens Calibration article](#).

For help on Sensor Tuning, please follow this [link to the HoloLens Sensor Tuning article](#).

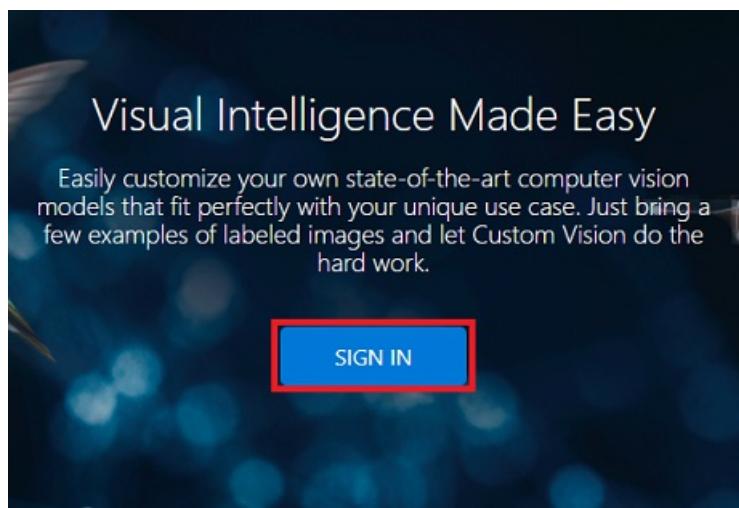
Chapter 1 - The Custom Vision Portal

To use the **Azure Custom Vision Service**, you will need to configure an instance of it to be made available to your application.

1. Navigate to the **Custom Vision Service** main page.
2. Click on **Getting Started**.



3. Sign in to the Custom Vision Portal.



4. If you do not already have an Azure account, you will need to create one. If you are following this tutorial in a classroom or lab situation, ask your instructor or one of the proctors for help setting up your new account.
5. Once you are logged in for the first time, you will be prompted with the *Terms of Service* panel. Click the checkbox to *agree to the terms*. Then click **I agree**.

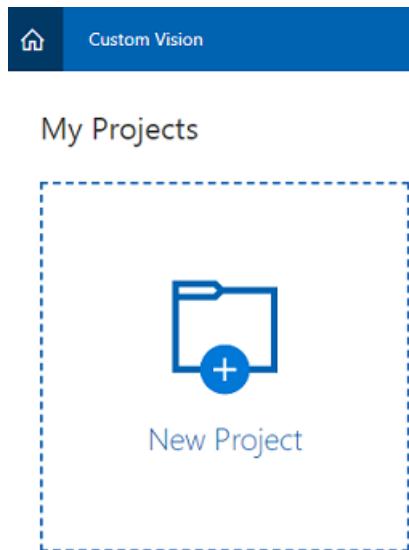
Terms of Service

Please note that Microsoft may retain copies of images uploaded for service improvement purposes. We won't publish your images or let other people use them.

I agree that my use of this service is governed by the [Microsoft Online Subscription Agreement](#), which incorporates the [Online Services Terms](#).

I agree

- Having agreed to the terms, you are now in the *My Projects* section. Click on **New Project**.



- A tab will appear on the right-hand side, which will prompt you to specify some fields for the project.

- Insert a name for your project
- Insert a description for your project (**Optional**)
- Choose a **Resource Group** or create a new one. A resource group provides a way to monitor, control access, provision and manage billing for a collection of Azure assets. It is recommended to keep all the Azure services associated with a single project (e.g. such as these courses) under a common resource group).



New project

Name*

CustomVisionObjDetection

Description

CustomVision - Object Detection on HoloLens using Unity

Resource Group*

[create new](#)

Limited trial

Project Types (i)

Classification

Object Detection (preview)

[Cancel](#)

[Create project](#)

NOTE

If you wish to [read more about Azure Resource Groups](#), navigate to the associated Docs

- d. Set the **Project Types** as **Object Detection (preview)**.
8. Once you are finished, click on **Create project**, and you will be redirected to the Custom Vision Service project page.

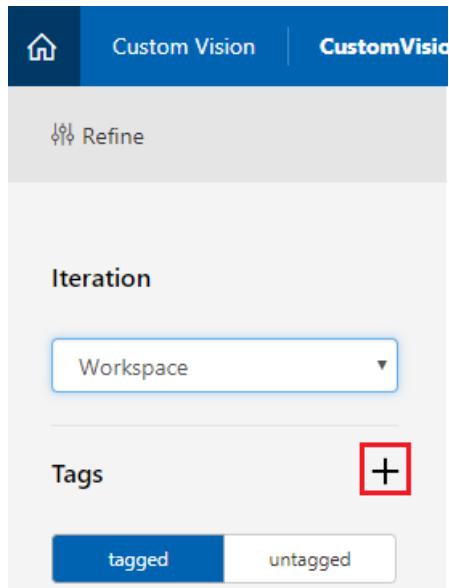
Chapter 2 - Training your Custom Vision project

Once in the Custom Vision Portal, your primary objective is to train your project to recognize specific objects in images.

You need at least fifteen (15) images for each object that you would like your application to recognize. You can use the images provided with this course ([a series of cups](#)).

To train your Custom Vision project:

1. Click on the + button next to **Tags**.



2. Add a **name** for the tag that will be used to associate your images with. In this example we are using images of cups for recognition, so have named the tag for this, **Cup**. Click **Save** once finished.

X

Name the tag

You will use tags to organize your images.

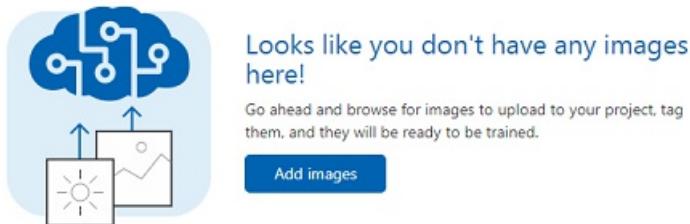
Cup

Save

3. You will notice your **Tag** has been added (you may need to reload your page for it to appear).

The screenshot shows the Microsoft Custom Vision Project Overview page. At the top, there's a navigation bar with icons for Home, Custom Vision, and a workspace named "CustomVisionOf...". Below the navigation, there's a "Refine" button with a gear icon. The main area is titled "Iteration" and contains a "Workspace" dropdown menu. Under "Tags", there are two buttons: "tagged" (which is selected and highlighted in blue) and "untagged". A message below says "Showing: all tagged images". There's a search bar labeled "Search for" and a checkbox labeled "Cup 0".

4. Click on **Add images** in the center of the page.

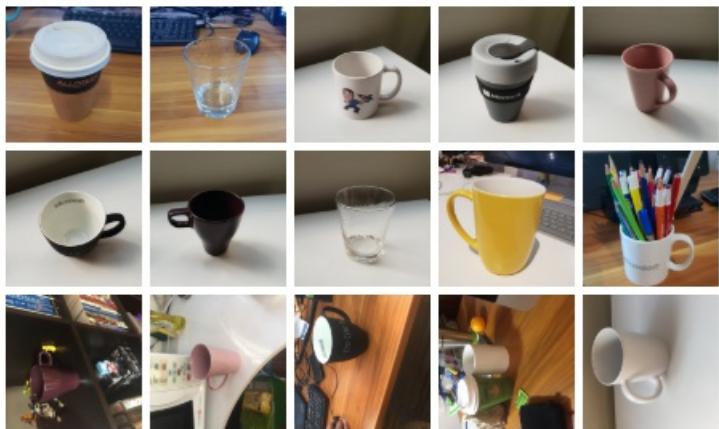


5. Click on **Browse local files**, and browse to the images you would like to upload for one object, with the minimum being fifteen (15).

TIP

You can select several images at a time, to upload.

X



15 images will be added...

Back

Upload 15 files

6. Press **Upload files** once you have selected all the images you would like to train the project with. The files will begin uploading. Once you have confirmation of the upload, click **Done**.

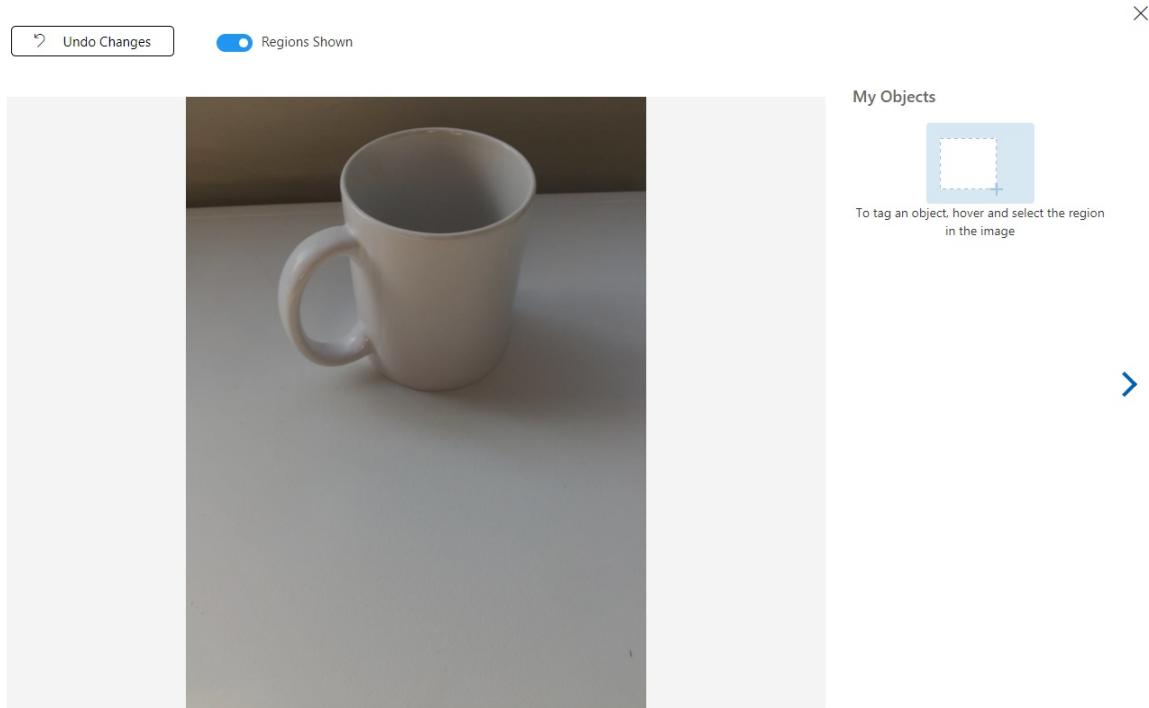
X



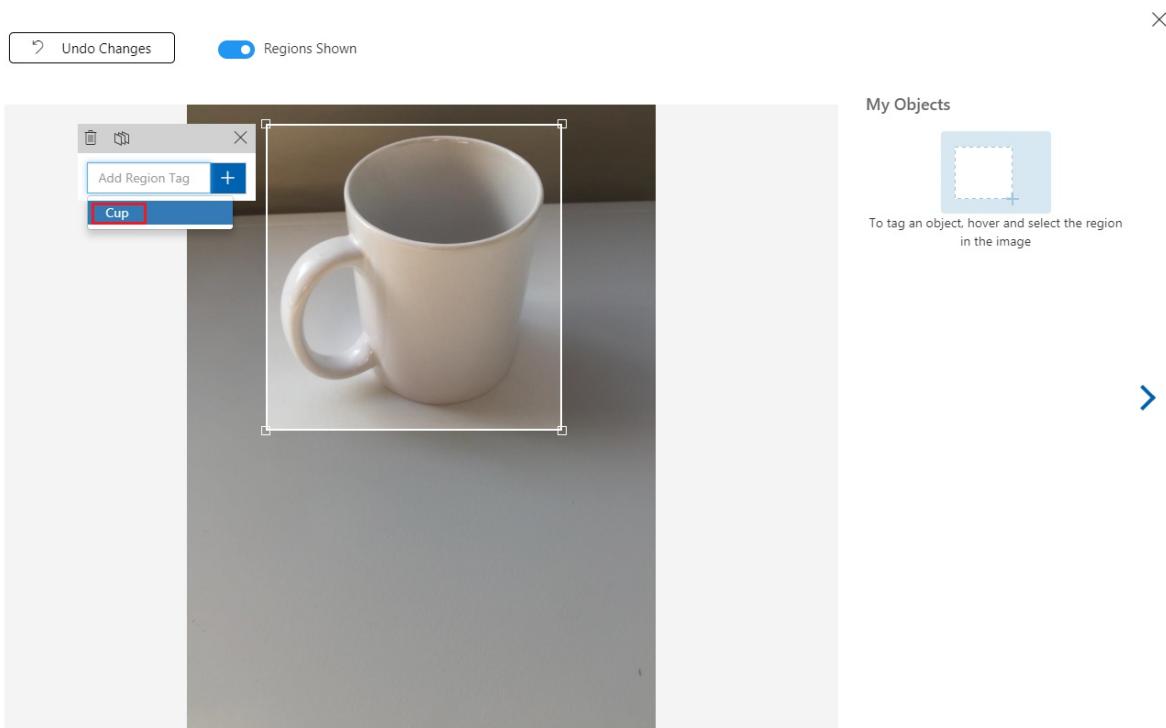
15 images uploaded successfully

Done

7. At this point your images are uploaded, but not tagged.



8. To tag your images, use your mouse. As you hover over your image, a selection highlight will aid you by automatically drawing a selection around your object. If it is not accurate, you can draw your own. This is accomplished by holding left-click on the mouse, and dragging the selection region to encompass your object.



9. Following the selection of your object within the image, a small prompt will ask for you to *Add Region Tag*. Select your previously created tag ('Cup', in the above example), or if you are adding more tags, type that in and click the **+** (plus) button.

X

 Undo Changes

 Regions Shown



My Objects

Cup X

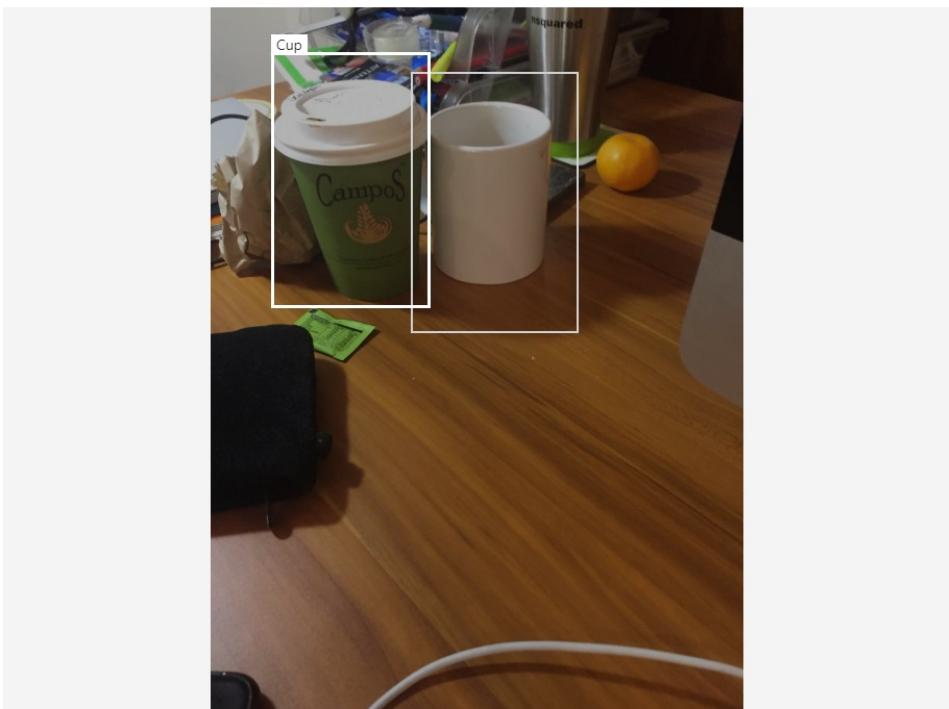
10. To tag the next image, you can click the arrow to the right of the blade, or close the tag blade (by clicking the X in the top-right corner of the blade) and then click the next image. Once you have the next image ready, repeat the same procedure. Do this for all the images you have uploaded, until they are all tagged.

NOTE

You can select several objects in the same image, like the image below:

 Undo Changes

 Regions Shown



My Objects

Cup X

11. Once you have tagged them all, click on the **tagged** button, on the left of the screen, to reveal the tagged images.

The screenshot shows the Microsoft Custom Vision interface for a project named "CustomVisionObjDet...". The top navigation bar includes links for "Custom Vision", "TRAINING IMAGES", "PERFORMANCE", "PREDICTIONS", "Train", "Quick Test", and a help icon. The main area is titled "Iteration" and shows a grid of training images. A sidebar on the left contains sections for "Iteration", "Tags" (with "tagged" selected), and a search bar. The "tagged" button is highlighted with a red box.

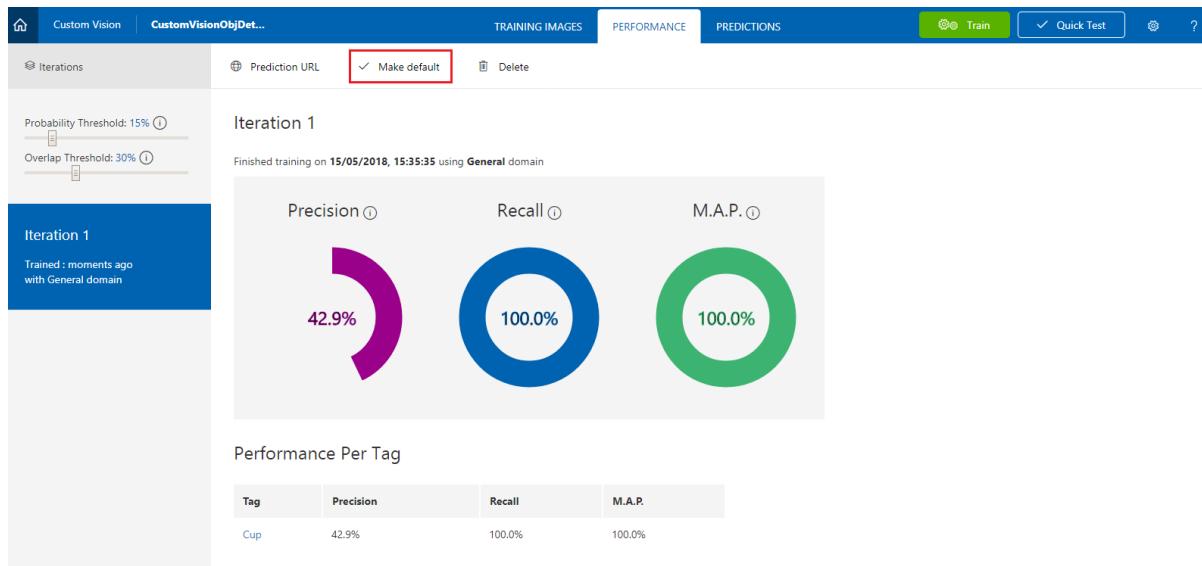
12. You are now ready to train your Service. Click the **Train** button, and the first training iteration will begin.

This screenshot shows the Microsoft Custom Vision interface with the "Train" button highlighted by a red box. The top navigation bar is visible with the "TRAINING IMAGES" tab active.



This screenshot shows the Microsoft Custom Vision interface focusing on the "Iterations" section for the first training iteration. It displays configuration options like Probability Threshold (15%) and Overlap Threshold (30%), and a status message indicating "Training..." and the last checked time as "15/05/2018, 15:35:11".

13. Once it is built, you will be able to see two buttons called **Make default** and **Prediction URL**. Click on **Make default** first, then click on **Prediction URL**.



NOTE

The endpoint which is provided from this, is set to whichever *Iteration* has been marked as default. As such, if you later make a new *Iteration* and update it as default, you will not need to change your code.

14. Once you have clicked on **Prediction URL**, open *Notepad*, and copy and paste the **URL** (also called your **Prediction-Endpoint**) and the **Service Prediction-Key**, so that you can retrieve it when you need it later in the code.



How to use the Prediction API

If you have an image URL:

```
https://southcentralus.api.cognitive.microsoft.com/customvision/v2.0/Predictic
Set Prediction-Key Header to : 1cf740dc97d84ae284be26c59a20322b
Set Content-Type Header to : application/json
Set Body to : {"Url": "https://example.com/image.png"}
```

If you have an image file:

```
https://southcentralus.api.cognitive.microsoft.com/customvision/v2.0/Predictic
Set Prediction-Key Header to : 1cf740dc97d84ae284be26c59a20322b
Set Content-Type Header to : application/octet-stream
Set Body to : <image file>
```

Remember, you can mark an iteration as Default so you can send data to it without specifying an iteration id. You can then change which iteration your app is pointing to without having to update your app.

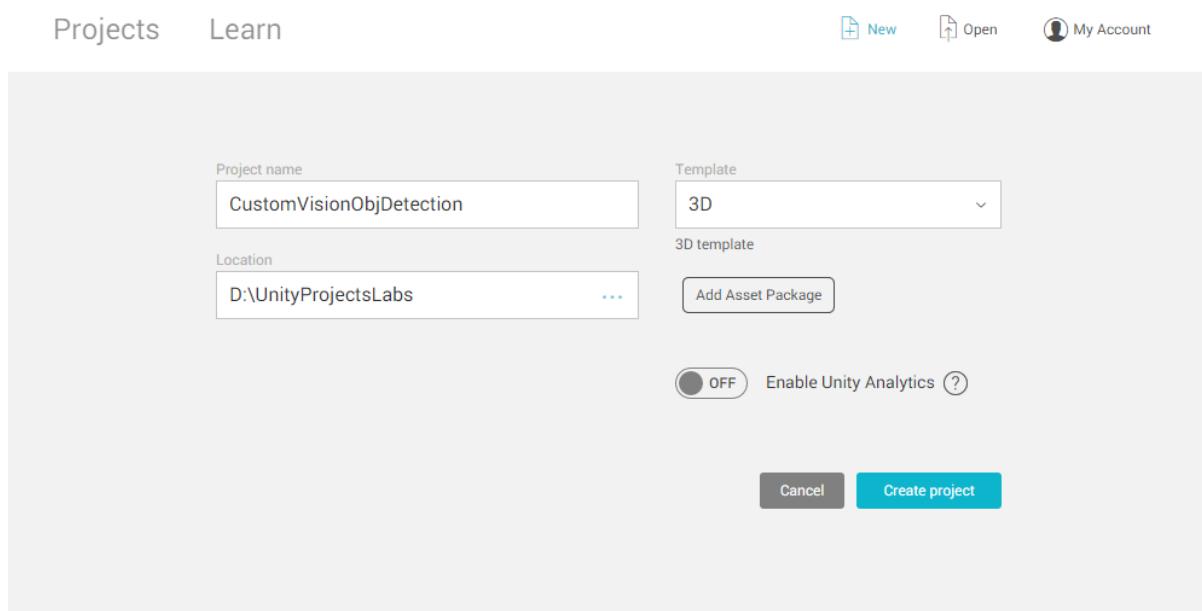
Chapter 3 - Set up the Unity project

The following is a typical set up for developing with mixed reality, and as such, is a good template for other projects.

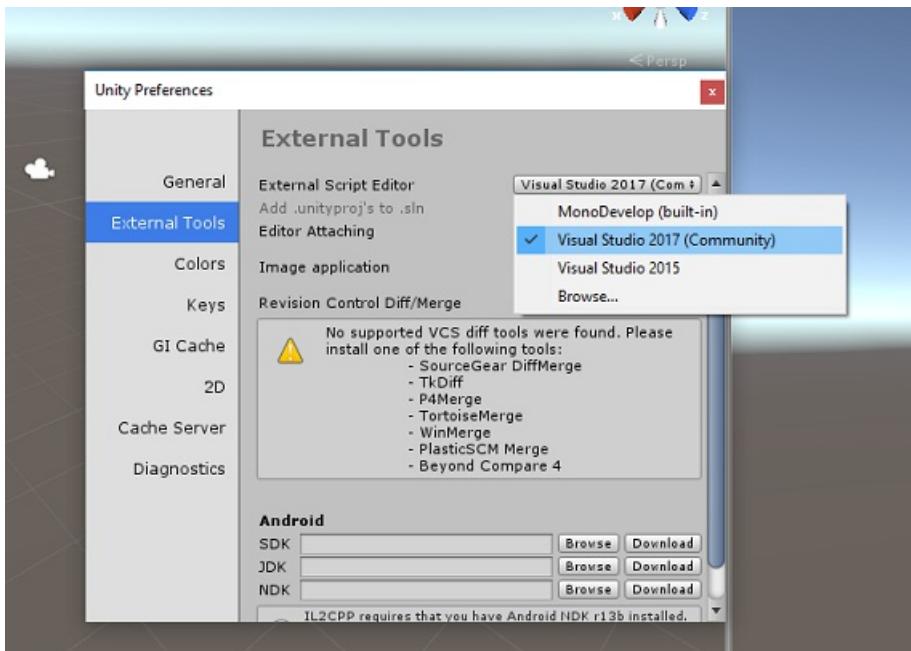
1. Open **Unity** and click **New**.



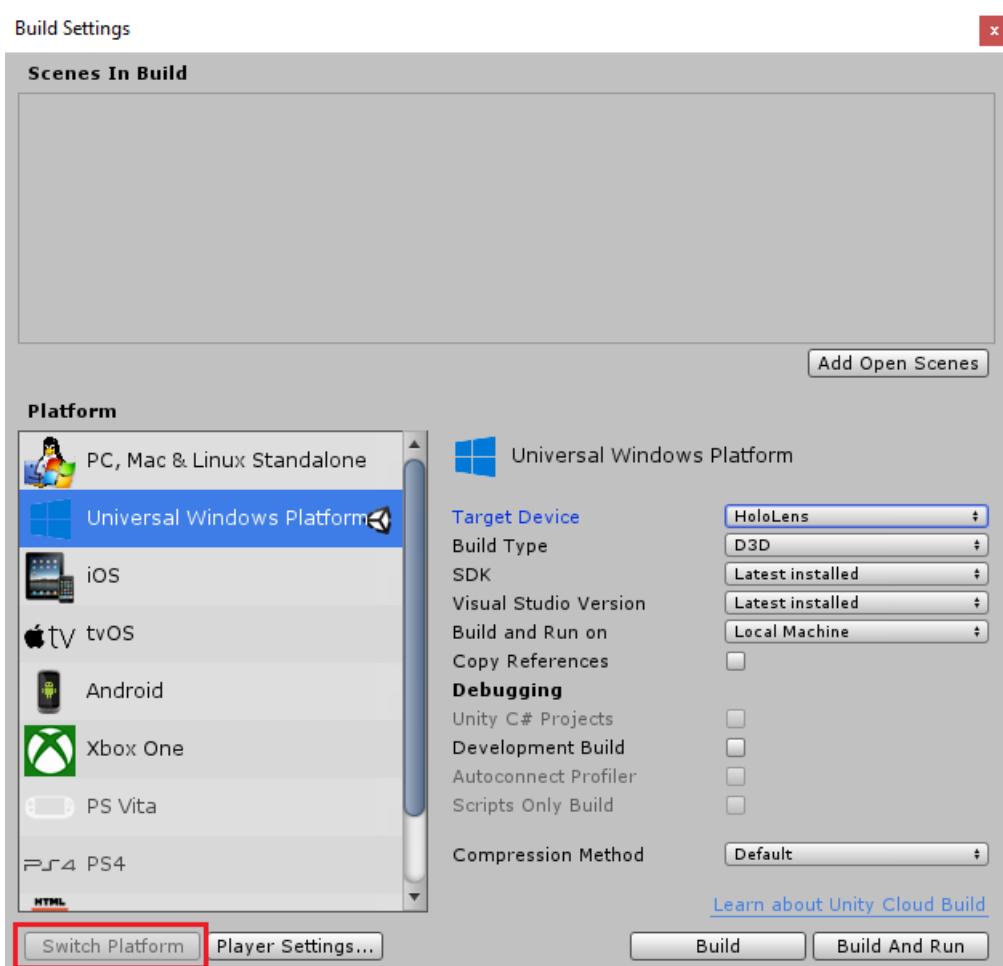
2. You will now need to provide a Unity project name. Insert **CustomVisionObjDetection**. Make sure the project type is set to **3D**, and set the **Location** to somewhere appropriate for you (remember, closer to root directories is better). Then, click **Create project**.



3. With Unity open, it is worth checking the default **Script Editor** is set to **Visual Studio**. Go to *Edit > Preferences* and then from the new window, navigate to **External Tools**. Change **External Script Editor** to **Visual Studio**. Close the **Preferences** window.



4. Next, go to **File > Build Settings** and switch the **Platform** to *Universal Windows Platform*, and then clicking on the **Switch Platform** button.

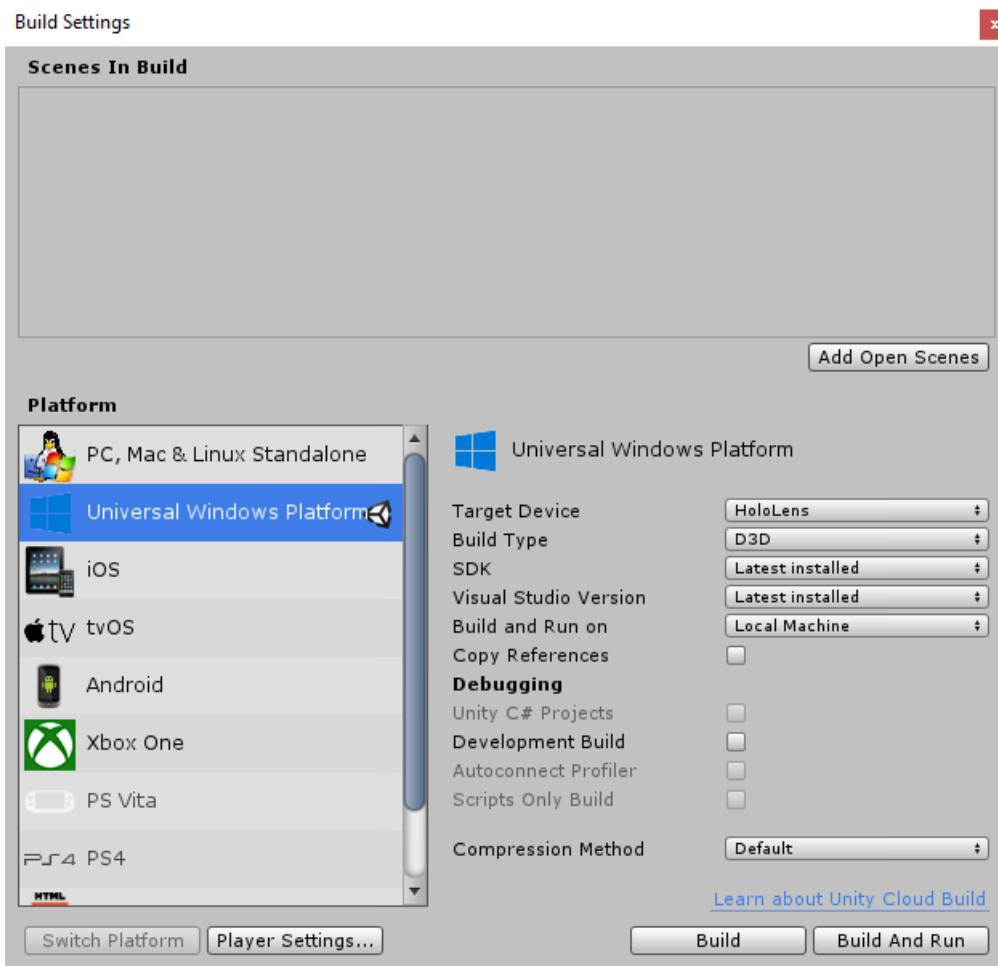


5. In the same **Build Settings** window, ensure the following are set:

- Target Device** is set to **HoloLens**
- Build Type** is set to **D3D**
- SDK** is set to **Latest installed**
- Visual Studio Version** is set to **Latest installed**

e. **Build and Run** is set to **Local Machine**

f. The remaining settings, in **Build Settings**, should be left as default for now.

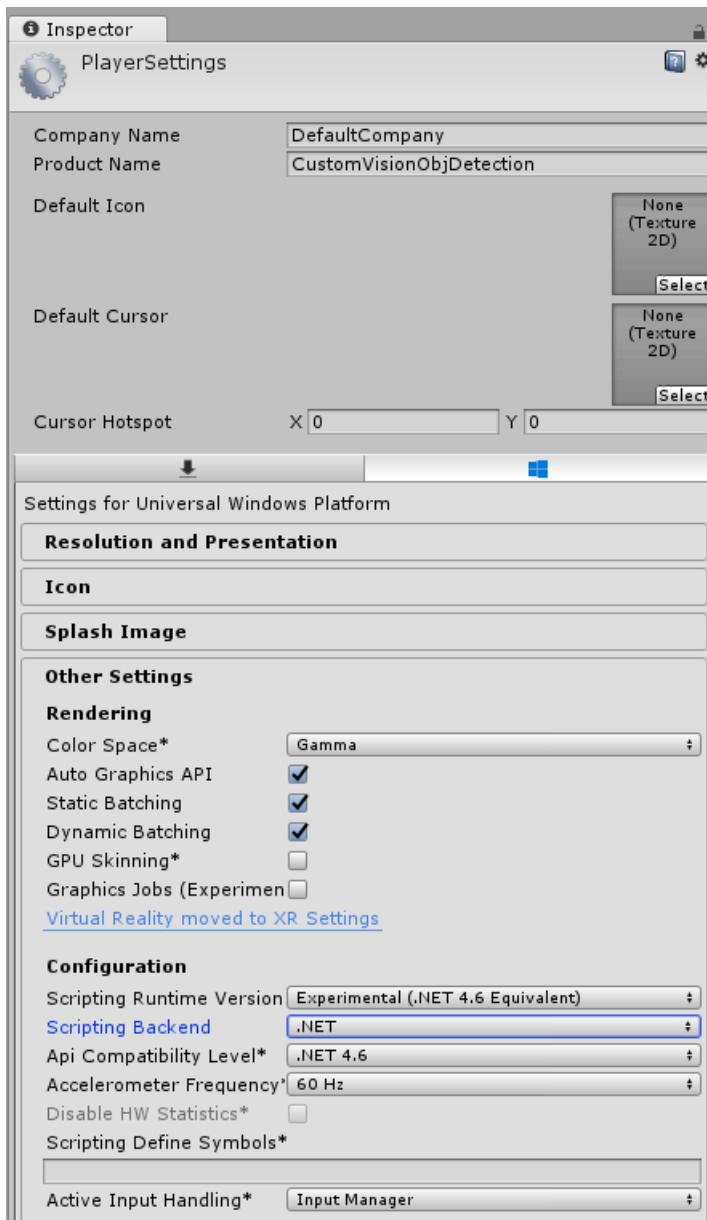


6. In the same **Build Settings** window, click on the **Player Settings** button, this will open the related panel in the space where the **Inspector** is located.

7. In this panel, a few settings need to be verified:

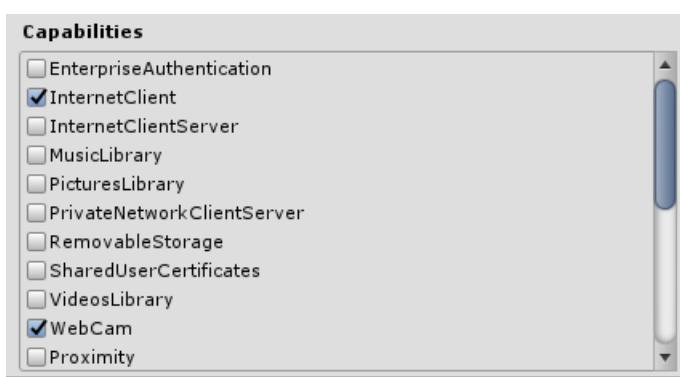
a. In the **Other Settings** tab:

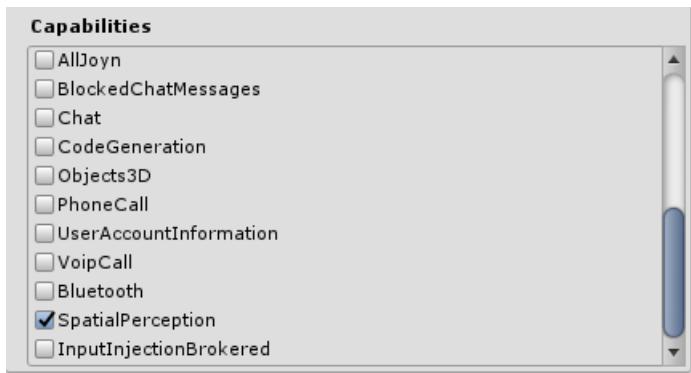
- Scripting Runtime Version** should be **Experimental (.NET 4.6 Equivalent)**, which will trigger a need to restart the Editor.
- Scripting Backend** should be **.NET**.
- API Compatibility Level** should be **.NET 4.6**.



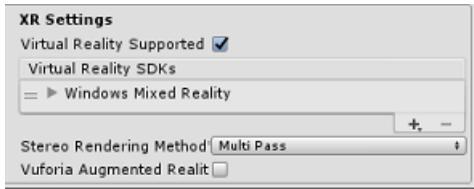
b. Within the **Publishing Settings** tab, under **Capabilities**, check:

- a. **InternetClient**
- b. **Webcam**
- c. **SpatialPerception**

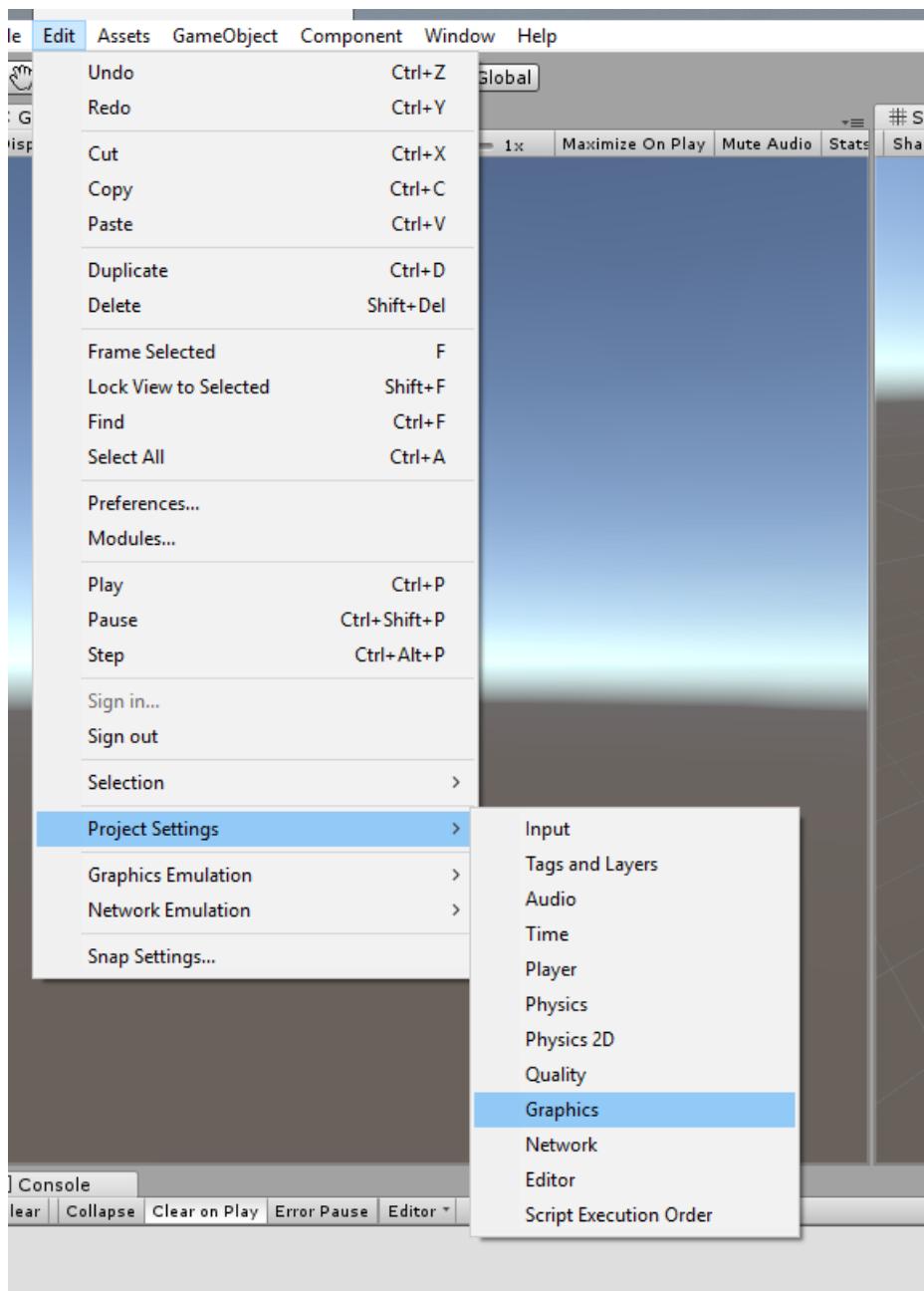




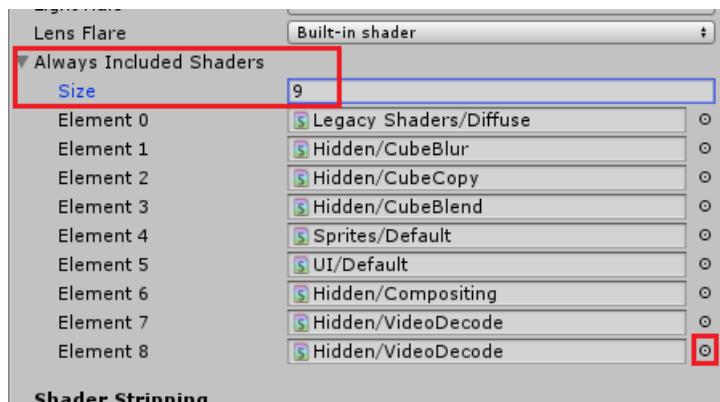
- c. Further down the panel, in **XR Settings** (found below **Publish Settings**), tick **Virtual Reality Supported**, then make sure the **Windows Mixed Reality SDK** is added.



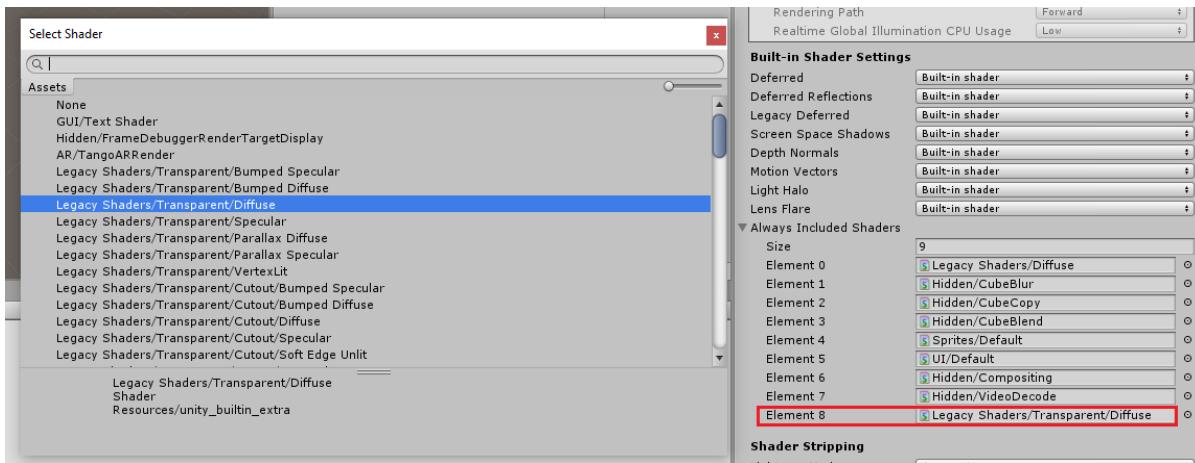
8. Back in **Build Settings**, *Unity C# Projects* is no longer greyed out: tick the checkbox next to this.
9. Close the **Build Settings** window.
10. In the **Editor**, click on **Edit > Project Settings > Graphics**.



11. In the **Inspector Panel** the *Graphics Settings* will be open. Scroll down until you see an array called **Always Include Shaders**. Add a slot by increasing the **Size** variable by one (in this example, it was 8 so we made it 9). A new slot will appear, in the last position of the array, as shown below:



12. In the slot, click on the small target circle next to the slot to open a list of shaders. Look for the **Legacy Shaders/Transparent/Diffuse** shader and double-click it.



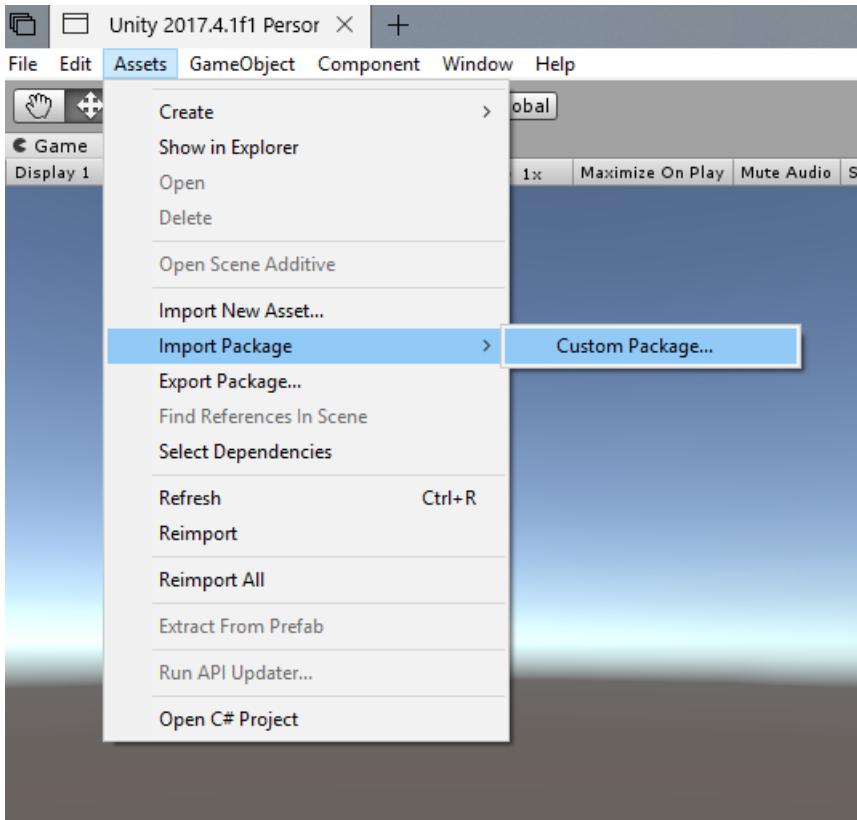
Chapter 4 - Importing the CustomVisionObjDetection Unity package

For this course you are provided with a Unity Asset Package called **Azure-MR-310.unitypackage**.

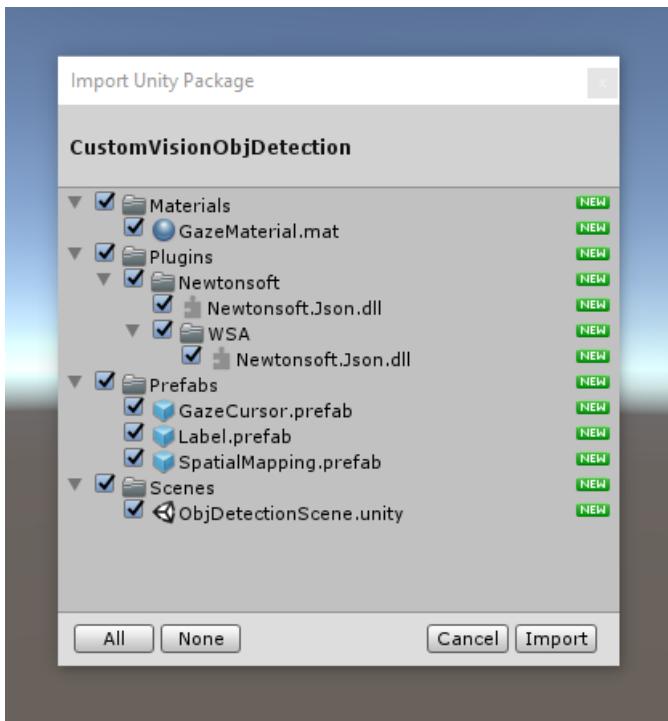
[TIP] Any objects supported by Unity, including entire scenes, can be packaged into a **.unitypackage** file, and exported / imported in other projects. It is the safest, and most efficient, way to move assets between different **Unity projects**.

You can find the [Azure-MR-310 package that you need to download here](#).

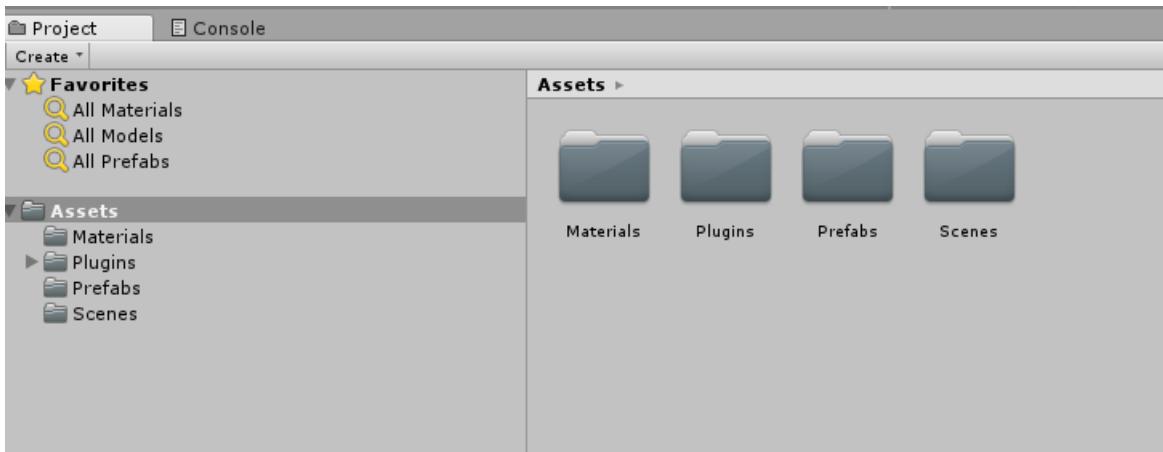
1. With the Unity dashboard in front of you, click on **Assets** in the menu at the top of the screen, then click on **Import Package > Custom Package**.



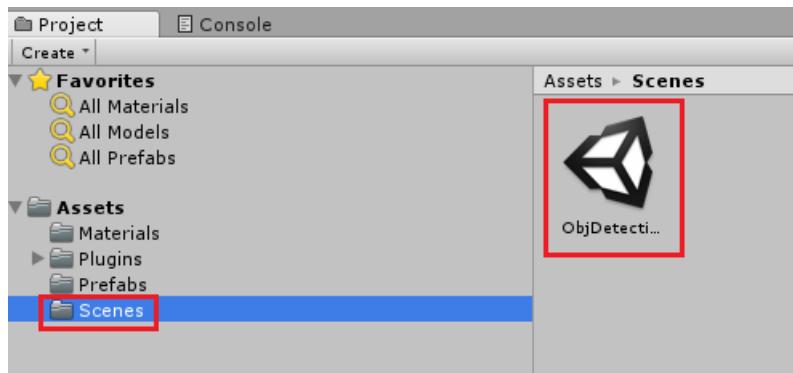
2. Use the file picker to select the **Azure-MR-310.unitypackage** package and click **Open**. A list of components for this asset will be displayed to you. Confirm the import by clicking the **Import** button.



3. Once it has finished importing, you will notice that folders from the package have now been added to your **Assets** folder. This kind of folder structure is typical for a Unity project.



- The **Materials** folder contains the material used by the **Gaze Cursor**.
 - The **Plugins** folder contains the Newtonsoft DLL used by the code to deserialize the Service web response. The two (2) different versions contained in the folder, and sub-folder, are necessary to allow the library to be used and built by both the Unity Editor and the UWP build.
 - The **Prefabs** folder contains the prefabs contained in the scene. Those are:
 - The **GazeCursor**, the cursor used in the application. Will work together with the SpatialMapping prefab to be able to be placed in the scene on top of physical objects.
 - The **Label**, which is the UI object used to display the object tag in the scene when required.
 - The **SpatialMapping**, which is the object that enables the application to use create a virtual map, using the Microsoft HoloLens' spatial tracking.
 - The **Scenes** folder which currently contains the pre-built scene for this course.
4. Open the **Scenes** folder, in the **Project Panel**, and double-click on the **ObjDetectionScene**, to load the scene that you will use for this course.



NOTE

No code is included, you will write the code by following this course.

Chapter 5 - Create the CustomVisionAnalyser class.

At this point you are ready to write some code. You will begin with the **CustomVisionAnalyser** class.

NOTE

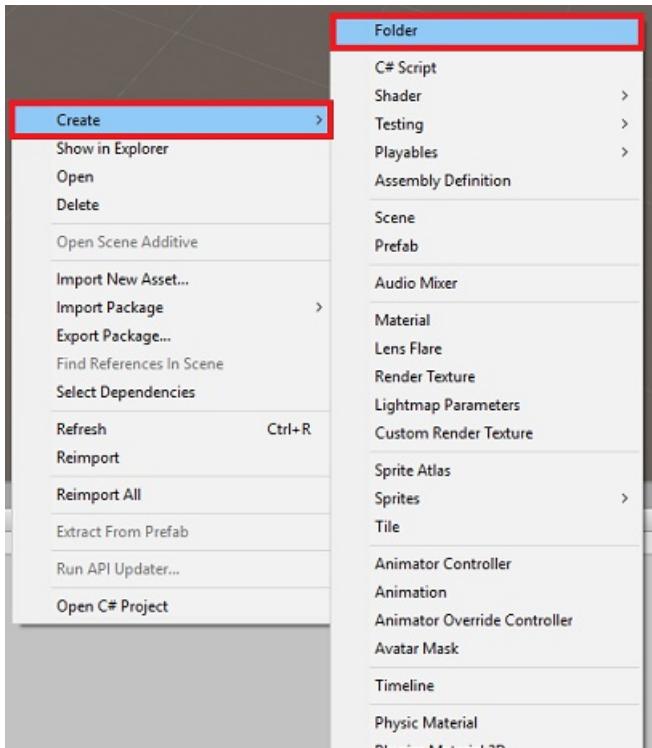
The calls to the **Custom Vision Service**, made in the code shown below, are made using the **Custom Vision REST API**. Through using this, you will see how to implement and make use of this API (useful for understanding how to implement something similar on your own). Be aware, that Microsoft offers a **Custom Vision SDK** that can also be used to make calls to the Service. For more information visit the [Custom Vision SDK article](#).

This class is responsible for:

- Loading the latest image captured as an array of bytes.
- Sending the byte array to your Azure **Custom Vision Service** instance for analysis.
- Receiving the response as a JSON string.
- Deserializing the response and passing the resulting **Prediction** to the **SceneOrganiser** class, which will take care of how the response should be displayed.

To create this class:

1. Right-click in the **Asset Folder**, located in the **Project Panel**, then click **Create > Folder**. Call the folder **Scripts**.



2. Double-click on the newly created folder, to open it.
3. Right-click inside the folder, then click **Create > C# Script**. Name the script **CustomVisionAnalyser**.
4. Double-click on the new **CustomVisionAnalyser** script to open it with **Visual Studio**.
5. Make sure you have the following namespaces referenced at the top of the file:

```
using Newtonsoft.Json;
using System.Collections;
using System.IO;
using UnityEngine;
using UnityEngine.Networking;
```

6. In the **CustomVisionAnalyser** class, add the following variables:

```
/// <summary>
/// Unique instance of this class
/// </summary>
public static CustomVisionAnalyser Instance;

/// <summary>
/// Insert your prediction key here
/// </summary>
private string predictionKey = "- Insert your key here -";

/// <summary>
/// Insert your prediction endpoint here
/// </summary>
private string predictionEndpoint = "Insert your prediction endpoint here";

/// <summary>
/// Byte array of the image to submit for analysis
/// </summary>
[HideInInspector] public byte[] imageBytes;
```

NOTE

Make sure you insert your **Service Prediction-Key** into the **predictionKey** variable and your **Prediction-Endpoint** into the **predictionEndpoint** variable. You copied these to [Notepad earlier, in Chapter 2, Step 14](#).

7. Code for **Awake()** now needs to be added to initialize the Instance variable:

```
/// <summary>
/// Initializes this class
/// </summary>
private void Awake()
{
    // Allows this instance to behave like a singleton
    Instance = this;
}
```

8. Add the coroutine (with the static **GetImageAsByteArray()** method below it), which will obtain the results of the analysis of the image, captured by the **ImageCapture** class.

NOTE

In the **AnalyseImageCapture** coroutine, there is a call to the **SceneOrganiser** class that you are yet to create. Therefore, **leave those lines commented for now**.

```

/// <summary>
/// Call the Computer Vision Service to submit the image.
/// </summary>
public IEnumerator AnalyseLastImageCaptured(string imagePath)
{
    Debug.Log("Analyzing...");

    WWWForm webForm = new WWWForm();

    using (UnityWebRequest unityWebRequest = UnityWebRequest.Post(predictionEndpoint, webForm))
    {
        // Gets a byte array out of the saved image
        imageBytes = GetImageAsByteArray(imagePath);

        unityWebRequest.SetRequestHeader("Content-Type", "application/octet-stream");
        unityWebRequest.SetRequestHeader("Prediction-Key", predictionKey);

        // The upload handler will help uploading the byte array with the request
        unityWebRequest.uploadHandler = new UploadHandlerRaw(imageBytes);
        unityWebRequest.uploadHandler.contentType = "application/octet-stream";

        // The download handler will help receiving the analysis from Azure
        unityWebRequest.downloadHandler = new DownloadHandlerBuffer();

        // Send the request
        yield return unityWebRequest.SendWebRequest();

        string jsonResponse = unityWebRequest.downloadHandler.text;

        Debug.Log("response: " + jsonResponse);

        // Create a texture. Texture size does not matter, since
        // LoadImage will replace with the incoming image size.
        //Texture2D tex = new Texture2D(1, 1);
        //tex.LoadImage(imageBytes);
        //SceneOrganiser.Instance.quadRenderer.material.SetTexture("_MainTex", tex);

        // The response will be in JSON format, therefore it needs to be deserialized
        //AnalysisRootObject analysisRootObject = new AnalysisRootObject();
        //analysisRootObject = JsonConvert.DeserializeObject<AnalysisRootObject>(jsonResponse);

        //SceneOrganiser.Instance.FinaliseLabel(analysisRootObject);
    }
}

/// <summary>
/// Returns the contents of the specified image file as a byte array.
/// </summary>
static byte[] GetImageAsByteArray(string imagePath)
{
    FileStream fileStream = new FileStream(imageFilePath, FileMode.Open, FileAccess.Read);

    BinaryReader binaryReader = new BinaryReader(fileStream);

    return binaryReader.ReadBytes((int)fileStream.Length);
}

```

9. Delete the **Start()** and **Update()** methods, as they will not be used.
10. Be sure to save your changes in **Visual Studio**, before returning to **Unity**.

IMPORTANT

As mentioned earlier, do not worry about code which might appear to have an error, as you will provide further classes soon, which will fix these.

Chapter 6 - Create the CustomVisionObjects class

The class you will create now is the **CustomVisionObjects** class.

This script contains a number of objects used by other classes to serialize and deserialize the calls made to the Custom Vision Service.

To create this class:

1. Right-click inside the **Scripts** folder, then click **Create > C# Script**. Call the script **CustomVisionObjects**.
2. Double-click on the new **CustomVisionObjects** script to open it with **Visual Studio**.
3. Make sure you have the following namespaces referenced at the top of the file:

```
using System;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Networking;
```

4. Delete the **Start()** and **Update()** methods inside the **CustomVisionObjects** class, this class should now be empty.

WARNING

It is important you follow the next instruction carefully. If you put the new class declarations inside the **CustomVisionObjects** class, you will get compile errors in [chapter 10](#), stating that **AnalysisRootObject** and **BoundingBox** are not found.

5. Add the following classes *outside* the **CustomVisionObjects** class. These objects are used by the **Newtonsoft** library to serialize and deserialize the response data:

```
// The objects contained in this script represent the deserialized version
// of the objects used by this application

/// <summary>
/// Web request object for image data
/// </summary>
class MultipartObject : IMultipartFormSection
{
    public string sectionName { get; set; }

    public byte[] sectionData { get; set; }

    public string fileName { get; set; }

    public string contentType { get; set; }
}

/// <summary>
/// JSON of all Tags existing within the project
/// contains the list of Tags
/// </summary>
public class Tags_RootObject
```

```

{
    public List<TagOfProject> Tags { get; set; }
    public int TotalTaggedImages { get; set; }
    public int TotalUntaggedImages { get; set; }
}

public class TagOfProject
{
    public string Id { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public int ImageCount { get; set; }
}

/// <summary>
/// JSON of Tag to associate to an image
/// Contains a list of hosting the tags,
/// since multiple tags can be associated with one image
/// </summary>
public class Tag_RootObject
{
    public List<Tag> Tags { get; set; }
}

public class Tag
{
    public string ImageId { get; set; }
    public string TagId { get; set; }
}

/// <summary>
/// JSON of images submitted
/// Contains objects that host detailed information about one or more images
/// </summary>
public class ImageRootObject
{
    public bool IsBatchSuccessful { get; set; }
    public List<SubmittedImage> Images { get; set; }
}

public class SubmittedImage
{
    public string SourceUrl { get; set; }
    public string Status { get; set; }
    public ImageObject Image { get; set; }
}

public class ImageObject
{
    public string Id { get; set; }
    public DateTime Created { get; set; }
    public int Width { get; set; }
    public int Height { get; set; }
    public string ImageUri { get; set; }
    public string ThumbnailUri { get; set; }
}

/// <summary>
/// JSON of Service Iteration
/// </summary>
public class Iteration
{
    public string Id { get; set; }
    public string Name { get; set; }
    public bool IsDefault { get; set; }
    public string Status { get; set; }
    public string Created { get; set; }
    public string LastModified { get; set; }
    public string TrainedAt { get; set; }
}

```

```

    public string ProjectId { get; set; }
    public bool Exportable { get; set; }
    public string DomainId { get; set; }
}

/// <summary>
/// Predictions received by the Service
/// after submitting an image for analysis
/// Includes Bounding Box
/// </summary>
public class AnalysisRootObject
{
    public string id { get; set; }
    public string project { get; set; }
    public string iteration { get; set; }
    public DateTime created { get; set; }
    public List<Prediction> predictions { get; set; }
}

public class BoundingBox
{
    public double left { get; set; }
    public double top { get; set; }
    public double width { get; set; }
    public double height { get; set; }
}

public class Prediction
{
    public double probability { get; set; }
    public string tagId { get; set; }
    public string tagName { get; set; }
    public BoundingBox boundingBox { get; set; }
}

```

6. Be sure to save your changes in **Visual Studio**, before returning to **Unity**.

Chapter 7 - Create the SpatialMapping class

This class will set the **Spatial Mapping Collider** in the scene so to be able to detect collisions between virtual objects and real objects.

To create this class:

1. Right-click inside the **Scripts** folder, then click **Create** > **C# Script**. Call the script **SpatialMapping**.
2. Double-click on the new **SpatialMapping** script to open it with **Visual Studio**.
3. Make sure you have the following namespaces referenced above the **SpatialMapping** class:

```

using UnityEngine;
using UnityEngine.XR.WSA;

```

4. Then, add the following variables inside the **SpatialMapping** class, above the **Start()** method:

```

/// <summary>
/// Allows this class to behave like a singleton
/// </summary>
public static SpatialMapping Instance;

/// <summary>
/// Used by the GazeCursor as a property with the Raycast call
/// </summary>
internal static int PhysicsRaycastMask;

/// <summary>
/// The layer to use for spatial mapping collisions
/// </summary>
internal int physicsLayer = 31;

/// <summary>
/// Creates environment colliders to work with physics
/// </summary>
private SpatialMappingCollider spatialMappingCollider;

```

5. Add the **Awake()** and **Start()**:

```

/// <summary>
/// Initializes this class
/// </summary>
private void Awake()
{
    // Allows this instance to behave like a singleton
    Instance = this;
}

/// <summary>
/// Runs at initialization right after Awake method
/// </summary>
void Start()
{
    // Initialize and configure the collider
    spatialMappingCollider = gameObject.GetComponent<SpatialMappingCollider>();
    spatialMappingCollider.surfaceParent = this.gameObject;
    spatialMappingCollider.freezeUpdates = false;
    spatialMappingCollider.layer = physicsLayer;

    // define the mask
    PhysicsRaycastMask = 1 << physicsLayer;

    // set the object as active one
    gameObject.SetActive(true);
}

```

6. Delete the **Update()** method.

7. Be sure to save your changes in **Visual Studio**, before returning to **Unity**.

Chapter 8 - Create the GazeCursor class

This class is responsible for setting up the cursor in the correct location in real space, by making use of the **SpatialMappingCollider**, created in the previous chapter.

To create this class:

1. Right-click inside the **Scripts** folder, then click **Create** > **C# Script**. Call the script **GazeCursor**
2. Double-click on the new **GazeCursor** script to open it with **Visual Studio**.

3. Make sure you have the following namespace referenced above the **GazeCursor** class:

```
using UnityEngine;
```

4. Then add the following variable inside the **GazeCursor** class, above the **Start()** method.

```
/// <summary>
/// The cursor (this object) mesh renderer
/// </summary>
private MeshRenderer meshRenderer;
```

5. Update the **Start()** method with the following code:

```
/// <summary>
/// Runs at initialization right after the Awake method
/// </summary>
void Start()
{
    // Grab the mesh renderer that is on the same object as this script.
    meshRenderer = gameObject.GetComponent<MeshRenderer>();

    // Set the cursor reference
    SceneOrganiser.Instance.cursor = gameObject;
    gameObject.GetComponent<Renderer>().material.color = Color.green;

    // If you wish to change the size of the cursor you can do so here
    gameObject.transform.localScale = new Vector3(0.01f, 0.01f, 0.01f);
}
```

6. Update the **Update()** method with the following code:

```
/// <summary>
/// Update is called once per frame
/// </summary>
void Update()
{
    // Do a raycast into the world based on the user's head position and orientation.
    Vector3 headPosition = Camera.main.transform.position;
    Vector3 gazeDirection = Camera.main.transform.forward;

    RaycastHit gazeHitInfo;
    if (Physics.Raycast(headPosition, gazeDirection, out gazeHitInfo, 30.0f,
    SpatialMapping.PhysicsRaycastMask))
    {
        // If the raycast hit a hologram, display the cursor mesh.
        meshRenderer.enabled = true;
        // Move the cursor to the point where the raycast hit.
        transform.position = gazeHitInfo.point;
        // Rotate the cursor to hug the surface of the hologram.
        transform.rotation = Quaternion.FromToRotation(Vector3.up, gazeHitInfo.normal);
    }
    else
    {
        // If the raycast did not hit a hologram, hide the cursor mesh.
        meshRenderer.enabled = false;
    }
}
```

NOTE

Do not worry about the error for the **SceneOrganiser** class not being found, you will create it in the next chapter.

7. Be sure to save your changes in **Visual Studio**, before returning to **Unity**.

Chapter 9 - Create the SceneOrganiser class

This class will:

- Set up the *Main Camera* by attaching the appropriate components to it.
- When an object is detected, it will be responsible for calculating its position in the real world and place a **Tag Label** near it with the appropriate **Tag Name**.

To create this class:

1. Right-click inside the **Scripts** folder, then click **Create** > **C# Script**. Name the script **SceneOrganiser**.
2. Double-click on the new **SceneOrganiser** script to open it with **Visual Studio**.
3. Make sure you have the following namespaces referenced above the **SceneOrganiser** class:

```
using System.Collections.Generic;
using System.Linq;
using UnityEngine;
```

4. Then add the following variables inside the **SceneOrganiser** class, above the **Start()** method:

```

/// <summary>
/// Allows this class to behave like a singleton
/// </summary>
public static SceneOrganiser Instance;

/// <summary>
/// The cursor object attached to the Main Camera
/// </summary>
internal GameObject cursor;

/// <summary>
/// The label used to display the analysis on the objects in the real world
/// </summary>
public GameObject label;

/// <summary>
/// Reference to the last Label positioned
/// </summary>
internal Transform lastLabelPlaced;

/// <summary>
/// Reference to the last Label positioned
/// </summary>
internal TextMesh lastLabelPlacedText;

/// <summary>
/// Current threshold accepted for displaying the label
/// Reduce this value to display the recognition more often
/// </summary>
internal float probabilityThreshold = 0.8f;

/// <summary>
/// The quad object hosting the imposed image captured
/// </summary>
private GameObject quad;

/// <summary>
/// Renderer of the quad object
/// </summary>
internal Renderer quadRenderer;

```

5. Delete the **Start()** and **Update()** methods.

6. Underneath the variables, add the **Awake()** method, which will initialize the class and set up the scene.

```

/// <summary>
/// Called on initialization
/// </summary>
private void Awake()
{
    // Use this class instance as singleton
    Instance = this;

    // Add the ImageCapture class to this Gameobject
    gameObject.AddComponent<ImageCapture>();

    // Add the CustomVisionAnalyser class to this Gameobject
    gameObject.AddComponent<CustomVisionAnalyser>();

    // Add the CustomVisionObjects class to this Gameobject
    gameObject.AddComponent<CustomVisionObjects>();
}

```

7. Add the **PlaceAnalysisLabel()** method, which will *Instantiate* the label in the scene (which at this point is

invisible to the user). It also places the quad (also invisible) where the image is placed, and overlaps with the real world. This is important because the box coordinates retrieved from the Service after analysis are traced back into this quad to determine the approximate location of the object in the real world.

```
/// <summary>
/// Instantiate a Label in the appropriate location relative to the Main Camera.
/// </summary>
public void PlaceAnalysisLabel()
{
    lastLabelPlaced = Instantiate(label.transform, cursor.transform.position, transform.rotation);
    lastLabelPlacedText = lastLabelPlaced.GetComponent<TextMesh>();
    lastLabelPlacedText.text = "";
    lastLabelPlaced.transform.localScale = new Vector3(0.005f, 0.005f, 0.005f);

    // Create a GameObject to which the texture can be applied
    quad = GameObject.CreatePrimitive(PrimitiveType.Quad);
    quadRenderer = quad.GetComponent<Renderer>() as Renderer;
    Material m = new Material(Shader.Find("Legacy Shaders/Transparent/Diffuse"));
    quadRenderer.material = m;

    // Here you can set the transparency of the quad. Useful for debugging
    float transparency = 0f;
    quadRenderer.material.color = new Color(1, 1, 1, transparency);

    // Set the position and scale of the quad depending on user position
    quad.transform.parent = transform;
    quad.transform.rotation = transform.rotation;

    // The quad is positioned slightly forward in front of the user
    quad.transform.localPosition = new Vector3(0.0f, 0.0f, 3.0f);

    // The quad scale has been set with the following value following experimentation,
    // to allow the image on the quad to be as precisely imposed to the real world as possible
    quad.transform.localScale = new Vector3(3f, 1.65f, 1f);
    quad.transform.parent = null;
}
```

8. Add the **FinaliseLabel()** method. It is responsible for:

- Setting the *Label* text with the *Tag* of the Prediction with the highest confidence.
- Calling the calculation of the *Bounding Box* on the quad object, positioned previously, and placing the label in the scene.
- Adjusting the label depth by using a Raycast towards the *Bounding Box*, which should collide against the object in the real world.
- Resetting the capture process to allow the user to capture another image.

```

/// <summary>
/// Set the Tags as Text of the last label created.
/// </summary>
public void FinaliseLabel(AnalysisRootObject analysisObject)
{
    if (analysisObject.predictions != null)
    {
        lastLabelPlacedText = lastLabelPlaced.GetComponent<TextMesh>();
        // Sort the predictions to locate the highest one
        List<Prediction> sortedPredictions = new List<Prediction>();
        sortedPredictions = analysisObject.predictions.OrderBy(p => p.probability).ToList();
        Prediction bestPrediction = new Prediction();
        bestPrediction = sortedPredictions[sortedPredictions.Count - 1];

        if (bestPrediction.probability > probabilityThreshold)
        {
            quadRenderer = quad.GetComponent<Renderer>() as Renderer;
            Bounds quadBounds = quadRenderer.bounds;

            // Position the label as close as possible to the Bounding Box of the prediction
            // At this point it will not consider depth
            lastLabelPlaced.transform.parent = quad.transform;
            lastLabelPlaced.transform.localPosition = CalculateBoundingBoxPosition(quadBounds,
bestPrediction.boundingBox);

            // Set the tag text
            lastLabelPlacedText.text = bestPrediction.tagName;

            // Cast a ray from the user's head to the currently placed label, it should hit the
            object detected by the Service.
            // At that point it will reposition the label where the ray HL sensor collides with the
            object,
            // (using the HL spatial tracking)
            Debug.Log("Repositioning Label");
            Vector3 headPosition = Camera.main.transform.position;
            RaycastHit objHitInfo;
            Vector3 objDirection = lastLabelPlaced.position;
            if (Physics.Raycast(headPosition, objDirection, out objHitInfo, 30.0f,
SpatialMapping.PhysicsRaycastMask))
            {
                lastLabelPlaced.position = objHitInfo.point;
            }
        }
        // Reset the color of the cursor
        cursor.GetComponent<Renderer>().material.color = Color.green;

        // Stop the analysis process
        ImageCapture.Instance.ResetImageCapture();
    }
}

```

9. Add the **CalculateBoundingBoxPosition()** method, which hosts a number of calculations necessary to translate the *Bounding Box* coordinates retrieved from the Service and recreate them proportionally on the quad.

```

/// <summary>
/// This method hosts a series of calculations to determine the position
/// of the Bounding Box on the quad created in the real world
/// by using the Bounding Box received back alongside the Best Prediction
/// </summary>
public Vector3 CalculateBoundingBoxPosition(Bounds b, BoundingBox boundingBox)
{
    Debug.Log($"BB: left {boundingBox.left}, top {boundingBox.top}, width {boundingBox.width},
height {boundingBox.height}");

    double centerFromLeft = boundingBox.left + (boundingBox.width / 2);
    double centerFromTop = boundingBox.top + (boundingBox.height / 2);
    Debug.Log($"BB CenterFromLeft {centerFromLeft}, CenterFromTop {centerFromTop}");

    double quadWidth = b.size.normalized.x;
    double quadHeight = b.size.normalized.y;
    Debug.Log($"Quad Width {b.size.normalized.x}, Quad Height {b.size.normalized.y}");

    double normalisedPos_X = (quadWidth * centerFromLeft) - (quadWidth/2);
    double normalisedPos_Y = (quadHeight * centerFromTop) - (quadHeight/2);

    return new Vector3((float)normalisedPos_X, (float)normalisedPos_Y, 0);
}

```

10. Be sure to save your changes in **Visual Studio**, before returning to **Unity**.

IMPORTANT

Before you continue, open the **CustomVisionAnalyser** class, and within the **AnalyseLastImageCaptured()** method, *uncomment* the following lines:

```

// Create a texture. Texture size does not matter, since
// LoadImage will replace with the incoming image size.
Texture2D tex = new Texture2D(1, 1);
tex.LoadImage(imageBytes);
SceneOrganiser.Instance.quadRenderer.material.SetTexture("_MainTex", tex);

// The response will be in JSON format, therefore it needs to be deserialized
AnalysisRootObject analysisRootObject = new AnalysisRootObject();
analysisRootObject = JsonConvert.DeserializeObject<AnalysisRootObject>(jsonResponse);

SceneOrganiser.Instance.FinaliseLabel(analysisRootObject);

```

NOTE

Do not worry about the **ImageCapture** class 'could not be found' message, you will create it in the next chapter.

Chapter 10 - Create the ImageCapture class

The next class you are going to create is the **ImageCapture** class.

This class is responsible for:

- Capturing an image using the HoloLens camera and storing it in the *App* folder.
- Handling *Tap* gestures from the user.

To create this class:

1. Go to the **Scripts** folder you created previously.

2. Right-click inside the folder, then click **Create > C# Script**. Name the script **ImageCapture**.
3. Double-click on the new **ImageCapture** script to open it with **Visual Studio**.
4. Replace the namespaces at the top of the file with the following:

```
using System;
using System.IO;
using System.Linq;
using UnityEngine;
using UnityEngine.XR.WSA.Input;
using UnityEngine.XR.WSA.WebCam;
```

5. Then add the following variables inside the **ImageCapture** class, above the **Start()** method:

```
/// <summary>
/// Allows this class to behave like a singleton
/// </summary>
public static ImageCapture Instance;

/// <summary>
/// Keep counts of the taps for image renaming
/// </summary>
private int captureCount = 0;

/// <summary>
/// Photo Capture object
/// </summary>
private PhotoCapture photoCaptureObject = null;

/// <summary>
/// Allows gestures recognition in HoloLens
/// </summary>
private GestureRecognizer recognizer;

/// <summary>
/// Flagging if the capture loop is running
/// </summary>
internal bool captureIsActive;

/// <summary>
/// File path of current analysed photo
/// </summary>
internal string filePath = string.Empty;
```

6. Code for **Awake()** and **Start()** methods now needs to be added:

```

/// <summary>
/// Called on initialization
/// </summary>
private void Awake()
{
    Instance = this;
}

/// <summary>
/// Runs at initialization right after Awake method
/// </summary>
void Start()
{
    // Clean up the LocalState folder of this application from all photos stored
    DirectoryInfo info = new DirectoryInfo(Application.persistentDataPath);
    var fileInfo = info.GetFiles();
    foreach (var file in fileInfo)
    {
        try
        {
            file.Delete();
        }
        catch (Exception)
        {
            Debug.LogFormat("Cannot delete file: ", file.Name);
        }
    }

    // Subscribing to the Microsoft HoloLens API gesture recognizer to track user gestures
    recognizer = new GestureRecognizer();
    recognizer.SetRecognizableGestures(GestureSettings.Tap);
    recognizer.Tapped += TapHandler;
    recognizer.StartCapturingGestures();
}

```

7. Implement a handler that will be called when a Tap gesture occurs:

```

/// <summary>
/// Respond to Tap Input.
/// </summary>
private void TapHandler(TappedEventArgs obj)
{
    if (!captureIsActive)
    {
        captureIsActive = true;

        // Set the cursor color to red
        SceneOrganiser.Instance.cursor.GetComponent<Renderer>().material.color = Color.red;

        // Begin the capture loop
        Invoke("ExecuteImageCaptureAndAnalysis", 0);
    }
}

```

IMPORTANT

When the cursor is **green**, it means the camera is available to take the image. When the cursor is **red**, it means the camera is busy.

8. Add the method that the application uses to start the image capture process and store the image:

```

/// <summary>
/// Begin process of image capturing and send to Azure Custom Vision Service.
/// </summary>
private void ExecuteImageCaptureAndAnalysis()
{
    // Create a label in world space using the ResultsLabel class
    // Invisible at this point but correctly positioned where the image was taken
    SceneOrganiser.Instance.PlaceAnalysisLabel();

    // Set the camera resolution to be the highest possible
    Resolution cameraResolution = PhotoCapture.SupportedResolutions.OrderByDescending
        ((res) => res.width * res.height).First();
    Texture2D targetTexture = new Texture2D(cameraResolution.width, cameraResolution.height);

    // Begin capture process, set the image format
    PhotoCapture.CreateAsync(true, delegate (PhotoCapture captureObject)
    {
        photoCaptureObject = captureObject;

        CameraParameters camParameters = new CameraParameters
        {
            hologramOpacity = 1.0f,
            cameraResolutionWidth = targetTexture.width,
            cameraResolutionHeight = targetTexture.height,
            pixelFormat = CapturePixelFormat.BGRA32
        };

        // Capture the image from the camera and save it in the App internal folder
        captureObject.StartPhotoModeAsync(camParameters, delegate (PhotoCapture.PhotoCaptureResult
result)
        {
            string filename = string.Format(@"CapturedImage{0}.jpg", captureCount);
            filePath = Path.Combine(Application.persistentDataPath, filename);
            captureCount++;
            photoCaptureObject.TakePhotoAsync(filePath, PhotoCaptureFileOutputFormat.JPG,
OnCapturedPhotoToDisk);
        });
    });
}

```

9. Add the handlers that will be called when the photo has been captured and for when it is ready to be analyzed. The result is then passed to the **CustomVisionAnalyser** for analysis.

```

/// <summary>
/// Register the full execution of the Photo Capture.
/// </summary>
void OnCapturedPhotoToDisk(PhotoCapture.PhotoCaptureResult result)
{
    try
    {
        // Call StopPhotoMode once the image has successfully captured
        photoCaptureObject.StopPhotoModeAsync(OnStoppedPhotoMode);
    }
    catch (Exception e)
    {
        Debug.LogFormat("Exception capturing photo to disk: {0}", e.Message);
    }
}

/// <summary>
/// The camera photo mode has stopped after the capture.
/// Begin the image analysis process.
/// </summary>
void OnStoppedPhotoMode(PhotoCapture.PhotoCaptureResult result)
{
    Debug.LogFormat("Stopped Photo Mode");

    // Dispose from the object in memory and request the image analysis
    photoCaptureObject.Dispose();
    photoCaptureObject = null;

    // Call the image analysis
    StartCoroutine(CustomVisionAnalyser.Instance.AnalyseLastImageCaptured(filePath));
}

/// <summary>
/// Stops all capture pending actions
/// </summary>
internal void ResetImageCapture()
{
    captureIsActive = false;

    // Set the cursor color to green
    SceneOrganiser.Instance.cursor.GetComponent<Renderer>().material.color = Color.green;

    // Stop the capture loop if active
    CancelInvoke();
}

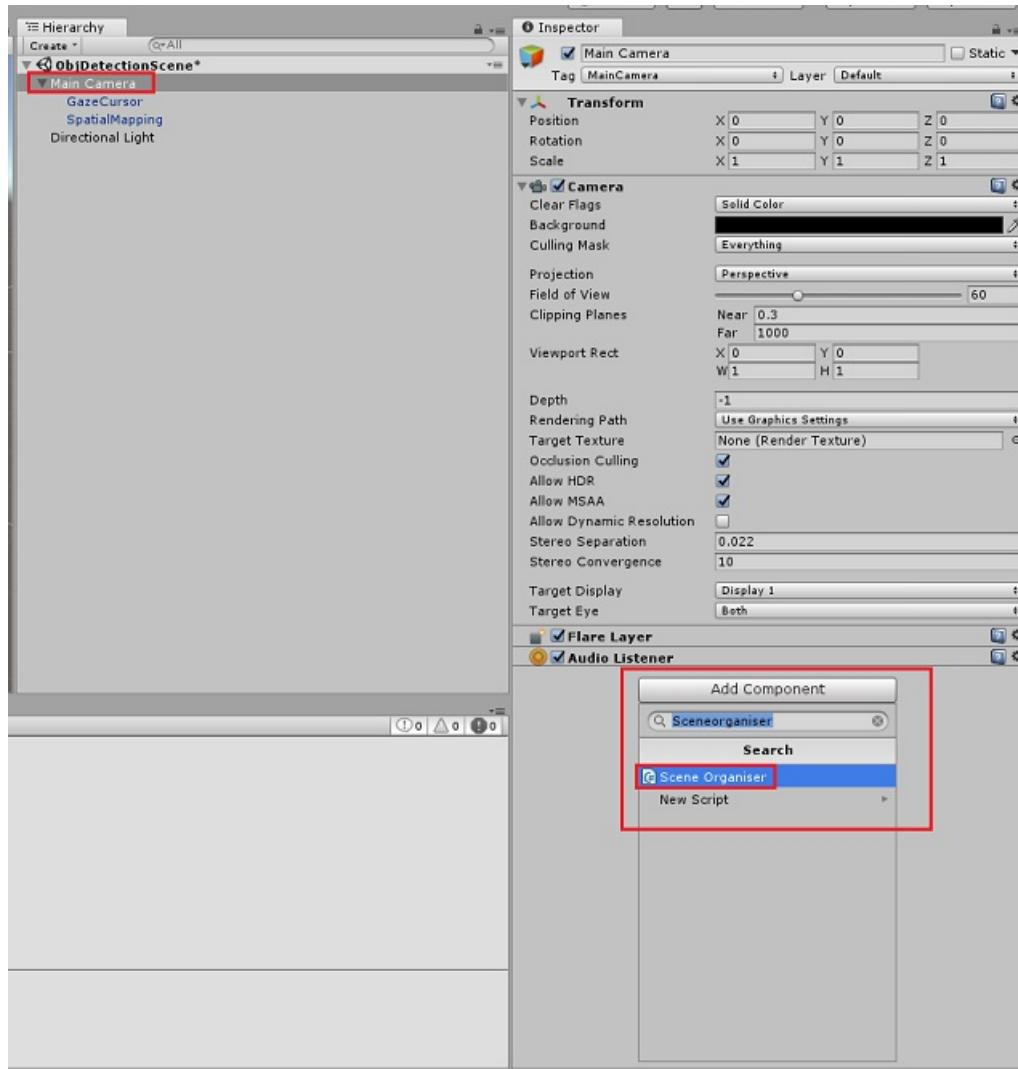
```

10. Be sure to save your changes in **Visual Studio**, before returning to **Unity**.

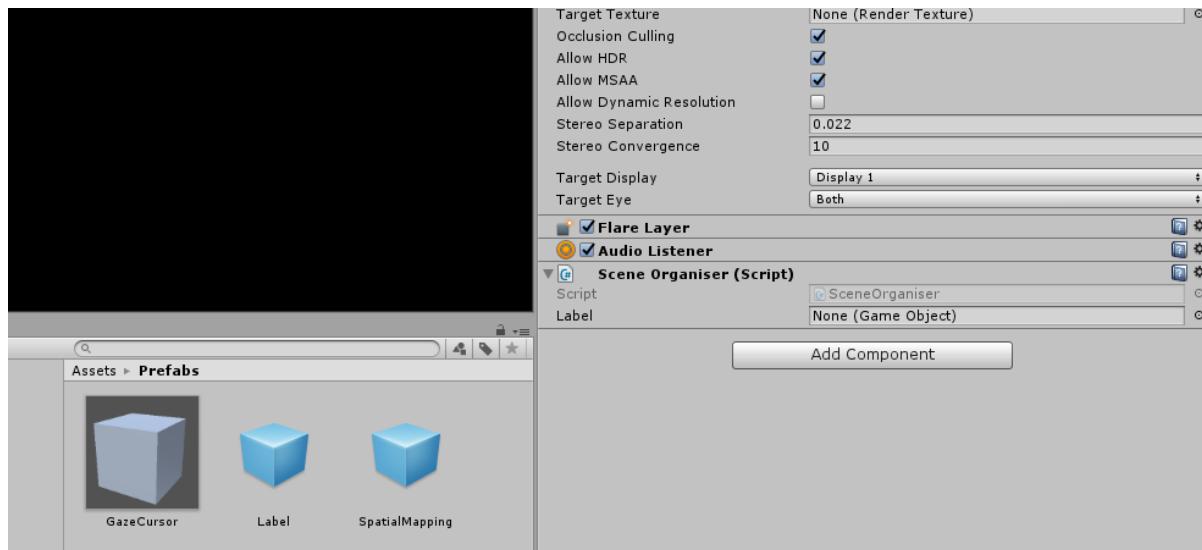
Chapter 11 - Setting up the scripts in the scene

Now that you have written all of the code necessary for this project, is time to set up the scripts in the scene, and on the prefabs, for them to behave correctly.

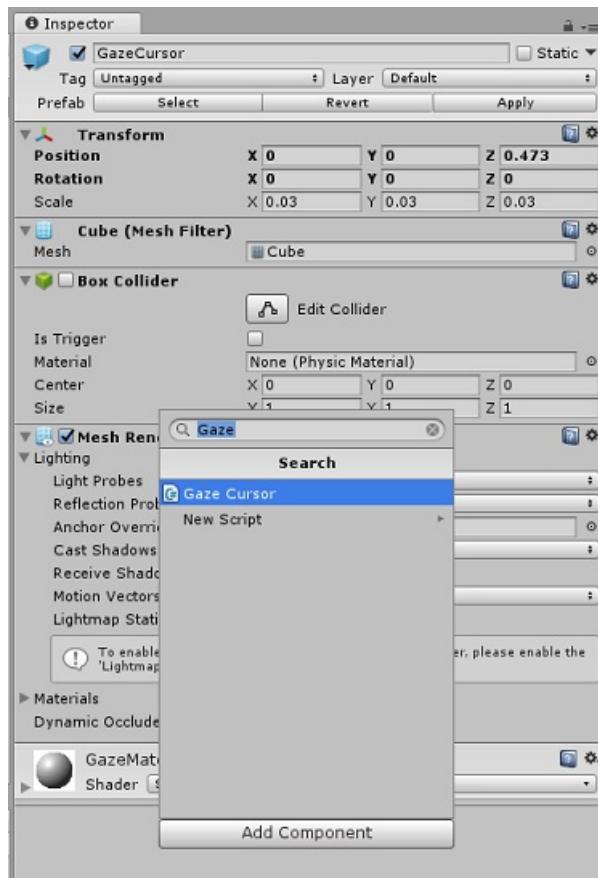
1. Within the **Unity Editor**, in the **Hierarchy Panel**, select the **Main Camera**.
2. In the **Inspector Panel**, with the **Main Camera** selected, click on **Add Component**, then search for **SceneOrganiser** script and double-click, to add it.



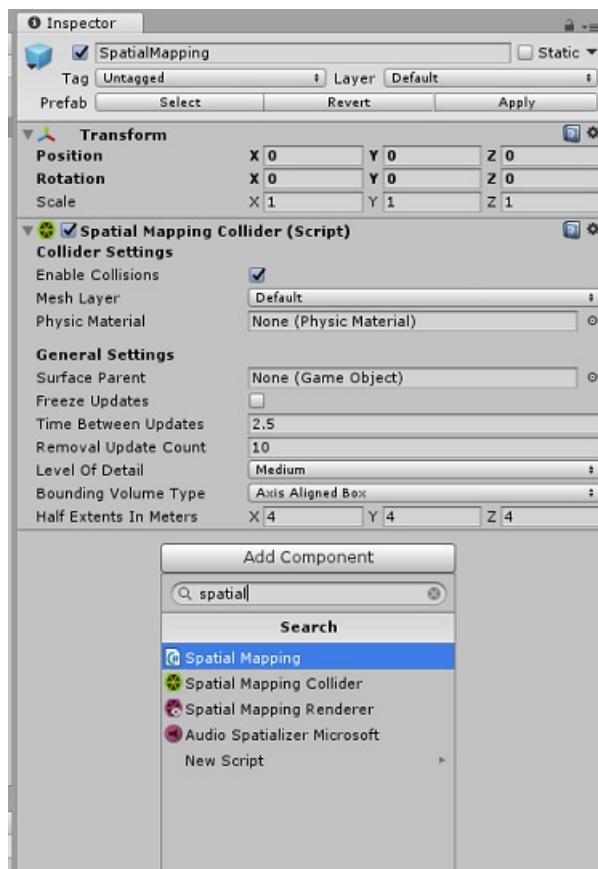
3. In the **Project Panel**, open the **Prefabs folder**, drag the **Label** prefab into the *Label* empty reference target input area, in the **SceneOrganiser** script that you have just added to the *Main Camera*, as shown in the image below:



4. In the **Hierarchy Panel**, select the **GazeCursor** child of the **Main Camera**.
5. In the **Inspector Panel**, with the **GazeCursor** selected, click on **Add Component**, then search for **GazeCursor** script and double-click, to add it.



6. Again, in the **Hierarchy Panel**, select the **SpatialMapping** child of the **Main Camera**.
7. In the **Inspector Panel**, with the **SpatialMapping** selected, click on **Add Component**, then search for **SpatialMapping** script and double-click, to add it.



The remaining scripts that you have not set will be added by the code in the **SceneOrganiser** script, during runtime.

Chapter 12 - Before building

To perform a thorough test of your application you will need to sideload it onto your Microsoft HoloLens.

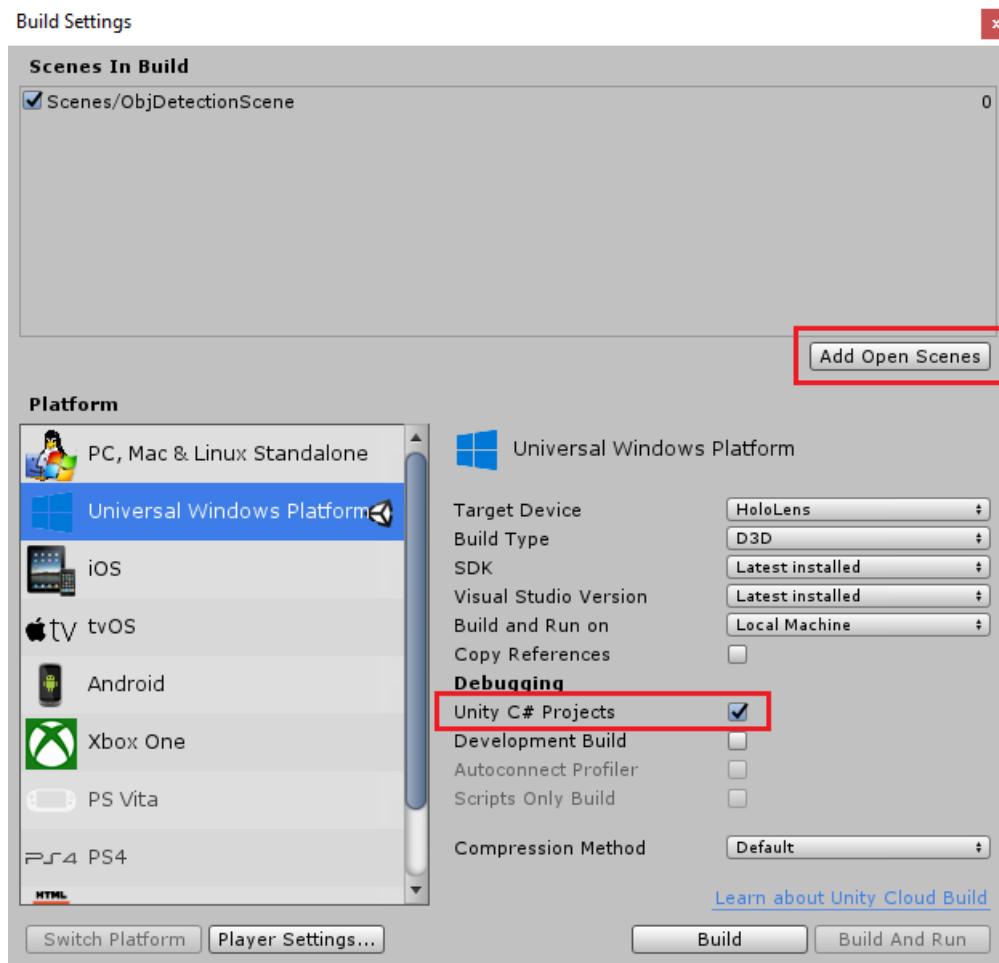
Before you do, ensure that:

- All the settings mentioned in the [Chapter 3](#) are set correctly.
- The script **SceneOrganiser** is attached to the **Main Camera** object.
- The script **GazeCursor** is attached to the **GazeCursor** object.
- The script **SpatialMapping** is attached to the **SpatialMapping** object.
- In [Chapter 5](#), Step 6:
 - Make sure you insert your **Service Prediction Key** into the **predictionKey** variable.
 - You have inserted your **Prediction Endpoint** into the **predictionEndpoint** class.

Chapter 13 - Build the UWP solution and sideload your application

You are now ready to build your application as a UWP Solution that you will be able to deploy on to the Microsoft HoloLens. To begin the build process:

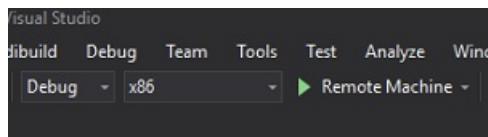
1. Go to **File > Build Settings**.
2. Tick **Unity C# Projects**.
3. Click on **Add Open Scenes**. This will add the currently open scene to the build.



4. Click **Build**. Unity will launch a *File Explorer* window, where you need to create and then select a folder to build the app into. Create that folder now, and name it **App**. Then with the **App** folder selected, click **Select**

Folder.

5. Unity will begin building your project to the **App** folder.
6. Once Unity has finished building (it might take some time), it will open a **File Explorer** window at the location of your build (check your task bar, as it may not always appear above your windows, but will notify you of the addition of a new window).
7. To deploy on to Microsoft HoloLens, you will need the IP Address of that device (for Remote Deploy), and to ensure that it also has **Developer Mode** set. To do this:
 - a. Whilst wearing your HoloLens, open the **Settings**.
 - b. Go to **Network & Internet > Wi-Fi > Advanced Options**
 - c. Note the **IPv4** address.
 - d. Next, navigate back to **Settings**, and then to **Update & Security > For Developers**
 - e. Set **Developer Mode On**.
8. Navigate to your new Unity build (the **App** folder) and open the solution file with **Visual Studio**.
9. In the Solution Configuration select **Debug**.
10. In the Solution Platform, select **x86, Remote Machine**. You will be prompted to insert the **IP address** of a remote device (the Microsoft HoloLens, in this case, which you noted).



11. Go to the **Build** menu and click on **Deploy Solution** to sideload the application to your HoloLens.
12. Your app should now appear in the list of installed apps on your Microsoft HoloLens, ready to be launched!

To use the application:

- Look at an object, which you have trained with your **Azure Custom Vision Service, Object Detection**, and use the **Tap gesture**.
- If the object is successfully detected, a world-space *Label Text* will appear with the tag name.

IMPORTANT

Every time you capture a photo and send it to the Service, you can go back to the Service page and retrain the Service with the newly captured images. At the beginning, you will probably also have to correct the *Bounding Boxes* to be more accurate and retrain the Service.

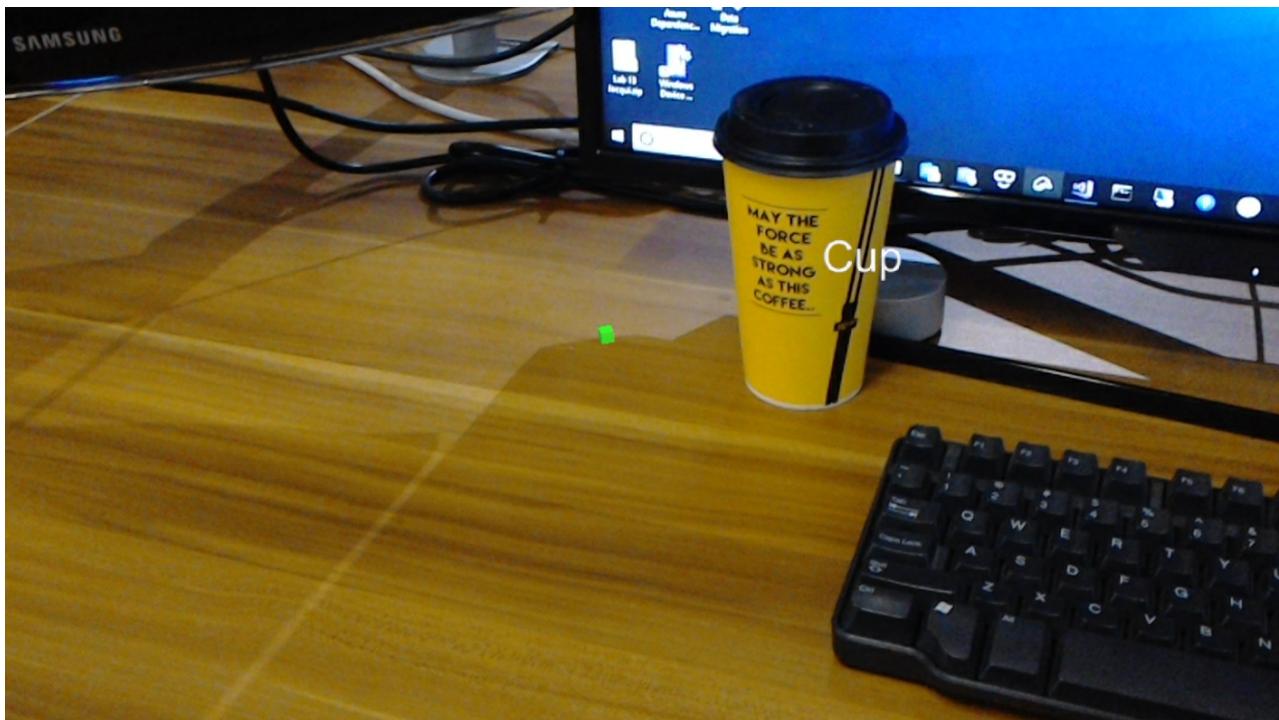
NOTE

The Label Text placed might not appear near the object when the Microsoft HoloLens sensors and/or the SpatialTrackingComponent in Unity fails to place the appropriate colliders, relative to the real world objects. Try to use the application on a different surface if that is the case.

Your Custom Vision, Object Detection application

Congratulations, you built a mixed reality app that leverages the Azure Custom Vision, Object Detection API, which can recognize an object from an image, and then provide an approximate position for that object in 3D

space.



Bonus exercises

Exercise 1

Adding to the Text Label, use a semi-transparent cube to wrap the real object in a 3D *Bounding Box*.

Exercise 2

Train your Custom Vision Service to recognize more objects.

Exercise 3

Play a sound when an object is recognized.

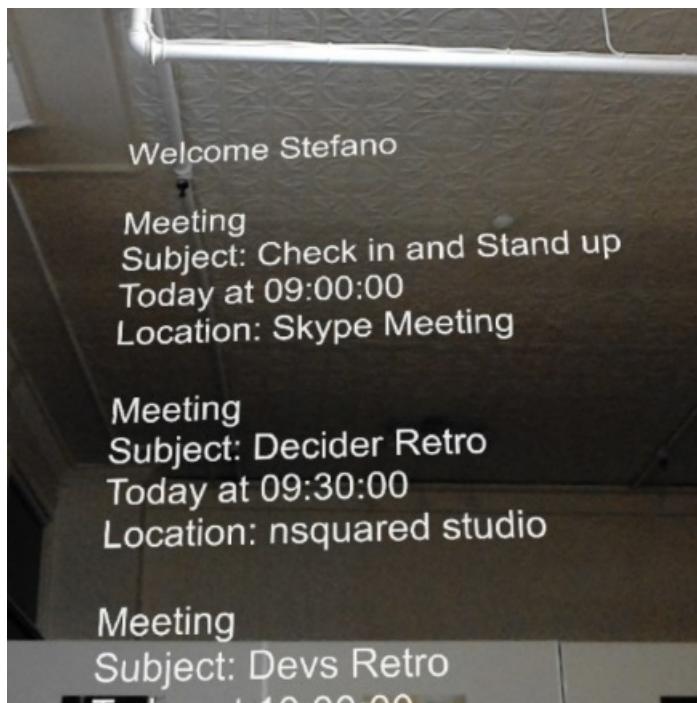
Exercise 4

Use the API to re-train your Service with the same images your app is analyzing, so to make the Service more accurate (do both prediction and training simultaneously).

MR and Azure 311 - Microsoft Graph

11/6/2018 • 22 minutes to read • [Edit Online](#)

In this course, you will learn how to use *Microsoft Graph* to log in into your Microsoft account using secure authentication within a mixed reality application. You will then retrieve and display your scheduled meetings in the application interface.



Microsoft Graph is a set of APIs designed to enable access to many of Microsoft's services. Microsoft describes Microsoft Graph as being a matrix of resources connected by relationships, meaning it allows an application to access all sorts of connected user data. For more information, visit the [Microsoft Graph page](#).

Development will include the creation of an app where the user will be instructed to gaze at and then tap a sphere, which will prompt the user to log in safely to a Microsoft account. Once logged in to their account, the user will be able to see a list of meetings scheduled for the day.

Having completed this course, you will have a mixed reality HoloLens application, which will be able to do the following:

1. Using the Tap gesture, tap on an object, which will prompt the user to log into a Microsoft Account (moving out of the app to log in, and then back into the app again).
2. View a list of meetings scheduled for the day.

In your application, it is up to you as to how you will integrate the results with your design. This course is designed to teach you how to integrate an Azure Service with your Unity project. It is your job to use the knowledge you gain from this course to enhance your mixed reality application.

Device support

COURSE	HOLOLENS	IMMERSIVE HEADSETS
MR and Azure 311: Microsoft Graph	✓	

Prerequisites

NOTE

This tutorial is designed for developers who have basic experience with Unity and C#. Please also be aware that the prerequisites and written instructions within this document represent what has been tested and verified at the time of writing (July 2018). You are free to use the latest software, as listed within the [install the tools](#) article, though it should not be assumed that the information in this course will perfectly match what you will find in newer software than what is listed below.

We recommend the following hardware and software for this course:

- A development PC
- [Windows 10 Fall Creators Update \(or later\) with Developer mode enabled](#)
- [The latest Windows 10 SDK](#)
- [Unity 2017.4](#)
- [Visual Studio 2017](#)
- A [Microsoft HoloLens](#) with Developer mode enabled
- Internet access for Azure setup and Microsoft Graph data retrieval
- A valid **Microsoft Account** (either personal or work/school)
- A few meetings scheduled for the current day, using the same Microsoft Account

Before you start

1. To avoid encountering issues building this project, it is strongly suggested that you create the project mentioned in this tutorial in a root or near-root folder (long folder paths can cause issues at build-time).
2. Set up and test your HoloLens. If you need support setting up your HoloLens, [make sure to visit the HoloLens setup article](#).
3. It is a good idea to perform Calibration and Sensor Tuning when beginning developing a new HoloLens App (sometimes it can help to perform those tasks for each user).

For help on Calibration, please follow this [link to the HoloLens Calibration article](#).

For help on Sensor Tuning, please follow this [link to the HoloLens Sensor Tuning article](#).

Chapter 1 - Create your app in the Application Registration Portal

To begin with, you will need to create and register your application in the **Application Registration Portal**.

In this Chapter you will also find the Service Key that will allow you to make calls to *Microsoft Graph* to access your account content.

1. Navigate to the [Microsoft Application Registration Portal](#) and login with your Microsoft Account. Once you have logged in, you will be redirected to the **Application Registration Portal**.
2. In the **My applications** section, click on the button **Add an app**.

My applications

Converged applications [Learn More](#)

Name	App ID / Client Id	Action
[REDACTED]	[REDACTED]	Delete
[REDACTED].app	[REDACTED]	Delete
[REDACTED].app	[REDACTED]	Delete

Add an app

Azure AD only applications [Learn More](#)

Name	App ID / Client Id	Action
[REDACTED]	[REDACTED]	Delete

Add an app

English

Contact us Terms of use Privacy stat

My applications [Learn More](#)

Add an app

Name	App ID / Client Id
Press the "Add an App" button to create a new application	

IMPORTANT

The **Application Registration Portal** can look different, depending on whether you have previously worked with *Microsoft Graph*. The below screenshots display these different versions.

3. Add a name for your application and click **Create**.

Register your application

Application Name

Guided Setup

Let us help you get started

By proceeding, you agree to the [Microsoft Platform Policies](#)

[Create](#)

4. Once the application has been created, you will be redirected to the application main page. Copy the **Application Id** and make sure to note this value somewhere safe, you will use it soon in your code.

My applications / MyNewGraphApplication

MyNewGraphApplication Registration

[Click here for help integrating your application with Microsoft.](#)

This application will be registered in the Azure Active Directory instance used to manage your Stefano@nsquaredsolutions.com account.

Properties

Name

MyNewGraphApplication

Application Id

[REDACTED]

Application Secrets

[Generate New Password](#) [Generate New Key Pair](#) [Upload Public Key](#)

Platforms

[Add Platform](#)

- In the **Platforms** section, make sure **Native Application** is displayed. If *not* click on **Add Platform** and select **Native Application**.

Platforms

[Add Platform](#)

Web

[Delete](#)

Allow Implicit Flow

Redirect URLs [Add URI](#)

http://localhost

Logout URL [Add URI](#)

e.g. http://www.contoso.com/logout

Native Application

[Delete](#)

Custom Redirect URIs [Add URI](#)

[REDACTED]3c7://auth

Built-in redirect URIs [Add URI](#)

- Scroll down in the same page and in the section called **Microsoft Graph Permissions** you will need to add additional permissions for the application. Click on **Add** next to **Delegated Permissions**.

Microsoft Graph Permissions

The settings you set here may vary depending on whether you get a token from our V1 or V2 endpoint. What's the difference?

Delegated Permissions [Add](#) [About delegated permissions](#)

User.Read [X](#)

Application Permissions [Add](#) [About application permissions](#)

- Since you want your application to access the user's Calendar, check the box called **Calendars.Read** and click **OK**.

Select Permission

Scope	User can consent <small>?</small>	Admin can consent <small>?</small>	Description
<input type="checkbox"/> Bookings.Manage.All	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> Yes	▼
<input type="checkbox"/> Bookings.Read.All	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> Yes	▼
<input type="checkbox"/> Bookings.ReadWrite.All	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> Yes	▼
<input type="checkbox"/> BookingsAppointment.ReadWrite.All	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> Yes	▼
<input checked="" type="checkbox"/> Calendars.Read	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> Yes	▼
<input type="checkbox"/> Calendars.Read.Shared	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> Yes	▼
<input type="checkbox"/> Calendars.ReadWrite	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> Yes	▼
<input type="checkbox"/> Calendars.ReadWrite.Shared	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> Yes	▼

2 Selected

Ok

Cancel

8. Scroll to the bottom and click the **Save** button.

Advanced Options

Live SDK support

Edit Application Manifest Cannot edit the manifest until all other changes to the application are saved.

Delete Application

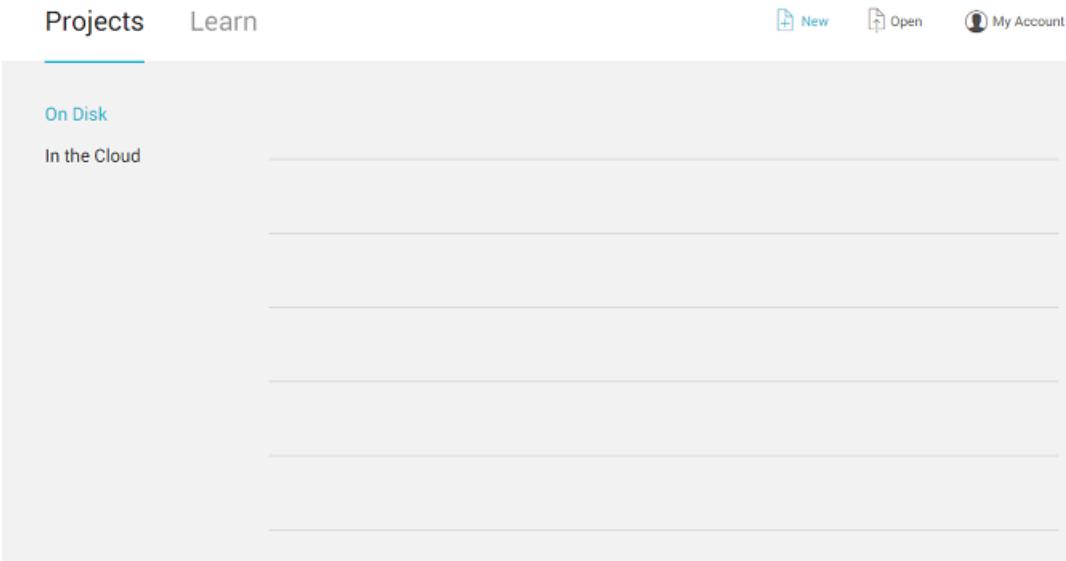
Save Discard Changes There are unsaved changes.

9. Your save will be confirmed, and you can log out from the **Application Registration Portal**.

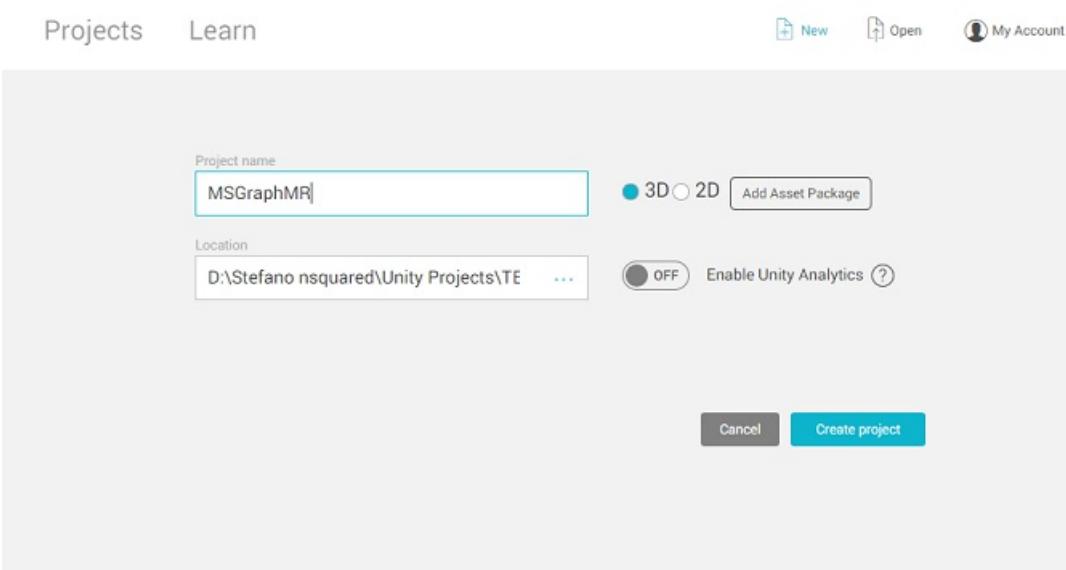
Chapter 2 - Set up the Unity project

The following is a typical set up for developing with mixed reality, and as such, is a good template for other projects.

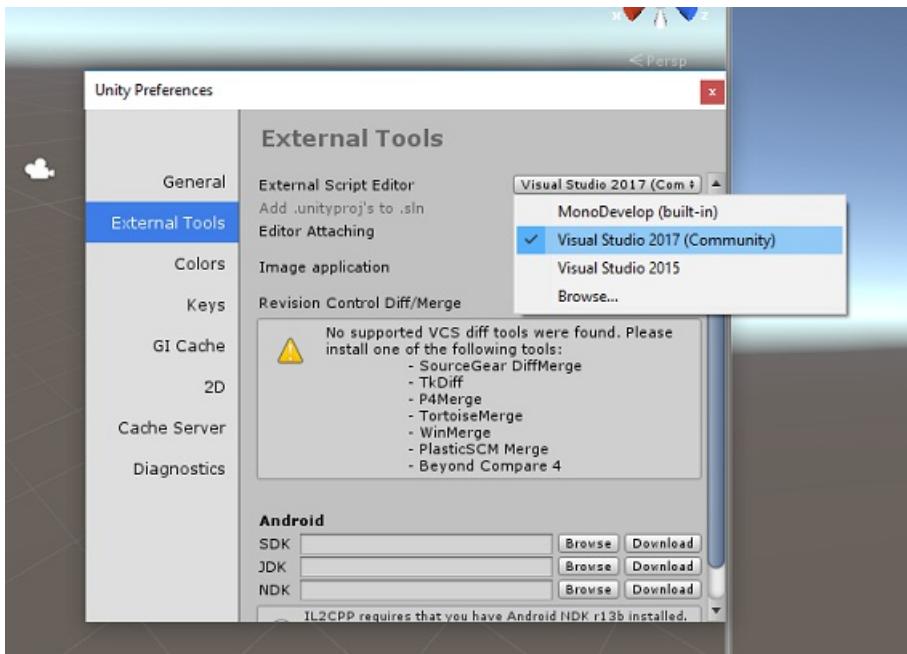
1. Open *Unity* and click **New**.



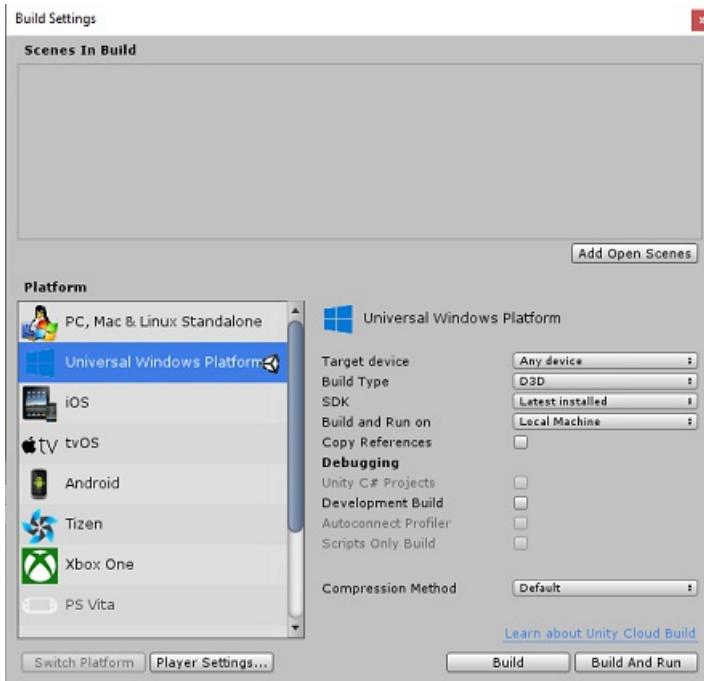
2. You need to provide a Unity project name. Insert **MSGraphMR**. Make sure the project template is set to **3D**. Set the **Location** to somewhere appropriate for you (remember, closer to root directories is better). Then, click **Create project**.



3. With Unity open, it is worth checking the default **Script Editor** is set to **Visual Studio**. Go to **Edit > Preferences** and then from the new window, navigate to **External Tools**. Change **External Script Editor** to **Visual Studio 2017**. Close the **Preferences** window.

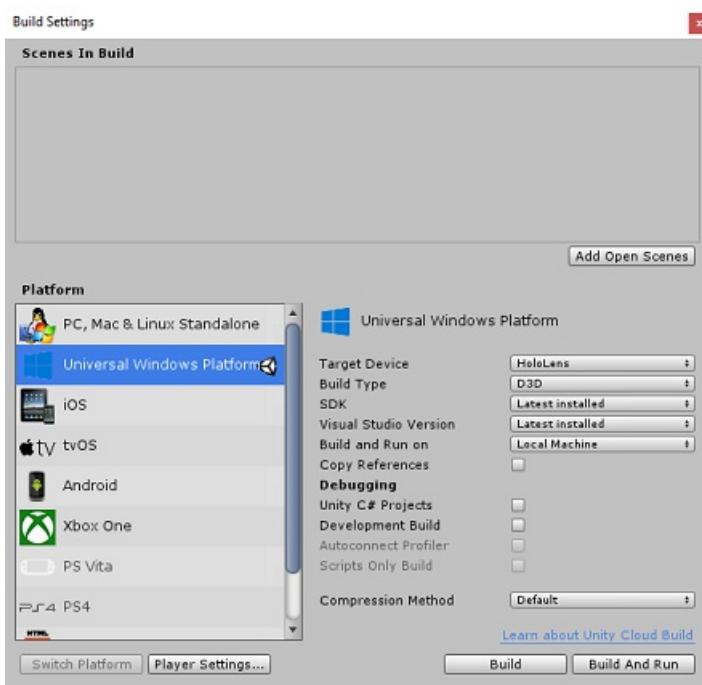


4. Go to **File > Build Settings** and select **Universal Windows Platform**, then click on the **Switch Platform** button to apply your selection.

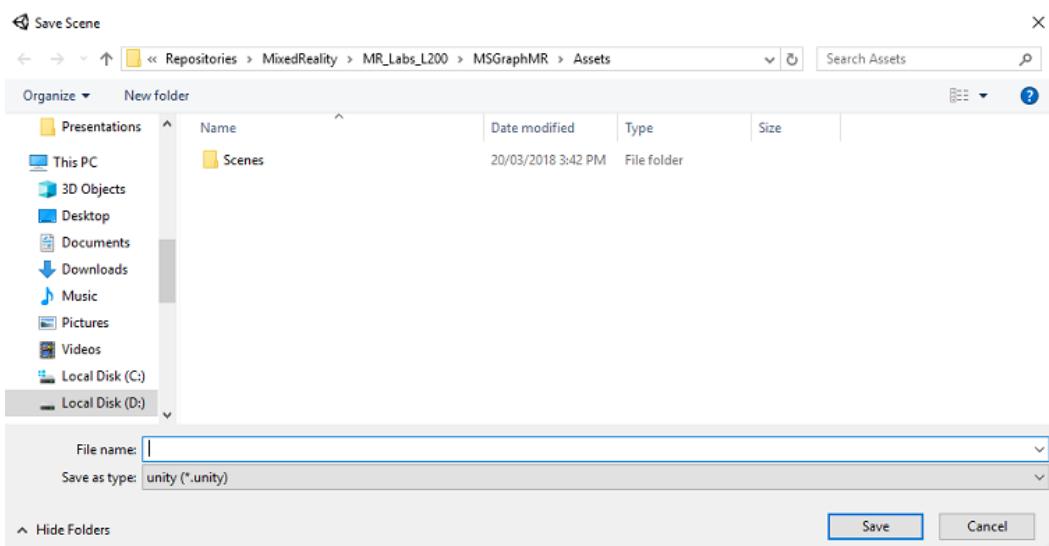


5. While still in **File > Build Settings**, make sure that:

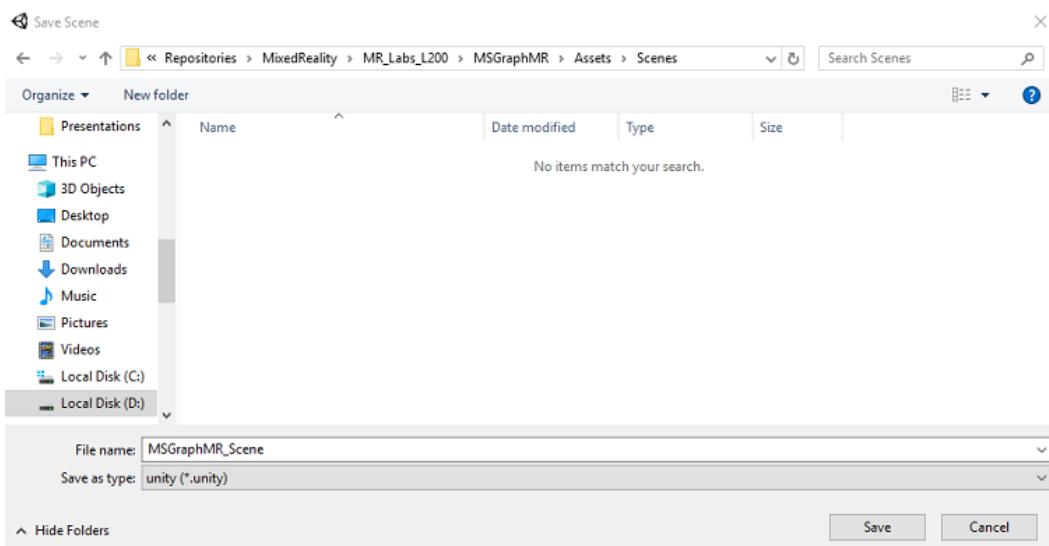
- Target Device** is set to **HoloLens**
- Build Type** is set to **D3D**
- SDK** is set to **Latest installed**
- Visual Studio Version** is set to **Latest installed**
- Build and Run** is set to **Local Machine**
- Save the scene and add it to the build.
 - Do this by selecting **Add Open Scenes**. A save window will appear.



- b. Create a new folder for this, and any future, scene. Select the **New folder** button, to create a new folder, name it **Scenes**.



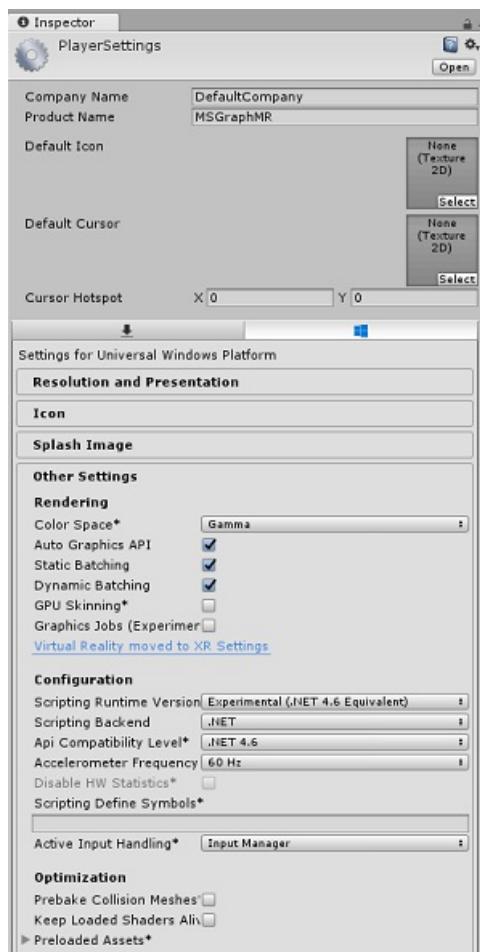
- c. Open your newly created **Scenes** folder, and then in the *File name:* text field, type **MR_ComputerVisionScene**, then click **Save**.



IMPORTANT

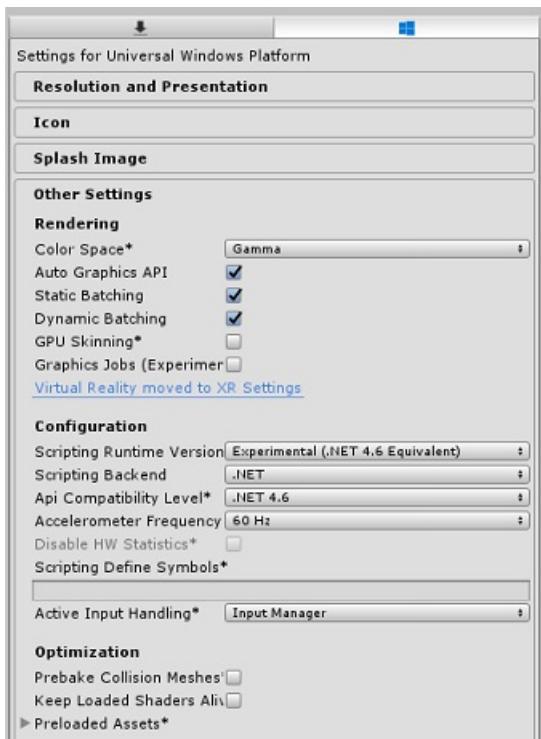
Be aware, you must save your Unity scenes within the *Assets* folder, as they must be associated with the Unity project. Creating the scenes folder (and other similar folders) is a typical way of structuring a Unity project.

- g. The remaining settings, in *Build Settings*, should be left as default for now.
6. In the *Build Settings* window, click on the **Player Settings** button, this will open the related panel in the space where the *Inspector* is located.



7. In this panel, a few settings need to be verified:

- In the **Other Settings** tab:
 - Scripting Runtime Version** should be **Experimental** (.NET 4.6 Equivalent), which will trigger a need to restart the Editor.
 - Scripting Backend** should be **.NET**
 - API Compatibility Level** should be **.NET 4.6**

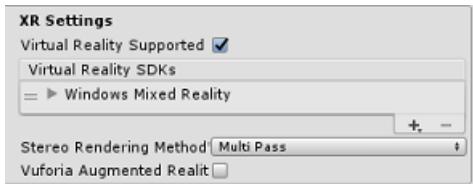


b. Within the **Publishing Settings** tab, under **Capabilities**, check:

- **InternetClient**



c. Further down the panel, in **XR Settings** (found below **Publish Settings**), check **Virtual Reality Supported**, make sure the **Windows Mixed Reality SDK** is added.



8. Back in *Build Settings, Unity C# Projects* is no longer greyed out; check the checkbox next to this.

9. Close the *Build Settings* window.

10. Save your scene and project (**FILE > SAVE SCENES / FILE > SAVE PROJECT**).

Chapter 3 - Import Libraries in Unity

IMPORTANT

If you wish to skip the *Unity Set up* component of this course, and continue straight into code, feel free to download this [Azure-MR-311.unitypackage](#), import it into your project as a **Custom Package**, and then continue from [Chapter 5](#).

To use *Microsoft Graph* within Unity you need to make use of the **Microsoft.Identity.Client** DLL. It is possible to use the Microsoft Graph SDK, however, it will require the addition of a NuGet package after you build the Unity project (meaning editing the project post-build). It is considered simpler to import the required DLLs directly into Unity.

NOTE

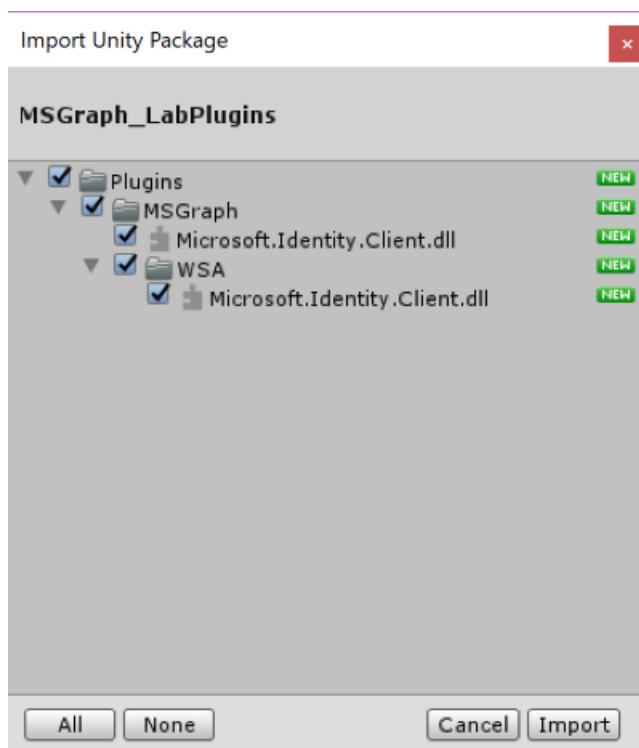
There is currently a known issue in Unity which requires plugins to be reconfigured after import. These steps (4 - 7 in this section) will no longer be required after the bug has been resolved.

To import *Microsoft Graph* into your own project, [download the MSGraph_LabPlugins.zip file](#). This package has been created with versions of the libraries that have been tested.

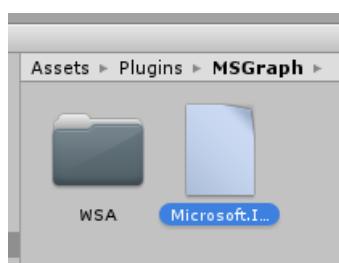
If you wish to know more about how to add custom DLLs to your Unity project, [follow this link](#).

To import the package:

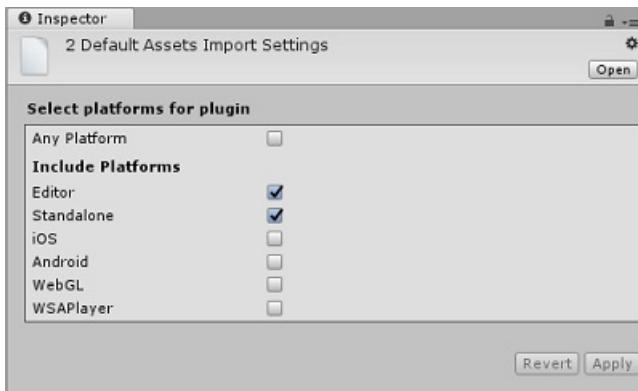
1. Add the Unity Package to Unity by using the *Assets > Import Package > Custom Package* menu option.
Select the package you just downloaded.
2. In the **Import Unity Package** box that pops up, ensure everything under (and including) **Plugins** is selected.



3. Click the **Import** button to add the items to your project.
4. Go to the **MSGraph** folder under **Plugins** in the *Project Panel* and select the plugin called **Microsoft.Identity.Client**.



5. With the *plugin* selected, ensure that **Any Platform** is unchecked, then ensure that **WSAPlayer** is also unchecked, then click **Apply**. This is just to confirm that the files are configured correctly.

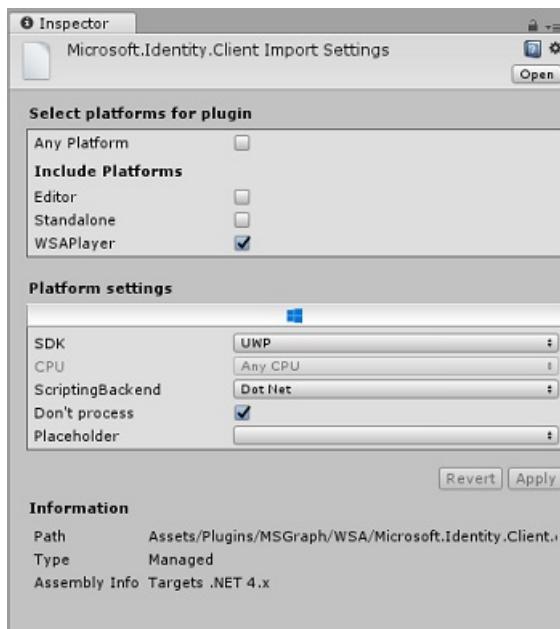


NOTE

Marking these plugins configures them to only be used in the Unity Editor. There are a different set of DLLs in the WSA folder which will be used after the project is exported from Unity as a Universal Windows Application.

6. Next, you need to open the **WSA** folder, within the **MSGraph** folder. You will see a copy of the same file you just configured. Select the file, and then in the inspector:

- ensure that **Any Platform** is **unchecked**, and that **only WSAPlayer** is **checked**.
- Ensure **SDK** is set to **UWP**, and **Scripting Backend** is set to **Dot Net**
- Ensure that **Don't process** is **checked**.



7. Click **Apply**.

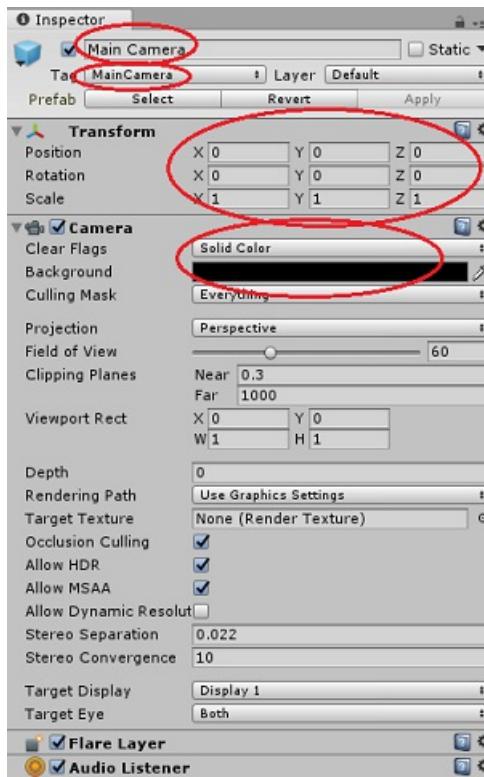
Chapter 4 - Camera Setup

During this Chapter you will set up the Main Camera of your scene:

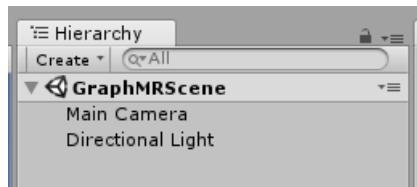
1. In the *Hierarchy Panel*, select the **Main Camera**.
2. Once selected, you will be able to see all the components of the **Main Camera** in the *Inspector* panel.
 - a. The **Camera object** must be named **Main Camera** (note the spelling!)
 - b. The Main Camera **Tag** must be set to **MainCamera** (note the spelling!)
 - c. Make sure the **Transform Position** is set to **0, 0, 0**

d. Set **Clear Flags** to **Solid Color**

e. Set the **Background Color** of the Camera Component to **Black, Alpha 0 (Hex Code: #00000000)**



3. The final object structure in the *Hierarchy Panel* should be like the one shown in the image below:

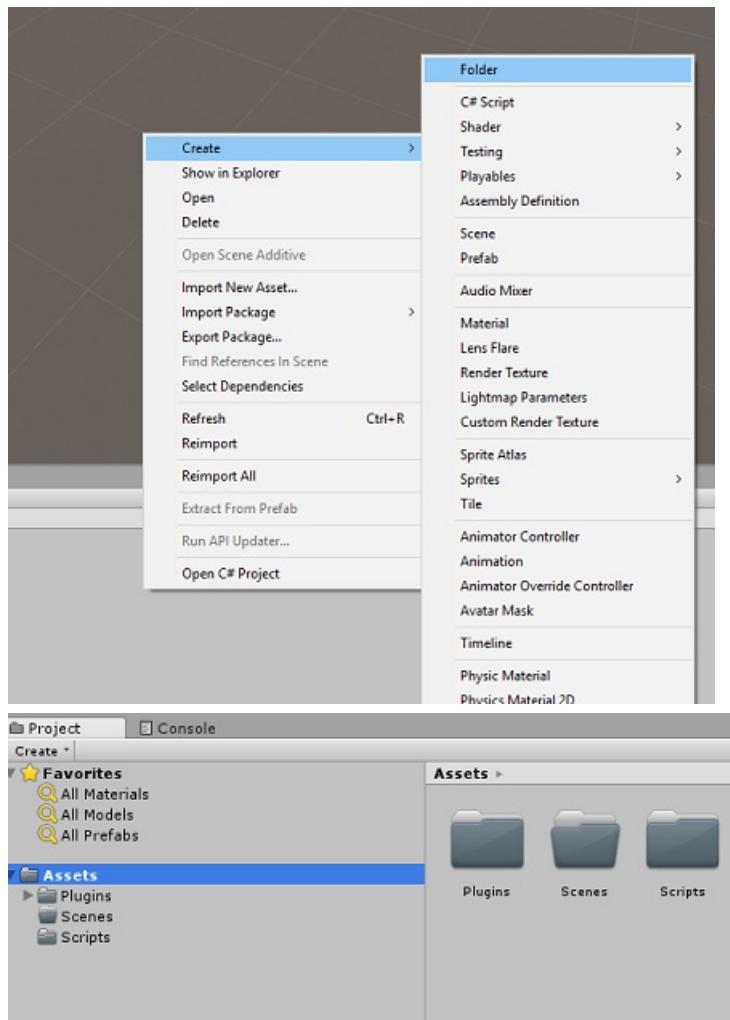


Chapter 5 - Create MeetingsUI class

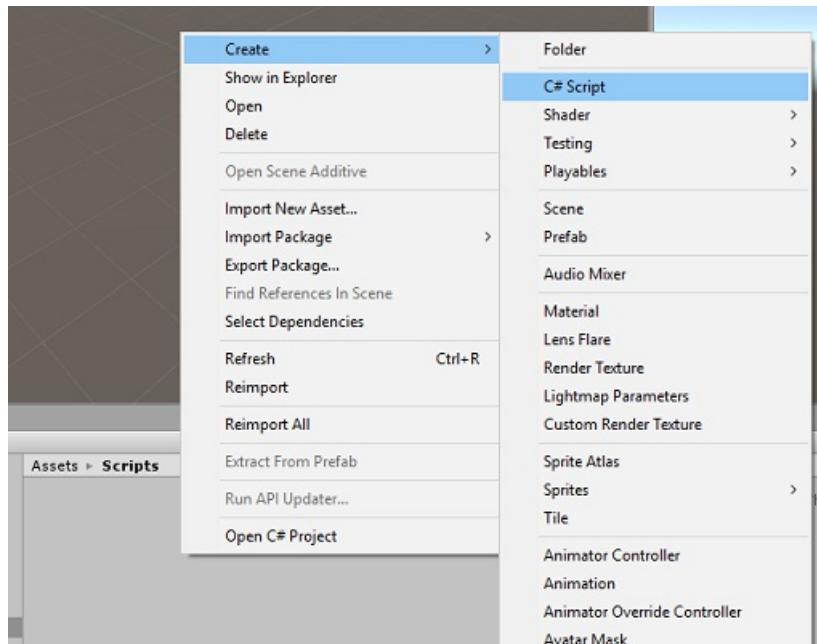
The first script you need to create is **MeetingsUI**, which is responsible for hosting and populating the UI of the application (welcome message, instructions and the meetings details).

To create this class:

1. Right-click on the **Assets** folder in the *Project Panel*, then select *Create > Folder*. Name the folder **Scripts**.



2. Open the **Scripts** folder then, within that folder, right-click, *Create > C# Script*. Name the script **MeetingsUI**.



3. Double-click on the new **MeetingsUI** script to open it with *Visual Studio*.
4. Insert the following namespaces:

```
using System;
using UnityEngine;
```

5. Inside the class insert the following variables:

```
/// <summary>
/// Allows this class to behave like a singleton
/// </summary>
public static MeetingsUI Instance;

/// <summary>
/// The 3D text of the scene
/// </summary>
private TextMesh _meetingDisplayTextMesh;
```

6. Then replace the **Start()** method and add an **Awake()** method. These will be called when the class initializes:

```
/// <summary>
/// Called on initialization
/// </summary>
void Awake()
{
    Instance = this;
}

/// <summary>
/// Called on initialization, after Awake
/// </summary>
void Start ()
{
    // Creating the text mesh within the scene
    _meetingDisplayTextMesh = CreateMeetingsDisplay();
}
```

7. Add the methods responsible for creating the *Meetings UI* and populate it with the current meetings when requested:

```

/// <summary>
/// Set the welcome message for the user
/// </summary>
internal void WelcomeUser(string userName)
{
    if(!string.IsNullOrEmpty(userName))
    {
        _meetingDisplayTextMesh.text = $"Welcome {userName}";
    }
    else
    {
        _meetingDisplayTextMesh.text = "Welcome";
    }
}

/// <summary>
/// Set up the parameters for the UI text
/// </summary>
/// <returns>Returns the 3D text in the scene</returns>
private TextMesh CreateMeetingsDisplay()
{
    GameObject display = new GameObject();
    display.transform.localScale = new Vector3(0.03f, 0.03f, 0.03f);
    display.transform.position = new Vector3(-3.5f, 2f, 9f);
    TextMesh textMesh = display.AddComponent<TextMesh>();
    textMesh.anchor = TextAnchor.MiddleLeft;
    textMesh.alignment = TextAlignment.Left;
    textMesh.fontSize = 80;
    textMesh.text = "Welcome! \nPlease gaze at the button" +
        "\nand use the Tap Gesture to display your meetings";

    return textMesh;
}

/// <summary>
/// Adds a new Meeting in the UI by chaining the existing UI text
/// </summary>
internal void AddMeeting(string subject, DateTime dateTime, string location)
{
    string newText = $"\\n{_meetingDisplayTextMesh.text}\\n\\n Meeting,\\nSubject: {subject},\\nToday at
{dateTime},\\nLocation: {location}";

    _meetingDisplayTextMesh.text = newText;
}

```

8. **Delete the `Update()` method, and save your changes** in Visual Studio before returning to Unity.

Chapter 6 - Create the Graph class

The next script to create is the **Graph** script. This script is responsible for making the calls to authenticate the user and retrieve the scheduled meetings for the current day from the user's calendar.

To create this class:

1. Double-click on the **Scripts** folder, to open it.
2. Right-click inside the **Scripts** folder, click **Create > C# Script**. Name the script **Graph**.
3. Double-click on the script to open it with Visual Studio.
4. Insert the following namespaces:

```
using System.Collections.Generic;
using UnityEngine;
using Microsoft.Identity.Client;
using System;
using System.Threading.Tasks;

#if !UNITY_EDITOR && UNITY_WSA
using System.Net.Http;
using System.Net.Http.Headers;
using Windows.Storage;
#endif
```

IMPORTANT

You will notice that parts of the code in this script are wrapped around [Precompile Directives](#), this is to avoid issues with the libraries when building the Visual Studio Solution.

5. Delete the **Start()** and **Update()** methods, as they will not be used.
6. Outside the **Graph** class, insert the following objects, which are necessary to deserialize the JSON object representing the daily scheduled meetings:

```

/// <summary>
/// The object hosting the scheduled meetings
/// </summary>
[Serializable]
public class Rootobject
{
    public List<Value> value;
}

[Serializable]
public class Value
{
    public string subject { get; set; }
    public StartTime start { get; set; }
    public Location location { get; set; }
}

[Serializable]
public class StartTime
{
    public string dateTime;

    private DateTime? _startDateTime;
    public DateTime StartDateTime
    {
        get
        {
            if (_startDateTime != null)
                return _startDateTime.Value;
            DateTime dt;
            DateTime.TryParse(dateTime, out dt);
            _startDateTime = dt;
            return _startDateTime.Value;
        }
    }
}

[Serializable]
public class Location
{
    public string displayName { get; set; }
}

```

7. Inside the **Graph** class, add the following variables:

```
/// <summary>
/// Insert your Application Id here
/// </summary>
private string _appId = "-- Insert your Application Id here --";

/// <summary>
/// Application scopes, determine Microsoft Graph accessibility level to user account
/// </summary>
private IEnumerable<string> _scopes = new List<string>() { "User.Read", "Calendars.Read" };

/// <summary>
/// Microsoft Graph API, user reference
/// </summary>
private PublicClientApplication _client;

/// <summary>
/// Microsoft Graph API, authentication
/// </summary>
private AuthenticationResult _authResult;
```

NOTE

Change the **appId** value to be the **App Id** that you have noted in [Chapter 1, step 4](#). This value should be the same as that displayed in the **Application Registration Portal**, in your application registration page.

8. Within the **Graph** class, add the methods **SignInAsync()** and **AquireTokenAsync()**, that will prompt the user to insert the log-in credentials.

```

/// <summary>
/// Begin the Sign In process using Microsoft Graph Library
/// </summary>
internal async void SignInAsync()
{
#if !UNITY_EDITOR && UNITY_WSA
    // Set up Grap user settings, determine if needs auth
    ApplicationDataContainer localSettings = ApplicationData.Current.LocalSettings;
    string userId = localSettings.Values["UserId"] as string;
    _client = new PublicClientApplication(_appId);

    // Attempt authentication
    _authResult = await AcquireTokenAsync(_client, _scopes, userId);

    // If authentication is successfull, retrieve the meetings
    if (!string.IsNullOrEmpty(_authResult.AccessToken))
    {
        // Once Auth as been completed, find the meetings for the day
        await ListMeetingsAsync(_authResult.AccessToken);
    }
#endif
}

/// <summary>
/// Attempt to retrieve the Access Token by either retrieving
/// previously stored credentials or by prompting user to Login
/// </summary>
private async Task<AuthenticationResult> AcquireTokenAsync(
    IPublicClientApplication app, IEnumerable<string> scopes, string userId)
{
    IUser user = !string.IsNullOrEmpty(userId) ? app.GetUser(userId) : null;
    string userName = user != null ? user.Name : "null";

    // Once the User name is found, display it as a welcome message
    MeetingsUI.Instance.WelcomeUser(userName);

    // Attempt to Log In the user with a pre-stored token. Only happens
    // in case the user Logged In with this app on this device previously
    try
    {
        _authResult = await app.AcquireTokenSilentAsync(scopes, user);
    }
    catch (MsalUiRequiredException)
    {
        // Pre-stored token not found, prompt the user to log-in
        try
        {
            _authResult = await app.AcquireTokenAsync(scopes);
        }
        catch (MsalException msalex)
        {
            Debug.Log($"Error Acquiring Token: {msalex.Message}");
            return _authResult;
        }
    }
}

MeetingsUI.Instance.WelcomeUser(_authResult.User.Name);

#if !UNITY_EDITOR && UNITY_WSA
    ApplicationData.Current.LocalSettings.Values["UserId"] =
    _authResult.User.Identifier;
#endif
    return _authResult;
}

```

9. Add the following two methods:

- a. **BuildTodayCalendarEndpoint()**, which builds the URI specifying the day, and time span, in which the scheduled meetings are retrieved.
- b. **ListMeetingsAsync()**, which requests the scheduled meetings from *Microsoft Graph*.

```

/// <summary>
/// Build the endpoint to retrieve the meetings for the current day.
/// </summary>
/// <returns>Returns the Calendar Endpoint</returns>
public string BuildTodayCalendarEndpoint()
{
    DateTime startOfTheDay = DateTime.Today.AddDays(0);
    DateTime endOfTheDay = DateTime.Today.AddDays(1);
    DateTime startOfTheDayUTC = startOfTheDay.ToUniversalTime();
    DateTime endOfTheDayUTC = endOfTheDay.ToUniversalTime();

    string todayDate = startOfTheDayUTC.ToString("o");
    string tomorrowDate = endOfTheDayUTC.ToString("o");
    string todayCalendarEndpoint = string.Format(
        "https://graph.microsoft.com/v1.0/me/calendarview?startdatetime={0}&enddatetime={1}",
        todayDate,
        tomorrowDate);

    return todayCalendarEndpoint;
}

/// <summary>
/// Request all the scheduled meetings for the current day.
/// </summary>
private async Task ListMeetingsAsync(string accessToken)
{
#if !UNITY_EDITOR && UNITY_WSA
    var http = new HttpClient();

    http.DefaultRequestHeaders.Authorization =
        new AuthenticationHeaderValue("Bearer", accessToken);
    var response = await http.GetAsync(BuildTodayCalendarEndpoint());

    var jsonResponse = await response.Content.ReadAsStringAsync();

    Rootobject rootObject = new Rootobject();
    try
    {
        // Parse the JSON response.
        rootObject = JsonUtility.FromJson<Rootobject>(jsonResponse);

        // Sort the meeting list by starting time.
        rootObject.value.Sort((x, y) => DateTime.Compare(x.start.StartDateTime,
y.start.StartDateTime));

        // Populate the UI with the meetings.
        for (int i = 0; i < rootObject.value.Count; i++)
        {
            MeetingsUI.Instance.AddMeeting(rootObject.value[i].subject,
                rootObject.value[i].start.StartDateTime.ToLocalTime(),
                rootObject.value[i].location.displayName);
        }
    }
    catch (Exception ex)
    {
        Debug.Log($"Error = {ex.Message}");
        return;
    }
#endif
}

```

10. You have now completed the **Graph** script. **Save your changes** in Visual Studio before returning to Unity.

Chapter 7 - Create the GazeInput script

You will now create the **GazeInput**. This class handles and keeps track of the user's gaze, using a **Raycast** coming from the **Main Camera**, projecting forward.

To create the script:

1. Double-click on the **Scripts** folder, to open it.
2. Right-click inside the **Scripts** folder, click **Create > C# Script**. Name the script **GazeInput**.
3. Double-click on the script to open it with Visual Studio.
4. Change the namespaces code to match the one below, along with adding the '**[System.Serializable]**' tag above your **GazeInput** class, so that it can be serialized:

```
using UnityEngine;

/// <summary>
/// Class responsible for the User's Gaze interactions
/// </summary>
[System.Serializable]
public class GazeInput : MonoBehaviour
{
```

5. Inside the **GazeInput** class, add the following variables:

```
[Tooltip("Used to compare whether an object is to be interacted with.")]
internal string InteractableTag = "SignInButton";

/// <summary>
/// Length of the gaze
/// </summary>
internal float GazeMaxDistance = 300;

/// <summary>
/// Object currently gazed
/// </summary>
internal GameObject FocusedObject { get; private set; }

internal GameObject oldFocusedObject { get; private set; }

internal RaycastHit HitInfo { get; private set; }

/// <summary>
/// Cursor object visible in the scene
/// </summary>
internal GameObject Cursor { get; private set; }

internal bool Hit { get; private set; }

internal Vector3 Position { get; private set; }

internal Vector3 Normal { get; private set; }

private Vector3 _gazeOrigin;

private Vector3 _gazeDirection;
```

6. Add the **CreateCursor()** method to create the HoloLens cursor in the scene, and call the method from the

Start() method:

```
/// <summary>
/// Start method used upon initialisation.
/// </summary>
internal virtual void Start()
{
    FocusedObject = null;
    Cursor = CreateCursor();
}

/// <summary>
/// Method to create a cursor object.
/// </summary>
internal GameObject CreateCursor()
{
    GameObject newCursor = GameObject.CreatePrimitive(PrimitiveType.Sphere);
    newCursor.SetActive(false);
    // Remove the collider, so it doesn't block raycast.
    Destroy(newCursor.GetComponent<SphereCollider>());
    newCursor.transform.localScale = new Vector3(0.05f, 0.05f, 0.05f);
    Material mat = new Material(Shader.Find("Diffuse"));
    newCursor.GetComponent<MeshRenderer>().material = mat;
    mat.color = Color.HSVToRGB(0.0223f, 0.7922f, 1.000f);
    newCursor.SetActive(true);

    return newCursor;
}
```

7. The following methods enable the gaze Raycast and keep track of the focused objects.

```
/// <summary>
/// Called every frame
/// </summary>
internal virtual void Update()
{
    _gazeOrigin = Camera.main.transform.position;

    _gazeDirection = Camera.main.transform.forward;

    UpdateRaycast();
}

/// <summary>
/// Reset the old focused object, stop the gaze timer, and send data if it
/// is greater than one.
/// </summary>
private void ResetFocusedObject()
{
    // Ensure the old focused object is not null.
    if (oldFocusedObject != null)
    {
        if (oldFocusedObject.CompareTag(InteractableTag))
        {
            // Provide the 'Gaze Exited' event.
            oldFocusedObject.SendMessage("OnGazeExited", SendMessageOptions.DontRequireReceiver);
        }
    }
}
```

```

private void UpdateRaycast()
{
    // Set the old focused gameobject.
    oldFocusedObject = FocusedObject;
    RaycastHit hitInfo;

    // Initialise Raycasting.
    Hit = Physics.Raycast(_gazeOrigin,
        _gazeDirection,
        out hitInfo,
        GazeMaxDistance);
    HitInfo = hitInfo;

    // Check whether raycast has hit.
    if (Hit == true)
    {
        Position = hitInfo.point;
        Normal = hitInfo.normal;

        // Check whether the hit has a collider.
        if (hitInfo.collider != null)
        {
            // Set the focused object with what the user just looked at.
            FocusedObject = hitInfo.collider.gameObject;
        }
        else
        {
            // Object looked on is not valid, set focused gameobject to null.
            FocusedObject = null;
        }
    }
    else
    {
        // No object looked upon, set focused gameobject to null.
        FocusedObject = null;

        // Provide default position for cursor.
        Position = _gazeOrigin + (_gazeDirection * GazeMaxDistance);

        // Provide a default normal.
        Normal = _gazeDirection;
    }

    // Lerp the cursor to the given position, which helps to stabilize the gaze.
    Cursor.transform.position = Vector3.Lerp(Cursor.transform.position, Position, 0.6f);

    // Check whether the previous focused object is this same. If so, reset the focused object.
    if (FocusedObject != oldFocusedObject)
    {
        ResetFocusedObject();
        if (FocusedObject != null)
        {
            if (FocusedObject.CompareTag(InteractableTag))
            {
                // Provide the 'Gaze Entered' event.
                FocusedObject.SendMessage("OnGazeEntered",
                    SendMessageOptions.DontRequireReceiver);
            }
        }
    }
}

```

8. **Save your changes** in Visual Studio before returning to Unity.

Chapter 8 - Create the Interactions class

You will now need to create the **Interactions** script, which is responsible for:

- Handling the **Tap** interaction and the **Camera Gaze**, which enables the user to interact with the log in "button" in the scene.
- Creating the log in "button" object in the scene for the user to interact with.

To create the script:

1. Double-click on the **Scripts** folder, to open it.
2. Right-click inside the **Scripts** folder, click **Create > C# Script**. Name the script **Interactions**.
3. Double-click on the script to open it with Visual Studio.
4. Insert the following namespaces:

```
using UnityEngine;
using UnityEngine.XR.WSA.Input;
```

5. Change the inheritance of the **Interaction** class from **MonoBehaviour** to **GazeInput**.

```
public class Interactions : MonoBehaviour
```

```
    public class Interactions : GazeInput
```

6. Inside the **Interaction** class insert the following variable:

```
/// <summary>
/// Allows input recognition with the HoloLens
/// </summary>
private GestureRecognizer _gestureRecognizer;
```

7. Replace the **Start** method; notice it is an override method, which calls the 'base' Gaze class method. **Start()** will be called when the class initializes, registering for input recognition and creating the sign in *button* in the scene:

```
/// <summary>
/// Called on initialization, after Awake
/// </summary>
internal override void Start()
{
    base.Start();

    // Register the application to recognize HoloLens user inputs
    _gestureRecognizer = new GestureRecognizer();
    _gestureRecognizer.SetRecognizableGestures(GestureSettings.Tap);
    _gestureRecognizer.Tapped += GestureRecognizer_Tapped;
    _gestureRecognizer.StartCapturingGestures();

    // Add the Graph script to this object
    gameObject.AddComponent<MeetingsUI>();
    CreateSignInButton();
}
```

8. Add the **CreateSignInButton()** method, which will instantiate the sign in *button* in the scene and set its properties:

```

/// <summary>
/// Create the sign in button object in the scene
/// and sets its properties
/// </summary>
void CreateSignInButton()
{
    GameObject signInButton = GameObject.CreatePrimitive(PrimitiveType.Sphere);

    Material mat = new Material(Shader.Find("Diffuse"));
    signInButton.GetComponent<Renderer>().material = mat;
    mat.color = Color.blue;

    signInButton.transform.position = new Vector3(3.5f, 2f, 9f);
    signInButton.tag = "SignInButton";
    signInButton.AddComponent<Graph>();
}

```

9. Add the **GestureRecognizer_Tapped()** method, which will respond for the *Tap* user event.

```

/// <summary>
/// Detects the User Tap Input
/// </summary>
private void GestureRecognizer_Tapped(TappedEventArgs obj)
{
    if(base.FocusedObject != null)
    {
        Debug.Log($"TAP on {base.FocusedObject.name}");
        base.FocusedObject.SendMessage("SignInAsync", SendMessageOptions.RequireReceiver);
    }
}

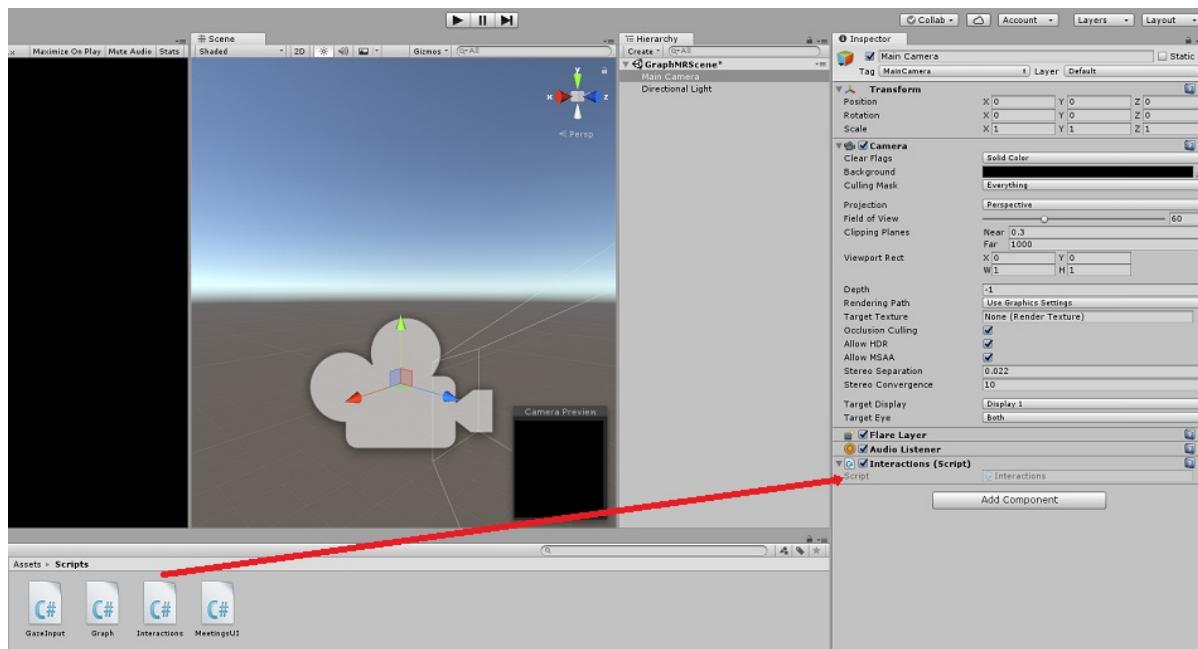
```

10. **Delete** the **Update()** method, and then **save your changes** in Visual Studio before returning to Unity.

Chapter 9 - Set up the script references

In this Chapter you need to place the **Interactions** script onto the **Main Camera**. That script will then handle placing the other scripts where they need to be.

- From the **Scripts** folder in the *Project Panel*, drag the script **Interactions** to the **Main Camera** object, as pictured below.

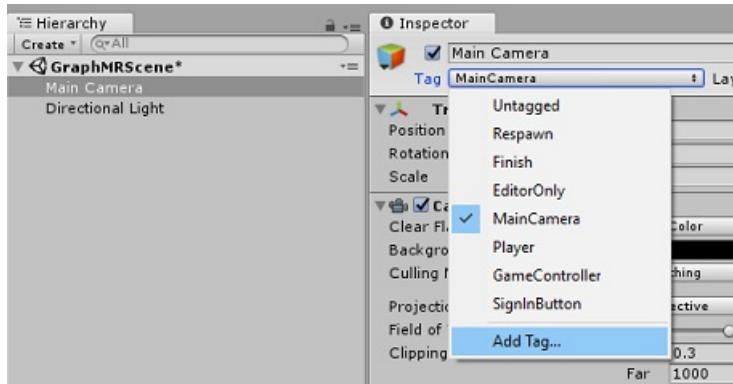


Chapter 10 - Setting up the Tag

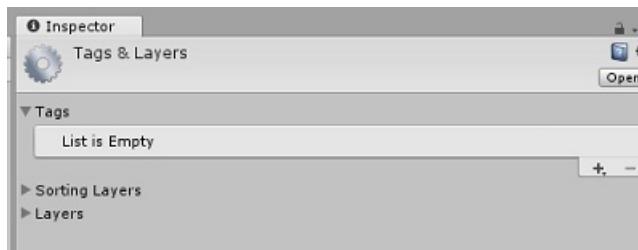
The code handling the gaze will make use of the Tag **SignInButton** to identify which object the user will interact with to sign-in to *Microsoft Graph*.

To create the Tag:

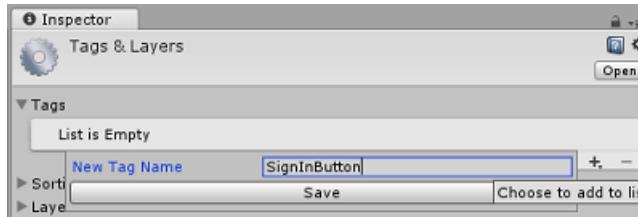
1. In the Unity Editor click on the **Main Camera** in the *Hierarchy Panel*.
2. In the *Inspector Panel* click on the **MainCamera Tag** to open a drop-down list. Click on **Add Tag...**



3. Click on the + button.



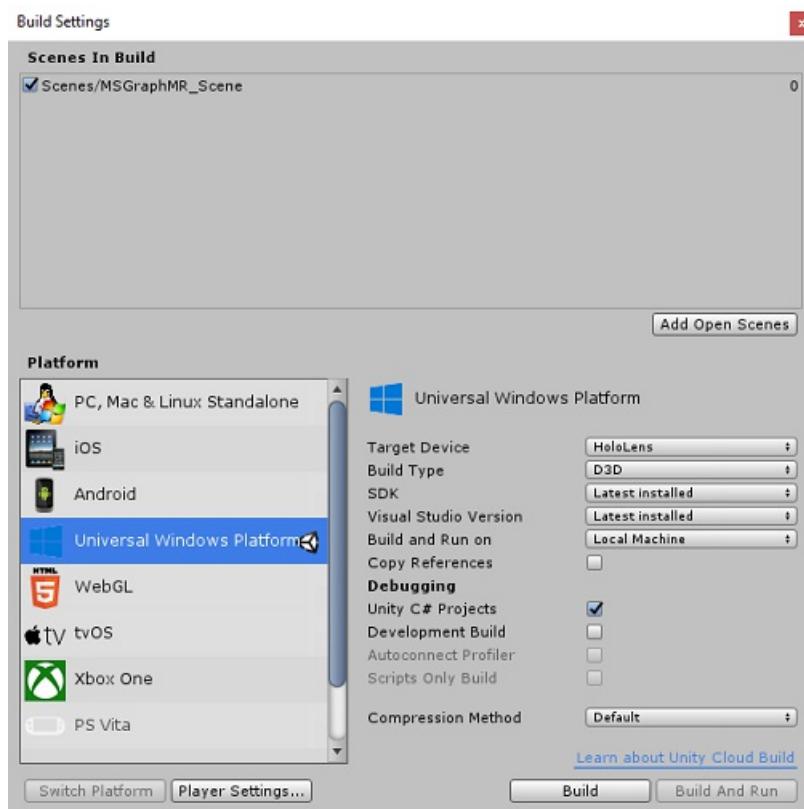
4. Write the Tag name as **SignInButton** and click Save.



Chapter 11 - Build the Unity project to UWP

Everything needed for the Unity section of this project has now been completed, so it is time to build it from Unity.

1. Navigate to *Build Settings* (*File > Build Settings*).

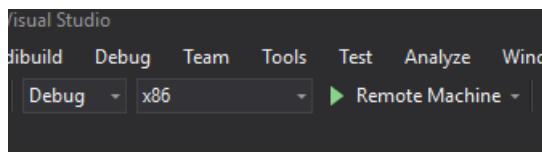


2. If not already, tick **Unity C# Projects**.
3. Click **Build**. Unity will launch a **File Explorer** window, where you need to create and then select a folder to build the app into. Create that folder now, and name it **App**. Then with the **App** folder selected, click **Select Folder**.
4. Unity will begin building your project to the **App** folder.
5. Once Unity has finished building (it might take some time), it will open a **File Explorer** window at the location of your build (check your task bar, as it may not always appear above your windows, but will notify you of the addition of a new window).

Chapter 12 - Deploy to HoloLens

To deploy on HoloLens:

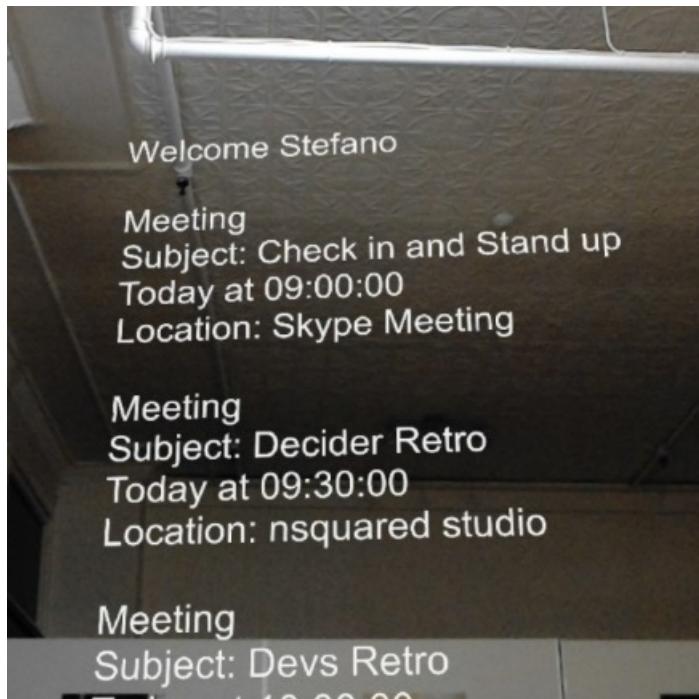
1. You will need the IP Address of your HoloLens (for Remote Deploy), and to ensure your HoloLens is in **Developer Mode**. To do this:
 - a. Whilst wearing your HoloLens, open the **Settings**.
 - b. Go to **Network & Internet > Wi-Fi > Advanced Options**
 - c. Note the **IPv4** address.
 - d. Next, navigate back to **Settings**, and then to **Update & Security > For Developers**
 - e. Set **Developer Mode On**.
2. Navigate to your new Unity build (the **App** folder) and open the solution file with **Visual Studio**.
3. In the **Solution Configuration** select **Debug**.
4. In the **Solution Platform**, select **x86, Remote Machine**. You will be prompted to insert the **IP address** of a remote device (the HoloLens, in this case, which you noted).



5. Go to **Build** menu and click on **Deploy Solution** to sideload the application to your HoloLens.
6. Your app should now appear in the list of installed apps on your HoloLens, ready to be launched!

Your Microsoft Graph HoloLens application

Congratulations, you built a mixed reality app that leverages the Microsoft Graph, to read and display user Calendar data.



Bonus exercises

Exercise 1

Use Microsoft Graph to display other information about the user

- User email / phone number / profile picture

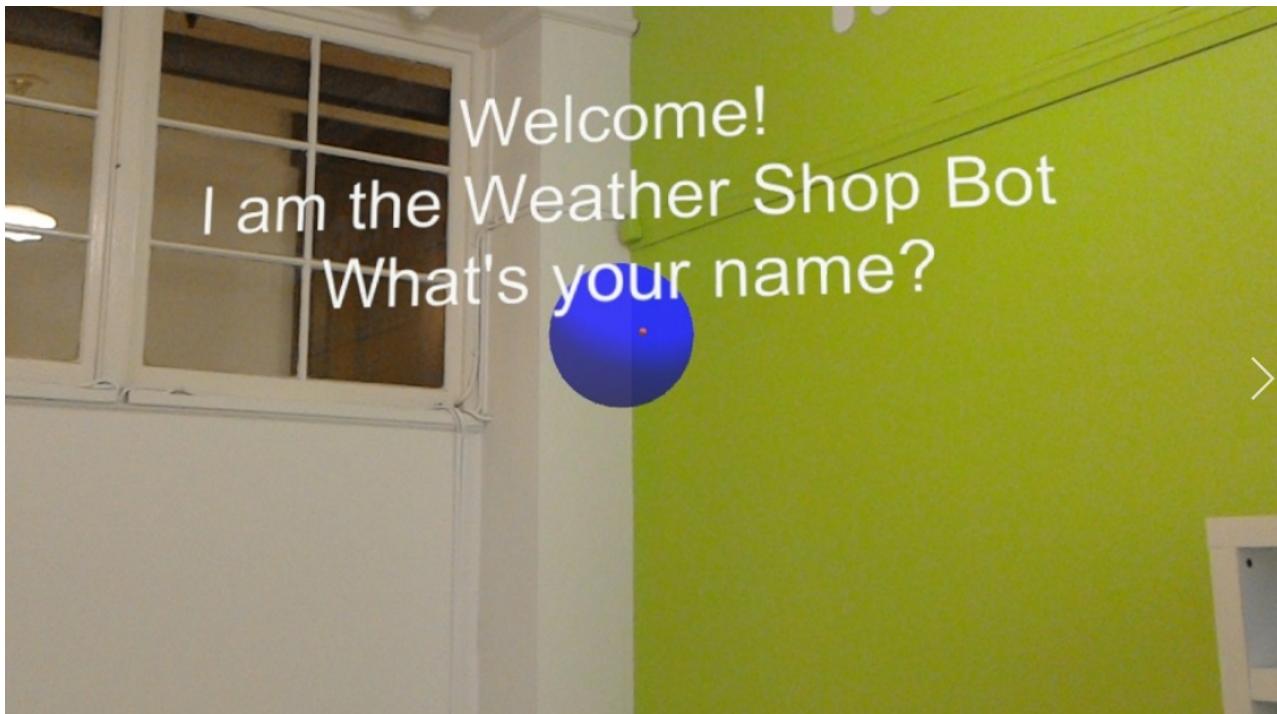
Exercise 1

Implement voice control to navigate the Microsoft Graph UI.

MR and Azure 312: Bot integration

11/6/2018 • 31 minutes to read • [Edit Online](#)

In this course, you will learn how to create and deploy a bot using the Microsoft Bot Framework V4 and communicate with it through a Windows Mixed Reality application.



The **Microsoft Bot Framework V4** is a set of APIs designed to provide developers with the tools to build an extensible and scalable bot application. For more information, visit the [Microsoft Bot Framework page](#) or the [V4 Git Repository](#).

After completing this course, you will have built a Windows Mixed Reality application, which will be able to do the following:

1. Use a **Tap Gesture** to start the bot listening for the users voice.
2. When the user has said something, the bot will attempt to provide a response.
3. Display the bots reply as text, positioned near the bot, in the Unity Scene.

In your application, it is up to you as to how you will integrate the results with your design. This course is designed to teach you how to integrate an Azure Service with your Unity project. It is your job to use the knowledge you gain from this course to enhance your mixed reality application.

Device support

COURSE	HOOLENS	IMMERSIVE HEADSETS
MR and Azure 312: Bot integration	✓ <input type="checkbox"/>	✓ <input type="checkbox"/>

NOTE

While this course primarily focuses on HoloLens, you can also apply what you learn in this course to Windows Mixed Reality immersive (VR) headsets. Because immersive (VR) headsets do not have accessible cameras, you will need an external camera connected to your PC. As you follow along with the course, you will see notes on any changes you might need to employ to support immersive (VR) headsets.

Prerequisites

NOTE

This tutorial is designed for developers who have basic experience with Unity and C#. Please also be aware that the prerequisites and written instructions within this document represent what has been tested and verified at the time of writing (July 2018). You are free to use the latest software, as listed within the [install the tools](#) article, though it should not be assumed that the information in this course will perfectly match what you will find in newer software than what is listed below.

We recommend the following hardware and software for this course:

- A development PC, [compatible with Windows Mixed Reality](#) for immersive (VR) headset development
- [Windows 10 Fall Creators Update \(or later\)](#) with Developer mode enabled
- [The latest Windows 10 SDK](#)
- [Unity 2017.4](#)
- [Visual Studio 2017](#)
- A [Windows Mixed Reality immersive \(VR\) headset](#) or [Microsoft HoloLens](#) with Developer mode enabled
- Internet access for Azure, and for Azure Bot retrieval. For more information, [please follow this link](#).

Before you start

1. To avoid encountering issues building this project, it is strongly suggested that you create the project mentioned in this tutorial in a root or near-root folder (long folder paths can cause issues at build-time).
2. Set up and test your HoloLens. If you need support setting up your HoloLens, [make sure to visit the HoloLens setup article](#).
3. It is a good idea to perform Calibration and Sensor Tuning when beginning developing a new HoloLens app (sometimes it can help to perform those tasks for each user).

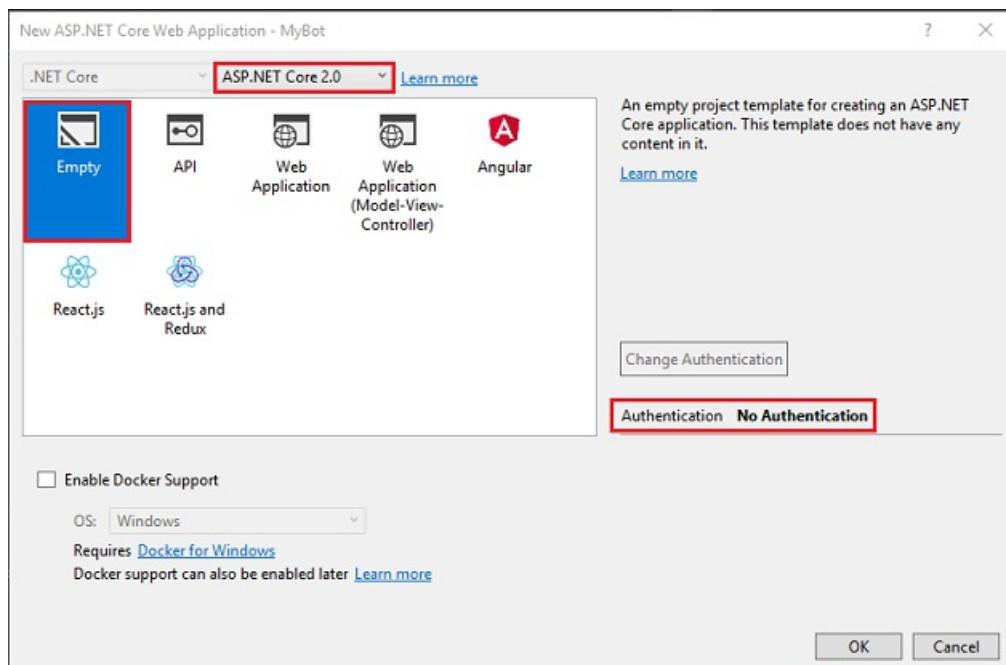
For help on Calibration, please follow this [link to the HoloLens Calibration article](#).

For help on Sensor Tuning, please follow this [link to the HoloLens Sensor Tuning article](#).

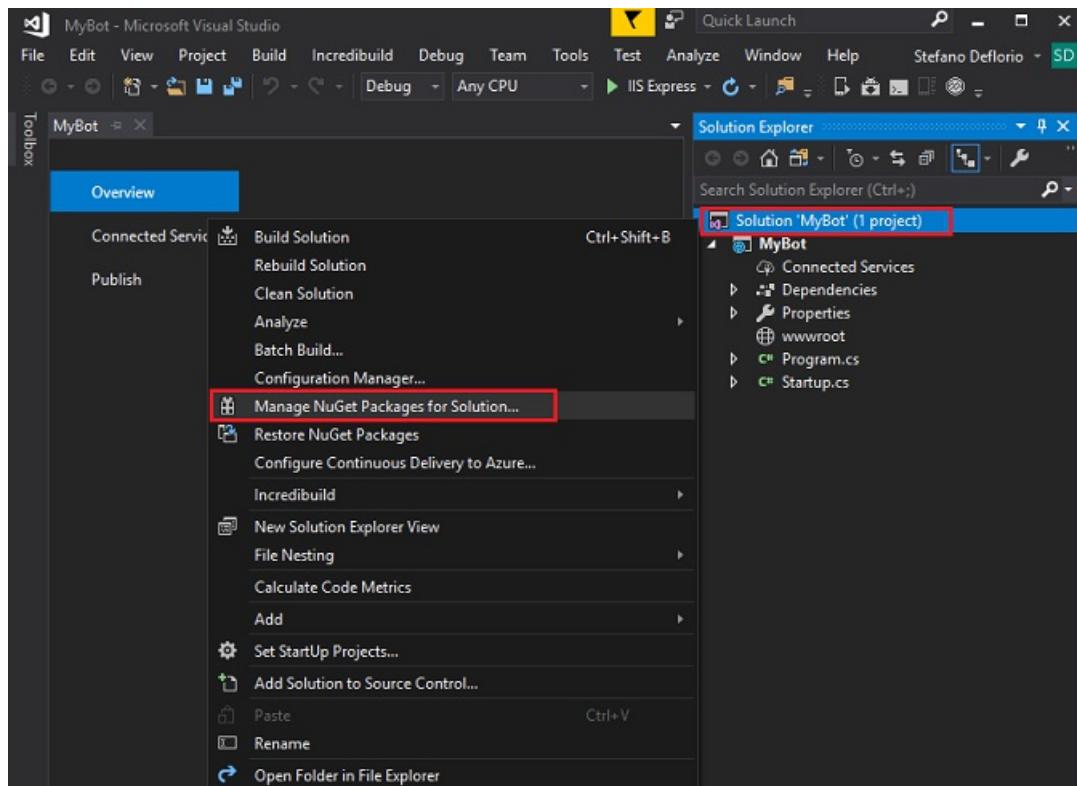
Chapter 1 – Create the Bot application

The first step is to create your bot as a local ASP.Net Core Web application. Once you have finished and tested it, you will publish it to the Azure Portal.

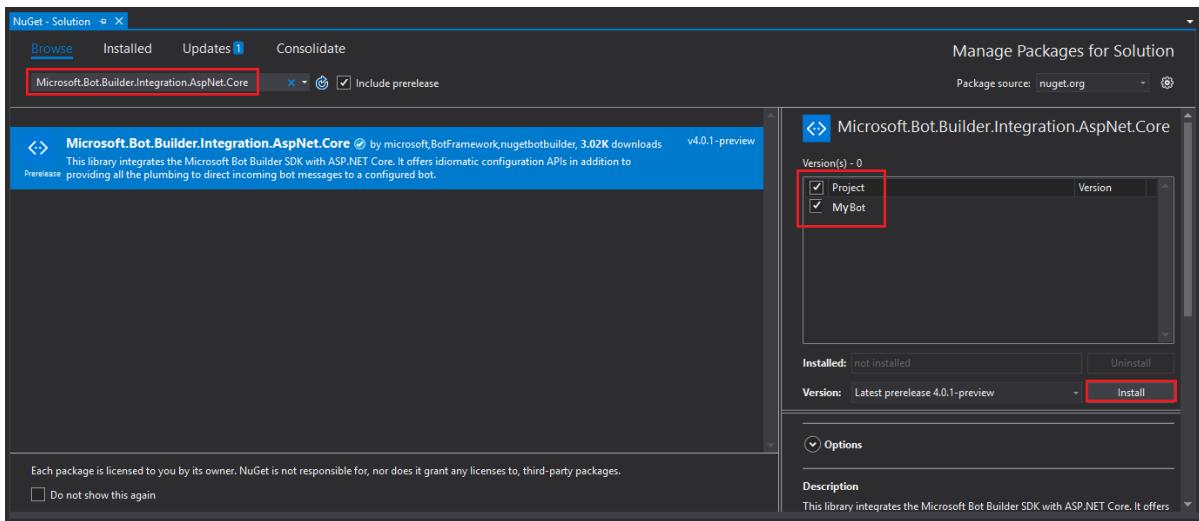
1. Open Visual Studio. Create a new project, select **ASP NET Core Web Application** as the project type (you will find it under the subsection .NET Core) and call it **MyBot**. Click **OK**.
2. In the Window that will appear select **Empty**. Also make sure the target is set to **ASP NET Core 2.0** and the Authentication is set to **No Authentication**. Click **OK**.



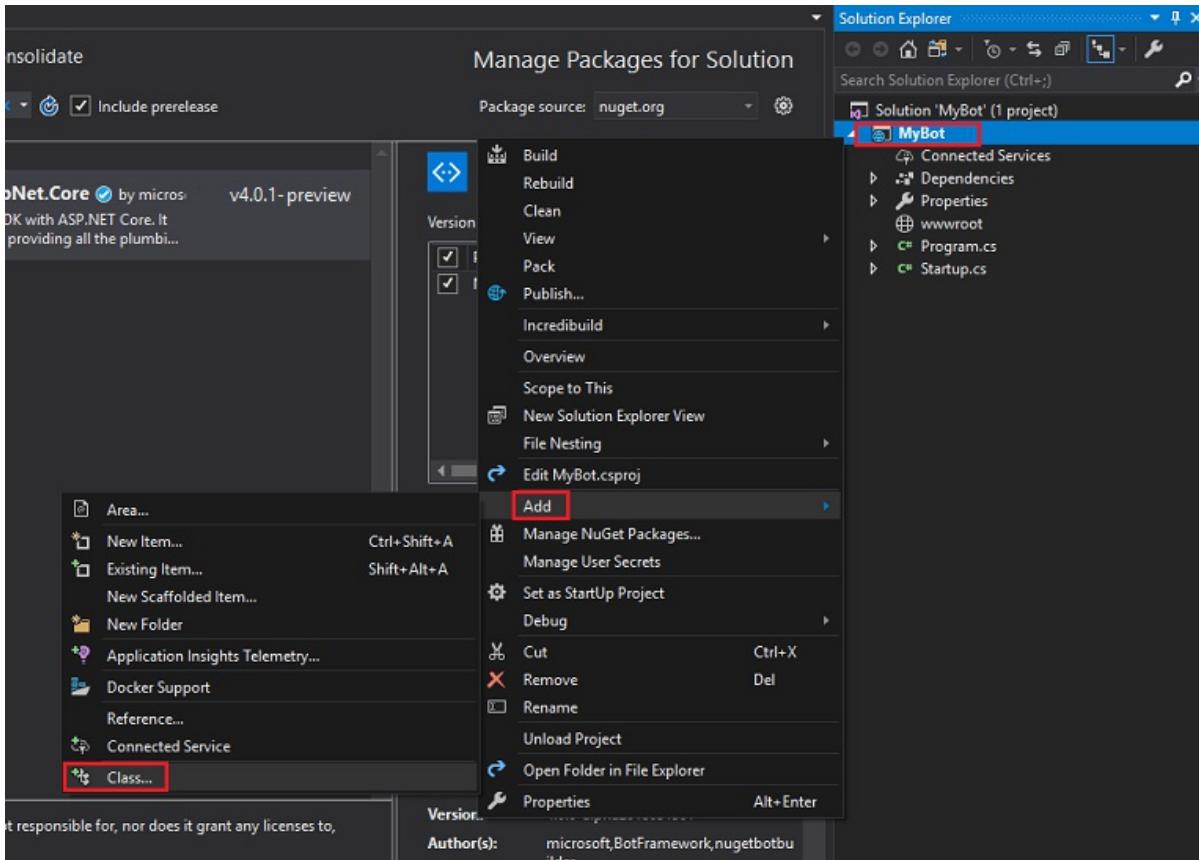
- The solution will now open. Right-click on Solution **Mybot** in the **Solution Explorer** and click on **Manage NuGet Packages for Solution**.



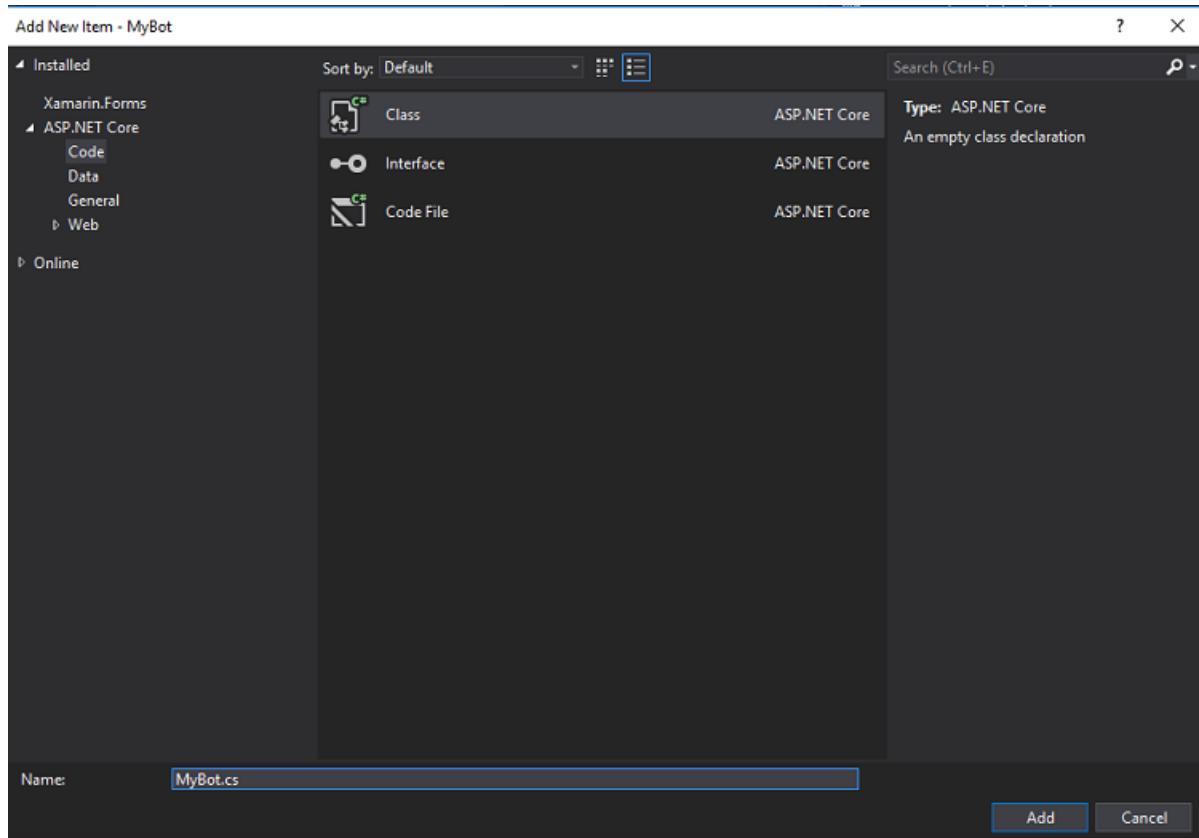
- In the **Browse** tab, search for **Microsoft.Bot.Builder.Integration.AspNet.Core** (make sure you have **Include pre-release** checked). Select the package version **4.0.1-preview**, and tick the project boxes. Then click on **Install**. You have now installed the libraries needed for the **Bot Framework v4**. Close the NuGet page.



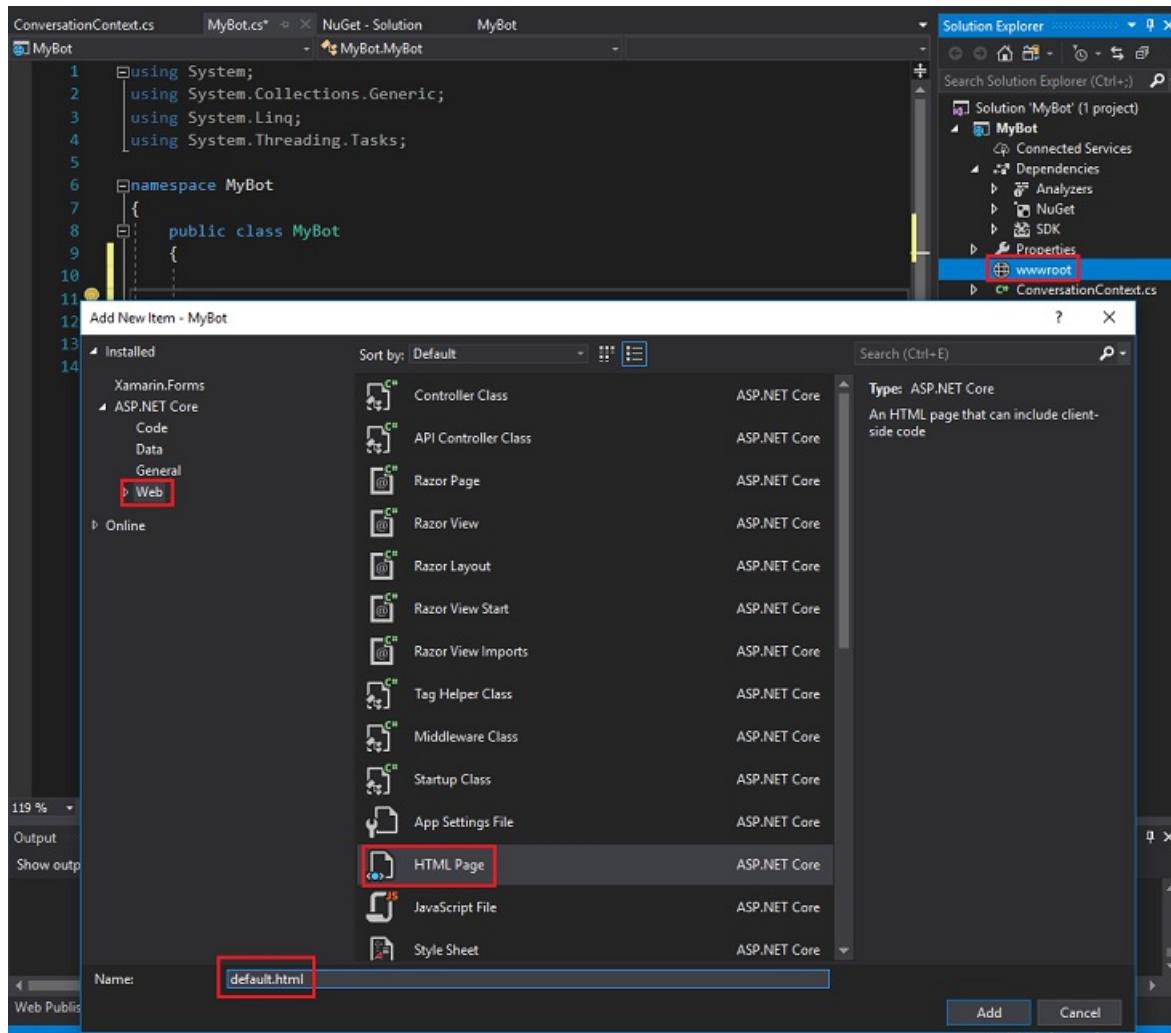
5. Right-click on your **Project**, **MyBot**, in the **Solution Explorer** and click on **Add | Class**.



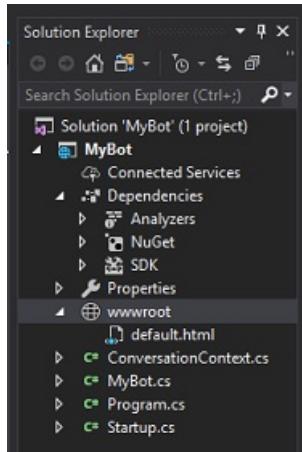
6. Name the class **MyBot** and click on **Add**.



7. Repeat the previous point, to create another class named **ConversationContext**.
8. Right-click on **wwwroot** in the **Solution Explorer** and click on **Add | New Item**. Select **HTML Page** (you will find it under the subsection Web). Name the file **default.html**. Click **Add**.



9. The list of classes / objects in the **Solution Explorer** should look like the image below.



10. Double-click on the **ConversationContext** class. This class is responsible for holding the variables used by the bot to maintain the context of the conversation. These conversation context values are maintained in an instance of this class, because any instance of the **MyBot** class will refresh each time an activity is received. Add the following code to the class:

```
namespace MyBot
{
    public static class ConversationContext
    {
        internal static string userName;

        internal static string userMsg;
    }
}
```

11. Double-click on the **MyBot** class. This class will host the handlers called by any incoming activity from the customer. In this class you will add the code used to build the conversation between the bot and the customer. As mentioned earlier, an instance of this class is initialized each time an activity is received. Add the following code to this class:

```

using Microsoft.Bot;
using Microsoft.Bot.Builder;
using Microsoft.Bot.Schema;
using System.Threading.Tasks;

namespace MyBot
{
    public class MyBot : IBot
    {
        public async Task OnTurn(ITurnContext context)
        {
            ConversationContext.userMsg = context.Activity.Text;

            if (context.Activity.Type is ActivityTypes.Message)
            {
                if (string.IsNullOrEmpty(ConversationContext.userName))
                {
                    ConversationContext.userName = ConversationContext.userMsg;
                    await context.SendActivity($"Hello {ConversationContext.userName}. Looks like today it is going to rain. \nLuckily I have umbrellas and waterproof jackets to sell!");
                }
                else
                {
                    if (ConversationContext.userMsg.Contains("how much"))
                    {
                        if (ConversationContext.userMsg.Contains("umbrella")) await context.SendActivity($"Umbrellas are $13.");
                        else if (ConversationContext.userMsg.Contains("jacket")) await context.SendActivity($"Waterproof jackets are $30.");
                        else await context.SendActivity($"Umbrellas are $13. \nWaterproof jackets are $30.");
                    }
                    else if (ConversationContext.userMsg.Contains("color") || ConversationContext.userMsg.Contains("colour"))
                    {
                        await context.SendActivity($"Umbrellas are black. \nWaterproof jackets are yellow.");
                    }
                    else
                    {
                        await context.SendActivity($"Sorry {ConversationContext.userName}. I did not understand the question");
                    }
                }
            }
            else
            {

                ConversationContext.userMsg = string.Empty;
                ConversationContext.userName = string.Empty;
                await context.SendActivity($"Welcome! \nI am the Weather Shop Bot \nWhat is your name?");
            }
        }
    }
}

```

12. Double-click on the **Startup** class. This class will initialize the bot. Add the following code to the class:

```

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Bot.Builder.BotFramework;
using Microsoft.Bot.Builder.Integration.AspNet.Core;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

namespace MyBot
{
    public class Startup
    {
        public IConfiguration Configuration { get; }

        public Startup(IHostingEnvironment env)
        {
            var builder = new ConfigurationBuilder()
                .SetBasePath(env.ContentRootPath)
                .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
                .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true)
                .AddEnvironmentVariables();
            Configuration = builder.Build();
        }

        // This method gets called by the runtime. Use this method to add services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddSingleton(_ => Configuration);
            services.AddBot<MyBot>(options =>
            {
                options.CredentialProvider = new ConfigurationCredentialProvider(Configuration);
            });
        }

        // This method gets called by the runtime. Use this method to configure the HTTP request
        pipeline.
        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }

            app.UseDefaultFiles();
            app.UseStaticFiles();
            app.UseBotFramework();
        }
    }
}

```

13. Open the **Program** class file and verify the code in it is the same as the following:

```

using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;

namespace MyBot
{
    public class Program
    {
        public static void Main(string[] args)
        {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}

```

14. Remember to save your changes, to do so, go to **File > Save All**, from the toolbar at the top of Visual Studio.

Chapter 2 - Create the Azure Bot Service

Now that you have built the code for your bot, you have to publish it to an instance of the *Web App Bot Service*, on the Azure Portal. This Chapter will show you how to create and configure the Bot Service on Azure and then publish your code to it.

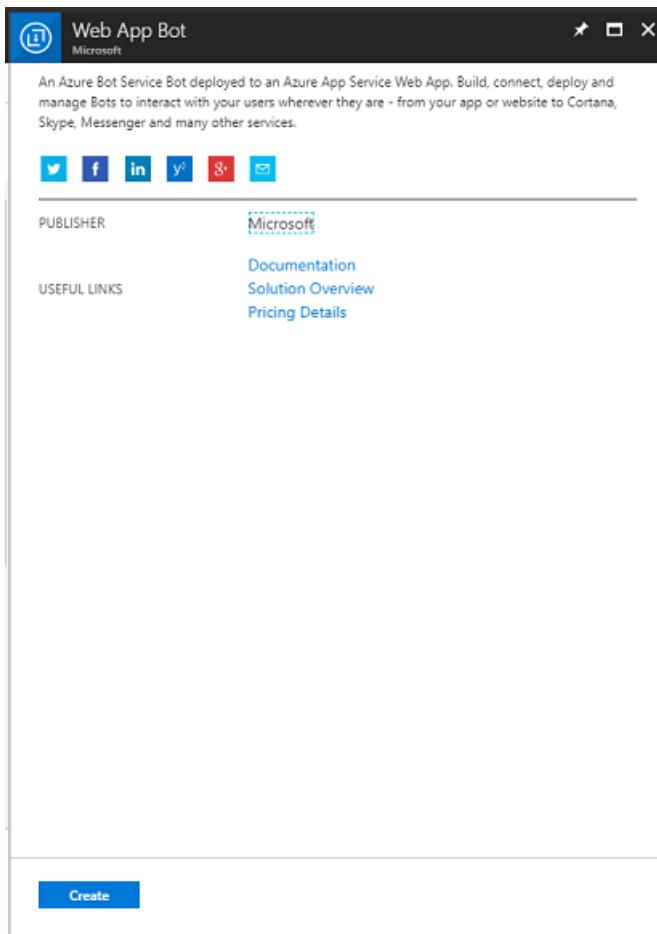
1. First, log in to the Azure Portal (<https://portal.azure.com>).

 - a. If you do not already have an Azure account, you will need to create one. If you are following this tutorial in a classroom or lab situation, ask your instructor or one of the proctors for help setting up your new account.

2. Once you are logged in, click on **Create a resource** in the top left corner, and search for *Web App bot*, and click **Enter**.

NAME	PUBLISHER	CATEGORY
Web App Bot	Microsoft	AI + Cognitive Services
Fortinet Web Application Firewall - FortiWeb	Fortinet	Compute
Signal Sciences WPP-BYOL	Signal Sciences	Compute
DenyAll rWeb	DenyAll	Compute
App Service Certificate	Microsoft	Security + Identity
VBOT CMS	Videobot	Compute
LEMP Stack on Ubuntu 16.04 LTS	CloudHub Technologies	Compute
Pulse Virtual Web Application Firewall	Pulse Secure	Compute
PHP 5.6 - Zend Server Developer Edition	Zend Technologies	Compute
C1 CMS	Orchestra	Web + Mobile

3. The new page will provide a description of the *Web App Bot Service*. At the bottom left of this page, select the **Create** button, to create an association with this Service.

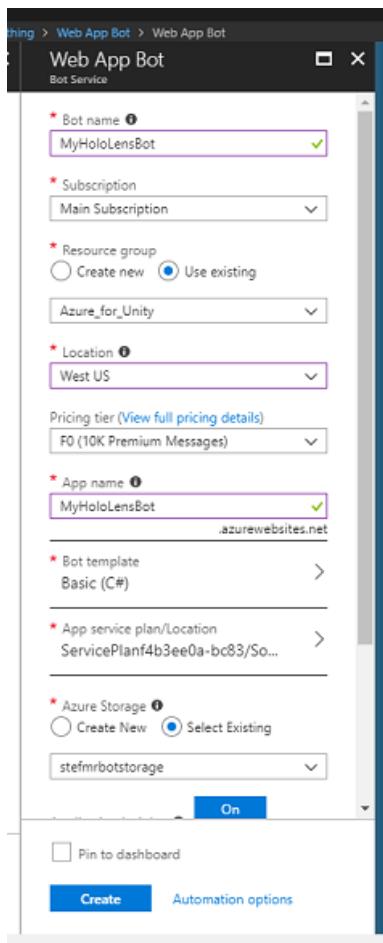


4. Once you have clicked on **Create**:

- a. Insert your desired **Name** for this Service instance.
- b. Select a **Subscription**.
- c. Choose a **Resource Group** or create a new one. A resource group provides a way to monitor, control access, provision and manage billing for a collection of Azure assets. It is recommended to keep all the Azure Services associated with a single project (e.g. such as these courses) under a common resource group.

If you wish to read more about Azure Resource Groups, [please follow this link](#)
- d. Determine the Location for your resource group (if you are creating a new Resource Group). The location would ideally be in the region where the application would run. Some Azure assets are only available in certain regions.
- e. Select the **Pricing Tier** appropriate for you, if this is the first time creating a *Web App Bot Service*, a free tier (named F0) should be available to you
- f. **App name** can just be left the same as the *Bot name*.
- g. Leave the *Bot template* as **Basic (C#)**.
- h. *App service plan/Location* should have been auto-filled for your account.
- i. Set the **Azure Storage** that you wish to use to host your Bot. If you don't have one already, you can create it here.
- j. You will also need to confirm that you have understood the Terms and Conditions applied to this Service.

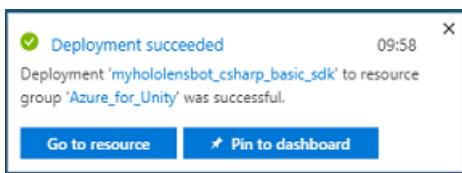
k. Click Create.



5. Once you have clicked on **Create**, you will have to wait for the Service to be created, this might take a minute.
6. A notification will appear in the Portal once the Service instance is created.



7. Click on the notification to explore your new Service instance.



8. Click the **Go to resource** button in the notification to explore your new Service instance. You will be taken to your new Azure Service instance.

The screenshot shows the Azure Bot Service Overview page for a bot named 'MyHoloLensBot'. The left sidebar contains navigation links for Overview, Activity log, Access control (IAM), Tags, Bot Management (Build, Test in Web Chat, Analytics, Channels, Settings, Speech priming, Bot Service pricing), App Service Settings (Application Settings, All App service settings), and Support + Troubleshooting (New support request). The main content area displays basic bot information: Resource group (Azure_for_Unity), Subscription (Main Subscription), Subscription ID (733...9b), Bot Service pricing tier (F0), and Messaging endpoint (https://myholo...). Below this, there are three resource cards: 'Documents' (book icon), 'Samples' (gear icon), and 'Feedback' (speech bubble icon).

9. At this point you need to setup a feature called **Direct Line** to allow your client application to communicate with this Bot Service. Click on **Channels**, then in the **Add a featured channel** section, click on **Configure Direct Line channel**.

The screenshot shows the Azure Bot Service Channels page for 'MyHoloLensBot - Channels'. The left sidebar includes the same navigation links as the previous screen. The main content area features a heading 'Connect to channels' and a table showing one active channel: 'Web Chat' (status: Running). Below this, there's a section titled 'Add a featured channel' with a button labeled 'Configure Direct Line channel' highlighted with a red box. A 'More channels' section lists several other communication platforms: Bing, Email, Kik, Skype for Business, and Twilio (SMS).

10. In this page you will find the **Secret keys** that will allow your client app to authenticate with the bot. Click on the **Show** button and take a copy of one of the displayed Keys, as you will need this later in your project.

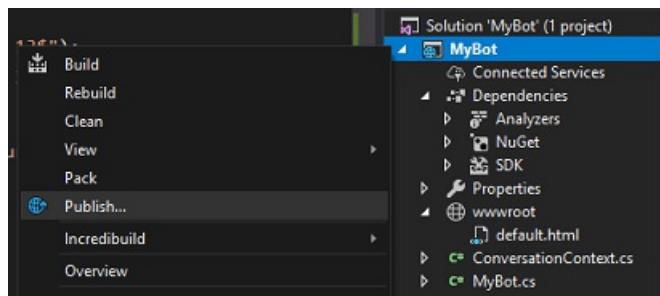
Chapter 3 – Publish the Bot to the Azure Web App Bot Service

Now that your Service is ready, you need to publish your Bot code, that you built previously, to your newly created Web App Bot Service.

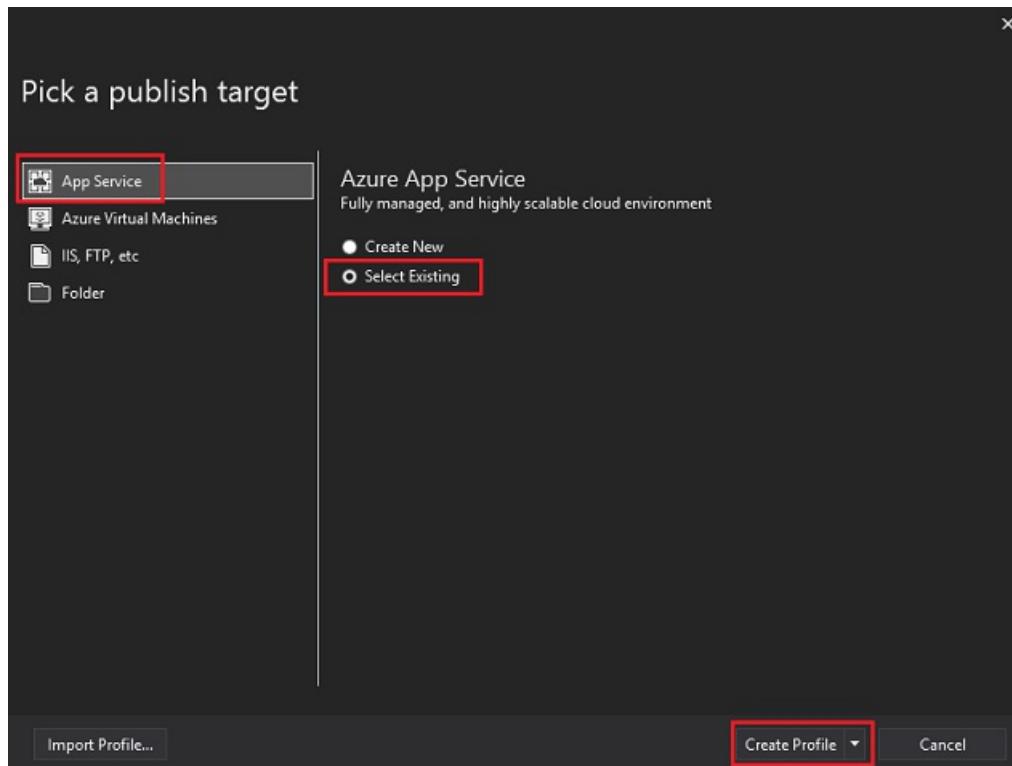
NOTE

You will have to publish your Bot to the Azure Service every time you make changes to the Bot solution / code.

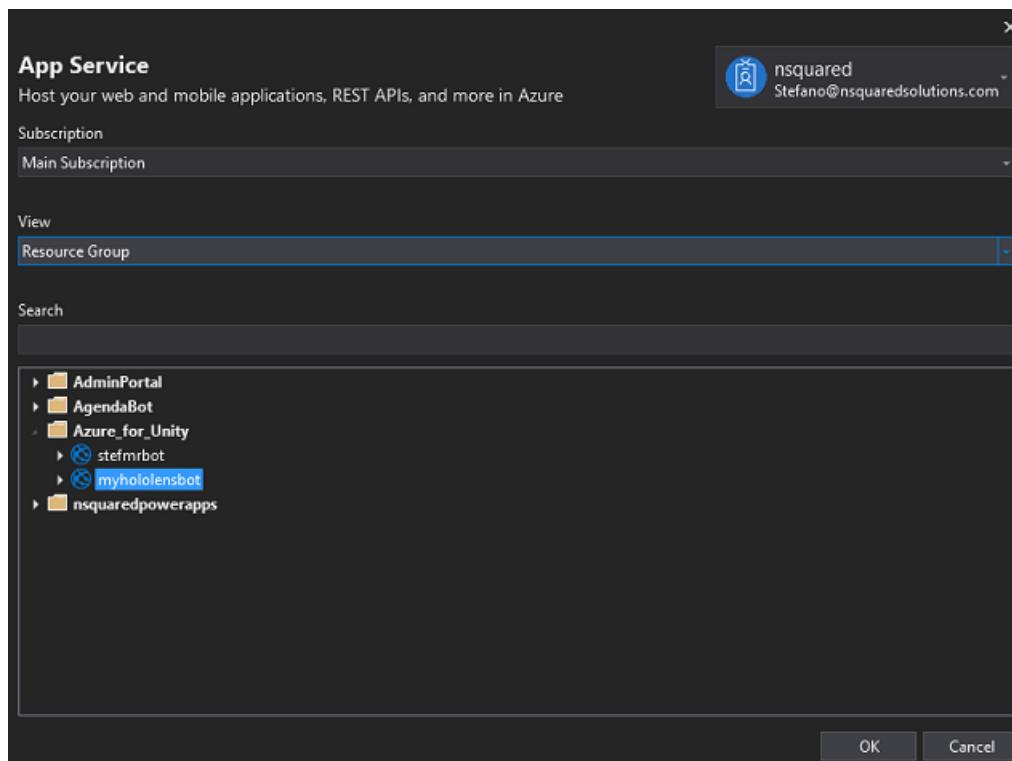
1. Go back to your Visual Studio Solution that you created previously.
2. Right-click on your **MyBot** project, in the **Solution Explorer**, then click on **Publish**.



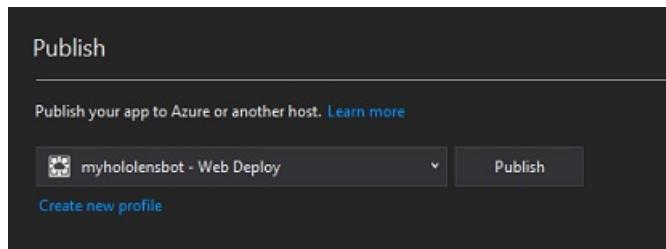
3. On the *Pick a publish target* page, click **App Service**, then **Select Existing**, lastly click on **Create Profile** (you may need to click on the dropdown arrow alongside the *Publish* button, if this is not visible).



4. If you are not yet logged in into your Microsoft Account, you have to do it here.
5. On the **Publish** page you will find you have to set the same **Subscription** that you used for the *Web App Bot* Service creation. Then set the **View as Resource Group** and, in the drop down folder structure, select the **Resource Group** you have created previously. Click **OK**.



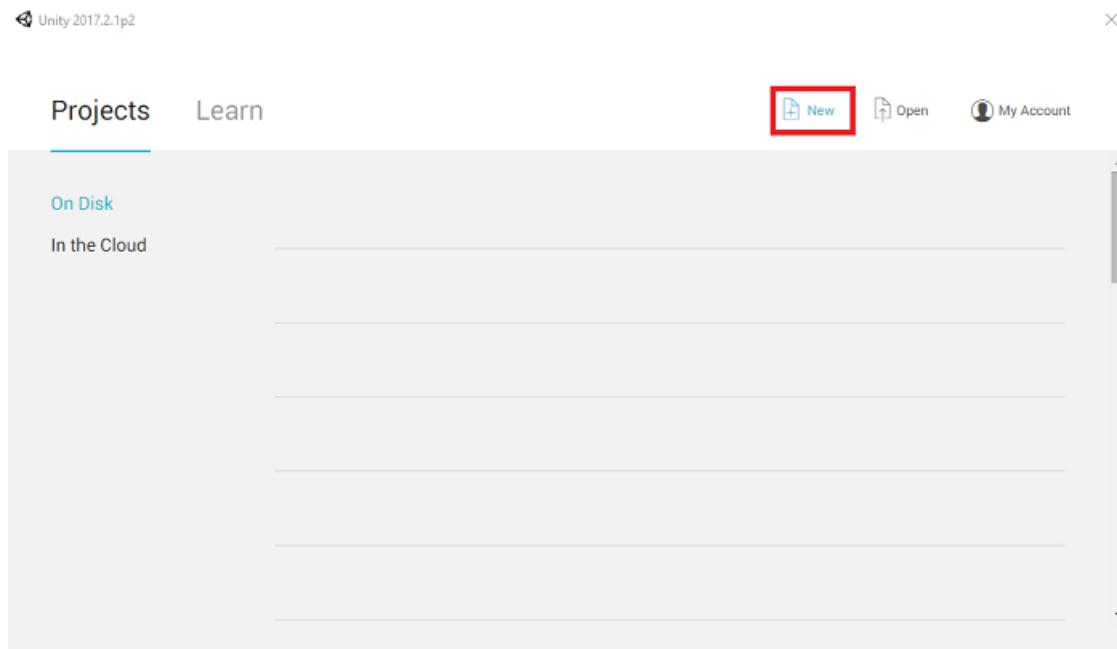
6. Now click on the **Publish** button, and wait for the Bot to be published (it might take a few minutes).



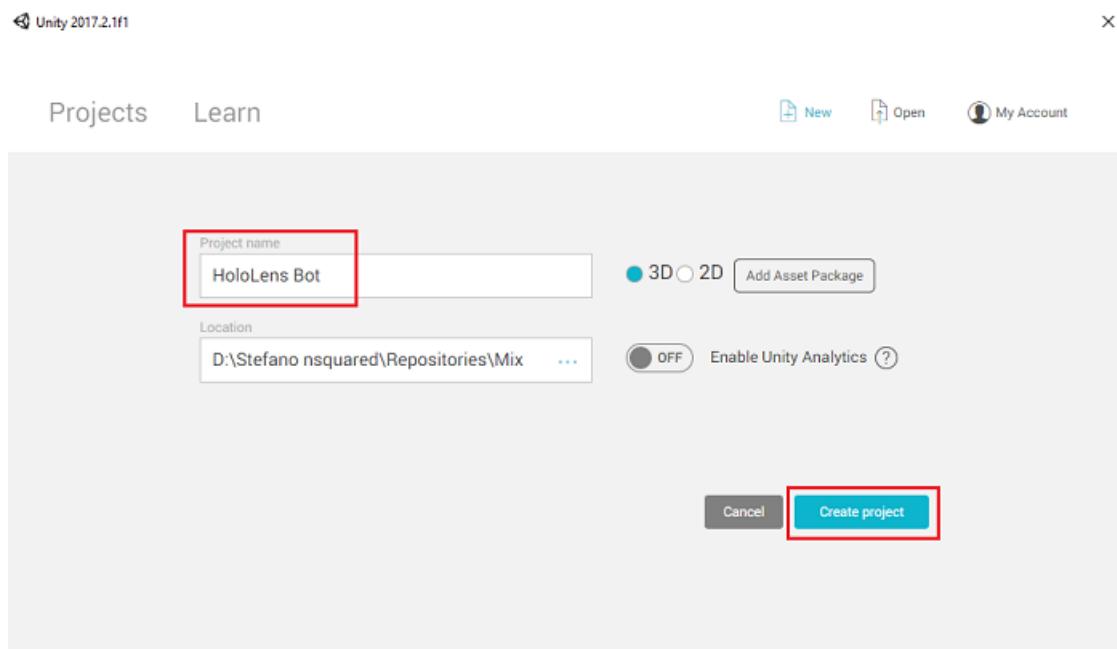
Chapter 4 – Set up the Unity project

The following is a typical set up for developing with mixed reality, and as such, is a good template for other projects.

1. Open *Unity* and click **New**.

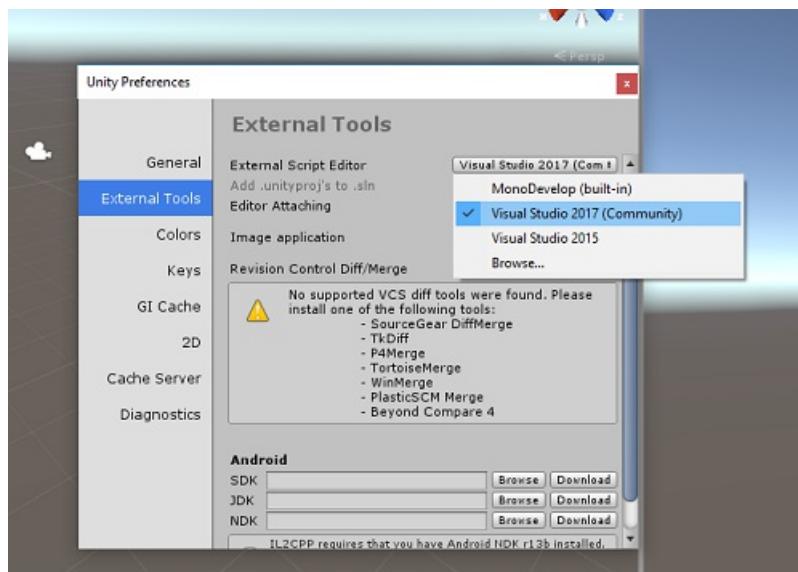


2. You will now need to provide a Unity project name. Insert **Hololens Bot**. Make sure the project template is set to **3D**. Set the **Location** to somewhere appropriate for you (remember, closer to root directories is better). Then, click **Create project**.

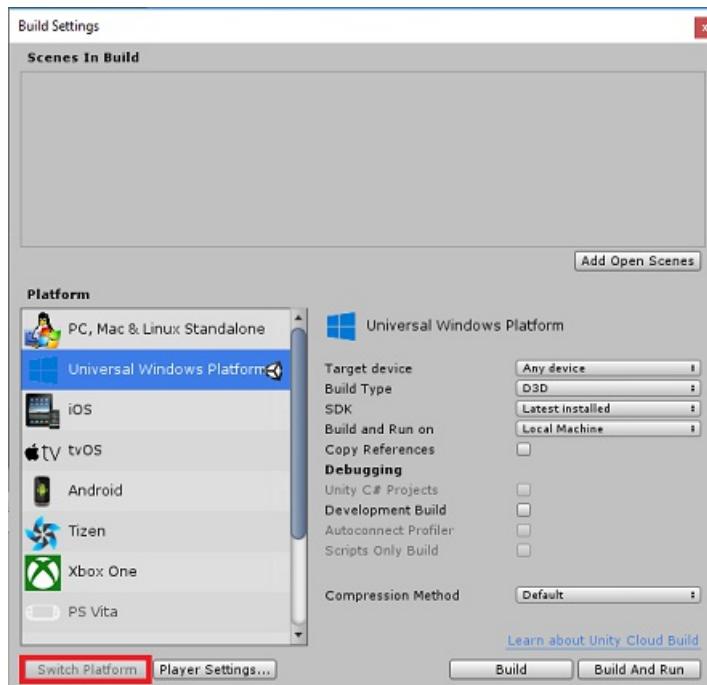


3. With Unity open, it is worth checking the default **Script Editor** is set to **Visual Studio**. Go to **Edit >**

Preferences and then from the new window, navigate to **External Tools**. Change **External Script Editor** to **Visual Studio 2017**. Close the **Preferences** window.



4. Next, go to **File > Build Settings** and select **Universal Windows Platform**, then click on the **Switch Platform** button to apply your selection.



5. While still in **File > Build Settings** and make sure that:

- a. **Target Device** is set to **Hololens**

For the immersive headsets, set **Target Device** to **Any Device**.

- b. **Build Type** is set to **D3D**

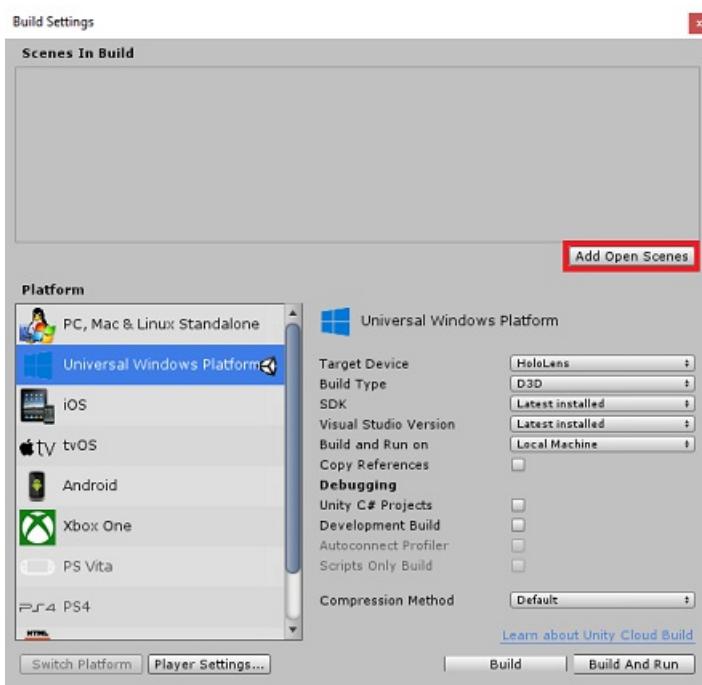
- c. **SDK** is set to **Latest installed**

- d. **Visual Studio Version** is set to **Latest installed**

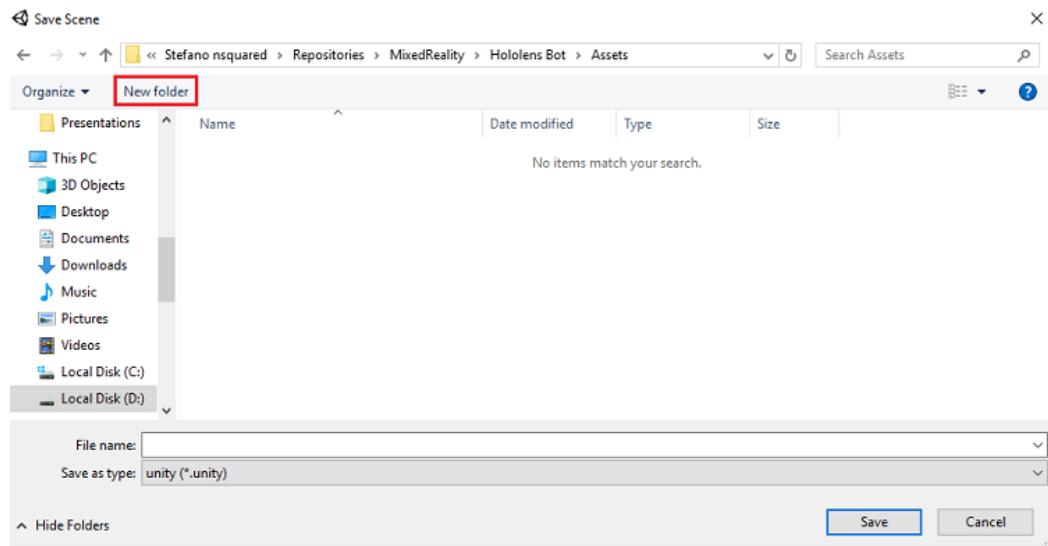
- e. **Build and Run** is set to **Local Machine**

- f. Save the scene and add it to the build.

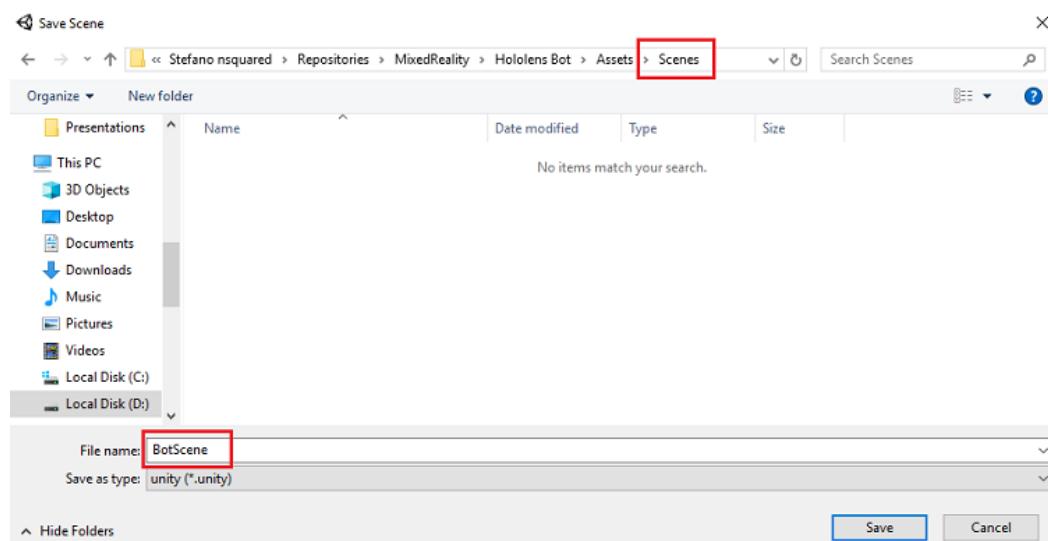
- a. Do this by selecting **Add Open Scenes**. A save window will appear.



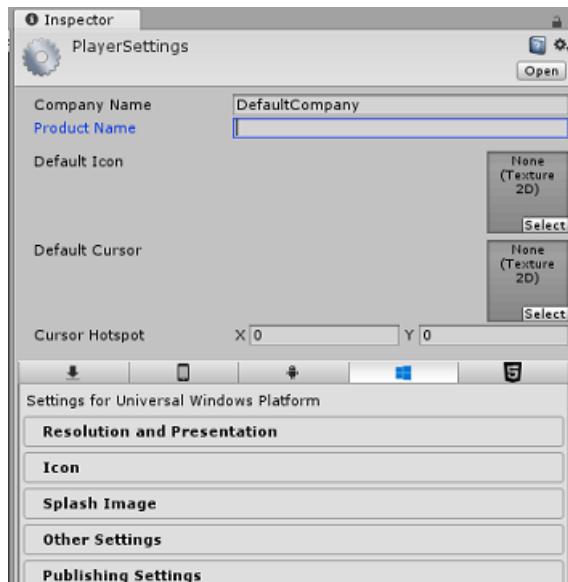
- b. Create a new folder for this, and any future, scene, then select the **New folder** button, to create a new folder, name it **Scenes**.



- c. Open your newly created **Scenes** folder, and then in the *File name:* text field, type **BotScene**, then click on **Save**.

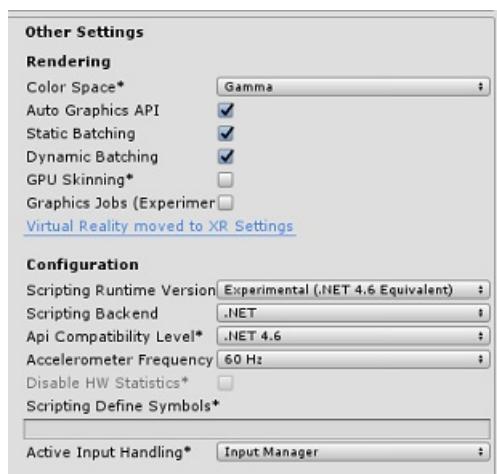


- g. The remaining settings, in **Build Settings**, should be left as default for now.
6. In the *Build Settings* window, click on the **Player Settings** button, this will open the related panel in the space where the *Inspector* is located.

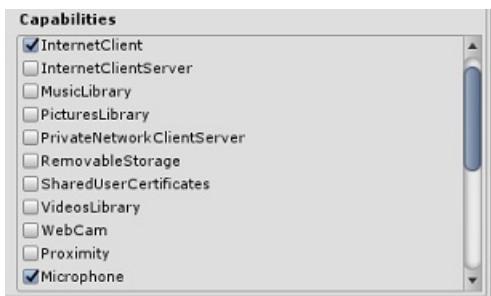


7. In this panel, a few settings need to be verified:

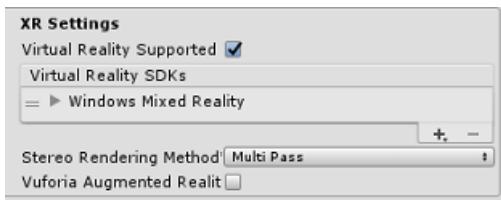
- In the **Other Settings** tab:
 - Scripting Runtime Version** should be **Experimental (NET 4.6 Equivalent)**; changing this will require a restart of the Editor.
 - Scripting Backend** should be **.NET**
 - API Compatibility Level** should be **.NET 4.6**



- Within the **Publishing Settings** tab, under **Capabilities**, check:
 - **InternetClient**
 - **Microphone**



- c. Further down the panel, in **XR Settings** (found below **Publish Settings**), tick **Virtual Reality Supported**, make sure the **Windows Mixed Reality SDK** is added.



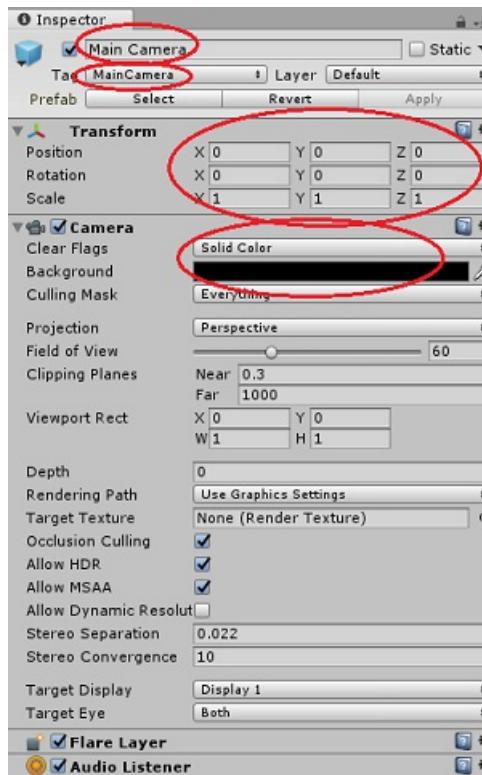
8. Back in *Build Settings Unity C# Projects* is no longer greyed out; tick the checkbox next to this.
9. Close the Build Settings window.
10. Save your scene and project (**FILE > SAVE SCENE / FILE > SAVE PROJECT**).

Chapter 5 – Camera setup

IMPORTANT

If you wish to skip the *Unity Set up* component of this course, and continue straight into code, feel free to download this [Azure-MR-312-Package.unitypackage](#), import it into your project as a **Custom Package**, and then continue from [Chapter 7](#).

1. In the *Hierarchy panel*, select the **Main Camera**.
2. Once selected, you will be able to see all the components of the **Main Camera** in the *Inspector panel*.
 - a. The **Camera object** must be named **Main Camera** (note the spelling)
 - b. The Main Camera **Tag** must be set to **MainCamera** (note the spelling)
 - c. Make sure the **Transform Position** is set to **0, 0, 0**
 - d. Set **Clear Flags** to **Solid Color**.
 - e. Set the **Background Color** of the Camera component to **Black, Alpha 0 (Hex Code: #00000000)**

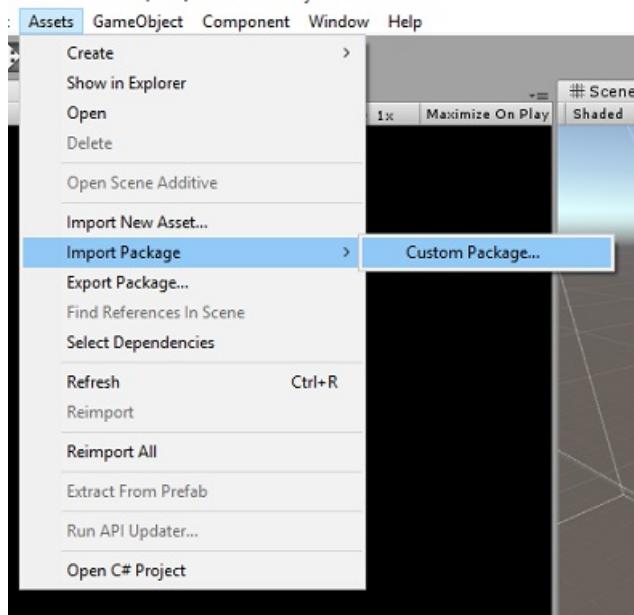


Chapter 6 – Import the Newtonsoft library

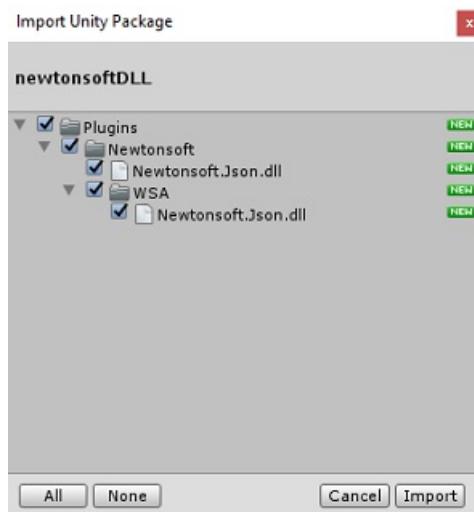
To help you deserialize and serialize objects received and sent to the Bot Service you need to download the **Newtonsoft** library. You will find a [compatible version already organized with the correct Unity folder structure here](#).

To import the Newtonsoft library into your project, use the Unity Package which came with this course.

1. Add the *.unitypackage* to Unity by using the **Assets > Import Package > Custom Package** menu option.



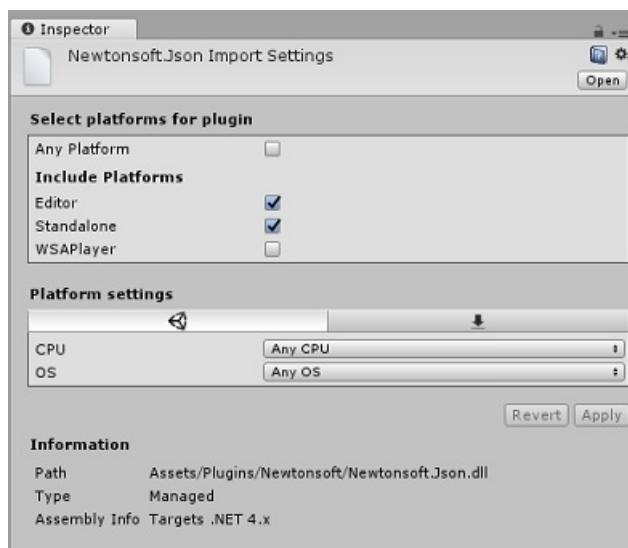
2. In the **Import Unity Package** box that pops up, ensure everything under (and including) **Plugins** is selected.



3. Click the **Import** button to add the items to your project.
4. Go to the **Newtonsoft** folder under **Plugins** in the project view and select the Newtonsoft plugin.



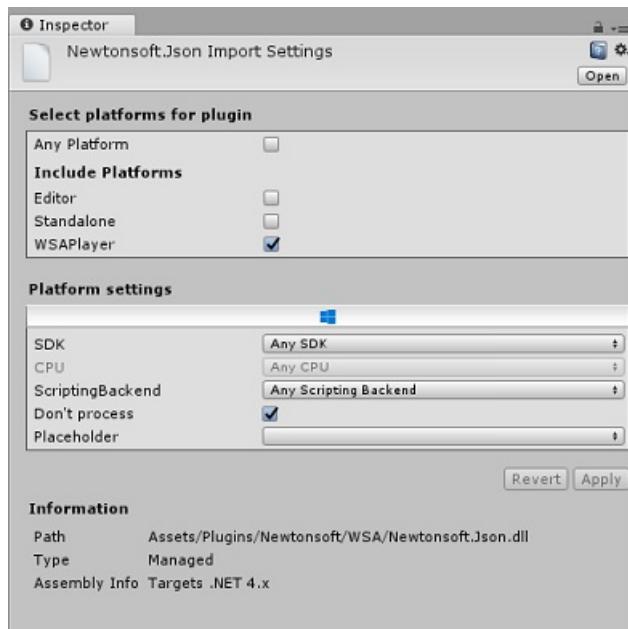
5. With the Newtonsoft plugin selected, ensure that **Any Platform** is **unchecked**, then ensure that **WSAPlayer** is also **unchecked**, then click **Apply**. This is just to confirm that the files are configured correctly.



NOTE

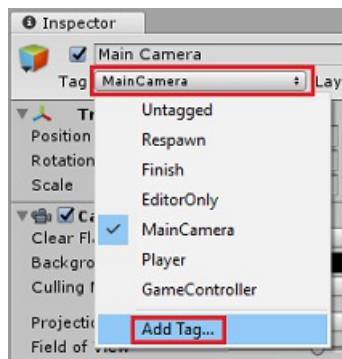
Marking these plugins configures them to only be used in the Unity Editor. There are a different set of them in the WSA folder which will be used after the project is exported from Unity.

6. Next, you need to open the **WSA** folder, within the **Newtonsoft** folder. You will see a copy of the same file you just configured. Select the file, and then in the inspector, ensure that
 - **Any Platform** is **unchecked**
 - **only WSAPlayer** is **checked**
 - **Dont process** is **checked**

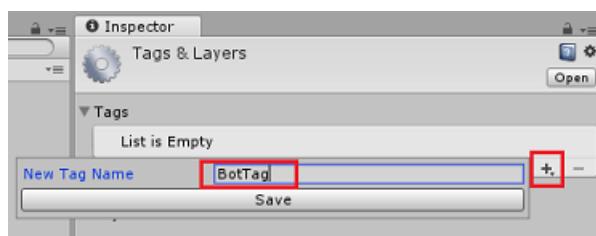


Chapter 7 – Create the BotTag

1. Create a new **Tag** object called **BotTag**. Select the Main Camera in the scene. Click on the Tag drop down menu in the Inspector panel. Click on **Add Tag...**.



2. Click on the + symbol. Name the new **Tag** as **BotTag**, Save.



WARNING

Do not apply the BotTag to the Main Camera. If you have accidentally done this, make sure to change the Main Camera tag back to *MainCamera*.

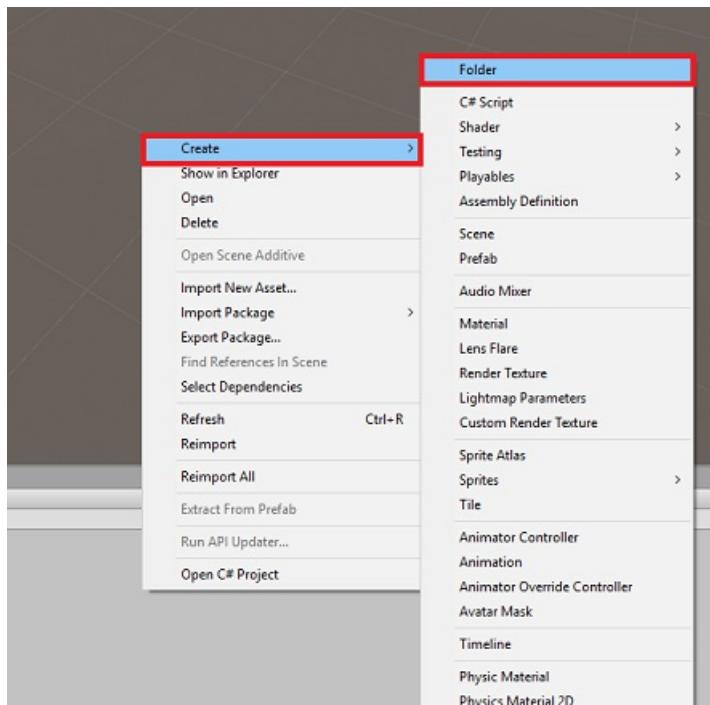
Chapter 8 – Create the BotObjects class

The first script you need to create is the **BotObjects** class, which is an empty class created so that a series of other class objects can be stored within the same script and accessed by other scripts in the scene.

The creation of this class is purely an architectural choice, these objects could instead be hosted in the Bot script that you will create later in this course.

To create this class:

1. Right-click in the *Project panel*, then **Create > Folder**. Name the folder **Scripts**.



2. Double-click on the **Scripts** folder to open it. Then within that folder, right-click, and select **Create > C# Script**. Name the script **BotObjects**.
3. Double-click on the new **BotObjects** script to open it with **Visual Studio**.
4. Delete the content of the script and replace it with the following code:

```

using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BotObjects : MonoBehaviour{}

/// <summary>
/// Object received when first opening a conversation
/// </summary>
[Serializable]
public class ConversationObject
{
    public string ConversationId;
    public string token;
    public string expires_in;
    public string streamUrl;
    public string referenceGrammarId;
}

/// <summary>
/// Object including all Activities
/// </summary>
[Serializable]
public class ActivitiesRootObject
{
    public List<Activity> activities { get; set; }
    public string watermark { get; set; }
}

[Serializable]
public class Conversation
{
    public string id { get; set; }
}

[Serializable]
public class From
{
    public string id { get; set; }
    public string name { get; set; }
}

[Serializable]
public class Activity
{
    public string type { get; set; }
    public string channelId { get; set; }
    public Conversation conversation { get; set; }
    public string id { get; set; }
    public From from { get; set; }
    public string text { get; set; }
    public string textFormat { get; set; }
    public DateTime timestamp { get; set; }
    public string serviceUrl { get; set; }
}

```

5. Be sure to save your changes in *Visual Studio* before returning to *Unity*.

Chapter 9 – Create the **GazelInput** class

The next class you are going to create is the **GazelInput** class. This class is responsible for:

- Creating a cursor that will represent the *gaze* of the player.
- Detecting objects hit by the gaze of the player, and holding a reference to detected objects.

To create this class:

1. Go to the **Scripts** folder you created previously.
2. Right-click inside the folder, **Create > C# Script**. Call the script **GazeInput**.
3. Double-click on the new **GazeInput** script to open it with **Visual Studio**.
4. Insert the following line right on top of the class name:

```
/// <summary>
/// Class responsible for the User's gaze interactions
/// </summary>
[System.Serializable]
public class GazeInput : MonoBehaviour
```

5. Then add the following variables inside the **GazeInput** class, above the **Start()** method:

```
[Tooltip("Used to compare whether an object is to be interacted with.")]
internal string InteractableTag = "BotTag";

/// <summary>
/// Length of the gaze
/// </summary>
internal float GazeMaxDistance = 300;

/// <summary>
/// Object currently gazed
/// </summary>
internal GameObject FocusedObject { get; private set; }

internal GameObject _oldFocusedObject { get; private set; }

internal RaycastHit HitInfo { get; private set; }

/// <summary>
/// Cursor object visible in the scene
/// </summary>
internal GameObject Cursor { get; private set; }

internal bool Hit { get; private set; }

internal Vector3 Position { get; private set; }

internal Vector3 Normal { get; private set; }

private Vector3 _gazeOrigin;

private Vector3 _gazeDirection;
```

6. Code for **Start()** method should be added. This will be called when the class initializes:

```
/// <summary>
/// Start method used upon initialization.
/// </summary>
internal virtual void Start()
{
    FocusedObject = null;
    Cursor = CreateCursor();
}
```

7. Implement a method that will instantiate and setup the gaze cursor:

```

/// <summary>
/// Method to create a cursor object.
/// </summary>
internal GameObject CreateCursor()
{
    GameObject newCursor = GameObject.CreatePrimitive(PrimitiveType.Sphere);
    newCursor.SetActive(false);
    // Remove the collider, so it does not block Raycast.
    Destroy(newCursor.GetComponent<SphereCollider>());
    newCursor.transform.localScale = new Vector3(0.05f, 0.05f, 0.05f);
    Material mat = new Material(Shader.Find("Diffuse"));
    newCursor.GetComponent<MeshRenderer>().material = mat;
    mat.color = Color.HSVToRGB(0.0223f, 0.7922f, 1.000f);
    newCursor.SetActive(true);

    return newCursor;
}

```

8. Implement the methods that will setup the Raycast from the Main Camera and will keep track of the current focused object.

```

/// <summary>
/// Called every frame
/// </summary>
internal virtual void Update()
{
    _gazeOrigin = Camera.main.transform.position;

    _gazeDirection = Camera.main.transform.forward;

    UpdateRaycast();
}

/// <summary>
/// Reset the old focused object, stop the gaze timer, and send data if it
/// is greater than one.
/// </summary>
private void ResetFocusedObject()
{
    // Ensure the old focused object is not null.
    if (_oldFocusedObject != null)
    {
        if (_oldFocusedObject.CompareTag(InteractableTag))
        {
            // Provide the OnGazeExited event.
            _oldFocusedObject.SendMessage("OnGazeExited",
                SendMessageOptions.DontRequireReceiver);
        }
    }
}

private void UpdateRaycast()
{
    // Set the old focused gameobject.
    _oldFocusedObject = FocusedObject;
    RaycastHit hitInfo;

    // Initialize Raycasting.
    Hit = Physics.Raycast(_gazeOrigin,
        _gazeDirection,
        out hitInfo,
        GazeMaxDistance);
    HitInfo = hitInfo;
}

```

```

// Check whether raycast has hit.
if (Hit == true)
{
    Position = hitInfo.point;
    Normal = hitInfo.normal;

    // Check whether the hit has a collider.
    if (hitInfo.collider != null)
    {
        // Set the focused object with what the user just looked at.
        FocusedObject = hitInfo.collider.gameObject;
    }
    else
    {
        // Object looked on is not valid, set focused gameobject to null.
        FocusedObject = null;
    }
}
else
{
    // No object looked upon, set focused gameobject to null.
    FocusedObject = null;

    // Provide default position for cursor.
    Position = _gazeOrigin + (_gazeDirection * GazeMaxDistance);

    // Provide a default normal.
    Normal = _gazeDirection;
}

// Lerp the cursor to the given position, which helps to stabilize the gaze.
Cursor.transform.position = Vector3.Lerp(Cursor.transform.position, Position, 0.6f);

// Check whether the previous focused object is this same. If so, reset the focused object.
if (FocusedObject != _oldFocusedObject)
{
    ResetFocusedObject();
    if (FocusedObject != null)
    {
        if (FocusedObject.CompareTag(InteractableTag))
        {
            // Provide the OnGazeEntered event.
            FocusedObject.SendMessage("OnGazeEntered",
                SendMessageOptions.DontRequireReceiver);
        }
    }
}
}

```

9. Be sure to save your changes in *Visual Studio* before returning to *Unity*.

Chapter 10 – Create the Bot class

The script you are going to create now is called **Bot**. This is the core class of your application, it stores:

- Your Web App Bot credentials
- The method that collects the user voice commands
- The method necessary to initiate conversations with your Web App Bot
- The method necessary to send messages to your Web App Bot

To send messages to the Bot Service, the **SendMessageToBot()** coroutine will build an activity, which is an object recognized by the Bot Framework as data sent by the user.

To create this class:

1. Double-click on the **Scripts** folder, to open it.
2. Right-click inside the **Scripts** folder, click **Create > C# Script**. Name the script **Bot**.
3. Double-click on the new script to open it with Visual Studio.
4. Update the namespaces to be the same as the following, at the top of the **Bot** class:

```
using Newtonsoft.Json;
using System.Collections;
using System.Text;
using UnityEngine;
using UnityEngine.Networking;
using UnityEngine.Windows.Speech;
```

5. Inside the **Bot** class add the following variables:

```

/// <summary>
/// Static instance of this class
/// </summary>
public static Bot Instance;

/// <summary>
/// Material of the sphere representing the Bot in the scene
/// </summary>
internal Material botMaterial;

/// <summary>
/// Speech recognizer class reference, which will convert speech to text.
/// </summary>
private DictationRecognizer dictationRecognizer;

/// <summary>
/// Use this variable to identify the Bot Id
/// Can be any value
/// </summary>
private string botId = "MRBotId";

/// <summary>
/// Use this variable to identify the Bot Name
/// Can be any value
/// </summary>
private string botName = "MRBotName";

/// <summary>
/// The Bot Secret key found on the Web App Bot Service on the Azure Portal
/// </summary>
private string botSecret = "-- Add your Secret Key here --";

/// <summary>
/// Bot Endpoint, v4 Framework uses v3 endpoint at this point in time
/// </summary>
private string botEndpoint = "https://directline.botframework.com/v3/directline";

/// <summary>
/// The conversation object reference
/// </summary>
private ConversationObject conversation;

/// <summary>
/// Bot states to regulate the application flow
/// </summary>
internal enum BotState {ReadyToListen, Listening, Processing}

/// <summary>
/// Flag for the Bot state
/// </summary>
internal BotState botState;

/// <summary>
/// Flag for the conversation status
/// </summary>
internal bool conversationStarted = false;

```

NOTE

Make sure you insert your **Bot Secret Key** into the **botSecret** variable. You will have noted your **Bot Secret Key** at the beginning of this course, in [Chapter 2, step 10](#).

6. Code for **Awake()** and **Start()** now needs to be added.

```

/// <summary>
/// Called on Initialization
/// </summary>
void Awake()
{
    Instance = this;
}

/// <summary>
/// Called immediately after Awake method
/// </summary>
void Start()
{
    botState = BotState.ReadyToListen;
}

```

7. Add the two handlers that are called by the speech libraries when voice capture begins and ends. The *DictationRecognizer* will automatically stop capturing the user voice when the user stops speaking.

```

/// <summary>
/// Start microphone capture.
/// </summary>
public void StartCapturingAudio()
{
    botState = BotState.Listening;
    botMaterial.color = Color.red;

    // Start dictation
    dictationRecognizer = new DictationRecognizer();
    dictationRecognizer.DictationResult += DictationRecognizer_DictationResult;
    dictationRecognizer.Start();
}

/// <summary>
/// Stop microphone capture.
/// </summary>
public void StopCapturingAudio()
{
    botState = BotState.Processing;
    dictationRecognizer.Stop();
}

```

8. The following handler collects the result of the user voice input and calls the coroutine responsible for sending the message to the Web App Bot Service.

```

/// <summary>
/// This handler is called every time the Dictation detects a pause in the speech.
/// </summary>
private void DictationRecognizer_DictationResult(string text, ConfidenceLevel confidence)
{
    // Update UI with dictation captured
    Debug.Log($"User just said: {text}");

    // Send dictation to Bot
    StartCoroutine(SendMessageToBot(text, botId, botName, "message"));
    StopCapturingAudio();
}

```

9. The following coroutine is called to begin a conversation with the Bot. You will notice that once the conversation call is complete, it will call the **SendMessageToCoroutine()** by passing a series of parameters

that will set the activity to be sent to the Bot Service as an empty message. This is done to prompt the Bot Service to initiate the dialogue.

```
/// <summary>
/// Request a conversation with the Bot Service
/// </summary>
internal IEnumerator StartConversation()
{
    string conversationEndpoint = string.Format("{0}/conversations", botEndpoint);

    WWWForm webForm = new WWWForm();

    using (UnityWebRequest unityWebRequest = UnityWebRequest.Post(conversationEndpoint, webForm))
    {
        unityWebRequest.SetRequestHeader("Authorization", "Bearer " + botSecret);
        unityWebRequest.downloadHandler = new DownloadHandlerBuffer();

        yield return unityWebRequest.SendWebRequest();
        string jsonResponse = unityWebRequest.downloadHandler.text;

        conversation = new ConversationObject();
        conversation = JsonConvert.DeserializeObject<ConversationObject>(jsonResponse);
        Debug.Log($"Start Conversation - Id: {conversation.ConversationId}");
        conversationStarted = true;
    }

    // The following call is necessary to create and inject an activity of type
    //"conversationUpdate" to request a first "introduction" from the Bot Service.
    StartCoroutine(SendMessageToBot("", botId, botName, "conversationUpdate"));
}
```

10. The following coroutine is called to build the activity to be sent to the Bot Service.

```

/// <summary>
/// Send the user message to the Bot Service in form of activity
/// and call for a response
/// </summary>
private IEnumerator SendMessageToBot(string message, string fromId, string fromName, string
activityType)
{
    Debug.Log($"SendMessageCoroutine: {conversation.ConversationId}, message: {message} from Id:
{fromId} from name: {fromName}");

    // Create a new activity here
    Activity activity = new Activity();
    activity.from = new From();
    activity.conversation = new Conversation();
    activity.from.id = fromId;
    activity.from.name = fromName;
    activity.text = message;
    activity.type = activityType;
    activity.channelId = "DirectLineChannelId";
    activity.conversation.id = conversation.ConversationId;

    // Serialize the activity
    string json = JsonConvert.SerializeObject(activity);

    string sendActivityEndpoint = string.Format("{0}/conversations/{1}/activities", botEndpoint,
conversation.ConversationId);

    // Send the activity to the Bot
    using (UnityWebRequest www = new UnityWebRequest(sendActivityEndpoint, "POST"))
    {
        www.uploadHandler = new UploadHandlerRaw(Encoding.UTF8.GetBytes(json));

        www.downloadHandler = new DownloadHandlerBuffer();
        www.SetRequestHeader("Authorization", "Bearer " + botSecret);
        www.SetRequestHeader("Content-Type", "application/json");

        yield return www.SendWebRequest();

        // extrapolate the response Id used to keep track of the conversation
        string jsonResponse = www.downloadHandler.text;
        string cleanedJsonResponse = jsonResponse.Replace("\r\n", string.Empty);
        string responseConvId = cleanedJsonResponse.Substring(10, 30);

        // Request a response from the Bot Service
        StartCoroutine(GetResponseFromBot(activity));
    }
}

```

11. The following coroutine is called to request a response after sending an activity to the Bot Service.

```

/// <summary>
/// Request a response from the Bot by using a previously sent activity
/// </summary>
private IEnumerator GetResponseFromBot(Activity activity)
{
    string getActivityEndpoint = string.Format("{0}/conversations/{1}/activities", botEndpoint,
conversation.ConversationId);

    using (UnityWebRequest unityWebRequest1 = UnityWebRequest.Get(getActivityEndpoint))
    {
        unityWebRequest1.downloadHandler = new DownloadHandlerBuffer();
        unityWebRequest1.SetRequestHeader("Authorization", "Bearer " + botSecret);

        yield return unityWebRequest1.SendWebRequest();

        string jsonResponse = unityWebRequest1.downloadHandler.text;

        ActivitiesRootObject root = new ActivitiesRootObject();
        root = JsonConvert.DeserializeObject<ActivitiesRootObject>(jsonResponse);

        foreach (var act in root.activities)
        {
            Debug.Log($"Bot Response: {act.text}");
            SetBotResponseText(act.text);
        }

        botState = BotState.ReadyToListen;
        botMaterial.color = Color.blue;
    }
}

```

12. The last method to be added to this class, is required to display the message in the scene:

```

/// <summary>
/// Set the UI Response Text of the bot
/// </summary>
internal void SetBotResponseText(string responseString)
{
    SceneOrganiser.Instance.botResponseText.text = responseString;
}

```

NOTE

You may see an error within the Unity Editor Console, about missing the **SceneOrganiser** class. Disregard this message, as you will create this class later in the tutorial.

13. Be sure to save your changes in *Visual Studio* before returning to *Unity*.

Chapter 11 – Create the Interactions class

The class you are going to create now is called **Interactions**. This class is used to detect the HoloLens Tap Input from the user.

If the user taps while looking at the *Bot* object in the scene, and the Bot is ready to listen for voice inputs, the Bot object will change color to **red** and begin listening for voice inputs.

This class inherits from the **GazeInput** class, and so is able to reference the **Start()** method and variables from that class, denoted by the use of **base**.

To create this class:

1. Double-click on the **Scripts** folder, to open it.
2. Right-click inside the **Scripts** folder, click **Create > C# Script**. Name the script **Interactions**.
3. Double-click on the new script to open it with Visual Studio.
4. Update the namespaces and the class inheritance to be the same as the following, at the top of the **Interactions** class:

```
using UnityEngine.XR.WSA.Input;

public class Interactions : GazeInput
{
```

5. Inside the **Interactions** class add the following variable:

```
/// <summary>
/// Allows input recognition with the HoloLens
/// </summary>
private GestureRecognizer _gestureRecognizer;
```

6. Then add the **Start()** method:

```
/// <summary>
/// Called on initialization, after Awake
/// </summary>
internal override void Start()
{
    base.Start();

    //Register the application to recognize HoloLens user inputs
    _gestureRecognizer = new GestureRecognizer();
    _gestureRecognizer.SetRecognizableGestures(GestureSettings.Tap);
    _gestureRecognizer.Tapped += GestureRecognizer_Tapped;
    _gestureRecognizer.StartCapturingGestures();
}
```

7. Add the handler that will be triggered when the user performs the tap gesture in front of the HoloLens camera

```

/// <summary>
/// Detects the User Tap Input
/// </summary>
private void GestureRecognizer_Tapped(TappedEventArgs obj)
{
    // Ensure the bot is being gazed upon.
    if(base.FocusedObject != null)
    {
        // If the user is tapping on Bot and the Bot is ready to listen
        if (base.FocusedObject.name == "Bot" && Bot.Instance.botState == Bot.BotState.ReadyToListen)
        {
            // If a conversation has not started yet, request one
            if(Bot.Instance.conversationStarted)
            {
                Bot.Instance.SetBotResponseText("Listening...");
                Bot.Instance.StartCapturingAudio();
            }
            else
            {
                Bot.Instance.SetBotResponseText("Requesting Conversation...");
                StartCoroutine(Bot.Instance.StartConversation());
            }
        }
    }
}

```

- Be sure to save your changes in *Visual Studio* before returning to *Unity*.

Chapter 12 – Create the SceneOrganiser class

The last class required in this Lab is called **SceneOrganiser**. This class will setup the scene programmatically, by adding components and scripts to the Main Camera, and creating the appropriate objects in the scene.

To create this class:

- Double-click on the **Scripts** folder, to open it.
- Right-click inside the **Scripts** folder, click **Create > C# Script**. Name the script **SceneOrganiser**.
- Double-click on the new script to open it with Visual Studio.
- Inside the **SceneOrganiser** class add the following variables:

```

/// <summary>
/// Static instance of this class
/// </summary>
public static SceneOrganiser Instance;

/// <summary>
/// The 3D text representing the Bot response
/// </summary>
internal TextMesh botResponseText;

```

- Then add the **Awake()** and **Start()** methods:

```

/// <summary>
/// Called on Initialization
/// </summary>
private void Awake()
{
    Instance = this;
}

/// <summary>
/// Called immediately after Awake method
/// </summary>
void Start ()
{
    // Add the GazeInput class to this object
    gameObject.AddComponent<GazeInput>();

    // Add the Interactions class to this object
    gameObject.AddComponent<Interactions>();

    // Create the Bot in the scene
    CreateBotInScene();
}

```

6. Add the following method, responsible for creating the Bot object in the scene and setting up the parameters and components:

```

/// <summary>
/// Create the Sign In button object in the scene
/// and sets its properties
/// </summary>
private void CreateBotInScene()
{
    GameObject botObjInScene = GameObject.CreatePrimitive(PrimitiveType.Sphere);
    botObjInScene.name = "Bot";

    // Add the Bot class to the Bot GameObject
    botObjInScene.AddComponent<Bot>();

    // Create the Bot UI
    botResponseText = CreateBotResponseText();

    // Set properties of Bot GameObject
    Bot.Instance.botMaterial = new Material(Shader.Find("Diffuse"));
    botObjInScene.GetComponent<Renderer>().material = Bot.Instance.botMaterial;
    Bot.Instance.botMaterial.color = Color.blue;
    botObjInScene.transform.position = new Vector3(0f, 2f, 10f);
    botObjInScene.tag = "BotTag";
}

```

7. Add the following method, responsible for creating the UI object in the scene, representing the responses from the Bot:

```

/// <summary>
/// Spawns cursor for the Main Camera
/// </summary>
private TextMesh CreateBotResponseText()
{
    // Create a sphere as new cursor
    GameObject textObject = new GameObject();
    textObject.transform.parent = Bot.Instance.transform;
    textObject.transform.localPosition = new Vector3(0,1,0);

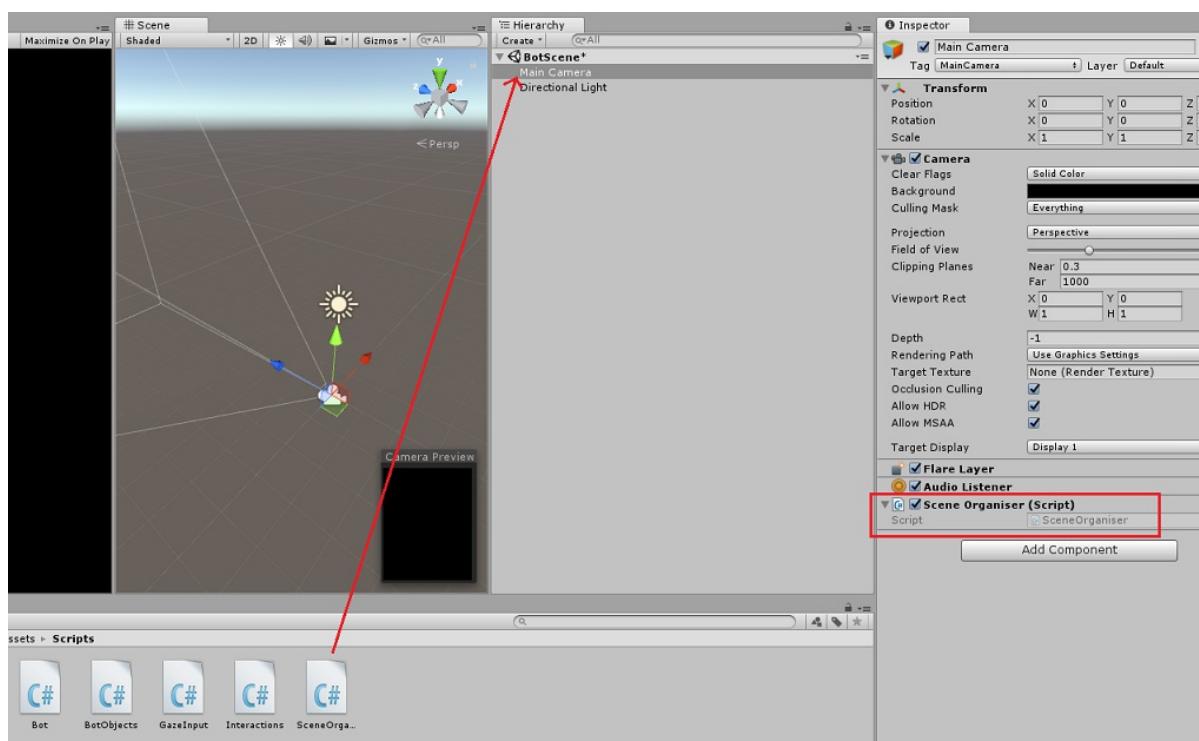
    // Resize the new cursor
    textObject.transform.localScale = new Vector3(0.1f, 0.1f, 0.1f);

    // Creating the text of the Label
    TextMesh textMesh = textObject.AddComponent<TextMesh>();
    textMesh.anchor = TextAnchor.MiddleCenter;
    textMesh.alignment = TextAlignment.Center;
    textMesh.fontSize = 50;
    textMesh.text = "Hi there, tap on me and I will start listening. ";

    return textMesh;
}

```

8. Be sure to save your changes in *Visual Studio* before returning to *Unity*.
9. In the Unity Editor, drag the **SceneOrganiser** script from the Scripts folder to the Main Camera. The Scene Organiser component should now appear on the Main Camera object, as shown in the image below.



Chapter 13 – Before building

To perform a thorough test of your application you will need to sideload it onto your HoloLens. Before you do, ensure that:

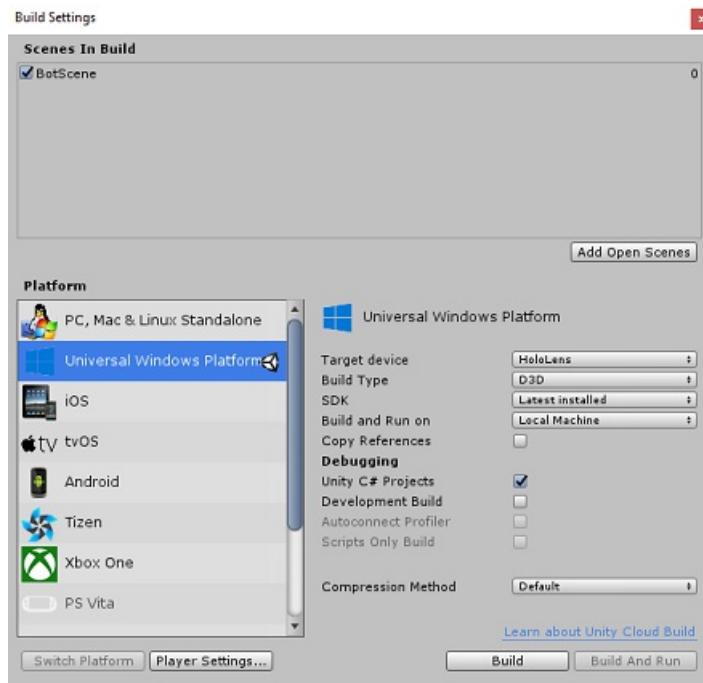
- All the settings mentioned in the [Chapter 4](#) are set correctly.
- The script **SceneOrganiser** is attached to the **Main Camera** object.
- In the **Bot** class, make sure you have inserted your **Bot Secret Key** into the **botSecret** variable.

Chapter 14 – Build and Sideload to the HoloLens

Everything needed for the Unity section of this project has now been completed, so it is time to build it from Unity.

1. Navigate to **Build Settings**, **File > Build Settings....**

2. From the **Build Settings** window, click **Build**.



3. If not already, tick **Unity C# Projects**.

4. Click **Build**. Unity will launch a **File Explorer** window, where you need to create and then select a folder to build the app into. Create that folder now, and name it **App**. Then with the **App** folder selected, click **Select Folder**.

5. Unity will begin building your project to the **App** folder.

6. Once Unity has finished building (it might take some time), it will open a **File Explorer** window at the location of your build (check your task bar, as it may not always appear above your windows, but will notify you of the addition of a new window).

Chapter 15 – Deploy to HoloLens

To deploy on HoloLens:

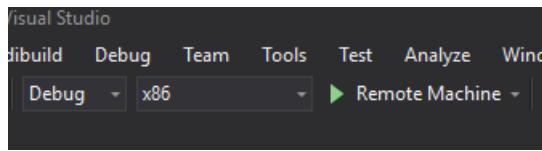
1. You will need the IP Address of your HoloLens (for Remote Deploy), and to ensure your HoloLens is in **Developer Mode**. To do this:

- a. Whilst wearing your HoloLens, open the **Settings**.
- b. Go to **Network & Internet > Wi-Fi > Advanced Options**
- c. Note the **IPv4** address.
- d. Next, navigate back to **Settings**, and then to **Update & Security > For Developers**
- e. Set Developer Mode On.

2. Navigate to your new Unity build (the **App** folder) and open the solution file with **Visual Studio**.

3. In the **Solution Configuration** select **Debug**.

4. In the **Solution Platform**, select **x86, Remote Machine**.



5. Go to the **Build menu** and click on **Deploy Solution**, to sideload the application to your HoloLens.
6. Your app should now appear in the list of installed apps on your HoloLens, ready to be launched!

NOTE

To deploy to immersive headset, set the **Solution Platform** to *Local Machine*, and set the **Configuration** to *Debug*, with *x86* as the **Platform**. Then deploy to the local machine, using the **Build menu**, selecting *Deploy Solution*.

Chapter 16 – Using the application on the HoloLens

- Once you have launched the application, you will see the Bot as a blue sphere in front of you.
- Use the **Tap Gesture** while you are gazing at the sphere to initiate a conversation.
- Wait for the conversation to start (The UI will display a message when it happens). Once you receive the introductory message from the Bot, tap again on the Bot so it will turn red and begin to listen to your voice.
- Once you stop talking, your application will send your message to the Bot and you will promptly receive a response that will be displayed in the UI.
- Repeat the process to send more messages to your Bot (you have to tap each time you want to send a message).

This conversation demonstrates how the Bot can retain information (your name), whilst also providing known information (such as the items that are stocked).

Some questions to ask the Bot:

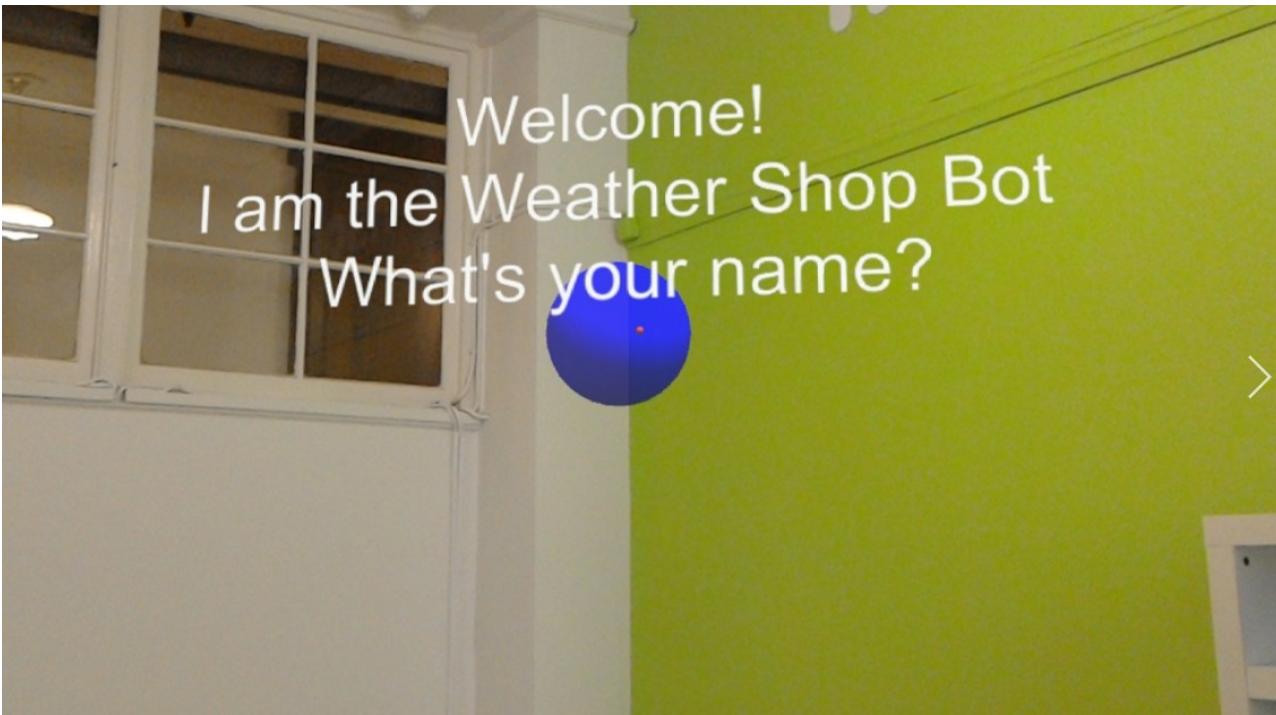
what do you sell?

how much are umbrellas?

how much are raincoats?

Your finished Web App Bot (v4) application

Congratulations, you built a mixed reality app that leverages the Azure Web App Bot, Microsoft Bot Framework v4.



Bonus exercises

Exercise 1

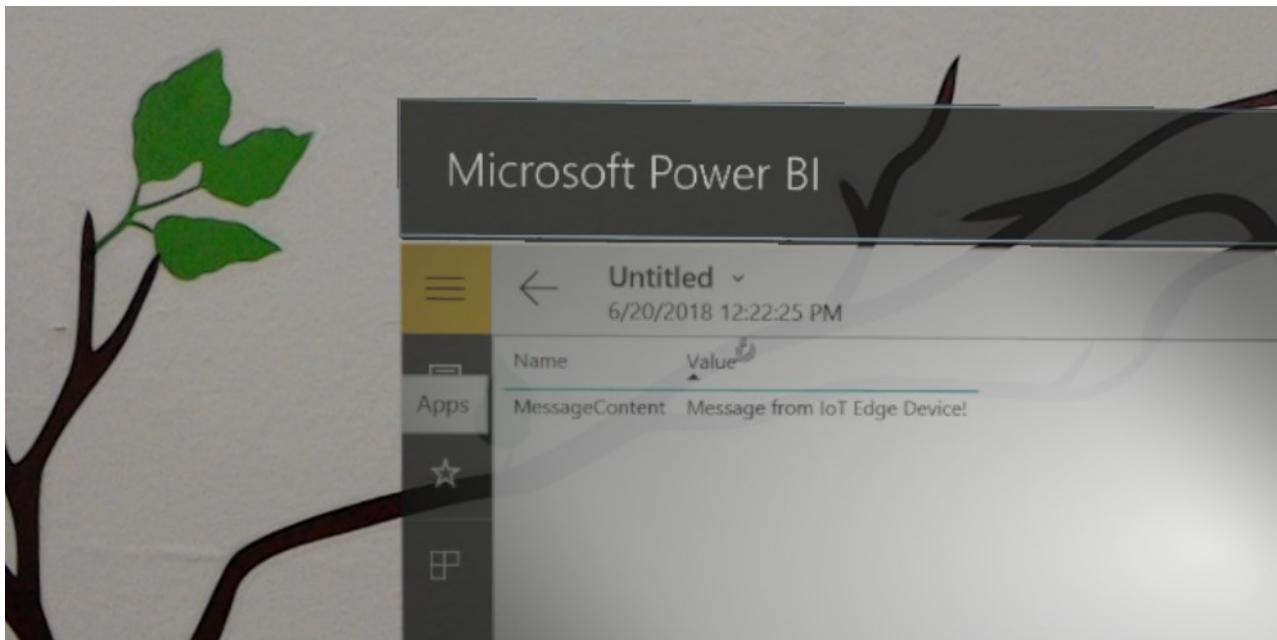
The conversation structure in this Lab is very basic. Use Microsoft LUIS to give your bot natural language understanding capabilities.

Exercise 2

This example does not include terminating a conversation and restarting a new one. To make the Bot feature complete, try to implement closure to the conversation.

MR and Azure 313: IoT Hub Service

11/6/2018 • 38 minutes to read • [Edit Online](#)



In this course, you will learn how to implement an **Azure IoT Hub Service** on a virtual machine running the Ubuntu 16.4 operating system. An **Azure Function App** will then be used to receive messages from your Ubuntu VM, and store the result within an **Azure Table Service**. You will then be able to view this data using **Power BI** on Microsoft HoloLens or immersive (VR) headset.

The content of this course *is applicable* to IoT Edge devices, though for the purpose of this course, the focus will be on a virtual machine environment, so that access to a physical Edge device is not necessary.

By completing this course, you will learn to:

- Deploy an **IoT Edge module** to a Virtual Machine (Ubuntu 16 OS), which will represent your IoT device.
- Add an **Azure Custom Vision Tensorflow Model** to the Edge module, with code that will analyze images stored in the container.
- Set up the module to send the analysis result message back to your **IoT Hub Service**.
- Use an **Azure Function App** to store the message within an **Azure Table**.
- Set up **Power BI** to collect the stored message and create a report.
- Visualize your IoT message data within **Power BI**.

The Services you will use include:

- **Azure IoT Hub** is a Microsoft Azure Service which allows developers to connect, monitor, and manage, IoT assets. For more information, visit the [Azure IoT Hub Service page](#).
- **Azure Container Registry** is a Microsoft Azure Service which allows developers to store container images, for various types of containers. For more information, visit the [Azure Container Registry Service page](#).
- **Azure Function App** is a Microsoft Azure Service, which allows developers to run small pieces of code, 'functions', in Azure. This provides a way to delegate work to the cloud, rather than your local application, which can have many benefits. **Azure Functions** supports several development languages, including C#, F#, Node.js, Java, and PHP. For more information, visit the [Azure Functions page](#).

- **Azure Storage: Tables** is a Microsoft Azure Service, which allows developers to store structured, non-SQL, data in the cloud, making it easily accessible anywhere. The Service boasts a schema-less design, allowing for the evolution of tables as needed, and thus is very flexible. For more information, visit the [Azure Tables page](#)

This course will teach you how to setup and use the IoT Hub Service, and then visualize a response provided by a device. It will be up to you to apply these concepts to a custom IoT Hub Service setup, which you might be building.

Device support

COURSE	HOLOLENS	IMMERSIVE HEADSETS
MR and Azure 313: IoT Hub Service	✓ <input type="checkbox"/>	✓ <input type="checkbox"/>

Prerequisites

For the most up-to-date prerequisites for developing with mixed reality, including with the Microsoft HoloLens, visit the [Install the tools](#) article.

NOTE

This tutorial is designed for developers who have basic experience with Python. Please also be aware that the prerequisites and written instructions within this document represent what has been tested and verified at the time of writing (July 2018). You are free to use the latest software, as listed within the [install the tools](#) article, though it should not be assumed that the information in this course will perfectly match what you will find in newer software than that listed below.

The following hardware and software is required:

- Windows 10 Fall Creators Update (or later), **Developer Mode enabled**

WARNING

You cannot run a Virtual Machine using Hyper-V on Windows 10 Home Edition.

- Windows 10 SDK (latest version)
- A HoloLens, **Developer Mode enabled**
- Visual Studio 2017.15.4 (Only used to access the Azure Cloud Explorer)
- Internet Access for Azure, and for IoT Hub Service. For more information, please follow this [link to IoT Hub Service page](#)
- A machine learning model. If you do not have your own ready to use model, [you can use the model provided with this course](#).
- **Hyper-V** software enabled on your Windows 10 development machine.
- A Virtual Machine running Ubuntu (16.4 or 18.4), running on your development machine or alternatively you can use a separate computer running Linux (Ubuntu 16.4 or 18.4). You can find more information on how to create a VM on Windows using Hyper-V in the "Before you Start" chapter. (<https://docs.microsoft.com/virtualization/hyper-v-on-windows/quick-start/quick-create-virtual-machine>).

Before you start

1. Set up and test your HoloLens. If you need support setting up your HoloLens, [make sure to visit the HoloLens setup article](#).
2. It is a good idea to perform **Calibration** and **Sensor Tuning** when beginning developing a new HoloLens app (sometimes it can help to perform those tasks for each user).

For help on Calibration, please follow this [link to the HoloLens Calibration article](#).

For help on Sensor Tuning, please follow this [link to the HoloLens Sensor Tuning article](#).

3. Set up your **Ubuntu Virtual Machine** using **Hyper-V**. The following resources will help you with the process.
 - a. First, follow this link to [download the Ubuntu 16.04.4 LTS \(Xenial Xerus\) ISO](#). Select the **64-bit PC (AMD64) desktop image**.
 - b. Make sure **Hyper-V** is enabled on your Windows 10 machine. You can follow this link for guidance on [installing and enabling Hyper-V on Windows 10](#).
 - c. Start Hyper-V and create a new Ubuntu VM. You can follow this link for a [step by step guide on how to create a VM with Hyper-V](#). When requested to "**Install an operating system from a bootable image file**", select the **Ubuntu ISO** you have download earlier.

NOTE

Using **Hyper-V Quick Create** is not suggested.

Chapter 1 - Retrieve the Custom Vision model

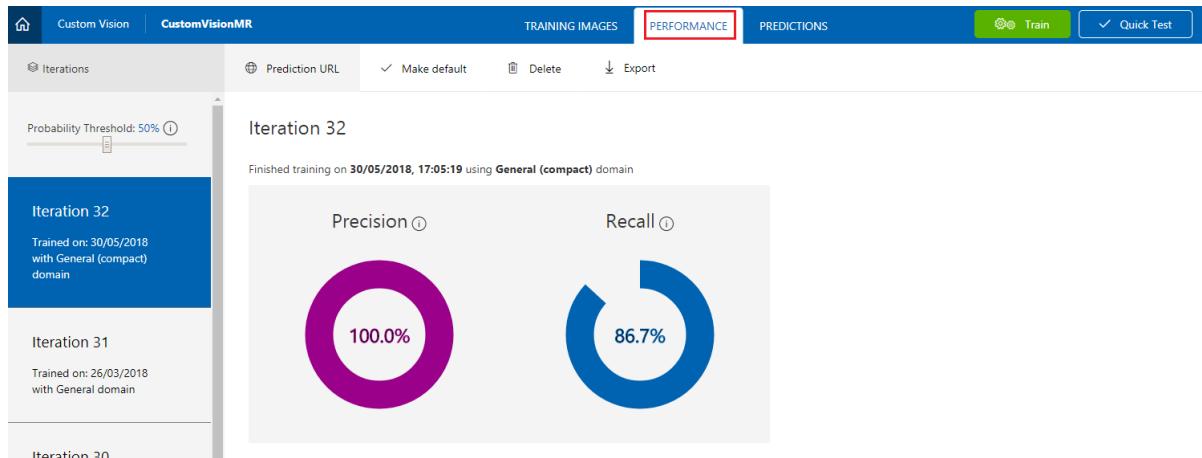
With this course you will have access to a [pre-built Custom Vision model](#) that detects keyboards and mice from images. If you use this, proceed to [Chapter 2](#).

However, you can follow these steps if you wish to use your own Custom Vision model:

1. In your **Custom Vision Project** go to the **Performance** tab.

WARNING

Your model must use a *compact* domain, to export the model. You can change your models domain in the settings for your project.



2. Select the **Iteration** you want to export and click on **Export**. A blade will appear.

Screenshot of the Custom Vision Performance blade showing Iteration 32 details. The 'Export' button is highlighted with a red box.

Iteration 32

Probability Threshold: 50% ⓘ

Precision ⓘ 100.0%

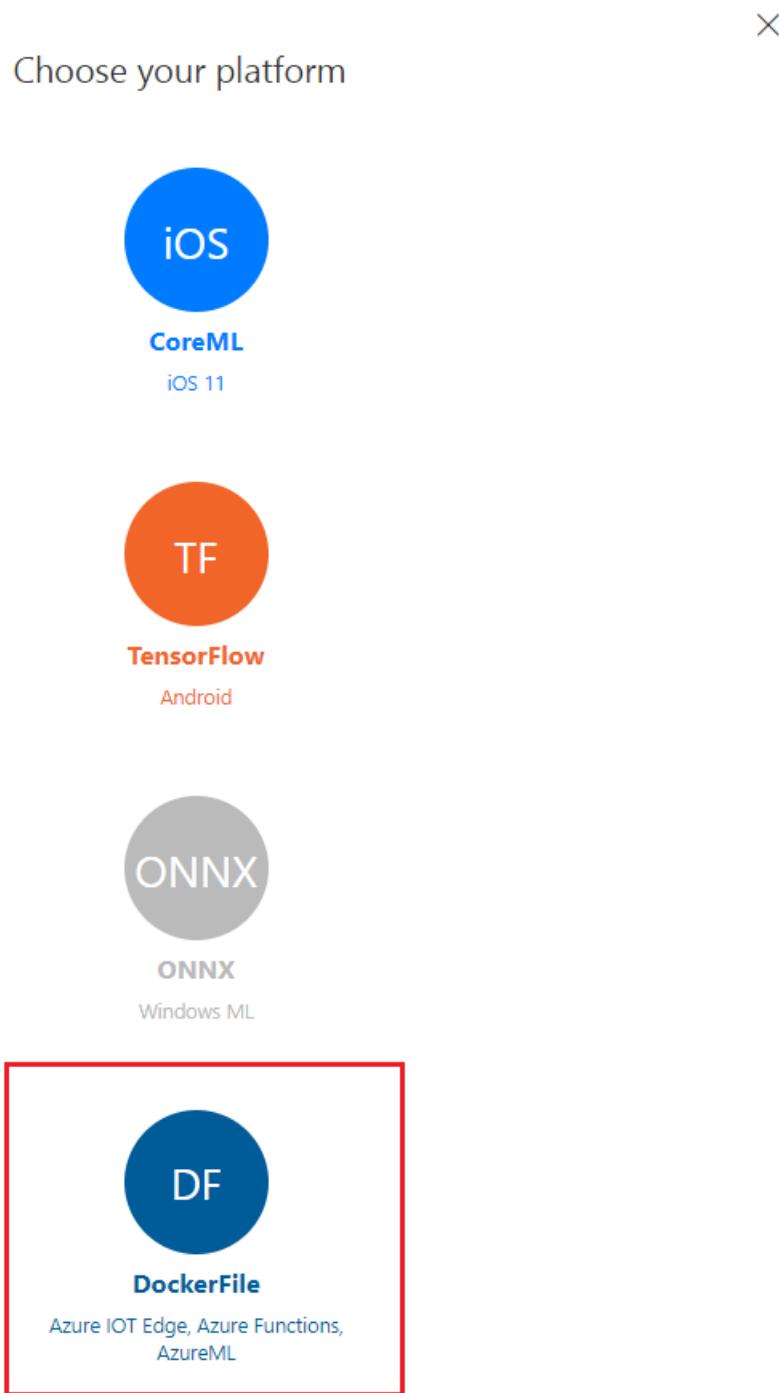
Recall ⓘ 86.7%

Iteration 32
Trained on: 30/05/2018 with General (compact) domain

Iteration 31
Trained on: 26/03/2018 with General domain

Iteration 30

3. In the blade click **Docker File**.



4. Click **Linux** in the drop-down menu and then click on **Download**.



Choose your platform



DockerFile

Linux

Download

How to use?

[Docker documentation](#)

[Licenses](#)

5. Unzip the content. You will use it later in this course.

Chapter 2 - The Container Registry Service

The **Container Registry Service** is the repository used to host your containers.

The **IoT Hub Service** that you will build and use in this course, refers to **Container Registry Service** to obtain the containers to deploy in your Edge Device.

1. First, follow this [link to the Azure Portal](#), and login with your credentials.
2. Go to **Create a resource** and look for **Container Registry**.

NAME	PUBLISHER	CATEGORY
Container Registry	Microsoft	Compute
Aqua Container Security 2.5 for Azure	Aqua Security	Compute
Web App for Containers	Microsoft	Web

3. Click on **Create**.



Container Registry
Microsoft

Azure Container Registry is a private registry for hosting container images. Using the Azure Container Registry, you can store Docker-formatted images for all types of container deployments. Azure Container Registry integrates well with orchestrators hosted in Azure Container Service, including Docker Swarm, DC/OS, and Kubernetes. Users can benefit from using familiar tooling capable of working with the open source Docker Registry v2.

Use Azure Container Registry to:

- Store and manage container images across all types of Azure deployments
- Use familiar, open-source Docker command line interface (CLI) tools
- Keep container images near deployments to reduce latency and costs
- Simplify registry access management with Azure Active Directory
- Maintain Windows and Linux container images in a single Docker registry

 Save for later

PUBLISHER

Microsoft

[Learn more](#)

USEFUL LINKS

[Documentation](#)

[Pricing details](#)

[Create](#)

4. Set the Service setup parameters:

- a. Insert a name for your project, In this example its called **IoTCRegistry**.
- b. Choose a **Resource Group** or create a new one. A resource group provides a way to monitor, control access, provision, and manage, billing for a collection of Azure assets. It is recommended to keep all the Azure Services associated with a single project (e.g. such as these courses) under a common resource group).
- c. Set the location of the Service.
- d. Set **Admin user** to **Enable**.
- e. Set **SKU** to **Basic**.

Home > New > Container Registry > Create container registry

Create container registry

*** Registry name**
IoTCRegistry .azurecr.io

*** Subscription**
Main Subscription

*** Resource group**
 Create new Use existing
Azure_for_Unity

*** Location**
West US

*** Admin user** [i](#)
 Enable Disable

*** SKU** [i](#)
Basic

Pin to dashboard

Create [Automation options](#)

5. Click **Create** and wait for the Services to be created.
6. Once the notification pops up informing you of the successful creation of the *Container Registry*, click on **Go to resource** to be redirected to your Service page.



7. In the *Container Registry* Service page, click on **Access keys**.
8. Take note (you could use your Notepad) of the following parameters:
 - a. **Login Server**
 - b. **Username**
 - c. **Password**

Chapter 3 - The IoT Hub Service

Now you will begin the creation and setup of your **IoT Hub Service**.

1. If not already signed in, log into the [Azure Portal](#).
2. Once logged in, click on **Create a resource** in the top left corner, and search for **IoT Hub**, and click **Enter**.

NAME	PUBLISHER	CATEGORY
IoT Hub	Microsoft	Internet of Things
IoT Hub Device Provisioning Service	Microsoft	Internet of Things
The Identity Hub	U2U Consult NV/SA	Security

3. The new page will provide a description of the **Storage account** Service. At the bottom left of this prompt, click the **Create** button, to create an instance of this Service.

IoT Hub

Microsoft

Simultaneously support millions of connected devices—whether they run Windows, Linux, or real-time operating systems. Then monitor performance and send commands to accelerate your digital transformation.

[Save for later](#)

PUBLISHER

Microsoft

[Documentation](#)
[Device management](#)

USEFUL LINKS

[Service overview](#)
[Pricing and scale details](#)
[Learn more about Azure IoT Hub](#)

[Create](#)

4. Once you have clicked on **Create**, a panel will appear:

- Choose a **Resource Group** or create a new one. A resource group provides a way to monitor, control access, provision and manage billing for a collection of Azure assets. It is recommended to keep all the Azure Services associated with a single project (e.g. such as these courses) under a common resource group).

If you wish to read more about Azure Resource Groups, please follow this [link on how to manage a Resource Group](#).

- Select an appropriate **Location** (Use the same location across all the Services you create in this course).
- Insert your desired **Name** for this Service instance.

5. On the bottom of the page click on **Next: Size and scale**.

Home > New > Marketplace > Everything > IoT Hub > IoT hub

IoT hub

Microsoft

Basics **Size and scale** Review + create

Create an IoT Hub to help you connect, monitor, and manage billions of your IoT assets. [Learn More](#)

PROJECT DETAILS

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

* Subscription <small>i</small>	Main Subscription
* Resource Group <small>i</small>	<input type="radio"/> Create new <input checked="" type="radio"/> Use existing Azure_for_Unity
* Region <small>i</small>	West US
* IoT Hub Name <small>i</small>	MRIoTHubEdge

[Review + create](#) [Next: Size and scale »](#) Automation options

6. In this page, select your **Pricing and scale tier** (if this is your first IoT Hub Service instance, a free tier should be available to you).
7. Click on **Review + Create**.

[Basics](#) [Size and scale](#) [Review + create](#)

Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send. [Learn more](#)

SCALE TIER AND UNITS

* Pricing and scale tier [?](#)

F1: Free tier

[Learn how to choose the right IoT Hub tier for your solution](#)

Number of F1 IoT Hub units [?](#)



1

This determines your IoT Hub scale capability and can be changed as your need increases.

Pricing and scale tier [?](#)

F1

Device-to-cloud-messages [?](#)

Enabled

Messages per day [?](#)

8,000

Message routing [?](#)

Enabled

Cost per month

0.00 AUD

Cloud-to-device commands [?](#)

Enabled

IoT Edge [?](#)

Enabled

Device management [?](#)

Enabled

[Advanced Settings](#)

[Review + create](#)

[« Previous: Basics](#)

[Automation options](#)

8. Review your settings and click on **Create**.

Home > New > Marketplace > Everything > IoT Hub > IoT hub

IoT hub

Microsoft

Basics Size and scale Review + create

BASICS

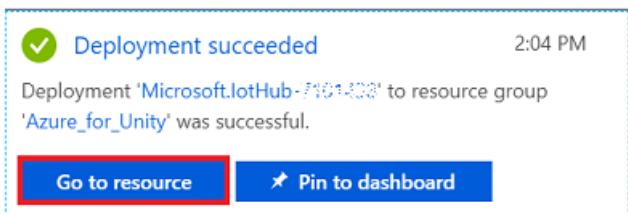
Subscription ⓘ	Main Subscription
Resource Group ⓘ	Azure_for_Unity
Region ⓘ	West US
IoT Hub Name ⓘ	MRIoTHubEdge

SIZE AND SCALE

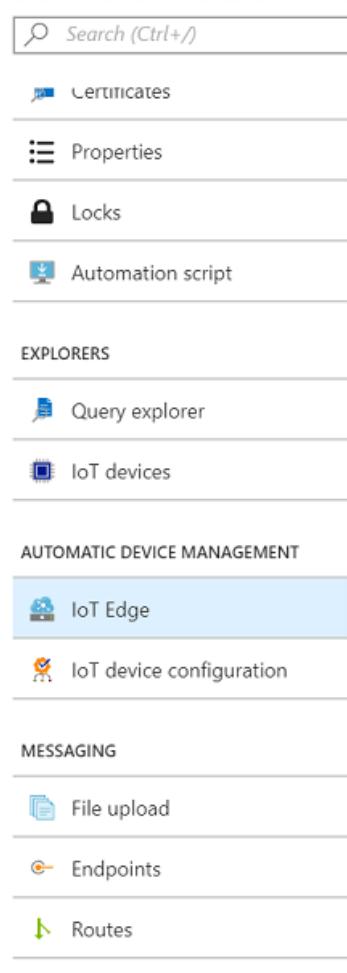
Pricing and scale tier ⓘ	F1
Number of F1 IoT Hub units ⓘ	1
Messages per day ⓘ	8,000
Cost per month	0.00 AUD

Create « Previous: Size and scale Automation options

- Once the notification pops up informing you of the successful creation of the *IoT Hub* Service, click on **Go to resource** to be redirected to your Service page.



- Scroll the side panel on the left until you see *Automatic Device Management*, then click on **IoT Edge**.



11. In the window that appears to the right, click on **Add IoT Edge Device**. A blade will appear to the right.
12. In the blade, provide your new device a **Device ID** (a name of your choice). Then, click **Save**. The *Primary* and *Secondary Keys* will auto generate, if you have **Auto Generate** ticked.

Add Device

Learn more about creating devices 

* Device ID  EdgeDevice01 

Authentication Type  Symmetric Key

* Primary Key  Enter your primary key here

* Secondary Key  Enter your secondary key here

Auto Generate Keys 

Connect device to IoT Hub   



13. You will navigate back to the *IoT Edge Devices* section, where your new device will be listed. Click on your new device (outlined in red in the below image).

 Add IoT Edge Device  Add IoT Edge Deployment  Refresh  Delete

Azure IoT Edge enables cloud-driven deployment of Azure services and solution-specific code to on-premise devices. IoT Edge devices can aggregate data from other devices to perform computing and analytics before the data is sent to the cloud. From this page, you can create and manage IoT Edge devices and deployments. 

[IoT Edge Devices](#) [IoT Edge Deployments](#)

 IoT Edge Devices

IoT Edge devices have the IoT Edge runtime installed and are flagged as "IoT Edge device" in the device details. Each IoT Hub supports up to 1000 IoT Edge devices. 

Query 
SELECT * FROM devices WHERE
optional (e.g. tags.location='US')



DEVICE ID	RUNTIME RESPONSE	MODULE COUNT	CONNECTED CLIENT COUNT	DEPLOYMENT COUNT
<input checked="" type="checkbox"/> EdgeDevice01	N/A	0	0	0

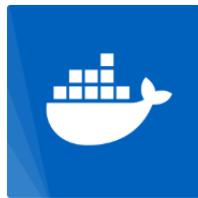
14. On the *Device Details* page that appears, take a copy of the **Connection String** (primary key).

15. Go back to the panel on the left, and click *Shared access policies*, to open it.
16. On the page that appears, click **iothubowner**, and a blade will appear to the right of the screen.
17. Take note (on your Notepad) of the **Connection string** (primary key), for later use when setting the *ConnectionString* to your device.

Chapter 4 - Setting up the development environment

In order to create and deploy modules for *IoT Hub Edge*, you will require the following components installed on your development machine running Windows 10:

1. [Docker for Windows](#), it will ask you to create an account to be able to download.



Docker Community Edition for Windows

By [Docker](#)

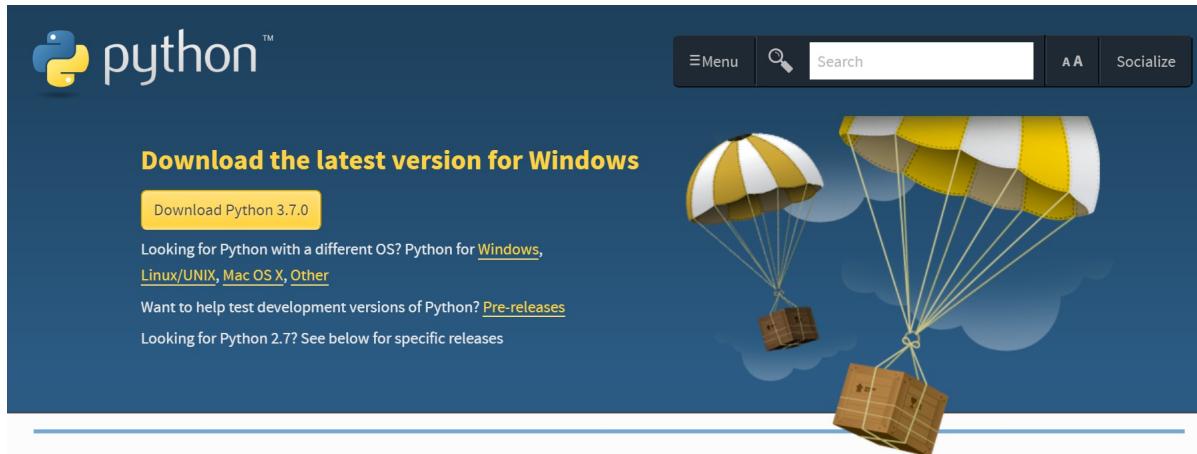
The fastest and easiest way to get started with Docker on Windows

Edition Windows x86-64

IMPORTANT

Docker requires *Windows 10 PRO, Enterprise 14393, or Windows Server 2016 RTM*, to run. If you are running other versions of Windows 10, you can try installing Docker using the [Docker Toolbox](#).

2. Python 3.6.



3. Visual Studio Code (also known as VS Code).

A screenshot of the Visual Studio Code download page. At the top, there's a navigation bar with icons for 'Visual Studio Code', 'Docs', 'Updates', 'Blog', 'Community', 'Extensions', and 'FAQ'. To the right of the navigation bar is a search bar with a magnifying glass icon and the text 'Search Docs'. Next to the search bar is a green 'Download' button with a downward arrow icon. Below the navigation bar, a message says 'Version 1.25 is now available! Read about the new features and fixes from June.' In the center, the text 'Download Visual Studio Code' is displayed above the subtext 'Free and open source. Integrated Git, debugging and extensions.'. Below this, there are three operating system logos: Windows, Linux (Tux), and Mac. Under each logo is a download button. The Windows button says 'Windows' with 'Windows 7, 8, 10' underneath, and '.zip | 32 bit versions' below it. The Linux button says '.deb' with 'Debian, Ubuntu' underneath, and '.tar.gz | 32 bit versions' below it. The Mac button says 'Mac' with 'macOS 10.9+' underneath.

After installing the software mentioned above, you will need to restart your machine.

Chapter 5 - Setting up the Ubuntu environment

Now you can move on to setting up your device **running Ubuntu OS**. Follow the steps below, to install the necessary software, to deploy your containers on your board:

IMPORTANT

You should always precede the terminal commands with **sudo** to run as admin user. i.e:

```
sudo docker \<option> \<command> \<argument>
```

1. Open the **Ubuntu Terminal**, and use the following command to install **pip**:

[!HINT] You can open *Terminal* very easily through using the keyboard shortcut: **Ctrl + Alt + T**.

```
sudo apt-get install python-pip
```

2. Throughout this Chapter, you may be prompted, by *Terminal*, for permission to use your device storage, and for you to input **y/n** (yes or no), type '**y**', and then press the **Enter** key, to accept.
3. Once that command has completed, use the following command to install **curl**:

```
sudo apt install curl
```

4. Once **pip** and **curl** are installed, use the following command to install the **IoT Edge runtime**, this is necessary to deploy and control the modules on your board:

```
curl https://packages.microsoft.com/config/ubuntu/16.04/prod.list > ./microsoft-prod.list  
sudo cp ./microsoft-prod.list /etc/apt/sources.list.d/  
  
curl https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor > microsoft.gpg  
  
sudo cp ./microsoft.gpg /etc/apt/trusted.gpg.d/  
  
sudo apt-get update  
  
sudo apt-get install moby-engine  
  
sudo apt-get install moby-cli  
  
sudo apt-get update  
  
sudo apt-get install iotedge
```

5. At this point you will be prompted to open up the *runtime config file*, to insert the **Device Connection String**, that you noted down (in your Notepad), when creating the **IoT Hub Service** ([at step 14, of Chapter 3](#)). Run the following line on the terminal to open that file:

```
sudo nano /etc/iotedge/config.yaml
```

6. The **config.yaml** file will be displayed, ready for you to edit:

WARNING

When this file opens, it may be somewhat confusing. You will be text editing this file, within the *Terminal* itself.

- a. Use the arrow keys on your keyboard to scroll down (you will need to scroll down a little way), to reach the line containing":
 "**<ADD DEVICE CONNECTION STRING HERE>**".
- b. Substitute line, **including the brackets**, with the **Device Connection String** you have noted earlier.
7. With your Connection String in place, on your keyboard, press the **Ctrl-X** keys to save the file. It will ask you to confirm by typing **Y**. Then, press the **Enter** key, to confirm. You will go back to the regular *Terminal*.
8. Once these commands have all run successfully, you will have installed the **IoT Edge Runtime**. Once initialized, the runtime will start on its own every time the device is powered up, and will sit in the background, waiting for modules to be deployed from the **IoT Hub Service**.

9. Run the following command line to initialize the *IoT Edge Runtime*:

```
sudo systemctl restart iotedge
```

IMPORTANT

If you make changes to your .yaml file, or the above setup, you will need to run the above restart line again, within *Terminal*.

10. Check the *IoT Edge Runtime* status by running the following command line. The runtime should appear with the status **active (running)** in green text.

```
sudo systemctl status iotedge
```

11. Press the **Ctrl-C** keys, to exit the status page. You can verify that the *IoT Edge Runtime* is pulling the containers correctly by typing the following command:

```
sudo docker ps
```

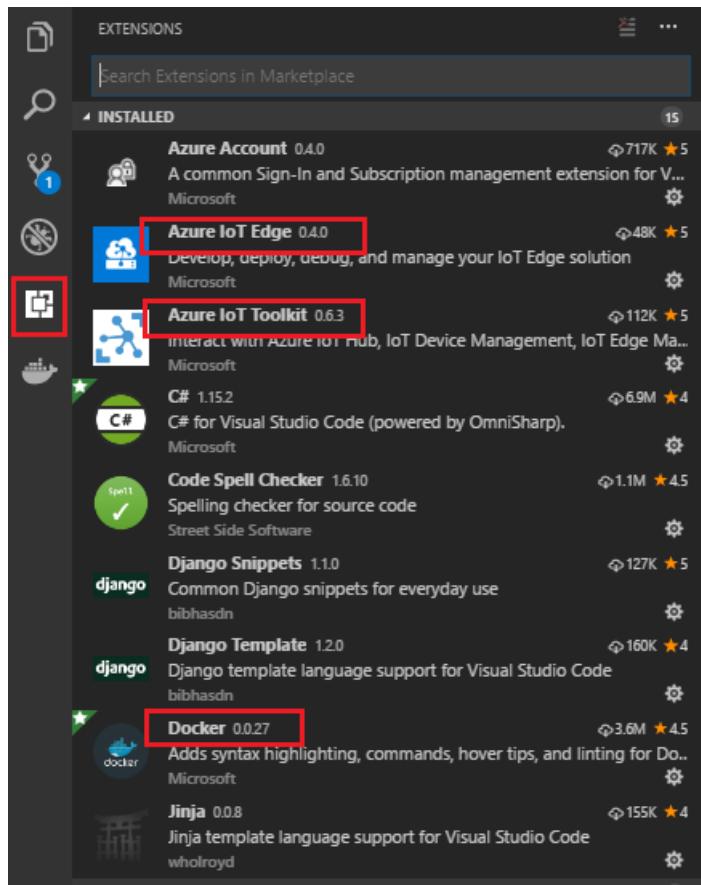
12. A list with two (2) containers should appear. These are the default modules that are automatically created by the IoT Hub Service (edgeAgent and edgeHub). Once you create and deploy your own modules, they will appear in this list, underneath the default ones.

Chapter 6 - Install the extensions

IMPORTANT

The next few Chapters (6-9) are to be performed on your Windows 10 machine.

1. Open **VS Code**.
2. Click on the **Extensions** (square) button on the left bar of VS Code, to open the **Extensions panel**.
3. Search for, and install, the following extensions (as shown in the image below):
 - a. Azure IoT Edge
 - b. Azure IoT Toolkit
 - c. Docker



4. Once the extensions are installed, close and re-open VS Code.
5. With VS Code open once more, navigate to **View > Integrated terminal**.
6. You will now install **Cookiecutter**. In the terminal run the following bash command:


```
pip install --upgrade --user cookiecutter
```

[!HINT] If you have trouble with this command:

 1. Restart VS Code, and/ or your computer.
 2. It might be necessary to switch the **VS Code Terminal** to the one you have been using to install Python, i.e. **Powershell** (especially in case the Python environment was already installed on your machine). With the Terminal open, you will find the drop down menu on the right side of the Terminal.
 3. Make sure the **Python** installation path is added as **Environment Variable** on your machine. Cookiecutter should be part of the same location path. Please follow this [link for more information on Environment Variables](#).
7. Once **Cookiecutter** has finished installing, you should restart your machine, so that **Cookiecutter** is recognized as a command, within your System's environment.

Chapter 7 - Create your container solution

At this point, you need to create the container, with the module, to be pushed into the *Container Registry*. Once you have pushed your container, you will use the *IoT Hub Edge* Service to deploy it to your device, which is running the *IoT Edge runtime*.

1. From VS Code, click **View > Command palette**.
2. In the palette, search and run **Azure IoT Edge: New IoT Edge Solution**.
3. Browse into a location where you want to create your solution. Press the **Enter** key, to accept the location.
4. Give a name to your solution. Press the **Enter** key, to confirm your provided name.
5. Now you will be prompted to choose the template framework for your solution. Click **Python Module**. Press the **Enter** key, to confirm this choice.
6. Give a name to your module. Press the **Enter** key, to confirm the name of your module. Make sure to take a note (with your Notepad) of the module name, as it is used later.
7. You will notice a pre-built *Docker Image Repository* address will appear on the palette. It will look like:

localhost:5000/-THE NAME OF YOUR MODULE-

8. Delete **localhost:5000**, and in its place insert the *Container Registry Login Server* address, which you noted when creating the **Container Registry Service** ([in step 8, of Chapter 2](#)). Press the **Enter** key, to confirm the address.
9. At this point, the solution containing the template for your Python module will be created and its structure will be displayed in the **Explore Tab**, of VS Code, on the left side of the screen. If the **Explore Tab** is not open, you can open it by clicking the top-most button, in the bar on the left.



10. The last step for this Chapter, is to click and open the **.env file**, from within the **Explore Tab**, and add your *Container Registry username* and **password**. This file is ignored by git, but on building the container, will set the credentials to access the **Container Registry Service**.

```

1 CONTAINER_REGISTRY_USERNAME=registrycm1iotedge
2 CONTAINER_REGISTRY_PASSWORD=fu/qlKjIMMzXWVqfGcng4PGD
3

```

Chapter 8 - Editing your container solution

You will now complete the container solution, by updating the following files:

- *main.py* python script.
- *requirements.txt*.
- *deployment.template.json*.
- *Dockerfile.amd64*

You will then create the *images* folder, used by the python script to check for images to match against your *Custom Vision model*. Lastly, you will add the *labels.txt* file, to help read your model, and the *model.pb* file, which is your model.

1. With VS Code open, navigate to your module folder, and look for the script called **main.py**. Double-click to open it.
2. Delete the content of the file and insert the following code:

```

# Copyright (c) Microsoft. All rights reserved.
# Licensed under the MIT license. See LICENSE file in the project root for
# full license information.

import random

```

```

import sched, time
import sys
import iothub_client
from iothub_client import IoTHubModuleClient, IoTHubClientError, IoTHubTransportProvider
from iothub_client import IoTHubMessage, IoTHubMessageDispositionResult, IoTHubError
import json
import os
import tensorflow as tf
import os
from PIL import Image
import numpy as np
import cv2

# messageTimeout - the maximum time in milliseconds until a message times out.
# The timeout period starts at IoTHubModuleClient.send_event_async.
# By default, messages do not expire.
MESSAGE_TIMEOUT = 10000

# global counters
RECEIVE_CALLBACKS = 0
SEND_CALLBACKS = 0

TEMPERATURE_THRESHOLD = 25
TWIN_CALLBACKS = 0

# Choose HTTP, AMQP or MQTT as transport protocol. Currently only MQTT is supported.
PROTOCOL = IoTHubTransportProvider.MQTT

# Callback received when the message that we're forwarding is processed.
def send_confirmation_callback(message, result, user_context):
    global SEND_CALLBACKS
    print ( "Confirmation[%d] received for message with result = %s" % (user_context, result) )
    map_properties = message.properties()
    key_value_pair = map_properties.get_internals()
    print ( "    Properties: %s" % key_value_pair )
    SEND_CALLBACKS += 1
    print ( "    Total calls confirmed: %d" % SEND_CALLBACKS )

def convert_to_opencv(image):
    # RGB -> BGR conversion is performed as well.
    r,g,b = np.array(image).T
    opencv_image = np.array([b,g,r]).transpose()
    return opencv_image

def crop_center(img,cropx,cropy):
    h, w = img.shape[:2]
    startx = w//2-(cropx//2)
    starty = h//2-(cropy//2)
    return img[starty:starty+cropy, startx:startx+cropx]

def resize_down_to_1600_max_dim(image):
    h, w = image.shape[:2]
    if (h < 1600 and w < 1600):
        return image

    new_size = (1600 * w // h, 1600) if (h > w) else (1600, 1600 * h // w)
    return cv2.resize(image, new_size, interpolation = cv2.INTER_LINEAR)

def resize_to_256_square(image):
    h, w = image.shape[:2]
    return cv2.resize(image, (256, 256), interpolation = cv2.INTER_LINEAR)

def update_orientation(image):
    exif_orientation_tag = 0x0112
    if hasattr(image, '_getexif'):
        exif = image._getexif()
        if (exif != None and exif[exif_orientation_tag] in exif):

```

```

    if exif := None and exif_orientation_tag in exif:
        orientation = exif.get(exif_orientation_tag, 1)
        # orientation is 1 based, shift to zero based and flip/transpose based on 0-based values
        orientation -= 1
        if orientation >= 4:
            image = image.transpose(Image.TRANSPOSE)
        if orientation == 2 or orientation == 3 or orientation == 6 or orientation == 7:
            image = image.transpose(Image.FLIP_TOP_BOTTOM)
        if orientation == 1 or orientation == 2 or orientation == 5 or orientation == 6:
            image = image.transpose(Image.FLIP_LEFT_RIGHT)
    return image

def analyse(hubManager):

    messages_sent = 0;

    while True:
        #def send_message():
        print ("Load the model into the project")
        # These names are part of the model and cannot be changed.
        output_layer = 'loss:0'
        input_node = 'Placeholder:0'

        graph_def = tf.GraphDef()
        labels = []

        labels_filename = "labels.txt"
        filename = "model.pb"

        # Import the TF graph
        with tf.gfile.FastGFile(filename, 'rb') as f:
            graph_def.ParseFromString(f.read())
            tf.import_graph_def(graph_def, name='')

        # Create a list of labels
        with open(labels_filename, 'rt') as lf:
            for l in lf:
                labels.append(l.strip())
        print ("Model loaded into the project")

        results_dic = dict()

        # create the JSON to be sent as a message
        json_message = ''

        # Iterate through images
        print ("List of images to analyse:")
        for file in os.listdir('images'):
            print(file)

            image = Image.open("images/" + file)

            # Update orientation based on EXIF tags, if the file has orientation info.
            image = update_orientation(image)

            # Convert to OpenCV format
            image = convert_to_opencv(image)

            # If the image has either w or h greater than 1600 we resize it down respecting
            # aspect ratio such that the largest dimension is 1600
            image = resize_down_to_1600_max_dim(image)

            # We next get the largest center square
            h, w = image.shape[:2]
            min_dim = min(w,h)
            max_square_image = crop_center(image, min_dim, min_dim)

            # Resize that square down to 256x256

```

```

augmented_image = resize_to_256_square(max_square_image)

# The compact models have a network size of 227x227, the model requires this size.
network_input_size = 227

# Crop the center for the specified network_input_Size
augmented_image = crop_center(augmented_image, network_input_size, network_input_size)

try:
    with tf.Session() as sess:
        prob_tensor = sess.graph.get_tensor_by_name(output_layer)
        predictions, = sess.run(prob_tensor, {input_node: [augmented_image] })
except Exception as identifier:
    print ("Identifier error: ", identifier)

print ("Print the highest probability label")
highest_probability_index = np.argmax(predictions)
print('FINAL RESULT! Classified as: ' + labels[highest_probability_index])

l = labels[highest_probability_index]

results_dic[file] = l

# Or you can print out all of the results mapping labels to probabilities.
label_index = 0
for p in predictions:
    truncated_probability = np.float64(round(p,8))
    print (labels[label_index], truncated_probability)
    label_index += 1

print("Results dictionary")
print(results_dic)

json_message = json.dumps(results_dic)
print("Json result")
print(json_message)

# Initialize a new message
message = IoTHubMessage(bytearray(json_message, 'utf8'))

hubManager.send_event_to_output("output1", message, 0)

messages_sent += 1
print("Message sent! - Total: " + str(messages_sent))
print('-----')

# This is the wait time before repeating the analysis
# Currently set to 10 seconds
time.sleep(10)

class HubManager(object):

    def __init__(self,
                 protocol=IoTHubTransportProvider.MQTT):
        self.client_protocol = protocol
        self.client = IoTHubModuleClient()
        self.client.create_from_environment(protocol)

        # set the time until a message times out
        self.client.set_option("messageTimeout", MESSAGE_TIMEOUT)

    # Forwards the message received onto the next stage in the process.
    def forward_event_to_output(self, outputQueueName, event, send_context):
        self.client.send_event_async(
            outputQueueName, event, send_confirmation_callback, send_context)

    def send_event_to_output(self, outputQueueName, event, send_context):

```

```

        self.client.send_event_async(outputQueueName, event, send_confirmation_callback, send_context)

def main(protocol):
    try:
        hub_manager = HubManager(protocol)
        analyse(hub_manager)
        while True:
            time.sleep(1)

    except IoTHubError as iothub_error:
        print ( "Unexpected error %s from IoTHub" % iothub_error )
        return
    except KeyboardInterrupt:
        print ( "IoTHubModuleClient sample stopped" )

if __name__ == '__main__':
    main(PROTOCOL)

```

3. Open the file called **requirements.txt**, and substitute its content with the following:

```

azure-iothub-device-client==1.4.0.0b3
opencv-python==3.3.1.11
tensorflow==1.8.0
pillow==5.1.0

```

4. Open the file called **deployment.template.json**, and substitute its content following the below guideline:
 - Because you will have your own, unique, JSON structure, you will need to edit it by hand (rather than copying an example). To make this easy, use the below image as a guide.
 - Areas which will look different to yours, but which you **should NOT change are highlighted yellow**.
 - Sections which you need to delete, are a highlighted red.**
 - Be careful to delete the correct brackets, and also remove the commas.

```

1  {
2     "moduleContent": {
3         "$edgeAgent": {
4             "properties.desired": {
5                 "schemaVersion": "1.0",
6                 "runtime": {
7                     "type": "docker",
8                     "settings": {
9                         "minDockerVersion": "v1.25",
10                        "loggingOptions": "",
11                        "registryCredentials": {
12                            "conta...": {
13                                "username": "${CONTAINER_REGISTRY_USERNAME}_iotedgeagent",
14                                "password": "${CONTAINER_REGISTRY_PASSWORD}_iotedgeagent",
15                                "address": "https://iotedgeagent.azurecr.io"
16                            }
17                        }
18                    }
19                },
20            "systemModules": {
21                "edgeAgent": {
22                    "type": "docker",
23                    "settings": {
24                        "image": "mcr.microsoft.com/azureiotedge-agent:1.0",
25                        "createOptions": ""
26                    }
27                },
28                "edgeHub": {
29                    "type": "docker",
30                    "status": "running",
31                    "restartPolicy": "always",
32                    "settings": {
33                        "image": "mcr.microsoft.com/azureiotedge-hub:1.0",
34                        "createOptions": "{\"HostConfig\":{\"PortBindings\":{\"8883/tcp\":[{\\\"HostPort\\\":\\\"8883\\\"}],\\\"443/tcp\":[{\\\"HostPort\\\":\\\"443\\\"}]}\\\"}"
35                    }
36                }
37            },
38            "modules": {
39                "tempSensor": {
40                    "version": "1.0",
41                    "type": "docker",
42                    "status": "running",
43                    "restartPolicy": "always",
44                    "settings": {
45                        "image": "mcr.microsoft.com/azureiotedge-simulated-temperature-sensor:1.0",
46                        "createOptions": ""
47                    }
48                },
49                "": {
50                    "version": "1.0",
51                    "type": "docker",
52                    "status": "running",
53                    "restartPolicy": "always",
54                    "settings": {
55                        "image": "${MODULES}/tempSensor:1.0",
56                        "createOptions": ""
57                    }
58                }
59            }
60        },
61        "$edgeHub": {
62            "properties.desired": {
63                "schemaVersion": "1.0",
64                "routes": {
65                    "sensorToEMPyModule": "FROM /messages/modules/tempSensor/outputs/temperatureOutput INTO BrokeredEndpoint(\"/modules/EMPyModule/inputs/input1\")",
66                    "": "ModuleToIoTHub": "FROM /messages/modules/:~/Module/outputs/output1 INTO $upstream"
67                },
68                "storeAndForwardConfiguration": {
69                    "timeToLiveSecs": 7200
70                }
71            }
72        }
73    }
74 }
75 }
```

- e. The completed JSON should look like the following image (though, with your unique differences:
username/password/module name/module references):

```

1  {
2     "moduleContent": {
3         "$edgeAgent": {
4             "properties.desired": {
5                 "schemaVersion": "1.0",
6                 "runtime": {
7                     "type": "docker",
8                     "settings": {
9                         "minDockerVersion": "v1.25",
10                        "loggingOptions": "",
11                        "registryCredentials": {
12                            "mcr.microsoft.com": {
13                                "username": "$CONTAINER_REGISTRY_USERNAME",
14                                "password": "$CONTAINER_REGISTRY_PASSWORD",
15                                "address": "mcr.microsoft.com.azurecr.io"
16                            }
17                        }
18                    },
19                },
20                "systemModules": {
21                    "edgeAgent": {
22                        "type": "docker",
23                        "settings": {
24                            "image": "mcr.microsoft.com/azureiotedge-agent:1.0",
25                            "createOptions": ""
26                        }
27                    },
28                    "edgeHub": {
29                        "type": "docker",
30                        "status": "running",
31                        "restartPolicy": "always",
32                        "settings": {
33                            "image": "mcr.microsoft.com/azureiotedge-hub:1.0",
34                            "createOptions": "{\"HostConfig\":{\"PortBindings\":{\"8883/tcp\":[{\"HostPort\":\"8883\"}],\"443/tcp\":[{\"HostPort\":\"443\"}]}}}"
35                        }
36                    }
37                },
38                "modules": {
39                    ".Module": {
40                        "version": "1.0",
41                        "type": "docker",
42                        "status": "running",
43                        "restartPolicy": "always",
44                        "settings": {
45                            "image": "${MODULES}.Module.amd64",
46                            "createOptions": ""
47                        }
48                    }
49                }
50            },
51        },
52        "edgeHub": {
53            "properties.desired": {
54                "schemaVersion": "1.0",
55                "routes": {
56                    "ModuleToIoTHub": "FROM /messages/modules/.Module/outputs/output1 INTO $upstream"
57                },
58                "storeAndForwardConfiguration": {
59                    "timeToLiveSecs": 7200
60                }
61            }
62        }
63    }
64 }

```

5. Open the file called **Dockerfile.amd64**, and substitute its content with the following:

```

FROM ubuntu:xenial

WORKDIR /app

RUN apt-get update && \
    apt-get install -y --no-install-recommends libcurl4-openssl-dev python-pip libboost-python-dev && \
    rm -rf /var/lib/apt/lists/*
RUN pip install --upgrade pip
RUN pip install setuptools

COPY requirements.txt ./
RUN pip install -r requirements.txt

RUN pip install pillow
RUN pip install numpy

RUN apt-get update && apt-get install -y \
    pkg-config \
    python-dev \
    python-opencv \
    libopencv-dev \
    libav-tools \
    libjpeg-dev \
    libpng-dev \
    libtiff-dev \
    libjasper-dev \
    python-numpy \
    python-pycurl \
    python-opencv

RUN pip install opencv-python
RUN pip install tensorflow
RUN pip install --upgrade tensorflow

COPY . .

RUN useradd -ms /bin/bash moduleuser
USER moduleuser

CMD [ "python", "-u", "./main.py" ]

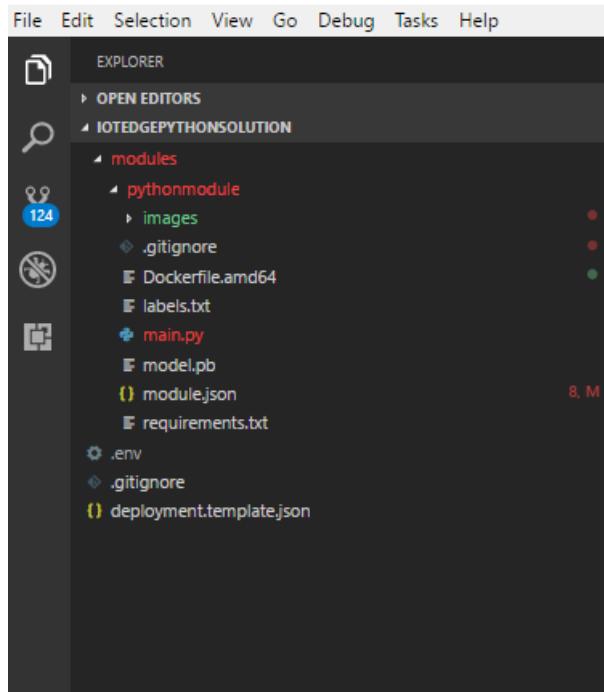
```

6. Right-click on the folder beneath **modules** (it will have the name you provided previously; in the example further down, it is called *pythonmodule*), and click on **New Folder**. Name the folder **images**.
7. Inside the folder, add some images containing mouse or keyboard. Those will be the images that will be analyzed by the Tensorflow model.

WARNING

If you are using your own model, you will need to change this to reflect your own models data.

8. You will now need to retrieve the **labels.txt** and **model.pb** files from the model folder, which you previous downloaded (or created from your own **Custom Vision Service**), in [Chapter 1](#). Once you have the files, place them within your solution, alongside the other files. The final result should look like the image below:



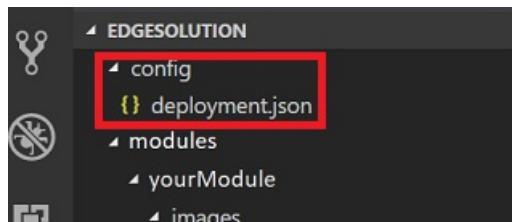
Chapter 9 - Package the solution as a container

1. You are now ready to "package" your files as a container and push it to your **Azure Container Registry**.

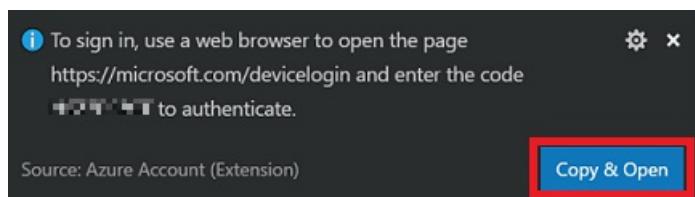
Within VS Code, open the *Integrated Terminal* (**View > Integrated Terminal / CTRL + `**), and use the following line to login to **Docker** (substitute the values of the command with the credentials of your **Azure Container Registry (ACR)**):

```
docker login -u <ACR username> -p <ACR password> <ACR login server>
```

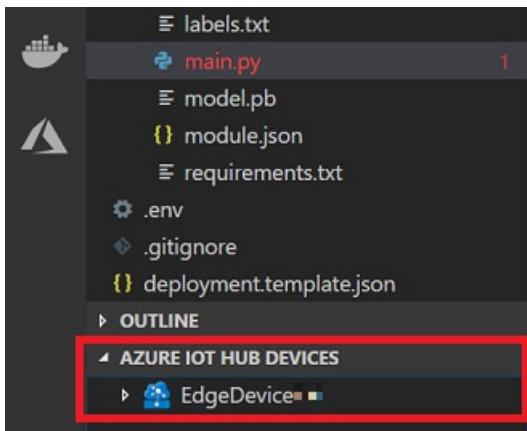
2. Right-click on the file **deployment.template.json**, and click **Build IoT Edge Solution**. This build process takes quite some time (depending on your device), so be prepared to wait. After the build process finishes, a **deployment.json** file will have been created inside a new folder called **config**.



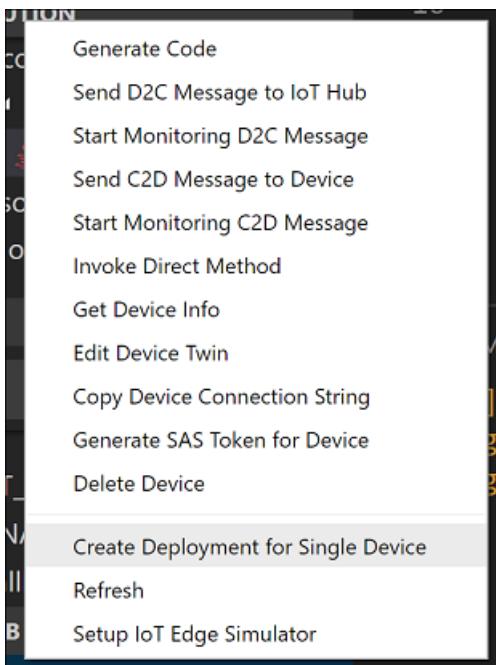
3. Open the **Command Palette** again, and search for **Azure: Sign In**. Follow the prompts using your Azure Account credentials; VS Code will provide you with an option to *Copy and Open*, which will copy the device code you will soon need, and open your default web browser. When asked, paste the device code, to authenticate your machine.



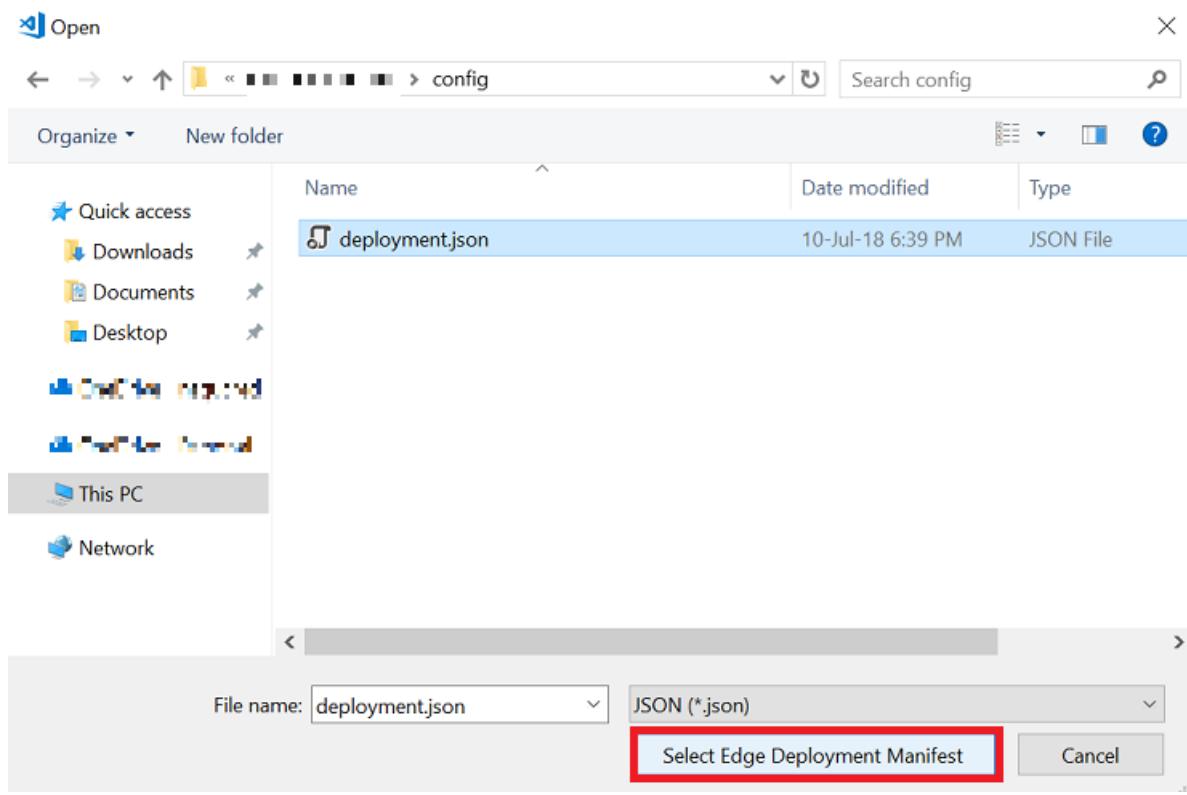
4. Once signed in you will notice, on the bottom side of the *Explore* panel, a new section called **Azure IoT Hub Devices**. Click this section to expand it.



5. If your device is not here, you will need to right-click *Azure IoT Hub Devices*, and then click **Set IoT Hub Connection String**. You will then see that the **Command Palette** (at the top of VS Code), will prompt you to input your *Connection String*. This is the *Connection String* you noted down at the end of [Chapter 3](#). Press the **Enter** key, once you have copied the string in.
6. Your device should load, and appear. Right-click on the device name, and then click, **Create Deployment for Single Device**.



7. You will get a *File Explorer* prompt, where you can navigate to the **config** folder, and then select the **deployment.json** file. With that file selected, click the **Select Edge Deployment Manifest** button.



8. At this point you have provided your **IoT Hub Service** with the manifest for it to deploy your container, as a module, from your **Azure Container Registry**, effectively deploying it to your device.
9. To view the messages sent from your device to the IoT Hub, right-click again on your device name in the **Azure IoT Hub Devices** section, in the **Explorer** panel, and click on **Start Monitoring D2C Message**. The messages sent from your device should appear in the VS Terminal. Be patient, as this may take some time. See the next Chapter for debugging, and checking if deployment was successful.

This module will now iterate between the images in the **images** folder and analyze them, with each iteration. This is obviously just a demonstration of how to get the basic machine learning model to work in an IoT Edge device environment.

To expand the functionality of this example, you could proceed in several ways. One way could be including some code in the container, that captures photos from a webcam that is connected to the device, and saves the images in the images folder.

Another way could be copying the images from the IoT device into the container. A practical way to do that is to run the following command in the IoT device Terminal (perhaps a small app could do the job, if you wished to automate the process). You can test this command by running it manually from the folder location where your files are stored:

```
sudo docker cp <filename> <modulename>:/app/images/<a name of your choice>
```

Chapter 10 - Debugging the IoT Edge Runtime

The following are a list of command lines, and tips, to help you monitor and debug the messaging activity of the *IoT Edge Runtime*, from your **Ubuntu device**.

- Check the *IoT Edge Runtime* status by running the following command line:

```
sudo systemctl status iotedge
```

NOTE

Remember to press **Ctrl + C**, to finish viewing the status.

- List the containers that are currently deployed. If the *IoT Hub Service* has deployed the containers successfully, they will be displayed by running the following command line:

```
sudo iotedge list
```

Or

```
sudo docker ps
```

NOTE

The above is a good way to check whether your module has been deployed successfully, as it will appear in the list; you will otherwise **only** see the *edgeHub* and *edgeAgent*.

- To display the code logs of a container, run the following command line:

```
journalctl -u iotedge
```

Useful commands to manage the IoT Edge Runtime:

- To delete all containers in the host:

```
sudo docker rm -f $(sudo docker ps -aq)
```

- To stop the *IoT Edge Runtime*:

```
sudo systemctl stop iotedge
```

Chapter 11 - Create Table Service

Navigate back to your Azure Portal, where you will create an Azure Tables Service, by creating a Storage resource.

1. If not already signed in, log into the [Azure Portal](#).
2. Once logged in, click on **Create a resource**, in the top left corner, and search for **Storage account**, and press the **Enter** key, to start the search.
3. Once it has appeared, click **Storage account - blob, file, table, queue** from the list.

The screenshot shows the Azure portal's 'New' blade. In the search bar at the top right, 'storage' is typed. Below the search bar, there are tabs for 'Azure Marketplace' (selected), 'See all', 'Featured', and 'See all'. The 'Storage' category in the sidebar is highlighted with a blue dashed box. The main content area shows several service cards: 'Storage account - blob, file, table, queue' (highlighted with a red box), 'Azure File Sync (preview) PREVIEW', 'Data Lake Store', and 'Storage Simple Physical Device Series'. Each card includes a 'Quickstart tutorial' link.

4. The new page will provide a description of the **Storage account** Service. At the bottom left of this prompt, click the **Create** button, to create an instance of this Service.

This screenshot shows the detailed description page for the Storage account service. It includes a summary of what Storage accounts are used for, social sharing links, publisher information, useful links, and a large 'Create' button at the bottom left which is highlighted with a red box.

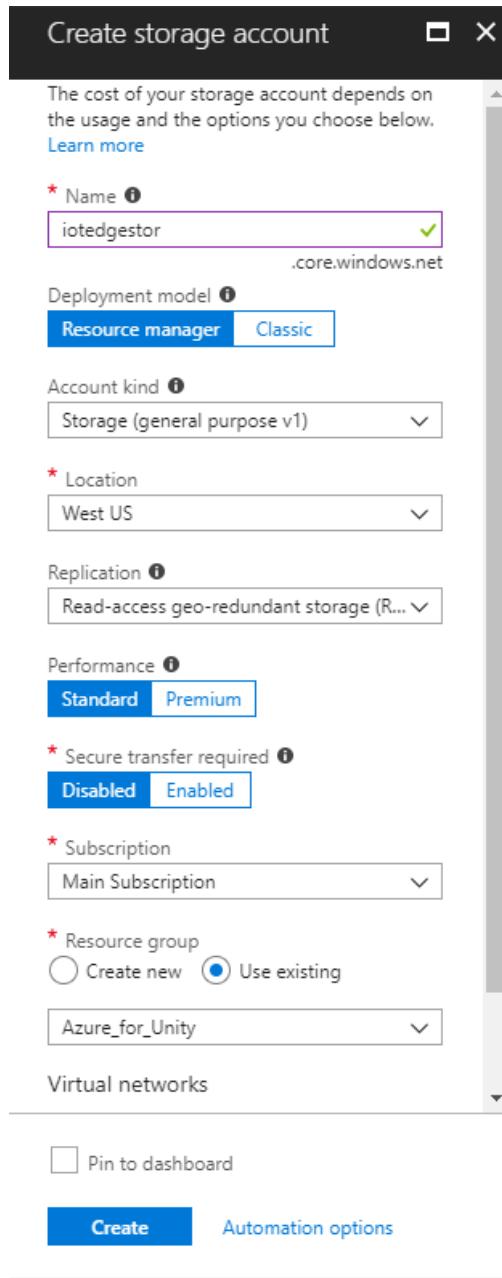
5. Once you have clicked on **Create**, a panel will appear:
 - Insert your desired **Name** for this Service instance (*must be all lowercase*).
 - For **Deployment model**, click **Resource manager**.
 - For **Account kind**, using the dropdown menu, click **Storage (general purpose v1)**.
 - Click an appropriate **Location**.
 - For the **Replication** dropdown menu, click **Read-access-geo-redundant storage (RA-GRS)**.
 - For **Performance**, click **Standard**.
 - Within the **Secure transfer required** section, click **Disabled**.
 - From the **Subscription** dropdown menu, click an appropriate subscription.
 - Choose a **Resource Group** or create a new one. A resource group provides a way to monitor, control access, provision, and manage, billing for a collection of Azure assets. It is recommended to keep all

the Azure Services associated with a single project (e.g. such as these courses) under a common resource group).

If you wish to read more about Azure Resource Groups, please follow this [link on how to manage a Resource Group](#).

j. Leave **Virtual networks** as **Disabled**, if this is an option for you.

k. Click **Create**.

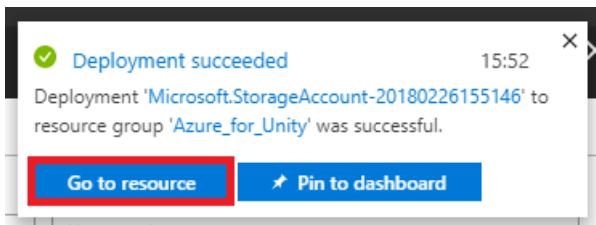


6. Once you have clicked on **Create**, you will have to wait for the Service to be created, this might take a minute.

7. A notification will appear in the Portal once the Service instance is created. Click on the notifications to explore your new Service instance.



8. Click the **Go to resource** button in the notification, and you will be taken to your new Storage Service instance overview page.



9. From the overview page, to the right-hand side, click **Tables**.

A screenshot of the Azure Storage Account overview page for the "iotedgestor" account. The left sidebar lists navigation options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Storage Explorer (preview), and various settings sections. The main content area shows general account details such as Resource group (Azure_for_Unity), Status (Primary: Available, Secondary: Available), Location (West US, East US), Subscription (Main Subscription), and Tags. On the right, there are sections for Services: Blobs, Files, and Tables. The "Tables" section is highlighted with a red box. It contains a table icon, the text "Tables Tabular data storage", and links to "Configure CORS rules" and "View metrics".

10. The panel on the right will change to show the **Table Service** information, wherein you need to add a new table. Do this by clicking the **+ Table** button to the top-left corner.

The screenshot shows the Azure Storage Tables service page. At the top, there are buttons for '+ Table', 'Refresh', and 'Delete tables'. A blue banner at the top says 'Check out premium Table experience with Azure Cosmos DB'. Below the banner, it says 'Storage account: iotedgestor'. There is a search bar labeled 'Search tables by prefix'. A table list shows one entry: 'TABLE' 'URL' 'IoTMessages' 'https://itedgestor.table.core.windows.net/... ...'. A message below the table says 'You don't have any tables yet.'

11. A new page will be shown, wherein you need to enter a **Table name**. This is the name you will use to refer to the data in your application in later Chapters (creating Function App, and Power BI). Insert **IoTMessages** as the name (you can choose your own, just remember it when used later in this document) and click **OK**.
12. Once the new table has been created, you will be able to see it within the **Table Service** page (at the bottom).

The screenshot shows the Azure Storage Tables service page after creating the 'IoTMessages' table. The interface is identical to the previous screenshot, but now the table list includes 'IoTMessages'.

13. Now click on **Access keys** and take a copy of the **Storage account name** and **Key** (using your Notepad), you will use these values later in this course, when creating the **Azure Function App**.

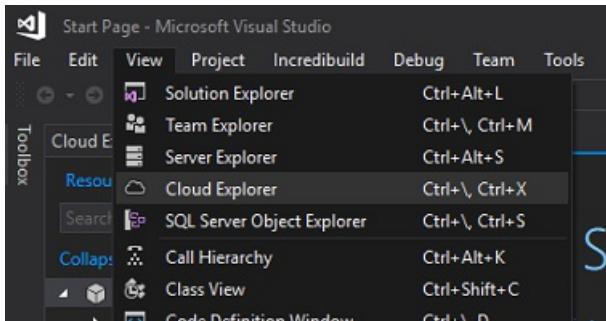
The screenshot shows the 'Access keys' page for the 'itedgestor' storage account. On the left, there is a sidebar with 'Overview', 'Activity log', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', and 'Storage Explorer (preview)'. Under 'SETTINGS', the 'Access keys' button is highlighted with a red box. The main content area shows the 'Storage account name' as 'itedgestor' and the 'key1' key as '0l2r7FgHdsIAvFnM55q8siJozB14igBORtAoND86wUcXMnbqh[REDACTED]'. Below the key, there is a 'Connection string' field with the value 'DefaultEndpointsProtocol=https;AccountName=itedgestor;AccountKey=0l2r7FgHdsIAvFnM55q8siJozB14igBORtAoND86wUcXMnbqhS31'.

14. Using the panel on the left again, scroll to the **Table Service** section, and click **Tables** (or **Browse Tables**, in newer Portals) and take a copy of the **Table URL** (using your Notepad). You will use this value later in this course, when linking your table to your **Power BI** application.

Chapter 12 - Completing the Azure Table

Now that your **Table Service** storage account has been setup, it is time to add data to it, which will be used to store and retrieve information. The editing of your Tables can be done through **Visual Studio**.

1. Open **Visual Studio** (**not** Visual Studio Code).
2. From the menu, click **View > Cloud Explorer**.

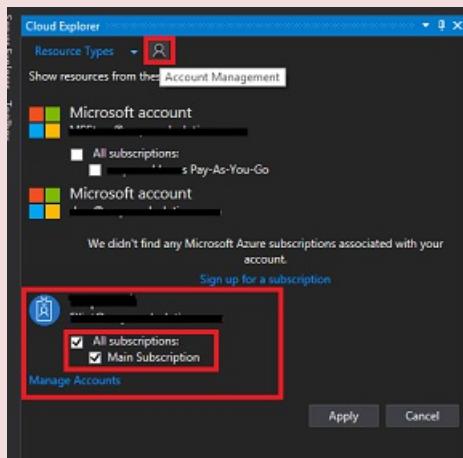


3. The **Cloud Explorer** will open as a docked item (be patient, as loading may take time).

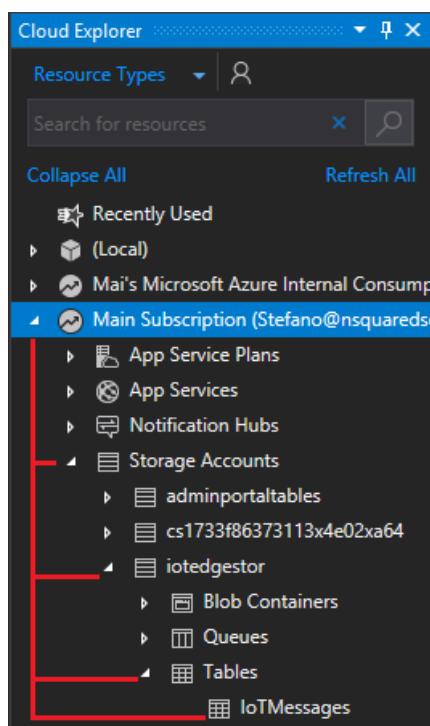
WARNING

If the subscription you used to create your **Storage Accounts** is not visible, ensure that you have:

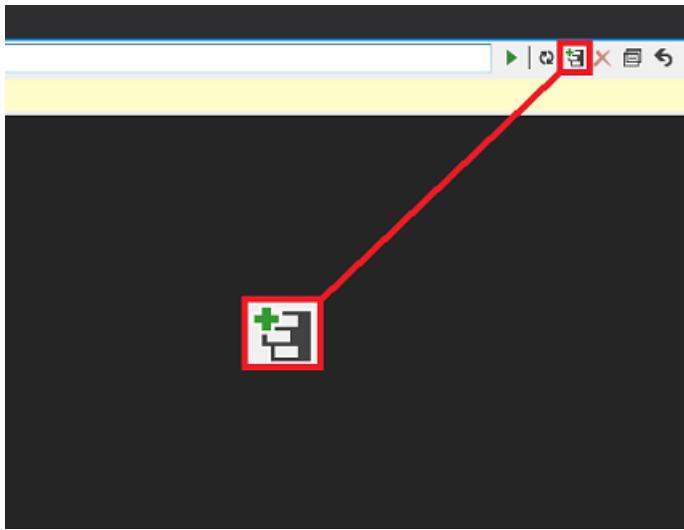
- Logged in to the same account as the one you used for the Azure Portal.
- Selected your subscription from the Account Management page (you may need to apply a filter from your account settings):



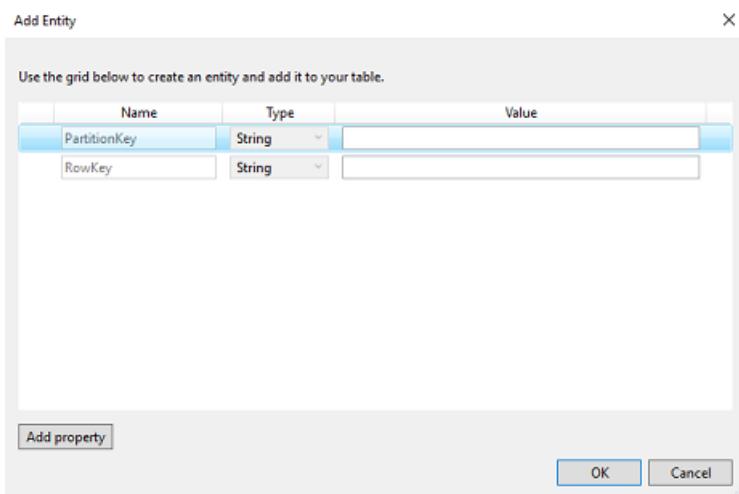
4. Your Azure cloud Services will be shown. Find **Storage Accounts** and click the arrow to the left of that to expand your accounts.



5. Once expanded, your newly created **Storage account** should be available. Click the arrow to the left of your storage, and then once that is expanded, find **Tables** and click the arrow next to that, to reveal the **Table** you created in the last Chapter. Double-click your **Table**.
6. Your table will be opened in the center of your Visual Studio window. Click the table icon with the + (plus) on it.



7. A window will appear prompting for you to *Add Entity*. You will create only one entity, though it will have three properties. You will notice that *PartitionKey* and *RowKey* are already provided, as these are used by the table to find your data.



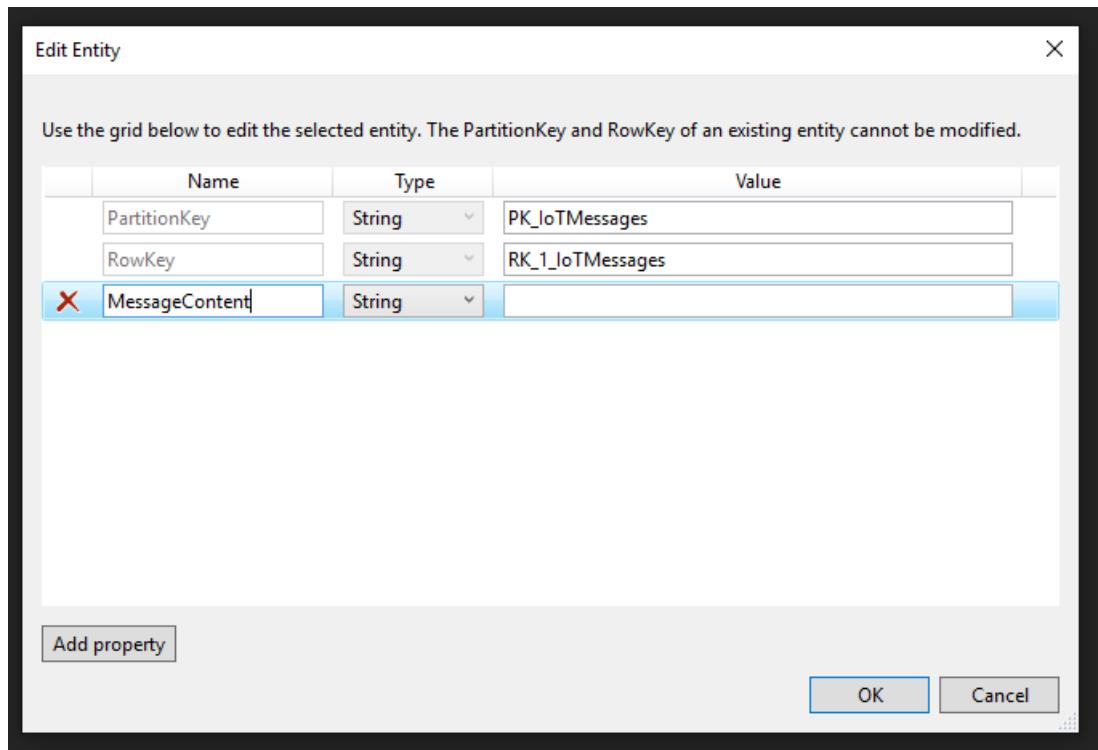
8. Update the following values:

- Name: **PartitionKey**, Value: **PK_IoTMessages**
- Name: **RowKey**, Value: **RK_1_IoTMessages**

9. Then, click **Add property** (to the lower left of the *Add Entity* window) and add the following property:

- **MessageContent**, as a *string*, leave the Value empty.

10. Your table should match the one in the image below:



NOTE

The reason why the entity has the number 1 in the row key, is because you might want to add more messages, should you desire to experiment further with this course.

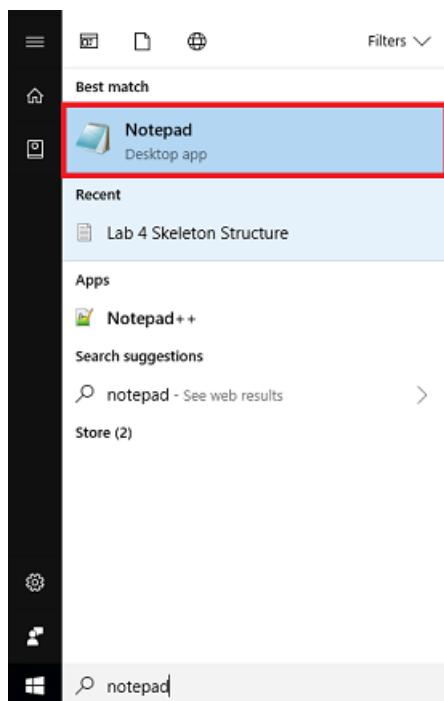
11. Click **OK** when you are finished. Your table is now ready to be used.

Chapter 13 - Create an Azure Function App

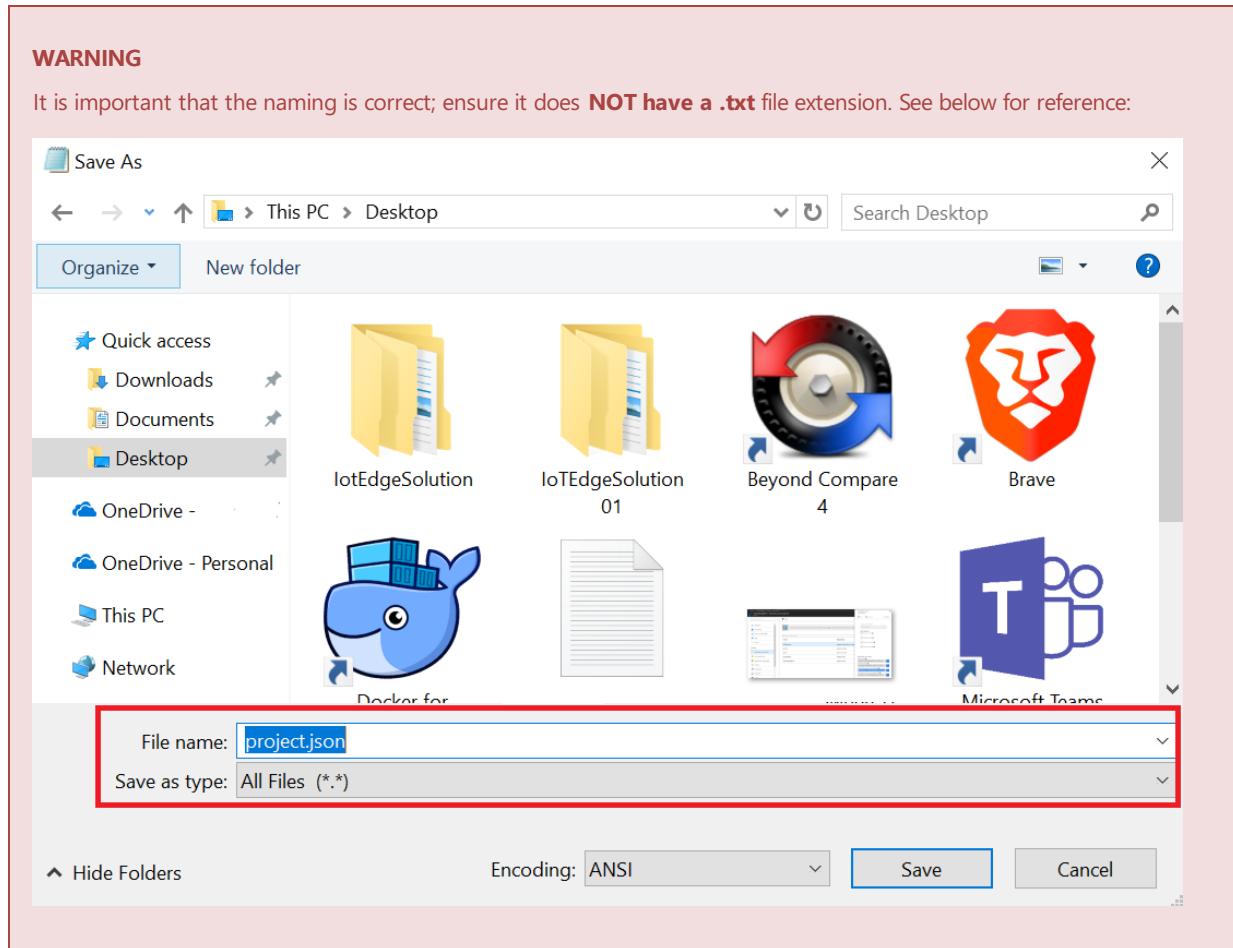
It is now time to create an *Azure Function App*, which will be called by the *IoT Hub Service* to store the *IoT Edge* device messages in the **Table** Service, which you created in the previous Chapter.

First, you need to create a file that will allow your Azure Function to load the libraries you need.

1. Open **Notepad** (press the *Windows Key*, and type *notepad*).



2. With Notepad open, insert the JSON structure below into it. Once you have done that, save it on your desktop as **project.json**. This file defines the libraries your function will use. If you have used NuGet, it will look familiar.



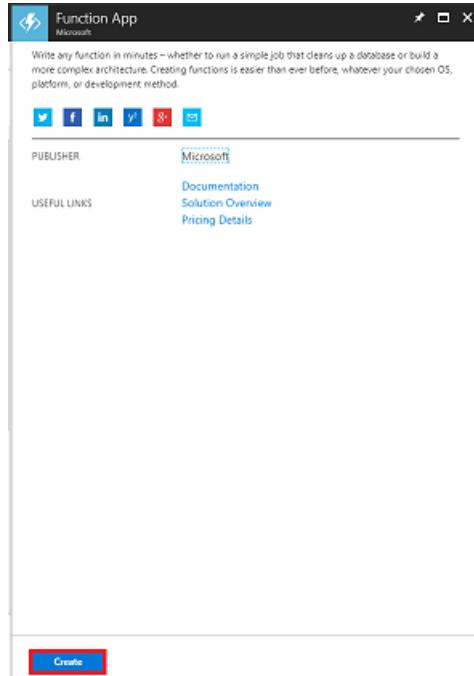
```
{
  "frameworks": {
    "net46": {
      "dependencies": {
        "WindowsAzure.Storage": "9.2.0"
      }
    }
  }
}
```

3. Log in to the [Azure Portal](#).
4. Once you are logged in, click on **Create a resource** in the top left corner, and search for **Function App**, and press the **Enter** key, to search. Click *Function App* from the results, to open a new panel.

NAME	PUBLISHER	CATEGORY
Function App	Microsoft	Web + Mobile
Functions Bot	Microsoft	AI + Cognitive Services
Logic Apps Custom Connector	Microsoft	Web + Mobile

5. The new panel will provide a description of the **Function App** Service. At the bottom left of this panel, click

the **Create** button, to create an association with this Service.



6. Once you have clicked on **Create**, fill in the following:

- a. For **App name**, insert your desired name for this Service instance.
- b. Select a **Subscription**.
- c. Select the pricing tier appropriate for you, if this is the first time creating a **Function App Service**, a free tier should be available to you.
- d. Choose a **Resource Group** or create a new one. A resource group provides a way to monitor, control access, provision, and manage, billing for a collection of Azure assets. It is recommended to keep all the Azure Services associated with a single project (e.g. such as these courses) under a common resource group).

If you wish to read more about Azure Resource Groups, please follow this [link on how to manage a Resource Group](#).

- e. For **OS**, click Windows, as that is the intended platform.
- f. Select a **Hosting Plan** (this tutorial is using a **Consumption Plan**).
- g. Select a **Location** (choose the same location as the storage you have built in the previous step)
- h. For the **Storage** section, **you must select the Storage Service you created in the previous step**.
 - i. You will not need *Application Insights* in this app, so feel free to leave it **Off**.
 - j. Click **Create**.

* App name
IoTEdgeFunctApp .azurewebsites.net

* Subscription
Main Subscription

* Resource Group ⓘ
 Create new Use existing
Azure_for_Unity

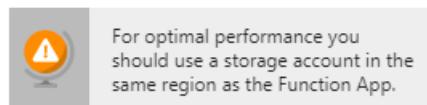
* OS **Windows** Linux (Preview) Docker

* Hosting Plan ⓘ Consumption Plan

* Location West US

* Storage ⓘ
 Create new Use existing
iotedgestor

Application Insights ⓘ **On** Off



Pin to dashboard

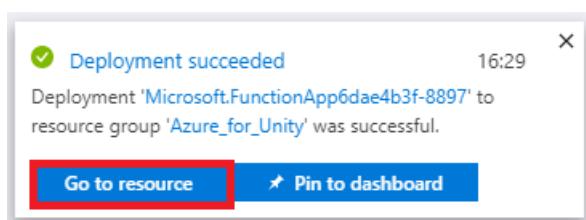
Create

Automation options

7. Once you have clicked on **Create**, you will have to wait for the Service to be created, this might take a minute.
8. A notification will appear in the Portal once the Service instance is created.



9. Click on the notification, once deployment is successful (has finished).
10. Click the **Go to resource** button in the notification to explore your new Service instance.



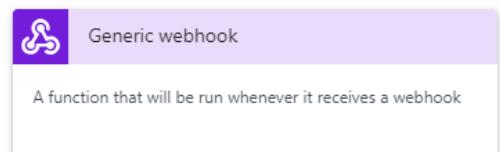
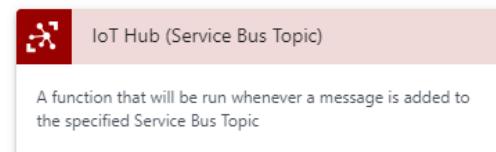
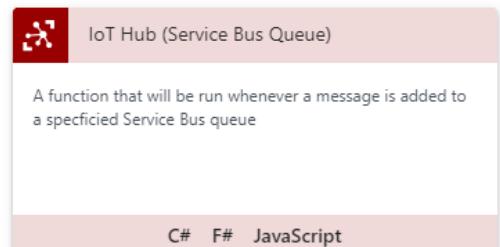
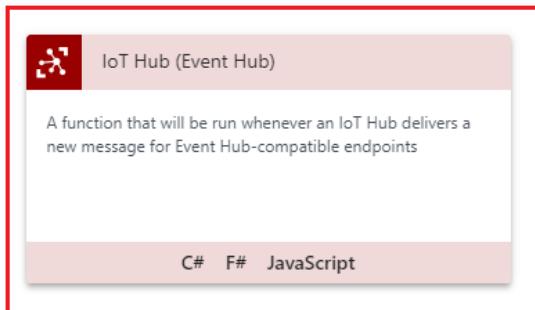
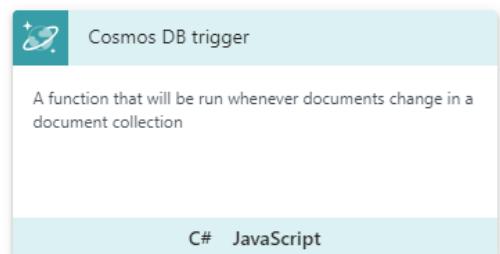
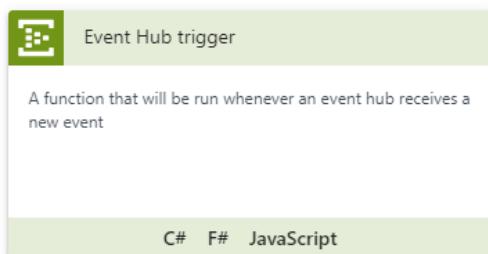
11. In the left side of the new panel, click the + (plus) icon next to *Functions*, to create a new function.

The screenshot shows the Azure Functions overview for the 'IoTEdgeFunctApp'. On the left, there's a sidebar with a search bar for 'IoTEdgeFunctApp', a dropdown for 'All subscriptions', and sections for 'Function Apps', 'Proxies', and 'Slots (preview)'. Under 'Functions', there's a 'Create new' button with a red box around it. The main right panel has a title 'Overview' with a 'Stop' button and a 'Swa' link. Below that is a 'Status' section showing 'Running' with a green checkmark. At the bottom, there's a 'Configure' button and a 'Function apps' link.

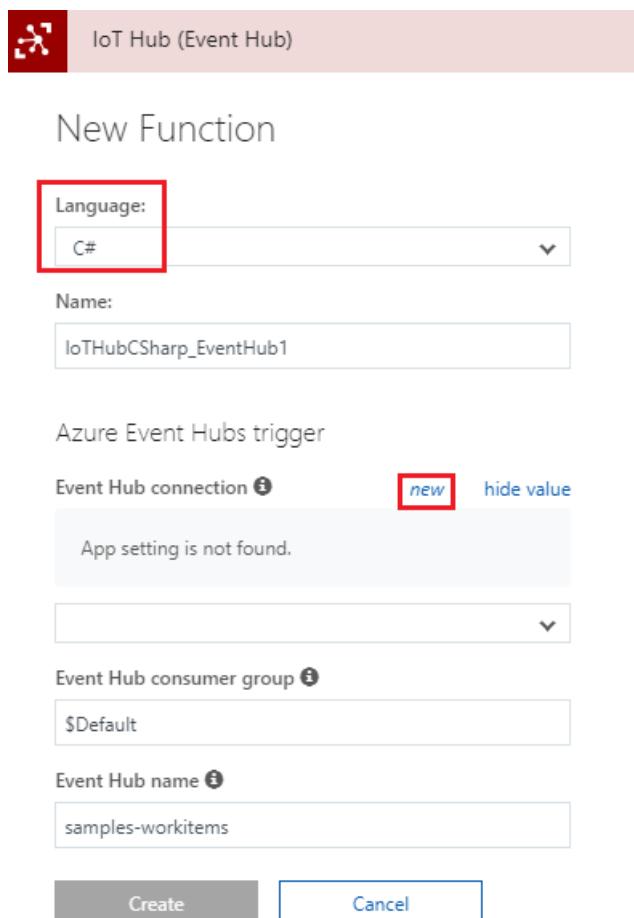
12. Within the central panel, the **Function** creation window will appear. Scroll down further, and click on **Custom function**.

The screenshot shows the 'Create this function' dialog. It starts with a heading 'Get started quickly with a premade function' and a lightning bolt icon. Below it, step 1 'Choose a scenario' shows three options: 'Webhook + API' (selected and highlighted with a blue border), 'Timer', and 'Data processing'. Step 2 'Choose a language' shows radio buttons for CSharp (selected), JavaScript, FSharp, and Java. A note below says 'For PowerShell, Python, and Batch, [create your own custom function](#)'. At the bottom is a large blue 'Create this function' button. Below the button, a blue banner says 'Get started on your own' with a 'Custom function' link (which has a red box around it) and a 'Start from source control' link.

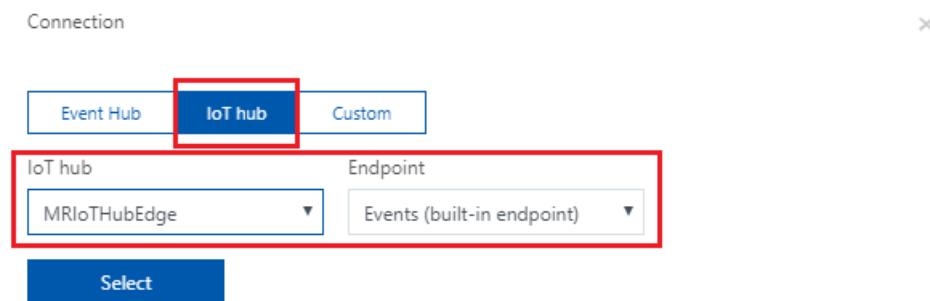
13. Scroll down the next page, until you find **IoT Hub (Event Hub)**, then click on it.



14. In the **IoT Hub (Event Hub)** blade, set the **Language** to **C#** and then click on **new**.



15. In the window that will appear, make sure that **IoT Hub** is selected and the name of the *IoT Hub* field corresponds with the name of your *IoT Hub Service* that you have created previously (in step 8, of Chapter 3). Then click the **Select** button.



16. Back on the **IoT Hub (Event Hub)** blade, click on **Create**.

Language: C#

Name: IoTHubCSharp_EventHub1

Azure Event Hubs trigger

Event Hub connection: MRIoTHubEdge_events_IOTHUB

Event Hub consumer group: \$Default

Event Hub name: samples-workitems

Create

17. You will be redirected to the function editor.

```

1 using System.Net;
2
3 public static async Task<HttpResponseMessage> Run(HttpRequestMessage req, TraceWriter log)
4 {
5     log.Info("C# HTTP trigger function processed a request.");
6
7     // parse query parameter
8     string name = req.GetQueryNameValuePairs()
9         .FirstOrDefault(q => string.Compare(q.Key, "name", true) == 0)
10        .Value;
11
12     if (name == null)
13     {
14         // Get request body
15         dynamic data = await req.Content.ReadAsAsync<object>();
16         name = data?.name;
17     }
18
19     return name == null
20         ? req.CreateResponse(HttpStatusCode.BadRequest, "Please pass a name on the query string or in the request body")
21         : req.CreateResponse(HttpStatusCode.OK, "Hello " + name);
22 }

```

18. Delete all the code in it and replace it with the following:

```
#r "Microsoft.WindowsAzure.Storage"
#r "NewtonSoft.Json"

using System;
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.Table;
using Newtonsoft.Json;
using System.Threading.Tasks;

public static async Task Run(string myIoTHubMessage, TraceWriter log)
{
    log.Info($"C# IoT Hub trigger function processed a message: {myIoTHubMessage}");

    //RowKey of the table object to be changed
    string tableName = "IoTMessages";
    string tableURL = "https://iothubmrstorage.table.core.windows.net/IoTMessages";

    // If you did not name your Storage Service as suggested in the course, change the name here with
    // the one you chose.
    string storageAccountName = "iotedgestor";

    string storageAccountKey = "<Insert your Storage Key here>";

    string partitionKey = "PK_IoTMessages";
    string rowKey = "RK_1_IoTMessages";

    Microsoft.WindowsAzure.Storage.Auth.StorageCredentials storageCredentials =
        new Microsoft.WindowsAzure.Storage.Auth.StorageCredentials(storageAccountName,
    storageAccountKey);

    CloudStorageAccount storageAccount = new CloudStorageAccount(storageCredentials, true);

    // Create the table client.
    CloudTableClient tableClient = storageAccount.CreateCloudTableClient();

    // Get a reference to a table named "IoTMessages"
    CloudTable messageTable = tableClient.GetTableReference(tableName);

    //Retrieve the table object by its RowKey
    TableOperation operation = TableOperation.Retrieve<MessageEntity>(partitionKey, rowKey);
    TableResult result = await messageTable.ExecuteAsync(operation);

    //Create a MessageEntity so to set its parameters
    MessageEntity messageEntity = (MessageEntity)result.Result;

    messageEntity.MessageContent = myIoTHubMessage;
    messageEntity.PartitionKey = partitionKey;
    messageEntity.RowKey = rowKey;

    //Replace the table appropriate table Entity with the value of the MessageEntity Ccass structure.
    operation = TableOperation.Replace(messageEntity);

    // Execute the insert operation.
    await messageTable.ExecuteAsync(operation);
}

// This MessageEntity structure which will represent a Table Entity
public class MessageEntity : TableEntity
{
    public string Type { get; set; }
    public string MessageContent { get; set; }
}
```

19. Change the following variables, so that they correspond to the appropriate values (**Table** and **Storage**

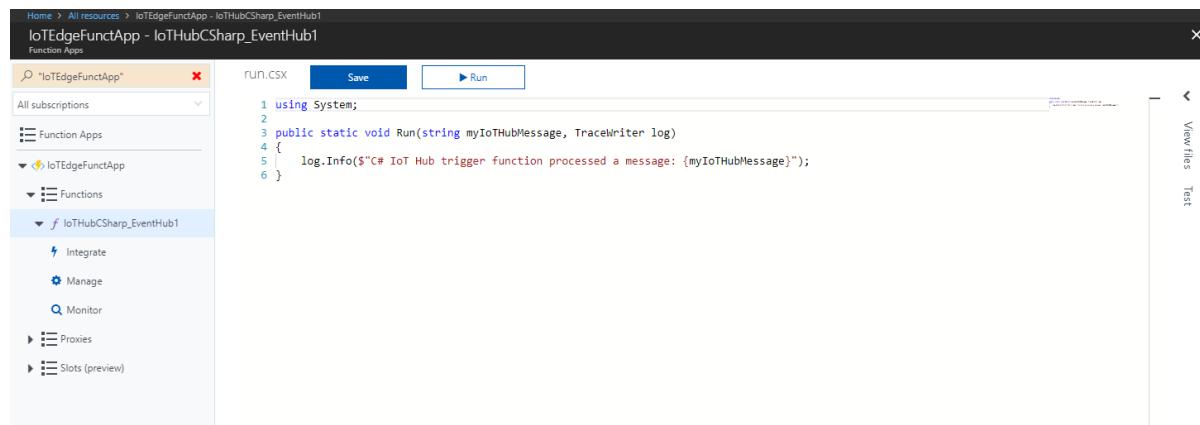
values, from step 11 and 13, respectively, of Chapter 11), that you will find in your **Storage Account**:

- **tableName**, with the name of your **Table** located in your **Storage Account**.
- **tableURL**, with the URL of your **Table** located in your **Storage Account**.
- **storageAccountName**, with the name of the value corresponding with the name of your **Storage Account** name.
- **storageAccountKey**, with the Key you have obtained in the Storage Service you have created previously.

```
14 //RowKey of the table object to be changed
15 string tableName = "IoTMessages";
16 string tableURL = "https://myStorageAccountName.table.core.windows.net/IoTMessages";
17
18 // If you did not name your Storage Service as suggested in the Lab, change the name here with
19 string storageAccountName = "myStorageAccountName";
20
21 string storageAccountKey = "myStorageAccountKey";  
22
23 string partitionKey = "PK_IoTMessages";
24 string rowKey = "RK_1_IoTMessages";
25
```

20. With the code in place, click **Save**.

21. Next, click the < (arrow) icon, on the right-hand side of the page.



22. A panel will slide in from the right. In that panel, click **Upload**, and a *File Browser* will appear.

23. Navigate to, and click, the **project.json** file, which you created in **Notepad** previously, and then click the **Open** button. This file defines the libraries that your function will use.

ction URL

The screenshot shows the Azure Storage Explorer interface. On the left, there's a tree view of storage accounts and containers. On the right, a detailed view of a folder named 'Json projects files'. Inside this folder, there are several files: 'function.json', 'readme.md', and 'run.csx'. A file named 'project.json' is highlighted with a red box. At the top of the right panel, there are buttons for '+ Add', 'Upload' (which is also highlighted with a red box), and 'Delete'.

24. When the file has uploaded, it will appear in the panel on the right. Clicking it will open it within the **Function** editor. It must look **exactly** the same as the next image.

The screenshot shows the Azure Function App blade for 'IoTEdgeFuncApp - HttpTriggerCSharp1'. On the left, there's a navigation menu with 'Integrate' highlighted with a red box. The main area displays the 'project.json' file content:

```
1 {  
2   "frameworks": {  
3     "net46": {  
4       "dependencies": {  
5         "WindowsAzure.Storage": "9.2.0"  
6       }  
7     }  
8   }  
9 }
```

Below the code, there are 'Save' and 'Run' buttons. To the right, there's a 'View files' tab and a file list: 'HttpTriggerCSharp1', 'function.json', 'project.json' (which is highlighted with a red box), 'readme.md', and 'run.csx'.

25. At this point it would be good to test the capability of your Function to store the message on your *Table*. On the top right side of the window, click on **Test**.

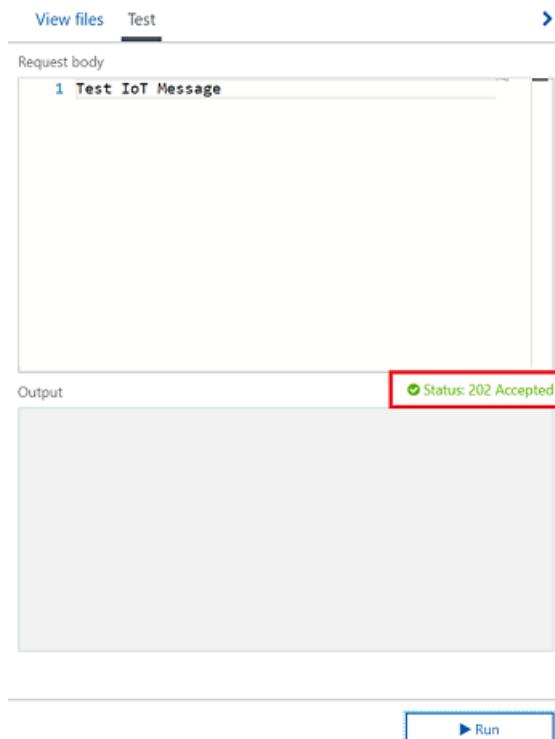
The screenshot shows the Azure Functions Test tab. On the left, the `Run.csx` file contains C# code for an IoT Hub trigger function. In the center, the `Request body` field has the value `1 Message from IoT Edge Device!`. On the right, there is an `Output` window and a `Run` button at the bottom right.

```

1 #r "Microsoft.WindowsAzure.Storage"
2 #r "Newtonsoft.Json"
3
4 using System;
5 using Microsoft.WindowsAzure.Storage;
6 using Microsoft.WindowsAzure.Storage.Table;
7 using Newtonsoft.Json;
8 using System.Threading.Tasks;
9
10 public static async Task Run(string myIoTHubMessage, TraceWriter log)
11 {
12     log.Info($"C# IoT Hub trigger function processed a message: {myIc
13
14     //RowKey of the table object to be changed
15     string tableName = "IoTMessages";
16     string tableURL = "https://iothubmrstorage.table.core.windows.net
17
18     string storageAccountName = "iotedgestor";
19     string storageAccountKey = "0I2r7FgHdsIAvFnM55q8siJozBl4igBORTAo
20
21     string partitionKey = "PK_IoTMessages";
22     string rowKey = "RK_1_IoTMessages";
23
24     Microsoft.WindowsAzure.Storage.Auth.StorageCredentials storageCre
25         new Microsoft.WindowsAzure.Storage.Auth.StorageCredentials(stor
26
27     CloudStorageAccount storageAccount = new CloudStorageAccount(stor
28
29     // Create the table client.
30     CloudTableClient tableClient = storageAccount.CreateCloudTableCli
31
32     // Get a reference to a table named "IoTMessages"
33     CloudTable messageTable = tableClient.GetTableReference("IoTMessag
34
35     //Retrieve the table object by its RowKey
36     TableOperation operation = TableOperation.Retrieve<MessageEntity>

```

26. Insert a message on the **Request body**, as shown in the image above, and click on **Run**.
27. The function will run, displaying the result status (you will notice the green **Status 202 Accepted**, above the *Output* window, which means it was a successful call):



Chapter 14 - View active messages

If you now open Visual Studio (**not** Visual Studio Code), you can visualize your test message result, as it will be stored in the *MessageContent* string area.

The screenshot shows the Microsoft Visual Studio interface with the 'Cloud Explorer' window open. The 'IoTMessages [Table]' tab is selected. A search bar at the top says 'Enter a WCF Data Services filter to limit the entities returned'. Below it is a table with four columns: PartitionKey, RowKey, Timestamp, and MessageContent. There is one visible row: PartitionKey 'PK_IoTMessages', RowKey 'RK_1_IoTMessa...', Timestamp '20/06/2018 1:51...', and MessageContent 'Message from IoT Edge Device!'. The rest of the interface shows other tabs like 'AzureWebJobsHostLogs201806 [Table]' and various toolbars.

With the Table Service and Function App in place, your Ubuntu device messages will appear in your *IoTMessages* Table. If not already running, start your device again, and you will be able to see the result messages from your device, and module, within your Table, through using Visual Studio *Cloud Explorer*.

This screenshot is similar to the one above, showing the 'Cloud Explorer' in Visual Studio. The 'IoTMessages [Table]' tab is selected. The table structure and data are identical: PartitionKey 'PK_IoTMessages', RowKey 'RK_1_IoTMessa...', Timestamp '11-Jul-18 1:57:...', and MessageContent '("mouse4.jif": "Mouse", "mouse.jpg": "Mouse", "mouse3.jif": "Mouse", "mouse2.jif": "...')'. The 'MessageContent' column is highlighted with a red box.

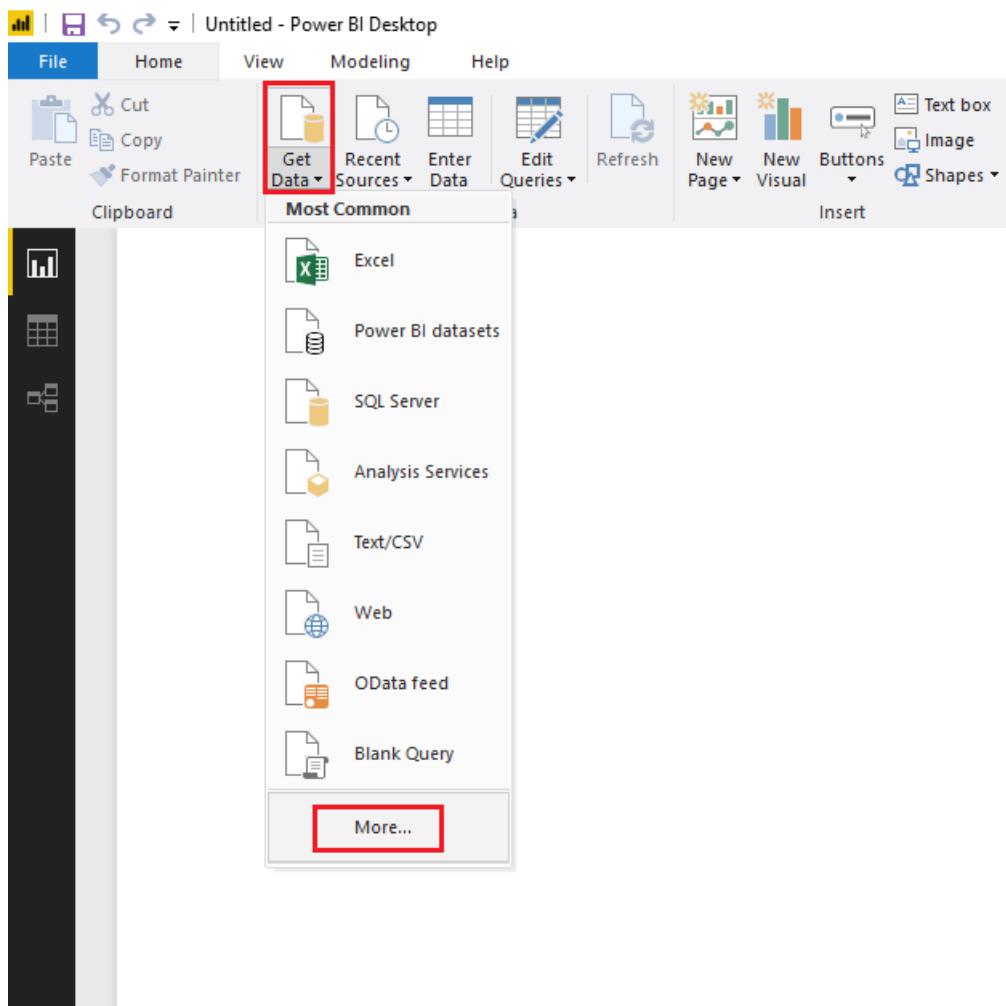
Chapter 15 - Power BI Setup

To visualize the data from your IOT device you will setup **Power BI** (desktop version), to collect the data from the *Table Service*, which you just created. The *HoloLens* version of Power BI will then use that data to visualize the result.

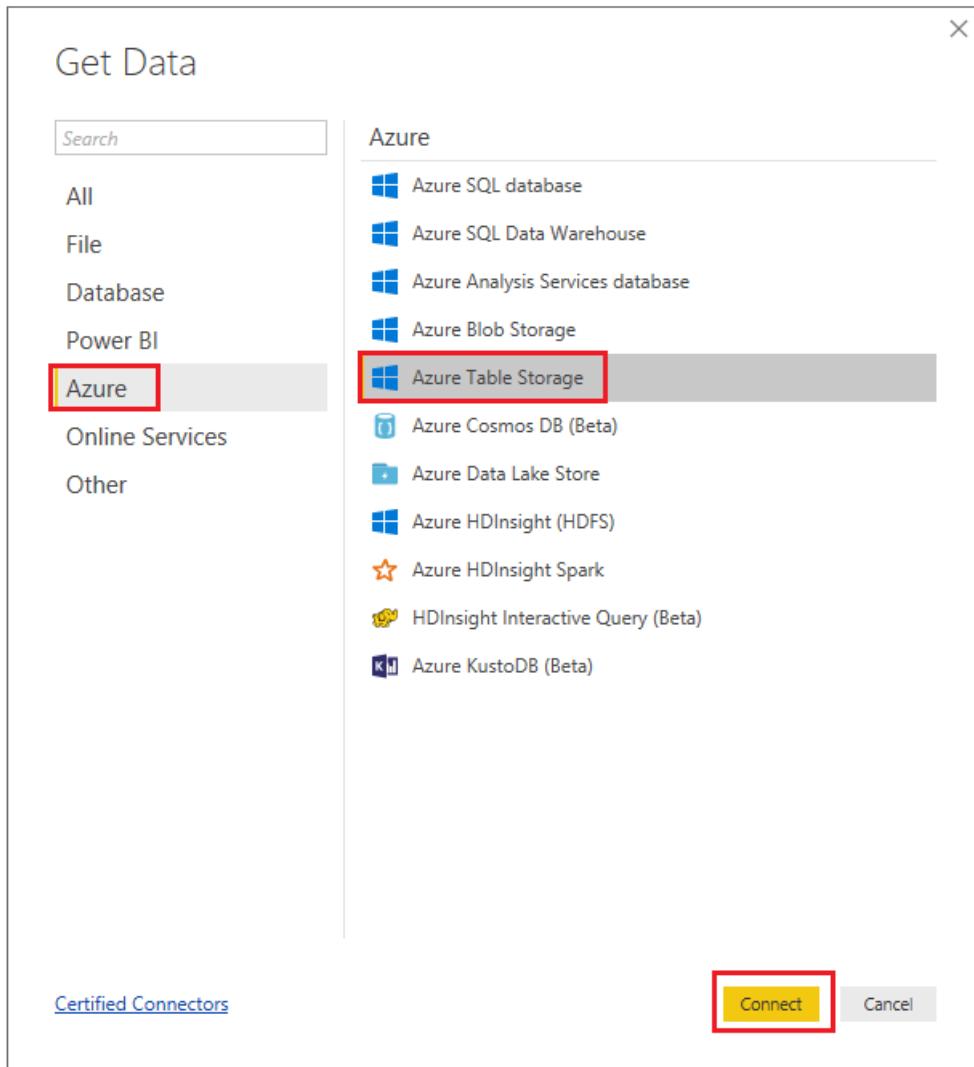
1. Open the Microsoft Store on Windows 10 and search for **Power BI Desktop**.

The screenshot shows the Microsoft Store application window. The search bar at the top contains 'Power BI'. Below the search bar, there are tabs for 'Home', 'Apps', 'Games', and 'Films & TV'. The 'Apps' tab is selected. On the right side, a search results panel titled 'Power BI' shows two items: 'Power BI Desktop App' and 'Power BI Mobile App'. The 'Power BI Desktop App' item is highlighted with a red box.

2. Download the application. Once it has finished downloading, open it.
3. Log into **Power BI** with your **Microsoft 365 account**. You may be redirected to a browser, to sign up. Once you are signed up, go back to the Power BI app, and sign in again.
4. Click on **Get Data** and then click on **More....**



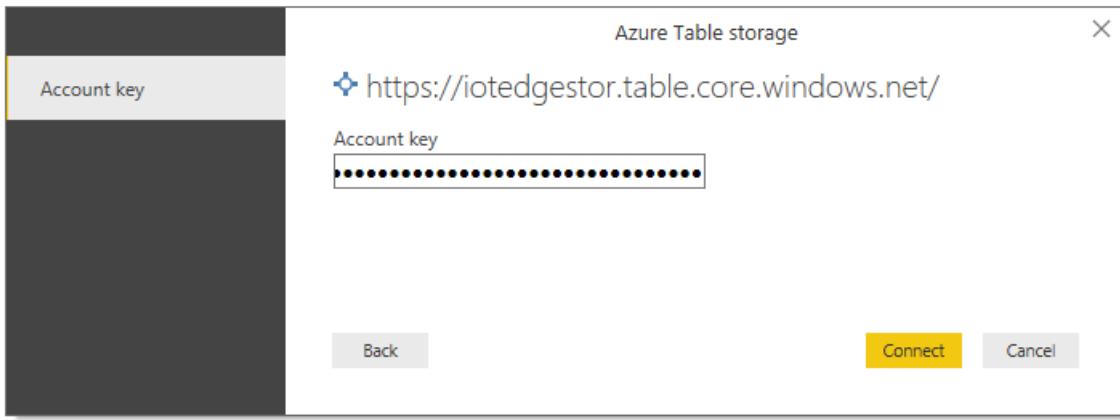
5. Click **Azure, Azure Table Storage**, then click on **Connect**.



6. You will be prompted to insert the **Table URL** that you collected earlier ([in step 13 of Chapter 11](#)), while creating your Table Service. After inserting the URL, delete the portion of the path referring to the Table "sub-folder" (which was IoTMessages, in this course). The final result should be as displayed in the image below. Then click on **OK**.



7. You will be prompted to insert the **Storage Key** that you noted ([in step 11 of Chapter 11](#)) earlier while creating your Table Storage. Then click on **Connect**.



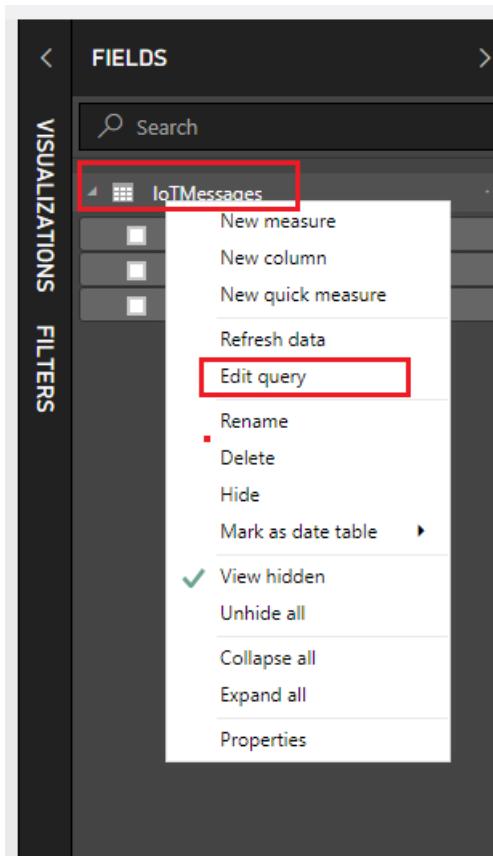
8. A **Navigator Panel** will be displayed, tick the box next to your Table and click on **Load**.

The screenshot shows the 'Navigator' panel in Power BI. On the left, under 'Display Options', there's a tree view with 'https://iotedgestor.table.core.windows.net [3]' expanded. Under it are three tables: 'AzureWebJobsHostLogs201806', 'AzureWebJobsHostLogscommon', and 'IoTMessages', with the last one checked. The main pane shows a table named 'IoTMessages' with the following data:

PartitionKey	RowKey	Timestamp	Content
PK_IoTMessages	RK_1_IoTMessages	19/06/2018 6:32:06 AM	Record

At the bottom are 'Load', 'Edit', and 'Cancel' buttons.

9. Your table has now been loaded on Power BI, but it requires a query to display the values in it. To do so, right-click on the table name located in the **FIELDS panel** at the right side of the screen. Then click on **Edit Query**.



10. A **Power Query Editor** will open up as a new window, displaying your table. Click on the word **Record** within the *Content* column of the table, to visualize your stored content.

The screenshot shows the Power Query Editor window. The ribbon at the top includes 'File', 'Home', 'Transform', 'Add Column', 'View', and 'Help'. The 'Home' tab is selected. The main area displays a table with one row. The columns are labeled 'PartitionKey', 'RowKey', 'Timestamp', and 'Content'. The 'Content' cell contains the word 'Record', which is highlighted by a red box. The 'Queries [1]' pane on the left shows a single item named 'IoTMessages'.

PartitionKey	RowKey	Timestamp	Content
PK_IoTMessages	RK_1_IoTMessages	19/06/2018 6:32:06 AM	Record

11. Click on **Into Table**, at the top-left of the window.

The screenshot shows the Microsoft Power Query Editor interface. The ribbon at the top has tabs: File, Home, Transform, Add Column, View, Help, Record Tools, and Convert. The 'Convert' tab is selected. On the left, there's a 'Convert' section with a 'Table' icon and a 'Convert' button. The main area shows 'Queries [1]' with 'IoTMessages' selected. The preview pane displays the data: 'MessageContent' - 'Hello from IoT Edge Device!'. The status bar at the bottom says 'Untitled - Power Query Editor'.

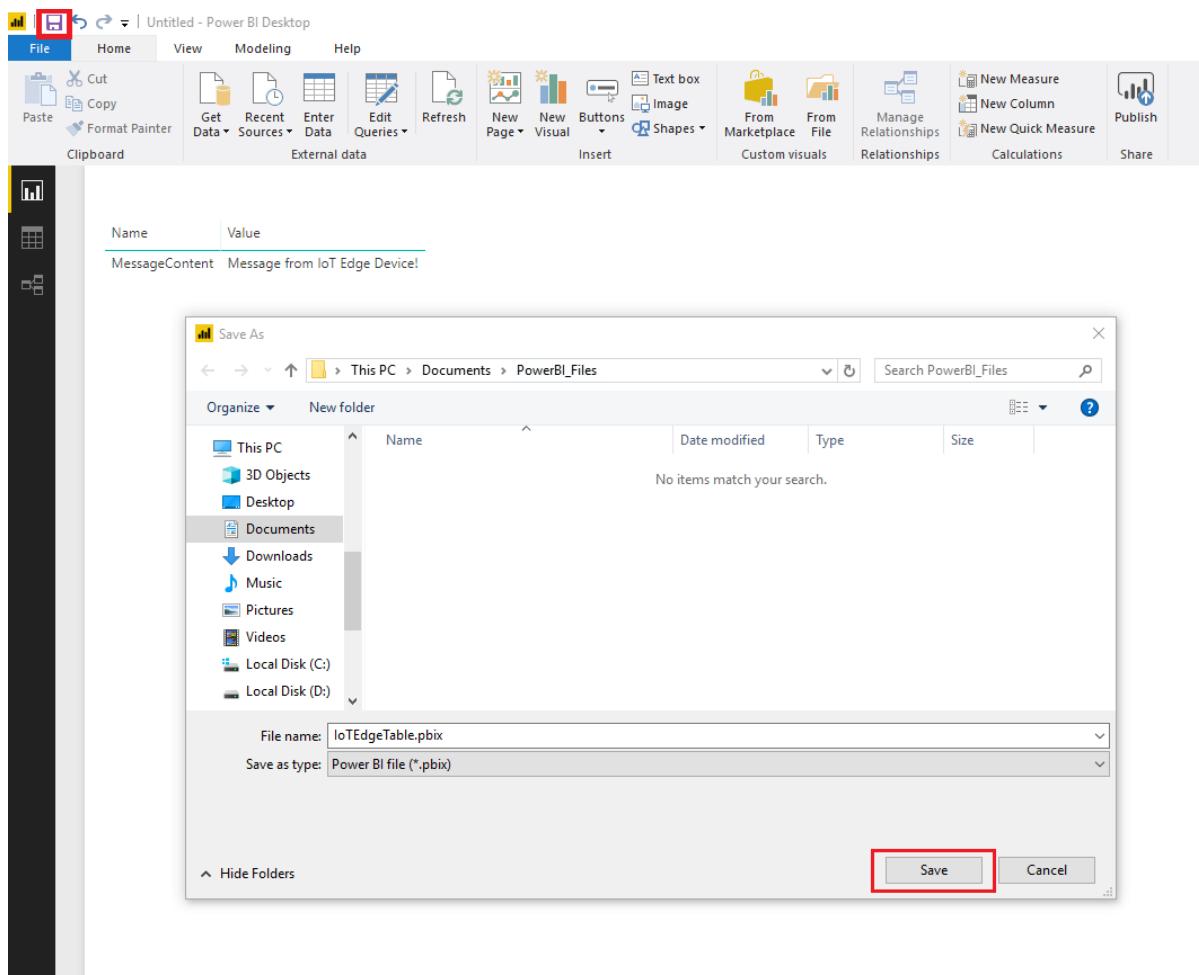
12. Click on **Close & Apply**.

This screenshot is similar to the previous one, but the 'Close & Apply' button in the ribbon is highlighted with a red box. The preview pane now shows the loaded data: 'MessageContent' - 'Hello from IoT Edge Device!'. The rest of the interface is identical to the first screenshot.

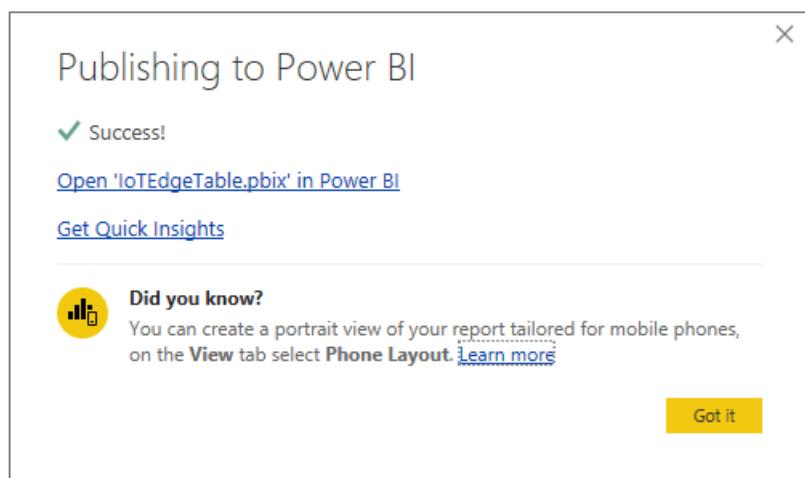
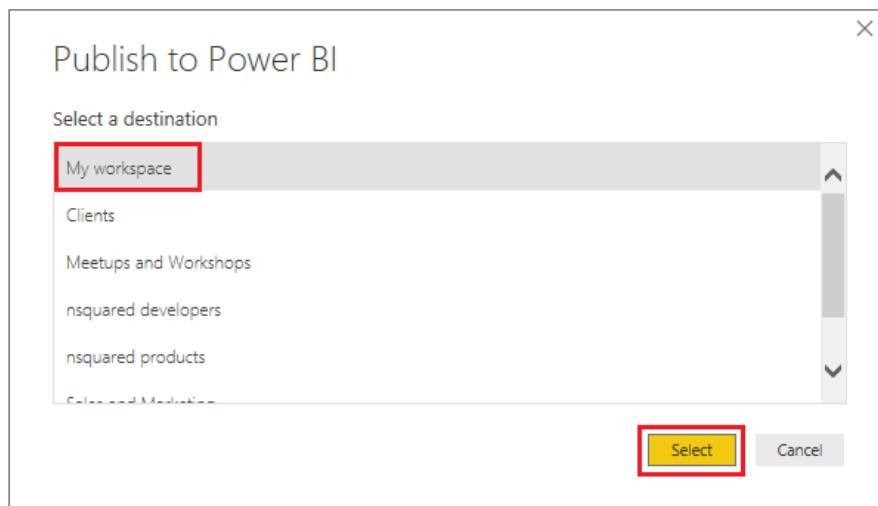
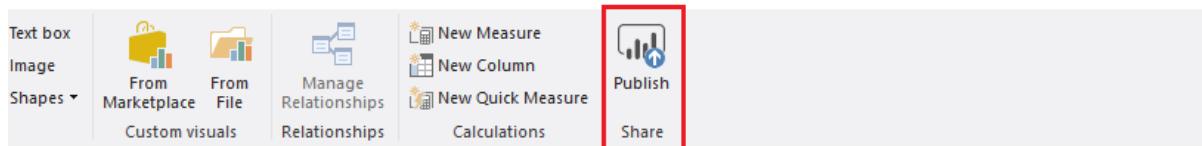
13. Once it has finished loading the query, within the **FIELDS panel**, on the right side of the screen, tick the boxes corresponding to the parameters **Name** and **Value**, to visualize the **MessageContent** column content.

This screenshot shows the Microsoft Power BI Desktop interface. The ribbon has various options like Cut, Copy, Paste, etc. The main area contains a table visual with two columns: 'Name' and 'Value'. The 'Value' column shows the text 'MessageContent - Message from IoT Edge Device!'. To the right is the 'FIELDS' panel, which lists the 'IoTMessages' query with 'Name' and 'Value' checked. Other sections like 'VISUALIZATIONS' and 'FILTERS' are also visible.

14. Click on the **blue disk icon** at the top left of the window to save your work in a folder of your choice.



15. You can now click on the Publish button to upload your table to your Workspace. When prompted, click **My workspace** and click *Select*. Wait for it to display the successful result of the submission.

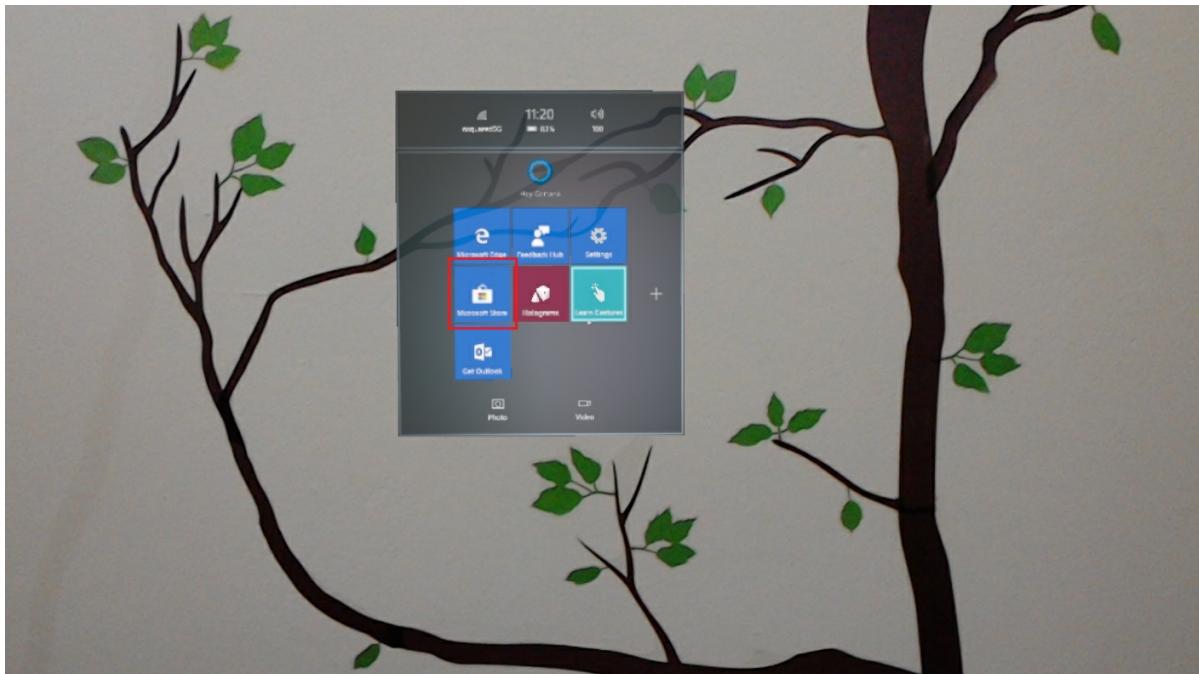


WARNING

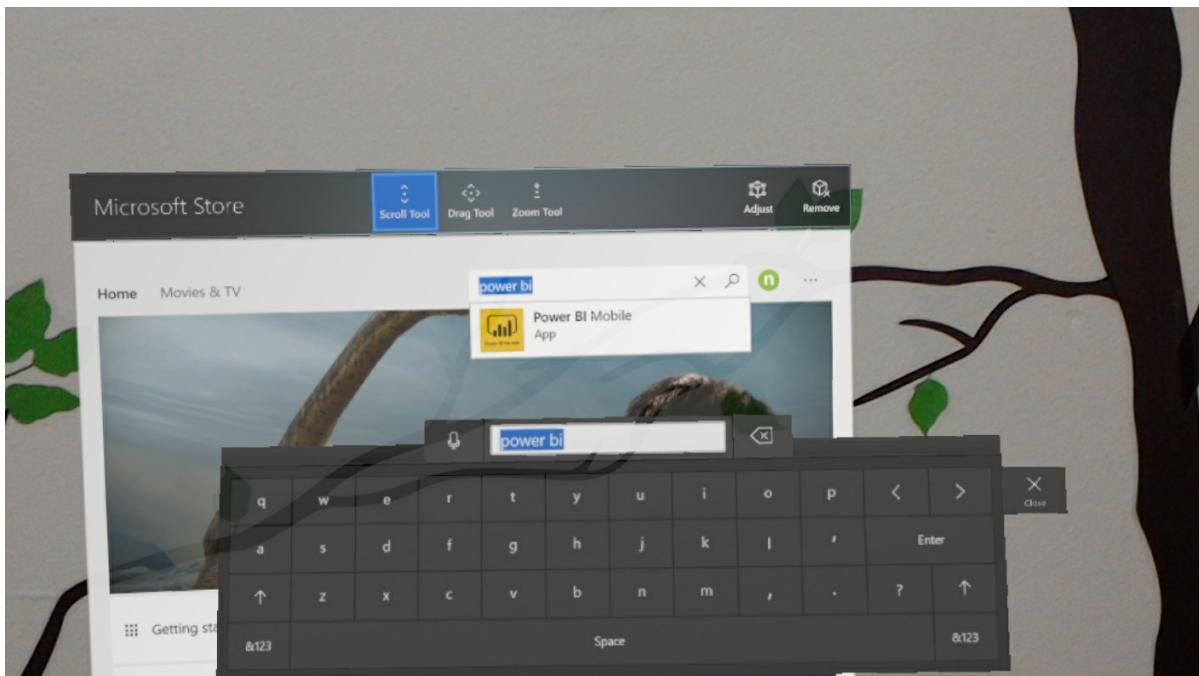
The following Chapter is HoloLens specific. Power BI is not currently available as an immersive application, however you can run the desktop version in the Windows Mixed Reality Portal (aka Cliff House), through the Desktop app.

Chapter 16 - Display Power BI data on HoloLens

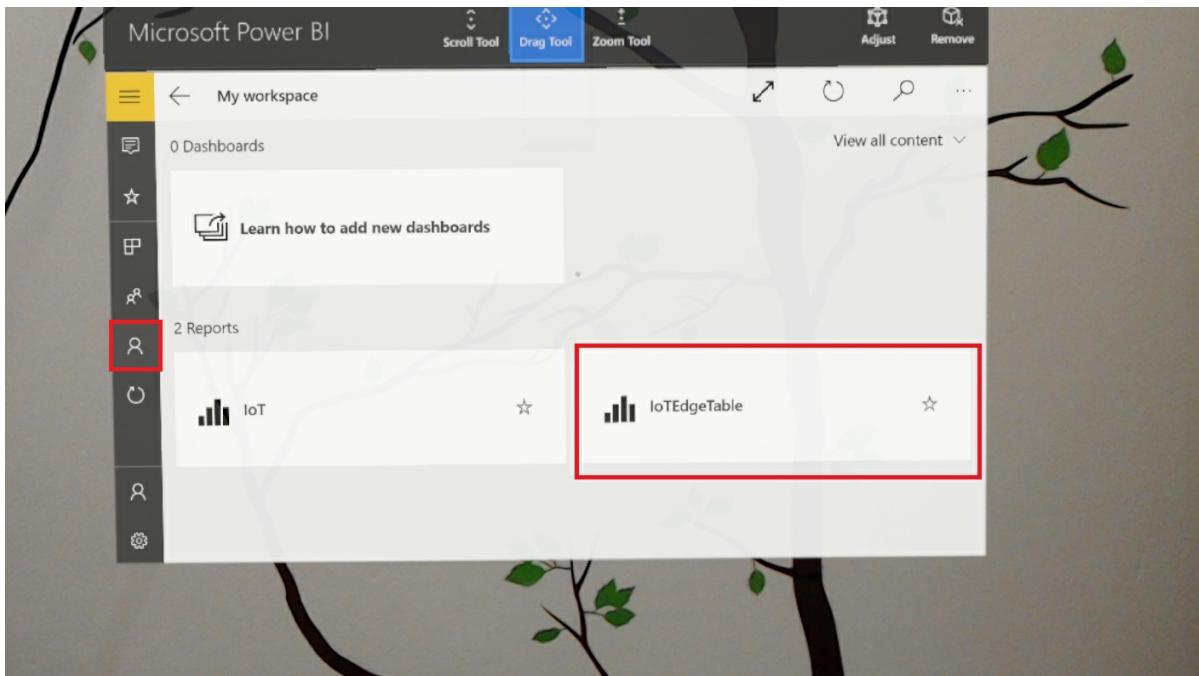
1. On your HoloLens, log in to the **Microsoft Store**, by tapping on its icon in the applications list.



2. Search and then download the **Power BI** application.

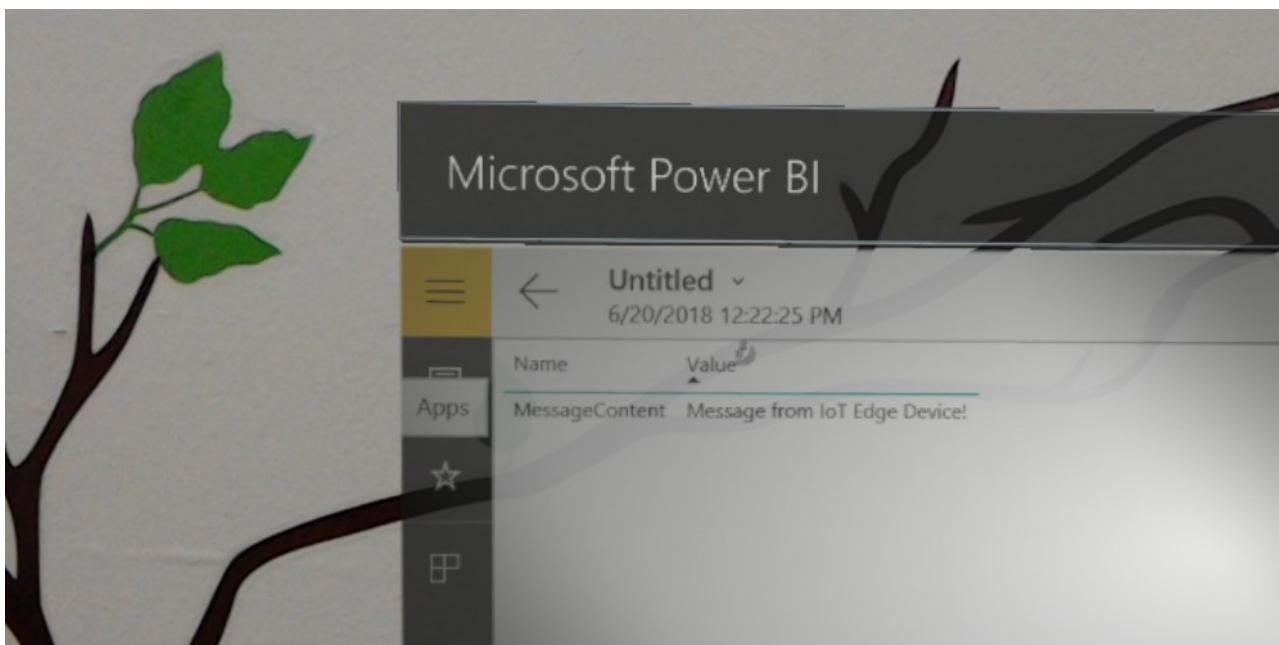


3. Start **Power BI** from your applications list.
4. **Power BI** might ask you to login to your **Microsoft 365 account**.
5. Once inside the app, the workspace should display by default as shown in the image below. If that does not happen, simply click on the workspace icon on the left side of the window.



Your finished your IoT Hub application

Congratulations, you have successfully created an IoT Hub Service, with a simulated Virtual Machine Edge device. Your device can communicate the results of a machine learning model to an Azure Table Service, facilitated by an Azure Function App, which is read into Power BI, and visualized within a Microsoft HoloLens.



Bonus exercises

Exercise 1

Expand the messaging structure stored in the table and display it as a graph. You might want to collect more data and store it in the same table, to be later displayed.

Exercise 2

Create an additional "camera capture" module to be deployed on the IoT board, so that it can capture images through the camera to be analyzed.



This guidance is authored by Microsoft designers, developers, program managers, and researchers, whose work spans holographic devices (like HoloLens) and immersive devices (like the Acer and HP Windows Mixed Reality headsets). So, consider this work as a set of topics for 'how to design for Windows head-mounted displays'.

Article categories



Get started with Design

[What is mixed reality?](#)

[About this guidance](#)

[My first year on the design team](#)

[Expanding the design process for mixed reality](#)

[The pursuit of more personal computing](#)



Interaction design

[Interaction fundamentals](#)

[Comfort](#)

[Gaze targeting](#)

[Gestures](#)

[Voice design](#)



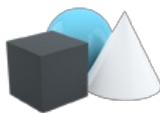
Style

[Color, light and materials](#)

[Spatial sound design](#)

[Typography](#)

[Scale](#)



App patterns

[Types of mixed reality apps](#)

[Room scan visualization](#)

[Cursors](#)

[Billboarding and tag-along](#)



Controls

[Text in Unity](#)

[Interactable object](#)

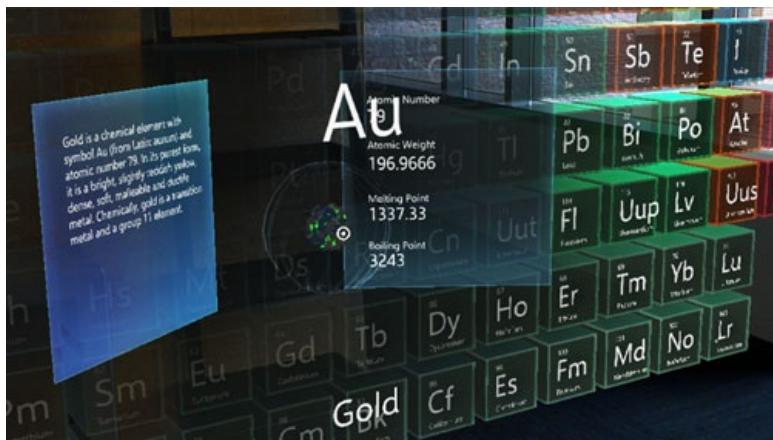
[Object collection](#)

[Displaying progress](#)

[App bar and bounding box](#)

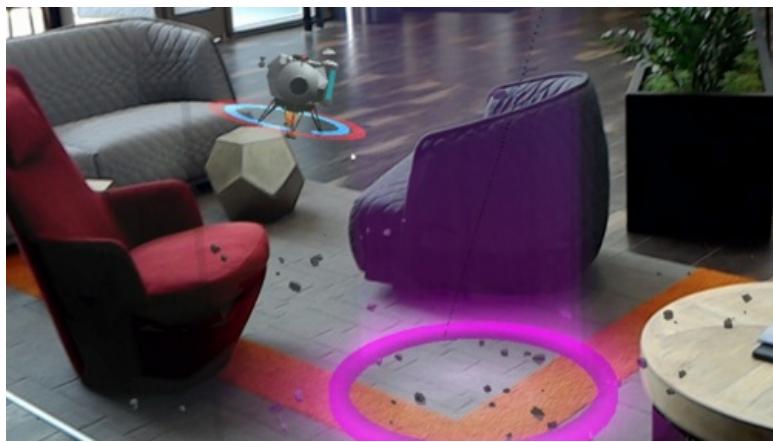
Sample apps

Build great experiences from samples designed and created by our team.



Periodic Table of the Elements

Learn how to lay out an array of objects in 3D space with various surface types using an Object collection.



Lunar Module

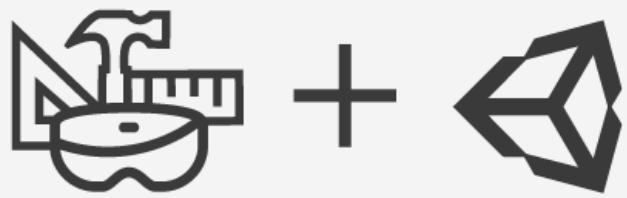
Learn how to extend HoloLens base gestures with two-handed tracking and Xbox controller input.



Galaxy Explorer

The Galaxy Explorer Project is ready. You shared your ideas with the community, chose an app, watched a team build it, and can now get the source code.

Design tools



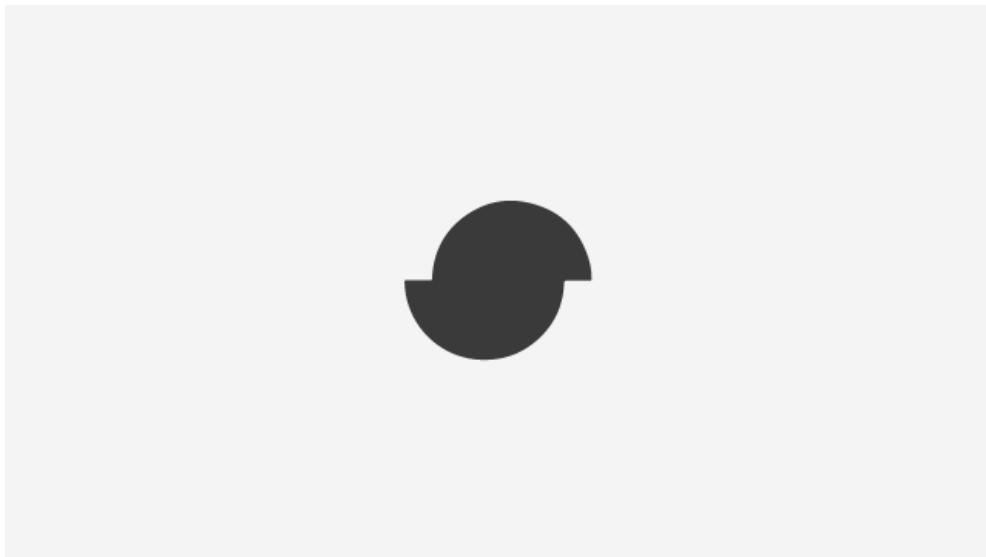
Mixed Reality Toolkit - Unity



Mixed Reality Toolkit

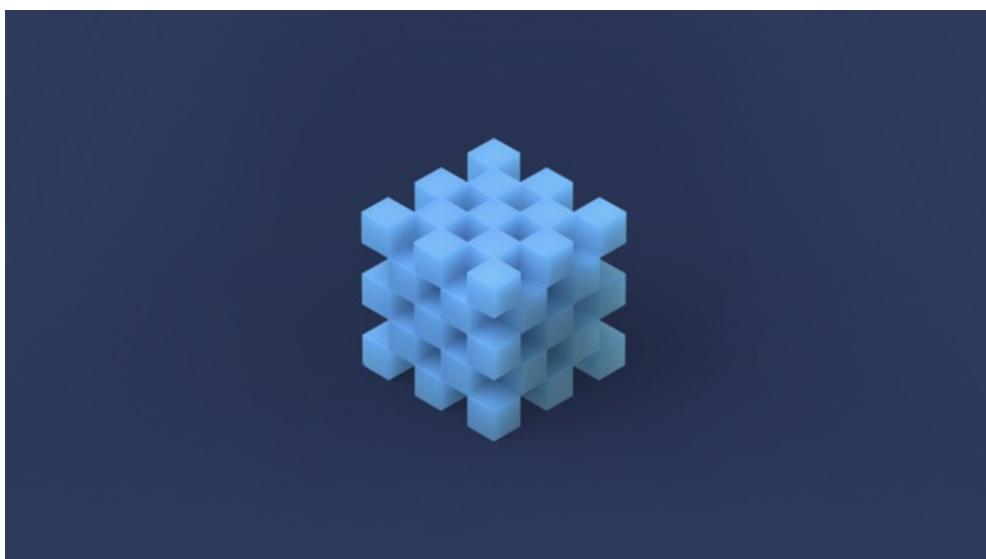


HoloSketch



Simplygon

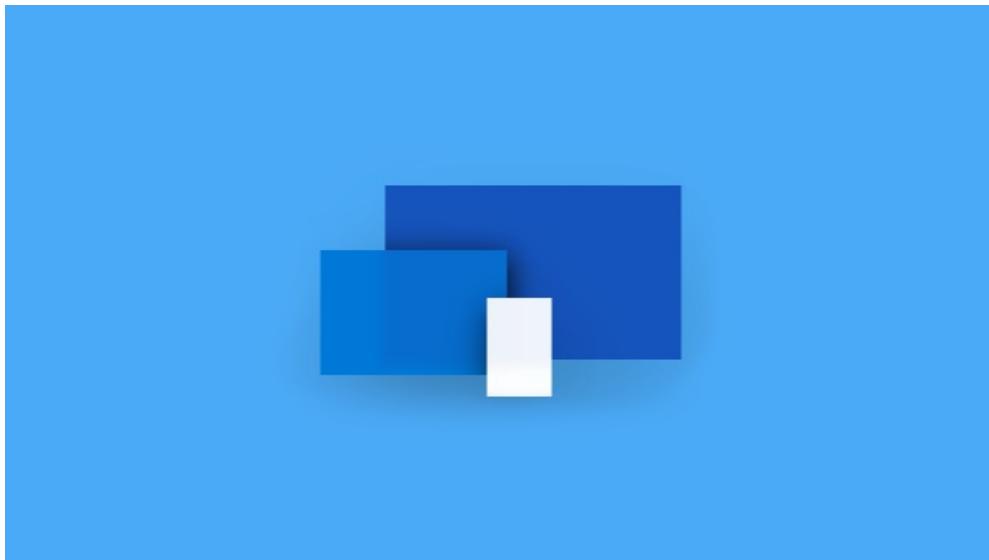
General design resources



Fluent Design System



Inclusive design at Microsoft



Universal Windows Platform (UWP) app design

About this design guidance

11/6/2018 • 6 minutes to read • [Edit Online](#)

Introduction

Hello, and welcome to your design guidance for mixed reality.

This guidance is authored by Microsoft designers, developers, program managers, and researchers, whose work spans holographic devices (like HoloLens) and immersive devices (like the Acer and HP Windows Mixed Reality headsets). So, consider this work as a set of topics for 'how to design for Windows head-mounted displays'.

With you, we are entering a tremendously exciting new era of computing. Breakthroughs in head mounted displays, spatial sound, sensors, environmental awareness, input, and 3D graphics lead us, and challenge us, to define new types of experiences... a new frontier that is dramatically more personal, intuitive, immersive, and contextual.

Wherever possible, we will offer actionable design guidance, with related code on GitHub. That said, because we are learning right along with you, we won't always be able to offer specific, actionable guidance here. Some of what we share will be in the spirit of 'lessons we've learned' and 'avoid going down that path'.

And we know many innovations will be generated by the larger design community, so we look forward to hearing from you, learning from you, and working closely with you. For our part, we'll do our best to share our insights, even if they are exploratory and early, with the intent of empowering developers and designers with design thinking, best practices, and the related open source controls, patterns, and sample apps that you can use directly in your own work.

Overview

Here's a quick overview of how this design guidance is organized. You'll find sections for each of these areas, with links to multiple articles.

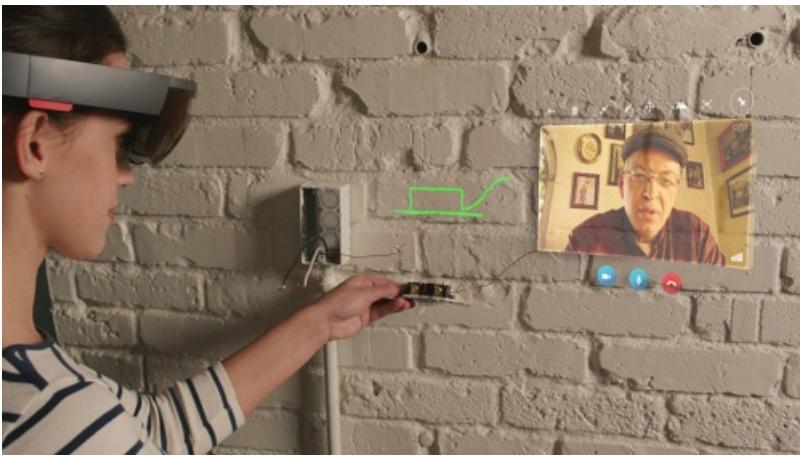
- **Get started with design** - Read our high-level thoughts and understand the principles we follow.
- **Interaction design** - Learn about input, commanding, navigation, and other interaction basics for designing your apps.
- **Style** - Make your app delightful by using color, typography, and motion.
- **App patterns** - Learn how apps can span scenarios across immersive and real world environments.
- **Controls** - Use controls and patterns as building blocks to create your own app experience.
- **Sample apps** - Build great experiences from samples designed and created by our team.
- **Design tools and resources** - Jump-start your project with design templates and tools.

For all the above, we aim to deliver the right mix of text, illustrations and diagrams, and videos, so you'll see us experimenting with different formats and techniques, all with the intent of delivering what you need. And in the months ahead, we'll expand this taxonomy to include a broader set of design topics. Whenever possible, we'll give you a heads-up about what is coming next, so please keep checking back.

Objectives

Here's a quick look at some high-level objectives that are guiding this work so you can understand where we're coming from:

Help solve customer challenges



We wrestle with many of the same issues that you do, and we understand how challenging your work is. It's exciting to explore and define a new frontier... and it can also be daunting. Old paradigms and practices need to be re-thought; customer need new experiences; and there is so much potential for innovation. Given that, we want this work to be as comprehensive as possible, moving well beyond a style guide. We aim to deliver a comprehensive set of design guidance that covers mixed reality interaction, commanding, navigation, input, and style – all grounded in human behavior and scenarios.

Point the way towards a new, more human way of computing



While it is important to focus on specific customer problems, we also want to push ourselves to think beyond that, and to deliver more. We believe great design is not "just" problem-solving, but also a way to meaningfully activate human evolution. New ways of human behavior; new ways of people relating to themselves, their activities, and their environments; new ways of seeing our world... we want our guidance to reflect all these more aspirational ways of thinking too.

Meet creators where they are



We hope many audiences find this guidance to be helpful. You have different skill-sets (beginning, intermediate, advanced), use different tools (Unity, DirectX, C++, C#, other), are familiar with various platforms (Windows, iOS, Android), come from different backgrounds (mobile, enterprise, gaming), and are working on different size teams (solo, small, medium, large). So, this guidance will be viewed with different perspectives and needs. Whenever possible, we will try to keep this diversity in mind, and make our guidance as relevant as possible to as many people as possible. In addition, we know that many of you are already on GitHub, so we will directly link to GitHub repos and forums to meet you where you already are.

Share as much as possible, from experimental to explicit



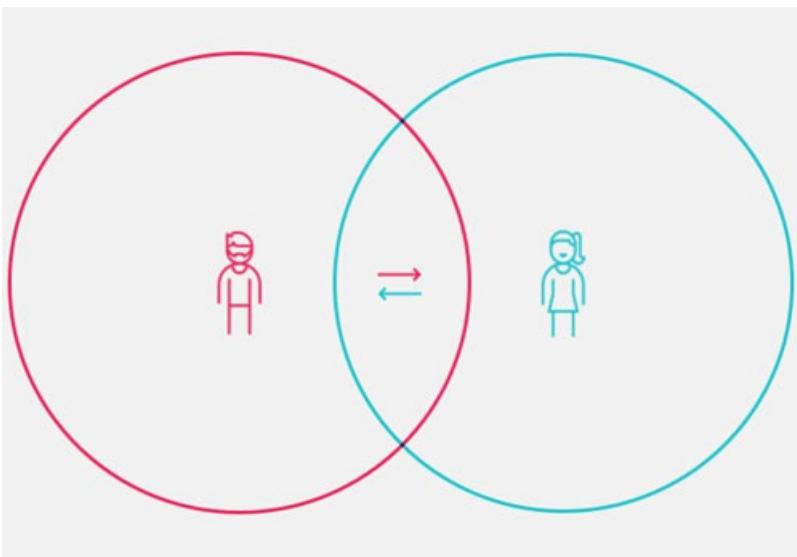
One of the challenges of offering design guidance in this new 3D medium is that we don't always have definitive guidance to offer. Just like you, we are learning, experimenting, prototyping, problem-solving, and adjusting as we hit obstacles. Rather than wait for some mythical future moment when we have it all figured out, we aim to share our thinking with you real time, even if it is not conclusive. Of course, our end goal is to be definitive wherever we can, providing clear, flexible design guidance tied to open-source code, and actionable in Microsoft dev and design tools. But getting to that point takes many rounds of iteration and learning. We want to engage with you, and learn with you, along the way, so we will be doing the best we can to share as we go, even our stuff that is experimental.

The right balance of global and local design



We'll offer two levels of design guidance: global and local. Our 'global' design guidance is embodied in the [Fluent Design System](#). Fluent details how we think about fundamentals like light, depth, motion, material, and scale across all Microsoft design – our devices, products, tools, and services. That said, significant device-specific differences exist across this larger system, so our 'local' design guidance for head-mounted displays will describe designing for holographic and immersive devices that often have different input and output methods, and different user needs and scenarios. So the local design guidance covers topics unique to HMDs, for example 3D environments and objects; shared environments; the use of sensors, eye tracking and spatial mapping; and the opportunities of spatial audio. Throughout our guidance you will likely see us refer to both these global and the local aspects, and hopefully this will help you ground your work in a larger foundation of design while taking advantage of the design differences of specific devices.

Have a discussion



Perhaps most importantly, we want to engage with you, the community of holographic and immersive designers and developers, to define this exciting new era of design. As mentioned above, we know we don't have all the answers, and we believe many exciting solutions and innovations will come from you. We aim to be open and available to hear about them, and discuss with you online and at events, and add value wherever we can. We are excited to be a part of this amazing design community, embarking on an adventure together.

Please dive in

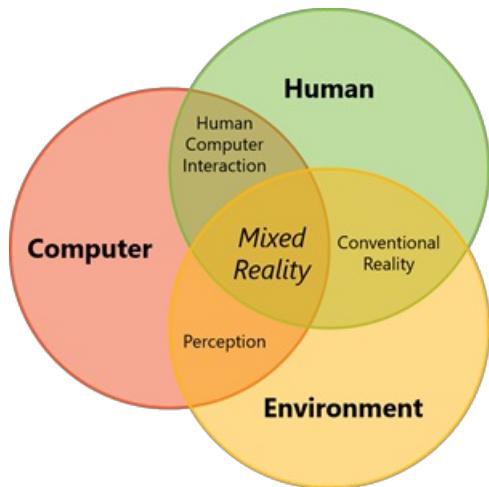
We hope this Welcome article provides some meaningful context as you explore our design guidance. Please dive in, and let us know your thoughts in the GitHub forums you'll find linked in our articles, or at Microsoft Design on [Twitter](#) and [Facebook](#). Let's co-design the future together!

What is mixed reality?

11/6/2018 • 6 minutes to read • [Edit Online](#)

Mixed reality is the result of blending the physical world with the digital world. Mixed reality is the next evolution in human, computer, and environment interaction and unlocks possibilities that before now were restricted to our imaginations. It is made possible by advancements in computer vision, graphical processing power, display technology, and input systems. The term *mixed reality* was originally introduced in a 1994 paper by Paul Milgram and Fumio Kishino, "[A Taxonomy of Mixed Reality Visual Displays](#)." Their paper introduced the concept of the *virtuality continuum* and focused on how the categorization of taxonomy applied to displays. Since then, the application of mixed reality goes beyond displays but also includes environmental input, spatial sound, and location.

Environmental input and perception



Over the past several decades, the relationship between human input and computer input has been well explored. It even has a widely studied discipline known as *human computer interaction* or HCI. Human input happens through a variety of means including keyboards, mice, touch, ink, voice, and even Kinect skeletal tracking.

Advancements in sensors and processing are giving rise to a new area of computer input from environments. The interaction between computers and environments is effectively environmental understanding, or *perception*. Hence the API names in Windows that reveal environmental information are called the [perception APIs](#). Environmental input captures things like a person's position in the world (e.g. [head tracking](#)), surfaces and boundaries (e.g. [spatial mapping](#) and [spatial understanding](#)), ambient lighting, environmental sound, object recognition, and location.

Now, the combination of all three – computer processing, human input, and environmental input – sets the opportunity to create true mixed reality experiences. Movement through the physical world can translate to movement in the digital world. Boundaries in the physical world can influence application experiences, such as game play, in the digital world. Without environmental input, experiences cannot blend between the physical and digital realities.

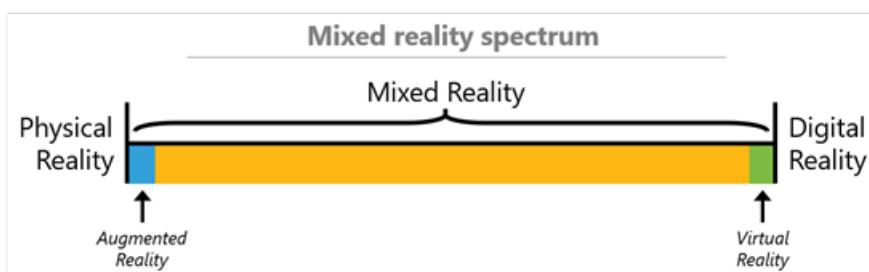
The mixed reality spectrum

Since mixed reality is the blending of the physical world and digital world, these two realities define the polar ends of a spectrum known as the *virtuality continuum*. For simplicity, we refer to this as the *mixed reality spectrum*. On the left-hand side we have physical reality in which we, humans, exist. Then on the right-hand side

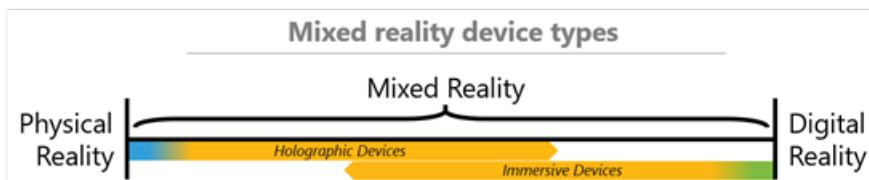
we have the corresponding digital reality.

Most mobile phones on the market today have little to no environmental understanding capabilities. Thus the experiences they offer cannot mix between physical and digital realities. The experiences that overlay graphics on video streams of the physical world are *augmented reality*, and the experiences that occlude your view to present a digital experience are *virtual reality*. As you can see, the experiences enabled between these two extremes is *mixed reality*:

- Starting with the physical world, placing a digital object, such as a hologram, as if it was really there.
- Starting with the physical world, a digital representation of another person – an avatar – shows the location where they were standing when leaving notes. In other words, experiences that represent asynchronous collaboration at different points in time.
- Starting with a digital world, physical boundaries from the physical world, such as walls and furniture, appear digitally within the experience to help users avoid physical objects.



Most augmented reality and virtual reality offerings available today represent a very small part of this spectrum. They are, however, subsets of the larger mixed reality spectrum. Windows 10 is built with the entire spectrum in mind, and allows blending digital representations of people, places and things with the real world.



There are two main types of devices that deliver Windows Mixed Reality experiences:

1. **Holographic devices.** These are characterized by the device's ability to place digital content in the real world as if it were really there.
2. **Immersive devices.** These are characterized by the device's ability to create a sense of "presence" – hiding the physical world and replacing it with a digital experience.

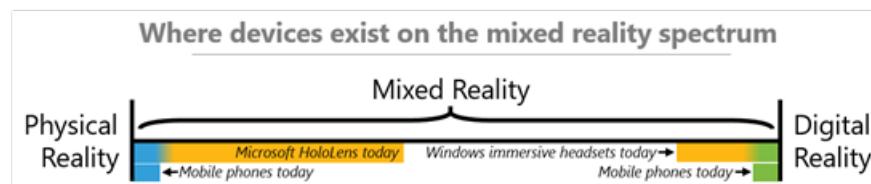
CHARACTERISTIC	HOLOGRAPHIC DEVICES	IMMERSIVE DEVICES
Example Device	Microsoft HoloLens 	Acer Windows Mixed Reality Development Edition 

Display	<i>See-through display.</i> Allows user to see physical environment while wearing the headset.	<i>Opaque display.</i> Blocks out the physical environment while wearing the headset.
Movement	Full six-degrees-of-freedom movement, both rotation and translation.	Full six-degrees-of-freedom movement, both rotation and translation.

Note, whether a device is connected to or tethered to a separate PC (via USB cable or Wi-Fi) or self-contained (untethered) does not reflect whether a device is holographic or immersive. Certainly, features that improve mobility lead to better experiences and both holographic and immersive devices could be tethered or untethered.

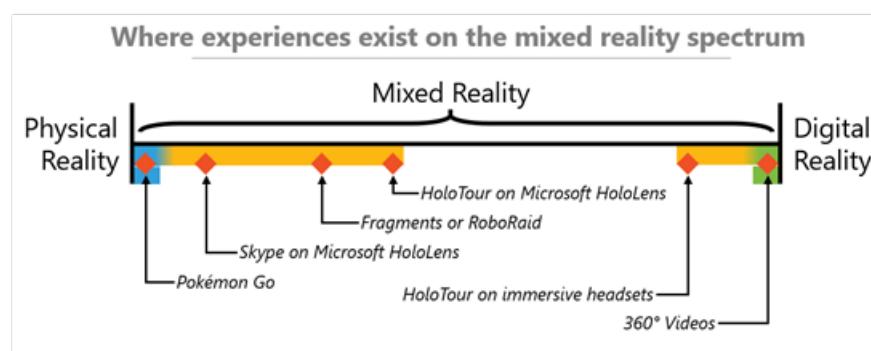
Devices and experiences

Technological advancement is what has enabled mixed reality experiences. There are no devices today that can run experiences across the entire spectrum; however, Windows 10 provides a common mixed reality platform for both device manufacturers and developers. Devices today can support a specific range within the mixed reality spectrum, and over time new devices should expand that range. In the future, holographic devices will become more immersive, and immersive devices will become more holographic.



Often, it is best to think what type of experience an app or game developer wants to create. The experiences will typically target a specific point or part on the spectrum. Then, developers should consider the capabilities of devices they want to target. For example, experiences that rely on the physical world will run best on HoloLens.

- **Towards the left (near physical reality).** Users remain present in their physical environment and are never made to believe they have left that environment.
- **In the middle (fully mixed reality).** These experiences perfectly blend the real world and the digital world. Viewers who have seen the movie *Jumanji* can reconcile how the physical structure of the house where the story took place was blended with a jungle environment.
- **Towards the right (near digital reality).** Users experience a completely digital environment and are oblivious to what occurs in the physical environment around them.



Here's how different experiences take advantage of their position on the mixed reality spectrum:

- **Skype on Microsoft HoloLens.** This experience allows collaboration through drawing in someone's physical environment. As an experience, it is currently further left on the spectrum because the physical environment remains the location of activity.
- **Fragments and RoboRaid.** Both of these take advantage of the layout of the user's physical environment, walls, floors, furniture to place digital content in the world. The experience moves further to the right on the spectrum, but the user always believes they are in their physical world.

- **HoloTour on Microsoft HoloLens.** HoloTour is designed with an immersive experience in mind. Users are meant to walk around tourist locations. On HoloLens, HoloTour pushes the limits of the device's immersive capabilities.
- **HoloTour on immersive devices.** Meanwhile when HoloTour runs on an immersive device, it showcases the environmental input by allowing users to walk around the tourist location. The boundaries that help users avoid walking into walls represent further capabilities that pull the experience towards the middle.
- **360° videos.** Since environmental input like translational movement does not affect the video playback experience, these experiences fall to the far right towards digital reality, effectively fitting into the narrow part of the spectrum that is virtual reality.

Skype for HoloLens, Fragments and RoboRaid are best experienced with HoloLens. Likewise, 360° video is best experienced on immersive devices. HoloTour showcases the best of what both types of devices can do today when trying to push towards the center experience of mixed reality.

See also

- [API Reference: Windows.Perception](#)
- [API Reference: Windows.Perception.Spatial](#)
- [API Reference: Windows.Perception.Spatial.Surfaces](#)

Case study - My first year on the HoloLens design team

11/6/2018 • 8 minutes to read • [Edit Online](#)

My journey from a 2D flatland to the 3D world started when I joined the HoloLens design team in January, 2016. Before joining the team, I had very little experience in 3D design. It was like the Chinese proverb about a journey of a thousand miles beginning with a single step, except in my case that first step was a leap!



Taking the leap from 2D to 3D

"I felt as though I had jumped into the driver's seat without knowing how to drive the car. I was overwhelmed and scared, yet very focused."

— Hae Jin Lee

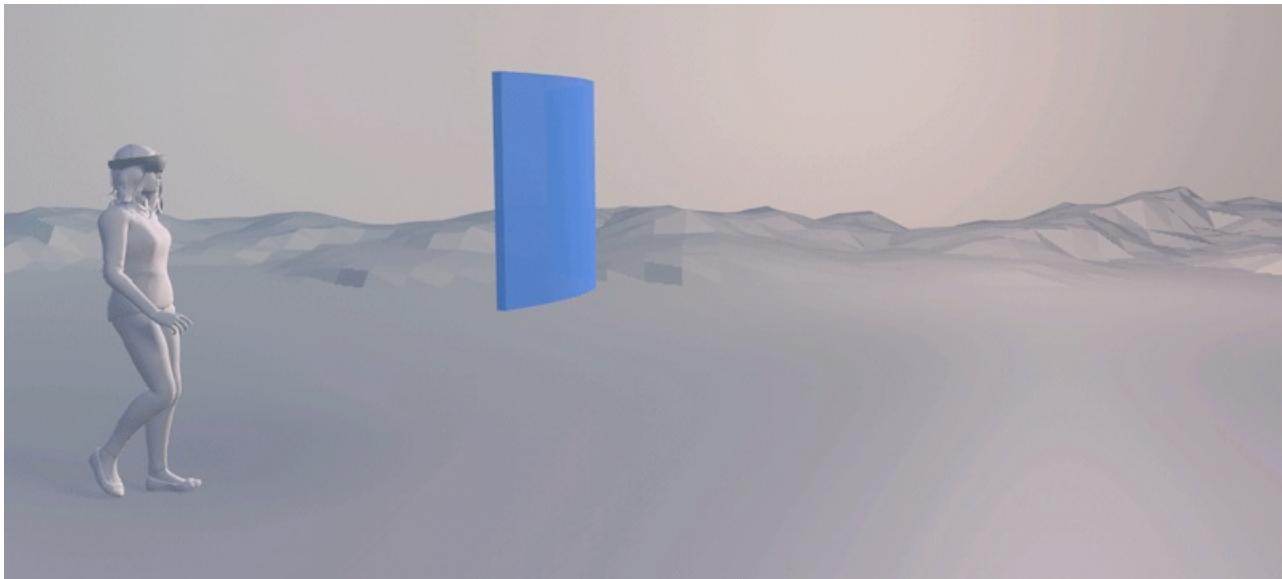
During the past year, I picked up skills and knowledge as fast as I could, but I still have a lot to learn. Here, I've written up 4 observations with a video tutorial documenting my transition from a 2D to 3D interaction designer. I hope my experience will inspire other designers to take the leap to 3D.

Good-bye frame. Hello spatial / diegetic UI

Whenever I designed posters, magazines, websites, or app screens, a defined frame (usually a rectangle) was a constant for every problem. Unless you are reading this post in a HoloLens or other VR device, you are *looking at this from the outside* through 2D screen safely guarded within a frame. Content is external to you. However, Mixed Reality headset *eliminates the frame*, so you are within the content space, looking and walking through the content from inside-out.

I understood this conceptually, but in the beginning I made the mistake of simply transferring 2D thinking into 3D space. That obviously didn't work well because 3D space has its own unique properties such as a view change (based on user's head movement) and [different requirement for user comfort](#) (based on the properties of the devices and the humans using them). For example, in a 2D UI design space, locking UI elements into the corner of a screen is a very common pattern, but this HUD (Head Up Display) style UI does not feel natural in MR/VR experiences; it hinders user's immersion into the space and causes user discomfort. It's like having an annoying

dust particle on your glasses that you are dying to get rid of. Over time, I learned that it feels more natural to position content in 3D space and add body-locked behavior that makes the content follow the user at a relative fixed distance.



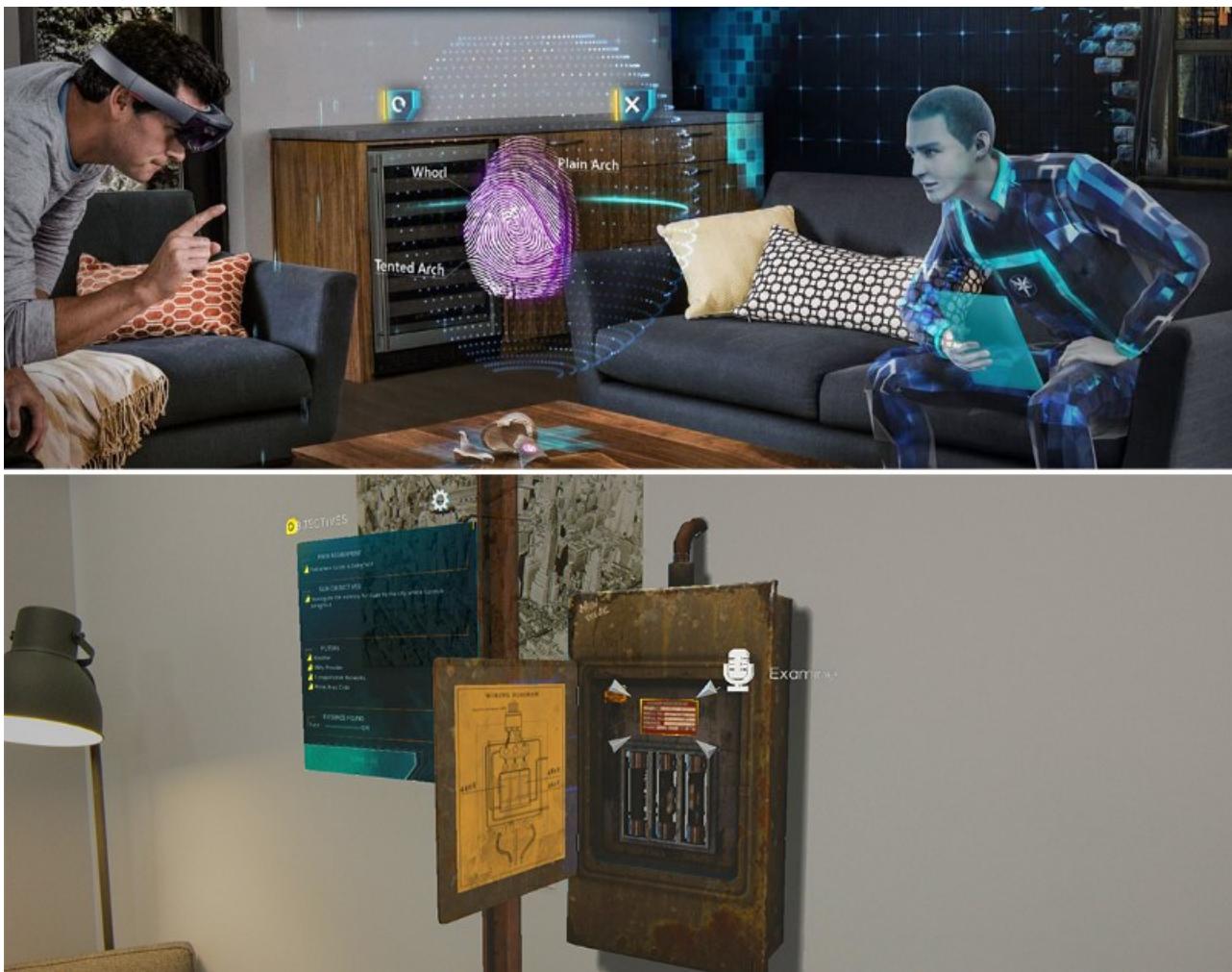
Body-locked



World-locked

Fragments: An example of great Diegetic UI

[Fragments](#), a first-person crime thriller developed by [Asobo Studio](#) for HoloLens demonstrates a great Diegetic UI. In this game, the user becomes a main character, a detective who tries to solve a mystery. The pivotal clues to solve this mystery get sprinkled in the user's physical room and are often times embedded inside a fictional object rather than existing on their own. This diegetic UI tends to be less discoverable than body-locked UI, so the Asobo team cleverly used many cues including virtual characters' gaze direction, sound, light, and guides (e.g., arrow pointing the location of the clue) to grab user's attention.



Fragments - Diegetic UI examples

Observations about diegetic UI

Spatial UI (both body-locked and world-locked) and diegetic UI have their own strengths and weaknesses. I encourage designers to try out as many MR/VR apps as possible, and to develop their own understanding and sensibility for various UI positioning methods.

The return of skeuomorphism and magical interaction

Skeuomorphism, a digital interface that mimics the shape of real world objects has been “uncool” for the last 5–7 years in the design industry. When Apple finally gave way to flat design in iOS 7, it seemed like Skeuomorphism was finally dead as an interface design methodology. But then, a new medium, MR/VR headset arrived to the market and it seems like Skeuomorphism returned again. :)

Job Simulator: An example of skeuomorphic VR design

[Job Simulator](#), a whimsical game developed by [Owlchemy Labs](#) is one of the most popular example for skeuomorphic VR design. Within this game, players are transported into future where robots replace humans and humans visit a museum to experience what it feels like to perform mundane tasks at one of four different jobs: Auto Mechanic, Gourmet Chef, Store Clerk, or Office Worker.

The benefit of Skeuomorphism is clear. Familiar environments and objects within this game help new VR users feel more comfortable and present in virtual space. It also makes them feel like they are in control by associating familiar knowledge and behaviors with objects and their corresponding physical reactions. For example, to drink a cup of coffee, people simply need to walk to the coffee machine, press a button, grab the cup handle and tilt it towards their mouth as they would do in the real world.



Job Simulator

Because MR/VR is still a developing medium, using a certain degree of skeuomorphism is necessary to demystify MR/VR technology and to introduce it to larger audiences around the world. Additionally, using skeuomorphism or realistic representation could be beneficial for specific types of applications like surgery or flight simulation. Since the goal of these apps is to develop and refine specific skills that can be directly applied in the real world, the closer the simulation is to the real world, the more transferable the knowledge is.

Remember that skeuomorphism is only one approach. The potential of the MR/VR world is far greater than that, and designers should strive to create magical hyper-natural interactions — new affordances that are uniquely possible in MR/VR world. As a start, consider adding magical powers to ordinary objects to enable users to fulfill their fundamental desires—including teleportation and omniscience.



Doraemon's magical door (left) and ruby slippers(right)

Observations about skeuomorphism in VR

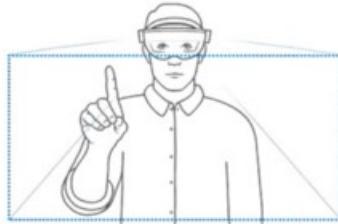
From "Anywhere door" in Doraemon, "Ruby Slippers" in The Wizard of Oz to "Maurader's map" in Harry Potter, examples of ordinary objects with magical power abound in popular fiction. These magical objects help us visualize a connection between the real-world and the fantastic, between what is and what could be. Keep in mind that when designing the magical or surreal object one needs to strike a balance between functionality and entertainment. Beware of the temptation to create something purely magical just for novelty's sake.

Understanding different input methods

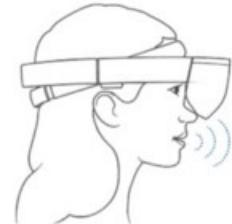
When I designed for the 2D medium, I had to focus on touch, mouse, and keyboard interactions for inputs. In the MR/VR design space, our body becomes the interface and users are able to use a broader selection of input methods: including speech, gaze, gesture, [6-dof controllers](#), and gloves that afford more intuitive and direct connection with virtual objects.



Gaze



Gesture



Voice

Available inputs in HoloLens

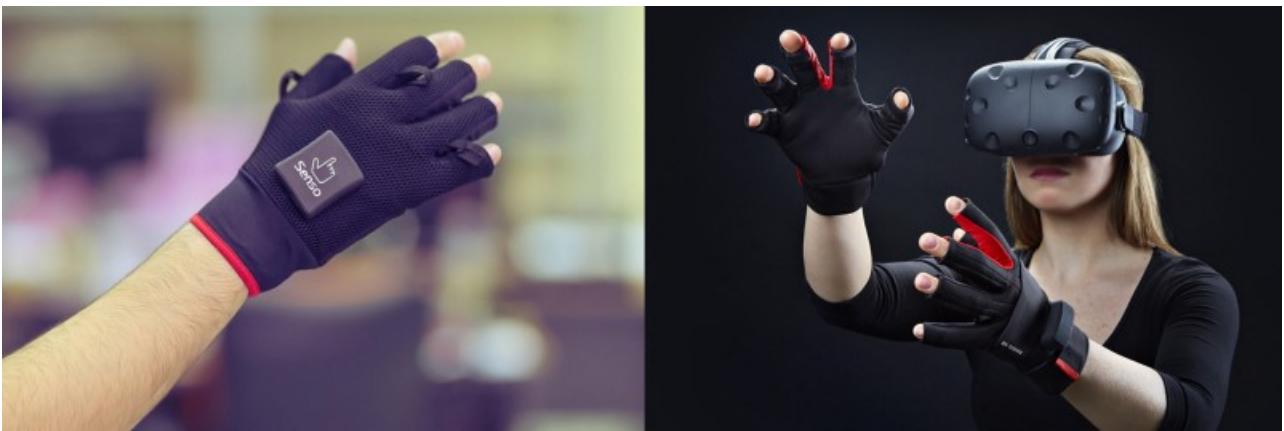
"Everything is best for something, and worst for something else."

— [Bill Buxton](#)

For example, gesture input using bare hand and camera sensors on an HMD device frees users hand from holding controllers or wearing sweaty gloves, but frequent use can cause physical fatigue (a.k.a gorilla arm). Also, users have to keep their hands within the line of sight; if the camera cannot see the hands, the hands cannot be used.

Speech input is good at traversing complex tasks because it allows users to cut through nested menus with one command (e.g., "Show me the movies made by Laika studio.") and also very economical when coupled with other modality (e.g., "Face me" command orients the hologram a user is looking at towards the user). However, speech input may not work well in noisy environment or may not appropriate in a very quiet space.

Besides gesture and speech, hand-held tracked controllers (e.g., Oculus touch, Vive, etc.) are very popular input methods because they are easy to use, accurate, leverage people's [proprioception](#), and provide passive haptic cues. However, these benefits come at the cost of not being able to be bare-hands and use full finger tracking.



Senso (Left) and Manus VR (Right)

While not as popular as controllers, gloves are gaining momentum again thanks to the MR/VR wave. Most recently, brain/mind input have started to gain traction as an interface for virtual environments by integrating EEG or EMG sensor to headset (e.g., [MindMaze VR](#)).

Observations about input methods

These are just a sample of input devices available in the market for MR/VR. They will continue to proliferate until the industry matures and agrees upon best practices. Until then, designers should remain aware of new input

devices and be well-versed in the specific input methods for their particular project. Designers need to look for creative solutions inside of limitations, while also playing to a device's strengths.

Sketch the scene and test in the headset

When I worked in 2D, I mostly sketched just the content. However, in mixed reality space that wasn't sufficient. I had to sketch out the entire scene to better imagine the relationships between the user and virtual objects. To help my spatial thinking, I started to sketch scenes in [Cinema 4D](#) and sometimes create simple assets for prototyping in [Maya](#). I had never used either program before joining the HoloLens team and I am still a newbie, but working with these 3D programs definitely helped me get comfortable with new terminology, such as [shader](#) and [IK \(inverse kinematics\)](#).

"No matter how closely I sketched out the scene in 3D, the actual experience in headset was almost never the same as the sketch. That's why it's important to test out the scene in the target headsets." — **Hae Jin Lee**

For HoloLens prototyping, I tried out all the tutorials at [Mixed Reality Academy](#) to start. Then I began to play with [HoloToolkit.Unity](#) that Microsoft provides to developers to accelerate development of holographic applications. When I got stuck with something, I posted my question to [HoloLens Question & Answer Forum](#).

After acquiring basic understanding of HoloLens prototyping, I wanted to empower other non-coders to prototype on their own. So I made a video tutorial that teaches how to develop a simple projectile using HoloLens. I briefly explain the basic concepts, so even if you have zero experience in HoloLens development, you should be able to follow along.

I made this simple tutorial for non-programmers like myself.

For VR prototyping, I took courses at [VR Dev School](#) and also took [3D Content Creation for Virtual Reality](#) at Lynda.com. VR Dev school provided me more in depth knowledge in coding and the Lynda course offered me a nice short introduction to creating assets for VR.

Take the leap

A year ago, I felt like all of this was a bit overwhelming. Now I can tell you that it was 100% worth the effort. MR/VR is still very young medium and there are so many interesting possibilities waiting to be realized. I feel inspired and fortunate be able to play one small part in designing the future. I hope you will join me on the journey into 3D space!

About the author



Hae Jin Lee

UX Designer @Microsoft

Case study: Expanding the design process for mixed reality

11/6/2018 • 11 minutes to read • [Edit Online](#)

As Microsoft launched the HoloLens to an audience of eager developers in 2016, the team had already partnered with studios inside and outside of Microsoft to build the device's launch experiences. These teams learned by doing, finding both opportunities and challenges in the new field of mixed reality design.

To help new teams and partners innovate more effectively, we turned their methods and insights into a curriculum of design and development lessons that we teach to developers in our Mixed Reality Academy (including week-long design workshops we offer to our enterprise partners).

Below is a snapshot of these learnings, part of a larger process we use to help our enterprise partners prepare their teams for mixed reality development. While many of these methods were originally targeted for HoloLens development, ideating and communicating in 3D are critical to the full spectrum of mixed reality experiences.

Thinking spatially during the design process

Any design process or design framework is meant to iterate thinking: To approach a problem broadly, to share ideas with others effectively, to evaluate those ideas and reach a solution. Today, we have well-established design and development methods for building experiences on desktops, phones, and tablets. Teams have clear expectations of what is necessary to iterate an idea and turn it into a product for users.

Often teams are a mix of development, design, research, and management, but all roles can participate in the design process. The barrier to entry to contribute an idea for a mobile app can be as simple as drawing a rectangle for the device's screen. While sketching boxes and lines for the UI elements, with arrows to indicate motion or interactions, can be enough to establish technical requirements or define potential user behavior.

With mixed reality, the traditional 2D design process begins to break down: sketching in 3D is difficult for most people and using 2D tools, like pen and paper or whiteboards, can often limit ideas to those dimensions. Meanwhile 3D tools, built for gaming or engineering, require a high degree of proficiency to quickly flesh out ideas. The lack of lightweight tools is compounded by the technical uncertainty inherent with new devices, where foundational interaction methods are still being established. These challenges can potentially limit the design contributions of your team to only those with 3D development backgrounds—drastically reducing the team's ability to iteration.



Teams from the Mixed Reality Partner Program in our workshop

When we work with external partners, we hear stories of teams ‘waiting for the developer to finish the prototype’ before they can continue with their design process. This often means the rest of the team is blocked in making meaningful progress on the product, overloading the developer to solve both the technical implementation as well as major components of the user experience (as they attempt to put a rough idea into code).

Techniques for expanding the design process

Our teams at Microsoft have a set of techniques to more effectively include their team and quickly iterate through complex design problems. While not a formal process, these techniques are intended to supplement rather than replace your workflow. These methods allow those without specialized 3D skills to offer ideas before diving into the prototyping phase. Different roles and disciplines (not just 3D designers and developers) can be part of the design process, uncovering opportunities and identifying possible challenges that might occur later in development.

Generating ideas with bodystorming

Getting your team to think about events occurring in the real world, beyond the traditional world of 2D devices, is key to developing innovative mixed reality experiences. At Microsoft, we have found the best way to do this is to encourage interaction with physical props in a real-world space. Using simple, cheap crafting materials we build physical props to represent digital objects, user interfaces, and animations in a proposed experience. This technique is called bodystorming and has been a staple of product ideation within industrial design for decades.



Simple, cheap art supplies used in bodystorming

Simple, physical props level the playing field for participants, allowing individuals with different skill sets and backgrounds to contribute ideas and uncover opportunities inherent to mixed reality experiences instead of being locked into the paradigm of 2D thinking. While technical prototyping or high-fidelity storyboarding requires a skilled 3D developer or artist, a few Styrofoam balls and cardboard can be enough to showcase how an interface might unfold in physical space. These techniques apply to both mixed reality development with HoloLens and the immersive headsets. For example, a set of plastic connectors might roughly illustrate the size of holograms that appear in a HoloLens experience or as props to act out interactable elements or motion designs in a virtual world.

Bodystorming is a technique to quickly generate ideas and evaluate ideas that are too nebulous to prototype. At Microsoft, bodystorming is most commonly used to quickly vet an idea, although it can be helpful to host a more in-depth session if you involve outside stakeholders who are not familiar with mixed reality development or need to distill very broad scenarios. Remember that the purpose of bodystorming is to ideate quickly and efficiently by encouraging participants to think spatially. Detailed artwork or precise measurements are not important at this stage. Physical props need only meet the minimum requirements to explore or communicate an idea. Ideas presented through bodystorming are not expected to be fully vetted, but the process can help narrow down possibilities to test later during the in-device prototyping phase. As such, bodystorming is not intended to replace technical prototyping, but rather offset the burden of solving both technical and design challenges during the prototyping phase.

Acting and expert feedback

Following the bodystorming process of ideating with physical objects in the real world, the next step is to walk through an experience with these objects. We call this phase of the process acting and it often involves staging how a user would move through the experience or a specific interaction.



Teams acting out a scenario during a workshop

Acting with physical props allows participants to not only experience the thinking through the perspective of the user but allow outside observers to see how events play out. This presents an ideal time to include a wider audience of team members or stakeholders who are able to provide specific 'expert' feedback. For example, if you are exploring a mixed reality experience designed for hospitals, acting out your thinking to a medical professional can provide invaluable feedback. Another example is when you might have a specific challenge you are trying to understand, like spatial audio or budgeting for asset quality vs. performance. In these cases, acting gives experts a quick, rough concept of how the experience might unfold, without the need for a device-based prototype.

This sort of acting is not formalized, with no need to bring in professional talent, but in situations where we want to share the thinking with others who are not present we will video record a 'scene' of interaction or include a storyboard artist to observe and sketch key moments. Acting can also be a very lightweight activity, often happening in-situ during the bodystorming phase. Deciding which to use depends on the audience (and fidelity needed to elicit the necessary type of feedback) but ultimately comes down to whatever will most effectively capture your team's thinking.

Capturing ideas with storyboards

The best method for conveying the ideas and concepts of your proposed experience depends on your intended audience, as well as the type of feedback your next iteration requires. When presenting new ideas to team members, low fidelity re-enactments of bodystorming can be enough to bring someone up to speed. When introducing your experience concepts to new stakeholders or potential users, the best method is often storyboarding. Storyboarding is a technique common to the entertainment industry, usually found behind the scenes in movies and video game development, and helps convey both the overall flow of an experience (at low fidelities) as well as the aesthetic look and feel (at high fidelities). Just as with prototyping, understanding the fidelity needs of your storyboard is key to gathering the right kind of feedback and avoiding counter-productive discussions.



Example of a low-fidelity storyboard

Low-fidelity storyboards are the right fidelity for quick discussions, especially when conveying high-level ideas. These can be as simple as stick-figure drawings and primitive shapes to denote virtual elements in a scene or the proximity of interactive components (both physical and virtual). While these are useful given the relative ease and low skill barrier to execute, remember the lesson from bodystorming: Not everyone can see 2D depictions of an experience and understand the 3D implications.



Example of a high-fidelity storyboard

High-fidelity storyboards are a powerful tool when bringing in new stakeholders or when combining insights from a bodystorming session with the proposed aesthetic direction of your experience. Storyboards can build off mood boards to illustrate the final appearance of a virtual experience, as well as capture key moments that may be pivotal to the final product. Keep in mind that high-fidelity storyboards often require an artist, especially one embedded within the team, who can capture difficult-to-describe ideas. At Microsoft we have added storyboard artists, often with backgrounds in game development, to our mixed reality teams who attend meetings prepared to quickly capture sketches that will later be expanded into higher fidelity mockups. These individuals work closely with technical artists, helping to convey the artistic direction of assets used in the final experience.

Expanding your team's skills

In addition to the techniques described, altering your process to better accommodate mixed reality experiences depends on more closely connecting technical and design roles. Working with partners at Microsoft, we have witnessed several teams successfully transition into 3D development, and find the biggest advantage comes from team members stepping out of their comfort zone and developing skills that give them more involvement throughout the development process (not just limiting their skills to design or development).

Feedback and iteration is key to any successful design, and with mixed reality experiences this sort of feedback often skews to more technical concepts, especially given the relative nascence of mixed reality's technology and tools. Building your design team's skills in technical areas, including a proficiency with tools like Unity or Unreal, helps those individuals better understand the approach of developers and provide more effective feedback during prototyping. Similarly, developers who understand the fundamental UX concepts of mixed reality experiences (and potential design pitfalls) can help them offer implementation insights during the early phases of planning.

We usually get questions from partners working in mixed reality about how best to grow their team and skill sets

with 3D tools. This is a difficult question to answer given the state of the industry, as teams are faced with building tomorrow's experiences with yesterday's tools. Many teams are gathering individuals with backgrounds in gaming and entertainment, where 3D development has been a staple for decades, or encouraging their existing teams to pick up tools like Unity and 3D modeling software. While these are both essential needs for meeting the baseline of mixed reality experiences today, there will very likely be a great number of specialized tools and skills to help build tomorrow's ground-breaking applications.

The field of mixed reality design will change dramatically in the near future, with more focus on data and cloud services in holographic experiences. This next generation of experiences will take advantage of advanced natural language processing and real-time computer vision, as well as more inherently social scenarios within immersive experiences. For example, experiences in virtual reality allows designers to build imaginative environments and spaces, requiring a skill set more akin to architecture or psychology rather than traditional interface design. Or consider experiences in the real world with HoloLens, where the use cases often involve highly specialized fields such as medicine, engineering, or manufacturing, where specific environments and real-time data are essential elements of the experience. Working across roles, leveraging both design and specialized knowledge, and having a willingness to learn new tools are invaluable skill for teams working in mixed reality.

Helping your team explore mixed reality quickly and effectively

In the early days of HoloLens, these techniques were borne out of necessity as the device's early prototype hardware proved to be a less than ideal format for quickly iterating through design ideas. Interaction methods, including fundamental UI in the operating system were explored with bodystorming, while acting helped our usability team understand fundamental user behaviors for scenarios in the workplace. In the end, the team was able to establish a strong baseline for the core UX of HoloLens, communicating concepts across team roles, allowing OS development to move quickly in parallel with hardware development.



Whether it is bodystorming, acting, reviewing with experts, or storyboarding, these techniques are intended to save you time and effort. While we are still very much in the early days of HoloLens and virtual reality development, expanding your design process for mixed reality will help your team spend their energy exploring new and challenging design problems rather than overcoming difficulties communicating ideas.

Sample list of workshop supplies

Simple, cheap art supplies are key to providing team members with the tools necessary to explain ideas without

requiring advanced artistic skills. Here is a sample of what our team commonly uses during bodystorming:

- Styrofoam discs
- Styrofoam cubes
- Styrofoam cones
- Styrofoam spheres
- Cardboard boxes
- Wooden dowels
- Lollipop sticks
- Cardstock
- Paper cups
- Duct tape
- Masking tape
- Scissors
- Paperclips
- Filing clamps
- Post-Its
- Twine
- Pencils
- Sharpies

See also

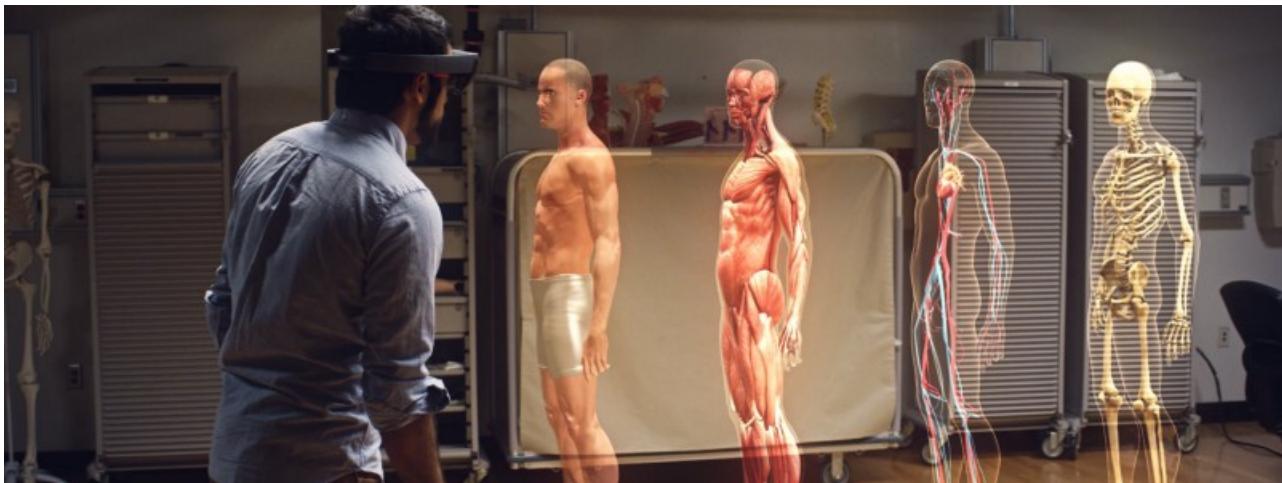
- [Case study - My first year on the HoloLens design team](#)
- [Case study - AfterNow's process - envisioning, prototyping, building](#)

Case study - The pursuit of more personal computing

11/6/2018 • 12 minutes to read • [Edit Online](#)

Tomorrow's opportunities are uncovered by building products today. The solutions these products provide reveal what's necessary to advance the future. With mixed reality this is especially true: Meaningful insight comes from getting hands-on with real work — real devices, real customers, real problems.

At Microsoft, I'm part of the design team helping enterprise partners build experiences for their business using Windows Mixed Reality. Over the past year, our team has focused on HoloLens and understanding how Microsoft's flagship holographic device can deliver value to customers today. Working closely with designers and developers from these companies, our team focuses on uncovering solutions that would be technically unfeasible, financially impractical, or otherwise impossible without HoloLens.



HoloAnatomy from Case Western Reserve University

Building these solutions helps Microsoft's internal teams prepare for the next generation of computing. Learning how individuals and businesses interact with core technologies like mixed reality, voice, and AI, helps Microsoft build better devices, platforms, and tools for developers. If you are a designer or a developer exploring this space, understanding what our teams are learning from partners today is critical to preparing for tomorrow's mixed reality opportunities.

Microsoft's ambition for mixed reality

We live our lives among two worlds: the physical and the digital. Both have fundamental strengths that we leverage to augment and extend our abilities. We can talk in-person with a friend, using our physical senses to understand things like body language, or we can augment our ability to talk by video-chatting with a friend from miles away. Until now these two worlds, and their respective strengths, have been fundamentally separated.

The physical world is one of atoms and physics. We use our senses to make decisions, leveraging years of learned behavior to interact with objects and people in our environments. Despite the ease of these interactions, we are limited by our physical abilities and the laws of nature.

The digital world is one of bits and logic. Computations are made instantly while information can be distributed effortlessly. Despite the speed and flow of information, interactions are too often limited to small screens, abstract inputs, and noisy feeds.



HoloStudio

What if we could combine the advantages of the physical and digital worlds? This is the cornerstone of experiences across the spectrum of mixed reality: a medium where the physical and digital co-exist and seamlessly interact. Combining these worlds builds a new foundation for interacting more naturally with technology — an evolution in personal computing.

The spectrum of mixed reality has exploded as developers have begun exploring the opportunities of immersion and presence. Bringing users into the digital world with immersive (virtual reality) experiences and creating digital objects in the physical world with holographic (augmented reality) experiences. But what are the benefits of mapping the physical world to the digital world? What happens when we give computers eyes to see?

The fundamental camera vision technology behind holograms acts like a pair of eyes for the computer to see the environment around you: Objects in the world, people around you, changes as they happen. A digital understanding of your context in the physical world. This leads to an enormous amount of information, the implications of which we are only beginning to understand.

Culminating core technologies

Computing is too often a tangible thing. Grabbing our devices to tell them who we are and what we want. Contorting our thinking and aligning what we say to match what we believe the computer needs to hear.

The promise of mixed reality, especially in the real world with holographic experiences, is to lessen the burden of interacting with technology. Reducing cognitive load as users navigate the layers of abstraction inherent to computing today. How can we design experiences that not only take advantage of contextual understanding, but make it easier to draw insight and take action? Two contributing technologies are also attempting to solve this problem:

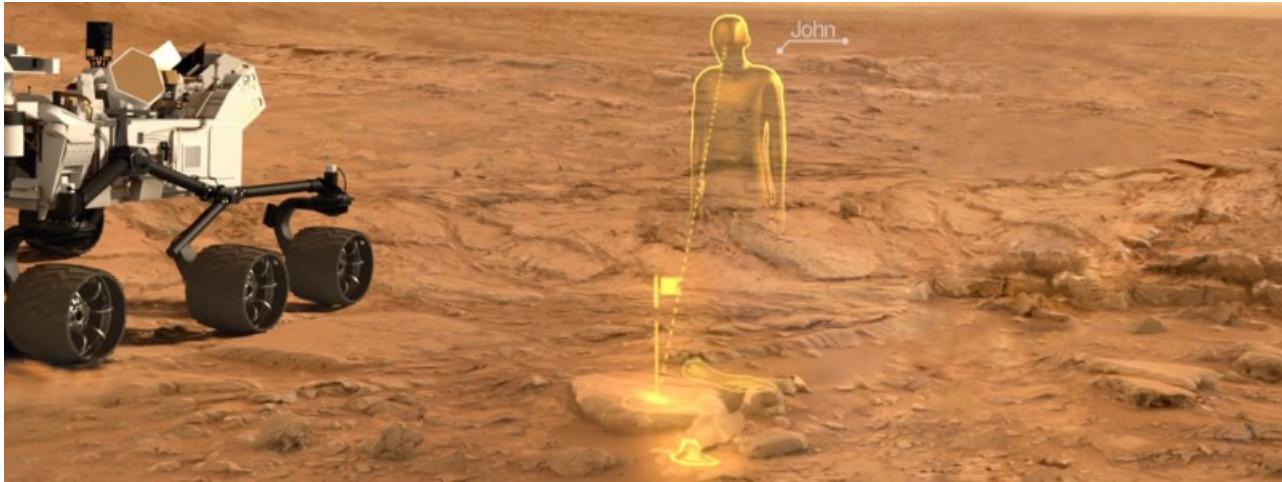
- **Voice**, in terms of speech and conversation, is enabling users to communicate with computers through more natural means — answering bots through text or issuing commands to conversational hardware.
- **AI** is powering experiences that distill insights from increasingly complex datasets. While AI is an enormous topic, recent progress has provided the groundwork for devices that rely on computer vision, more natural digital assistants, and recommending actions to users.

Mixed reality provides a medium to combine these technologies into a single user experience. Voice becomes a powerful, natural method for input when wearing a holographic headset. AI acts as a critical cipher to contextualize the enormous amounts of information connecting the physical and digital worlds. This is why Satya Nadella refers to HoloLens as ‘the ultimate computer’, it’s a culminating device for three core technologies. A platform to empower humans to more easily interact with the growing complexity of devices and services.

Less interface in your face

A culminating device that connects the physical world to the digital world allows us to design experiences that fit more naturally, without cumbersome abstractions. Consider the experiences you’ve created: When the barriers of abstraction are removed, how much of your interface is left? Which parts of your app flow change when you know the user and their context? How many menus and buttons remain?

For example, think about shared experiences in mixed reality like the OnSight tool NASA's Jet Propulsion Laboratory built for scientists. Instead of building a system to look at Martian data (abstracting the data onto screens or displays), they brought scientists inside the data, effectively putting them on the surface of Mars as they collaborated.



NASA/JPL OnSight

Instead of finding the button to draw attention to some Martian geology, scientists can point to it directly. No menus, no pens, no learning curve to using the tool effectively. By leveraging our known abilities from the physical world, more natural interactions in mixed reality can circumvent deep technical literacy in even the most advanced industry tools.

Likewise, voice and AI can extend natural interaction in experience like this. When digital assistants can 'see' into the world around us, conversations will feel less cumbersome. A bot in NASA's tool might extract context from the scientists conferring over Martian geology, acting as a ready (and conversational) source of information when scientists gesture and ask about 'this' or 'that'. A computer that knows your context is ready to jump in with information appropriate to you, through a method of interaction most appropriate to your context.

Building on a foundation

In this world of contextual experiences, catering to the mobility of the user is key. Moving fluidly between devices and services, with highly personalized contexts, users will travel about the physical world seamlessly — creating a massive platform challenge. When the physical world becomes a digital end-point, interoperability reigns supreme.

"Bringing together the digital universe and the physical world will unlock human potential... enabling every person and organization on the planet to achieve more."

— Satya Nadella

Windows Mixed Reality is an effort to create a platform for an ecosystem of devices, enabling developers to create immersive, affordable, and compatible experiences for the largest possible audience. The future will not be limited to a single manufacturer, let alone a single device. Headsets, mobile devices, PCs, accessories... all these physical things must interoperate (as well as digital things like graphs and services) through an underlying platform to successfully deliver on the promise of mixed reality.

Designing for tomorrow's experiences today



Each one of the core technologies behind this new class of experiences are enabling designers and developers to create compelling and successful experiences today. By reducing abstraction, we can interact more directly with the digital world, allowing us to design in ways that augment and amplify human abilities. Voice technology (through bots and digital assistants like Cortana) is allowing users to carry out increasingly complex conversations and scenarios, while AI technology (through tools like Microsoft Cognitive Services) is causing companies to rethink how users will interact with everything from social media to supply chain management.

These types of interactions will rely on both a new class of design tools as well fundamental support from the platform. Creating these tools and building this platform for devices and services relies on understanding how tomorrow's experiences will solve real, tangible problems. We've identified five areas of opportunity where our enterprise partners have delivered valuable solutions and where we believe continued investment will help us prepare for this new class of computing.

Areas of opportunity

The past year of developer partnerships has uncovered areas of opportunity that resonate with customers and create successful enterprise solutions. From scientists and technicians to designers and clients, five areas of opportunity have emerged where Microsoft partners are find value with mixed reality. These areas are already providing massive insight into the future needs of platforms like Windows Mixed Reality and can help you understand how these new experiences will impact the ways we learn, collaborate, communicate, and create.

1. Creation and design

One of the chief opportunities of mixed reality is the ability to see and manipulate 3D designs in real time, in a real-world setting, at true size and scale. Design and prototyping tools have escaped the confines of screens, returning to a realm of design usually reserved for tangible, physical materials like wood and clay.

[Autodesk](#) created a mixed reality experience aimed at improving collaboration across the entire product development process. The ability for engineers and designers to survey and critique 3D models in their environment allowed them to iterate on a design in real-time. This not only enabled faster prototyping but gave way to more confident decisions in the process.

Experiences like this highlight the need for core collaboration experiences, the ability for users to see and communicate with shared objects. Autodesk's goal is to expand the product from design professional and engineers to digital artists, students, and hobbyists. As the level of 3D expertise of users decreases, the ability to interact with objects naturally becomes crucial.

2. Assembly and manufacturing

From the increasing amount of specialization on factory floors to the rapid advancements of supply chain management, seamless access to relevant information is key. Mixed reality offers the ability to synthesize extensive data sets and provide visual displays that aid in areas like navigation and operations. These are often highly

technical, niche fields where integration with custom datasets and services are crucial to reducing complexity and providing a successful experience.

Elevator manufacturer [ThyssenKrupp](#) created an experience for elevator service technicians, allowing them to visualize and identify problems in preparation for a job. With a team spanning over 24,000 technicians, devices like HoloLens allow these technicians to have remote, hands-free access to technical and expert information.

ThyssenKrupp highlights a powerful concept here that critical, contextually-relevant information can be delivered quickly to users. As we look ahead to a new class of experiences, distilling vast amounts of possible information to content that is relevant to the user will be key.

3. Training and development

Representing objects and information in three dimensions offer new ways to explain scenarios visually and with spatial understanding. Training becomes a key area of opportunity, leveraging the ability to digitally represent enormous, complex objects (like jet engines) to create training simulations for a fraction of the cost of a physical solution.

[Japan Airlines](#) has been experimenting with concept programs to provide supplemental training for engine mechanics and flight crews. The massive jet engines (in both size and complexity) can be represented in 3D, ignoring the limitations of the physical world to move and rotate virtual objects around trainees, allowing them to see how airline components work in real-time.

Training with virtual components (and reducing the need for expensive, physical training simulators) is a key way to deliver value in enterprise scenarios today. As this scenario expands (as we've seen in areas like medicine), computer vision becomes especially important to recognize unique objects in the environment, understand the context of the user, and deliver relevant instructions.

4. Communication and understanding

Interactions between two people (whether both participants are in mixed reality devices or one is on a traditional PC or phone) can provide both a sense of immersion within a new environment or a sense of presence when communicating virtually with others. In enterprise scenarios this sense of presence is a boon to mobile teams, helping to increase understanding of projects and lessen the need for travel.

Commercial tech manufacturer [Trimble](#) developed a solution for architecture and construction industry professionals to collaborate and review work during building development. Professionals can remotely immerse themselves into a project to discuss progress or be on location and review plans as they would look (in their final form) in the environment around them.

Shared experiences are a major area of investment for Microsoft, with apps like Skype exploring new ways to represent humans in digital space. Teams are exploring volumetric video recordings, avatars, and recreations of a participant's physical space.

5. Entertainment and engagement

The nature of immersion in mixed reality can have a tremendous impact on the way customers engage with entertainment and marketing. Artists and entertainment studios have explored mixed reality as a compelling medium for storytelling, while companies are exploring the implications for brand marketing. From product demos in private showrooms to vision pieces told at trade expos, content can be delivered in more immersive and tailored ways.

Volvo created an experience for showcasing their latest car models (immersing users in different colors and configurations) while highlighting how advanced sensors work to create a better, safer driving experience. Taking customers through a guided showroom experience allows Volvo to tell the story behind cutting-edge car feature while delivering a memorable portrayal of their brand story.

Entertainment is in many ways pushing the bounds of mixed reality (especially virtual reality) and perhaps most compelling in this space is how it will interact combine with the previous area of opportunity: communication and understanding. With multiple users, each with their own variant of devices and interface methods, you can imagine a vast future of personalization in the realm of entertainment.

Start building today

It's hard to say what the far future of mixed reality will look like for consumers, but focusing on unique problems, getting hands-on with real hardware, and **experimenting today with the intersection between mixed reality, voice, and AI is key**. Microsoft is just getting started with mixed reality but learning from the successes realized by businesses today will help you create the experiences of tomorrow.

About the author



Mark Vitazko

UX Designer @Microsoft

Case study - AfterNow's process - envisioning, prototyping, building

11/6/2018 • 6 minutes to read • [Edit Online](#)

At [AfterNow](#), we work with you to turn your ideas and aspirations into concrete, fully operational products and experiences ready for the market. But before we write a single line of code, we create a blueprint through a process called envisioning.



What is Envisioning?

Envisioning is an ideation process used for product design. We streamline a large pool of ideas into one -- or a few -- calculated, realistic concepts to execute.

This process will bring to light many problems and blockers that would have eventually come up in the product. Failing early and fast ultimately saves a lot of time and money.

Throughout this process, we'll provide you with notes on the group's ideas and findings. When we're done, you'll receive a document of a script and a video of our "act it out" step, a simulation of the proposed product in action. These are concrete blueprints to your product -- a framework you can keep. Ideally, the next step is for us to build the product, which lets us test and experience the real deal.

Here's a daily play-by-play of the process, how long each step takes, and who you'll need on your team.



Envisioning participants

It's important to select the right participants for this process. The number can vary, but we recommend keeping it under seven people for the sake of efficiency.

Decider (critical)- The Decider is an important person to have in the group. He or she is the one who will be making the final decision or tipping the scale if there is an indecision during the process (typically the CEO, founder, product manager, or head of design). If they aren't able to participate for the entire process, appoint someone most fit to take on the role.

Finance (optional)- Someone who can explain how the project is funded.

Marketing (optional)- Someone who crafts the company's messages.

Customer (important)- Someone who talks to customers frequently and can be their advocate.

Tech/logistics (important)- Someone who understands what the company can build and deliver.

Design (important)- Someone who designs the product your company makes.

DAY 1 - Map

List goals & constraints; set a long-term goal - 5 min

If we were being completely optimistic, what is our long-term goal? Why are we building this solution in mixed reality and what problems are we trying to solve?

List sprint questions - 5 min

Get a bit pragmatic and list out fears in question format. How could we fail? To meet the long-term goal, what has to be true? If the project failed, what may have caused that?

Make a flow map - Up to 1 hour

This is a rough, functional map (not detailed). We write all of the user types on the left and the endings on the right. Then we connect them in the middle with words and arrows showing how users interact with the product. This is done in 5-15 steps.

Interview experts & take notes

The goal of this exercise is to enhance our collective understanding of the problem. Here we can add things to the map and fix any mistakes on it. If part of the flow can be expanded in more detail, we talk to the expert.

Our best practice for this process happens when we take individual notes in How might we (HMW) format. HMW is a technique developed by P&G in the 70s. It's a method of note taking in the form of a question which ultimately results in organized and prioritized notes. We will jot down one idea per sticky note which starts with the phrase, "How might we." For example, how might we communicate the values of [your company] in mixed reality?



Organize and decide

Organize HMW notes

We take all the "How might we" notes and stick them onto a wall. We will then categorize them into noticeable themes as they emerge.

Vote on HMW notes

Once all the notes are in their appropriate categories, we will vote on our favorite questions. Everyone gets two votes; the Decider gets four votes.

Pick a target

Circle the most important customer and one target moment on the map and look over the initial questions for the sprint. The Decider has the task of making the call.

DAY 2 - Sketch

Existing product demos

Everyone in the sprint group will research existing products they like that solve similar problems to the ones we're tackling. We'll then demo our findings to the rest of the group. This gives us a good idea of the features we find interesting.

Notes

We spend 20 minutes silently walking around the room to gather notes.

Ideas

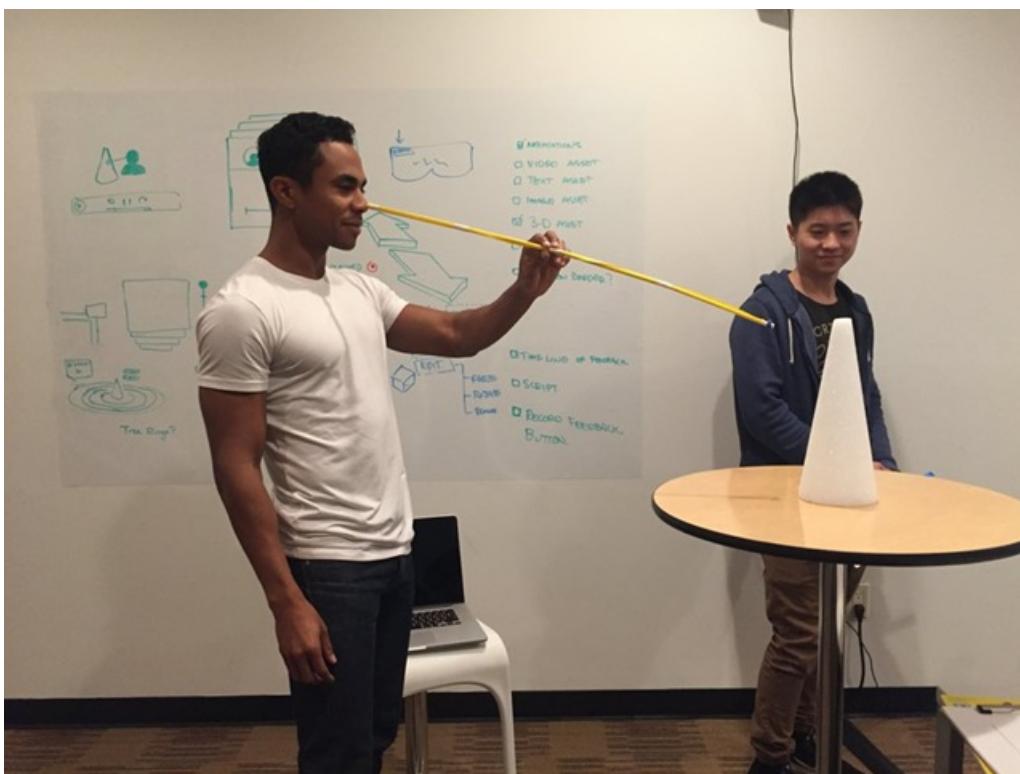
We spend 20 minutes privately jotting down some rough ideas.

Crazy 8s

We spend eight minutes creating eight frames. This is where we rapidly sketch a variation of one of our best ideas in each frame.

Solution

We spend 30 minutes sketching storyboards. We take the best variations of ideas from Crazy 8s and create a three-panel storyboard of the idea.



Sketching storyboards

DAY 3 - Decide

This day involves a lot of critiquing and voting. Here's the breakdown:

Quiet critiquing

We tape solutions onto the wall and take time to look at them silently. After everyone has gotten time to look at all the sketches, we will collectively create a heat map. Each person puts marks on the sketches they like. This can be the entire flow or an area of a flow.

Discussion & explanations

We will begin to see a heat map of ideas people find interesting, and spend three minutes on each sketch that got the most attention. People can openly discuss why they like certain sketches and the artist can correctly convey their ideas when there is a misunderstanding. This can surface new ideas at times.

In this step we also want to give people opportunities to explain ideas that didn't get a lot of attention. Sometimes you'll find a gem in an idea people didn't really understand previously.

Straw poll & supervote

Each person will silently choose their favorite idea and, when ready, people will place a mark on their favorite one

all at once. The Decider will then get three large dots for the idea he/she liked the most. The ones marked by the Decider are the ideas that are going to be prototyped.

DAY 4 - Prototype

Preparation

In preparation for the prototyping phase, we will gather an assortment of physical tools -- much like a child's play kit -- to construct 3D objects that will represent holograms. We will also have physical tools that represent the hardware devices people are using to access these holograms.

Prototype!

Prepare a script

Script out the entire user flow for this prototype and decide on the actors that will be playing a part in the skit.

Act it out

Physically act out the user flow which involves all the user parties in the order they will be using it. This gives us a good idea of how viable this product will be when we build it out. It will also bring to light some major kinks we may have overlooked when coming up with the idea.

About the author



Rafai Eddy
AfterNow

Interaction fundamentals

11/6/2018 • 4 minutes to read • [Edit Online](#)

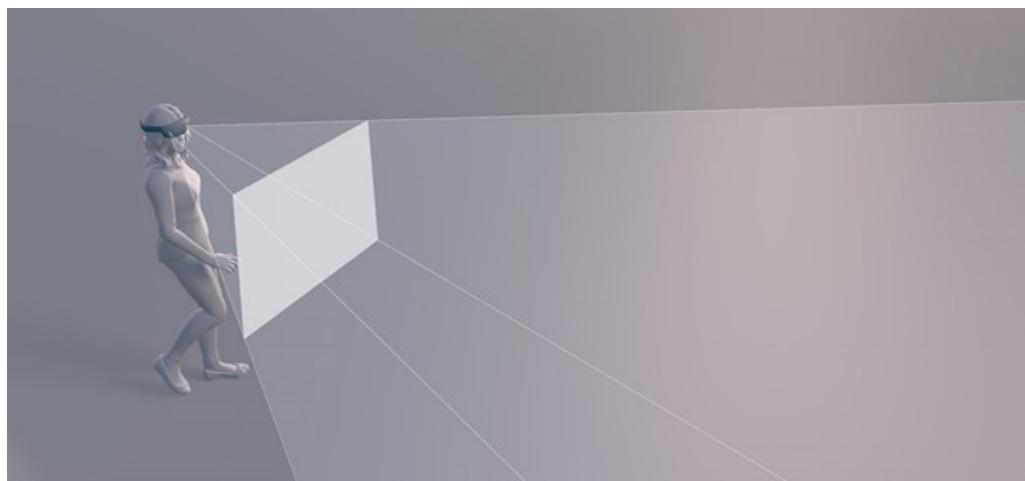
As we've built experiences across HoloLens and immersive headsets, we've started writing down some things we found useful to share. Think of these as the fundamental building blocks for mixed reality interaction design.

Device support

Here's an outline of the available Interaction design articles and which device type or types they apply to.

INPUT	HOLOLENS	IMMERSIVE HEADSETS
Gaze targeting	✓□	✓□
Gestures	✓□	
Voice design	✓□	✓□
Gamepad	✓□	✓□
Motion controllers		✓□
PERCEPTION AND SPATIAL FEATURES		
Spatial sound design	✓□	✓□
Spatial mapping design	✓□	
Holograms	✓□	

The user is the camera



Always think about design for your user's point of view as they move about their real and virtual worlds.

Some questions to ask

- Is the user sitting, reclining, standing, or walking while using your experience?

- How does your content adjust to different positions?
- Can the user adjust it?
- Will the user be comfortable using your app?

Best practices

- The user is the camera and they control the movement. Let them drive.
- If you need to virtually transport the user, be sensitive to issues around vestibular discomfort.
- Use shorter animations
- Animate from down/left/right or fade in instead of Z
- Slow down timing
- Allow user to see the world in the background

What to avoid

- Don't shake the camera or purposely lock it to 3DOF (only orientation, no translation), it can make users feel uncomfortable.
- No abrupt movement. If you need to bring content to or from the user, move it slowly and smoothly toward them for maximum comfort. Users will react to large menus coming at them.
- Don't accelerate or turn the user's camera. Users are sensitive to acceleration (both angular and translational).

Leverage the user's perspective

Users see the world of mixed reality through displays on immersive and holographic devices. On the HoloLens, this display is called the [holographic frame](#).

In 2D development, frequently accessed content and settings may be placed in the corners of a screen to make them easily accessible. However, in holographic apps, content in the corners of the user's view may be uncomfortable to access. In this case, the center of the holographic frame is the prime location for content.

The user may need to be guided to help locate important events or objects beyond their immediate view. You can use arrows, light trails, character head movement, thought bubbles, pointers, spatial sound, and voice prompts to help guide the user to important content in your app.

It is recommended to not lock content to the screen for the user's comfort. If you need to keep content in view, place it in the world and make the content "tag-along" like the Start menu. Content that gets pulled along with the user's perspective will feel more natural in the environment.



The Start menu follows the user's view when it reaches the edge of the frame

On HoloLens, holograms feel real when they fit within the holographic frame since they don't get cut off. Users will move in order to see the bounds of a hologram within the frame. On HoloLens, it's important to simplify your UI to fit within the user's view and keep your focus on the main action. For immersive headsets, it's important to maintain the illusion of a persistent virtual world within the device's field of view.

User comfort

To ensure maximum [comfort](#) on head-mounted displays, it's important for designers and developers to create and present content in a way that mimics how humans interpret 3D shapes and the relative position of objects in

the natural world. From a physical perspective, it is also important to design content that does not require fatiguing motions of the neck or arms.

Whether developing for HoloLens or immersive headsets, it is important to render visuals for both eyes. Rendering a heads-up display in one eye only can make an interface hard to understand, as well as causing uneasiness to the user's eye and brain.

Share your experience

Using [mixed reality capture](#), users can capture a photo or video of their experience at any time. Consider experiences in your app where you may want to encourage snapshots or videos.

Leverage basic UI elements of the Windows Mixed Reality home

Just like the Windows PC experience starts with the desktop, Windows Mixed Reality starts with the home. The [Windows Mixed Reality home](#) leverages our innate ability to understand and navigate 3D places. With HoloLens, your home is your physical space. With immersive headsets, your home is a virtual place.

Your home is also where you'll use the Start menu to open and place apps and content. You can fill your home with mixed reality content and multitask by using multiple apps at the same time. The things you place in your home stay there, even if you restart your device.

See also

- [Gaze targeting](#)
- [Gestures](#)
- [Voice design](#)
- [Motion controllers](#)
- [Spatial sound design](#)
- [Spatial mapping design](#)
- [Comfort](#)
- [Navigating the Windows Mixed Reality home](#)

Comfort

11/6/2018 • 8 minutes to read • [Edit Online](#)

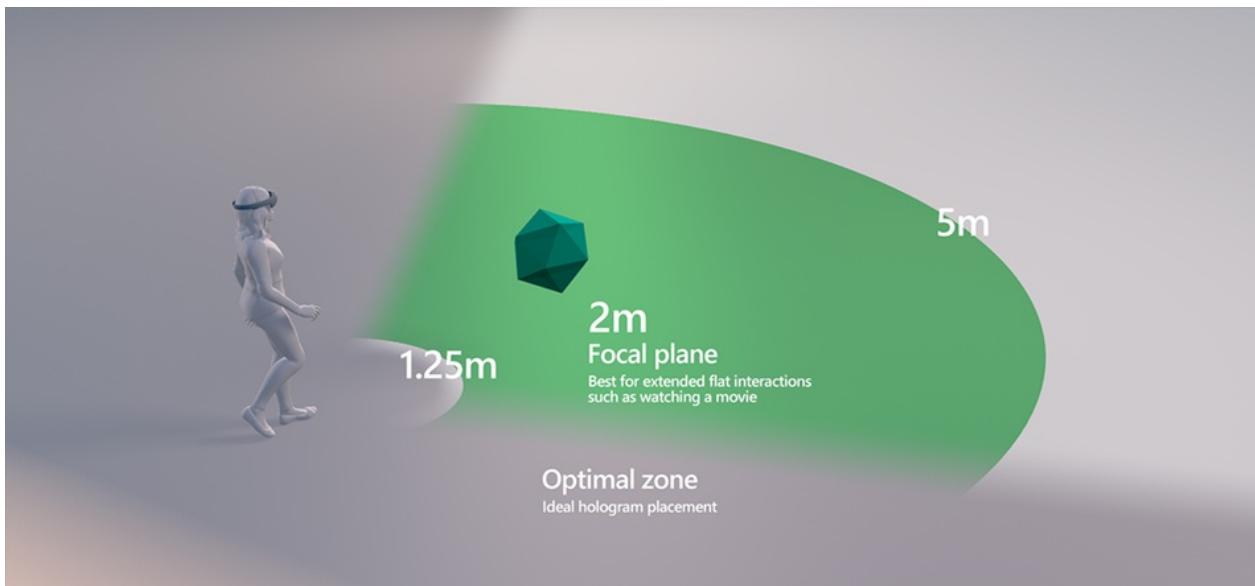
During natural viewing, the human visual system relies on multiple sources of information, or “cues,” to interpret 3D shapes and the relative position of objects. Some cues are monocular, including [linear perspective](#), familiar size, occlusion, depth-of-field blur, and [accommodation](#). Other cues are binocular, such as [vergence](#) (essentially the relative rotations of the eyes required to look at an object) and [binocular disparity](#) (the pattern of differences between the projections of the scene on the back of the two eyes). To ensure maximum comfort on head-mounted displays, it’s important for designers and developers to create and present content in a way that mimics how these cues operate in the natural world. From a physical perspective, it is also important to design content that does not require fatiguing motions of the neck or arms. In this article, we’ll go over key considerations to keep in mind to achieve these goals.

Vergence-accommodation conflict

To view objects clearly, humans must [accommodate](#), or adjust their eyes’ focus, to the distance of the object. At the same time, the two eyes must [converge](#) to the object’s distance to avoid seeing double images. In natural viewing, vergence and accommodation are linked. When you view something near (e.g. a housefly close to your nose) your eyes cross and accommodate to a near point. Conversely, if you view something at infinity, your eyes’ lines of sight become parallel and the your eyes accommodates to infinity. In most head-mounted displays users will always accommodate to the focal distance of the display (to get a sharp image), but converge to the distance of the object of interest (to get a single image). When users accommodate and converge to different distances, the natural link between the two cues must be broken and this can lead to visual discomfort or fatigue.

Guidance for holographic devices

HoloLens displays are fixed at an optical distance approximately 2.0m away from the user. Thus, users must always accommodate near 2.0m to maintain a clear image in the device. App developers can guide where users’ eyes converge by placing content and holograms at various depths. Discomfort from the vergence-accommodation conflict can be avoided or minimized by keeping content to which users converge to as close to 2.0m as possible (i.e. in a scene with lots of depth, place the areas of interest near 2.0m from the user when possible). When content cannot be placed near 2.0m, discomfort from the vergence-accommodation conflict is greatest when the user’s gaze switches back and forth between different distances. In other words, it is much more comfortable to look at a stationary hologram that stays 50cm away than to look at a hologram 50cm away that moves toward and away from you over time.



Optimal distance for placing holograms from the user

Best practices

For maximum comfort, **the optimal zone for hologram placement is between 1.25m and 5m**. In every case, designers should attempt to structure content scenes to encourage users to interact 1m or farther away from the content (e.g. adjust [content size and default placement parameters](#)). When possible, we recommend starting to fade out content at 40cm and placing a rendering clipping plane at 30cm to avoid any nearer objects.

In applications where content must be placed closer than 1m, extra care should be taken to ensure user comfort. For instance, as stated above, objects that move in depth are more likely than stationary objects to produce discomfort due to the vergence-accommodation conflict. Additionally, the odds of discomfort due to the conflict increase exponentially with decreasing viewing distance. We recommend creating a "depth budget" for apps based on the amount of time a user is expected to view content that is near (<1m) and/or moving in depth. An example is to avoid placing the user in those situations more than 25% of the time. If the depth budget is exceeded, we recommend careful user testing to ensure it remains a comfortable experience.

Guidance for immersive devices

For immersive devices, the guidance for HoloLens still applies, but the specific values for the Zone of Comfort are shifted depending on the focal distance to the display. In general, the focal distances to these displays are between 1.25m-2.5m. When in doubt, avoid rendering objects of interest too near to users and instead try to keep most content 1m or farther away.

Rendering rates

Mixed reality apps are unique because users can move freely in the world and interact with virtual content like as though they were real objects. To maintain this impression, it is critical to render holograms so they appear stable in the world and animate smoothly. Rendering at a [minimum of 60 frames per second \(FPS\)](#) helps achieve this goal. There are some Mixed Reality devices that support rendering at framerates higher than 60 FPS and for these devices it is strongly recommended to render at the higher framerates to provide an optimal user experience.

Diving deeper

To draw holograms to look like [they're stable in the real or virtual world](#), apps need to render images from the user's position. Since image rendering takes time, HoloLens and other Windows Mixed Reality devices predict where a user's head will be when the images are shown in the displays. This prediction algorithm is an approximation. Windows Mixed Reality algorithms and hardware adjust the rendered image to account for the discrepancy between the predicted head position and the actual head position. This process makes the image

seen by the user appear as if it were rendered from the correct location, and holograms feel stable. The updates work best for small changes in head position, and they can't completely account for some rendered image differences, like those caused by motion-parallax.

By rendering at a minimum framerate of 60 FPS, you are doing two things to help make stable holograms:

1. Reducing the appearance of judder, which is characterized by uneven motion and double images. Faster hologram motion and lower render rates are associated with more pronounced judder. Therefore, striving to always maintain 60 FPS (or your device's maximum render rate) will help avoid judder for moving holograms.
2. Minimizing the overall latency. In an engine with a game thread and a render thread running in lockstep, running at 30FPS can add 33.3ms of extra latency. By reducing latency, this decreases prediction error, and increases hologram stability.

Performance analysis

There are a variety of tools that can be used to benchmark your application frame rate such as:

- GPUView
- Visual Studio Graphics Debugger
- Profilers built into 3D engines such as Unity

Self-motion and user locomotion

The only limitation is the size of your physical space; if you want to allow users to move farther in the virtual environment than they can in their real rooms, then a form of purely virtual motion must be implemented. However, sustained virtual motion that does not match the user's real, physical motion can often induce motion sickness. This outcome is due to the *visual cues* for self-motion from the *virtual world* conflicting with the *vestibular cues* for self-motion coming from the *real world*.

Fortunately, there are tips for implementing user locomotion that can help avoid the issue:

- Always put the user in control of their movements; unexpected self-motion is particularly problematic
- Humans are very sensitive to the direction of gravity. Therefore, non-user-initiated vertical motions especially should be avoided.
- **For holographic devices** - One method to allow the user to move to another location in a large virtual environment is to give the impression they're moving a small object in the scene. This effect can be achieved as follows:
 1. Provide an interface where the user can select a spot in the virtual environment where they want to move.
 2. Upon selection, shrink the scene rendering down to a disk around the desired spot.
 3. While keeping the spot selected, allow the user to move it as though it were a small object. The user can then move the selection close to their feet.
 4. Upon deselection, resume rendering the entire scene.
- **For immersive devices** - The preceding holographic device approach does not work as well in an immersive device because it requires the app to render a large black void or another default environment while moving the "disk." This treatment disrupts one's sense of immersion. One trick for user locomotion in an immersive headset is the "blink" approach. This implementation provides the user with control over their motion and gives a brief impression of movement, but makes it so brief that the user is less likely to feel disoriented by the purely virtual self-motion:
 1. Provide an interface where the user can select a spot in the virtual environment where they want to move.
 2. Upon selection, begin a very rapid simulated (100 m/s) motion towards that location while quickly fading out the rendering.

3. Fade the rendering back in after finishing the translation.

Heads-up displays

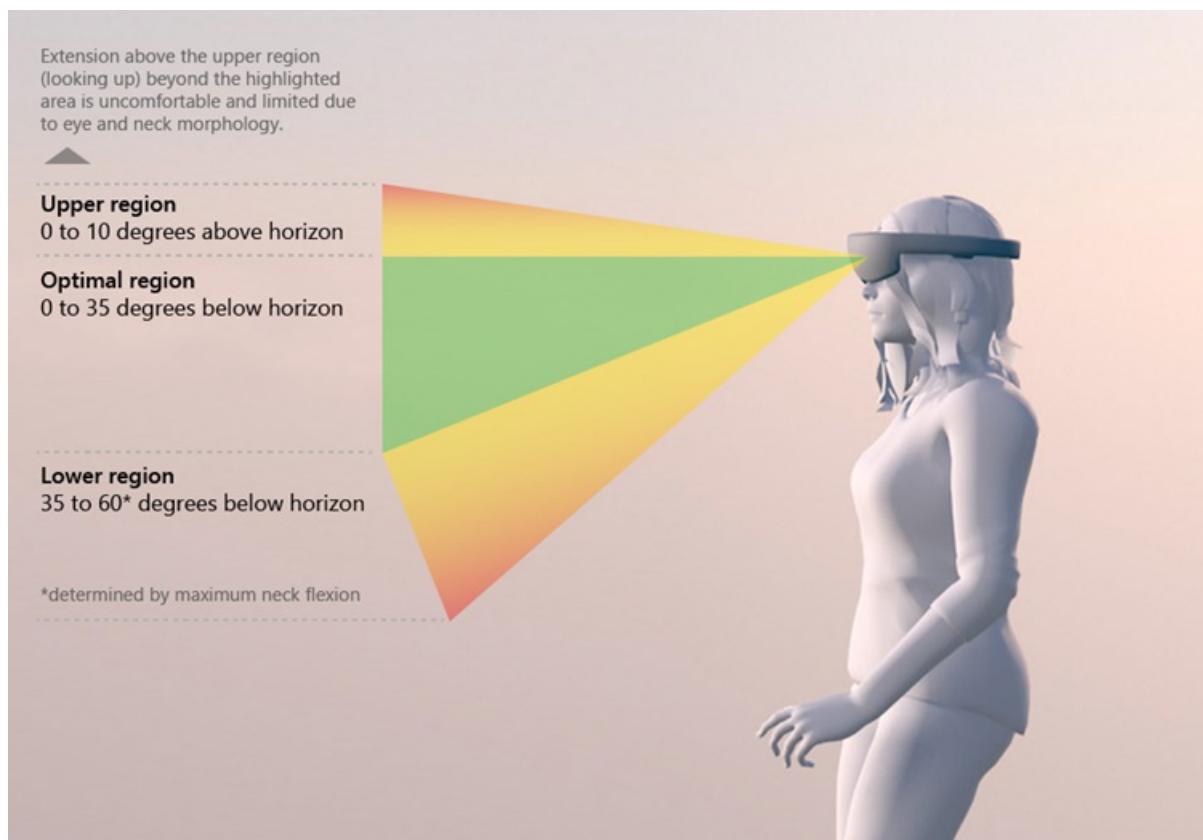
In first-person-shooter videogames, heads-up displays (HUDs) persistently present information such as player health, mini-maps, and inventories directly on the screen. HUDs work well to keep the player informed without intruding on the gameplay experience. In mixed reality experiences, HUDs have the potential to cause significant discomfort and must be adapted to the more immersive context. Specifically, HUDs that are rigidly locked to the user's head orientation are likely to produce discomfort. If an app requires a HUD, we recommend *body* locking rather than head locking. This treatment can be implemented as a set of displays that immediately translate with the user, but do not rotate with the user's head until a threshold of rotation is reached. Once that rotation is achieved, the HUD may reorient to present the information within the user's field of view. Implementing 1:1 HUD rotation and translation relative to the user's head motions should always be avoided.

Gaze direction

To avoid eye and neck strain content should be designed so that excessive eye and neck movements are avoided.

- **Avoid** gaze angles more than 10 degrees above the horizon (vertical movement)
- **Avoid** gaze angles more than 60 degrees below the horizon (vertical movement)
- **Avoid** neck rotations more than 45 degrees off-center (horizontal movement)

The optimal (resting) gaze angle is considered between 10-20 degrees below horizontal, as the head tends to tilt downward slightly, especially during activities.



Allowable field of view (FOV) as determined by neck range of motion

Arm positions

Muscle fatigue can accumulate when users are expected to keep a hand raised throughout the duration of an experience. It can also be fatiguing to require the user to repeatedly make air tap gestures over long durations. We therefore recommend that experiences avoid requiring constant, repeated gesture input. This goal can be achieved by incorporating short breaks or offering a mixture of gesture and speech input to interact with the app.

See also

- [Gaze](#)
- [Hologram stability](#)
- [Interaction fundamentals](#)

Gaze targeting

11/6/2018 • 3 minutes to read • [Edit Online](#)

All interactions are built upon the ability of a user to target the element they want to interact with, regardless of the input modality. In Windows Mixed Reality, this is generally done using the user's gaze.

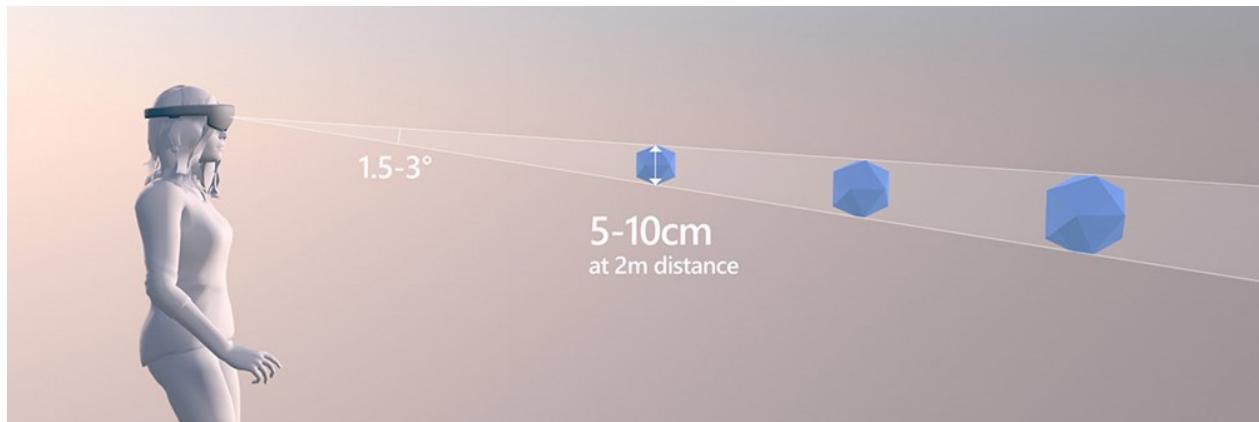
To enable a user to work with an experience successfully, the system's calculated understanding of a user's intent, and the user's actual intent, must align as closely as possible. To the degree that the system interprets the user's intended actions correctly, satisfaction increases and performance improves.

Device support

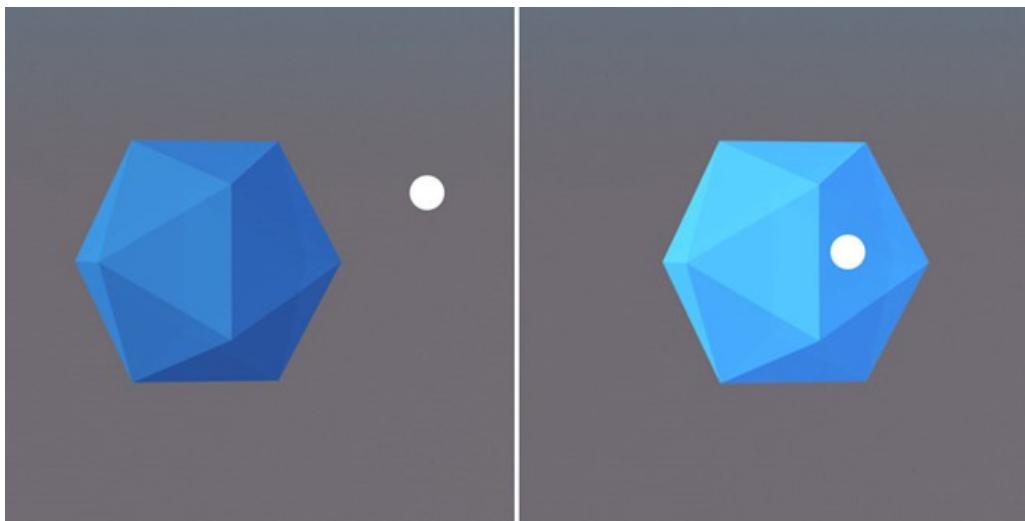
FEATURE	HOLOLENS	IMMERSIVE HEADSETS
Gaze targeting	✓ <input type="checkbox"/>	✓ <input type="checkbox"/>

Target sizing and feedback

The gaze vector has been shown repeatedly to be usable for fine targeting, but often works best for gross targeting (acquiring somewhat larger targets). Minimum target sizes of 1 to 1.5 degrees should allow successful user actions in most scenarios, though targets of 3 degrees often allow for greater speed. Note that the size that the user targets is effectively a 2D area even for 3D elements--whichever projection is facing them should be the targetable area. Providing some salient cue that an element is "active" (that the user is targeting it) is extremely helpful - this can include treatments like visible "hover" effects, audio highlights or clicks, or clear alignment of a cursor with an element.



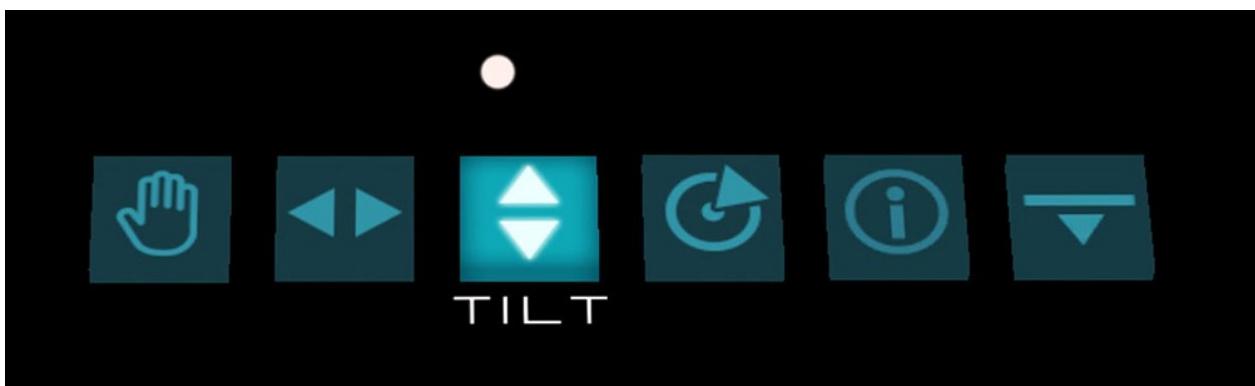
Optimal target size at 2 meter distance



An example of highlighting a gaze targeted object

Target placement

Users will often fail to find UI elements that are positioned very high or very low in their field of view, focusing most of their attention on areas around their main focus (usually roughly eye level). Placing most targets in some reasonable band around eye level can help. Given the tendency for users to focus on a relatively small visual area at any time (the attentional cone of vision is roughly 10 degrees), grouping UI elements together to the degree that they're related conceptually can leverage attention-chaining behaviors from item to item as a user moves their gaze through an area. When designing UI, keep in mind the potential large variation in field of view between HoloLens and immersive headsets.



An example of grouped UI elements for easier gaze targeting in Galaxy Explorer

Improving targeting behaviors

If user intent to target something can be determined (or approximated closely), it can be very helpful to accept "near miss" attempts at interaction as though they were targeted correctly. There are a handful of successful methods that can be incorporated in mixed reality experiences:

Gaze stabilization ("gravity wells")

This should be turned on most/all of the time. This technique removes the natural head/neck jitters that users may have. Also movement due to looking/speaking behaviors.

Closest link algorithms

These work best in areas with sparse interactive content. If there is a high probability that you can determine what a user was attempting to interact with, you can supplement their targeting abilities by simply assuming some level of intent.

Backdating/postdating actions

This mechanism is useful in tasks requiring speed. When a user is moving through a series of

targeting/activation maneuvers at speed, it can be useful to assume some intent and allow *missed steps* to act upon targets which the user had in focus slightly before or slightly after the tap (50ms before/after was effective in early testing).

Smoothing

This mechanism is useful for pathing movements, reducing the slight jitter/wobble due to natural head movement characteristics. When smoothing over pathing motions, smooth by size/distance of movements rather than over time

Magnetism

This mechanism can be thought of as a more general version of "Closest link" algorithms - drawing a cursor toward a target, or simply increasing hitboxes (whether visibly or not) as users approach likely targets, using some knowledge of the interactive layout to better approach user intent. This can be particularly powerful for small targets.

Focus stickiness

When determining which nearby interactive elements to give focus to, provide a bias to the element that is currently focused. This will help reduce erratic focus switching behaviours when floating at a midpoint between two elements with natural noise.

See also

- [Gestures](#)
- [Voice design](#)
- [Cursors](#)

Gestures

11/6/2018 • 7 minutes to read • [Edit Online](#)

Hand gestures allow users take action in mixed reality. Interaction is built on **gaze** to target and **gesture** or **voice** to act upon whatever element has been targeted. Hand gestures do not provide a precise location in space, but the simplicity of putting on a HoloLens and immediately interacting with content allows users to get to work without any other accessories.

Device support

FEATURE	HOLOLENS	IMMERSIVE HEADSETS
Gestures	✓ <input type="checkbox"/>	

Gaze-and-commit

To take actions, hand gestures use **Gaze** as the targeting mechanism. The combination of **Gaze** and the **Air tap** gesture results in a **gaze-and-commit** interaction. An alternative to gaze-and-commit is **point-and-commit**, enabled by **motion controllers**. Apps that run on HoloLens only need to support gaze-and-commit since HoloLens does not support motion controllers. Apps that run on both HoloLens and immersive headsets should support both gaze-driven and pointing-driven interactions, to give users choice in what input device they use.

The two core gestures of HoloLens

HoloLens currently recognizes two core component gestures - **Air tap** and **Bloom**. These two core interactions are the lowest level of spatial input data that a developer can access. They form the foundation for a variety of possible user actions.

Air tap

Air tap is a tapping gesture with the hand held upright, similar to a mouse click or select. This is used in most HoloLens experiences for the equivalent of a "click" on a UI element after targeting it with **Gaze**. It is a universal action that you learn once and then apply across all your apps. Other ways to perform select are by pressing the single button on a **HoloLens Clicker** or by speaking the voice command "Select".



Ready state for Air tap on HoloLens.

Air tap is a discrete gesture. A selection is either completed or it is not, an action is or is not taken within an experience. It is possible, though not recommended without some specific purpose, to create other discrete gestures from combinations of the main components (e.g. a double-tap gesture to mean something different than a single-tap).



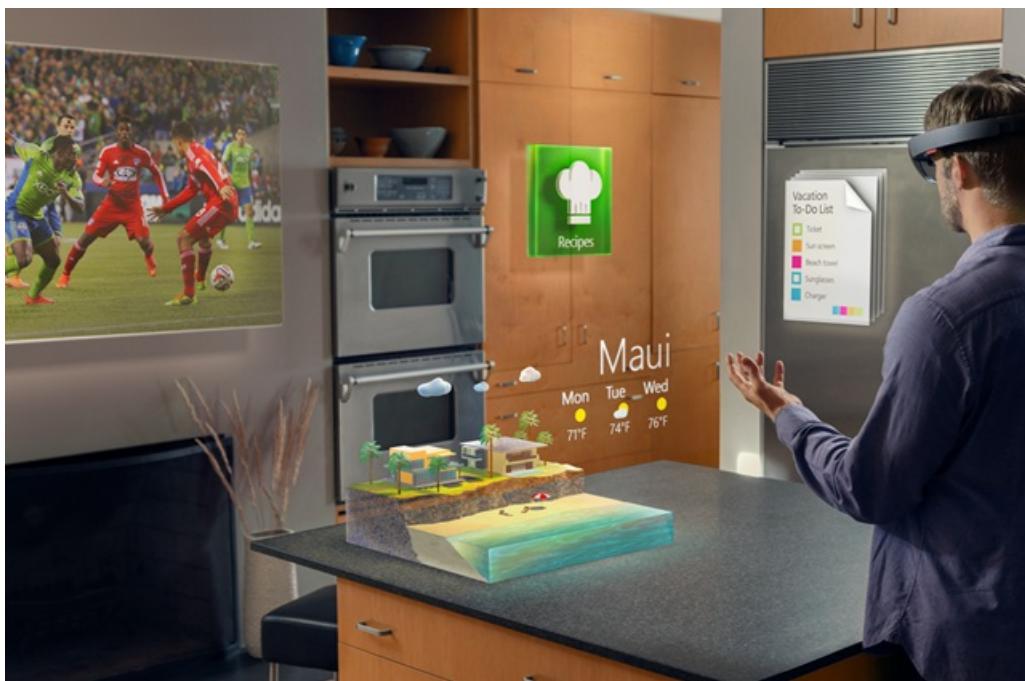
1. Finger in the ready position

2. Press finger down to tap or click

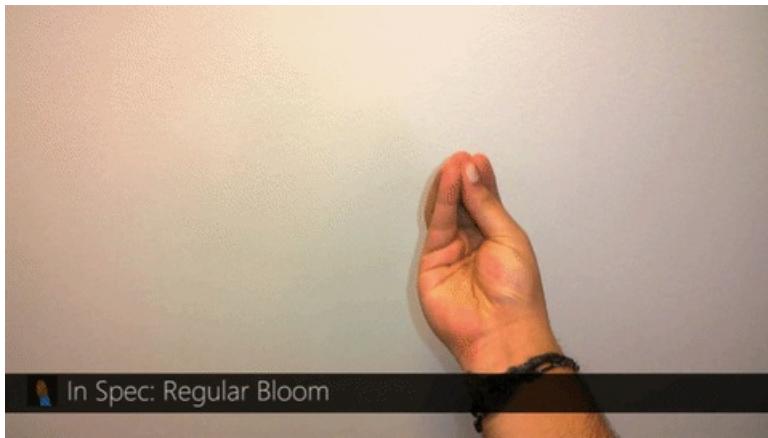
How to perform an Air tap: Raise your index finger to the ready position, press your finger down to tap or select and then back up to release.

Bloom

Bloom is the "home" gesture and is reserved for that alone. It is a special system action that is used to go back to the Start Menu. It is equivalent to pressing the Windows key on a keyboard or the Xbox button on an Xbox controller. The user can use either hand.



To do the bloom gesture on HoloLens, hold out your hand, palm up, with your fingertips together. Then open your hand. Note, you can also always return to Start by saying "Hey Cortana, Go Home". Apps cannot react specifically to a home action, as these are handled by the system.



How to perform the bloom gesture on HoloLens.

Composite gestures

Apps can recognize more than just individual taps. By combining tap, hold and release with the movement of the hand, more complex composite gestures can be performed. These composite or high-level gestures build on the low-level spatial input data (from Air tap and Bloom) that developers have access to.

COMPOSITE GESTURE	HOW TO APPLY
Air tap	The Air tap gesture (as well as the other gestures below) reacts only to a specific tap. To detect other taps, such as Menu or Grasp, your app must directly use the lower-level interactions described in two key component gestures section above.
Tap and hold	Hold is simply maintaining the downward finger position of the air tap. The combination of air tap and hold allows for a variety of more complex "click and drag" interactions when combined with arm movement such as picking up an object instead of activating it or "mousedown" secondary interactions such as showing a context menu. Caution should be used when designing for this gesture however, as users can be prone to relaxing their hand postures during the course of any extended gesture.
Manipulation	Manipulation gestures can be used to move, resize or rotate a hologram when you want the hologram to react 1:1 to the user's hand movements. One use for such 1:1 movements is to let the user draw or paint in the world. The initial targeting for a manipulation gesture should be done by gaze or pointing. Once the tap and hold starts, any manipulation of the object is then handled by hand movements, freeing the user to look around while they manipulate.

Navigation

Navigation gestures operate like a virtual joystick, and can be used to navigate UI widgets, such as radial menus. You tap and hold to start the gesture and then move your hand within a normalized 3D cube, centered around the initial press. You can move your hand along the X, Y or Z axis from a value of -1 to 1, with 0 being the starting point.

Navigation can be used to build velocity-based continuous scrolling or zooming gestures, similar to scrolling a 2D UI by clicking the middle mouse button and then moving the mouse up and down.

Navigation with rails refers to the ability of recognizing movements in certain axis until certain threshold is reached on that axis. This is only useful, when movement in more than one axis is enabled in an application by the developer, e.g. if an application is configured to recognize navigation gestures across X, Y axis but also specified X axis with rails. In this case system will recognize hand movements across X axis as long as they remain within an imaginary rails (guide) on X axis, if hand movement also occurs Y axis.

Within 2D apps, users can use vertical navigation gestures to scroll, zoom, or drag inside the app. This injects virtual finger touches to the app to simulate touch gestures of the same type. Users can select which of these actions take place by toggling between the tools on the bar above the app, either by selecting the button or saying '<Scroll/Drag/Zoom> Tool'.

Gesture recognizers

One benefit of using gesture recognition is that you can configure a gesture recognizer just for the gestures the currently targeted hologram can accept. The platform will do only the disambiguation necessary to distinguish those particular supported gestures. That way, a hologram that just supports air tap can accept any length of time between press and release, while a hologram that supports both tap and hold can promote the tap to a hold after the hold time threshold.

Hand recognition

HoloLens recognizes hand gestures by tracking the position of either or both hands that are visible to the device. HoloLens sees hands when they are in either the **ready state** (back of the hand facing you with index finger up) or the **pressed state** (back of the hand facing you with the index finger down). When hands are in other poses, the HoloLens will ignore them.

For each hand that HoloLens detects, you can access its position (without orientation) and its pressed state. As the hand nears the edge of the gesture frame, you're also provided with a direction vector, which you can show to the user so they know how to move their hand to get it back where HoloLens can see it.

Gesture frame

For gestures on HoloLens, the hand must be within a "gesture frame", in a range that the gesture-sensing cameras can see appropriately (very roughly from nose to waist, and between the shoulders). Users need to be trained on this area of recognition both for success of action and for their own comfort (many users will initially assume that the gesture frame must be within their view through HoloLens, and hold their arms up uncomfortably in order to interact). When using the HoloLens Clicker, your hands do

not need to be within the gesture frame.

In the case of continuous gestures in particular, there is some risk of users moving their hands outside of the gesture frame while in mid-gesture (while moving some holographic object, for example), and losing their intended outcome.

There are three things that you should consider:

- User education on the gesture frame's existence and approximate boundaries (this is taught during HoloLens setup).
- Notifying users when their gestures are nearing/breaking the gesture frame boundaries within an application, to the degree that a lost gesture will lead to undesired outcomes. Research has shown the key qualities of such a notification system, and the HoloLens shell provides a good example of this type of notification (visual, on the central cursor, indicating the direction in which boundary crossing is taking place).
- Consequences of breaking the gesture frame boundaries should be minimized. In general, this means that the outcome of a gesture should be stopped at the boundary, but not reversed. For example, if a user is moving some holographic object across a room, movement should stop when the gesture frame is breached, but **not** be returned to the starting point. The user may experience some frustration then, but may more quickly understand the boundaries, and not have to restart their full intended actions each time.

See also

- [Gaze targeting](#)
- [Voice design](#)
- [MR Input 211: Gesture](#)
- [Gestures and motion controllers in Unity](#)
- [Gaze, gestures, and motion controllers in DirectX](#)
- [Motion controllers](#)

Voice design

11/6/2018 • 5 minutes to read • [Edit Online](#)

Gaze, gesture and voice (GGV) are the primary means of interaction on HoloLens. **Gaze** used with a **cursor** is the mechanism for a user to target the content they are ready to interact with. **Gesture** or **voice** are the intention mechanisms. Gaze can be used with either gesture or voice to complete an interaction.

On immersive headsets, the primary means of interaction are gaze-and-commit and point-and-commit (with a **motion controller**). If the user has a headset with voice capabilities, voice can be used in combination with gaze or point to complete an action.

While designing apps, you should consider how you can make these interactions work together well.

Device support

FEATURE	HOLOLENS	IMMERSIVE HEADSETS
Voice	✓ <input type="checkbox"/>	✓ <input type="checkbox"/> (with headset attached)

How to use voice

Consider adding voice commands to any experience that you build. Voice is a powerful and convenient way control the system and apps. Because users speak with a variety of dialects and accents, proper choice of speech keywords will make sure that your users' commands are interpreted unambiguously.

Best practices

Below are some practices that will aid in smooth speech recognition.

- **Use concise commands** - When possible, choose keywords of two or more syllables. One-syllable words tend to use different vowel sounds when spoken by persons of different accents. Example: "Play video" is better than "Play the currently selected video"
- **Use simple vocabulary** - Example: "Show note" is better than "Show placard"
- **Make sure commands are non destructive** - Make sure any action that can be taken by a speech command is non destructive and can easily be undone in case another person speaking near the user accidentally triggers a command.
- **Avoid similar sounding commands** - Avoid registering multiple speech commands that sound very similar. Example: "Show more" and "Show store" can be very similar sounding.
- **Unregister your app when not in use** - When your app is not in a state in which a particular speech command is valid, consider unregistering it so that other commands are not confused for that one.
- **Test with different accents** - Test your app with users of different accents.
- **Maintain voice command consistency** - If "Go back" goes to the previous page, maintain this behavior in your applications.
- **Avoid using system commands** - The following voice commands are reserved for the system. These should not be used by applications.
 - "Hey Cortana"
 - "Select"

What users can say

As a user targets any button through gaze or pointing, they can say the word "**Select**" to activate that button. "Select" is one of the low power keywords that is always listened for. Going further, a user can also use "button grammar" across the system or in apps. For example, while looking at an app, a user can say the command "Remove" (which is in the app bar) to remove the app from the world.

See it, say it

Windows Mixed Reality has employed a "see it, say it" voice model where **labels on buttons are identical to the associated voice commands**. Because there isn't any dissonance between the label and the voice command, users can better understand what to say to control the system. To reinforce this, while dwelling on a button, a "**voice dwell tip**" appears to communicate which buttons are voice enabled.



Examples of "see it, say it"

Voice's strengths

Voice input is a natural way to communicate our intents. Voice is especially good at interface **traversals** because it can help users cut through multiple steps of an interface (a user might say "go back" while looking at Web page, instead of having to go up and hit the back button in the app). This small time savings has a powerful **emotional effect** on user's perception of the experience and gives them a small amount superpower. Using voice is also a convenient input method when we have our arms full or are **multi-tasking**. On devices where typing on a keyboard is difficult, **voice dictation** can be an efficient alternative way to input. Lastly, in some cases when the **range of accuracy** for gaze and gesture are limited, Voice might be a user's only trusted method input.

How using voice can benefit the user

- Reduces time - it should make the end goal more efficient.
- Minimizes effort - it should make tasks more fluid and effortless.
- Reduces cognitive load - it's intuitive, easy to learn, and remember.
- It's socially acceptable - it should fit in with societal norms in terms of behavior.

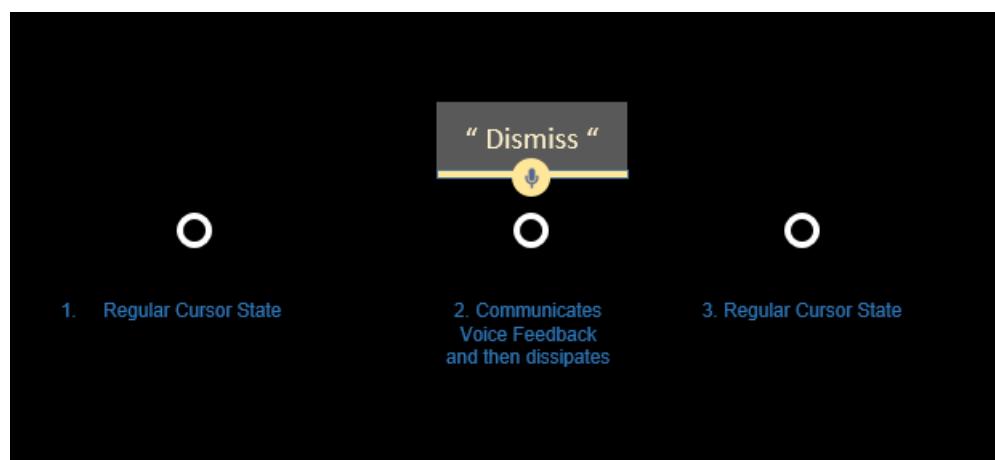
- It's routine - voice can readily become a habitual behavior.

Voice's weaknesses

Voice also has some weaknesses. Fine-grained control is one of them. (for example a user might say "louder," but can't say how much. "A little" is hard to quantify. Moving or scaling things with voice is also difficult (voice does not offer the granularity of control). Voice can also be imperfect. Sometimes a voice system incorrectly hears a command or fails to hear a command. Recovering from such an errors is a challenge in any interface. Lastly, voice may not be socially acceptable in public places. There are some things that users can't or shouldn't say. These cliffs allow speech to be used for what it is best at.

Voice feedback states

When Voice is applied properly, the user understands **what they can say and get clear feedback** the system **heard them correctly**. These two signals make the user feel confident in using Voice as a primary input. Below is a diagram showing what happens to the cursor when voice input is recognized and how it communicates that to the user.



Voice feedback states for cursor

Top things users should know about "speech" on Windows Mixed Reality

- Say "**Select**" while targeting a button (you can use this anywhere to click a button).
- You can say the **label name of an app bar button** in some apps to take an action. For example, while looking at an app, a user can say the command "Remove" to remove the app from the world (this saves time from having to click it with your hand).
- You can initiate Cortana listening by saying "**Hey Cortana.**" You can ask her questions ("Hey Cortana, how tall is the Eiffel tower"), tell her to open an app ("Hey Cortana, open Netflix"), or tell her to bring up the Start Menu ("Hey Cortana, take me home") and more.

Common questions and concerns users have about voice

- What can I say?
- How do I know the system heard me correctly?
 - The system keeps getting my voice commands wrong.
 - It doesn't react when I give it a voice command.
- It reacts the wrong way when I give it a voice command.
- How do I target my voice to a specific app or app command?
- Can I use voice to command things out the holographic frame on HoloLens?

See also

- Gestures
- Gaze targeting

What is a hologram?

11/6/2018 • 4 minutes to read • [Edit Online](#)

HoloLens lets you create **holograms**, objects made of light and sound that appear in the world around you, just as if they were real objects. Holograms respond to your [gaze](#), [gestures](#) and [voice commands](#), and can interact with [real-world surfaces](#) around you. With holograms, you can create digital objects that are part of your world.

Device support

FEATURE	HOLOLENS	IMMERSIVE HEADSETS
Holograms	✓ <input type="checkbox"/>	

A hologram is made of light and sound

The holograms that HoloLens [renders](#) appear in the holographic frame directly in front of the user's eyes. Holograms add light to your world, which means that you see both the light from the display and the light from your surroundings. HoloLens doesn't remove light from your eyes, so holograms can't be rendered with the color black. Instead, black content appears as transparent.

Holograms can have many different appearances and behaviors. Some are realistic and solid, and others are cartoonish and ethereal. Holograms can highlight features in your surroundings, and they can be elements in your app's user interface.



Holograms can also make [sounds](#), which will appear to come from a specific place in your surroundings. On HoloLens, sound comes from two speakers that are located directly above your ears, without covering them. Similar to the displays, the speakers are additive, introducing new sounds without blocking the sounds from your environment.

A hologram can be placed in the world or tag along with you

When you have a particular location where you want a hologram, you can [place](#) it precisely there in the world. As you walk around that hologram, it will appear stable relative to the world around you. If you use a [spatial anchor](#) to pin that object firmly to the world, the system can even remember where you left it when you come back later.



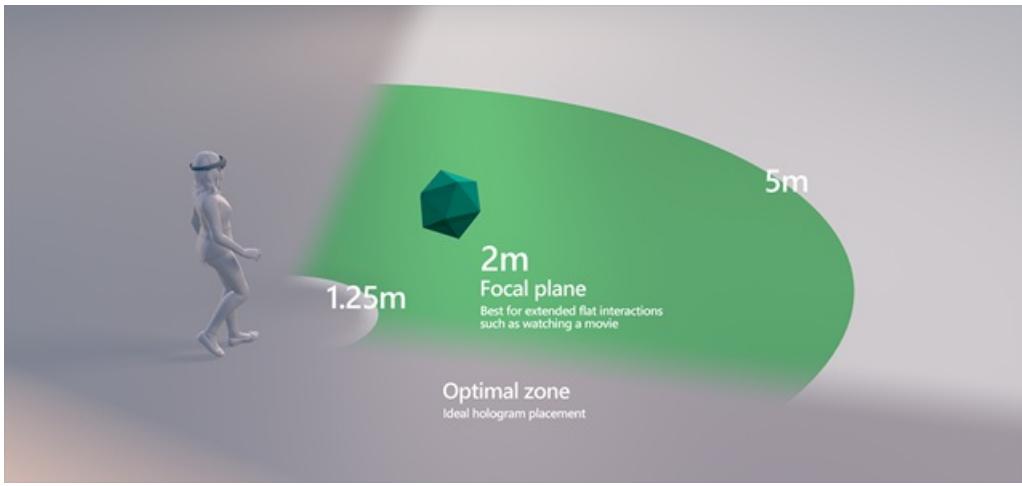
Some holograms follow the user instead. These tag-along holograms position themselves relative to the user, no matter where they walk. You may even choose to bring a hologram with you for a while and then place it on the wall once you get to another room.

Best practices

- Some scenarios may demand that holograms remain easily discoverable or visible throughout the experience. There are two high-level approaches to this kind of positioning. Let's call them "**display-locked**" and "**body-locked**".
 - Display-locked content is positionally "locked" to the device display. This is tricky for a number of reasons, including an unnatural feeling of "clingyness" that makes many users frustrated and wanting to "shake it off." In general, many designers have found it better to avoid display-locking content.
 - The body-locked approach is far more forgivable. Body-locking is when a hologram is tethered to the user's body or gaze vector, but is positioned in 3d space around the user. Many experiences have adopted a body-locking behavior where the hologram "follows" the user's gaze, which allows the user to rotate their body and move through space without losing the hologram. Incorporating a delay helps the hologram movement feel more natural. For example, some core UI of the Windows Holographic OS uses a variation on body-locking that follows the user's gaze with a gentle, elastic-like delay while the user turns their head.
- Place the hologram at a comfortable viewing distance typically about 1-2 meters away from the head.
- Provide an amount of drift for elements that must be continually in the holographic frame, or consider animating your content to one side of the display when the user changes their point of view.

Place holograms in the optimal zone - between 1.25m and 5m

Two meters is the most optimal, and the experience will degrade the closer you get from one meter. At distances nearer than one meter, holograms that regularly move in depth are more likely to be problematic than stationary holograms. Consider gracefully clipping or fading out your content when it gets too close so as not to jar the user into an unexpected experience.



A hologram interacts with you and your world

Holograms aren't only about light and sound; they're also an active part of your world. Gaze at a hologram and gesture with your hand, and a hologram can start to follow you. Give a voice command to a hologram, and it can reply.



Holograms enable personal interactions that aren't possible elsewhere. Because the HoloLens knows where it is in the world, a holographic character can look you directly in the eyes as you walk around the room.

A hologram can also interact with your surroundings. For example, you can place a holographic bouncing ball above a table. Then, with an [air tap](#), watch the ball bounce and make sound when it hits the table.

Holograms can also be occluded by real-world objects. For example, a holographic character might walk through a door and behind a wall, out of your sight.

Tips for integrating holograms and the real world

- Aligning to gravitational rules makes holograms easier to relate to and more believable. eg: Place a holographic dog on the ground & a vase on the table rather than have them floating in space.
- Many designers have found that they can even more believably integrate holograms by creating a "negative shadow" on the surface that the hologram is sitting on. They do this by creating a soft glow on the ground around the hologram and then subtracting the "shadow" from the glow. The soft glow integrates with the light from the real world and the shadow grounds the hologram in the environment.

A hologram is whatever you dream up

As a holographic developer, you have the power to break your creativity out of 2D screens and into the world around you. What will *you* build?



See also

- [Spatial sound](#)
- [Color, light and materials](#)

Holographic frame

11/6/2018 • 12 minutes to read • [Edit Online](#)

Users see the world of mixed reality through a rectangular viewport powered by their headset. On the HoloLens, this rectangular area is called the holographic frame and allows users to see digital content overlaid onto the real world around them. Designing experiences optimized for the holographic frame creates opportunities, mitigates challenges and enhances the user experience of mixed reality applications.

Designing for content

Often designers feel the need to limit the scope of their experience to what the user can immediately see, sacrificing real-world scale to ensure the user sees an object in its entirety. Similarly designers with complex applications often overload the holographic frame with content, overwhelming users with difficult interactions and cluttered interfaces. Designers creating mixed reality content need not limit their experience to directly in front of the user and within their immediate view. If the physical world around the user is mapped, then all these surfaces should be considered a potential canvas for digital content and interactions. Proper design of interactions and content within an experience should encourage the user to move around their space, directing their attention to key content, and helping see the full potential of mixed reality.

Perhaps the most important technique to encouraging movement and exploration within an app is to **let users adjust to the experience**. Give users a short period of 'task-free' time with the device. This can be as simple as placing an object in the space and letting users move around it or narrating an introduction to the experience. This time should be free of any critical tasks or specific gestures (such as air-tapping), instead the purpose to let users accommodate to viewing content through the device before requiring interactivity or progressing through the stages of the app. If this is a user's first time with the device this is especially important as they get comfortable seeing content through the holographic frame and the nature of holograms.

Large objects

Often the content an experience calls for, especially real-world content, will be larger than the holographic frame. Objects that cannot normally fit within the holographic frame should be shrunk to fit when they are first introduced (either at a smaller scale or at a distance). The key is to **let users see the full size of the object** before the scale overwhelms the frame. For example, a holographic elephant should be displayed to fit fully within the frame, allowing users to form a spatial understanding of the animal's overall shape, before sizing it to **real-world scale** near the user.

With the full size of the object in mind, users then have an expectation of where to move around and look for specific parts of that object. Similarly, in an experience with immersive content, it can help to have some way to refer back to the full size of that content. For example, if the experience involves walking around a model of a virtual house, it may help to have a smaller doll-house size version of the experience that users can trigger to understand where they are inside the house.

For an example of designing for large objects, see [Volvo Cars](#).

Many objects

Experiences with many objects or components should consider using the full space around the user to avoid cluttering the holographic frame directly in front of the user. In general, it is good practice to introduce content to an experience slowly and this is especially true with experiences that plan to serve many objects to the user. Much like with large objects, the key is to **let users understand the layout of content** in the experience, helping them gain a spatial understanding of what's around them as content is added to the experience.

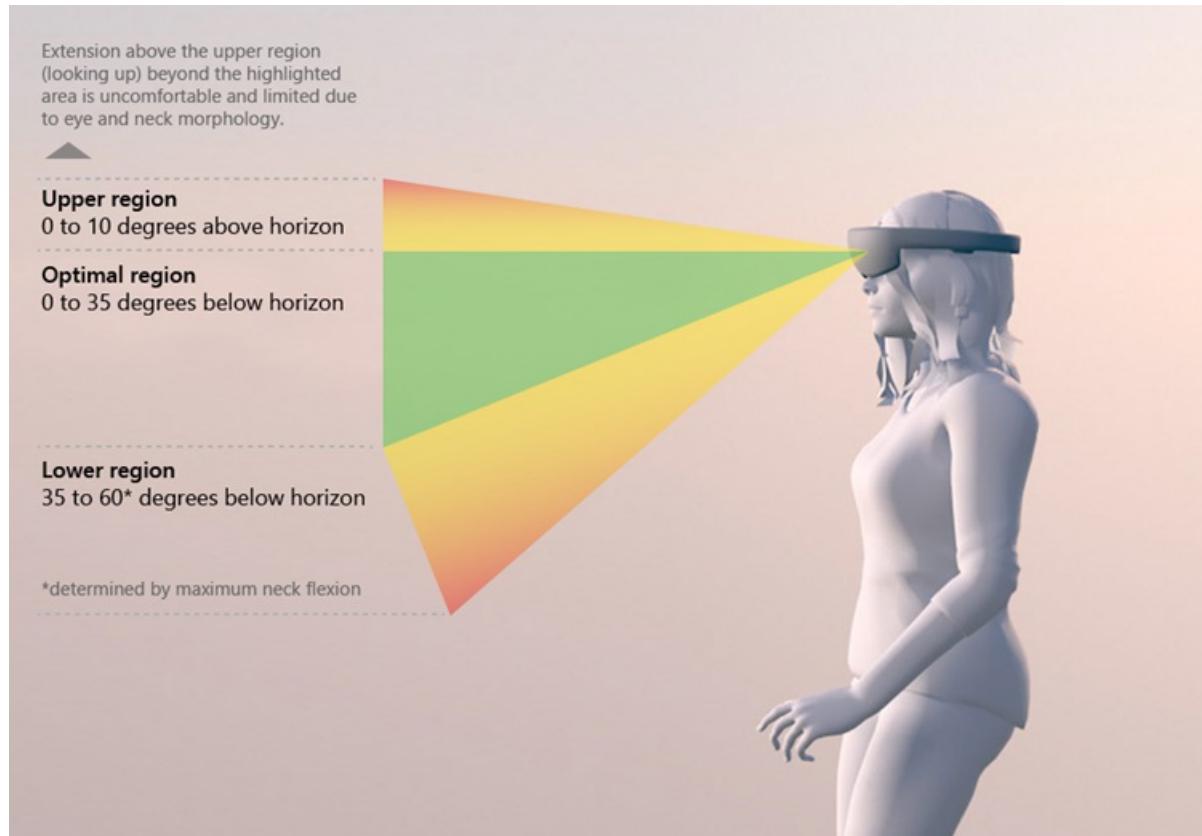
One technique to achieve this is to provide persistent points (also known as landmarks) in the experience that

anchor content to the real world. For example, a landmark could be a physical object in the real-world, such as a table where digital content appears, or a digital object, such as a set of digital screens where content frequently appears. Objects can also be placed in the periphery of the holographic frame to encourage user to look toward key content, while the discovery of content beyond the periphery can be aided by [attention directors](#).

Placing objects in the periphery can encourage users to look to the side and this can be aided by attention directors, as described below.

User comfort

For mixed reality experiences with large objects or many objects, it is crucial to consider how much head and neck movement is necessary to interact with content. Experiences can be divided into three categories in terms of head movement: **Horizontal** (side to side), **vertical** (up and down), or **immersive** (both horizontal and vertical). When possible, limit the majority of interactions to either horizontal or vertical categories, ideally with most experiences taking place in the center of the holographic frame while the user's head is in a neutral position. Avoid interactions that cause the user to constantly move their view to an unnatural head positions (for example, always looking up to access a key menu interaction).



Optimal region for content is 0 to 35 degrees below horizon

Horizontal head movement is more [comfortable](#) for frequent interactions, while vertical movements should be reserved for uncommon events. For example, an experience involving a long horizontal timeline should limit vertical head movement for interactions (like looking down at a menu).

Consider encouraging full-body movement, rather than just head movement, by placing objects around the user's space. Experiences with moving objects or large objects should pay special attention to head movement, especially where they require frequent movement along both the horizontal and vertical axes.

Interaction considerations

As with content, interactions in a mixed reality experience need not be limited to what the user can immediately see. Interactions can take place anywhere in the real-world space around the user and these interactions can help encourage users to move around and explore experiences.

Attention directors

Indicating points of interest or key interactions can be crucial to progressing users through an experience. User attention and movement of the holographic frame can be directed in subtle or heavy-handed ways. Remember to balance attention directors with periods of free exploration in mixed reality (especially at the start of an experience) to avoid overwhelming the user. In general, there are two types of attention directors:

- **Visual directors:** The easiest way to let the user know they should move in a specific direction is to provide a visual indication. This can be done through a visual effect (for example, a path the user can visually follow toward the next part of the experience) or even as simple directional arrows. Note that any visual indicator should be grounded within the user's environment, not 'attached' to the holographic frame or the cursor.
- **Audio directors:** [Spatial sound](#) can provide a powerful way to establish objects in a scene (alerting users of objects entering an experience) or to direct attention to a specific point in space (to move the user's view toward key objects). Using audio directors to guide the user's attention can be more subtle and less intrusive than visual directors. In some cases, it can be best to start with an audio director, then move on to a visual director if the user does not recognize the cue. Audio directors can also be paired with visual directors for added emphasis.

Commanding, navigation and menus

Interfaces in mixed reality experiences ideally are paired tightly with the digital content they control. As such, free-floating 2D menus are often not ideal for interaction and can be difficult for users to comfortably interact with inside the holographic frame. For experiences that do require interface elements such as menus or text fields, consider using a [tag-along method](#) to follow the holographic frame after a short delay. Avoid locking content to the frame like a heads-up display, as this can be disorienting for the user and break the sense of immersion for other digital objects in the scene.

Alternatively, consider placing interface elements directly on the specific content they control, allowing interactions to happen naturally around the user's physical space. For example, break a complex menu into separate parts. With each button or group of controls attached to the specific object the interaction affects. To take this concept further, consider the use of [interactable objects](#).

Gaze and gaze targeting

The holographic frame presents a tool for the developer to trigger interactions as well as evaluate where a user's attention dwells. [Gaze](#) is one of the [key interactions on HoloLens](#), where gaze can be paired with [gestures](#) (such as with air-tapping) or [voice](#) (allowing for shorter, more natural voice-based interactions). As such, this makes the holographic frame both a space for observing digital content as well as interacting with it. If the experience calls for interacting with multiple objects around the user's space (for example, multi-selecting objects around the user's space with gaze + gesture), consider bringing those objects into the user's view or limiting the amount of necessary head movement to promote [user comfort](#).

Gaze can also be used to track user attention through an experience and see which objects or parts of the scene the user paid the most attention to. This can be especially useful for debugging an experience, allowing for analytical tools like heatmaps to see where users are spending the most time or are missing certain objects or interactions. Gaze tracking can also provide a powerful tool for facilitators in experiences (see the [Lowe's Kitchen](#) example).

Performance

Proper use of the holographic frame is fundamental to the [performance quality](#) experiences. A common technical (and usability) challenge is overloading the user's frame with digital content, causing rendering performance to degrade. Consider instead using the full space around the user to arrange digital content, using the techniques described above, to lessen the burden of rendering and ensure an optimal display quality.

Digital content within the holographic frame of the HoloLens can also be paired with the [stabilization plane](#) for optimal performance and [hologram stability](#).

Examples

Volvo Cars

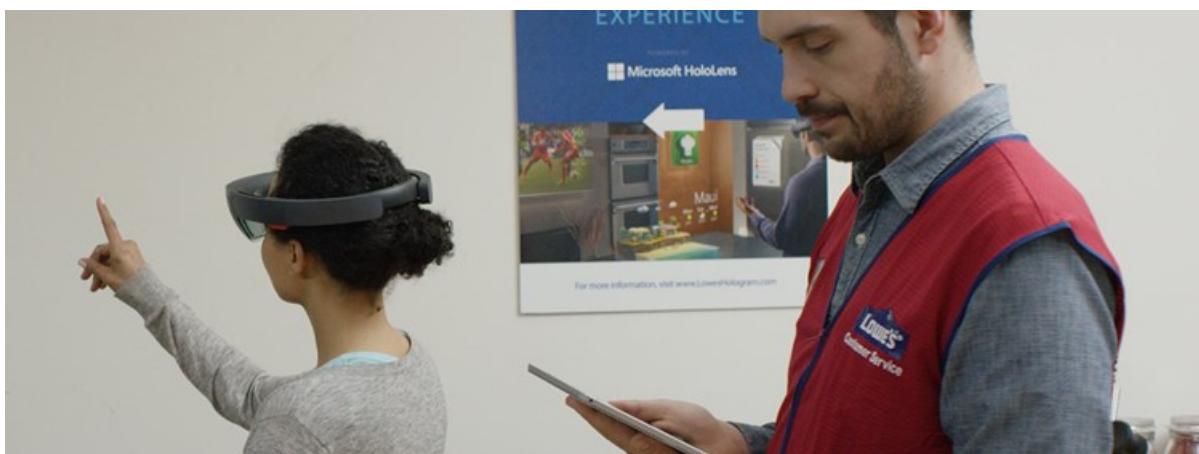
In the showroom experience from Volvo Cars, customers are invited to learn about a new car's capabilities in a HoloLens experience guided by a Volvo associate. Volvo faced a challenge with the holographic frame: a full-size car is too large to put right next to a user. The solution was to begin the experience with a physical landmark, a central table in the showroom, with a smaller digital model of the car placed on top of the table. This ensures the user is seeing the full car when it is introduced, allowing for a sense of spatial understanding once the car grows to its real-world scale later in the experience.

Volvo's experience also makes use of visual directors, creating a long visual effect from the small-scale car model on the table to a wall in the show room. This leads to a 'magic window' effect, showing the full view of the car at a distance, illustrating further features of the car at real-world scale. The head movement is horizontal, without any direct interaction from the user (instead gathering cues visually and from the Volvo associate's narration of the experience).

Lowe's Kitchen

A store experience from Lowe's invites customers into a full-scale mockup of a kitchen to showcase various remodeling opportunities as seen through the HoloLens. The kitchen in the store provides a physical backdrop for digital objects, a blank canvas of appliances, countertops, and cabinets for the mixed reality experience to unfold.

Physical surfaces act as static landmarks for the user to ground themselves in the experience, as a Lowe's associate guides the user through different product options and finishes. In this way, the associate can verbally direct the user's attention to the 'refrigerator' or 'center of the kitchen' to showcase digital content.



A Lowe's associate uses a tablet to guide customers through the HoloLens experience.

The user's experience is managed, in part, by a tablet experience controlled by the Lowe's associate. Part of the associate's role in this case would also be to limit excessive head movement, directing their attention smoothly across the points of interest in the kitchen. The tablet experience also provides the Lowe's associate with gaze data in the form of a heatmap view of the kitchen, helping understand where the user is dwelling (for example, on a specific area of cabinetry) to more accurately provide them with remodeling guidance.

For a deeper look at the Lowe's Kitchen experience, see [Microsoft's keynote at Ignite 2016](#).

Fragments

In the HoloLens game Fragments, your living room is transformed into a virtual crime scene showing clues and evidence, as well as a virtual meeting room, where you talk with characters that sit on your chairs and lean on your walls.



Fragments was designed to take place in a user's home, with characters interacting with real-world objects and surfaces.

When users initially begin the experience, they are given a short period of adjustment, where very little interaction is required, instead encouraging them to look around. This also helps ensure the room is properly mapped for the game's interactive content.

Throughout the experience, characters become focal points and act as visual directors (head movements between characters, turning to look or gesture toward areas of interest). The game also relies on more prominent visual cues when a user takes too long to find an object or event and makes heavy use of spatial audio (especially with characters voices when entering a scene).

Destination: Mars

In the Destination: Mars experience featured at [NASA's Kennedy Space Center](#), visitors were invited into an immersive trip to the surface of Mars, guided by virtual representation of legendary astronaut Buzz Aldrin.



A virtual Buzz Aldrin becomes the focal point for users in *Destination: Mars*.

As an immersive experience, these users were encouraged to look around, moving their head in all directions to see the virtual Martian landscape. Although to ensure the comfort of the users, Buzz Aldrin's narration and virtual presence provided a focal point throughout the experience. This virtual recording of Buzz (created by [Microsoft's Mixed Reality Capture Studios](#)) stood at real, human size, in the corner of the room allowing users to see him in near-complete view. Buzz's narration directed users to focus on different points in the environment (for example, a

set of Martian rocks on the floor or a mountain range in the distance) with specific scene changes or objects introduced by him.



The virtual narrators will turn to follow a user's movement, creating a powerful focal point throughout the experience.

The realistic representation of Buzz provided a powerful focal point, complete with subtle techniques to turn Buzz toward the user to feel as though he is there, speaking to you. As the user moves about the experience, Buzz will shift toward you to a threshold before returning to a neutral state if the user moves too far beyond his periphery. If the user looks way from Buzz completely (for example, to look at something elsewhere in the scene) then back to Buzz, the narrator's directional position will once again be focused on the user. Techniques like this provide a powerful sense of immersion and create a focal point within the holographic frame, reducing excessive head movement and promoting [user comfort](#).

See also

- [Interaction fundamentals](#)
- [Comfort](#)
- [Scale](#)
- [Gaze targeting](#)
- [Hologram stability](#)

Spatial mapping design

11/6/2018 • 15 minutes to read • [Edit Online](#)

Effective use of spatial mapping within HoloLens requires careful consideration of many factors.

Device support

FEATURE	HOLOLENS	IMMERSIVE HEADSETS
Spatial mapping	✓ <input type="checkbox"/>	

Why is spatial mapping important?

Spatial mapping makes it possible to place objects on real surfaces. This helps anchor objects in the user's world and takes advantage of real world depth cues. Occluding your holograms based on other holograms and real world objects helps convince the user that these holograms are actually in their space. Holograms floating in space or moving with the user will not feel as real. When possible, place items for comfort.

Visualize surfaces when placing or moving holograms (use a simple projected grid). This will help the user know where they can best place their holograms, and shows the user if the spot they are trying to place the hologram hasn't been mapped yet. You can "billboard items" toward the user if they end up at too much of an angle.

What influences spatial mapping quality?

In order to provide the best user experience, it is important to understand the factors that affect the quality of spatial mapping data gathered by HoloLens.

Errors in spatial mapping data fall into one of three categories:

- **Holes:** Real-world surfaces are missing from the spatial mapping data.
- **Hallucinations:** Surfaces exist in the spatial mapping data that do not exist in the real world.
- **Bias:** Surfaces in the spatial mapping data are imperfectly aligned with real-world surfaces, either pushed in or pulled out.

Several factors can affect the frequency and severity of these errors:

- **User motion**

- How the user moves through their environment determines how well the environment will be scanned, so the user may require guidance in order to achieve a good scan.
- The camera used for scanning provides data within a 70-degree cone, from a minimum of 0.8 meters to a maximum of 3.1 meters distance from the camera. Real-world surfaces will only be scanned within this field of view. Note that these values are subject to change in future versions.
- If the user never gets within 3.1 meters of an object then it will not be scanned.
- If the user only views an object from a distance of less than 0.8 meters then it will not be scanned (this avoids scanning the user's hands).
- If the user never looks upward (which is fairly normal) then the ceiling will likely not be scanned.
- If a user never looks behind furniture or a wall then the objects occluded by them will not be scanned.
- Surfaces tend to be scanned at a higher quality when they are viewed head-on rather than at a shallow

angle.

- If the head-tracking system of the HoloLens fails momentarily (which may happen due to rapid user motion, poor lighting, featureless walls or the cameras becoming covered), this can introduce errors in the spatial mapping data. Any such errors will be corrected over time as the user continues to move around and scan their environment.

- **Surface materials**

- The materials found on real-world surfaces vary greatly. These impact the quality of spatial mapping data because they affect how infrared light is reflected.
- Dark surfaces may not scan until they come closer to the camera, because they reflect less light.
- Some surfaces may be so dark that they reflect too little light to be scanned from any distance, so they will introduce hole errors at the location of the surface and sometimes also behind the surface.
- Particularly shiny surfaces may only scan when viewed head-on, and not when viewed from a shallow angle.
- Mirrors, because they create illusory reflections of real spaces and surfaces, can cause both hole errors and hallucination errors.

- **Scene motion**

- Spatial mapping adapts rapidly to changes in the environment, such as moving people or opening and closing doors.
- However, spatial mapping can only adapt to changes in an area when that area is clearly visible to the camera that is used for scanning.
- Because of this, it is possible for this adaptation to lag behind reality, which can cause hole or hallucination errors.
- As an example, a user scans a friend and then turns around while the friend leaves the room. A 'ghost' representation of the friend (a hallucination error) will persist in the spatial mapping data, until the user turns back around and re-scans the space where the friend was standing.

- **Lighting interference**

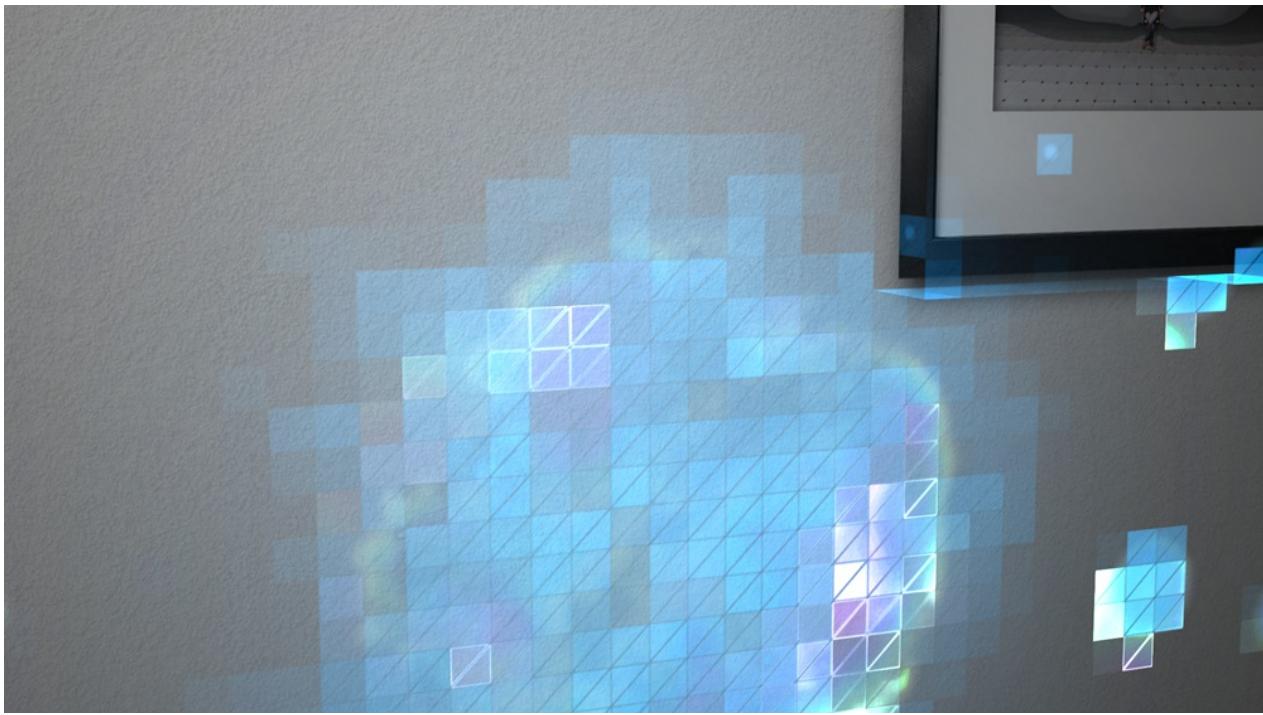
- Ambient infrared light in the scene may interfere with scanning, for example strong sunlight coming through a window.
- Particularly shiny surfaces may interfere with the scanning of nearby surfaces, the light bouncing off them causing bias errors.
- Shiny surfaces reflecting light directly back into the camera may interfere with nearby space, either by causing floating mid-air hallucinations or by delaying adaptation to scene motion.
- Two HoloLens devices in the same room should not interfere with one another, but the presence of more than five HoloLens devices may cause interference.

It may be possible to avoid or correct for some of these errors. However, you should design your application so that the user is able to achieve their goals even in the presence of errors in the spatial mapping data.

The environment scanning experience

HoloLens learns about the surfaces in its environment as the user looks at them. Over time, the HoloLens builds up a scan of all parts of the environment that have been observed. It also updates the scan as changes in the environment are observed. This scan will automatically persist between app launches.

Each application that uses spatial mapping should consider providing a 'scanning experience'; the process through which the application guides the user to scan surfaces that are necessary for the application to function correctly.



Example of scanning

The nature of this scanning experience can vary greatly depending upon each application's needs, but two main principles should guide its design.

Firstly, **clear communication with the user is the primary concern**. The user should always be aware of whether the application's requirements are being met. When they are not being met, it should be immediately clear to the user why this is so and they should be quickly led to take the appropriate action.

Secondly, **applications should attempt to strike a balance between efficiency and reliability**. When it is possible to do so **reliably**, applications should automatically analyze spatial mapping data to save the user time. When it is not possible to do so reliably, applications should instead enable the user to quickly provide the application with the additional information it requires.

To help design the right scanning experience, consider which of the following possibilities are applicable to your application:

- **No scanning experience**

- An application may function perfectly without any guided scanning experience; it will learn about surfaces that are observed in the course of natural user movement.
- For example an application that lets the user draw on surfaces with holographic spraypaint requires knowledge only of the surfaces currently visible to the user.
- The environment may be completely scanned already if it is one in which the user has already spent a lot of time using the HoloLens.
- Bear in mind however that the camera used by spatial mapping can only see 3.1m in front of the user, so spatial mapping will not know about any more distant surfaces unless the user has observed them from a closer distance in the past.
- So the user understands which surfaces have been scanned, the application should provide visual feedback to this effect, for example casting virtual shadows onto scanned surfaces may help the user place holograms on those surfaces.
- For this case, the spatial surface observer's bounding volumes should be updated each frame to a body-locked **spatial coordinate system**, so that they follow the user.

- **Find a suitable location**

- An application may be designed for use in a location with specific requirements.

- For example, the application may require an empty area around the user so they can safely practice holographic kung-fu.
- Applications should communicate any specific requirements to the user up-front, and reinforce them with clear visual feedback.
- In this example, the application should visualize the extent of the required empty area and visually highlight the presence of any undesired objects within this zone.
- For this case, the spatial surface observer's bounding volumes should use a world-locked [spatial coordinate system](#) in the chosen location.

- **Find a suitable configuration of surfaces**

- An application may require a specific configuration of surfaces, for example two large, flat, opposing walls to create a holographic hall of mirrors.
- In such cases the application will need to analyze the surfaces provided by spatial mapping to detect suitable surfaces, and direct the user toward them.
- The user should have a fallback option if the application's surface analysis is not completely reliable. For example, if the application incorrectly identifies a doorway as a flat wall, the user needs a simple way to correct this error.

- **Scan part of the environment**

- An application may wish to only capture part of the environment, as directed by the user.
- For example, the application scans part of a room so the user may post a holographic classified ad for furniture they wish to sell.
- In this case, the application should capture spatial mapping data within the regions observed by the user during their scan.

- **Scan the whole room**

- An application may require a scan of all of the surfaces in the current room, including those behind the user.
- For example, a game may put the user in the role of Gulliver, under siege from hundreds of tiny Lilliputians approaching from all directions.
- In such cases, the application will need to determine how many of the surfaces in the current room have already been scanned, and direct the user's gaze to fill in significant gaps.
- The key to this process is providing visual feedback that makes it clear to the user which surfaces have not yet been scanned. The application could for example use [distance-based fog](#) to visually highlight regions that are not covered by spatial mapping surfaces.

- **Take an initial snapshot of the environment**

- An application may wish to ignore all changes in the environment after taking an initial 'snapshot'.
- This may be appropriate to avoid disruption of user-created data that is tightly coupled to the initial state of the environment.
- In this case, the application should make a copy of the spatial mapping data in its initial state once the scan is complete.
- Applications should continue receiving updates to spatial mapping data if holograms are still to be correctly occluded by the environment.
- Continued updates to spatial mapping data also allow visualizing any changes that have occurred, clarifying to the user the differences between prior and present states of the environment.

- **Take user-initiated snapshots of the environment**

- An application may only wish to respond to environmental changes when instructed by the user.
- For example, the user could create multiple 3D 'statues' of a friend by capturing their poses at different moments.

- **Allow the user to change the environment**

- An application may be designed to respond in real-time to any changes made in the user's environment.
- For example, the user drawing a curtain could trigger 'scene change' for a holographic play taking place on the other side.

- **Guide the user to avoid errors in the spatial mapping data**

- An application may wish to provide guidance to the user while they are scanning their environment.
- This can help the user to avoid certain kinds of [errors in the spatial mapping data](#), for example by staying away from sunlit windows or mirrors.

One additional detail to be aware of is that the 'range' of spatial mapping data is not unlimited. Whilst spatial mapping does build a permanent database of large spaces, it only makes that data available to applications in a 'bubble' of limited size around the user. Thus if you start at the beginning of a long corridor and walk far enough away from the start, then eventually the spatial surfaces back at the beginning will disappear. You can of course mitigate this by caching those surfaces in your application after they have disappeared from the available spatial mapping data.

Mesh processing

It may help to detect common types of errors in surfaces and to filter, remove or modify the spatial mapping data as appropriate.

Bear in mind that spatial mapping data is intended to be as faithful as possible to real-world surfaces, so any processing you apply risks shifting your surfaces further from the 'truth'.

Here are some examples of different types of mesh processing that you may find useful:

- **Hole filling**

- If a small object made of a dark material fails to scan, it will leave a hole in the surrounding surface.
- Holes affect occlusion: holograms can be seen 'through' a hole in a supposedly opaque real-world surface.
- Holes affect raycasts: if you are using raycasts to help users interact with surfaces, it may be undesirable for these rays to pass through holes. One mitigation is to use a bundle of multiple raycasts covering an appropriately sized region. This will allow you to filter 'outlier' results, so that even if one raycast passes through a small hole, the aggregate result will still be valid. However, be aware that this approach comes at a computational cost.
- Holes affect physics collisions: an object controlled by physics simulation may drop through a hole in the floor and become lost.
- It is possible to algorithmically fill such holes in the surface mesh. However, you will need to tune your algorithm so that 'real holes' such as windows and doorways do not get filled in. It can be difficult to reliably differentiate 'real holes' from 'imaginary holes', so you will need to experiment with different heuristics such as 'size' and 'boundary shape'.

- **Hallucination removal**

- Reflections, bright lights and moving objects can leave small lingering 'hallucinations' floating in mid-air.
- Hallucinations affect occlusion: hallucinations may become visible as dark shapes moving in front of and occluding other holograms.
- Hallucinations affect raycasts: if you are using raycasts to help users interact with surfaces, these rays could hit a hallucination instead of the surface behind it. As with holes, one mitigation is to use many raycasts instead of a single raycast, but again this will come at a computational cost.
- Hallucinations affect physics collisions: an object controlled by physics simulation may become stuck

against a hallucination and be unable to move through a seemingly clear area of space.

- It is possible to filter such hallucinations from the surface mesh. However, as with holes, you will need to tune your algorithm so that real small objects such as lamp-stands and door handles do not get removed.

- **Smoothing**

- Spatial mapping may return surfaces that appear to be rough or 'noisy' in comparison to their real-world counterparts.
- Smoothness affects physics collisions: if the floor is rough, a physically simulated golf ball may not roll smoothly across it in a straight line.
- Smoothness affects rendering: if a surface is visualized directly, rough surface normals can affect its appearance and disrupt a 'clean' look. It is possible to mitigate this by using appropriate lighting and textures in the shader that is used to render the surface.
- It is possible to smooth out roughness in a surface mesh. However, this may push the surface further away from the corresponding real-world surface. Maintaining a close correspondence is important to produce accurate hologram occlusion, and to enable users to achieve precise and predictable interactions with holographic surfaces.
- If only a cosmetic change is required, it may be sufficient to smooth vertex normals without changing vertex positions.

- **Plane finding**

- There are many forms of analysis that an application may wish to perform on the surfaces provided by spatial mapping.
- One simple example is 'plane finding'; identifying bounded, mostly-planar regions of surfaces.
- Planar regions can be used as holographic work-surfaces, regions where holographic content can be automatically placed by the application.
- Planar regions can constrain the user interface, to guide users to interact with the surfaces that best suit their needs.
- Planar regions can be used as in the real world, for holographic counterparts to functional objects such as LCD screens, tables or whiteboards.
- Planar regions can define play areas, forming the basis of videogame levels.
- Planar regions can aid virtual agents to navigate the real world, by identifying the areas of floor that real people are likely to walk on.

Prototyping and debugging

Useful tools

- The [HoloLens emulator](#) can be used to develop applications using spatial mapping without access to a physical HoloLens. It allows you to simulate a live session on a HoloLens in a realistic environment, with all of the data your application would normally consume, including HoloLens motion, spatial coordinate systems and spatial mapping meshes. This can be used to provide reliable, repeatable input, which can be useful for debugging problems and evaluating changes to your code.
- To reproduce a scenarios, capture spatial mapping data over the network from a live HoloLens, then save it to disk and reuse it in subsequent debugging sessions.
- The [Windows device portal 3D view](#) provides a way to see all of the spatial surfaces currently available via the spatial mapping system. This provides a basis of comparison for the spatial surfaces inside your application; for example you can easily tell if any spatial surfaces are missing or are being displayed in the wrong place.

General prototyping guidance

- Because [errors](#) in the spatial mapping data may strongly affect your user's experience, we recommend that you test your application in a wide variety of environments.

- Don't get trapped in the habit of always testing in the same location, for example at your desk. Make sure to test on various surfaces of different positions, shapes, sizes and materials.
- Similarly, while synthetic or recorded data can be useful for debugging, don't become too reliant upon the same few test cases. This may delay finding important issues that more varied testing would have caught earlier.
- It is a good idea to perform testing with real (and ideally un-coached) users, because they may not use the HoloLens or your application in exactly the same way that you do. In fact, it may surprise you how divergent people's behavior, knowledge and assumptions can be!

See also

- [Room scan visualization](#)
- [Spatial sound design](#)
- [Persistence in Unity](#)

Spatial sound design

11/6/2018 • 6 minutes to read • [Edit Online](#)

Spatial sound is a powerful tool for immersion, accessibility, and UX design in mixed reality applications.

If you've ever played [Marco Polo](#), or had someone call your phone to help you locate it, you are already familiar with the importance of spatial sound. We use sound cues in our daily lives to locate objects, get someone's attention, or get a better understanding of our environment. The more closely your app's sound behaves like it does in the real world, the more convincing and engaging your virtual world will be.

Device support

FEATURE	HOLOLENS	IMMERSIVE HEADSETS
Spatial sound	✓ <input type="checkbox"/>	✓ <input type="checkbox"/>

Four key things spatial sound does for mixed reality development

By default, sounds are played back in stereo. This means the sound will play with no spatial position, so the user does not know where the sound came from. Spatial sound does four key things for mixed reality development:

Grounding

Without sound, virtual objects effectively cease to exist when we turn our head away from them. Just like real objects, you want to be able to hear these objects even when you can't see them, and you want to be able to locate them anywhere around you. Just as virtual objects need to be grounded visually to blend with your real world, they also need to be grounded audibly. Spatial sound seamlessly blends your real world audio environment with the digital audio environment.

User attention

In mixed reality experiences, you can't assume where your user is looking and expect them to see something you place in the world visually. But users can always hear a sound play even when the object playing the sound is behind them. People are used to having their attention drawn by sound - we instinctually look toward an object that we hear around us. When you want to direct your user's gaze to a particular place, rather than using an arrow to point them visually, placing a sound in that location is a very natural and fast way to guide them.

Immersion

When objects move or collide, we usually hear those interactions between materials. So when your objects don't make the same sound they would in the real world, a level of immersion is lost - like watching a scary movie with the volume all the way down. All sounds in the real world come from a particular point in space - when we turn our heads, we hear the change in where those sounds are coming from relative to our ears, and we can track the location of any sound this way. Spatialized sounds make up the "feel" of a place beyond what we can see.

Interaction design

In most traditional interactive experiences, interaction sounds like UI sound effects are played in standard mono or stereo. But because everything in mixed reality exists in 3D space - including the UI - these objects benefit from

spatialized sounds. When we press a button in the real world, the sound we hear comes from that button. By spatializing interaction sounds, we again provide a more natural and realistic user experience.

Best practices when using spatial sound

Real sounds work better than synthesized or unnatural sounds

The more familiar your user is with a type of sound, the more real it will feel, and the more easily they will be able to locate it in their environment. A human voice, for example, is a very common type of sound, and your users will locate it just as quickly as a real person in the room talking to them.

Expectation trumps simulation

If you are used to a sound coming from a particular direction, your attention will be guided in that direction regardless of spatial cues. For example, most of the time that we hear birds, they are above us. Playing the sound of a bird will most likely cause your user to look up, even if you place the sound below them. This is usually confusing, and it is recommended that you work with expectations like these rather than going against them for a more natural experience.

Most sounds should be spatialized

As mentioned above, everything in Mixed Reality exists in 3D space - your sounds should as well. Even music can sometimes benefit from spatialization, particularly when it's tied to a menu or some other UI.

Avoid invisible emitters

Because we've been conditioned to look at sounds that we hear around us, it can be an unnatural and even unnerving experience to locate a sound that has no visual presence. Sounds in the real world don't come from empty space, so be sure that if an audio emitter is placed within the user's immediate environment that it can also be seen.

Avoid spatial masking

Spatial sound relies on very subtle acoustic cues that can be overpowered by other sounds. If you do have stereo music or ambient sounds, make sure they are low enough in the mix to give room for the details of your spatialized sounds that will allow your users to locate them easily, and keep them sounding realistic and natural.

General concepts to keep in mind when using spatial sound

Spatial sound is a simulation

The most frequent use of spatial sound is making a sound seem as though it is emanating from a real or virtual object in the world. Thus, spatialized sounds may make the most sense coming from such objects.

Note that the perceived accuracy of spatial sound means that a sound shouldn't necessarily emit from the center of an object, as the difference will be noticeable depending on the size of the object and distance from the user. With small objects, the center point of the object is usually sufficient. For larger objects, you may want a sound emitter or multiple emitters at the specific location within the object that is supposed to be producing the sound.

Normalize all sounds

Distance attenuation happens quickly within the first meter from the user, as it does in the real world. All audio files should be normalized to ensure physically accurate distance attenuation, and ensure that a sound can be heard when several meters away (when applicable). The spatial audio engine will handle the attenuation necessary for a sound to "feel" like it's at a certain distance (with a combination of attenuation and "distance cues"), and applying any attenuation on top of that could reduce the effect. Outside of simulating a real object, the initial distance decay of *spatial sound* sounds will likely be more than enough for a proper mix of your audio.

Object discovery and user interfaces

When using audio cues to direct the user's attention beyond their current view, the sound should be audible and prominent in the mix, well above any stereo sounds, and any other spatialized sounds which might distract from the directional audio cue. For sounds and music that are associated with an element of the user interface (e.g. a menu), the sound emitter should be attached to that object. Stereo and other non-positional audio playing can make spatialized elements difficult for users to locate (See above: Avoid spatial masking).

Use spatial sound over standard 3D sound as much as possible

In mixed reality, for the best user experience, 3D audio should be achieved using spatial sound rather than legacy 3D audio technologies. In general, the improved spatialization is worth the small CPU cost over standard 3D sound. Standard 3D audio can be used for low-priority sounds, sounds that are spatialized but not necessarily tied to a physical or virtual object, and objects that the user never need locate to interact with the app.

See also

- [Spatial sound](#)
- [Spatial mapping](#)

Motion controllers

11/6/2018 • 13 minutes to read • [Edit Online](#)

Motion controllers are [hardware accessories](#) that allow users to take action in mixed reality. An advantage of motion controllers over [gestures](#) is that the controllers have a precise position in space, allowing for fine grained interaction with digital objects. For Windows Mixed Reality immersive headsets, motion controllers are the primary way that users will take action in their world.



Device support

FEATURE	HOLOLENS	IMMERSIVE HEADSETS
Motion controllers		✓ <input type="checkbox"/>

Hardware details

Windows Mixed Reality motion controllers offer precise and responsive tracking of movement in your field of view using the sensors in the immersive headset, meaning there is no need to install hardware on the walls in your space. These motion controllers will offer the same ease of setup and portability as Windows Mixed Reality immersive headsets. Our device partners plan to market and sell these controllers on retail shelves this holiday.

Get to know your controller



Get to know your controller

Features:

- Optical tracking
- Trigger
- Grab button
- Thumbstick
- Touchpad

Setup

Before you begin

You will need:

- A set of two motion controllers.
- Four AA batteries.
- A PC capable of Bluetooth 4.0.

Check for Windows, Unity, and driver updates

- Visit [Install the tools](#) for the preferred versions of Windows, Unity, etc. for mixed reality development.
- Make sure you have the most up-to-date [headset and motion controller drivers](#).

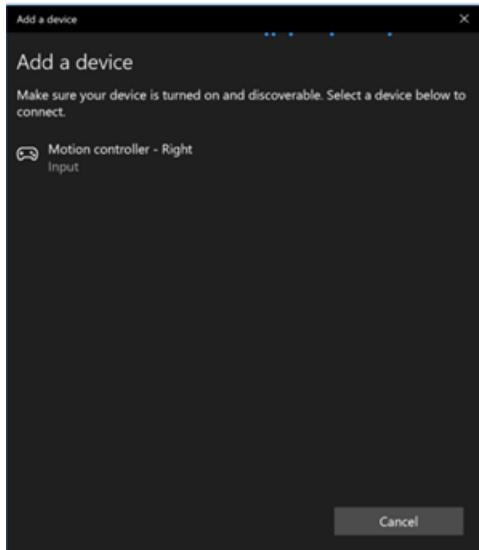
Pairing controllers

Motion controllers can be bonded with host PC using Windows settings like any other Bluetooth device.

1. Insert 2 AA batteries into the back of the controller. Leave the battery cover off for now.
2. If you're using an external USB Bluetooth Adapter instead of a built-in Bluetooth radio, please review the [Bluetooth best practices](#) before proceeding. For desktop configuration with built-in radio, please ensure antenna is connected.
3. Open **Windows Settings** -> **Devices** -> **Add Bluetooth or other device** -> **Bluetooth** and

remove any earlier instances of "Motion controller – Right" and "Motion controller – Left". Check also Other devices category at the bottom of the list.

4. Select **Add Bluetooth or other device** and see it starting to discover Bluetooth devices.
5. Press and hold the controller's Windows button to turn on the controller, release once it buzzes.
6. Press and hold the pairing button (tab in the battery compartment) until the LEDs begin pulsing.
7. Wait "Motion controller - Left" or "Motion controller - Right" to appear to the bottom of the list. Select to pair. Controller will vibrate once when connected.



Select "Motion controller" to pair; if there are multiple instances, select one from the bottom of the list

8. You will see the controller appear in the Bluetooth settings under "**Mouse, keyboard, & pen**" category as **Connected**. At this point, you may get a firmware update – see [next section](#).
9. Reattach battery cover.
10. Repeat steps 1-9 for the second controller.

After successfully pairing both controllers, your settings should look like this under "**Mouse, keyboard, & pen**" category

Mouse, keyboard, & pen

Motion controller - Left
Connected

Motion controller - Right
Connected

Motion controllers connected

If the controllers are turned off after pairing, their status will show up as Paired. If controllers stay permanently under "Other devices" category pairing may have been only partially completed and need to be performed again to get controller functional.

Updating controller firmware

- If an immersive headset is connected to your PC, and new controller firmware is available, the firmware will be pushed to your motion controllers automatically the next time they're turned on. Controller firmware updates are indicated by a pattern of illuminating LED quadrants in a circular motion, and take 1-2 minutes.

- After the firmware update completes, the controllers will reboot and reconnect. Both controllers should be connected now.

 Motion controller - Left
Connected

 Motion controller - Right
Connected

Controllers connected in Bluetooth settings

- Verify your controllers work properly:

- Launch **Mixed Reality Portal** and enter your Mixed Reality Home.
- Move your controllers and verify tracking, test buttons, and verify [teleportation](#) works. If they don't, then check out [motion controller troubleshooting](#).

Gazing and pointing

Windows Mixed Reality supports two key models for interaction, **gaze and commit** and **point and commit**:

- With **gaze and commit**, users target an object with their [gaze](#) and then select objects with hand air-taps, a gamepad, a clicker or their voice.
- With **point and commit**, a user can aim a pointing-capable motion controller at the target object and then select objects with the controller's trigger.

Apps that support pointing with motion controllers should also enable gaze-driven interactions where possible, to give users choice in what input devices they use.

Managing recoil when pointing

When using motion controllers to point and commit, your users will use the controller to target and then take action by pulling its trigger. Users who pull the trigger vigorously may end up aiming the controller higher at the end of their trigger pull than they'd intended.

To manage any such recoil that may occur when users pull the trigger, your app can snap its targeting ray when the trigger's analog axis value rises above 0.0. You can then take action using that targeting ray a few frames later once the trigger value reaches 1.0, as long as the final press occurs within a short time window. When using the higher-level [composite Tap gesture](#), Windows will manage this targeting ray capture and timeout for you.

Grip pose vs. pointing pose

Windows Mixed Reality supports motion controllers in a variety of form factors, with each controller's design differing in its relationship between the user's hand position and the natural "forward" direction that apps should use for pointing when rendering the controller.

To better represent these controllers, there are two kinds of poses you can investigate for each interaction source, the **grip pose** and the **pointer pose**.

Grip pose

The **grip pose** represents the location of either the palm of a hand detected by a HoloLens, or the palm holding a motion controller.

On immersive headsets, the grip pose is best used to render **the user's hand** or **an object held in the user's hand**, such as a sword or gun. The grip pose is also used when visualizing a motion controller, as the **renderable model** provided by Windows for a motion controller uses the grip pose as its origin and center of rotation.

The grip pose is defined specifically as follows:

- The **grip position**: The palm centroid when holding the controller naturally, adjusted left or right to center the position within the grip. On the Windows Mixed Reality motion controller, this position generally aligns with the Grasp button.
- The **grip orientation's Right axis**: When you completely open your hand to form a flat 5-finger pose, the ray that is normal to your palm (forward from left palm, backward from right palm)
- The **grip orientation's Forward axis**: When you close your hand partially (as if holding the controller), the ray that points "forward" through the tube formed by your non-thumb fingers.
- The **grip orientation's Up axis**: The Up axis implied by the Right and Forward definitions.

Pointer pose

The **pointer pose** represents the tip of the controller pointing forward.

The system-provided pointer pose is best used to raycast when you are **rendering the controller model itself**. If you are rendering some other virtual object in place of the controller, such as a virtual gun, you should point with a ray that is most natural for that virtual object, such as a ray that travels along the barrel of the app-defined gun model. Because users can see the virtual object and not the physical controller, pointing with the virtual object will likely be more natural for those using your app.

Controller tracking state

Like the headsets, the Windows Mixed Reality motion controller requires no setup of external tracking sensors. Instead, the controllers are tracked by sensors in the headset itself.

If the user moves the controllers out of the headset's field of view, in most cases Windows will continue to infer controller positions and provide them to the app. When the controller has lost visual tracking for long enough, the controller's positions will drop to approximate-accuracy positions.

At this point, the system will body-lock the controller to the user, tracking the user's position as they move around, while still exposing the controller's true orientation using its internal orientation sensors. Many apps that use controllers to point at and activate UI elements can operate normally while in approximate accuracy without the user noticing.

Reasoning about tracking state explicitly

Apps that wish to treat positions differently based on tracking state may go further and inspect properties on the controller's state, such as `SourceLossRisk` and `PositionAccuracy`:

TRACKING STATE	SOURCELOSSRISK	POSITIONACCURACY	TRYGETPOSITION
High accuracy	< 1.0	High	true
High accuracy (at risk of losing)	== 1.0	High	true
Approximate accuracy	== 1.0	Approximate	true
No position	== 1.0	Approximate	false

These motion controller tracking states are defined as follows:

- **High accuracy**: While the motion controller is within the headset's field of view, it will generally

provide high-accuracy positions, based on visual tracking. Note that a moving controller that momentarily leaves the field of view or is momentarily obscured from the headset sensors (e.g. by the user's other hand) will continue to return high-accuracy poses for a short time, based on inertial tracking of the controller itself.

- **High accuracy (at risk of losing):** When the user moves the motion controller past the edge of the headset's field of view, the headset will soon be unable to visually track the controller's position. The app knows when the controller has reached this FOV boundary by seeing the **SourceLossRisk** reach 1.0. At that point, the app may choose to pause controller gestures that require a steady stream of very high-quality poses.
- **Approximate accuracy:** When the controller has lost visual tracking for long enough, the controller's positions will drop to approximate-accuracy positions. At this point, the system will body-lock the controller to the user, tracking the user's position as they move around, while still exposing the controller's true orientation using its internal orientation sensors. Many apps that use controllers to point at and activate UI elements can operate as normal while in approximate accuracy without the user noticing. Apps with heavier input requirements may choose to sense this drop from **High** accuracy to **Approximate** accuracy by inspecting the **PositionAccuracy** property, for example to give the user a more generous hitbox on off-screen targets during this time.
- **No position:** While the controller can operate at approximate accuracy for a long time, sometimes the system knows that even a body-locked position is not meaningful at the moment. For example, a controller that was just turned on may have never been observed visually, or a user may put down a controller that's then picked up by someone else. At those times, the system will not provide any position to the app, and **TryGetPosition** will return false.

Interactions: Low-level spatial input

The core interactions across hands and motion controllers are **Select**, **Menu**, **Grasp**, **Touchpad**, **Thumbstick**, and **Home**.

- **Select** is the primary interaction to activate a hologram, consisting of a press followed by a release. For motion controllers, you perform a Select press using the controller's trigger. Other ways to perform a Select are by speaking the [voice command](#) "Select". The same select interaction can be used within any app. Think of Select as the equivalent of a mouse click, a universal action that you learn once and then apply across all your apps.
- **Menu** is the secondary interaction for acting on an object, used to pull up a context menu or take some other secondary action. With motion controllers, you can take a menu action using the controller's *menu* button. (i.e. the button with the hamburger "menu" icon on it)
- **Grasp** is how users can directly take action on objects at their hand to manipulate them. With motion controllers, you can do a grasp action by squeezing your fist tightly. A motion controller may detect a Grasp with a grab button, palm trigger or other sensor.
- **Touchpad** allows the user to adjust an action in two dimensions along the surface of a motion controller's touchpad, committing the action by clicking down on the touchpad. Touchpads provide a pressed state, touched state and normalized XY coordinates. X and Y range from -1 to 1 across the range of the circular touchpad, with a center at (0, 0). For X, -1 is on the left and 1 is on the right. For Y, -1 is on the bottom and 1 is on the top.
- **Thumbstick** allows the user to adjust an action in two dimensions by moving a motion controller's thumbstick within its circular range, committing the action by clicking down on the thumbstick. Thumbsticks also provide a pressed state and normalized XY coordinates. X and Y range from -1 to 1 across the range of the circular touchpad, with a center at (0, 0). For X, -1 is on the left and 1 is on the right. For Y, -1 is on the bottom and 1 is on the top.
- **Home** is a special system action that is used to go back to the Start Menu. It is similar to pressing the Windows key on a keyboard or the Xbox button on an Xbox controller. You can go home by pressing the Windows button on a motion controller. Note, you can also always return to Start by saying "Hey

Cortana, Go Home". Apps cannot react specifically to home actions, as these are handled by the system.

Composite gestures: High-level spatial input

Both [hand gestures](#) and motion controllers can be tracked over time to detect a common set of high-level **composite gestures**. This enables your app to detect high-level **tap, hold, manipulation** and **navigation** gestures, whether users end up using hands or controllers.

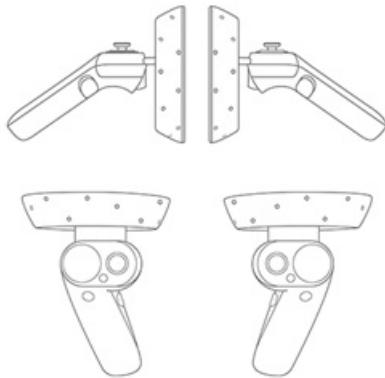
Rendering the motion controller model

3D controller models Windows makes available to apps a renderable model of each motion controller currently active in the system. By having your app dynamically load and articulate these system-provided controller models at runtime, you can ensure your app is forward-compatible to any future controller designs.

These renderable models should all be rendered at the **grip pose** of the controller, as the origin of the model is aligned with this point in the physical world. If you are rendering controller models, you may then wish to raycast into your scene from the **pointer pose**, which represents the ray along which users will naturally expect to point, given that controller's physical design.

For more information about how to load controller models dynamically in Unity, see the [Rendering the motion controller model in Unity](#) section.

2D controller line art While we recommend attaching in-app controller tips and commands to the in-app controller models themselves, some developers may want to use 2D line art representations of the motion controllers in flat "tutorial" or "how-to" UI. For those developers, we've made .png motion controller line art files available in both black and white below (right-click to save).



[Full-resolution motion controllers line art in "white"](#)

[Full-resolution motion controllers line art in "black"](#)

FAQ

Can I pair motion controllers to multiple PCs?

Motion controllers support pairing with a single PC. Follow instructions on [motion controller setup](#) to pair your controllers.

How do I update motion controller firmware?

Motion controller firmware is part of the headset driver and will be updated automatically on connection if required. Firmware updates typically take 1-2 minutes depending on Bluetooth radio and link quality. In rare cases, controller firmware updates may take up to 10 minutes, which can indicate poor Bluetooth

connectivity or radio interference. Please see [Bluetooth best practices in the Enthusiast Guide](#) to troubleshoot connectivity issues. After a firmware update, controllers will reboot and reconnect to the host PC (you may notice the LEDs go bright for tracking). If a firmware update is interrupted (for example, the controllers lose power), it will be attempted again the next time the controllers are powered on.

How I can check battery level?

In the [Windows Mixed Reality home](#), you can turn your controller over to see its battery level on the reverse side of the virtual model. There is no physical battery level indicator.

Can you use these controllers without a headset? Just for the joystick/trigger/etc input?

Not for Universal Windows Applications.

Troubleshooting

See [motion controller troubleshooting](#) in the Enthusiast Guide.

Filing motion controller feedback/bugs

[Give us feedback](#) in Feedback Hub, using the "Mixed Reality -> Input" category.

See also

- [Gestures and motion controllers in Unity](#)
- [Gaze, gestures, and motion controllers in DirectX](#)
- [Gestures](#)
- [MR Input 213: Motion controllers](#)
- [Enthusiast's Guide: Your Windows Mixed Reality home](#)
- [Enthusiast's Guide: Using games & apps in Windows Mixed Reality](#)
- [How inside-out tracking works](#)

Color, light and materials

11/6/2018 • 4 minutes to read • [Edit Online](#)

Designing content for mixed reality requires careful consideration of color, lighting, and materials for each of the visual assets used in your experience. These decisions can be for both aesthetic purposes, like using light and material to set the tone of an immersive environment, and functional purposes, like using striking colors to alert users of an impending action. Each of these decisions must be weighed against the opportunities and constraints of your experience's target device.

Below are guidelines specific to rendering assets on both immersive and holographic headsets. Many of these are closely tied to other technical areas and a list of related subjects can be found in the [See also](#) section at the end of this article.

Rendering on immersive vs. holographic devices

Content rendered in immersive headsets will appear visually different when compared to content rendered in holographic headsets. While immersive headsets generally render content much as you would expect on a 2D screen, holographic headsets like HoloLens use color-sequential, see-through RGB displays to render holograms.

Always take time to test your holographic experiences in a holographic headset. The appearance of content, even if it is built specifically for holographic devices, will differ as seen on secondary monitors, snapshots, and in spectator view. Remember to walk around experiences with a device, testing the lighting of holograms and observing from all sides (as well as from above and below) how your content is rendered. Be sure to test at a range of brightness settings on the device, as it is unlikely all users will share an assumed default, as well as a diverse set of lighting conditions.

Fundamentals of rendering on holographic devices

- **Holographic devices have additive displays** – Holograms are created by adding light to the light from the real world – white will appear brightly, while black will appear transparent.
- **Colors impact varies with the user's environment** – There are many diverse lighting conditions in a user's room. Create content with appropriate levels of contrast to help with clarity.
- **Avoid dynamic lighting** – Holograms that are uniformly lit in holographic experiences are the most efficient. Using advanced, dynamic lighting will likely exceed the capabilities of mobile shaders.

Designing with color

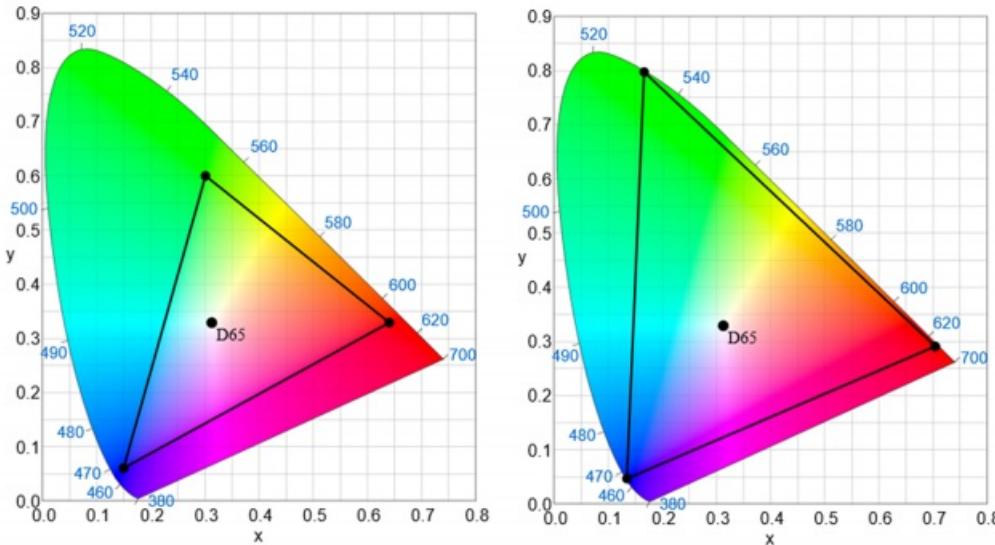
Due to the nature of additive displays, certain colors can appear different on holographic displays. Some colors will pop in lighting environments while others will appear as less impactful. Cool colors tend to recede into the background while warm colors jump to the foreground. Consider these factors as you explore color in your experiences:

- **Gamut** - HoloLens benefits from a "wide gamut" of color, conceptually similar to Adobe RGB. As a result, some colors can exhibit different qualities and representation in the device.
- **Gamma** - The brightness and contrast of the rendered image will vary between immersive and holographic devices. These device differences often appear to make dark areas of color and shadows, more or less bright.
- **Color separation** - Also called "color breakup" or "color fringing", color separation most commonly occurs with moving holograms (including cursor) when a user tracks objects with their eyes.
- **Color uniformity** - Typically holograms are rendered brightly enough so that they maintain color uniformity, regardless of the background. Large areas may become blotchy. Avoid large regions of bright, solid color.

- **Rendering light colors** - White appears very bright and should be used sparingly. For most cases, consider a white value around R 235 G 235 B 235. Large bright areas may cause user discomfort.

Rendering dark colors

Due to the nature of additive displays, dark colors appear transparent. A solid black object will appear no different from the real world. See Alpha channel below. To give the appearance of "black" try a very dark grey RGB value such as 16,16,16.



Normal vs. wide color gamut

Technical considerations

- **Aliasing** - Be considerate of aliasing, jagged or "stair steps" where the edge of a hologram's geometry meets the real world. Using textures with high detail can aggravate this effect. Textures should be mapped and filtering enabled. Consider fading the edges of holograms or adding a texture that creates a black edge border around objects. Avoid thin geometry where possible.
- **Alpha channel** - You must clear your alpha channel to fully transparent for any parts where you are not rendering a hologram. Leaving the alpha undefined leads to visual artifacts when taking images/videos from the device or with Spectator View.
- **Texture softening** - Since light is additive in holographic displays, it is best to avoid large regions of bright, solid color as they often do not produce the intended visual effect.

Storytelling with light and color

Light and color can help make your holograms appear more naturally in a user's environment as well as offer guidance and help for the user. For holographic experiences, consider these factors as you explore lighting and color:

- **Vignetting** - A 'vignette' effect to darken materials can help focus the user's attention on the center of the field of view. This effect darkens the hologram's material at some radius from the user's gaze vector. Note that this is also effective when the user's views holograms from an oblique or glancing angle.
- **Emphasis** - Draw attention to objects or points of interaction by contrasting colors, brightness, and lighting. For a more detailed look at lighting methods in storytelling, see [Pixel Cinematography - A Lighting Approach for Computer Graphics](#).



Use of color to show emphasis for storytelling elements, shown here in a scene from [Fragments](#).

See also

- [Color Separation](#)
- [Holograms](#)
- [Microsoft Design Language - color](#)
- [Universal Windows Platform - color](#)

Spatial sound design

11/6/2018 • 6 minutes to read • [Edit Online](#)

Spatial sound is a powerful tool for immersion, accessibility, and UX design in mixed reality applications.

If you've ever played [Marco Polo](#), or had someone call your phone to help you locate it, you are already familiar with the importance of spatial sound. We use sound cues in our daily lives to locate objects, get someone's attention, or get a better understanding of our environment. The more closely your app's sound behaves like it does in the real world, the more convincing and engaging your virtual world will be.

Device support

FEATURE	HOLOLENS	IMMERSIVE HEADSETS
Spatial sound	✓ <input type="checkbox"/>	✓ <input type="checkbox"/>

Four key things spatial sound does for mixed reality development

By default, sounds are played back in stereo. This means the sound will play with no spatial position, so the user does not know where the sound came from. Spatial sound does four key things for mixed reality development:

Grounding

Without sound, virtual objects effectively cease to exist when we turn our head away from them. Just like real objects, you want to be able to hear these objects even when you can't see them, and you want to be able to locate them anywhere around you. Just as virtual objects need to be grounded visually to blend with your real world, they also need to be grounded audibly. Spatial sound seamlessly blends your real world audio environment with the digital audio environment.

User attention

In mixed reality experiences, you can't assume where your user is looking and expect them to see something you place in the world visually. But users can always hear a sound play even when the object playing the sound is behind them. People are used to having their attention drawn by sound - we instinctually look toward an object that we hear around us. When you want to direct your user's gaze to a particular place, rather than using an arrow to point them visually, placing a sound in that location is a very natural and fast way to guide them.

Immersion

When objects move or collide, we usually hear those interactions between materials. So when your objects don't make the same sound they would in the real world, a level of immersion is lost - like watching a scary movie with the volume all the way down. All sounds in the real world come from a particular point in space - when we turn our heads, we hear the change in where those sounds are coming from relative to our ears, and we can track the location of any sound this way. Spatialized sounds make up the "feel" of a place beyond what we can see.

Interaction design

In most traditional interactive experiences, interaction sounds like UI sound effects are played in standard mono

or stereo. But because everything in mixed reality exists in 3D space - including the UI - these objects benefit from spatialized sounds. When we press a button in the real world, the sound we hear comes from that button. By spatializing interaction sounds, we again provide a more natural and realistic user experience.

Best practices when using spatial sound

Real sounds work better than synthesized or unnatural sounds

The more familiar your user is with a type of sound, the more real it will feel, and the more easily they will be able to locate it in their environment. A human voice, for example, is a very common type of sound, and your users will locate it just as quickly as a real person in the room talking to them.

Expectation trumps simulation

If you are used to a sound coming from a particular direction, your attention will be guided in that direction regardless of spatial cues. For example, most of the time that we hear birds, they are above us. Playing the sound of a bird will most likely cause your user to look up, even if you place the sound below them. This is usually confusing, and it is recommended that you work with expectations like these rather than going against them for a more natural experience.

Most sounds should be spatialized

As mentioned above, everything in Mixed Reality exists in 3D space - your sounds should as well. Even music can sometimes benefit from spatialization, particularly when it's tied to a menu or some other UI.

Avoid invisible emitters

Because we've been conditioned to look at sounds that we hear around us, it can be an unnatural and even unnerving experience to locate a sound that has no visual presence. Sounds in the real world don't come from empty space, so be sure that if an audio emitter is placed within the user's immediate environment that it can also be seen.

Avoid spatial masking

Spatial sound relies on very subtle acoustic cues that can be overpowered by other sounds. If you do have stereo music or ambient sounds, make sure they are low enough in the mix to give room for the details of your spatialized sounds that will allow your users to locate them easily, and keep them sounding realistic and natural.

General concepts to keep in mind when using spatial sound

Spatial sound is a simulation

The most frequent use of spatial sound is making a sound seem as though it is emanating from a real or virtual object in the world. Thus, spatialized sounds may make the most sense coming from such objects.

Note that the perceived accuracy of spatial sound means that a sound shouldn't necessarily emit from the center of an object, as the difference will be noticeable depending on the size of the object and distance from the user. With small objects, the center point of the object is usually sufficient. For larger objects, you may want a sound emitter or multiple emitters at the specific location within the object that is supposed to be producing the sound.

Normalize all sounds

Distance attenuation happens quickly within the first meter from the user, as it does in the real world. All audio files should be normalized to ensure physically accurate distance attenuation, and ensure that a sound can be heard when several meters away (when applicable). The spatial audio engine will handle the attenuation necessary for a sound to "feel" like it's at a certain distance (with a combination of attenuation and "distance cues"), and applying any attenuation on top of that could reduce the effect. Outside of simulating a real object, the initial distance decay of *spatial sound* sounds will likely be more than enough for a proper mix of your audio.

Object discovery and user interfaces

When using audio cues to direct the user's attention beyond their current view, the sound should be audible and prominent in the mix, well above any stereo sounds, and any other spatialized sounds which might distract from the directional audio cue. For sounds and music that are associated with an element of the user interface (e.g. a menu), the sound emitter should be attached to that object. Stereo and other non-positional audio playing can make spatialized elements difficult for users to locate (See above: Avoid spatial masking).

Use spatial sound over standard 3D sound as much as possible

In mixed reality, for the best user experience, 3D audio should be achieved using spatial sound rather than legacy 3D audio technologies. In general, the improved spatialization is worth the small CPU cost over standard 3D sound. Standard 3D audio can be used for low-priority sounds, sounds that are spatialized but not necessarily tied to a physical or virtual object, and objects that the user never need locate to interact with the app.

See also

- [Spatial sound](#)
- [Spatial mapping](#)

Typography

11/6/2018 • 3 minutes to read • [Edit Online](#)

Text is an important element for delivering information in your app experience. Just like typography on 2D screens, the goal is to be clear and readable. With the three-dimensional aspect of mixed reality, there is an opportunity to affect the text and the overall user experience in an even greater way.



Typography example in HoloLens

When we talk about type in 3D, we tend to think extruded, volumetric 3D text. Except for some logotype designs and a few other limited applications, extruded text tends to degrade the readability of the text. Even though we are designing experiences for 3D, we use 2D for the type because it is more legible and easier to read.

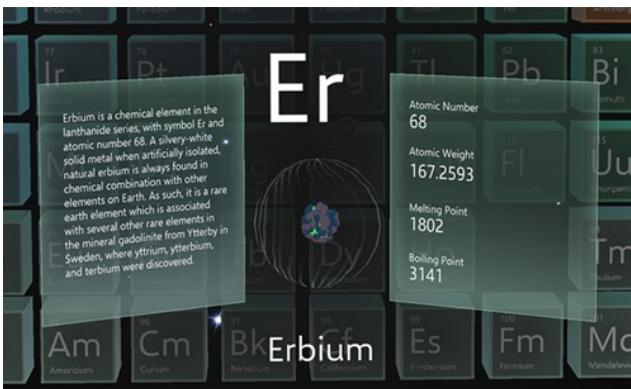
In HoloLens, type is constructed with holograms using light based on the additive color system. Just like other holograms, type can be placed in the actual environment where it can be world locked and observed from any angle. The [parallax](#) effect between the type and the environment also adds depth to the experience.

Typography in mixed reality

Typographic rules in mixed reality are no different from anywhere else. Text in both the physical world and the virtual world needs to be legible and readable. Text could be on a wall or superimposed on a physical object. It could be floating along with a digital user interface. Regardless of the context, we apply the same typographic rules for reading and recognition.

Create clear hierarchy

Build contrast and hierarchy by using different type sizes and weights. Defining a type ramp and following it throughout the app experience will provide a great user experience with consistent information hierarchy.



Symbol Element Name Data Numbers

Data Label



City Name Data Numbers

Data Label

Type ramp examples

Limit your fonts

Avoid using more than two different font families in a single context. This will break the harmony and consistency of your experience and make it harder to consume information. In HoloLens, since the information is overlaid on top of the physical environment, using too many font styles will degrade the experience. Segoe UI is the font for all Microsoft digital designs. It is used consistently in the Windows Mixed Reality shell. You can download the Segoe UI font file from the [Windows design toolkit page](#).

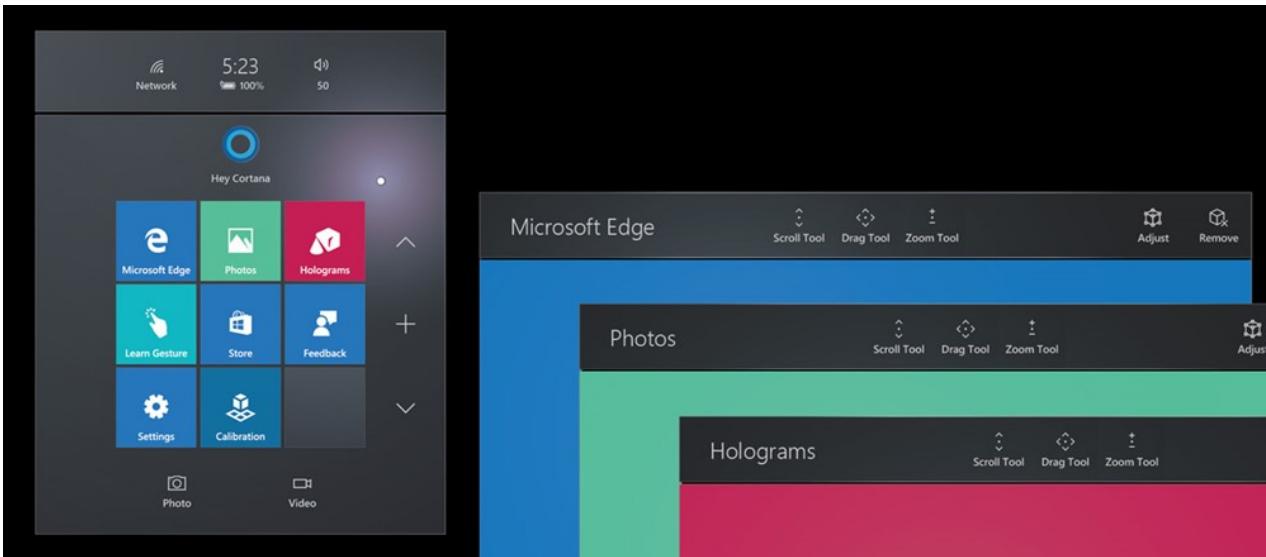
[More information about the Segoe UI typeface](#)

Avoid thin font weights

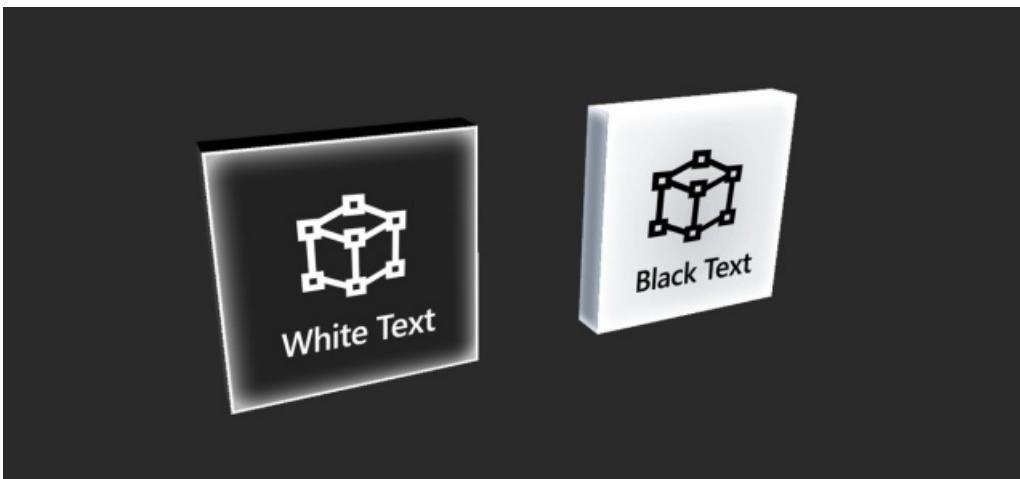
Avoid using light or semilight font weights for type sizes under 42pt since thin vertical strokes will vibrate and degrade legibility. Modern fonts with enough stroke thickness work well. For example, Helvetica and Arial are very legible in HoloLens using regular or bold weights.

Color

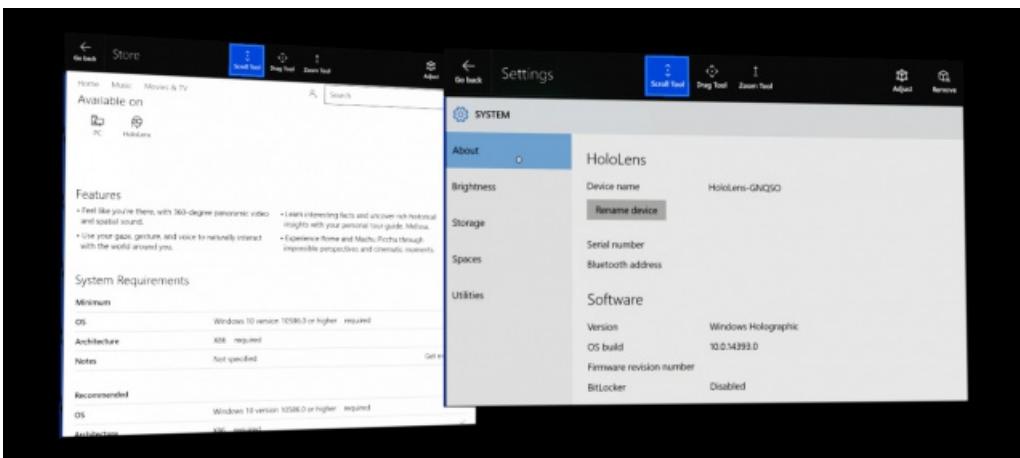
In HoloLens, since the holograms are constructed with an additive light system, white text is highly legible. You can find examples of white text on the Start menu and the App bar. Even though white text works well without a back plate on HoloLens, a complex physical background could make the type difficult to read. To improve the user's focus and minimize the distraction from a physical background, we recommend using white text on a dark or colored back plate.



We recommend using white text on a dark or colored back plate.

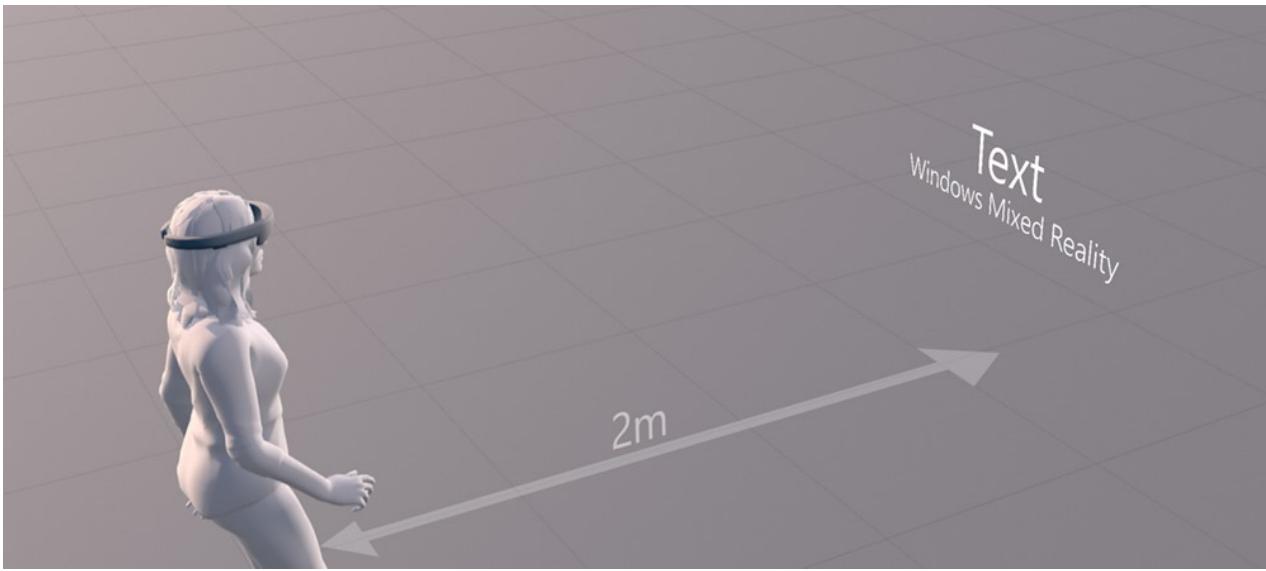


To use dark text, you should use a bright back plate to make it readable. In additive color systems, black is displayed as transparent. This means you will not be able to see the black text without a colored back plate.



You can find examples of black text in UWP apps such as the Store or Settings.

Recommended font size



Two meters is the optimal distance for displaying text.

Since mixed reality involves three-dimensional depth, it is not always easy to communicate the size of the font. For the user's comfort, two meters is the optimal distance for placing holograms. We can use this distance as a basis to find the optimal font size.

As you can expect, type sizes that we use on a PC or a tablet device (typically between 12–32pt) look quite small at a distance of 2 meters. It depends on the characteristics of each font, but in general, the recommended minimum type size for legibility without stroke vibration is around 30pt. If your app is supposed to be used at a closer distance, smaller type sizes could be used. **The point size is based on the Unity's 3D Text Mesh and UI Text. For the detailed metrics and scaling factors, please refer to [Text in Unity](#).**

Resources

- [Segoe fonts](#)
- [HoloLens font](#)



The HoloLens font gives you the symbol glyphs used in Windows Mixed Reality.

See also

- [Text in Unity](#)
- [Color, light and materials](#)

Scale

11/6/2018 • 4 minutes to read • [Edit Online](#)

A key to displaying content that looks realistic in holographic form is to mimic the visual statistics of the real world as closely as possible. This means incorporating as many of the visual cues as we can that help us (in the real world) understand where objects are, how big they are, and what they're made of. The scale of an object is one of the most important of those visual cues, giving a viewer a sense of the size of an object as well as cues to its location (especially for objects which have a known size). Further, viewing objects at real scale has been seen as one of the key experience differentiators for mixed reality in general – something that hasn't been possible on screen-based viewing previously.

How to suggest the scale of objects and environments

There are many ways to suggest the scale of an object, some of which have possible effects on other perceptual factors. The key one is to simply display objects at a 'real' size, and maintain that realistic size as users move. This means holograms will take up a different amount of a user's visual angle of a user as they come closer or further away, the same way that real objects do.

Utilize the distance of objects as they are presented to the user

One common method is to utilize the distance of objects as they are presented to the user. For example, consider visualizing a large family car in front of the user. If the car were directly in front of them, within arm's length, it would be too large to fit in the user's field of view. This would require the user to move their head and body to understand the entirety of the object. If the car were placed further away (across the room), the user can establish a sense of scale by seeing the entirety of the object in their field of view, then moving themselves closer to it to inspect areas in detail.

[Volvo](#) used this technique to create a showroom experience for a new car, utilizing the scale of the holographic car in a way that felt realistic and intuitive to the user. The experience begins with a hologram of the car on a physical table, allowing the user to understand the total size and shape of the model. Later in the experience, the car expands to a larger scale (beyond the size of the device's field of view) but, since the user already acquired a frame of reference from the smaller model, they can adequately navigate around features of the car.



Volvo Cars experience for HoloLens

Use holograms to modify the user's real space

Another method is to use holograms to modify the user's real space, replacing the existing walls or ceilings with environments or appending 'holes' or 'windows', allowing over-sized objects to seemingly 'break-through' the physical space. For example, a large tree might not fit in most users' living rooms, but by putting a virtual sky on their ceiling, the physical space expands into the virtual. This allows the user to walk around the base of the virtual tree, and gather a sense of scale of how it would appear in real life, then look up to see it extend far beyond the physical space of the room.

[Minecraft](#) developed a concept experience using a similar technique. By adding a virtual window to a physical surface in a room, the existing objects in the room are placed in the context of a vastly larger environment, beyond the physical scale limitations of the room.



Minecraft concept experience for HoloLens

Experimenting with scale

In some cases, designers have experimented with modifying the scale (by changing the displayed 'real' size of the object) while maintaining a single position of the object, to approximate an object getting closer or further to a viewer without any actual movement. This was tested in some cases as a way to simulate up-close viewing of items while still respecting potential comfort limitations of viewing virtual content closer than the "zone of comfort" would suggest.

This can create a few possible artifacts in the experience however:

- For virtual objects that represent some object with a 'known' size to the viewer, changing the scale without changing the position leads to conflicting visual cues – the eyes may still 'see' the object at some depth due to vergence cues (see the [Comfort](#) article for more on this), but the size acts as a monocular cue that the object might be getting closer. These conflicting cues lead to confused perceptions – viewers often see the object as staying in place (due to the constant depth cue) but growing rapidly.
- In some cases, change of scale is seen as a 'looming' cue instead, where the object may or may not be seen to change scale by a viewer, but does appear to be moving directly toward the viewer's eyes (which can be an uncomfortable sensation).
- With comparison surfaces in the real world, such scaling changes are sometimes seen as changing position along multiple axes – objects may appear to drop lower instead of moving closer (similar in a 2D projection of 3D movement in some cases).
- Finally, for objects without a known 'real world' size (e.g. arbitrary shapes with arbitrary sizes, UI elements, etc.), changing scale may act functionally as a way to mimic changes in distance – viewers do not have as many preexisting top-down cues by which to understand the object's true size or location, so the scale can be processed as a more important cue.

See also

- [Color, light and materials](#)
- [Typography](#)
- [Spatial sound design](#)

Case study - Spatial sound design for HoloTour

11/6/2018 • 7 minutes to read • [Edit Online](#)

To create a truly immersive 3D virtual tour for Microsoft HoloLens, the panoramic videos and holographic scenery are only part of the formula. Audio designer Jason Syltebo talks about how sound was captured and processed to make you feel like you're actually in each of the locations in HoloTour.

The tech

The beautiful imagery and holographic scenes you see in HoloTour are only one part of creating a believable mixed reality experience. While holograms can only appear visually in front of a user, the [spatial sound](#) feature of HoloLens allows audio to come from all directions, which gives the user a more complete sensory experience.

Spatial sound allows us to give audio cues to indicate a direction in which the user should turn, or to let the user know there are more holograms for them to see within their space. We can also attach a sound directly to a hologram and continually update the direction and distance the hologram is from a user to make it seem as if the sound is coming directly from that object.

With HoloTour, we wanted to take advantage of the spatial sound capabilities of HoloLens to create a 360-degree ambient environment, synchronized with the video to reveal the sonic highlights of specific locations.

Behind the scenes

We created HoloTour experiences of two different locations: Rome and Machu Picchu. To make these tours feel authentic and compelling we wanted to avoid using generic sounds and instead capture audio directly from the locations where we were filming.

Capturing the audio

In our [case study about capturing the visual content for HoloTour](#), we talked about the custom design of our camera rig. It consisted of 14 GoPro cameras contained in a 3D-printed housing, designed to the specific dimensions of the tripod. To capture audio from this rig, we added a quad-microphone array beneath the cameras, which fed into a compact 4-channel recording unit that sat at the base of the tripod. We chose microphones that not only performed well, but which had a very small footprint, so as to not occlude the view of the camera.



Custom camera and microphone rig

This setup captured sound in four directions from the precise location of our camera, giving us enough information to re-create a 3D aural panorama using spatial sound, which we could later synchronize to the 360-degree video.

One of the challenges with the camera array audio is that you are at the mercy of what is recorded at the time of capture. Even if the video capture is good, the sound capture can become problematic due to off-camera sounds such as sirens, airplanes, or high winds. To assure we had all the elements we need, we used a series of stereo and mono mobile recording units to get asynchronous, ambient elements at specific points of interest in each location. This capture is important as it gives the sound designer the ability to seek out clean and usable content that can be used in post-production to craft interest and add further directionality.

Any given capture day would generate a large number of files. It was important to develop a system to track which files correspond to a particular location or camera shot. Our recording unit was set up to auto-name files by date and take number and we would back these up at the end of the day to external drives. At the very least, verbally slating the beginning of audio recordings was important as this allows easy contextual identification of the content should filenames become a problem. It was also important for us to visually slate the camera rig capture as the video and audio were recorded as separate media and needed to be synchronized during post.

Editing the audio

Back at the studio after the capture trip, the first step in assembling a directional and immersive aural experience is to review all of the audio capture for a location, picking out the best takes and identifying any highlights that could be applied creatively during integration. The audio is then edited and cleaned up. For example, a loud car horn lasting a second or so and repeating a few times, can be replaced and stitched in with sections of quiet, ambient audio from the same capture.

Once the video edit for a location has been established the sound designer can synchronize the corresponding audio. At this point we worked with both camera rig capture and mobile capture to decide what elements, or combination thereof, will work to build an immersive audio scene. A technique we found useful was to add all the sound elements into an audio editor and build quick linear mock ups to experiment with different mix ideas. This gave us better formed ideas when it came time to build the actual HoloTour scenes.

Assembling the scene

The first step to building a 3D ambient scene is to create a bed of general background ambient looping sounds that will support other features and interactive sound elements in a scene. In doing so, we took a holistic approach towards different implementation techniques determined by the specific needs and design criteria of any particular scene. Some scenes might index towards using the synchronized camera capture, whereas others might require a more curated approach that uses more discretely placed sounds, interactive elements and music and sound effects for the more cinematic moments in HoloTour.

When indexing on the use of the camera capture audio, we placed spatial sound-enabled ambient audio emitters corresponding to the directional coordinates of the camera orientation such that the north camera view plays audio from the north microphone and likewise for the other cardinal directions. These emitters are world-locked, meaning the user can freely turn their head in relation to these emitters and the sound will change accordingly, effectively modeling the sound of standing at that location. Listen to Piazza Navona or The Pantheon for examples of scenes that use a good mix of camera captured audio.

A different approach involved playing a looping stereo ambience in conjunction with spatial sound emitters placed around the scene playing one-off sounds that are randomized in terms of volume, pitch and trigger frequency. This creates an ambience with an enhanced sense of directionality. In Aguas Calientes, for example, you can listen to how each quadrant of the panorama has specific emitters that intentionally highlight specific areas of the geography, but work together to create an overall immersive ambience.

Tips and tricks

When you're putting together audio for a scene, there are some additional methods you can use to further highlight directionality and immersion, making full use of the spatial sound capabilities of HoloLens. We've provided a list of some below—listen for them the next time you try HoloTour.

- **Look Targets:** These are sounds that trigger only when you are looking at a specific object or area of the holographic frame. For example, looking in the direction of the street-side café in Rome's Piazza Navona will subtly trigger the sounds of a busy restaurant.
- **Local Vision:** The journey through HoloTour contains certain beats where your tour guide, aided by holograms, will explore a topic in-depth. For instance, as the façade of the Pantheon dissolves to reveal the oculus, reverberating audio placed as a 3D emitter from the inside of the Pantheon encourages the user to explore the interior model.
- **Enhanced directionality:** Within many scenes, we placed sounds in various ways to add to the directionality. In the Pantheon scene, for example, the sound of the fountain was placed as a separate emitter close enough to the user so that they could get a sense of 'sonic parallax' as they walked around the play space. In Peru's Salinas de Maras scene, the individual perspective of some of the little streams were placed as separate emitters to build a more immersive ambient environment, surrounding the user with the authentic sounds of that location.
- **Spline emitter:** This special spatial sound emitter moves in 3D space relative to the visual position of the object it's attached to. An example of this was the train in Machu Picchu, where we used a spline emitter to give a distinct sense of directionality and movement.
- **Music and SFX:** Certain aspects of HoloTour that represent a more stylized or cinematic approach use music and sound effects to heighten the emotional impact. In the gladiator battle at the end of the Rome tour, special effects like whooshes or stingers were used to help strengthen the effect of labels appearing in scenes.

About the author



Jason Syltebo

Audio Designer @Microsoft

See also

- [Spatial sound](#)

- Spatial sound design
- Spatial sound in Unity
- MR Spatial 220
- Video: Microsoft HoloLens: HoloTour

Types of mixed reality apps

11/6/2018 • 3 minutes to read • [Edit Online](#)

One of the advantages of developing apps for Windows Mixed Reality is that there is a spectrum of experiences that the platform can support. From fully immersive, virtual environments, to light information layering over a user's current environment, Windows Mixed Reality provides a robust set of tools to bring any experience to life. It is important for an app maker to understand early in their development process as to where along this spectrum their experience lies. This decision will ultimately impact both the app design makeup and the technological path for development.

Enhanced environment apps (HoloLens only)

One of the most powerful ways that mixed reality can bring value to users is by facilitating the placement of digital information or content in a user's current environment. This is an enhanced environment app. This approach is popular for apps where the contextual placement of digital content in the real world is paramount and/or keeping the user's real world environment "present" during their experience is key. This approach also allows users to easily move from real world tasks to digital tasks and back easily, lending even more credence to promise that the mixed reality apps the user sees are truly a part of their environment.



Enhanced environment apps

Example uses

- A mixed reality notepad style app that allows users to create and place notes around their environment
- A mixed reality television app placed in a comfortable spot for viewing
- A mixed reality cooking app placed above the kitchen island to assist in a cooking task
- A mixed reality app that gives users the feeling of "x-ray vision" (i.e. a hologram placed on top of and mimics a real world object, while allowing the user to see "inside it" holographically)
- Mixed reality annotations placed throughout a factory to give worker's necessary information
- Mixed reality wayfinding in an office space
- Mixed reality tabletop experiences (i.e. board game style experiences)
- Mixed reality communication apps like Skype

Blended environment apps

Given Windows Mixed Reality's ability to recognize and map the user's environment, it is capable of creating a digital layer that can be completely overlaid on the user's space. This layer respects the shape and boundaries of the user's environment, but the app may choose to transform certain elements best suited to immerse the user in the app. This is called a blended environment app. Unlike an enhanced environment app, blended environment apps may only care enough about the environment to best use its makeup for encouraging specific user behavior (like encouraging movement or exploration) or by replacing elements with changes (a kitchen counter is virtually skinned to show a different tile pattern). This type of experience may even transform an element into an entirely different object, but still retain the rough dimensions of the object as its base (a kitchen island is transformed into a dumpster for a crime thriller game).



Blended environment apps

Example uses

- A mixed reality interior design app that can paint walls, countertops or floors in different colors and patterns
- A mixed reality app that allows an automotive designer to layer new design iterations for an upcoming car refresh on top of an existing car
- A bed is "covered" and replaced by a mixed reality fruit stand in children's game
- A desk is "covered" and replaced with a mixed reality dumpster in a crime thriller game
- A hanging lantern is "covered" and replaced with signpost using roughly the same shape and dimension
- An app that allows users to blast holes in their real or immersive world walls, which reveal a magical world

Immersive environment apps

Immersive environment apps are centered around an environment that completely changes the user's world and can place them in a different time and space. These environments can feel very real, creating immersive and thrilling experiences that are only limited by the app creator's imagination. Unlike blended environment apps, once Windows Mixed Reality identifies the user's space, an immersive environment app may totally disregard the user's current environment and replace it whole stock with one of its own. These experiences may also completely separate time and space, meaning a user could walk the streets of Rome in an immersive experience, while remaining relatively still in their real world space. Context of the real world environment may not be important to an immersive environment app.



Immersive environment apps

Example uses

- An immersive app that lets a user tour a space completely separate from their own (i.e. walk through a famous building, museum, popular city)
- An immersive app that orchestrates an event or scenario around the user (i.e. a battle or a performance)

See also

- [Development overview](#)
- [App model](#)
- [App views](#)

Room scan visualization

11/6/2018 • 3 minutes to read • [Edit Online](#)

Applications that require spatial mapping data rely on the device to automatically collect this data over time and across sessions as the user explores their environment with the device active. The completeness and quality of this data depends on a number of factors including the amount of exploration the user has done, how much time has passed since the exploration and whether objects such as furniture and doors have moved since the device scanned the area.

To ensure useful spatial mapping data, applications developers have several options:

- Rely on what may have already been collected. This data may be incomplete initially.
- Ask the user to use the bloom gesture to get to the Windows Mixed Reality home and then explore the area they wish to use for the experience. They can use air-tap to confirm that all the necessary area is known to the device.
- Build a custom exploration experience in their own application.

Note that in all these cases the actual data gathered during the exploration is stored by the system and the application does not need to do this.

Device support

FEATURE	HOLOLENS	IMMERSIVE HEADSETS
Room scan visualization	✓ <input type="checkbox"/>	

Building a custom scanning experience

Applications may decide to analyze the spatial mapping data at the start of the experience to judge whether they want the user to perform additional steps to improve its completeness and quality. If analysis indicates quality should be improved, developers should provide a visualization to overlay on the world to indicate:

- How much of the total volume in the users vicinity needs to be part of the experience
- Where the user should go to improve data

Users do not know what makes a "good" scan. They need to be shown or told what to look for if they're asked to evaluate a scan – flatness, distance from actual walls, etc. The developer should implement a feedback loop that includes refreshing the spatial mapping data during the scanning or exploration phase.

In many cases, it may be best to tell the user what they need to do (e.g. look at the ceiling, look behind furniture), in order to get the necessary scan quality.

Cached versus continuous spatial mapping

The spatial mapping data is the most heavy weight data source applications can consume. To avoid performance issues such as dropped frames or stuttering, consumption of this data should be done carefully.

Active scanning during an experience can be both beneficial or detrimental and the developer will need to decide which method to use based on the experience.

Cached spatial mapping

In the case of cached spatial mapping, the application typically takes a snapshot of the spatial mapping data and uses this snapshot for the duration of the experience.

Benefits

- Reduced overhead on the system while the experience is running leading to dramatic power, thermal and cpu performance gains.
- A simpler implementation of the main experience since it is not interrupted by changes in the spatial data.
- A single one time cost on any post processing of the spatial data for physics, graphics and other purposes.

Drawbacks

- The movement of real world objects or people is not reflected by the cached data. E.g. the application might consider a door open when it is actually closed now.
- Potentially more application memory to maintain the cached version of the data.

A good case for this method is a controlled environment or a table top game.

Continuous spatial mapping

Certain applications may rely on continues scanning to refresh spatial mapping data.

Benefits

- You don't need to build in a separate scanning or exploration experience upfront in your application.
- The movement of real world objects can be reflected by the game, although with some delay.

Drawbacks

- Higher complexity in the implementation of the main experience.
- Potential overhead of the additional processing for graphic or physics as changes need to be incrementally ingested by these systems.
- Higher power, thermal and CPU impact.

A good case for this method is one where holograms are expected to interact with moving objects, e.g. a holographic car that drives on the floor may want to correctly bump into a door depending on whether it is open or closed.

See also

- [Spatial mapping design](#)
- [Coordinate systems](#)
- [Spatial sound design](#)

Cursors

11/6/2018 • 6 minutes to read • [Edit Online](#)

A cursor, or indicator of your current [gaze](#) vector, provides continuous feedback for the user to understand what HoloLens understands about their intentions.

Targeting of content is done primarily with the [gaze](#) vector (a ray controlled by the position and rotation of the head). This provides a form of direct input for the user that needs little teaching. However, users have difficulty using an unmarked center of gaze for precise targeting. The cursor allows the user to understand their more-precise center of gaze and acts as feedback to indicate what area, hologram, or point will respond to input. It is the digital representation of where the device understands the user's attention to be (though that may not be the same as determining anything about their intentions).

The feedback provided by the gaze indicator offers users the ability to anticipate how the system will respond, use that signal as feedback to better communicate their intention to the device, and ultimately be more confident about their interactions.

Positioning

In general, the indicator should move in approximately a 1:1 ratio with head movement. There are some cases where gain (augmenting or diminishing movement noticeably) might be used as an intentional mechanic, but it will cause trouble for users if used unexpectedly (note that there is a tiny bit of 'lag' recommended for the cursor to avoid issues with fully display-locked content). Most importantly though, experiences should be "honest" in the representation of cursor position - if smoothing, magnetism, gain, or other effects are included, the system should still show the cursor wherever the system's understanding of position is, with those effects included. The cursor is the system's way of telling the user what they can or can't interact with, not the user's way of telling the system.

The indicator should ideally lock in depth to whatever elements the user can plausibly target. This may mean surface-locking if there is some [Spatial Mapping](#) mesh or locking to the depth of any "floating" UI elements, to help the user understand what they can interact with in real-time.

Cursor design principles

Always present

- We recommend that the cursor is always present.
- If the user can't find the cursor, then they're lost.
- Exceptions to this are instances where having a cursor provides a suboptimal experience for the user. An example is when a user is watching a video. The cursor becomes undesirable at this point as it's in the middle of the video all the time. This is a scenario where you may consider making the cursor only visible when the user has their hand up indicating a desire to take action. Otherwise, it's not visible on the video.

Cursor scale

- The cursor should be no larger than the available targets, allowing users to easily interact with and view the content.
- Depending on the experience you create, scaling the cursor as the user looks around is also an important consideration. For example, as the user looks further away in your experience perhaps the cursor should not become too small such that its lost.
- When scaling the cursor, consider applying a soft animation to it as it scales giving it an organic feeling.

- Avoid obstructing content. Holograms are what make the experience memorable and the cursor should not be taking away from them.

Directionless cursor

- Although there is no one right cursor shape, we recommend that you use a directionless shape like a torus or something else. A cursor that points in some direction (ex: a traditional arrow cursor) might confuse the user to always look that way.
- An exception to this is when using the cursor to communicate interaction instruction to the user. For example, when scaling holograms in the Windows Holographic shell, the cursor temporarily includes arrows that help instruct the user on how to move their hand to scale the hologram.

Look and feel

- A donut or torus shaped cursor works for a lot of applications.
- Pick a color and shape that best represents the experience you are creating.
- Cursors are especially prone to [color separation](#).
- A small cursor with balanced opacity keeps it informative without dominating the visual hierarchy.
- Be cognizant of using shadows or highlights behind your cursor as they might obstruct content and distract from the purpose.
- Cursors should align to and hug the surfaces in your app, this will give the user a feeling that the system can see where they are looking, but also that the system is aware of their surroundings.
- For example, in the Windows Holographic OS, the cursor aligns to the surfaces of the user's world, creating a feeling of awareness of the world even when the user isn't looking directly at a hologram..
- Magnetically locking the cursor to an interactive element when it is within close proximity. This can help improve confidence that user will interact with that element when they perform a selection action.

Visual cues

- There is a lot of information in our world and with holograms we are adding more information. A cursor is a great way of communicating to the user what is important.
- If your experience is focused on a single hologram, then perhaps your cursor aligns and hugs only that hologram and changes shape when you look away from that hologram. This can convey to the user that the hologram is special and they can interact with it.
- If your application uses spatial mapping, then your cursor could align and hug every surface it sees. This gives feedback to the users that HoloLens and your application can see their space.
- These things help reinforce the fact that holograms are real and in our world. They help bridge the gap between real and virtual.
- Have an idea of what the cursor should do when there are no holograms or surfaces in view. Placing it at a predetermined distance in front of the user is one option.

Cursor feedback

As we mentioned that its good practice to have the cursor always present, you can use the cursor to convey some important bits of information.

Possible actions

- As the user is gazing at a hologram and the cursor is on that hologram, you could use the cursor to convey possible actions on that hologram.
- You could display an icon on the cursor that the user can for example purchase that item or perhaps scale that hologram. Or even something as simple like the hologram can be tapped on.
- Only add extra information on the cursor if it means something to the user. Otherwise, users might either not notice the state changes or get confused by the cursor.

Input state

- We could use the cursor to display the user's input state. For example, we could display an icon telling the user if the system sees their hand state. This will tell the user that the application knows the user is ready to take action.
- We could also use the cursor to make the user aware that there is a voice command available. Or perhaps change the color of the cursor momentarily to tell the user that the voice command was heard by the system.

These different cursor states can be implemented in different ways. You might implement these different states by modeling the cursor like a state machine. For example:

- Idle state is where you show the default cursor.
- Ready state is when you have detected the user's hand in the ready position.
- Interaction state is when the user is performing a particular interaction.
- Possible Actions state is when you convey possible actions that can be performed on a hologram.

You could also implement these states in a skin-able manner such that you can display different art assets when different states are detected.

Going "cursor-free"

Designing without a cursor is recommended only when the sense of immersion is a key component of an experience, and interactions that involve targeting (through gaze and gesture) do not require great precision. The system must still meet the needs that are normally met by a cursor though - providing users with continuous feedback on the system's understanding of their targeting and helping them to confidently communicate their intentions to the system. This may be achievable through other noticeable state changes.

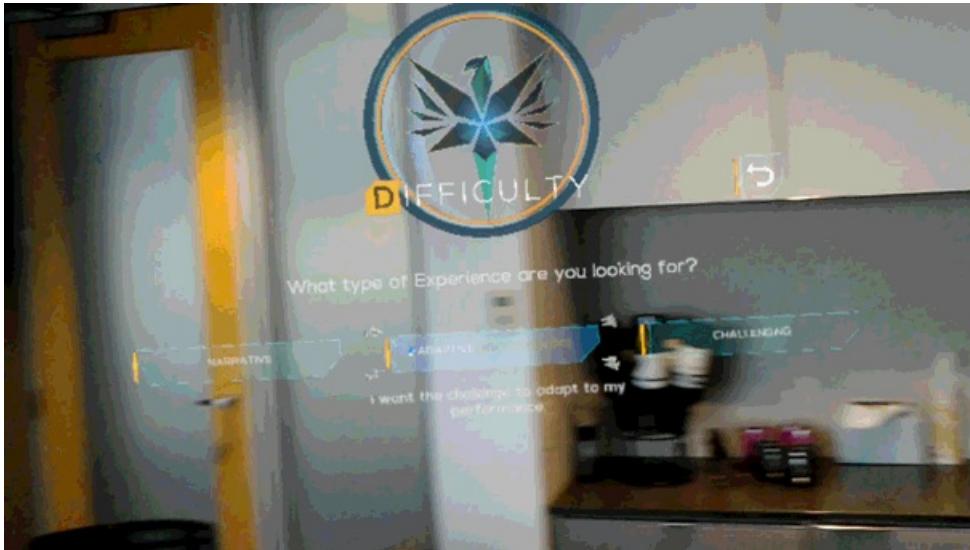
See also

- [Gestures](#)
- [Gaze targeting](#)

Billboarding and tag-along

11/6/2018 • 2 minutes to read • [Edit Online](#)

Billboarding is a behavioral concept that can be applied to objects in mixed reality. Objects with billboarding always orient themselves to face the user. This is especially helpful with text and menuing systems where static objects placed in the user's environment (world-locked) would be otherwise obscured or unreadable if a user were to move around.



HoloLens perspective of a menu system that always faces the user

Objects with billboarding enabled can rotate freely in the user's environment. They can also be constrained to a single axis depending on design considerations. Keep in mind, billboarded objects may clip or occlude themselves if they are placed too close to other objects, or in HoloLens, too close scanned surfaces. To avoid this, think about the total footprint an object may produce when rotated on the axis enabled for billboarding.

What is a tag-along?

Tag-along is a behavioral concept that can be added to holograms, including billboarded objects. This interaction is a more natural and friendly way of achieving the effect of head-locked content. A tag-along object attempts to never leave the user's view. This allows the user to freely interact with what is front of them while also still seeing the holograms outside their direct view.



The HoloLens Start menu is a great example of tag-along behavior

Tag-along objects have parameters which can fine-tune the way they behave. Content can be in or out of the user's line of sight as desired while the user moves around their environment. As the user moves, the content will attempt to stay within the user's periphery by sliding towards the edge of the view, depending on how quickly a user moves may leave the content temporarily out of view. When the user gazes towards the tag-along object, it comes more fully into view. Think of content always being "a glance away" so users never forget what direction their content is in.

Additional parameters can make the tag-along object feel attached to the user's head by a rubber band.

Dampening acceleration or deceleration gives weight to the object making it feel more physically present. This spring behavior is an affordance that helps the user build an accurate mental model of how tag-along works.

Audio helps provide additional affordances for when users have objects in tag-along mode. Audio should reinforce the speed of movement; a fast head turn should provide a more noticeable sound effect while walking at a natural speed should have minimal audio if any effects at all.

Just like truly head-locked content, tag-along objects can prove overwhelming or nauseating if they move wildly or spring too much in the user's view. As a user looks around and then quickly stop, their senses tell them they have stopped. Their balance informs them that their head has stopped turning as well as their vision sees the world stop turning. However, if tag-along keeps on moving when the user has stopped, it may confuse their senses.

See also

- [Cursors](#)
- [Interaction fundamentals](#)
- [Comfort](#)

Updating 2D UWP apps for mixed reality

11/6/2018 • 9 minutes to read • [Edit Online](#)

Windows Mixed Reality enables a user to see holograms as if they are right around you, in your physical or digital world. At its core, both HoloLens and the Desktop PCs you attach immersive headset accessories to are Windows 10 devices; this means that you're able to run almost all of the Universal Windows Platform (UWP) apps in the Store as 2D apps.

Creating a 2D UWP app for mixed reality

The first step to bringing a 2D app to mixed reality headsets is to get your app running as a standard 2D app on your desktop monitor.

Building a new 2D UWP app

To build a new 2D app for mixed reality, you simply build a standard 2D Universal Windows Platform (UWP) app. No other app changes are required for that app to then run as a slate in mixed reality.

To get started building a 2D UWP app, check out the [Create your first app](#) article.

Bringing an existing 2D Store app to UWP

If you already have a 2D Windows app in the Store, you must first ensure it is targeting the Windows 10 Universal Windows Platform (UWP). Here are all the potential starting points you may have with your Store app today:

STARTING POINT	APPX MANIFEST PLATFORM TARGET	HOW TO MAKE THIS UNIVERSAL?
Windows Phone (Silverlight)	Silverlight App Manifest	Migrate to WinRT
Windows Phone 8.1 Universal	8.1 AppX Manifest that Doesn't Include Platform Target	Migrate your app to the Universal Windows Platform
Windows Store 8	8 AppX Manifest that Doesn't Include Platform Target	Migrate your app to the Universal Windows Platform
Windows Store 8.1 Universal	8.1 AppX Manifest that Doesn't Include Platform Target	Migrate your app to the Universal Windows Platform

If you have a 2D Unity app today built as a Win32 app (the "PC, Mac & Linux Standalone" build target), you can target mixed reality by switching Unity to the "Universal Windows Platform" build target instead.

We'll talk about ways that you can restrict your app specifically to HoloLens using the Windows.Holographic device family [below](#).

Run your 2D app in a Windows Mixed Reality immersive headset

If you've deployed your 2D app to the desktop machine where you are developing and tried it out on your monitor, you're already ready to try it out in an immersive desktop headset!

Just go to the Start menu within the mixed reality headset and launch the app from there. The desktop shell and the holographic shell both share the same set of UWP apps, and so the app should already be present once you've deployed from Visual Studio.

Targeting both immersive headsets and HoloLens

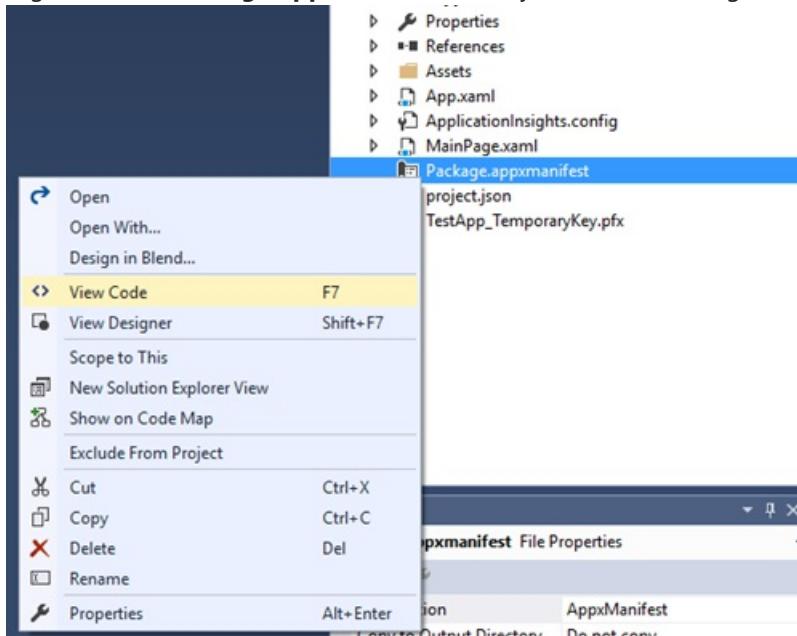
Congratulations! Your app is now using the Windows 10 Universal Windows Platform (UWP).

Your app is now capable of running on today's Windows devices like Desktop, Mobile, Xbox, Windows Mixed Reality immersive headsets, and HoloLens, as well as future Windows devices. However, to actually target all of those devices, you will need to ensure your app is targeting the Windows.Universal device family.

Change your device family to Windows.Universal

Now let's jump into your AppX manifest to ensure your Windows 10 UWP app can run on HoloLens:

- Open your app's solution file with **Visual Studio** and navigate to the app package manifest
- Right click the **Package.appxmanifest** file in your Solution and go to **View Code**



- Ensure your Target Platform is Windows.Universal in the dependencies section

```
<Dependencies>
  <TargetDeviceFamily Name="Windows.Universal" MinVersion="10.0.10240.0" MaxVersionTested="10.0.10586.0"
  />
</Dependencies>
```

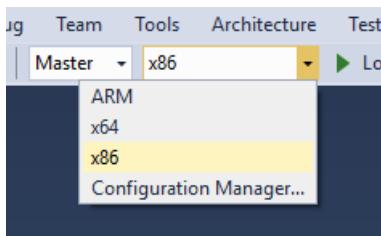
- Save!

If you do not use Visual Studio for your development environment, you can open **AppXManifest.xml** in the text editor of your choice to ensure you're targeting the **Windows.Universal** *TargetDeviceFamily*.

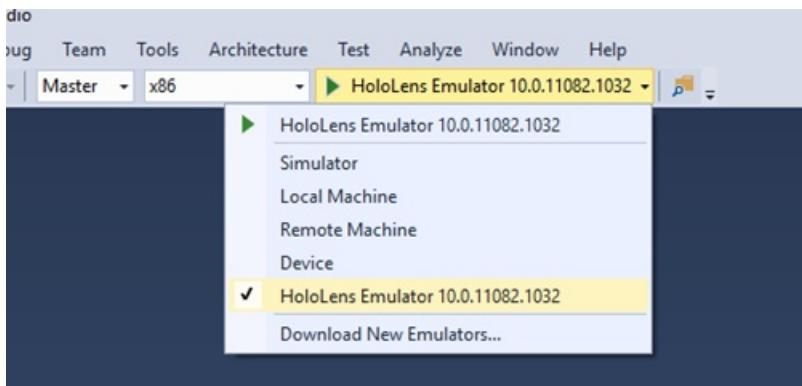
Run in the HoloLens Emulator

Now that your UWP app targets "Windows.Universal", let's build your app and run it in the **HoloLens Emulator**.

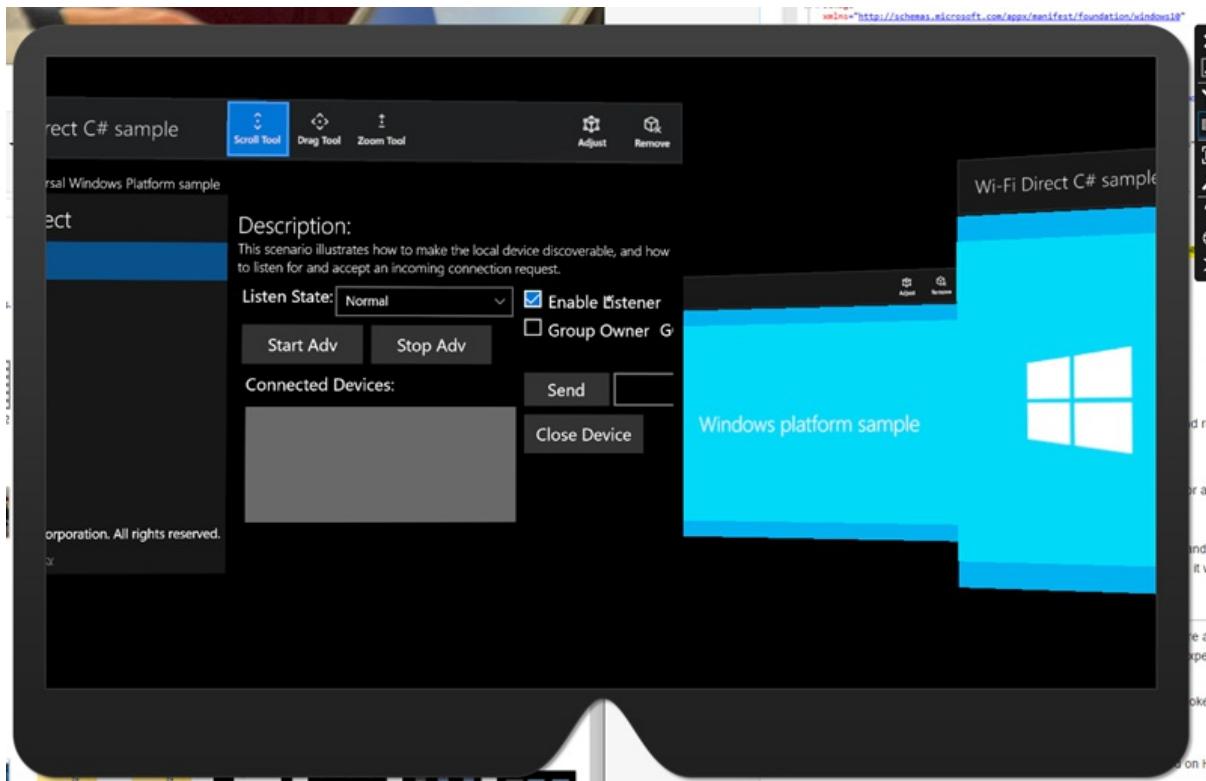
- Make sure you have [installed the HoloLens Emulator](#).
- In Visual Studio, select the **x86** build configuration for your app



- Select **HoloLens Emulator** in the deployment target drop-down menu



- Select **Debug > Start Debugging** to deploy your app and start debugging.
- The emulator will start and run your app.
- With a keyboard, mouse, and/or an Xbox controller, place your app in the world to launch it.



Next steps

At this point, one of two things can happen:

1. Your app will show its splash and start running after it is placed in the Emulator! Awesome!
2. Or after you see a loading animation for a 2D hologram, loading will stop and you will just see your app at its splash screen. This means that something went wrong and it will take more investigation to understand how to bring your app to life in Mixed Reality.

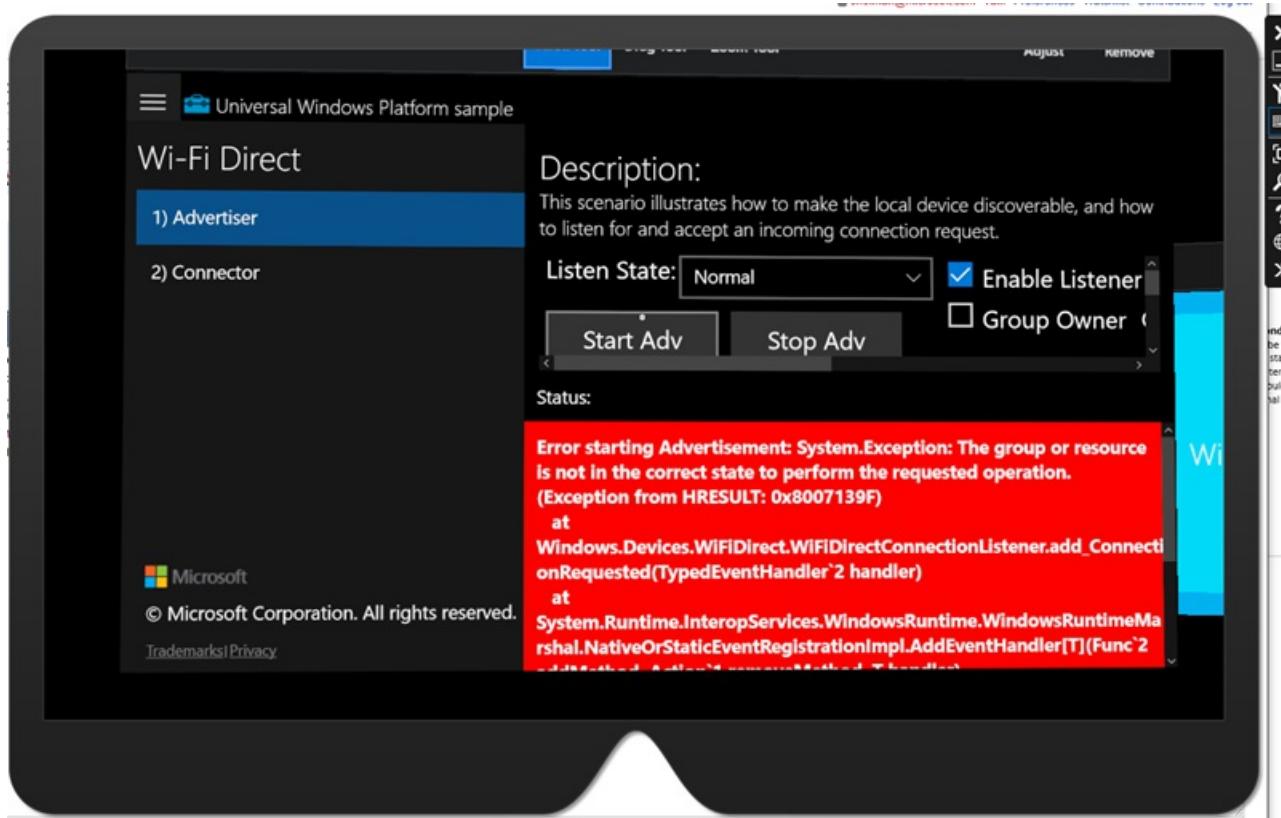
To get to the bottom of what may be causing your UWP app not to start on HoloLens, you'll have to debug.

Running your UWP app in the debugger

These steps will walk you through debugging your UWP app using the Visual Studio debugger.

- If you haven't already done so, open your solution in Visual Studio. Change the target to the **HoloLens Emulator** and the build configuration to **x86**.
- Select **Debug > Start Debugging** to deploy your app and start debugging.
- Place the app in the world with your mouse, keyboard, or Xbox controller.
- Visual Studio should now break somewhere in your app code.

- If your app doesn't immediately crash or break into the debugger because of an unhandled error, then go through a test pass of the core features of your app to make sure everything is running and functional. You may see errors like pictured below (internal exceptions that are being handled). To ensure you don't miss internal errors that impact the experience of your app, run through your automated tests and unit tests to make sure everything behaves as expected.

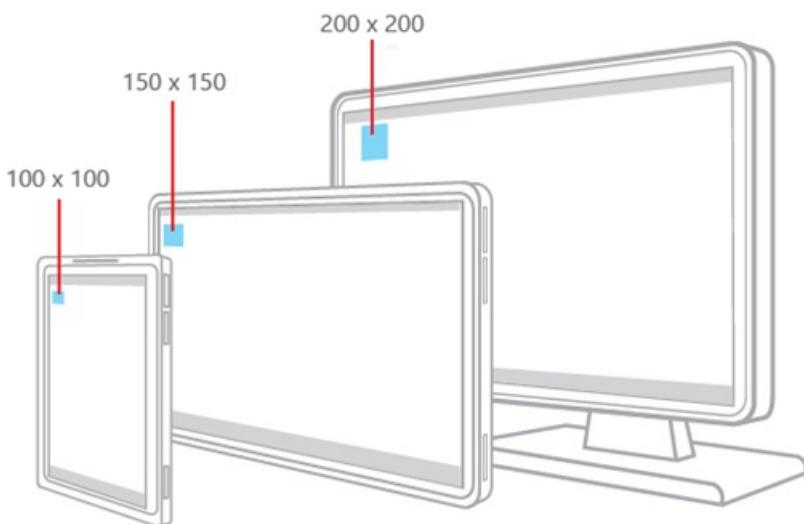


Update your UI

Now that your UWP app is running on immersive headsets and/or HoloLens as a 2D hologram, next we'll make sure it looks beautiful. Here are some things to consider:

- Windows Mixed Reality will run all 2D apps at a fixed resolution and DPI that equates to 853x480 effective pixels. Consider if your design needs refinement at this scale and review the design guidance below to improve your experience on HoloLens and immersive headsets.
- Windows Mixed Reality [does not support](#) 2D live tiles. If your core functionality is showing information on a live tile, consider moving that information back into your app or explore [3D app launchers](#).

2D app view resolution and scale factor



Windows 10 moves all visual design from real screen pixels to **effective pixels**. That means, developers design their UI following the Windows 10 Human Interface Guidelines for effective pixels, and Windows scaling ensures those effective pixels are the right size for usability across devices, resolutions, DPI, etc. See this [great read on MSDN](#) to learn more as well as this [BUILD presentation](#).

Even with the unique ability to place apps in your world at a range of distances, TV-like viewing distances are recommended to produce the best readability and interaction with gaze/gesture. Because of that, a virtual slate in the Mixed Reality Home will display your flat UWP view at:

1280x720, 150%DPI (853x480 effective pixels)

This resolution has several advantages:

- This effective pixel layout will have about the same information density as a tablet or small desktop.
- It matches the fixed DPI and effective pixels for UWP apps running on Xbox One, enabling seamless experiences across devices.
- This size looks good when scaled across our range of operating distances for apps in the world.

2D app view interface design best practices

Do:

- Follow the [Windows 10 Human Interface Guidelines \(HIG\)](#) for styles, font sizes and button sizes. HoloLens will do the work to ensure your app will have compatible app patterns, readable text sizes, and appropriate hit target sizing.
- Ensure your UI follows best practices for [responsive design](#) to look best at HoloLen's unique resolution and DPI.
- Use the "light" color theme recommendations from Windows.

Don't:

- Change your UI too drastically when in mixed reality, to ensure users have a familiar experience in and out of the headset.

Understand the app model

The [app model](#) for mixed reality is designed to use the Mixed Reality Home, where many apps live together. Think of this as the mixed reality equivalent of the desktop, where you run many 2D apps at once. This has implications on app life cycle, Tiles, and other key features of your app.

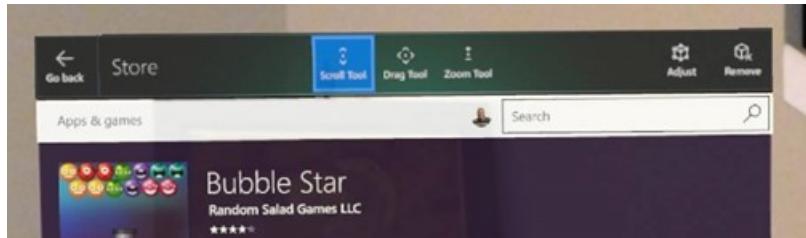
App bar and back button

2D views are decorated with a app bar above their content. The app bar has two points of app-specific

personalization:

Title: displays the *displayname* of the Tile associated with the app instance

Back Button: raises the *BackRequested* event when pressed. Back Button visibility is controlled by *SystemNavigationManager.AppView backButtonVisibility*.



App bar UI in 2D app view

Test your 2D app's design

It is important to test your app to make sure the text is readable, the buttons are targetable, and the overall app looks correct. You can [test](#) on a desktop headset, a HoloLens, an emulator, or a touch device with resolution set to 1280x720 @150%.

New input possibilities

HoloLens uses advanced depth sensors to see the world and see users. This enables advanced hand gestures like [bloom](#) and [air-tap](#). Powerful microphones also enable [voice experiences](#).

With Desktop headsets, users can use motion controllers to point at apps and take action. They can also use a gamepad, targeting objects with their gaze.

Windows takes care of all of this complexity for UWP apps, translating your [gaze](#), gestures, voice and motion controller input to [pointer events](#) that abstract away the input mechanism. For example, a user may have done an air-tap with their hand or pulled the Select trigger on a motion controller, but 2D applications don't need to know where the input came from - they just see a 2D touch press, as if on a touchscreen.

Here are the high level concepts/scenarios you should understand for input when bringing your UWP app to HoloLens:

- [Gaze](#) turns into hover events, which can unexpectedly trigger menus, flyouts or other user interface elements to pop up just by gazing around your app.
- Gaze is not as precise as mouse input. Use appropriately sized hit targets for HoloLens, similar to touch-friendly mobile applications. Small elements near the edges of the app are especially hard to interact with.
- Users must switch input modes to go from scrolling to dragging to two finger panning. If your app was designed for touch input, consider ensuring that no major functionality is locked behind two finger panning. If so, consider having alternative input mechanisms like buttons that can initiate two finger panning. For example, the Maps app can zoom with two finger panning but has a plus, minus, and rotate button to simulate the same zoom interactions with single clicks.

[Voice input](#) is a critical part of the mixed reality experience. We've enabled all of the speech APIs that are in Windows 10 powering Cortana when using a headset.

Publish and Maintain your Universal app

Once your app is up and running, package your app to [submit it to the Microsoft Store](#).

See also

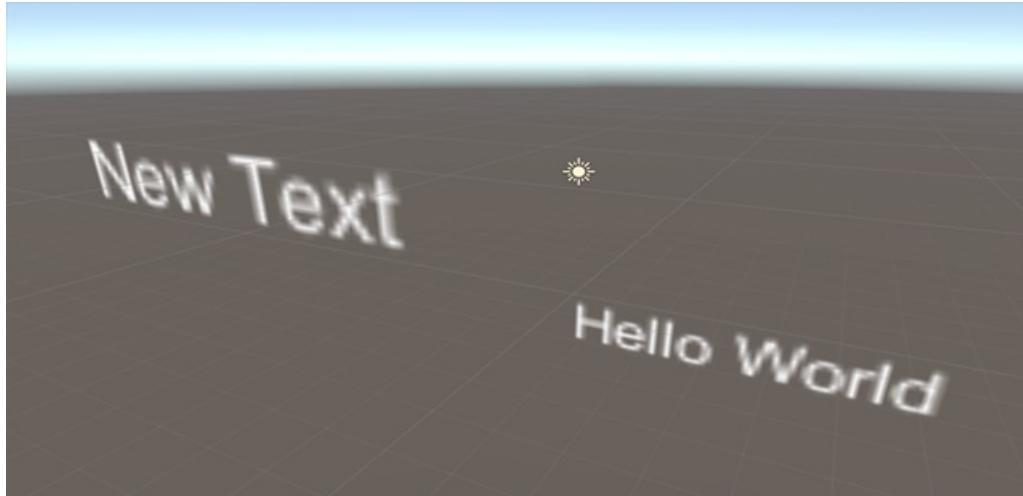
- [App model](#)

- [Gaze](#)
- [Gesture](#)
- [Motion controllers](#)
- [Voice](#)
- [Submitting an app to the Microsoft Store](#)
- [Using the HoloLens emulator](#)

Text in Unity

11/6/2018 • 3 minutes to read • [Edit Online](#)

Text is one of the most important components in holographic apps. To display text in Unity, there are two types of text components you can use — UI Text and 3D Text Mesh. By default they appear blurry and are too big. You need to tweak a few variables to get sharp, high-quality text that has a manageable size in HoloLens. By applying scaling factor to get proper dimensions when using the UI Text and 3D Text Mesh components, you can achieve better rendering quality.



Blurry default text in Unity

Working with fonts in Unity

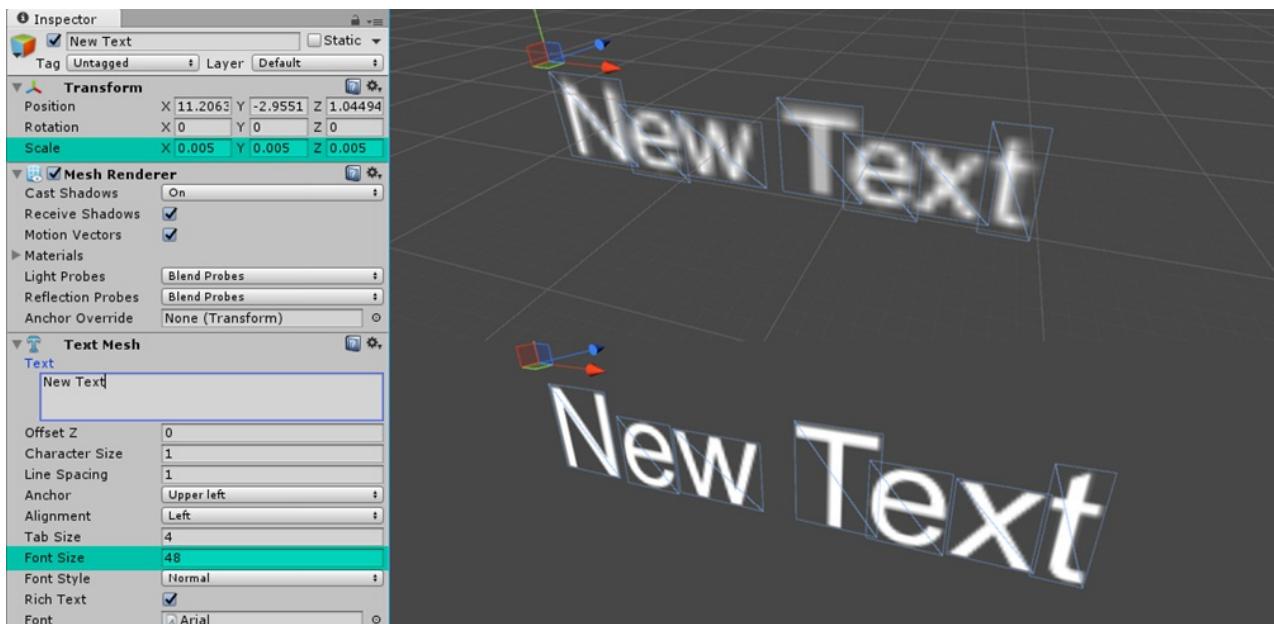
Unity assumes all new elements added to a scene are 1 Unity Unit in size, or 100% transform scale, which translates to about 1 meter on HoloLens. In the case of fonts, the bounding box for a 3D TextMesh comes in by default at about 1 meter in height.



Default Unity text occupies 1 Unity Unit which is 1 meter

Most visual designers use points to define font sizes in the real world. There are about 2835 (2,834.645666399962) points in 1 meter. Based on the point system conversion to 1 meter and Unity's default Text Mesh font size of 13, the simple math of 13 divided by 2835 equals 0.0046 (0.004586111116 to be exact) provides a good standard scale to start with (some may wish to round to 0.005). Scaling the text object or container to these values will not only allow

for the 1:1 conversion of font sizes in a design program, but also provides a standard so you can maintain consistency throughout your application or game.



Unity 3D Text Mesh with optimized values

When adding a UI or canvas based text element to a scene, the size disparity is greater still. The difference in the two sizes is about 1000%, which would bring the scale factor for UI based text components to 0.00046 (0.0004586111116 to be exact) or 0.0005 for the rounded value.



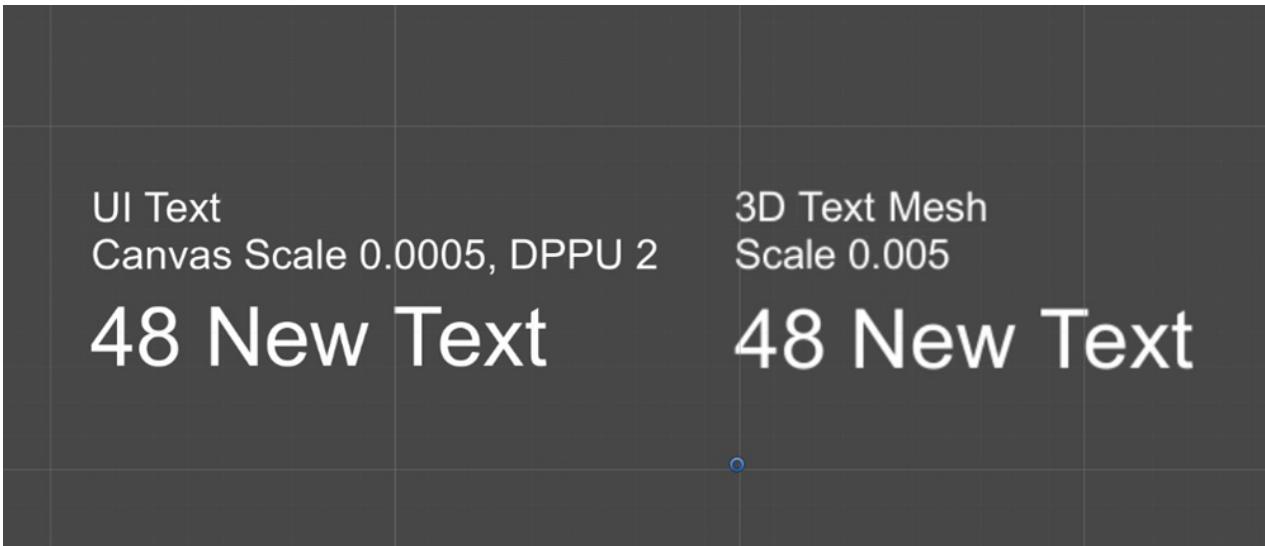
Unity UI Text with optimized values

NOTE

The default value of any font may be affected by the texture size of that font or how the font was imported into Unity. These tests were performed based on the default Arial font in Unity, as well as one other imported font.

Sharp text rendering quality with proper dimension

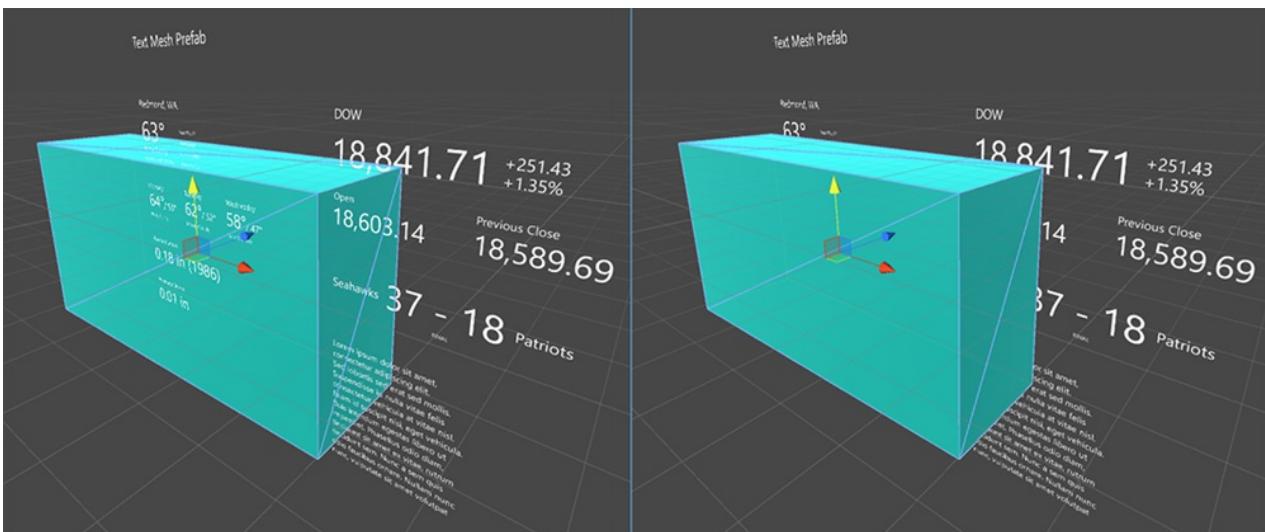
Based on these scaling factors, we have created [two prefabs - UI Text and 3D Text Mesh](#). Developers can use these prefabs to get sharp text and consistent font size.



Sharp text rendering quality with proper dimension

Shader with occlusion support

Unity's default font material does not support occlusion. Because of this, you will see the text behind the objects by default. We have included a simple [shader that supports the occlusion](#). The image below shows the text with default font material (left) and the text with proper occlusion (right).



Shader with occlusion support

Recommended type size

As you can expect, type sizes that we use on a PC or a tablet device (typically between 12–32pt) look quite small at a distance of 2 meters. It depends on the characteristics of each font, but in general the recommended minimum type size for legibility without stroke vibration is around 30pt, based on the scaling factor introduced above. If your app is supposed to be used at a closer distance, smaller type sizes could be used. For the font selection, Segoe UI (the default font for Windows) works well in most cases. However, avoid using light or semi light fonts for type sizes under 42pt since thin vertical strokes will vibrate and it will degrade the legibility. Modern fonts with enough stroke thickness work well. For example, Helvetica and Arial look gorgeous and are very legible in HoloLens with regular or bold weights.



Type ramp example

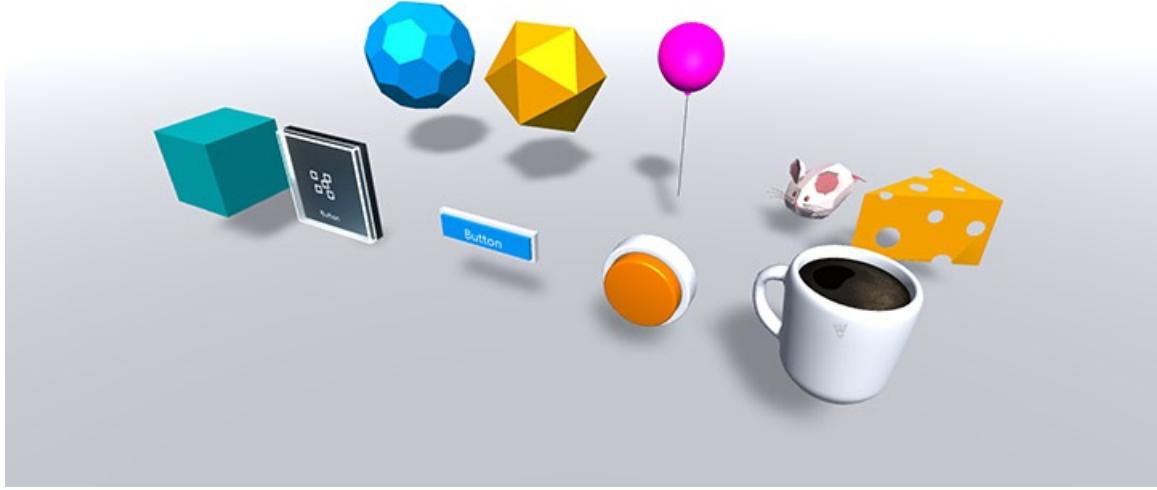
See also

- [Text Prefab in the MixedRealityToolkit](#)
- [Sample text layout prefab and scene](#)
- [Typography](#)

Interactable object

11/6/2018 • 3 minutes to read • [Edit Online](#)

A button has long been a metaphor used for triggering an event in the 2D abstract world. In the three-dimensional mixed reality world, we don't have to be confined to this world of abstraction anymore. Anything can be an **Interactable object** that triggers an event. An interactable object can be represented as anything from a coffee cup on the table to a balloon floating in the air. We still do make use of traditional buttons in certain situations such as in dialog UI. The visual representation of the button depends on the context.



What is a button?

In the two-dimensional world, a button control is commonly used to trigger an event or action. Typically, it's presented as a rectangular shape with different visuals for each interaction state such as idle, hover and pressed. To reinforce this affordance, buttons sometimes include additional visual cues such as shadow or bevel - this was typical in the early days of UI design. Buttons have gradually become flat and more abstract with modern design trends.



The evolution of button design in the two-dimensional world

In the **Mixed Reality Toolkit**, we have created a series of Unity scripts and prefabs that will help you create interactable objects. You can use these to create any type of object that the user can interact with, using these standard interaction states: observation, targeted and pressed. You can easily customize the visual design with your own assets. Detailed animations can be customized by either creating and assigning corresponding animation clips for the interaction states in the Unity's animation controller or using offset and scale. You can find various examples in the [InteractableObjectExample scene](#).



Any type of object can now be a button

Visual feedback for the different input interaction states

In mixed reality, since the holographic objects are mixed with the real-world environment, it could be difficult to understand which objects are interactable. For any interactable objects in your experience, it is important to provide differentiated visual feedback for each input state. This helps the user understand which part of your experience is interactable and makes the user confident with consistent interaction method.

For any objects that user can interact with, we recommended to have different visual feedback for these three input states:

- **Observation:** Default idle state of the object.
- **Targeted:** When the object is targeted with gaze cursor or motion controller's pointer.
- **Pressed:** When the object is pressed with air-tap gesture or motion controller's select button.

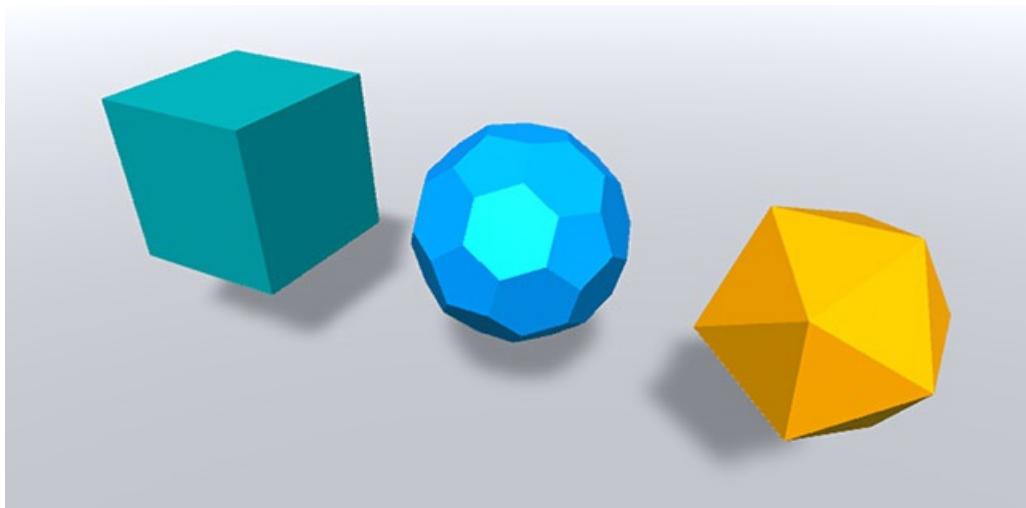


Observation state, targeted state, and pressed state

In Windows Mixed Reality, you can find the examples of visualizing different input states on Start menu and App Bar buttons. You can use techniques such as highlighting or scaling to provide visual feedback to the user's input states.

Interactable object samples

Mesh button



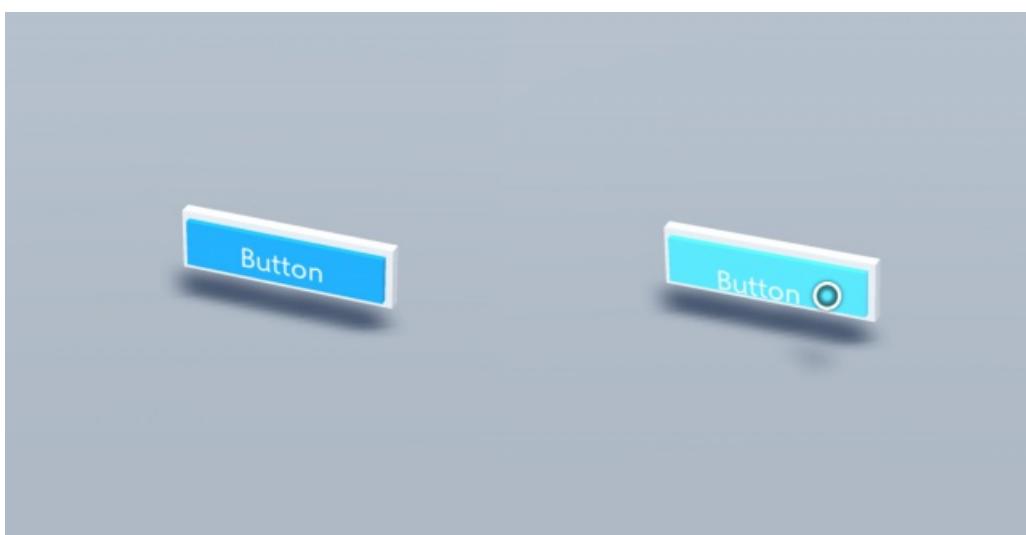
These are examples using primitives and imported 3D meshes as Interactable objects. You can easily assign different scale, offset and color values to respond to each input interaction state.

Toolbar



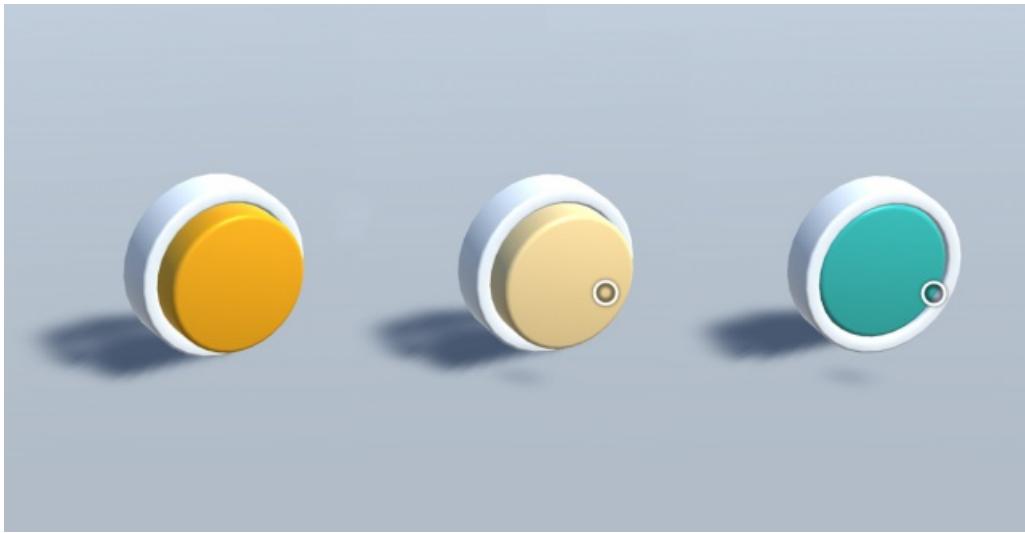
A toolbar is a widely used pattern in mixed reality experiences. It is a simple collection of buttons with additional behaviors such as [Billboarding and tag-along](#). This example uses a Billboarding and tag-along script from the MixedRealityToolkit. You can control detailed behaviors including distance, moving speed and threshold values.

Traditional button



This example shows a traditional 2D style button. Each input state has a slightly different depth and animation property.

Other examples



Push button



Real life object

With HoloLens, you can leverage physical space. Imagine a holographic push button on a physical wall. Or how about a coffee cup on a real table? Using 3D models imported from modeling software, we can create an Interactable object that resembles real life object. Since it's a digital object, we can add magical interactions to it.

Interactable object in Unity

You can find the [examples of Interactable object in Mixed Reality Toolkit](#)

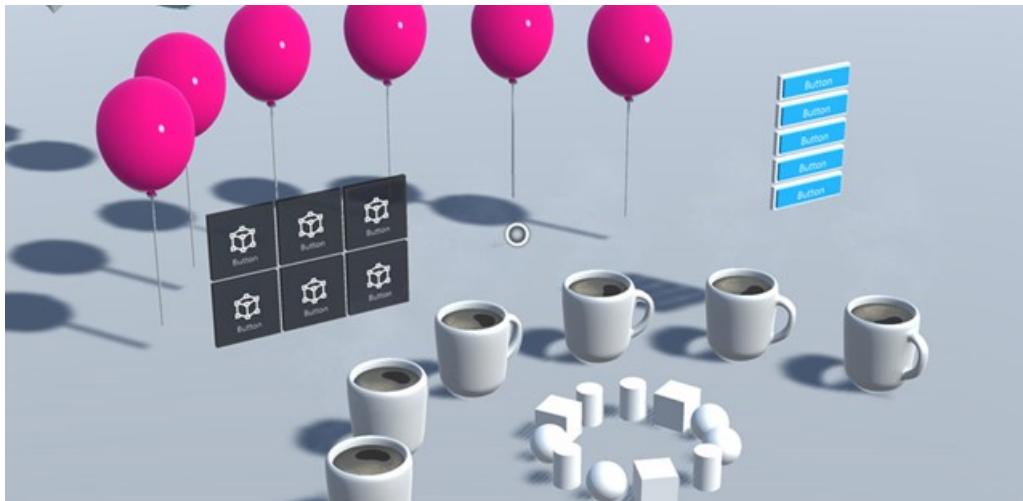
See also

- [Object collection](#)
- [Billboarding and tag-along](#)

Object collection

11/6/2018 • 2 minutes to read • [Edit Online](#)

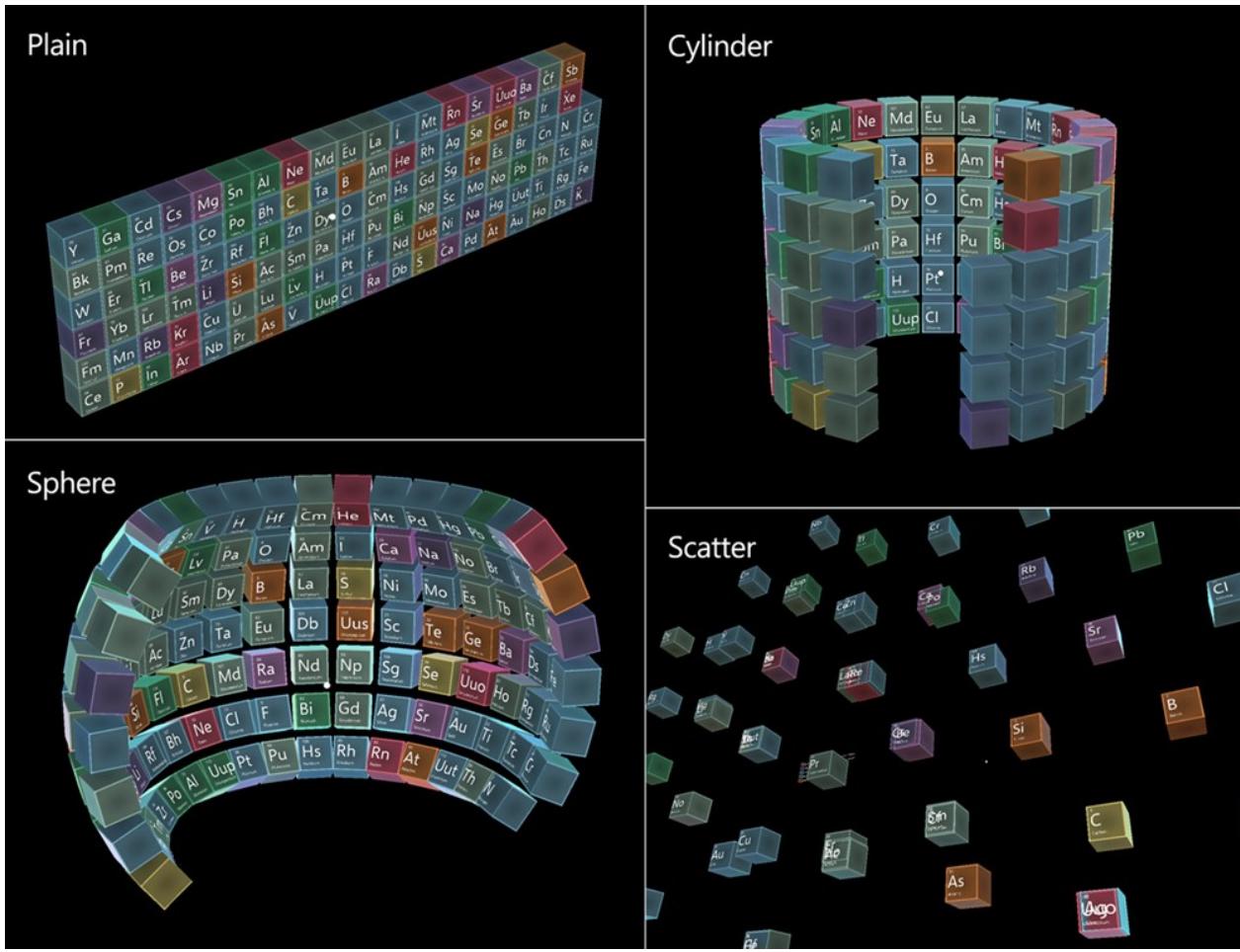
Object collection is a layout control which helps you lay out an array of objects in a predefined three-dimensional shape. It supports four different surface styles - **plane**, **cylinder**, **sphere** and **scatter**. You can adjust the radius and size of the objects and the space between them. Object collection supports any object from Unity - both 2D and 3D. In the [Mixed Reality Toolkit](#), we have created Unity script and [example scene](#) that will help you create an object collection.



Object collection used in the Periodic Table of the Elements sample app

Object collection examples

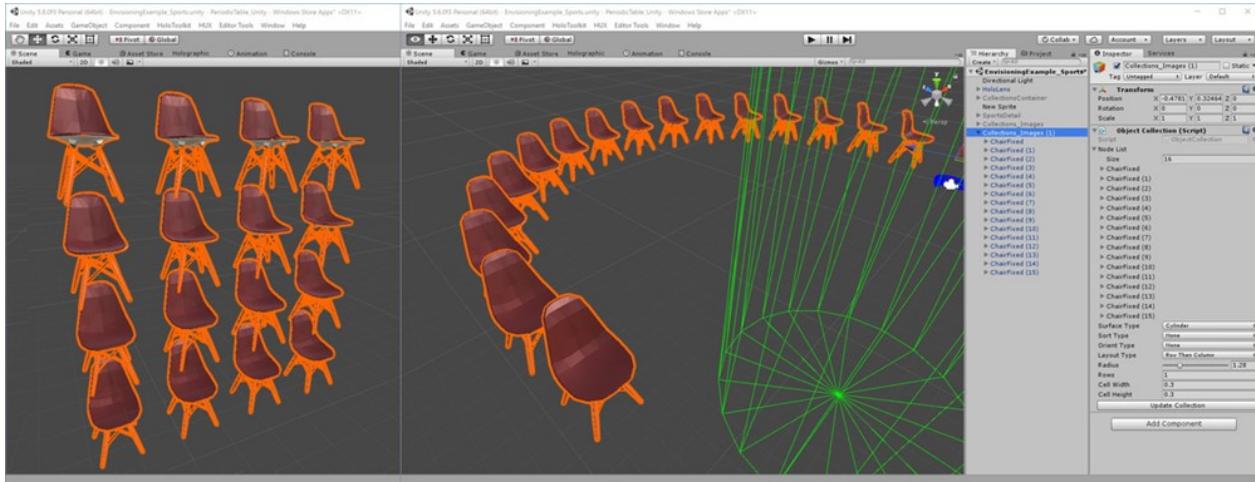
[Periodic Table of the Elements](#) is a sample app that demonstrates how Object collection works. It uses Object collection to lay out 3D chemical element boxes in different shapes.



Object collection examples shown in the Periodic Table of the Elements sample app

3D objects

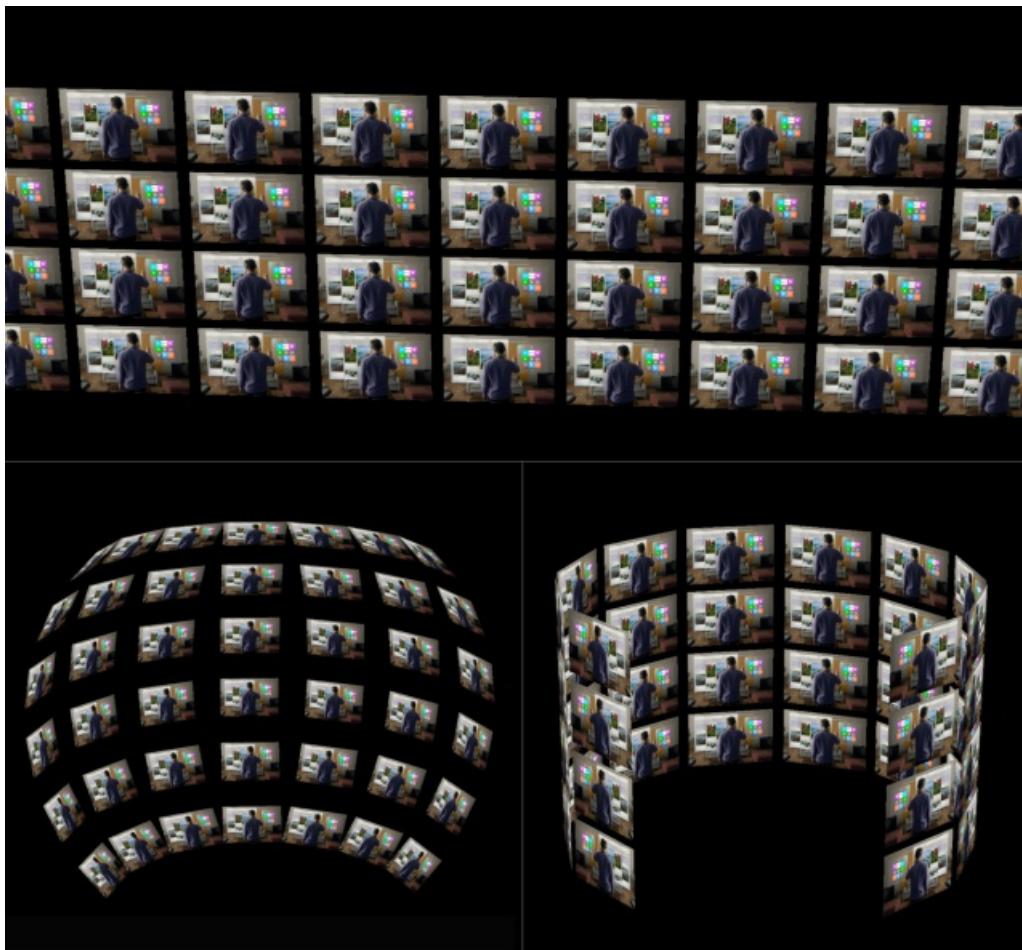
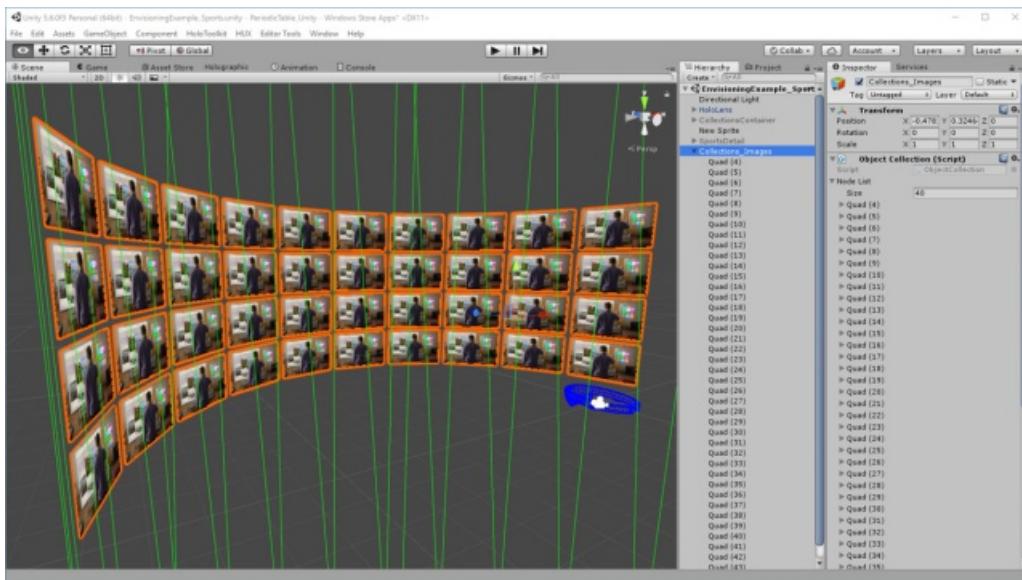
You can use Object collection to lay out imported 3D objects. The example below shows a plane and a cylinder layout of some 3D chair objects.



Examples of plane and cylindrical layouts of 3D objects

2D objects

You can also use 2D images with Object collection. The examples below demonstrate how 2D images can be displayed in a grid.



Examples of 2D images with object collection

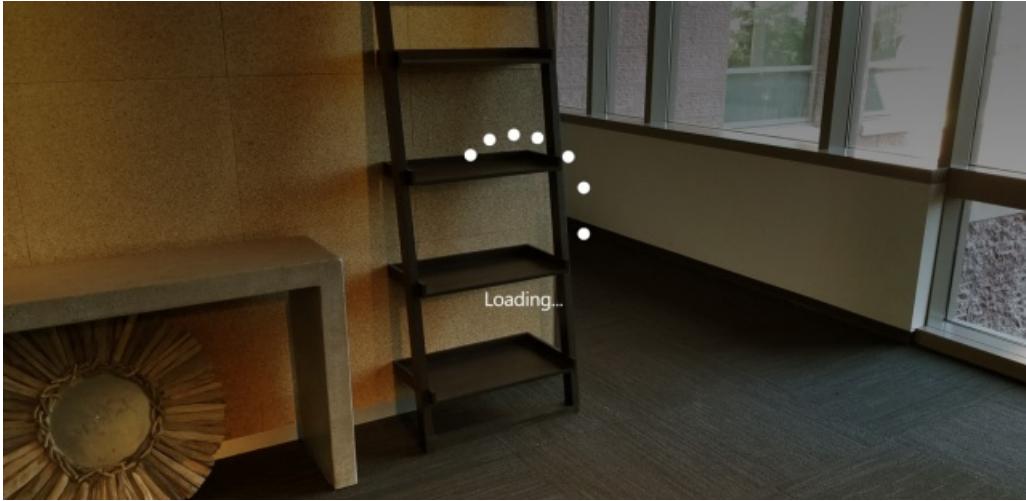
See also

- Scripts and prefabs for Object collection in the Mixed Reality Toolkit on [GitHub](#)
- Interactable object

Displaying progress

11/6/2018 • 2 minutes to read • [Edit Online](#)

A progress control provides feedback to the user that a long-running operation is underway. It can mean that the user cannot interact with the app when the progress indicator is visible, and can also indicate how long the wait time might be, depending on the indicator used.

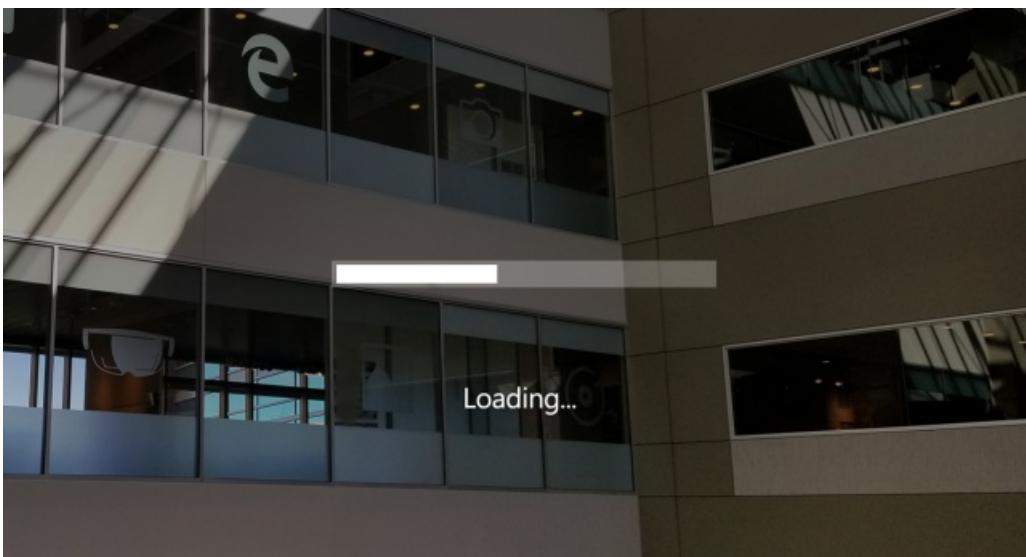


Progress ring example in HoloLens

Types of progress

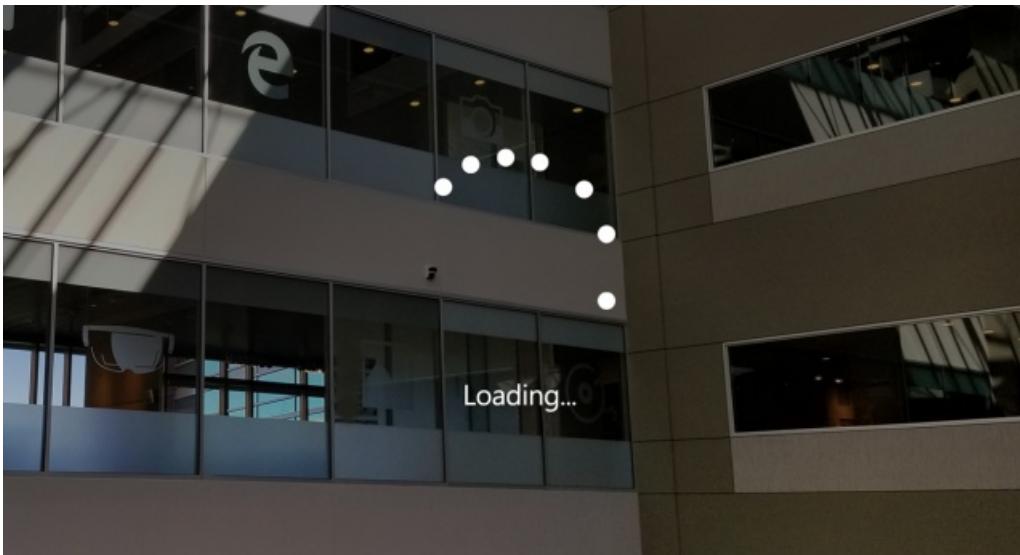
It is important to provide the user information about what is happening. In mixed reality users can be easily distracted by physical environment or objects if your app does not give good visual feedback. For situations that take a few seconds, such as when data is loading or a scene is updating, it is a good idea to show a visual indicator. There are two options to show the user that an operation is underway – a **Progress bar** or a **Progress ring**.

Progress bar



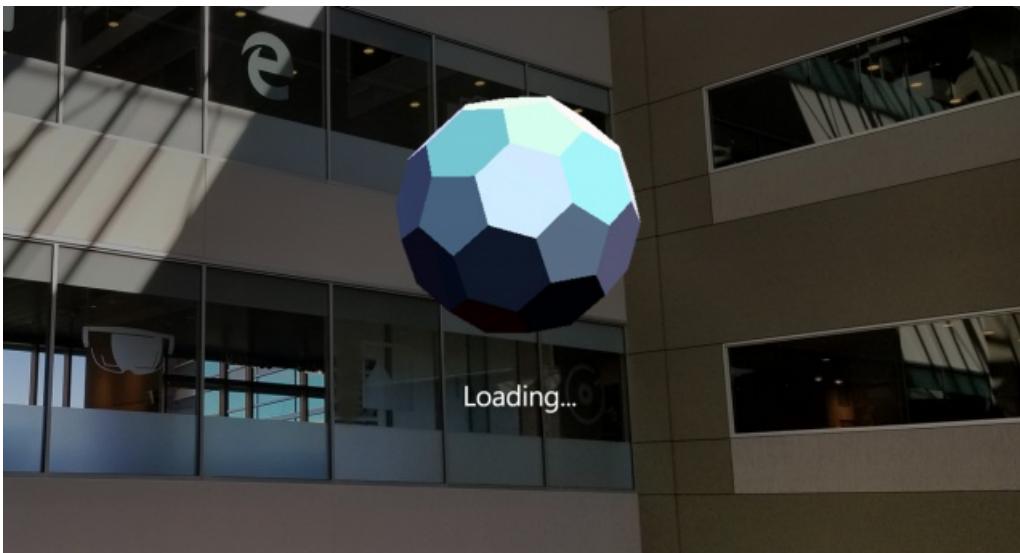
A Progress bar shows the percentage completed of a task. It should be used during an operation whose duration is known (determinate), but its progress should not block the user's interaction with the app.

Progress ring



A Progress ring only has an indeterminate state, and should be used when any further user interaction is blocked until the operation has completed.

Progress with a custom object



You can add to your app's personality and brand identity by customizing the Progress control with your own custom 2D/3D objects.

Best practices

- Tightly couple [billboarding or tag-along](#) to the display of Progress since the user can easily move their head into empty space and lose context. Your app might look like it has crashed if the user is unable to see anything. Billboarding and tag-along is built into the Progress prefab.
- It's always good to provide status information about what is happening to the user. The Progress prefab provides various visual styles including the Windows standard ring-type progress for providing status. You can also use a custom mesh with an animation if you want the style of your progress to align to your app's brand.

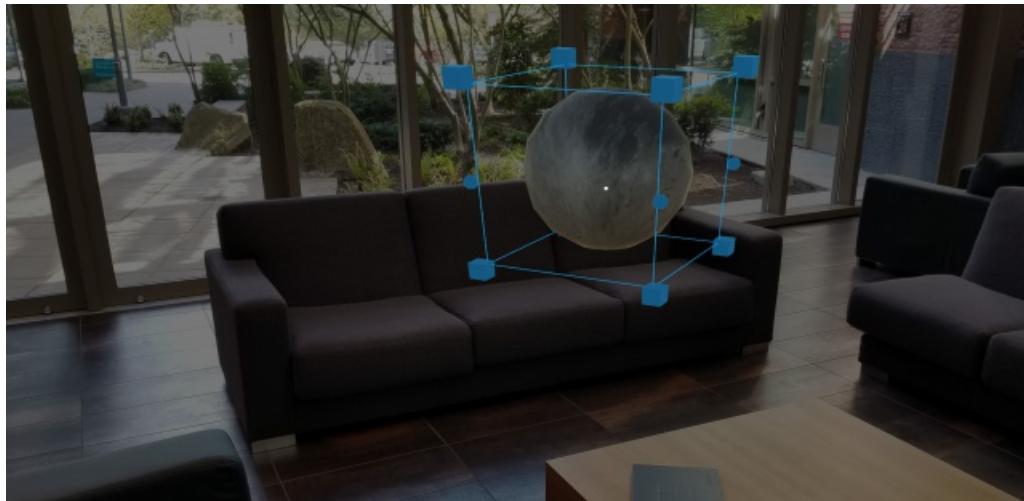
See also

- [Scripts and prefabs for Progress on Mixed Reality Design Labs GitHub](#)
- [Interactable object](#)
- [Object collection](#)
- [Billboarding and tag-along](#)

App bar and bounding box

11/6/2018 • 2 minutes to read • [Edit Online](#)

The App bar is a object-level menu containing a series of buttons that displays on the bottom edge of a hologram's bounds. This pattern is commonly used to give users the ability to remove and adjust holograms.



Bounding boxes are a necessary control for object manipulation in mixed reality

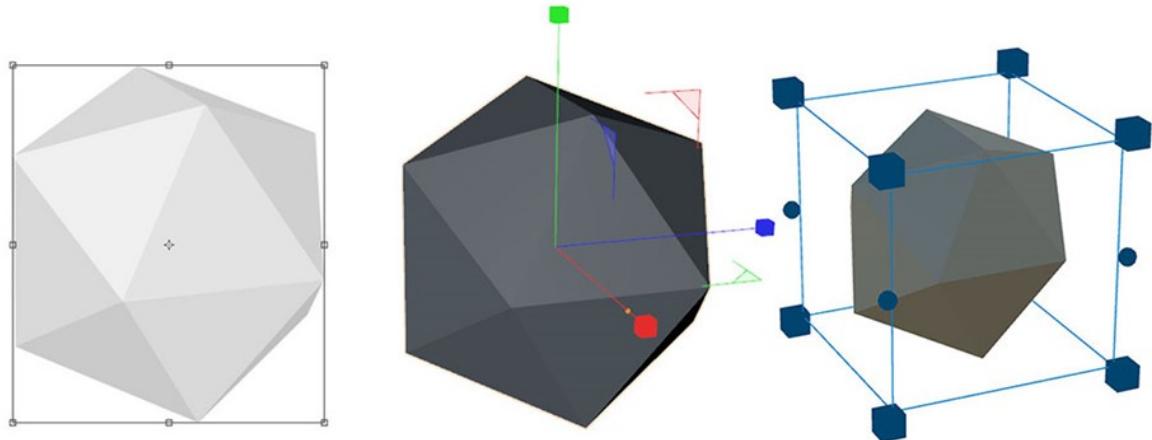
Since this pattern is used with objects that are world locked, as a user moves around the object the App bar will always display on the objects' side closest to the user. While this isn't billboarding, it effectively achieves the same result; preventing a user's position to occlude or block functionality that would otherwise be available from a different location in their environment.



Walking around a hologram, the App bar follows

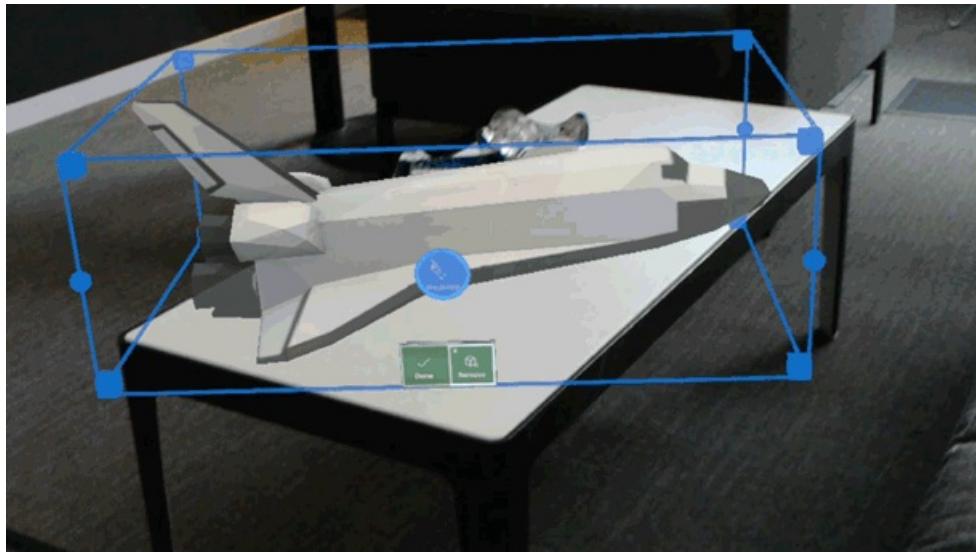
The App bar was designed primarily as a way to manage placed objects in a user's environment. Coupled with the bounding box, a user has full control over where and how objects are oriented in mixed reality.

What is the bounding box?



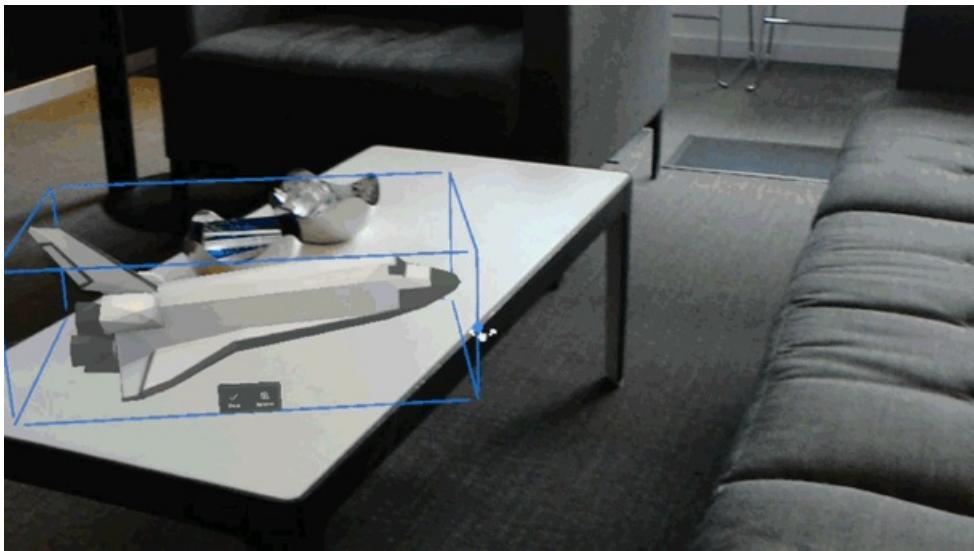
A 2D and 3D resize gizmo, and a holographic bounding box

The bounding box is a state for objects editable in a user's environment. It provides the user an affordance that the object is currently adjustable. The corners tell the user that the object can also scale. This visual affordance shows users the total area of the object – even if it's not visible outside of an adjustment mode. This is especially important because if it weren't there, an object snapped to another object or surface may appear to behave as if there was space around it that shouldn't be there.



Scaling an object via bounding box

The cube-like corners of the bounding box follow a widely understood pattern for adjusting scale. Coupled with the explicit action of putting an object into "adjust mode" it's clear they can both move the object, but also scale it in their environment.



Rotating an object via bounding box

The spherical affordances on the edges of the bounding box are rotation indicators. This gives the user more fine adjustment over their placed holograms. Not only can they adjust and scale, but now rotate as well.

See also

- [Scripts and prefabs for App bar and bounding box on Mixed Reality Design Labs GitHub](#)
- [Interactable object](#)
- [Text in Unity](#)
- [Object collection](#)
- [Displaying progress](#)

3D app launcher design guidance

11/6/2018 • 6 minutes to read • [Edit Online](#)

When you put on a Windows Mixed Reality immersive (VR) headset, you enter the Windows Mixed Reality home, visualized as a house on a cliff surrounded by mountains and water (though you can [choose other environments and even create your own](#)). Within the space of this home, a user is free to arrange and organize the 3D objects and apps that they care about any way they want. A **3D app launcher** is a “physical” object in the user’s mixed reality house that they can select to launch an app.



Floaty Bird 3D app launcher example (fictional app)

3D app launcher creation process

There are 3 steps to creating a 3D app launcher:

1. Designing and concepting (this article)
2. [Modeling and exporting](#)
3. Integrating it into your application:
 - [UWP apps](#)
 - [Win32 apps](#)

Design concepts

Fantastic yet familiar

The Windows Mixed Reality environment your app launcher lives in is part familiar, part fantastical/sci-fi. The best launchers follow the rules of this world. Think of how you can take a familiar, representative object from your app, but bend some of the rules of actual reality. Magic will result.

Intuitive

When you look at your app launcher, its purpose - to launch your app - should be obvious and shouldn’t cause any confusion. For example, be sure your launcher is an obvious-enough representative of your app that it won’t be confused for a piece of decor in the Cliff House. Your app launcher should invite people to touch/select it.



Fresh Note 3D app launcher example (fictional app)

Home scale

3D app launchers live in the Cliff House and their default size should make sense with the other “physical” objects in the space. If you place your launcher beside, say, a house plant or some furniture, it should feel at home, size-wise. A good starting point is to see how it looks at 30 cubic centimeters, but remember that users can scale it up or down if they like.

Own-able

The app launcher should feel like an object a person would be excited to have in their space. They’ll be virtually surrounding themselves with these things, so the launcher should feel like something the user thought was desirable enough to seek out and keep nearby.



Astro Warp 3D app launcher example (fictional app)

Recognizable

Your 3D app launcher should instantly express “your app’s brand” to people who see it. If you have a star character or an especially identifiable object in your app, we recommend using that as a big part of your design. In a mixed reality world, an object will draw more interest from users than just a logo alone. Recognizable objects communicate brand quickly and clearly.

Volumetric

Your app deserves more than just putting your logo on a flat plane and calling it a day. Your launcher should feel like an exciting, 3D, physical object in the user’s space. A good approach is to imagine your app was going to have a balloon in the Macy’s Thanksgiving Day Parade. Ask yourself, what would really wow people as it came down the street? What would look great from all viewing angles?



Logo only



More recognizable

with a character



Flat approach, not

surprisingly, feels flat



Volumetric

approach better showcases your app

Tips for good 3D models

Best practices

- When planning dimensions for your app launcher, shoot for roughly a 30cm cube. So, a 1:1:1 size ratio.
- Models must be under 10,000 polygons. [Learn more about triangle counts and levels of details \(LODs\)](#)
- Test on an immersive headset when possible.
- Build details into your model's geometry where possible – don't rely on textures for detail.
- Build "water tight" closed geometry. No holes that are not modeled in.
- Use natural materials in your object. Imagine crafting it in the real world.
- Make sure your model reads well at different distances and sizes.
- When your model is ready to go, read the [exporting assets guidelines](#).



Model with subtle details in the texture

What to avoid

- Don't use high-contrast details or small, busy patterns and textures.
- Don't use thin geometry – it doesn't work well at a distance and will alias badly.
- Don't let parts of your model extend too much beyond the 1:1:1 size ratio. It will create scaling problems.



Avoid high-contrast, small, busy patterns

How to handle type

Best practices

- We recommend your type comprises about 1/3 of your app launcher (or more). Type is the main thing that gives people an idea that your launcher is, in fact, a launcher so it's nice if it's pretty substantial.
- Avoid making type super wide – try to keep it within the confines of the app launcher's core dimensions (more or less).
- Flat type can work, but be aware that it can be hard to view from certain angles and in certain environments. You might consider putting it a solid object or backdrop behind it to help with this.
- Adding dimension to your type feels nice in 3D. Shading the sides of the type a different, darker color can help with readability.



backdrop can be hard to view from certain angles and in certain environments

Flat type without a



Type with a built-in backdrop can work well



Extruded type can

work well if you shade the sides

Type colors that work

- White
- Black
- Bright semi-saturated color



Type colors that work

What to avoid

Type colors that cause trouble

- Mid-tones
- Gray
- Over-saturated colors or desaturated colors



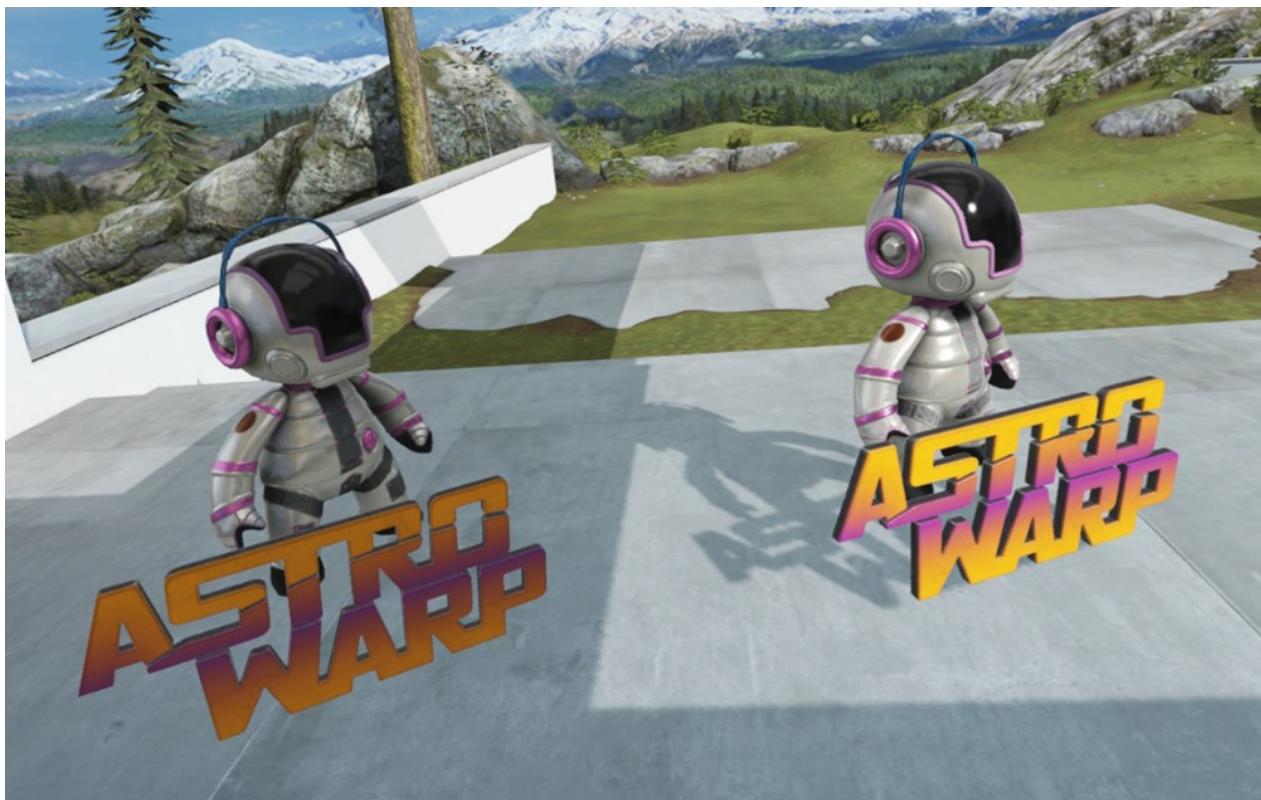
Type colors that cause trouble

Lighting

The lighting for your app launcher comes from the Cliff House environment. Be sure to test your launcher in several places throughout the house so it looks good in both light and shadows. The good news is, if you've followed the other design guidance in this document, your launcher should be in pretty good shape for most lighting in the Cliff House.

Good places to test how your launcher looks in the various lights in the environment are the Studio, the Media

Room, anywhere outside and on the Back Patio (the concrete area with the lawn). Another good test is to put it in half light and half shadow and see what it looks like.



Make sure your launcher looks good in both light and shadows

Texturing

Authoring your textures

The end format of your 3D app launcher will be a .glb file, which is made using the PBR (Physically Based Rendering) pipeline. This can be a tricky process - now is a good time to employ a technical artist if you haven't already. If you're a brave DIY-er, taking the time to [research and learn PBR terminology](#) and what's happening under the hood before you begin will help you avoid common mistakes.



Fresh Note 3D app launcher example (fictional app)

Recommended authoring tool

We recommend using [Substance Painter](#) by Allegorithmic to author your final file. If you're not familiar with authoring PBR shaders in Substance Painter, here's a [tutorial](#).

(Alternately [3D-Coat](#), [Quixel Suite 2](#), or [Marmoset Toolbag](#) would also work if you're more familiar with one of these.)

Best practices

- If your app launcher object was authored for PBR, it should be pretty straightforward to convert it for the Cliff House environment.
- Our shader is expecting a Metal/Roughness work flow – The Unreal PBR shader is a close facsimile.
- When exporting your textures keep the [recommended texture sizes](#) in mind.
- Make sure to build your objects for real-time lighting — this means:
 - Avoid baked shadows – or painted shadows
 - Avoid baked lighting in the textures
 - Use one of the PBR material authoring packages to get the right maps generated for our shader

See also

- [Create 3D models for use in the mixed reality home](#)
- [Implement 3D app launchers \(UWP apps\)](#)
- [Implement 3D app launchers \(Win32 apps\)](#)

Create 3D models for use in the home

11/6/2018 • 12 minutes to read • [Edit Online](#)

The [Windows Mixed Reality home](#) is the starting point where users land before launching applications. You can design your application for Windows Mixed Reality headsets to leverage a [3D model as an app launcher](#) and to allow [3D deep links to be placed into the Windows Mixed Reality home](#) from within your app. This article outlines the guidelines for creating 3D models compatible with the Windows Mixed Reality home.

Asset requirements overview

When creating 3D models for Windows Mixed Reality there are some requirements that all assets must meet:

1. [Exporting](#) - Assets must be delivered in the .glb file format (binary glTF)
2. [Modeling](#) - Assets must be less than 10k triangles, have no more than 64 nodes and 32 submeshes per LOD
3. [Materials](#) - Textures cannot be larger than 4096 x 4096 and the smallest mip map should be no larger than 4 on either dimension
4. [Animation](#) - Animations cannot be longer than 20 minutes at 30 FPS (36,000 keyframes) and must contain <= 8192 morph target vertices
5. [Optimizing](#) - Assets should be optimized using the [WindowsMRAssetConverter](#). This is **required on Windows OS Versions <= 1709** and recommended on Windows OS versions >= 1803

The rest of this article includes a detailed overview of these requirements as well as additional guidelines to ensure your models work well with the Windows Mixed Reality home.

Detailed guidance

Exporting models

The Windows Mixed Reality home expects 3D assets to be delivered using the .glb file format with embedded images and binary data. Glb is the binary version of the glTF format which is a royalty free open standard for 3D asset delivery maintained by the Khronos group. As glTF evolves as an industry standard for interoperable 3D content so will Microsoft's support for the format across Windows apps and experiences. If you haven't created a glTF asset before you can find a [list of supported exporters and converters](#) on the glTF working group github page.

Modeling guidelines

Windows expects assets to be generated using the following modeling guidelines to ensure compatibility with the Mixed Reality home experience. When modeling in your program of your choice keep in mind the following recommendations and limitations:

1. The Up axis should be set to "Y".
2. The asset should face "forward" towards the positive Z axis.
3. All assets should be built on the ground plane at the scene origin (0,0,0)
4. Working Units should be set to meters and assets so that assets can be authored at world scale
5. All meshes do not need to be combined but it is recommended if you are targeting resource constrained devices
6. All meshes should share 1 material, with only 1 texture set being used for the whole asset
7. UVs must be laid out in a square arrangement in the 0-1 space. Avoid tiling textures although they are

- permitted.
8. Multi-UVs are not supported
 9. Double sided materials are not supported

Triangle counts and levels of detail (LODs)

The Windows Mixed Reality home does not support models with more than 10,000 triangles. It's recommended that you triangulate your meshes before exporting to ensure that they do not exceed this count. Windows MR also supports optional geometry levels of detail (LODs) to ensure a performant and high-quality experience. [The WindowsMRAssetConverter](#) will help you combine 3 versions of your model into a single .glb model. Windows determines which LOD to display based on the amount of screen real estate the model is taking up. Only 3 LOD levels are supported with the following recommended triangle counts:

LOD LEVEL	RECOMMENDED TRIANGLE COUNT	MAX TRIANGLE COUNT
LOD 0	10,000	10,000
LOD 1	5,000	10,000
LOD 2	2,500	10,000

Node Counts and Submesh limits

The Windows Mixed Reality home does not support models with more than 64 nodes or 32 submeshes per LOD. Nodes are a concept in the [glTF specification](#) that define the objects in the scene. Submeshes are defined in the array of [primitives](#) on the mesh in the object.

FEATURE	DESCRIPTION	MAX SUPPORTED	DOCUMENTATION
Nodes	Objects in the glTF Scene	64 per LOD	Here
Submeshes	Sum of primitives on all meshes	32 per LOD	Here

Material Guidelines

Textures should be prepared using a PBR metal roughness workflow. Begin by creating a full set of textures including Albedo, Normal, Occlusion, Metallic, and Roughness. Windows Mixed Reality supports textures with resolutions up to 4096x4096 but its recommended that you author at 512x512. Additionally textures should be authored at resolutions in multiples of 4 as this is a requirement for the compression format applied to textures in the exporting steps outlined below. Finally, when generating mip maps or a texture the lowest mip must be a maximum of 4x4.

RECOMMENDED TEXTURE SIZE	MAX TEXTURE SIZE	LOWEST MIP
512x512	4096x4096	max 4x4

Albedo (base color) map

Raw color with no lighting information. This map also contains the reflectance and diffuse information for metal (white in the metallic map) and insulator (black in the metallic map) surfaces respectively.

Normal

Tangent Space Normal map

Roughness map

Describes the microsurface of the object. White 1.0 is rough Black 0.0 is smooth. This map gives the asset the most character as it truly describes the surface e.g. scratches, fingerprints, smudges, grime etc.

Ambient occlusion map

Value scale map depicting areas of occluded light which blocks reflections

Metallic map

Tells the shader if something is metal or not. Raw Metal = 1.0 white Non metal = 0.0 black. There can be transitional gray values that indicate something covering the raw metal such as dirt, but in general this map should be black and white only.

Optimizations

Windows Mixed Reality home offers a series of optimizations on top of the core glTF spec defined using custom extensions. These optimizations are required on Windows versions <= 1709 and recommended on newer versions of Windows. You can easily optimize any glTF 2.0 model using the [Windows Mixed Reality Asset Converter available on GitHub](#). This tool will perform the correct texture packing and optimizations as specified below. For general usage we recommend using the WindowsMRAssetConverter, but if you need more control over the experience and would like to build your own optimization pipeline then you can refer to the detailed specification below.

Materials

To improve asset loading time in Mixed Reality environments Windows MR supports rendering compressed DDS textures packed according to the texture packing scheme defined in this section. DDS textures are referenced using the [MSFT_texture_dds extension](#). Compressing textures is highly recommended.

HoloLens

HoloLens-based mixed reality experiences expect textures to be packed using a 2-texture setup using the following packing specification:

GLTF PROPERTY	TEXTURE	PACKING SCHEME
pbrMetallicRoughness	baseColorTexture	Red (R), Green (G), Blue (B)
MSFT_packing_normalRoughnessMetallic	normalRoughnessMetallicTexture	Normal (RG), Roughness (B), Metallic (A)

When compressing the DDS textures the following compression is expected on each map:

TEXTURE	EXPECTED COMPRESSION
baseColorTexture, normalRoughnessMetallicTexture	BC7

Immersive (VR) headsets

PC-based Windows Mixed Reality experiences for immersive (VR) headsets expect textures to be packed using a 3-texture setup using the following packing specification:

Windows OS >= 1803

GLTF PROPERTY	TEXTURE	PACKING SCHEME
pbrMetallicRoughness	baseColorTexture	Red (R), Green (G), Blue (B)

GLTF PROPERTY	TEXTURE	PACKING SCHEME
MSFT_packing_occlusionRoughnessMetallic	occlusionRoughnessMetallicTexture	Occlusion (R), Roughness (G), Metallic (B)
MSFT_packing_occlusionRoughnessMetallic	normalTexture	Normal (RG)

When compressing the DDS textures the following compression is expected on each map:

TEXTURE	EXPECTED COMPRESSION
normalTexture	BC5
baseColorTexture, occlusionRoughnessMetallicTexture	BC7

Windows OS <= 1709

GLTF PROPERTY	TEXTURE	PACKING SCHEME
pbrMetallicRoughness	baseColorTexture	Red (R), Green (G), Blue (B)
MSFT_packing_occlusionRoughnessMetallic	roughnessMetallicOcclusionTexture	Roughness (R), Metallic (G), Occlusion (B)
MSFT_packing_occlusionRoughnessMetallic	normalTexture	Normal (RG)

When compressing the DDS textures the following compression is expected on each map:

TEXTURE	EXPECTED COMPRESSION
normalTexture	BC5
baseColorTexture, roughnessMetallicOcclusionTexture	BC7

Adding mesh LODs

Windows MR uses geometry node LODs to render 3D models in different levels of detail depending on screen coverage. While this feature is technically not required, it's strongly recommended for all assets. Currently Windows supports 3 levels of detail. The default LOD is 0, which represents the highest quality. Other LODs are numbered sequentially, e.g. 1, 2 and get progressively lower in quality. The [Windows Mixed Reality Asset Converter](#) supports generating assets that meet this LOD specification by accepting multiple glTF models and merging them into a single asset with valid LOD levels. The following table outlines the expected LOD ordering and triangle targets:

LOD LEVEL	RECOMMENDED TRIANGLE COUNT	MAX TRIANGLE COUNT
LOD 0	10,000	10,000
LOD 1	5,000	10,000
LOD 2	2,500	10,000

When using LODs always specify 3 LOD levels. Missing LODs will cause the model to not render unexpectedly as the LOD system switches to the missing LOD level. glTF 2.0 does not currently support LODs as part of the core spec. LODs should therefore be defined using the [MSFT_Lod extension](#).

Screen coverage

LODs are displayed in Windows Mixed Reality based on a system driven by the screen coverage value set on each LOD. Objects that are currently consuming a larger portion of the screen space are displayed at a higher LOD level. Screen coverage is not a part of the core glTF 2.0 spec and must be specified using [MSFT_ScreenCoverage](#) in the "extras" section of the [MSFT_lod extension](#).

LOD LEVEL	RECOMMENDED RANGE	DEFAULT RANGE
LOD 0	100% - 50%	.5
LOD 1	Under 50% - 20%	.2
LOD 2	Under 20% - 1%	.01
LOD 4	Under 1%	-

Animation guidelines

NOTE

This feature was added as part of [Windows 10 April 2018 Update](#). On older versions of Windows these animations will not play back, however, they will still load if authored according to the guidance in this article.

The mixed reality home supports animated glTF objects on HoloLens and immersive (VR) headsets. If you wish to trigger animations on your model, you'll need to use the Animation Map extension on the glTF format. This extension lets you trigger animations in the glTF model based on the users presence in the world, for example trigger an animation when the user is close to the object or while they are looking at it. If your glTF object has animations, but doesn't define triggers the animations will not be played back. The section below describes one workflow for adding these triggers to any animated glTF object.

Tools

First, download the following tools if you don't have them already. These tools will make it easy to open any glTF model, preview it, make changes and save back out as glTF or .glb:

1. [Visual Studio Code](#)
2. [glTF Tools for Visual Studio Code](#)

Opening and previewing the model

Start by opening up the glTF model in VSCode by dragging the .glTF file into the editor window. Note that if you have a .glb instead of a .glTF file you can import it into VSCode using the glTF Tools addin that you downloaded. Go to "View -> Command Palette" and then begin typing "glTF" in the command palette and select "glTF: Import from glb" which will pop up a file picker for you to import a .glb with.

Once you've opened your glTF model you should see the JSON in the editor window. Note that you can also preview the model in a live 3D viewer using the by right clicking the file name and selecting the "glTF: Preview 3D Model" command shortcut from the right click menu.

Adding the triggers

Animation triggers are added to glTF model JSON using the Animation Map extension. The animation map

extension is publicly documented [here on GitHub](#) (NOTE: THIS IS A DRAFT EXTENSION). To add the extension to your model just scroll to the end of the glTF file in the editor and add the "extensionsUsed" and "extensions" block to your file if they don't already exist. In the "extensionsUsed" section you'll add a reference to the "EXT_animation_map" extension and in the "extensions" block you'll add your mappings to the animations in the model.

As noted [in the spec](#) you define what triggers the animation using the "semantic" string on a list of "animations" which is an array of animation indices. In the example below we've specified the animation to play while the user is gazing at the object:

```
"extensionsUsed": [
    "EXT_animation_map"
],
"extensions" : {
    "EXT_animation_map" : {
        "bindings": [
            {
                "semantic": "GAZE",
                "animations": [0]
            }
        ]
    }
}
```

The following animation triggers semantics are supported by the Windows Mixed Reality home.

- "ALWAYS": Constantly loop an animation
- "HELD": Looped during the entire duration an object is grabbed.
- "GAZE": Looped while an object is being looked at
- "PROXIMITY": Looped while a viewer is near to an object
- "POINTING": Looped while a user is pointing at an object

Saving and exporting

Once you've made the changes to your glTF model you can save it directly as glTF or you can right click the name of the file in the editor and select "glTF: Export to GLB (binary file)" to instead export a .glb.

Restrictions

Animations cannot be longer than 20 minutes and cannot contain more than 36,000 keyframes (20 mins at 30 FPS). Additionally when using morph target based animations do not exceed 8192 morph target vertices or less. Exceeding these count will cause the animated asset to be unsupported in the Windows Mixed Reality home.

FEATURE	MAXIMUM
Duration	20 minutes
Keyframes	36,000
Morph Target Vertices	8192

glTF Implementation notes

Windows MR does not support flipping geometry using negative scales. Geometry with negative scales will likely result in visual artifacts.

The glTF asset MUST point to the default scene using the scene attribute to be rendered by Windows MR. Additionally the Windows MR glTF loader prior to the [Windows 10 April 2018 update](#) **requires** accessors:

- Must have min and max values.
- Type SCALAR must be componentType UNSIGNED_SHORT (5123) or UNSIGNED_INT (5125).
- Type VEC2 and VEC3 must be componentType FLOAT (5126).

The following material properties are used from core glTF 2.0 spec but not required:

- baseColorFactor, metallicFactor, roughnessFactor
- baseColorTexture: Must point to a texture stored in dds.
- emissiveTexture: Must point to a texture stored in dds.
- emissiveFactor
- alphaMode

The following material properties are ignored from core spec:

- All Multi-UVs
- metalRoughnessTexture: Must instead use Microsoft optimized texture packing defined below
- normalTexture: Must instead use Microsoft optimized texture packing defined below
- normalScale
- occlusionTexture: Must instead use Microsoft optimized texture packing defined below
- occlusionStrength

Windows MR does not support primitive mode lines and points.

Only a single UV vertex attribute is supported.

Additional resources

- [glTF Exporters and Converters](#)
- [glTF Toolkit](#)
- [glTF 2.0 Specification](#)
- [Microsoft glTF LOD Extension Specification](#)
- [PC Mixed Reality Texture Packing Extensions Specification](#)
- [HoloLens Mixed Reality Texture Packing Extensions Specification](#)
- [Microsoft DDS Textures glTF extensions specification](#)

See also

- [Implement 3D app launchers \(UWP apps\)](#)
- [Implement 3D app launchers \(Win32 apps\)](#)
- [Navigating the Windows Mixed Reality home](#)

Add custom home environments

11/6/2018 • 7 minutes to read • [Edit Online](#)

NOTE

This is an experimental feature. Give it a try and have fun with it, but don't be surprised if everything doesn't quite work as expected. We're evaluating the viability of this feature and interest in using it, so please tell us about your experience (and any bugs you've found) in the [developer forums](#).

Starting with the [Windows 10 April 2018 update](#), we've enabled an experimental feature that lets you add custom environments to the Places picker (on the Start menu) to use as the [Windows Mixed Reality home](#). Windows Mixed Reality has two default environments, Cliff House and Skyloft, that you can choose as your home. Creating custom environments allows you to expand that list with your own creations. We are making this available in an early state to evaluate interest from creators and developers, see what kinds of worlds you create, and understand how you work with different authoring tools.

When using a custom environment you'll notice that teleporting, interacting with apps, and placing holograms works just like it does in the Cliff House and Skyloft. You can browse the web in a fantasy landscape or fill a futuristic city with holograms - the possibilities are endless!

Device support

FEATURE	HOLOLENS	IMMERSIVE HEADSETS
Custom home environments		✓ <input type="checkbox"/>

Trying a sample environment

We've created a sample environment that shows off some of the creative possibilities of custom home environments. Follow these steps to try it out:

1. [Download our sample Fantasy Island environment](#) (link points to self-extracting executable).



Fantasy Island sample environment

- Run the **Fantasy_Island.exe** file you just downloaded.

NOTE

When attempting to run a .exe file downloaded from the web (like this one), you may encounter a "Windows protected your PC" pop-up. To run Fantasy_Island.exe from this pop-up, select **More info** and then **Run anyway**. This security setting is meant to protect you from downloading files you may not want to trust, so please only choose this option when you trust the source of the file.

- Open **File Explorer** and navigate to the environments folder by pasting the following in the address bar:

```
%LOCALAPPDATA%\Packages\EnvironmentsApp_cw5n1h2txyewy\LocalState
```

- Copy the sample environment that you downloaded into this folder.
- Restart **Mixed Reality Portal**. This will refresh the list of environments in the Places picker.
- Put on your headset. Once you're in the home, open the **Start menu** using the Windows button your controller.
- Select the **Places** icon above the list of pinned apps to choose a home environment.
- You will find the Fantasy Island environment that you downloaded in your list of places. Select **Fantasy Island** to enter your new custom home environment!

Creating your own custom environment

In addition to using our sample environments, you can export your own custom environments using your favorite 3D editing software.

Modeling guidelines

When modeling your environment, keep the following recommendations in mind. This will help ensure the user spawns in the correct orientation in a believably-sized world:

- Users will spawn at 0,0,0 so center your desired spawn location around the origin.
- Working Units should be set to meters so that assets can be authored at world scale.
- The Up axis should be set to "Y".
- The asset should face "forward" towards the positive Z axis.
- All meshes do not need to be combined, but it is recommended if you are targeting resource-constrained devices.

Exporting your environment

Windows Mixed Reality relies on binary glTF (.glb) as the asset delivery format for environments. glTF is a royalty free open standard for 3D asset delivery maintained by the Khronos group. As glTF evolves as an industry standard for interoperable 3D content, so will Microsoft's support for the format across Windows apps and experiences.

The first step in exporting assets to be used as custom home environments is generating a glTF 2.0 model. The glTF working group maintains a [list of supported exporters and converters](#) to create a glTF 2.0 model. To get started, use one of the programs listed on this page to create and export a glTF 2.0 model, or convert an existing model using one of the supported converters.

Additionally, check out [this helpful article](#) which provides an overview of an art workflow for exporting glTF models from Blender and 3DS Max directly.

Environment limits

All environments must be < 256 mbs. Environments larger than 256 mbs will fail to load and fall back to an empty world with just the default skybox surrounding the user. Please keep this file size limit in mind when creating your models. Additionally, if you plan to optimize your environment using the WindowsMRAssetConverter as described below, be cognizant that the texture size will increase as the optimizer creates textures that have a larger file size, but load faster.

Optimizing your environment

Windows Mixed Reality supports a number of optional optimizations that will significantly reduce the load time of your environments. This can be especially important for environments with many textures, as they will sometimes time out while loading. In general, we recommend this step for all assets, however, smaller environments with few or low-resolution textures won't always require it.

To make this process easier, we have created the [Windows Mixed Reality Asset Converter \(available on GitHub\)](#) to perform your optimizations. This tool uses a set of utilities available in the Microsoft glTF toolkit to optimize any standard 2.0 glTF or .glb by performing an additional texture packing, compression and resolution down-scaling.

The converter currently supports a number of flags to tweak the exact behavior of the optimizations. We recommend running with the following flags for best results:

FLAG	RECOMMENDED VALUE(S)	DESCRIPTION
-max-texture-size	1024 or 2048	Tweak this to improve the quality of the textures, default is 512x512. Note that a larger value will significantly impact the file size of the environment so keep the 256 mb limit in mind
-min-version	1803	Custom environments are only supported on versions of windows >= 1803. This flag will remove textures for older versions and reduce the file size of the final asset

For example:

```
WindowsMRAssetConverter FileToConvert.gltf -max-texture-size 1024 -min-version 1803
```

Testing your environment

Once you have your final .glb environment you're ready to test it out in the headset. Start at step 2 in the "[Trying a sample environment](#)" section to use your custom environment as the mixed reality home.

Feedback

While we're evaluating this experimental feature, we're interested in learning how you're using custom environments, any bugs you may encounter, and how you like the feature. Please share any and all feedback for creating and using custom home environments in the [developer forums](#).

Troubleshooting and tips

How do I change the name of the environment?

The file name in the environments folder will be used in the Places picker. To change the name of your environment simply rename the environment file name, and then restart Mixed Reality Portal.

How do I remove custom environments from my Places picker?

To remove a custom environment, open the environments folder on your PC (`%LOCALAPPDATA%\Packages\EnvironmentsApp_cw5n1h2txyewy\LocalState`) and delete the environment. Once you restart Mixed Reality Portal, this environment will no longer appear in the Places picker.

How do I default to my favorite custom environment?

You can't currently change the default environment. Each time you restart Mixed Reality Portal, you will be returned to the Cliff House environment.

I spawn into a blank space

Windows Mixed Reality [doesn't support environments that exceed 256 mb](#). When an environment exceeds this limit, you will land in the empty sky box with no model.

It takes a long time to load my environment

You can add optional optimizations to your environment to make it load faster. See "[Optimizing your environment](#)" for details.

The scale of my environment is incorrect

Windows Mixed Reality translates glTF units to 1 meter when loading environments. If your environment loads up an unexpected scale, double check your exporter to ensure that you're modeling at a 1 meter scale.

The spawn location in my environment is incorrect

The default spawn location is located at 0,0,0 in the environment. Its not currently possible to customize this location, so you must modify the spawn point by exporting your environment with the origin positioned at the desired spawn point.

The audio doesn't sound correct in the environment

When you create your custom environment, it will be using an acoustics rendering simulation that does not match the physical space you have created. Sound may come from the wrong directions and may sound muffled.

See also

- [Navigating the Windows Mixed Reality Home](#)
- [Windows Mixed Reality Asset Converter \(on GitHub\)](#)

Periodic Table of the Elements

11/6/2018 • 3 minutes to read • [Edit Online](#)

NOTE

This article discusses an exploratory sample we've created in the [Mixed Reality Design Labs](#), a place where we share our learnings about and suggestions for mixed reality app development. Our design-related articles and code will evolve as we make new discoveries.

[Periodic Table of the Elements](#) is a open-source sample app from Microsoft's Mixed Reality Design Labs. With this project, you can learn how to lay out an array of objects in 3D space with various surface types using an **Object collection**. Also learn how to create interactable objects that respond to standard inputs from HoloLens. You can use this project's components to create your own mixed reality app experience.



About the app

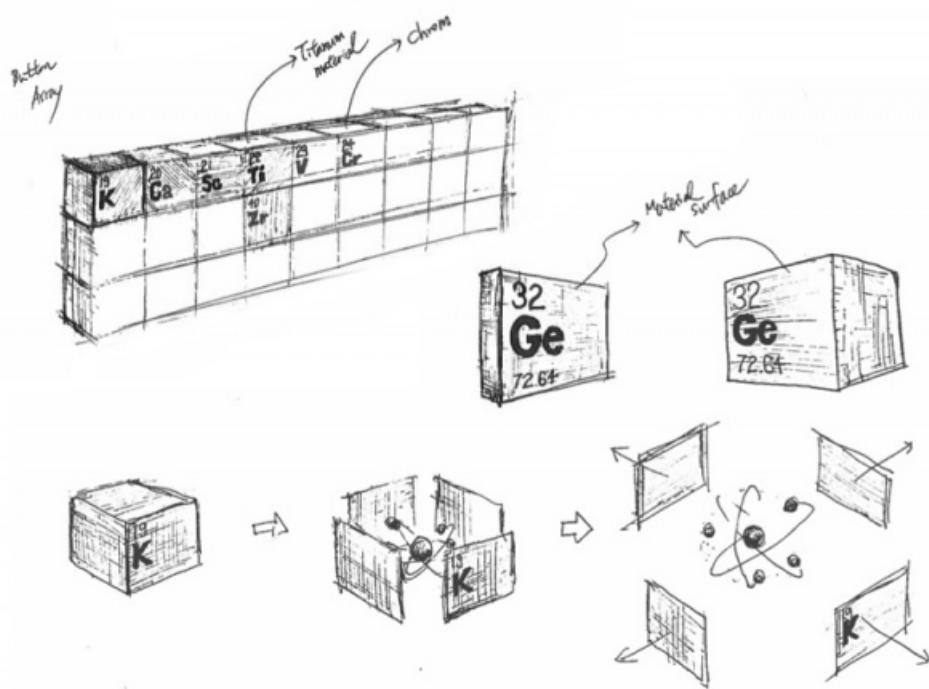
Periodic Table of the Elements visualizes the chemical elements and each of their properties in a 3D space. It incorporates the basic interactions of HoloLens such as gaze and air tap. Users can learn about the elements with animated 3D models. They can visually understand an element's electron shell and its nucleus - which is composed of protons and neutrons.

Background

After I first experienced HoloLens, a periodic table app was an idea I knew that I wanted to experiment with in mixed reality. Since each element has many data points that are displayed with text, I thought it would be great subject matter for exploring typographic composition in a 3D space. Being able to visualize the element's electron model was another interesting part of this project.

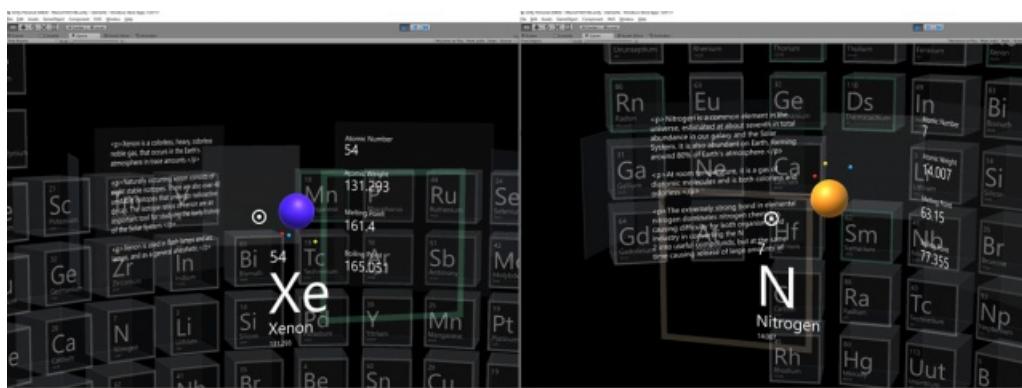
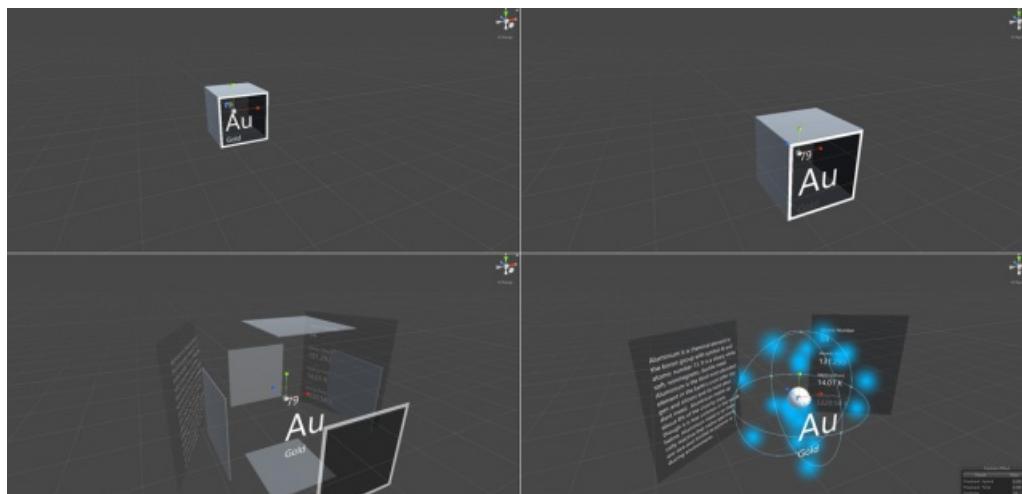
Design

For the default view of the periodic table, I imagined three-dimensional boxes that would contain the electron model of each element. The surface of each box would be translucent so that the user could get a rough idea of the element's volume. With gaze and air tap, the user could open up a detailed view of each element. To make the transition between table view and detail view smooth and natural, I made it similar to the physical interaction of a box opening in real life.



Design sketches

In detail view, I wanted to visualize the information of each element with beautifully rendered text in 3D space. The animated 3D electron model is displayed in the center area and can be viewed from different angles.



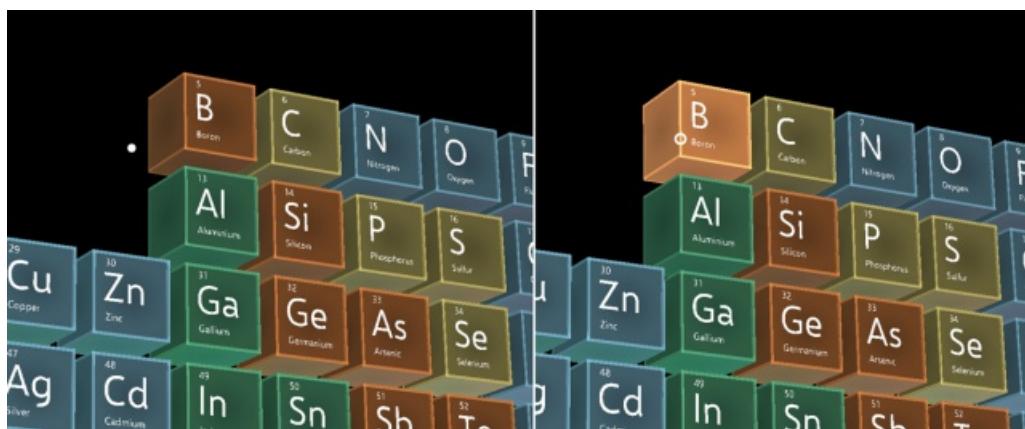
Interaction prototypes

The user can change the surface type by air tapping the buttons on the bottom of the table - they can switch between plane, cylinder, sphere and scatter.

Common controls and patterns used in this app

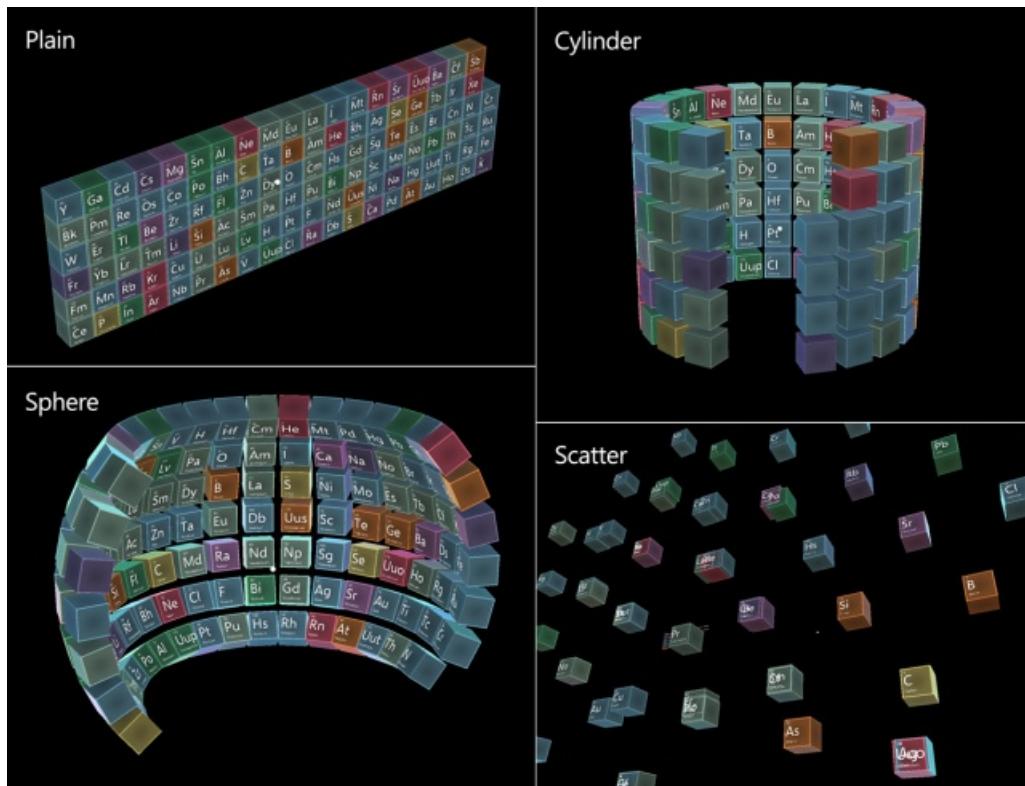
Interactable object (button)

[Interactable object](#) is an object which can respond to basic HoloLens inputs. It is provided as a prefab/script which you can easily apply to any object. For example, you can make a coffee cup in your scene interactable and respond to inputs such as gaze, air tap, navigation and manipulation gestures. [Learn more](#)



Object collection

[Object collection](#) is an object which helps you lay out multiple objects in various shapes. It supports plane, cylinder, sphere and scatter. You can configure additional properties such as radius, number of rows and the spacing. [Learn more](#)



Fitbox

By default, holograms will be placed in the location where the user is gazing at the moment the application is launched. This sometimes leads to unwanted result such as holograms being placed behind a wall or in the middle of a table. A fitbox allows a user to use gaze to determine the location where the hologram will be placed. It is made with a simple PNG image texture which can be easily customized with your own images or 3D objects.



Technical details

You can find scripts and prefabs for the Periodic Table of the Elements app on the [Mixed Reality Design Labs GitHub](#).

Application examples

Here are some ideas for what you could create by leveraging the components in this project.

Stock data visualization app

Using the same controls and interaction model as the Periodic Table of the Elements sample, you could build an app which visualizes stock market data. This example uses the Object collection control to lay out stock data in a spherical shape. You can imagine a detail view where additional information about each stock could be displayed in an interesting way.

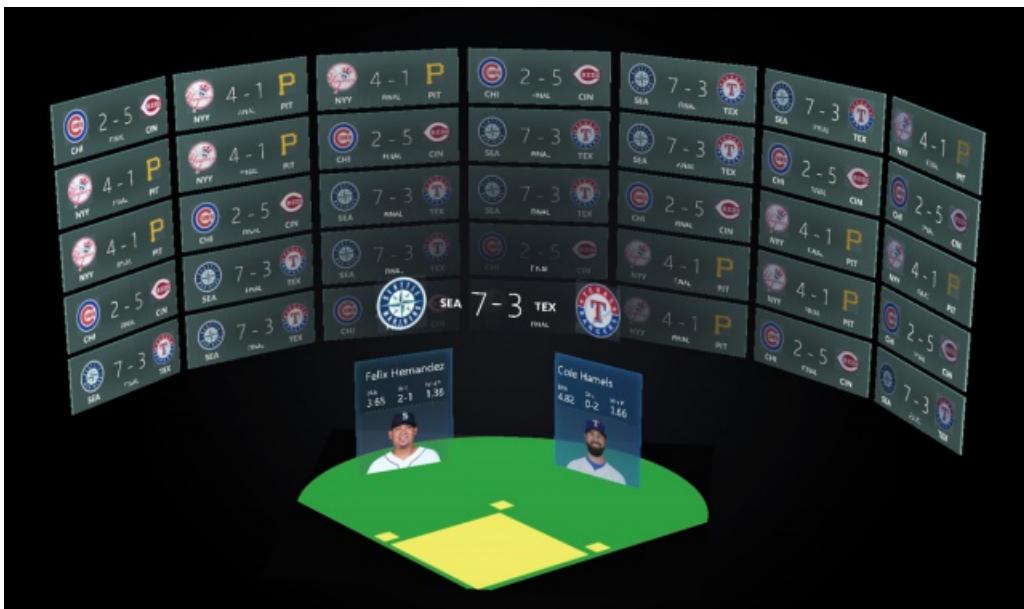




An example of how the Object collection used in the Periodic Table of the Elements sample app could be used in a finance app

Sports app

This is an example of visualizing sports data using Object collection and other components from the Periodic Table of the Elements sample app.



An example of how the Object collection used in the Periodic Table of the Elements sample app could be used in a sports app

About the author



Dong Yoon Park
UX Designer @Microsoft

See also

- [Interactable object](#)
- [Object collection](#)

Lunar Module

11/6/2018 • 8 minutes to read • [Edit Online](#)

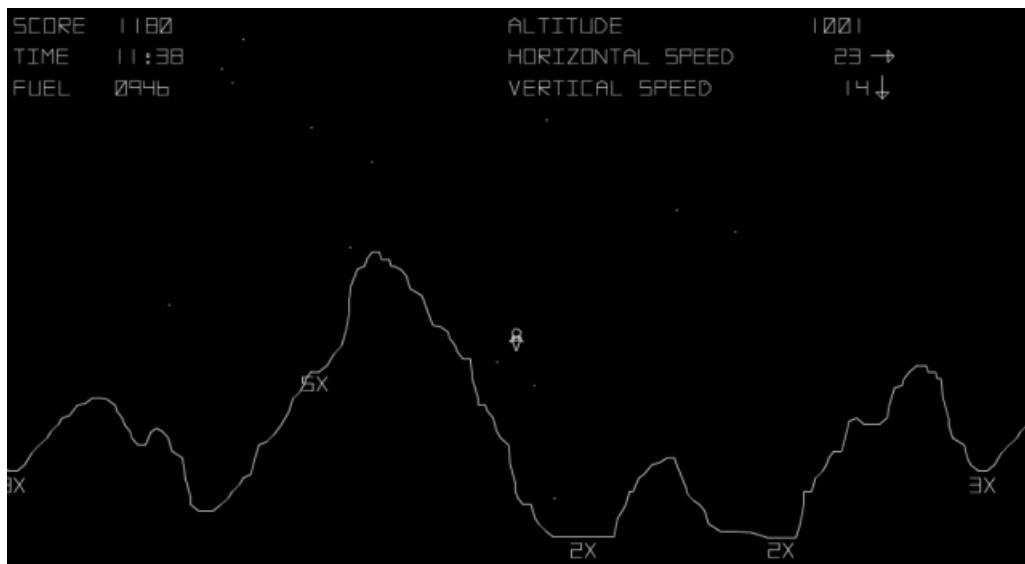
NOTE

This article discusses an exploratory sample we've created in the [Mixed Reality Design Labs](#), a place where we share our learnings about and suggestions for mixed reality app development. Our design-related articles and code will evolve as we make new discoveries.

[Lunar Module](#) is an open-source sample app from Microsoft's Mixed Reality Design Labs. With this project, you can learn how to extend Hololens' base gestures with two-handed tracking and Xbox controller input, create objects that are reactive to surface mapping and plane finding and implement simple menu systems. All of the project's components are available for use in your own mixed reality app experiences.

Rethinking classic experiences for Windows Mixed Reality

High up in the atmosphere, a small ship reminiscent of the Apollo module methodically surveys jagged terrain below. Our fearless pilot spots a suitable landing area. The descent is arduous but thankfully, this journey has been made many times before...



Original interface from Atari's 1979 Lunar Lander

[Lunar Lander](#) is an arcade classic where players attempt to pilot a moon lander onto a flat spot of lunar terrain. Anyone born in the 1970s has most likely spent hours in an arcade with their eyes glued to this vector ship plummeting from the sky. As a player navigates their ship toward a desired landing area the terrain scales to reveal progressively more detail. Success means landing within the safe threshold of horizontal and vertical speed. Points are awarded for time spent landing and remaining fuel, with a multiplier based on the size of the landing area.

Aside from the gameplay, the arcade era of games brought constant innovation of control schemes. From the simplest 4-way joystick and button configurations (seen in the iconic [Pac-Man](#)) to the highly specific and complicated schemes seen in the late 90s and 00s (like those in golf simulators and rail shooters). The input scheme used in the Lunar Lander machine is particularly intriguing for two reasons: curb appeal and immersion.



Atari's Lunar Lander arcade console

Why did Atari and so many other game companies decide to rethink input?

A kid walking through an arcade will naturally be intrigued by the newest, flashiest machine. But Lunar Lander features a novel input mechanic that stood out from the crowd.

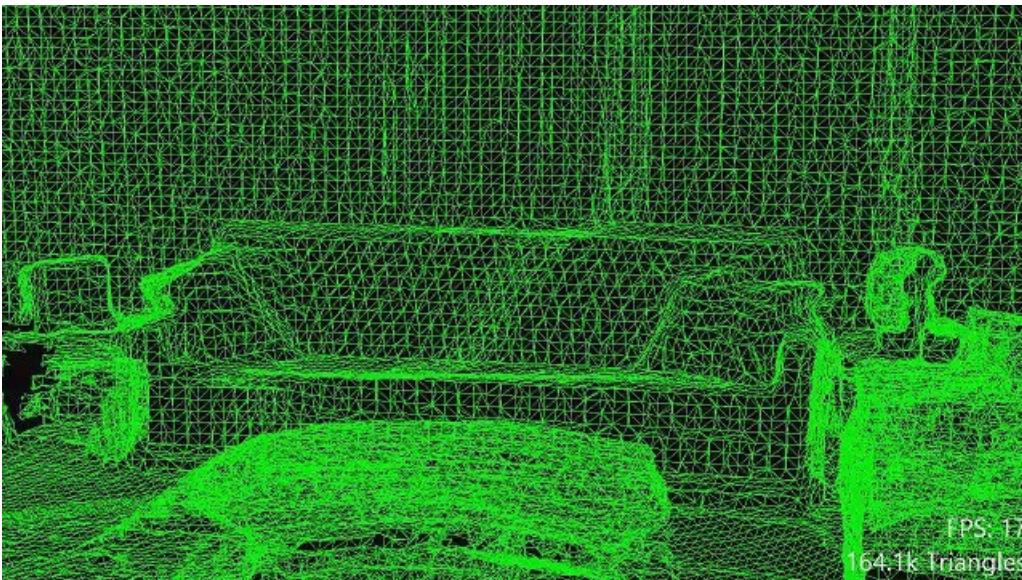
Lunar Lander uses two buttons for rotating the ship left and right and a **thrust lever** to control the amount of thrust the ship produces. This lever gives users a certain level of finesse a regular joystick can't provide. It also happens to be a component common to modern aviation cockpits. Atari wanted Lunar Lander to immerse the user in the feeling that they were in fact piloting a lunar module. This concept is known as **tactile immersion**.

Tactile immersion is the experience of sensory feedback from performing repetitive actions. In this case, the repetitive action of adjusting the throttle lever and rotation which our eyes see and our ears hear, helps connect the player to the act of landing a ship on the moon's surface. This concept can be tied to the psychological concept of "flow." Where a user is fully absorbed in a task that has the right mixture of challenge and reward, or put more simply, they're "in the zone."

Arguably, the most prominent type of immersion in mixed reality is spatial immersion. The whole point of mixed reality is to fool ourselves into believing these digital objects exist in the real world. We're synthesizing holograms in our surroundings, spatially immersed in entire environments and experiences. This doesn't mean we can't still employ other types of immersion in our experiences just as Atari did with tactile immersion in Lunar Lander.

Designing with immersion

How might we apply tactile immersion to an updated, volumetric sequel to the Atari classic? Before tackling the input scheme, the game construct for 3-dimensional space needs to be addressed.



Visualizing spatial mapping in HoloLens

By leveraging a user's surroundings, we effectively have infinite terrain options for landing our lunar module. To make the game most like the original title, a user could potentially manipulate and place landing pads of varying difficulties in their environment.



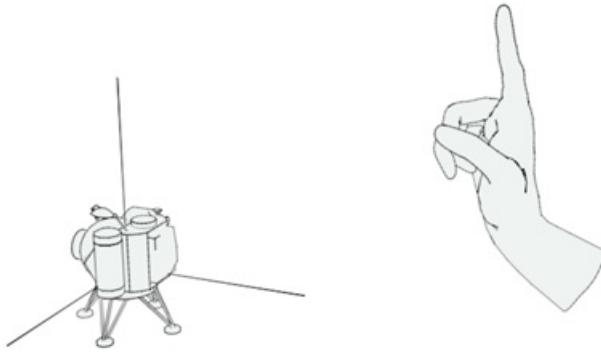
Flying the Lunar Module

Requiring the user to learn the input scheme, control the ship, and have a small target to land on is a lot to ask. A successful game experience features the right mix of challenge and reward. The user should be able to choose a level of difficulty, with the easiest mode simply requiring the user to successfully land in a user-defined area on a surface scanned by the HoloLens. Once a user gets the hang of the game, they can then crank up the difficulty as they see fit.

Adding input for hand gestures

HoloLens base input has only two gestures - [Air Tap and Bloom](#). Users don't need to remember contextual nuances or a laundry list of specific gestures which makes the platform's interface both versatile and easy to learn. While the system may only expose these two gestures, HoloLens as a device is capable of tracking two hands at once. Our ode to Lunar Lander is an [immersive app](#) which means we have the ability to extend the base set of gestures to leverage two hands and add our own delightfully tactile means for lunar module navigation.

Looking back at the original control scheme, **we needed to solve for thrust and rotation**. The caveat is rotation in the new context adds an additional axis (technically two, but the Y axis is less important for landing). The two distinct ship movements naturally lend themselves to be mapped to each hand:



Tap and drag gesture to rotate lander on all three axes

Thrust

The lever on the original arcade machine mapped to a scale of values, the higher the lever was moved the more thrust was applied to the ship. An important nuance to point out here is the user can take their hand off of the control and maintain a desired value. We can effectively use tap-and-drag behavior to achieve the same result. The thrust value starts at zero. The user taps and drags to increase the value. At that point they could let go to maintain it. Any tap-and-drag gesture value change would be the delta from the original value.

Rotation

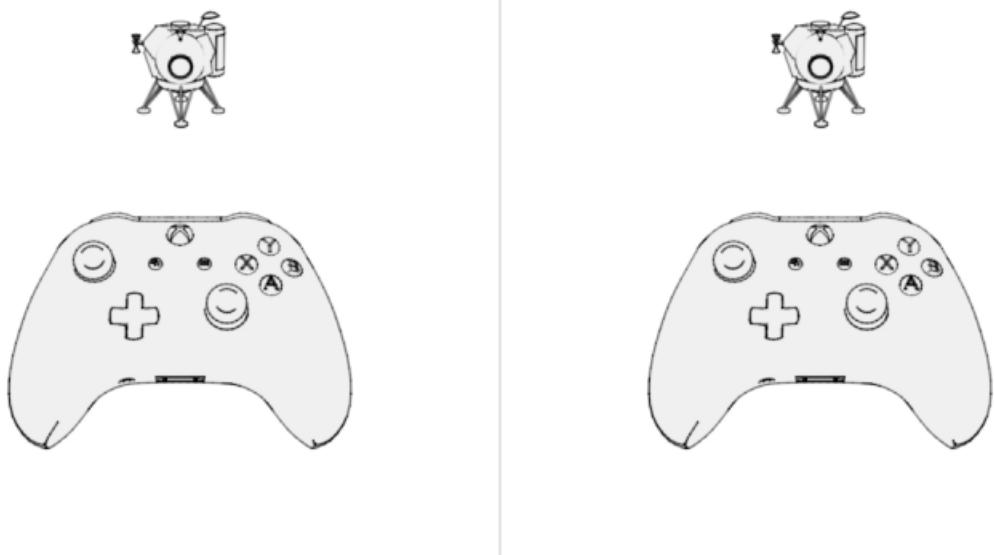
This is a little more tricky. Having holographic "rotate" buttons to tap makes for a terrible experience. There isn't a physical control to leverage, so the behavior must come from manipulation of an object representing the lander, or with the lander itself. We came up with a method using tap-and-drag which enables a user to effectively "push and pull" it in the direction they want it to face. Any time a user taps and holds, the point in space where the gesture was initiated becomes the origin for rotation. Dragging from the origin converts the delta of the hand's translation (X,Y,Z) and applies it to the delta of the lander's rotation values. Or more simply, *dragging left <-> right, up <-> down, forward <-> back in spaces rotates the ship accordingly*.

Since the HoloLens can track two hands, rotation can be assigned to the right hand while thrust is controlled by the left. Finesse is the driving factor for success in this game. The *feel* of these interactions is the absolute highest priority. Especially in context of tactile immersion. A ship that reacts too quickly would be unnecessarily difficult to steer, while one too slow would require the user to "push and pull" on the ship for an awkwardly long amount of time.

Adding input for game controllers

While hand gestures on the HoloLens provide a novel method of fine-grain control, there is still a certain lack of 'true' tactile feedback that you get from analog controls. Connecting an Xbox game controller allows to bring back this sense of physicality while leveraging the control sticks to retain fine-grain control.

There are multiple ways to apply the relatively straight-forward control scheme to the Xbox controller. Since we're trying to stay as close to the original arcade set up as possible, **Thrust** maps best to the trigger button. These buttons are analog controls, meaning they have more than simple *on and off* states, they actually respond to the degree of pressure put on them. This gives us a similar construct as the **thrust lever**. Unlike the original game and the hand gesture, this control will cut the ship's thrust once a user stops putting pressure on the trigger. It still gives the user the same degree of finesse as the original arcade game did.



Left thumbstick is mapped to yaw and roll; right thumbstick is mapped to pitch and roll

The dual thumbsticks naturally lend themselves to controlling ship rotation. Unfortunately, there are 3 axes on which the ship can rotate and two thumbsticks which both support two axes. This mismatch means either one thumbstick controls one axis; or there is overlap of axes for the thumbsticks. The former solution ended up feeling "broken" since thumbsticks inherently blend their local X and Y values. The latter solution required some testing to find which redundant axes feel the most natural. The final sample uses *yaw* and *roll* (Y and X axes) for the left thumbstick, and *pitch* and *roll* (Z and X axes) for the right thumbstick. This felt the most natural as *roll* seems to independently pair well with *yaw* and *pitch*. As a side note, using both thumbsticks for *roll* also happens to double the rotation value; it's pretty fun to have the lander do loops.

This sample app demonstrates how spatial recognition and tactile immersion can significantly change an experience thanks to Windows Mixed Reality's extensible input modalities. While Lunar Lander may be nearing 40 years in age, the concepts exposed with that little octagon-with-legs will live on forever. When imagining the future, why not look at the past?

Technical details

You can find scripts and prefabs for the Lunar Module sample app on the [Mixed Reality Design Labs GitHub](#).

About the author



Addison Linville
UX Designer @Microsoft

See also

- [Motion controllers](#)
- [Gestures](#)
- [Types of mixed reality apps](#)

Galaxy Explorer

11/6/2018 • 3 minutes to read • [Edit Online](#)

You shared your ideas. We're sharing the code.

The Galaxy Explorer Project is ready. You shared your ideas with the community, chose an app, watched a team build it, and can now get the source code. If you have a device, Galaxy Explorer Project is also available for download from the Windows Store for Microsoft HoloLens.

TIP

[Get the code on GitHub](#)

Our HoloLens [development team](#) of designers, artists, and developers built Galaxy Explorer and invited all of you to be part of this journey with them. After six weeks of core development and two weeks of refinement, this app is now ready for you! You can also follow along our whole journey through the video series below.

Share your idea

The Galaxy Explorer journey begins with the "Share your idea" campaign.

The Microsoft HoloLens community is bursting with spectacular ideas for how holographic computing will transform our world. We believe the most incredible HoloLens apps will come out of ideas you imagine together.

You shared over 5000 amazing ideas throughout those few weeks! Our development team reviewed the most successful and viable ideas and offered to build one of the top three ideas.

After a 24-hour Twitter poll, Galaxy Explorer was the winning idea! Our HoloLens development team of designers, artists, and developers built Galaxy Explorer and invited all of you to be part of this journey with them. You can follow the development process in the videos below.

Ep 1: Trust the Process

In Episode 1, the development team begins the creative process: brainstorming, conceiving, and deciding what to prototype.

Ep 2: Let's Do This

In Episode 2, the development team completes the prototyping phase – tackling hard problems and figuring out which ideas to pursue further.

Ep 3: Laying Foundations

In Episode 3, the team starts the first week of development – creating a plan, writing production code, creating art assets, and figuring out the user interface.

Ep 4: Make It Real

In Episode 4, the team dives deeper into development – bringing in scientific data, optimizing the rendering process, and incorporating spatial mapping.

Ep 5: See What Happens

In Episode 5, the development team tests the app, searches for bugs that need to be fixed, and refines the experience.

Ep 6: Coming to Life

In Episode 6, the team finishes the last week of development, prepares for two weeks of polish work, and reflects on the progress they've made

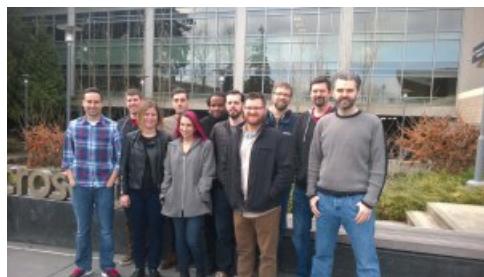
Ep 7: The Final Product

In Episode 7, the team completes the project and shares their code.

Case study

You can find even more insights and lessons from developing Galaxy Explorer by reading the "[Creating a galaxy in mixed reality](#)" case study.

Meet the team



Galaxy Explorer development team

We learned the building the right team is one of the most important investments we could make. We decided to organize similarly to a game studio for those of you familiar with that development model. We chose to have eleven core team members to control scope, since we had a fixed timeframe (create something cool before Build on March 30).

For this project, we started with a producer, Jessica who conducted planning, reviewing progress, and keeping things running day to day. She's the one with pink hair. We had a design director (Jon) and a senior designer (Peter). They held the creative vision for Galaxy Explorer. Jon is the one in glasses in the front row, and Peter is the second from the right in the back.

We had three developers – BJ (between Jon and Jessica), Mike (second row on the left), and Karim (second row middle, next to BJ). They figured out the technical solutions needed to realize that creative vision.

We started out with four artists full-time – a concept artist (Jedd, second from left in the back), a modeler (Andy, third from right in the back), a technical artist (Alex (right-most person)) and an animator (Steve (left-most person)). Each of them does more than that, too – but those are their primary responsibilities.

We'll had one full-time tester – Lena – who tested our builds every day, set up our build reviews, and reviewed features as they come online. Everyone tested constantly though, as we were always looking at our builds. Lena's the one rocking the leather jacket.

We are all a part of a larger studio here at Microsoft (think team in non-game development). There were a bunch of other people involved as well – we called on the talents of our art director, audio engineer and studio leadership frequently throughout the project, but those folks were shared resources with other projects our broader team has.

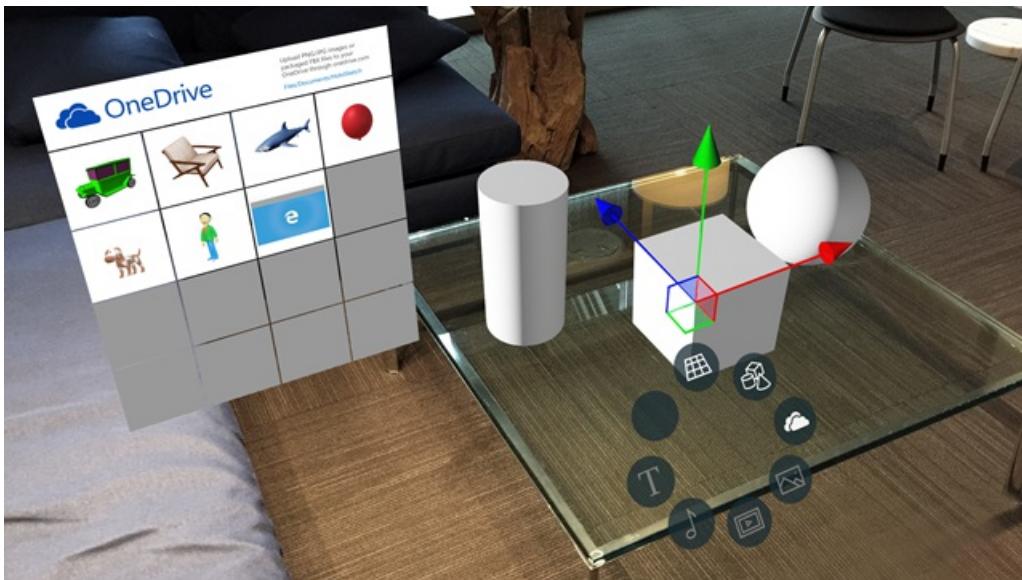
See also

- [Case study - Creating a galaxy in mixed reality](#)
- [Galaxy Explorer GitHub repo](#)

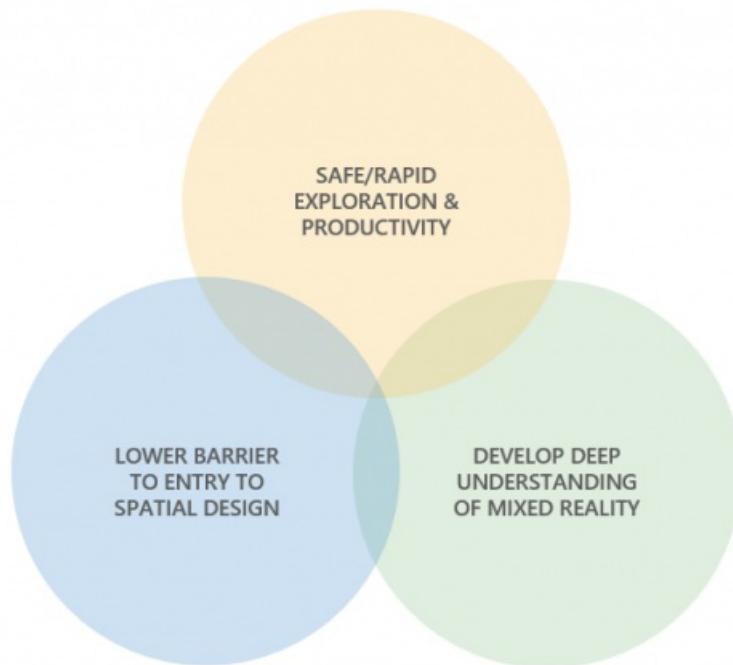
Case study - Building HoloSketch, a spatial layout and UX sketching app for HoloLens

11/6/2018 • 4 minutes to read • [Edit Online](#)

HoloSketch is an on-device spatial layout and UX sketching tool for HoloLens to help build holographic experiences. HoloSketch works with a paired Bluetooth keyboard and mouse as well as gesture and voice commands. The purpose of HoloSketch is to provide a simple UX layout tool for quick visualization and iteration.



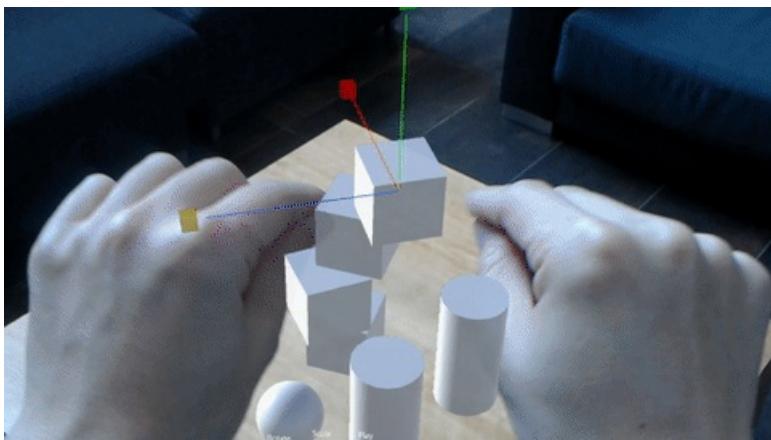
HoloSketch: spatial layout and UX sketching app for HoloLens



A simple UX layout tool for quick visualization and iteration

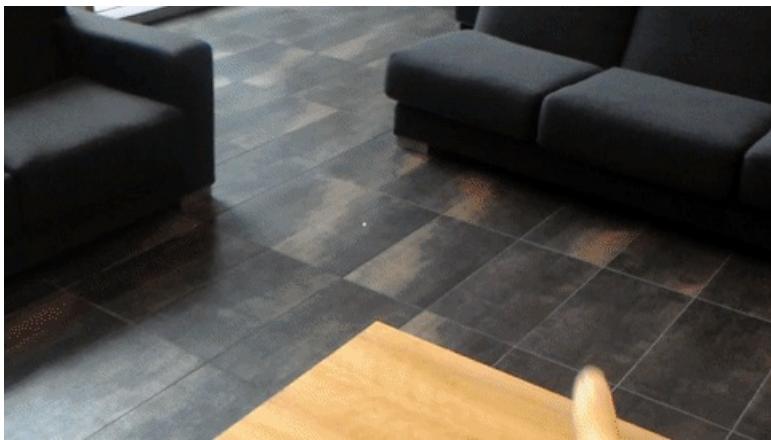
Features

Primitives for quick studies and scale-prototyping



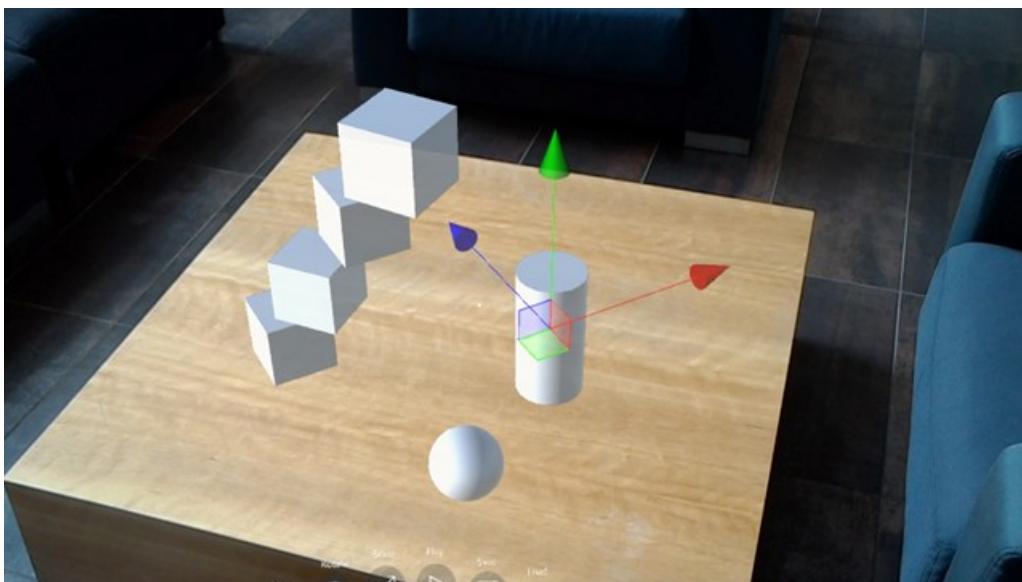
Use primitive shapes for quick massing studies and scale-prototyping.

Import objects through OneDrive



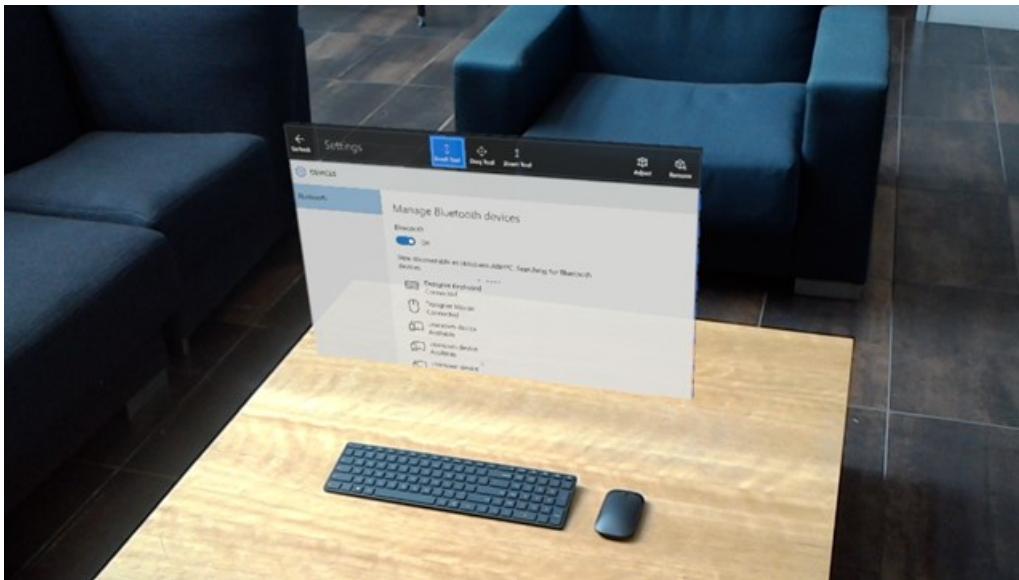
Import PNG/JPG images or 3D FBX objects(requires packaging in Unity) into the mixed reality space through OneDrive.

Manipulate objects



Manipulate objects (move/rotate/scale) with a familiar 3D object interface.

Bluetooth, mouse/keyboard, gestures and voice commands



HoloSketch supports Bluetooth mouse/keyboard, gestures and voice commands.

Background

Importance of experiencing your design in the device

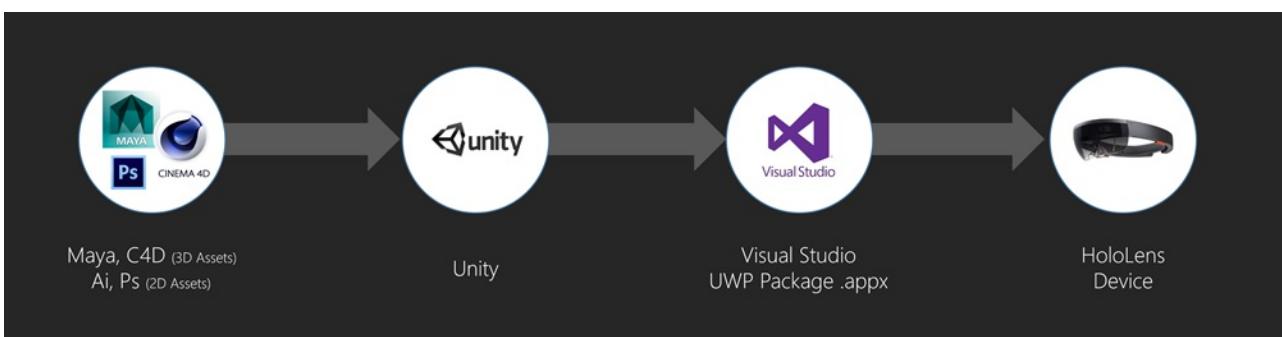
When you design something for HoloLens, it is important to experience your design in the device. One of the biggest challenges in mixed reality app design is that it is hard to get the sense of scale, position and depth, especially through traditional 2D sketches.

Cost of 2D based communication

To effectively communicate UX flows and scenarios to others, a designer may end up spending a lot of time creating assets using traditional 2D tools such as Illustrator, Photoshop and PowerPoint. These 2D designs often require additional effort to convert them into 3D. Some ideas are lost in this translation from 2D to 3D.

Complex deployment process

Since mixed reality is a new canvas for us, it involves a lot of design iteration and trial and error by its nature. For designer who are not familiar with tools such as Unity and Visual Studio, it is not easy to put something together in HoloLens. Typically you have to go through the process below to see your 2D/3D artwork in the device. This was a big barrier for designers iterating on ideas and scenarios quickly.



Deployment process

Simplified process with HoloSketch

With HoloSketch, we wanted to simplify this process without involving development tools and device portal pairing. Using OneDrive, users can easily put 2D/3D assets into HoloLens.



Simplified process with HoloSketch

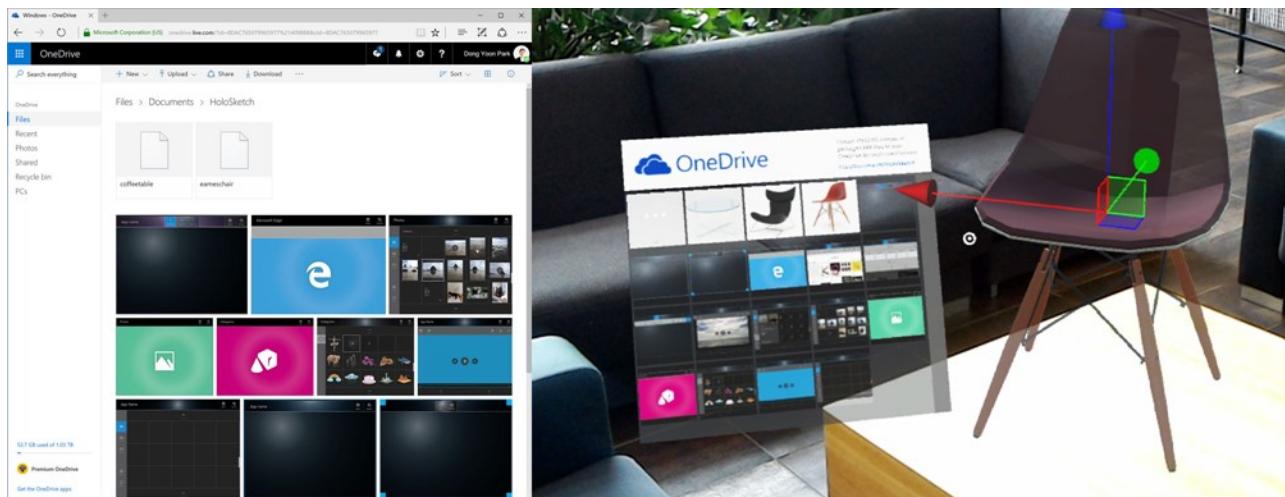
Encouraging three-dimensional design thinking and solutions

We thought this tool would give designers an opportunity to explore solutions in a truly three-dimensional space and not be stuck in 2D. They don't have to think about creating a 3D background for their UI since the background is the real world in the case of Hololens. HoloSketch opens up a way for designers to easily explore 3D design on Hololens.

Get Started

How to import 2D images (JPG/PNG) into HoloSketch

- Upload JPG/PNG images to your OneDrive folder 'Documents/HoloSketch'.
- From the OneDrive menu in HoloSketch, you will be able to select and place the image in the environment.



Importing images and 3D objects through OneDrive

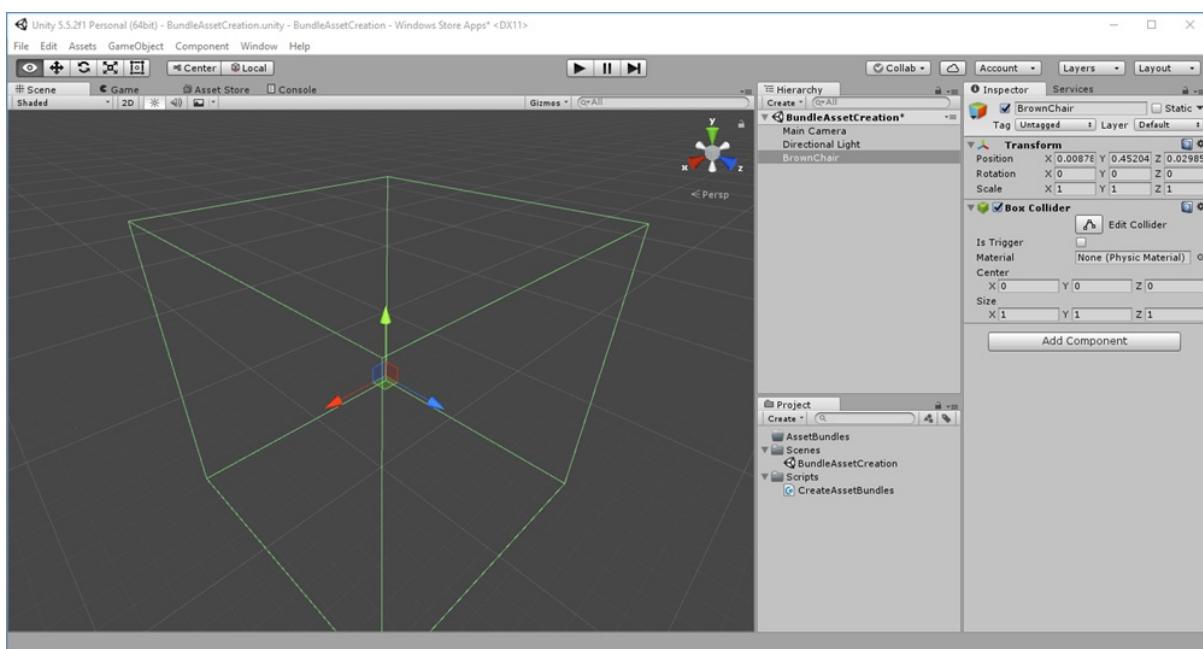
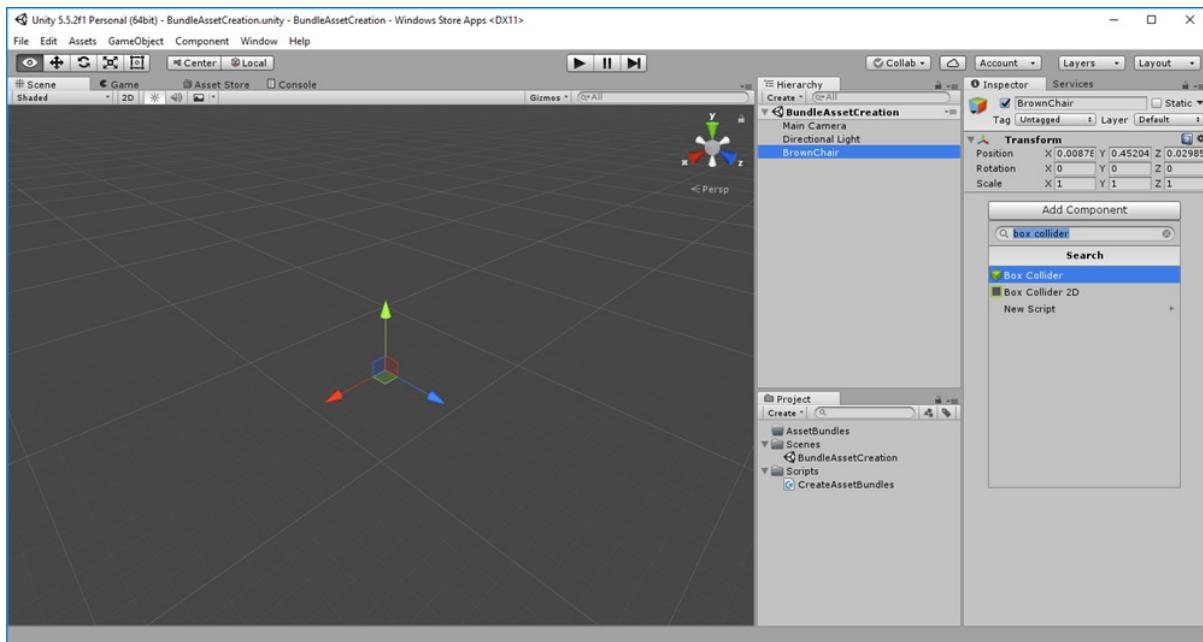
How to import 3D objects into HoloSketch

Before uploading to your OneDrive folder, please follow the steps below to package your 3D objects into a Unity asset bundle. Using this process you can import your FBX/OBJ files from 3D software such as Maya, Cinema 4D and Microsoft Paint 3D.

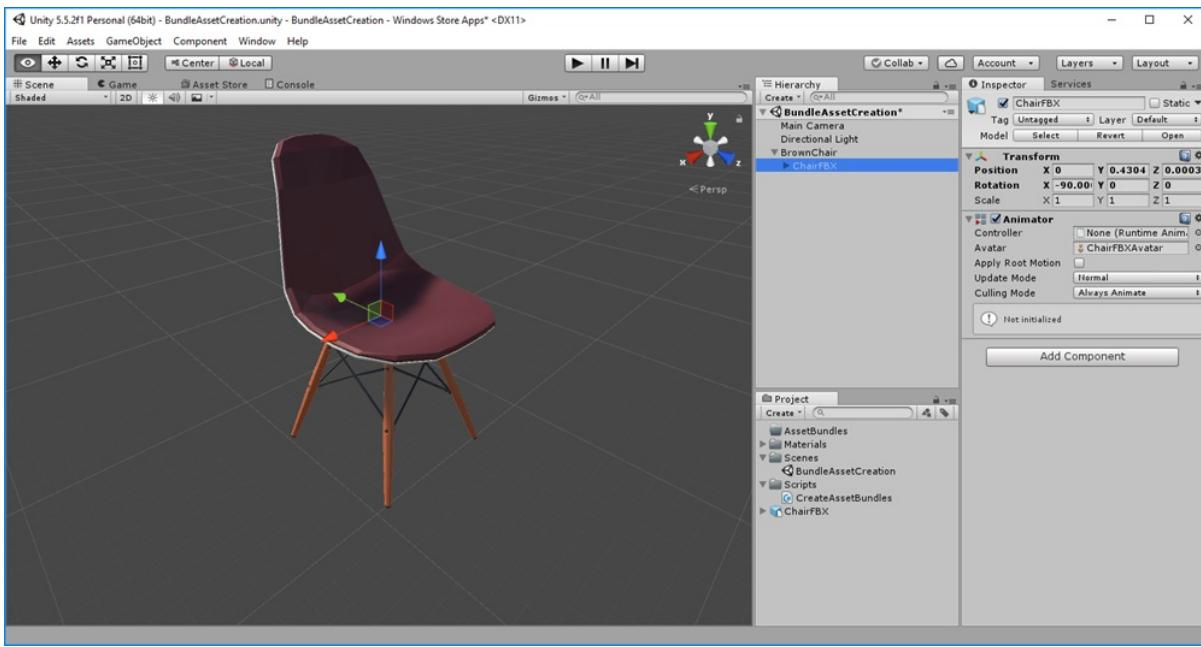
IMPORTANT

Currently, asset bundle creation is supported with Unity version 5.4.5f1.

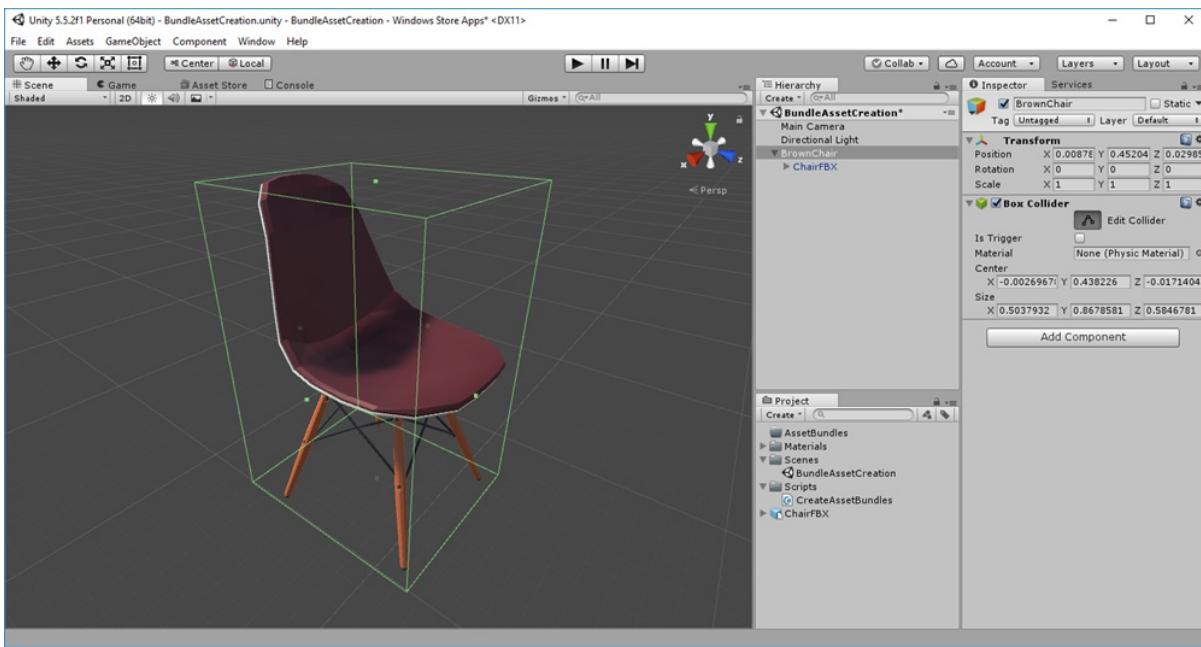
1. Download and open the Unity project '[AssetBuilder_Unity](#)'. This Unity project includes the script for the bundle asset generation.
2. Create a new GameObject.
3. Name the GameObject based on the contents.
4. In the Inspector panel, click 'Add Component' and add 'Box Collider'.



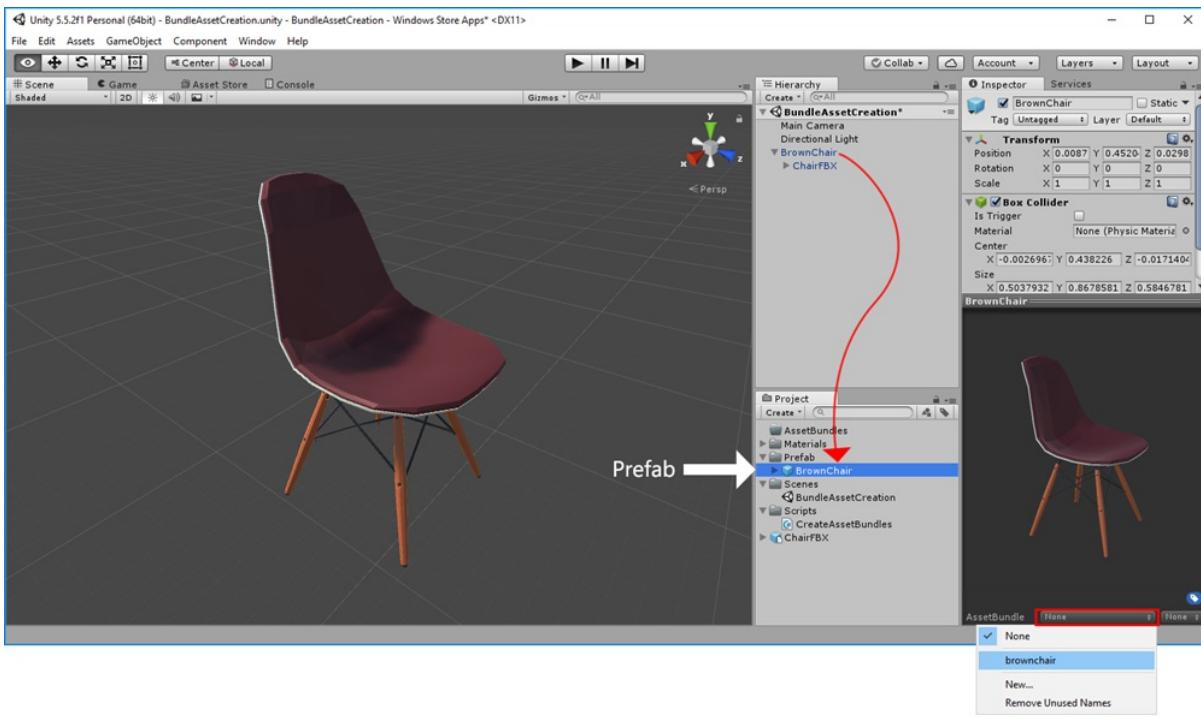
5. Import the 3D FBX file by dragging it into the project panel.
6. Drag the object into the Hierarchy panel **under your new GameObject**.



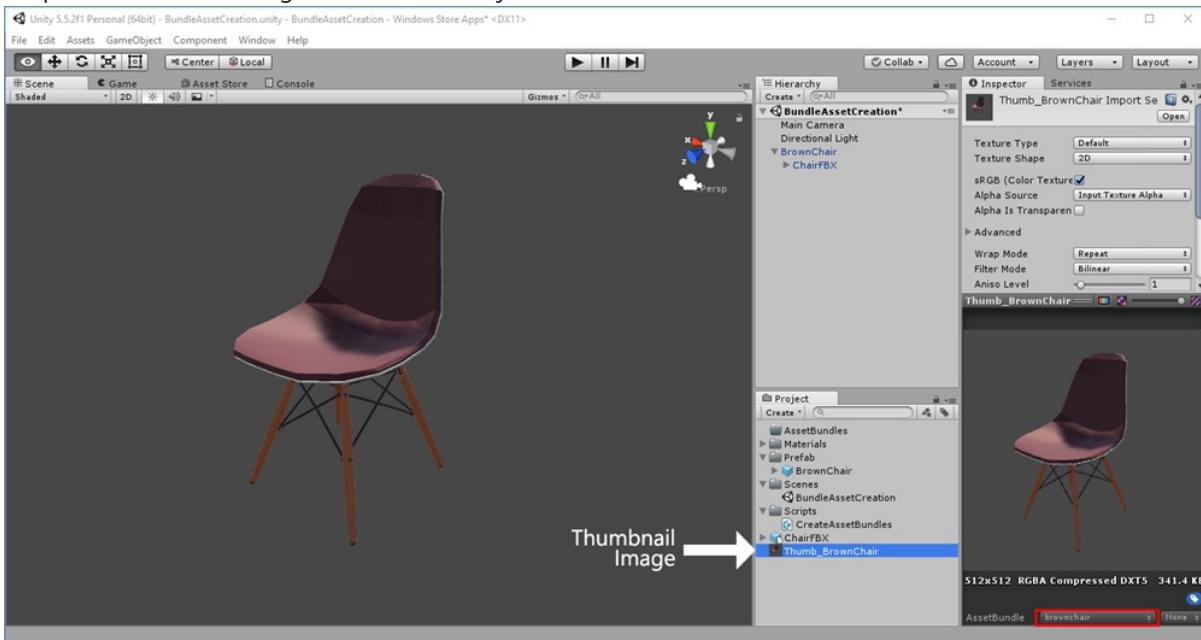
7. Adjust the collider dimension if it does not match the object. Rotate the object to face **Z-axis**.



8. Drag the object from the Hierarchy panel to the Project panel to **make it prefab**.
9. On the bottom of the inspector panel, click the dropdown, create and assign a new unique name. Below example shows adding and assigning 'brownchair' for the AssetBundle Name.

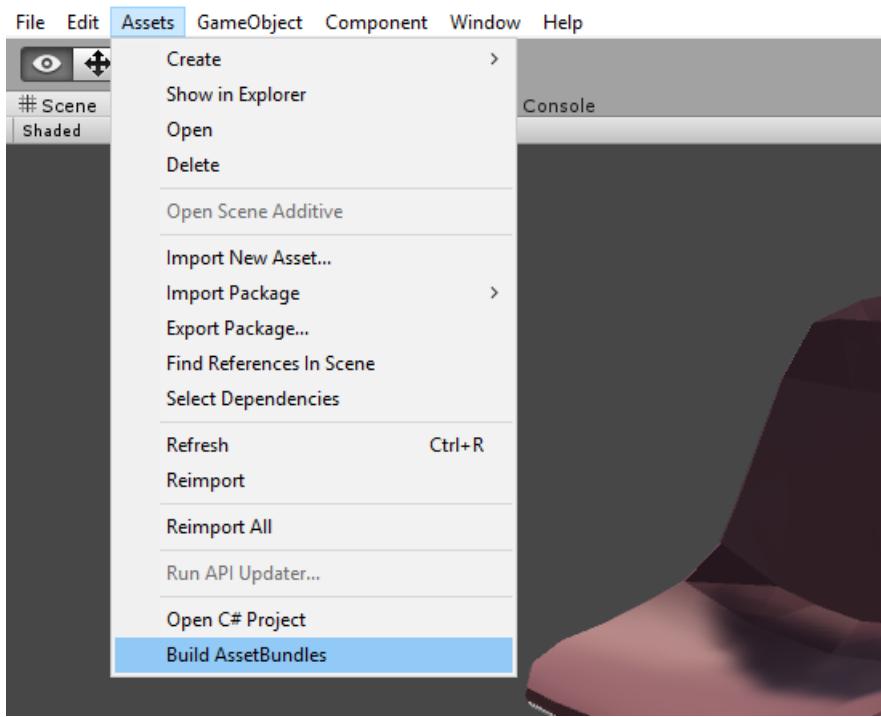


10. Prepare a thumbnail image for the model object.

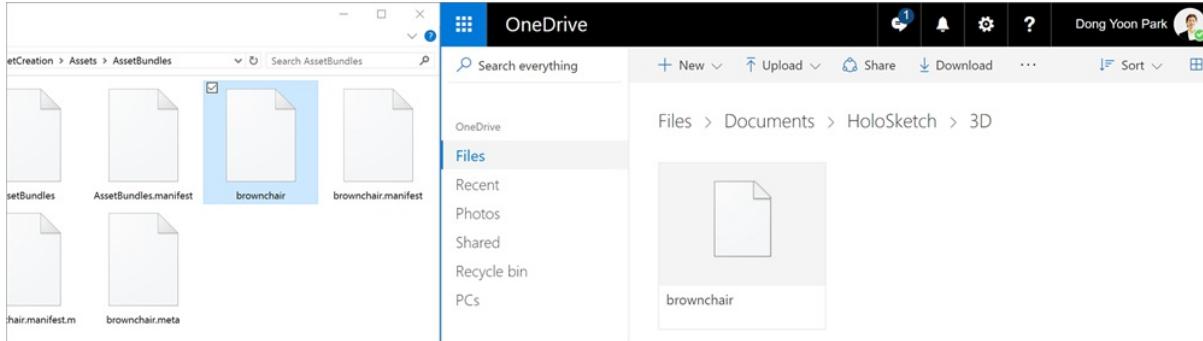


11. Create a folder named 'Assetbundles' under the Unity project's 'Asset' folder.

12. From the Assets menu, select 'Build AssetBundles' to generate file.



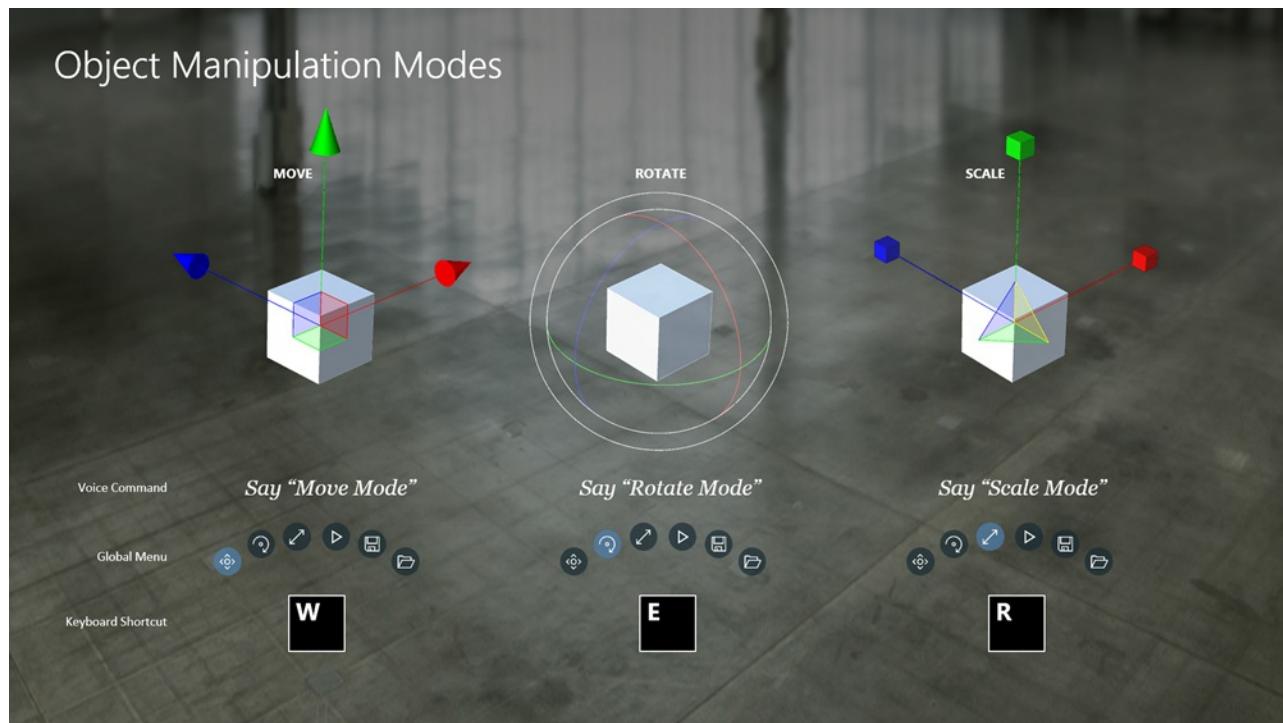
13. **Upload the generated file to the /Files/Documents/HoloSketch folder on OneDrive.** Upload the asset_unique_name file only. You don't need to upload manifest files or AssetBundles file.



How to manipulate the objects

HoloSketch supports the traditional interface that is widely used in 3D software. You can use move, rotate, scale the objects with gestures and a mouse. You can quickly switch between different modes with voice or keyboard.

Object manipulation modes



How to manipulate the objects

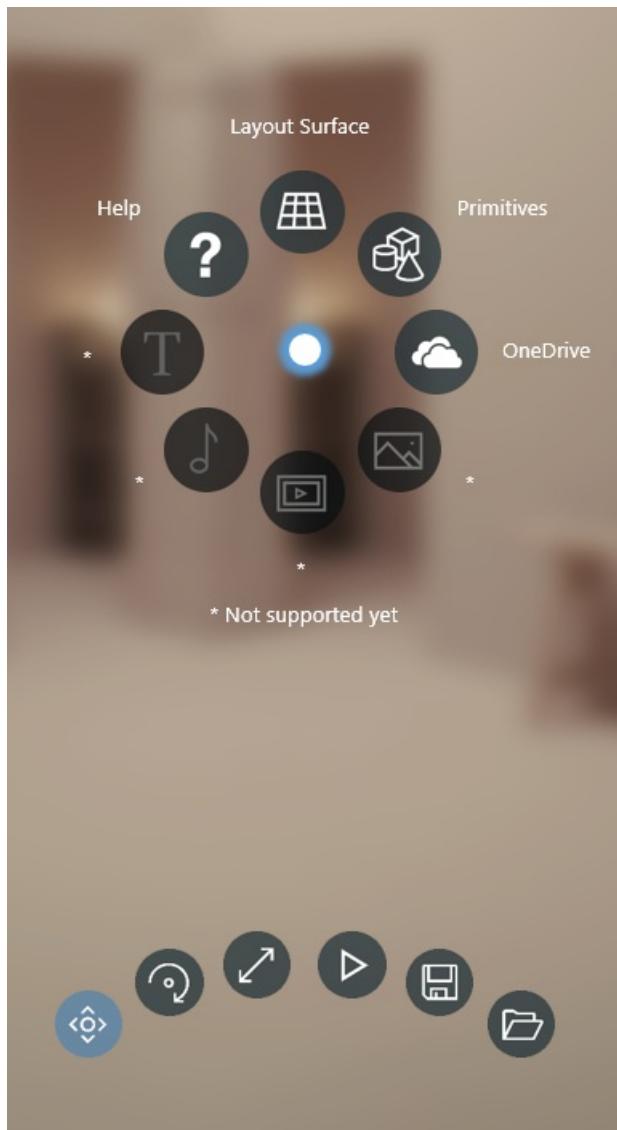
Contextual and Tool Belt menus

Using the Contextual Menu

Double air tap to open the Contextual Menu.

Menu items:

- **Layout Surface:** This is a 3D grid system where you can layout multiple objects and manage them as a group.
Double air-tap on the Layout Surface to add objects to it.
- **Primitives:** Use cubes, spheres, cylinders and cones for massing studies.
- **OneDrive:** Open the OneDrive menu to import objects.
- **Help:** Displays help screen.



Contextual menu

Using the Tool Belt Menu

Move, Rotate, Scale, Save, and Load Scene are available from the Tool Belt Menu.

Using keyboard, gestures and voice commands

⌨️ Keyboard shortcuts

W	Move Mode	CTRL + C	Copy
E	Rotate Mode	CTRL + V	Paste
R	Scale Mode	CTRL + U	Undo
H	Show Help	CTRL + Y	Redo
J	Hide Help	DEL	Delete Object

🎙 Voice commands



“move mode” “rotate mode” “scale mode”

↗️ Gestures



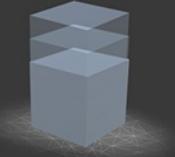
Double tap to open the main menu



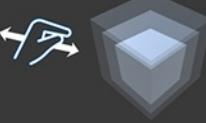
“delete object”



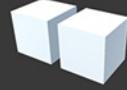
“turn to me”



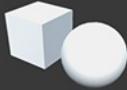
“snap to surface”



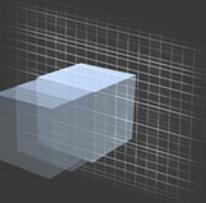
In scale mode use two hands for uniform scaling



“copy object”
“paste object”



“add cube”
“add sphere”
“add plane”



“snap to grid”
“snap to object”

Keyboard, gestures, and voice commands

Download the app



[Download and install the HoloSketch app for free from the Microsoft Store](#)

Known issues

- Currently asset bundle creation is supported with **Unity version 5.4.5f1**.
- Depending on the amount of data in your OneDrive, the app might appear as if it has stopped while loading OneDrive contents
- Currently, Save and Load feature only supports primitive objects
- Text, Sound, Video and Photo menus are disabled on the contextual menu
- The Play button on the Tool Belt menu clears the manipulation gizmos

Sharing your sketches

You can use the video recording feature in HoloLens by saying 'Hey Cortana, Start/Stop recording'. Press the volume up/down key together to take a picture of your sketch.

About the authors



Dong Yoon Park

UX Designer @Microsoft



Patrick Sebring

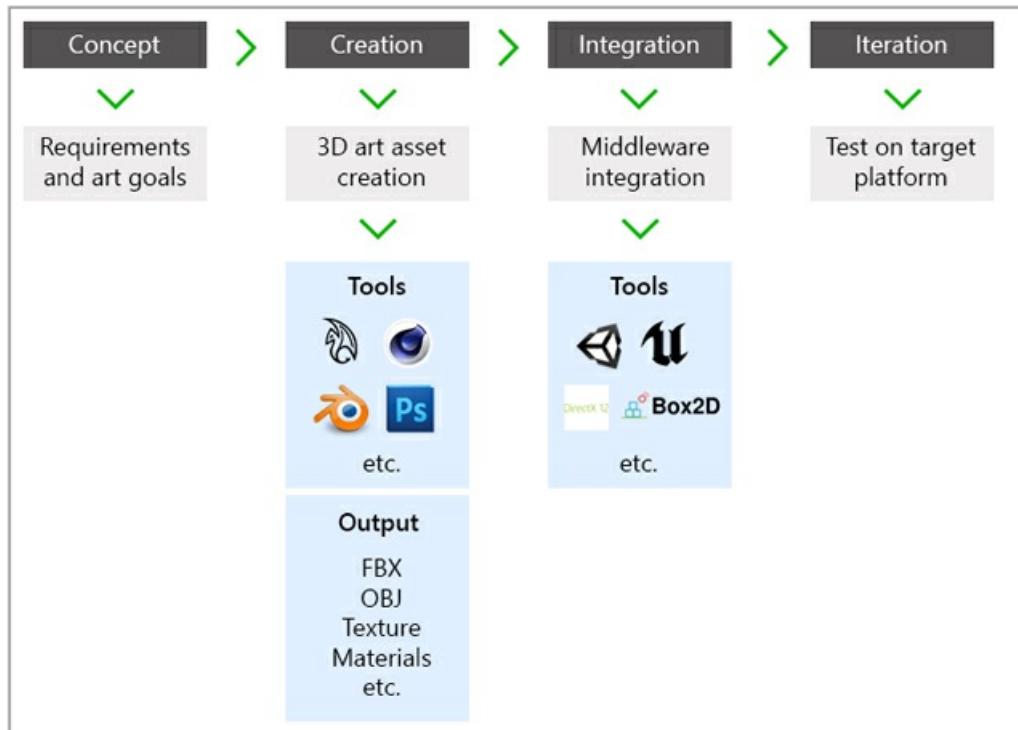
Developer @Microsoft

Asset creation process

11/6/2018 • 5 minutes to read • [Edit Online](#)

Windows Mixed Reality builds on the decades of investment Microsoft has made into DirectX. This means that all of the experience and skills developers have with building 3D graphics continues to be valuable with HoloLens.

The assets you create for a project come in many shapes and forms. They can be comprised of a series of textures/images, audio, video, 3D models and animations. We can't begin to cover all the tools that are available to create the different types of assets used in a project. For this article we will focus on 3D asset creation methods.



Concept, creation, integration and iteration flow

Things to consider

When looking at the experience you're trying to create think of it as a **budget** that you can spend to try to create the best experience. There is not necessarily hard limits on the number of **polygons** or **types of materials** use in your assets but more a budgeted set of tradeoffs.

Below is an example budget for your experience. Performance is usually not a single point of failure but death by a thousand cuts per-se.

ASSETS	CPU	GPU	MEMORY
Polygons	0%	5%	10%
Textures	5%	15%	25%
Shaders	15%	35%	0%
Dynamics			

Physics	5%	15%	0%
Realtime lighting	10%	0%	0%
Media (audio/video)	-	15%	25%
Script/logic	25%	0%	5%
General overhead	5%	5%	5%
Total	65%	90%	70%

Total number of assets

- How many assets are active in the scene?

Complexity of assets

- How many triangles / polygons?
- How complex is the shader?

Both the developers and artists have to consider the capabilities of the device and the graphics engine. Microsoft HoloLens has all of the computational and graphics built into the device. It shares the capabilities developers would find on a mobile platform.

The creation process for assets is the same regardless of whether you're targeting an experience for a [holographic device or an immersive device](#). The primary thing to note is the device capability as mentioned above as well as scale since you can see the real world in mixed reality you will want to maintain the correct scale based on the experience.

Authoring Assets

We'll start with the ways to get assets for your project:

1. Creating Assets (Authoring tools and object capture)
2. Purchasing Assets (Buying assets online)
3. Porting Assets (Taking existing assets)
4. Outsourcing Assets (Importing assets from 3rd parties)

Creating assets

Authoring tools

First you can create your own assets in a number of different ways. 3D artists use a number of applications and tools to create models which consist of **meshes**, **textures**, and **materials**. This is then saved in a file format that can be imported or used by the graphics engine used by the app, such as **.FBX** or **.OBJ**. Any tool that generates a model that your chosen graphics engine supports will work on **HoloLens**. Among 3D artists, many choose to use [Autodesk's Maya which itself is able to use HoloLens](#) to transform the way assets are created. If you want to get something in quick you can also use [3D Builder](#) that comes with Windows to export .OBJ for use in your application.

Object capture

There is also the option to capture objects in 3D. Capturing inanimate objects in 3D and editing them with digital content creation software is increasingly popular with the rise of 3D printing. Using the **Kinect 2** sensor and [3D Builder](#) you can use the capture feature to create assets from real world objects. This is also a [suite of tools](#) to do the same with **photogrammetry** by processing a number of images to stitch together and mesh and textures.

Purchasing assets

Another excellent option is to purchase assets for your experience. There are a ton of assets available through services such as the [Unity Asset Store](#) or [TurboSquid](#) among others.

When you purchase assets from a 3rd party you always want to check the following:

- **What's the poly count?**
 - Does it fit within your budget?
- **Are there levels of detail (LODs) for the model?**
 - Level of detail of a model allow you to scale the detail of a model for performance.
- **Is the source file available?**
 - Usually not included with [Unity Asset Store](#) but always included with services like [TurboSquid](#).
 - Without the source file you won't be able to modify the asset.
 - Make sure the source file provided can be imported by your 3D tools.
- **Know what you're getting**
 - Are animations provided?
 - Make sure to check the contents list of the asset you're purchasing.

Porting assets

In some cases you'll be handed existing assets that were originally built for other devices and different apps. In most cases these assets can be converted to formats compatible with the graphics engine their app is using.

When porting assets to use in your HoloLens application you will want to ask the following:

- **Can you import directly or does need to be converted to another format?** Check the format you're importing with the graphics engine you're using.
- **If converting to a compatible format is anything lost?** Sometimes details can be lost or importing can cause artifacts that need to be cleaned up in a 3D authoring tool.
- **What is the triangles / polygons count for the asset?** Based on the budget for your application you can use [Simplygon](#) or similar tools to decimate (procedurally or manually reduce poly count) the original asset to fit within your applications budget.

Outsourcing assets

Another option for larger projects that require more assets than your team is equipped to create is to outsource asset creation. The process of outsourcing involves finding the right studio or agency that specializes in outsourcing assets. This can be the most expensive option but also be the most flexible in what you get.

- **Clearly define what you're requesting**
 - Provide as much detail as possible
 - Front, side and back concept images
 - Reference art showing asset in context
 - Scale of object (Usually specified in centimeters)
- **Provide a Budget**
 - Poly count range
 - Number of textures
 - Type of shader (For Unity and HoloLens you should always default to mobile shaders first)
- **Understand the costs**
 - What's the outsourcing policy for change requests?

Outsourcing can work extremely well based on your projects timeline but requires more oversight to guarantee that you get the right assets you need the first time.

Article categories



Get started

[Install the tools](#)

[Development overview](#)

[Academy](#)



Unity

[Unity development overview](#)

[Using Vuforia with Unity](#)

[Porting guides](#)



DirectX

[DirectX development overview](#)

[Creating a holographic DirectX project](#)

[Rendering in DirectX](#)



Open source projects

[Mixed Reality Toolkit](#)

[Mixed Reality Design Labs](#)

[Galaxy Explorer](#)



Unique to HoloLens

[HoloLens hardware details](#)

[Gestures](#)

[Spatial mapping](#)



Unique to immersive headsets

[Immersive headset hardware details](#)

[Motion controllers](#)

[Performance recommendations for immersive headset apps](#)

Windows API reference (external)

Windows API reference

[Windows.Graphics.Holographic](#)

[Windows.Perception](#)

[Windows.Perception.Spatial](#)

[Windows.Perception.Spatial.Surfaces](#)

[Windows.UI.Input.Spatial](#)

[Windows.ApplicationModel.Preview.Holographic](#)

Unity API reference (external)

Gaze, gesture, and motion controller input

[UnityEngine.XR.InputTracking](#)

[UnityEngine.XR.WSA.Input.InteractionManager](#)

[UnityEngine.XR.WSA.Input.GestureRecognizer](#)

World-locking, persistence, and sharing

[UnityEngine.XR.InputTracking](#)

[UnityEngine.XR.WSA.Persistence.WorldAnchorStore](#)

[UnityEngine.XR.WSA.Sharing.WorldAnchorTransferBatch](#)

[UnityEngine.XR.WSA.WorldManager](#)

Spatial mapping

[UnityEngine.XR.WSA.SpatialMappingCollider](#)

[UnityEngine.XR.WSA.SpatialMappingRenderer](#)

App tailoring

[UnityEngine.XR.XRSettings](#)

[UnityEngine.XR.WSA.HolographicSettings](#)

Development overview

11/6/2018 • 2 minutes to read • [Edit Online](#)

Mixed reality apps are built with the [Universal Windows Platform](#). All mixed reality apps are Universal Windows apps, and all Universal Windows apps can be made to run on Windows Mixed Reality devices. With Windows 10 and familiarity with middleware tools like Unity, you can start building mixed reality experiences today.

Basics of mixed reality development

[Mixed reality](#) experiences are enabled by new Windows features for environmental understanding. These enable developers to place a [hologram](#) in the real world, and allow users to move through digital worlds by literally walking about.

These are the core building blocks for mixed reality development:

INPUT	HOOLENS	IMMERSIVE HEADSETS
Gaze	✓□	✓□
Gestures	✓□	
Voice	✓□	✓□
Gamepad	✓□	✓□
Motion controllers		✓□
PERCEPTION AND SPATIAL FEATURES	HOOLENS	IMMERSIVE HEADSETS
World coordinates	✓□	✓□
Spatial sound	✓□	✓□
Spatial mapping	✓□	

The basic interaction model for [HoloLens](#) is [gaze](#), [gesture](#), and [voice](#), sometimes referred to as [GGV](#). [Windows Mixed Reality immersive headsets](#) also use gaze and voice, but swap [motion controllers](#) for gestures.

All mixed reality devices benefit from the input ecosystem available to Windows, including mouse, keyboard, gamepads, and more. With HoloLens, [hardware accessories](#) are connected via Bluetooth. With immersive headsets, accessories connect to the host PC via Bluetooth, USB, and other supported protocols.

The environmental understanding features like [coordinates](#), [spatial sound](#), and [spatial mapping](#) provide the necessary capabilities for mixing reality. Spatial mapping is unique to HoloLens, and enables holograms to interact with both the user and the physical world around them. Coordinate systems allow the user's movement to affect movement in the digital world.

Holograms are made of light and sound, which rely on [rendering](#). Understanding the experience of placement and persistence, as demonstrated in the [Windows Mixed Reality home](#) (sometimes called the "shell") is a great way

ground yourself in the user experience.

Tools for developing for mixed reality

The tools you use will depend on the [style of app](#) you want to build.

- [Apps with a 2D view](#) leverage tools for building Universal Windows Platform apps suited for environments like Windows Phone, PC, and tablets. These apps are experienced as 2D projections placed in the Windows Mixed Reality home, and can work across multiple device types (including phone and PC).
- Immersive and holographic apps need tools designed to take advantage of the Windows Mixed Reality APIs. We recommend using [Unity](#) to build mixed reality apps. Developers interested in building their own engine can use [DirectX](#) and other [Windows APIs](#).

Regardless of the type of app you're building, these tools will facilitate your app development experience:

- [Visual Studio and the Windows SDK](#)
- [Windows Device Portal](#)
- [HoloLens emulator](#)
- [Windows Mixed Reality simulator](#)
- [App quality criteria](#)

See also

- [Install the tools](#)
- [Mixed Reality Academy tutorials](#)
- [Open source projects](#)
- [MR Basics 100: Getting started with Unity](#)
- [Windows Mixed Reality minimum PC hardware compatibility guidelines](#)
- [Submitting an app to the Windows Store](#)

Unity development overview

11/6/2018 • 5 minutes to read • [Edit Online](#)

The fastest path to building a [mixed reality app](#) is with [Unity](#). We recommend you take the time to explore the [Unity tutorials](#). If you need assets, Unity has a comprehensive [Asset Store](#). Once you have built up a basic understanding of Unity, you can visit the [Mixed Reality Academy](#) to learn the specifics of mixed reality development with Unity. Be sure to visit the [Unity Mixed Reality forums](#) to engage with the rest of the community building mixed reality apps in Unity and find solutions to problems you might run into.

Porting an existing Unity app to Windows Mixed Reality

If you have an existing Unity project that you're porting to Windows Mixed Reality, follow along with the [Unity porting guide](#) to get started.

Configuring a new Unity project for Windows Mixed Reality

To get started building mixed reality apps with Unity, first [install the tools](#). If you'll be targeting Windows Mixed Reality immersive headsets rather than HoloLens, you'll need a special version of Unity for now.

If you've just created a new Unity project, there are a small set of Unity settings you'll need to change for Windows Mixed Reality, broken down into two categories: per-project and per-scene.

Per-project settings

To target Windows Mixed Reality, you first need to set your Unity project to export as a Universal Windows Platform app:

1. Select **File > Build Settings...**
2. Select **Universal Windows Platform** in the Platform list and click **Switch Platform**
3. Set **SDK** to **Universal 10**
4. Set **Target device** to **Any Device** to support immersive headsets or switch to **HoloLens**
5. Set **Build Type** to **D3D**
6. Set **UWP SDK** to **Latest installed**

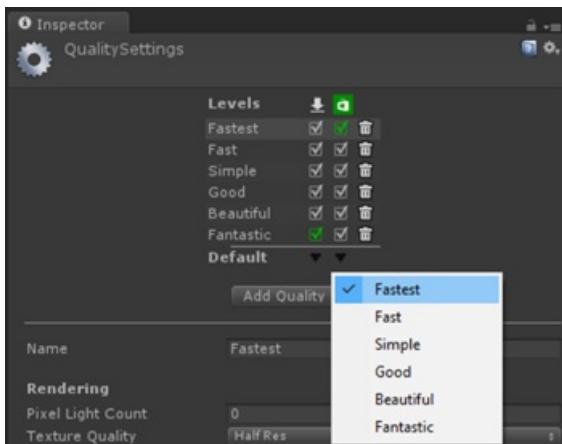
We then need to let Unity know that the app we are trying to export should create an [immersive view](#) instead of a 2D view. We do that by enabling "Virtual Reality Supported":

1. From the **Build Settings...** window, open **Player Settings...**
2. Select the **Settings for Universal Windows Platform** tab
3. Expand the **XR Settings** group
4. In the **XR Settings** section, check the **Virtual Reality Supported** checkbox to add the **Virtual Reality Devices** list.
5. In the **XR Settings** group, confirm that "**Windows Mixed Reality**" is listed as a supported device. (this may appear as "Windows Holographic" in older versions of Unity)

Your app can now do basic holographic rendering and spatial input. To go further and take advantage of certain functionality, your app must declare the appropriate capabilities in its manifest. The manifest declarations can be made in Unity so they are included in every subsequent project export. The setting are found in **Player Settings > Settings for Universal Windows Platform > Publishing Settings > Capabilities**. The applicable capabilities for enabling commonly-used Unity APIs for Mixed Reality are:

CAPABILITY	APIS REQUIRING CAPABILITY
SpatialPerception	SurfaceObserver (access to spatial mapping meshes on HoloLens)— <i>No capability needed for general spatial tracking of the headset</i>
WebCam	PhotoCapture and VideoCapture
PicturesLibrary / VideosLibrary	PhotoCapture or VideoCapture, respectively (when storing the captured content)
Microphone	VideoCapture (when capturing audio), DictationRecognizer, GrammarRecognizer, and KeywordRecognizer
InternetClient	DictationRecognizer (and to use the Unity Profiler)

Unity quality settings



Unity quality settings

HoloLens has a mobile-class GPU. If your app is targeting HoloLens, you'll want the quality settings tuned for fastest performance to ensure we maintain full framerate:

1. Select **Edit > Project Settings > Quality**
2. Select the **dropdown** under the **Windows Store** logo and select **Fastest**. You'll know the setting is applied correctly when the box in the Windows Store column and **Fastest** row is green.

Per-scene settings

Unity camera settings



Unity camera settings

Once you enable the "Virtual Reality Supported" checkbox, the [Unity Camera](#) component handles [head tracking](#) and [stereoscopic rendering](#). There is no need to replace it with a custom camera to do this.

If your app is targeting HoloLens specifically, there are a few settings that need to be changed to optimize for the device's transparent displays, so your app will show through to the physical world:

1. In the **Hierarchy**, select the **Main Camera**
2. In the **Inspector** panel, set the transform **position** to **0, 0, 0** so the location of the users head starts at the Unity world origin.
3. Change **Clear Flags** to **Solid Color**.
4. Change the **Background** color to **RGB A 0,0,0,0**. Black renders as transparent in HoloLens.
5. Change **Clipping Planes - Near** to the [HoloLens recommended](#) 0.85 (meters).

If you delete and create a new camera, make sure your camera is **Tagged** as **MainCamera**.

Adding mixed reality capabilities and inputs

Once you've configured your project as described above, standard Unity game objects (such as the camera) will light up immediately for a **seated-scale experience**, with the camera's position updated automatically as the user moves their head through the world.

Adding support for Windows Mixed Reality features such as [spatial stages](#), [gestures](#), [motion controllers](#) or [voice input](#) is achieved using APIs built directly into Unity.

Your first step should be to review the [experience scales](#) that your app can target:

- If you're looking to build an **orientation-only** or **seated-scale experience**, you'll need to set Unity's tracking space type to **Stationary**.

- If you're looking to build a **standing-scale** or **room-scale experience**, you'll need to ensure Unity's tracking space type is successfully set to [RoomScale](#).
- If you're looking to build a **world-scale** experience on HoloLens, letting users roam beyond 5 meters, you'll need to use the [WorldAnchor](#) component.

All of the core building blocks for mixed reality apps are exposed in a manner consistent with other Unity APIs:

- [Camera](#)
- [Coordinate systems](#)
- [Gaze](#)
- [Gestures and motion controllers](#)
- [Voice input](#)
- [Persistence](#)
- [Spatial sound](#)
- [Spatial mapping](#)

There are other key features that many mixed reality apps will want to use, which are also exposed to Unity apps:

- [Shared experiences](#)
- [Locatable camera](#)
- [Focus point](#)
- [Tracking loss](#)
- [Keyboard](#)

Running your Unity project on a real or simulated device

Once you've got your holographic Unity project ready to test out, your next step is to [export and build a Unity Visual Studio solution](#).

With that VS solution in hand, you can then run your app in one of three ways, using either a real or simulated device:

- [Deploy to a real HoloLens or Windows Mixed Reality immersive headset](#)
- [Deploy to the HoloLens emulator](#)
- [Deploy to the Windows Mixed Reality immersive headset simulator](#)

Unity documentation

In addition to this documentation available on the Windows Dev Center, Unity installs documentation for Windows Mixed Reality functionality alongside the Unity Editor. The Unity provided documentation includes two separate sections:

- 1. Unity scripting reference**
 - This section of the documentation contains details of the scripting API that Unity provides.
 - Accessible from the Unity Editor through **Help > Scripting Reference**
- 2. Unity manual**
 - This manual is designed to help you learn how to use Unity, from basic to advanced techniques.
 - Accessible from the Unity Editor through **Help > Manual**

See also

- [MR Basics 100: Getting started with Unity](#)
- [Recommended settings for Unity](#)

- Performance recommendations for Unity
- Exporting and building a Unity Visual Studio solution
- Using the Windows namespace with Unity apps for HoloLens
- Best practices for working with Unity and Visual Studio
- Unity Play Mode
- Porting guides

Recommended settings for Unity

11/6/2018 • 3 minutes to read • [Edit Online](#)

Unity provides a set of default options that are generally the average case for all platforms. However, Unity offers some behaviors specific to mixed reality that can be toggled through project settings.

Holographic splash screen

HoloLens has a mobile-class CPU and GPU, which means apps may take a bit longer to load. While the app is loading, users will just see black, and so they may wonder what's going on. To reassure them during loading, you can add a holographic splash screen.

To toggle the holographic splash screen, go to **Edit > Project Settings... > Player** page, click on the **Windows Store** tab and find the **Splash Image > Show Unity Splash Screen** setting and the **Windows Holographic > Holographic Splash Image**.

- Toggling the **Show Unity Splash Screen** option will enable or disable the Unity branded splash screen. If you do not have a Unity Pro license, the Unity branded splash screen will always be displayed.
- If a **Holographic Splash Image** is applied, it will always be displayed regardless of whether the Show Unity Splash Screen checkbox is enabled or disabled. Specifying a custom holographic splash image is only available to developers with a Unity Pro license.

SHOW UNITY SPLASH SCREEN	HOLOGRAPHIC SPLASH IMAGE	BEHAVIOR
On	None	Show default Unity splash screen for 5 seconds or until the app is loaded, whichever is longer.
On	Custom	Show Custom splash screen for 5 seconds or until the app is loaded, whichever is longer.
Off	None	Show transparent black (nothing) until app is loaded.
Off	Custom	Show Custom splash screen for 5 seconds or until the app is loaded, whichever is longer.

Tracking loss

A Mixed reality headset depends on seeing the environment around it to construct [world-locked coordinate systems](#), which allow holograms to remain in position. When the headset is unable to locate itself in the world, the headset is said to have *lost tracking*. In these cases, functionality dependent on world-locked coordinate systems, such as spatial stages, spatial anchors and spatial mapping, do not work.

If a loss of tracking occurs, Unity's default behavior is to stop rendering holograms, pause the [game loop](#), and display a tracking lost notification that comfortably follows the user's gaze. Custom notifications can also be provided in the form of a tracking loss image. For apps that depend upon tracking for their whole experience, it's sufficient to let Unity handle this entirely until tracking is regained.

Customize tracking loss image

Developers can supply a custom image to be shown during tracking loss. To customize the tracking lost image, go to **Edit > Project Settings... > Player** page, click on the **Windows Store** tab and find the **Windows Holographic > Tracking Loss Image**.

Opt-out of automatic pause

Some apps may not require tracking (e.g. [orientation-only apps](#) such as 360-degree video viewers) or may need to continue processing uninterrupted while tracking is lost. In these cases, apps can opt out of the default loss of tracking behavior. Developers who choose this are responsible for hiding/disabling any objects which would not render properly in a tracking-loss scenario. In most cases, the only content that is recommended to be render in that case is body-locked content, centered around the main camera.

To opt out of automatic pause behavior, go to **Edit > Project Settings... > Player** page, click on the **Windows Store** tab and find the **Windows Holographic > On Tracking Loss Pause and Show Image** checkbox.

Tracking loss events

To define custom behavior when tracking is lost, handle the global [tracking loss events](#).

Capabilities

For an app to take advantage of certain functionality, it must declare the appropriate capabilities in its manifest. The manifest declarations can be made in Unity so they are included in every subsequent project export. The setting are found in **Player Settings > Windows Store > Publishing Settings > Capabilities**. The applicable capabilities for enabling the commonly used APIs for Holographic apps are:

CAPABILITY	APIS REQUIRING CAPABILITY
SpatialPerception	SurfaceObserver
WebCam	PhotoCapture and VideoCapture
PicturesLibrary / VideosLibrary	PhotoCapture or VideoCapture, respectively (when storing the captured content)
Microphone	VideoCapture (when capturing audio), DictationRecognizer, GrammarRecognizer, and KeywordRecognizer
InternetClient	DictationRecognizer (and to use the Unity Profiler)

See also

- [Unity development overview](#)
- [Performance recommendations for Unity](#)

Unity Play Mode

11/6/2018 • 2 minutes to read • [Edit Online](#)

A fast way to work on your Unity project is to use "Play Mode". This runs your app locally in the Unity editor on your PC. Unity uses Holographic Remoting to provide a fast way to preview your content on a real HoloLens device.

Unity Play Mode with Holographic Remoting

With Holographic Remoting, you can experience your app on the HoloLens, while it runs in the Unity editor on your PC. Gaze, gesture, voice, and spatial mapping input is sent from your HoloLens to your PC. Rendered frames are then sent back to your HoloLens. This is a great way to quickly debug your app without building and deploying a full project.

1. On your HoloLens, go to the **Microsoft Store** and install the **Holographic Remoting Player** app.
2. On your HoloLens, start the **Holographic Remoting Player** app.
3. In Unity, go to the **Window** menu and select **Holographic Emulation**.
4. Set **Emulation Mode** to **Remote to Device**.
5. For **Remote Machine**, enter the IP address of your HoloLens.
6. Click **Connect**. You should see **Connection Status** change to **Connected** and see the screen go blank in the HoloLens.
7. Click the **Play** button to start Play Mode and experience the app on your HoloLens.

Holographic Remoting requires a fast PC and Wi-Fi connection. See [Holographic Remoting Player](#) for full details.

For best results, make sure your app properly sets the [focus point](#). This helps Holographic Remoting to best adapt your scene to the latency of your wireless connection.

See Also

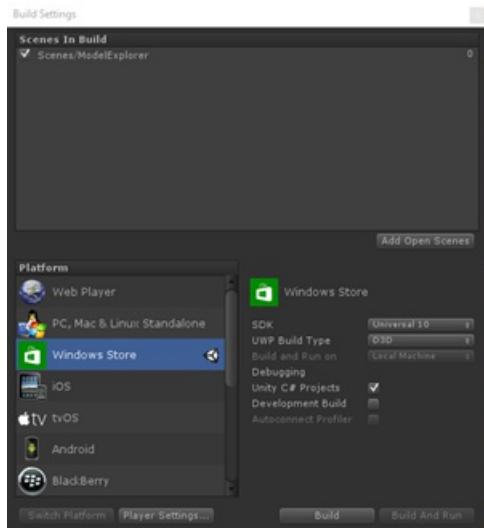
- [Holographic Remoting Player](#)

Exporting and building a Unity Visual Studio solution

11/6/2018 • 2 minutes to read • [Edit Online](#)

If you don't intend on using the system keyboard in your application, our recommendation is to use *D3D* as your application will use slightly less memory and have a slightly faster launch time. If you are using the *TouchScreenKeyboard API* in your project to use the system keyboard, you need to export as *XAML*.

How to export from Unity



Unity build settings

1. When you are ready to export your project from Unity, open the **File** menu and select **Build Settings...**
2. Click **Add Open Scenes** to add your scene to the build.
3. In the **Build Settings** dialog, choose the following options to export for HoloLens:
 - **Platform:** *Universal Windows Platform* and be sure to select **Switch Platform** for your selection to take effect.
 - **SDK:** *Universal 10*.
 - **UWP Build Type:** *D3D*.
4. **Optional: Unity C# Projects:** Checked.

NOTE

Checking this box allows you to:

- Debug your app in the Visual Studio remote debugger.
- Edit scripts in the Unity C# project while using IntelliSense for WinRT APIs.

5. From the **Build Settings...** window, open **Player Settings...**
6. Select the **Settings for Universal Windows Platform** tab.
7. Expand the **XR Settings** group.
8. In the **XR Settings** section, check the **Virtual Reality Supported** checkbox to add a new **Virtual Reality Devices** list and confirm "**Windows Mixed Reality**" is listed as a supported device.
9. Return to the **Build Settings** dialog.
10. Select **Build**.

11. In the Windows Explorer dialog that appears, create a new folder to hold Unity's build output. Generally, we name the folder "App".
12. Select the newly created folder and click **Select Folder**.
13. Once Unity has finished building, a Windows Explorer window will open to the project root directory. Navigate into the newly created folder.
14. Open the generated Visual Studio solution file located inside this folder.

When to re-export from Unity

Checking the "C# Projects" checkbox when exporting your app from Unity creates a Visual Studio solution that includes all your Unity script files. This allows you to iterate on your scripts without re-exporting from Unity. However, if you want to make changes to your project that aren't just changing the contents of scripts, you will need to re-export from Unity. Some examples of times you need to re-export from Unity are:

- You add or remove assets in the Project tab.
- You change any value in the Inspector tab.
- You add or remove objects from the Hierarchy tab.
- You change any Unity project settings

Building and deploying a Unity Visual Studio solution

The remainder of building and deploying apps happens in [Visual Studio](#). You will need to specify a Unity build configuration. Unity's naming conventions may differ from what you're usually used to in Visual Studio:

CONFIGURATION	EXPLANATION
Debug	All optimizations off and the profiler is enabled. Used to debug scripts.
Master	All optimizations are turned on and the profiler is disabled. Used to submit apps to the Store.
Release	All optimizations are turned on and the profiler is enabled. Used to evaluate app performance.

Note, the above list is a subset of the common triggers that will cause the Visual Studio project to need to be generated. In general, editing .cs files from within Visual Studio will not require the project to be regenerated from within Unity.

Troubleshooting

If you find that edits to your .cs files are not being recognized in your Visual Studio project, ensure that "Unity C# Projects" is checked when you generate the VS project from Unity's Build menu.

Best practices for working with Unity and Visual Studio

11/6/2018 • 2 minutes to read • [Edit Online](#)

A developer creating a mixed reality application with Unity will need to switch between Unity and Visual Studio to build the application package that is deployed to HoloLens and/or an immersive headset. By default two instances of Visual Studio are required (one to modify Unity scripts and one to deploy to the device and debug). The following procedure allows development using single Visual Studio instance, reduces the frequency of exporting Unity projects, and improves the debugging experience.

Improving iteration time

A common workflow problem when working with Unity and Visual Studio is having multiple windows of Visual Studio open and the need to constantly switch between Visual Studio and Unity to iterate.

1. **Unity** - for modifying the scene and exporting a Visual Studio solution
2. **Visual Studio (1)** - for modifying scripts
3. **Visual Studio (2)** - for building and deploying the Unity exported Visual Studio solution to the device

Luckily, there's a way to streamline to a single instance of Visual Studio and cut down on frequent exports from Unity.

When [exporting your project from Unity](#), check the **Unity C# Projects** checkbox in the "File > Build Settings" menu. Now, the project you export from Unity includes all of your project's C# scripts and has several benefits:

- Use the same instance of Visual Studio for writing scripts and building/deploying your project
- Export from Unity only when the scene is changed in the Unity Editor; changing the contents of scripts can be done in Visual Studio without re-exporting.

With **Unity C# Projects** enabled, only one instance of each program need be opened:

1. **Unity** - for modifying the scene and exporting a Visual Studio solution
2. **Visual Studio** - for modifying scripts then building and deploying the Unity exported Visual Studio solution to the device

Visual Studio Tools for Unity

Download [Visual Studio Tools for Unity](#)

Benefits of Visual Studio Tools for Unity

- Debug Unity in-editor play mode from Visual Studio by putting breakpoints, evaluating variables and complex expressions.
- Use the Unity Project Explorer to find your script with the exact same hierarchy that Unity displays.
- Get the Unity console directly inside Visual Studio.
- Use wizards to quickly create or navigate to scripts.

Expose C# class variables for easy tuning

Make C# class variables public to expose them in the editor UI. This makes it possible to easily tweak variables while playing in-editor. This is especially useful for tuning interaction mechanic properties.

Regenerate UWP Visual Studio solutions after Windows SDK or Unity upgrade

UWP Visual Studio solutions checked in to source control can get out-of-date after upgrading to a new Windows SDK or Unity engine. You can resolve this after the upgrade by building a new UWP solution from Unity, then merging any differences into the checked-in solution.

Use text-format assets for easy comparison of content changes

Storing assets in text format makes it easier to review content change diffs in Visual Studio. You can enable this in "Edit > Project Settings > Editor" by changing **Asset Serialization** mode to **Force Text**. However, merging text asset file changes is error-prone and not recommended, so consider enabling exclusive binary checkouts in your source control system.

Camera in Unity

11/6/2018 • 5 minutes to read • [Edit Online](#)

When you wear a mixed reality headset, it becomes the center of your holographic world. The Unity [Camera](#) component will automatically handle stereoscopic rendering and will follow your head movement and rotation when your project has "Virtual Reality Supported" selected with "Windows Mixed Reality" as the device (in the Other Settings section of the Windows Store Player Settings). This may be listed as "Windows Holographic" in older versions of Unity.

However, to fully optimize visual quality and [hologram stability](#), you should set the camera settings described below.

NOTE

These settings need to be applied to the Camera in each scene of your app.

By default, when you create a new scene in Unity, it will contain a Main Camera GameObject in the Hierarchy which includes the Camera component, but does not have the settings below properly applied.

Holographic vs. immersive headsets

The default settings on the Unity Camera component are for traditional 3D applications which need a skybox-like background as they don't have a real world.

- When running on an [immersive headset](#), you are rendering everything the user sees, and so you'll likely want to keep the skybox.
- However, when running on a [holographic headset](#) like [HoloLens](#), the real world should appear behind everything the camera renders. To do this, set the camera background to be transparent (in HoloLens, black renders as transparent) instead of a Skybox texture:
 1. Select the Main Camera in the Hierarchy panel
 2. In the Inspector panel, find the Camera component and change the Clear Flags dropdown from Skybox to Solid Color
 3. Select the Background color picker and change the RGBA values to (0, 0, 0, 0)

You can use script code to determine at runtime whether the headset is immersive or holographic by checking [HolographicSettings.IsDisplayOpaque](#).

Positioning the Camera

It will be easier to lay out your app if you imagine the starting position of the user as (X: 0, Y: 0, Z: 0). Since the Main Camera is tracking movement of the user's head, the starting position of the user can be set by setting the starting position of the Main Camera.

1. Select Main Camera in the Hierarchy panel
2. In the Inspector panel, find the Transform component and change the Position from (X: 0, Y: 1, Z: -10) to (X: 0, Y: 0, Z: 0)



Camera in the Inspector pane in Unity

Clip planes

Rendering content too close to the user can be uncomfortable in mixed reality. You can adjust the [near and far clip planes](#) on the Camera component.

1. Select the Main Camera in the Hierarchy panel
2. In the Inspector panel, find the Camera component Clipping Planes and change the Near textbox from 0.3 to .85. Content rendered even closer can lead to user discomfort and should be avoided per the [render distance guidelines](#).

Multiple Cameras

When there are multiple Camera components in the scene, Unity knows which camera to use for stereoscopic rendering and head tracking by checking which GameObject has the MainCamera tag.

Recentering a seated experience

If you're building a [seated-scale experience](#), you can recenter Unity's world origin at the user's current head position by calling the [XR.InputTracking.Recenter](#) method.

Reprojection modes

Both HoloLens and immersive headsets will reproject each frame your app renders to adjust for any misprediction of the user's actual head position when photons are emitted.

By default:

- **Immersive headsets** will perform positional reprojection, adjusting your holograms for misprediction in both position and orientation, if the app provides a depth buffer for a given frame. If a depth buffer is not provided, the system will only correct mispredictions in orientation.
- **Holographic headsets** like HoloLens will perform positional reprojection whether the app provides its depth buffer or not. Positional reprojection is possible without depth buffers on HoloLens as rendering is often sparse with a stable background provided by the real world.

If you know that you are building an [orientation-only experience](#) with rigidly body-locked content (e.g. 360-degree video content), you can explicitly set the reprojection mode to be orientation only by setting [HolographicSettings.ReprojectionMode](#) to [HolographicReprojectionMode.OrientationOnly](#).

Sharing your depth buffers with Windows

Sharing your app's depth buffer to Windows each frame will give your app one of two boosts in hologram stability, based on the type of headset you're rendering for:

- **Immersive headsets** can perform positional reprojection when a depth buffer is provided, adjusting your holograms for misprediction in both position and orientation.
- **Holographic headsets** like HoloLens will automatically select a [focus point](#) when a depth buffer is provided, optimizing hologram stability along the plane that intersects the most content.

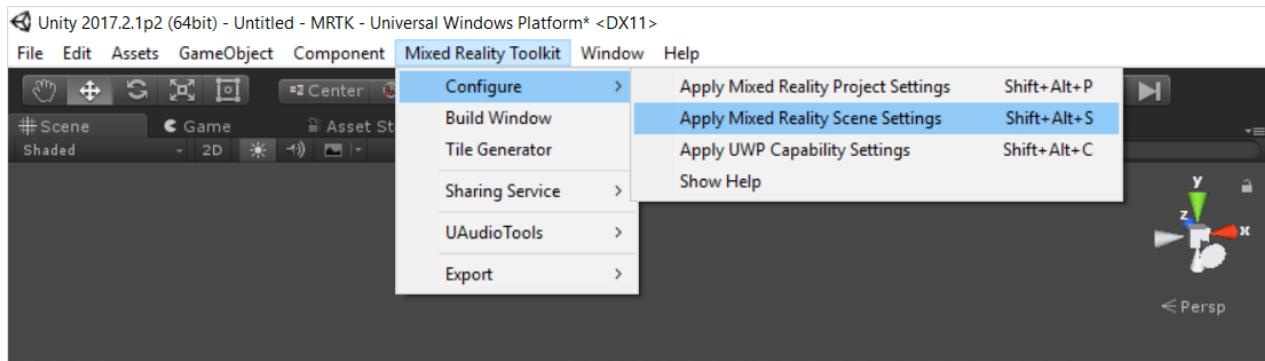
To set whether your Unity app will provide a depth buffer to Windows:

1. Go to **Edit > Project Settings > Player > Universal Windows Platform tab > XR Settings**.
2. Expand the **Windows Mixed Reality SDK** item.
3. Check or uncheck the **Enable Depth Buffer Sharing** check box. This will be checked by default in new projects created since this feature was added to Unity and will be unchecked by default for older projects that were upgraded.

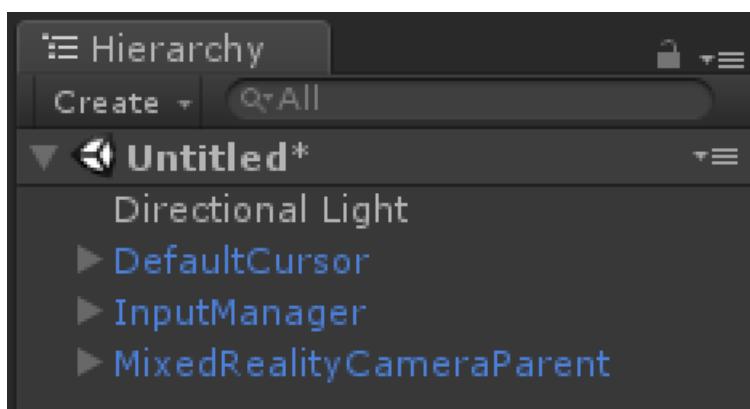
Providing a depth buffer to Windows can improve visual quality so long as Windows can accurately map the normalized per-pixel depth values in your depth buffer back to distances in meters, using the near and far planes you've set in Unity on the main camera. If your render passes handle depth values in typical ways, you should generally be fine here, though translucent render passes that write to the depth buffer while showing through to existing color pixels can confuse the reprojection. If you know that your render passes will leave many of your final depth pixels with inaccurate depth values, you are likely to get better visual quality by unchecking "Enable Depth Buffer Sharing".

Mixed Reality Toolkit's automatic scene setup

When you import [MRTK release Unity packages](#) or clone the project from the [GitHub repository](#), you are going to find a new menu 'Mixed Reality Toolkit' in Unity. Under 'Configure' menu, you will see the menu 'Apply Mixed Reality Scene Settings'. When you click it, it removes the default camera and adds foundational components - [InputManager](#), [MixedRealityCameraParent](#), and [DefaultCursor](#).



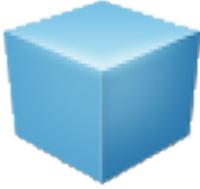
MRTK Menu for scene setup



Automatic scene setup in MRTK

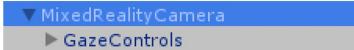
MixedRealityCamera prefab

You can also manually add these from the project panel. You can find these components as prefabs. When you search **MixedRealityCamera**, you will be able to see two different camera prefabs. The difference is, **MixedRealityCamera** is the camera only prefab whereas, **MixedRealityCameraParent** includes additional components for the immersive headsets such as Teleportation, Motion Controller and, Boundary.



MixedRealityCamera.prefab

Camera only



MixedRealityCameraParent.prefab

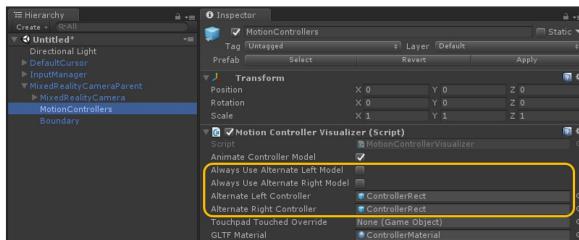
Camera + Motion Controller
+ Teleportation + Boundary



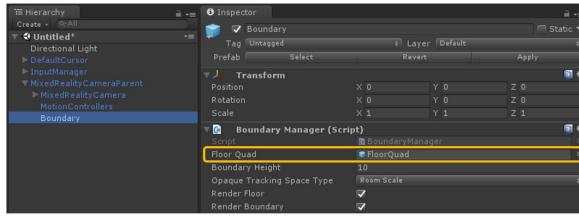
Camera prefabs in MRTK

MixedRealityCamera supports both HoloLens and immersive headset. It detects the device type and optimizes the properties such as clear flags and Skybox. Below you can find some of the useful properties you can customize such as custom Cursor, Motion Controller models, and Floor.

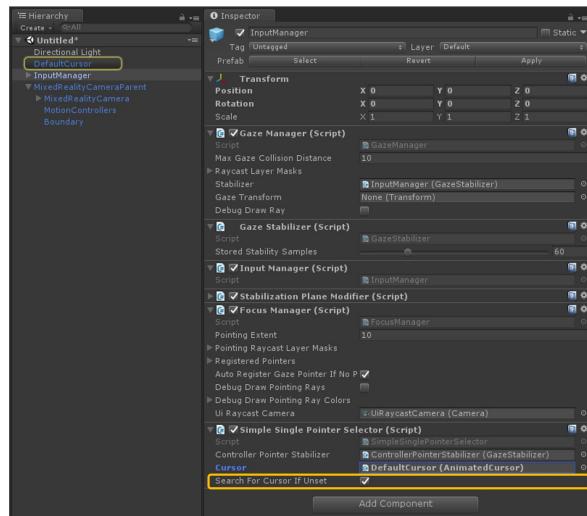
Custom Motion Controller Model



Floor visualization



Cursor



Properties for the Motion controller, Cursor and Floor

See also

- [Hologram stability](#)
- [MixedRealityToolkit Main Camera.prefab](#)

Coordinate systems in Unity

11/6/2018 • 5 minutes to read • [Edit Online](#)

Windows Mixed Reality supports apps across a wide range of [experience scales](#), from orientation-only and seated-scale apps up through room-scale apps. On HoloLens, you can go further and build world-scale apps that let users walk beyond 5 meters, exploring an entire floor of a building and beyond.

Your first step in building a mixed reality experience in Unity is to determine which [experience scale](#) your app will target.

Building an orientation-only or seated-scale experience

Namespace: *UnityEngine.XR*

Type: *XRDevice*

To build an **orientation-only** or **seated-scale experience**, you must set Unity to the Stationary tracking space type. This sets Unity's world coordinate system to track the [stationary frame of reference](#). In the Stationary tracking mode, content placed in the editor just in front of the camera's default location (forward is -Z) will appear in front of the user when the app launches.

```
XRDevice.SetTrackingSpaceType(TrackingSpaceType.Stationary);
```

Namespace: *UnityEngine.XR*

Type: *InputTracking*

For a pure **orientation-only experience** such as a 360-degree video viewer (where positional head updates would ruin the illusion), you can then set [XR.InputTracking.disablePositionalTracking](#) to true:

```
InputTracking.disablePositionalTracking = true;
```

For a **seated-scale experience**, to let the user later recenter the seated origin, you can call the [XR.InputTracking.Recenter](#) method:

```
InputTracking.Recenter();
```

Building a standing-scale or room-scale experience

Namespace: *UnityEngine.XR*

Type: *XRDevice*

For a **standing-scale** or **room-scale experience**, you'll need to place content relative to the floor. You reason about the user's floor using the [spatial stage](#), which represents the user's defined floor-level origin and optional room boundary, set up during first run.

To ensure that Unity is operating with its world coordinate system at floor-level, you can set Unity to the RoomScale tracking space type, and ensure that the set succeeds:

```

if (XRDevice.SetTrackingSpaceType(TrackingSpaceType.RoomScale))
{
    // RoomScale mode was set successfully. App can now assume that y=0 in Unity world coordinate represents
    // the floor.
}
else
{
    // RoomScale mode was not set successfully. App cannot make assumptions about where the floor plane is.
}

```

- If SetTrackingSpaceType returns true, Unity has successfully switched its world coordinate system to track the [stage frame of reference](#).
- If SetTrackingSpaceType returns false, Unity was unable to switch to the stage frame of reference, likely because the user has not set up even a floor in their environment. This is not common, but can happen if the stage was set up in a different room and the device was moved to the current room without the user setting up a new stage.

Once your app successfully sets the RoomScale tracking space type, content placed on the y=0 plane will appear on the floor. The origin at (0, 0, 0) will be the specific place on the floor where the user stood during room setup, with -Z representing the forward direction they were facing during setup.

Namespace: `UnityEngine.Experimental.XR`

Type: `Boundary`

In script code, you can then call the TryGetGeometry method on your the `UnityEngine.Experimental.XR.Boundary` type to get a boundary polygon, specifying a boundary type of `TrackedArea`. If the user defined a boundary (you get back a list of vertices), you know it is safe to deliver a **room-scale experience** to the user, where they can walk around the scene you create.

Note that the system will automatically render the boundary when the user approaches it. Your app does not need to use this polygon to render the boundary itself. However, you may choose to lay out your scene objects using this boundary polygon, to ensure the user can physically reach those objects without teleporting:

```

var vertices = new List<Vector3>();
if (UnityEngine.Experimental.XR.Boundary.TryGetGeometry(vertices, Boundary.Type.TrackedArea))
{
    // Lay out your app's content within the boundary polygon, to ensure that users can reach it without
    // teleporting.
}

```

Building a world-scale experience

Namespace: `UnityEngine.XR.WSA`

Type: `WorldAnchor`

For true **world-scale experiences** on HoloLens that let users wander beyond 5 meters, you'll need new techniques beyond those used for room-scale experiences. One key technique you'll use is to create a [spatial anchor](#) to lock a cluster of holograms precisely in place in the physical world, regardless of how far the user has roamed, and then [find those holograms again in later sessions](#).

In Unity, you create a spatial anchor by adding the **WorldAnchor** Unity component to a GameObject.

Adding a World Anchor

To add a world anchor, call `AddComponent()` on the game object with the transform you want to anchor in the real world.

```
WorldAnchor anchor = gameObject.AddComponent<WorldAnchor>();
```

That's it! This game object will now be anchored to its current location in the physical world - you may see its Unity world coordinates adjust slightly over time to ensure that physical alignment. Use [persistence](#) to find this anchored location again in a future app session.

Removing a World Anchor

If you no longer want the GameObject locked to a physical world location and don't intend on moving it this frame, then you can just call Destroy on the World Anchor component.

```
Destroy(gameObject.GetComponent<WorldAnchor>());
```

If you want to move the GameObject this frame, you need to call DestroyImmediate instead.

```
DestroyImmediate(gameObject.GetComponent<WorldAnchor>());
```

Moving a World Anchored GameObject

GameObject's cannot be moved while a World Anchor is on it. If you need to move the GameObject this frame, you need to:

1. DestroyImmediate the World Anchor component
2. Move the GameObject
3. Add a new World Anchor component to the GameObject.

```
DestroyImmediate(gameObject.GetComponent<WorldAnchor>());
gameObject.transform.position = new Vector3(0, 0, 2);
WorldAnchor anchor = gameObject.AddComponent<WorldAnchor>();
```

Handling Locatability Changes

A WorldAnchor may not be locatable in the physical world at a point in time. If that occurs, Unity will not be updating the transform of the anchored object. This also can change while an app is running. Failure to handle the change in locatability will cause the object to not appear in the correct physical location in the world.

To be notified about locatability changes:

1. Subscribe to the OnTrackingChanged event
2. Handle the event

The **OnTrackingChanged** event will be called whenever the underlying spatial anchor changes between a state of being locatable vs. not being locatable.

```
anchor.OnTrackingChanged += Anchor_OnTrackingChanged;
```

Then handle the event:

```
private void Anchor_OnTrackingChanged(WorldAnchor self, bool located)
{
    // This simply activates/deactivates this object and all children when tracking changes
    self.gameObject.SetActiveRecursively(located);
}
```

Sometimes anchors are located immediately. In this case, this `isLocated` property of the anchor will be set to true when `AddComponent()` returns. As a result, the `OnTrackingChanged` event will not be triggered. A clean pattern would be to call your `OnTrackingChanged` handler with the initial `IsLocated` state after attaching an anchor.

```
Anchor_OnTrackingChanged(anchor, anchor.isLocated);
```

See Also

- [Experience scales](#)
- [Spatial stage](#)
- [Spatial anchors](#)
- [Persistence in Unity](#)
- [Tracking loss in Unity](#)

Persistence in Unity

11/6/2018 • 2 minutes to read • [Edit Online](#)

Namespace: *UnityEngine.XR.WSA.Persistence*

Class: *WorldAnchorStore*

The WorldAnchorStore is the key to creating holographic experiences where holograms stay in specific real world positions across instances of the application. This lets your users pin individual holograms or a workspace wherever they want it, and then find it later where they expect over many uses of your app.

How to persist holograms across sessions

The WorldAnchorStore will allow you to persist the location of WorldAnchor's across sessions. To actually persist holograms across sessions, you will need to separately keep track of your GameObjects that use a particular world anchor. It often makes sense to create a GameObject root with a world anchor and have children holograms anchored by it with a local position offset.

To load holograms from previous sessions:

1. Get the WorldAnchorStore
2. Load app data relating to the world anchor which gives you the id of the world anchor
3. Load a world anchor from its id

To save holograms for future sessions:

1. Get the WorldAnchorStore
2. Save a world anchor specifying an id
3. Save app data relating to the world anchor along with an id

Getting the WorldAnchorStore

We will want to keep a reference to the WorldAnchorStore around so we know we are ready to go when we want to perform an operation. Since this is an async call, potentially as soon as start up, we want to call

```
WorldAnchorStore.GetAsync(StoreLoaded);
```

StoreLoaded in this case is our handler for when the WorldAnchorStore has completed loading:

```
private void StoreLoaded(WorldAnchorStore store)
{
    this.store = store;
}
```

We now have a reference to the WorldAnchorStore which we will use to save and load specific World Anchors.

Saving a WorldAnchor

To save, we simply need to name what we are saving and pass it in the WorldAnchor we got before when we want to save. Note: trying to save two anchors to the same string will fail (store.Save will return false). You need to Delete the previous save before saving the new one:

```
private void SaveGame()
{
    // Save data about holograms positioned by this world anchor
    if (!this.savedRoot) // Only Save the root once
    {
        this.savedRoot = this.store.Save("rootGameObject", anchor);
        Assert(this.savedRoot);
    }
}
```

Loading a WorldAnchor

And to load:

```
private void LoadGame()
{
    // Save data about holograms positioned by this world anchor
    this.savedRoot = this.store.Load("rootGameObject", rootGameObject);
    if (!this.savedRoot)
    {
        // We didn't actually have the game root saved! We have to re-place our objects or start over
    }
}
```

We additionally can use store.Delete() to remove an anchor we previously saved and store.Clear() to remove all previously saved data.

Enumerating Existing Anchors

To discover previously stored anchors, call GetAllIds.

```
string[] ids = this.store.GetAllIds();
for (int index = 0; index < ids.Length; index++)
{
    Debug.Log(ids[index]);
}
```

See Also

- [Spatial anchor persistence](#)

Gaze in Unity

11/6/2018 • 2 minutes to read • [Edit Online](#)

Gaze is a primary way for users to target the [holograms](#) your app creates in [mixed reality](#).

Implementing Gaze

Conceptually, [gaze](#) is implemented by projecting a ray from the user's head where the headset is, in the forward direction they are facing and determining what that ray collides with. In Unity, the user's head position and direction are exposed through the Unity Main [Camera](#), specifically [UnityEngine.Camera.main.transform.forward](#) and [UnityEngine.Camera.main.transform.position](#).

Calling [Physics.Raycast](#) results in a [RaycastHit](#) structure which contains information about the collision including the 3D point where collision occurred and the other [GameObject](#) the gaze ray collided with.

Example: Implement Gaze

```
void Update()
{
    RaycastHit hitInfo;
    if (Physics.Raycast(
        Camera.main.transform.position,
        Camera.main.transform.forward,
        out hitInfo,
        20.0f,
        Physics.DefaultRaycastLayers))
    {
        // If the Raycast has succeeded and hit a hologram
        // hitInfo's point represents the position being gazed at
        // hitInfo's collider GameObject represents the hologram being gazed at
    }
}
```

Best Practices

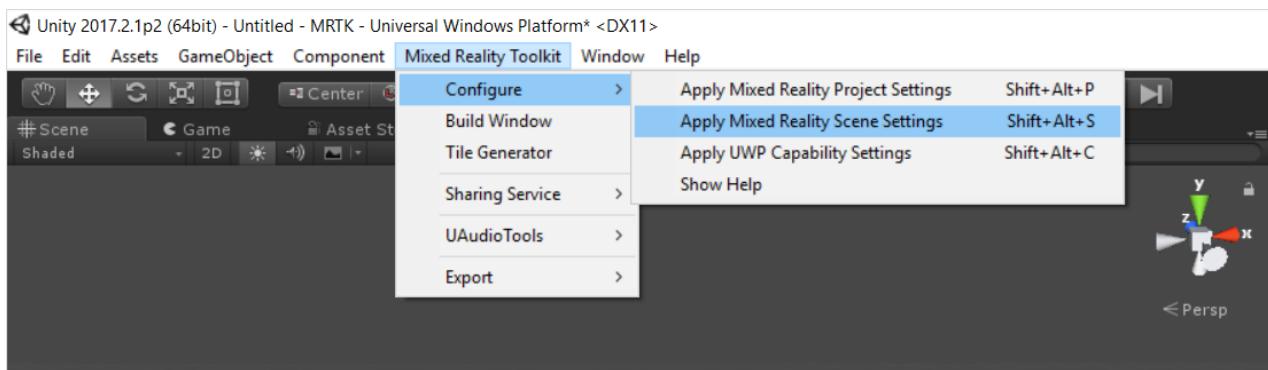
While the example above demonstrates how to do a single raycast in an update loop to find the Gaze target, it is recommended to do this in a single object managing gaze instead of doing this in any object that is potentially interested in the object being gazed at. This lets your app save processing by doing just one gaze raycast each frame.

Visualizing Gaze

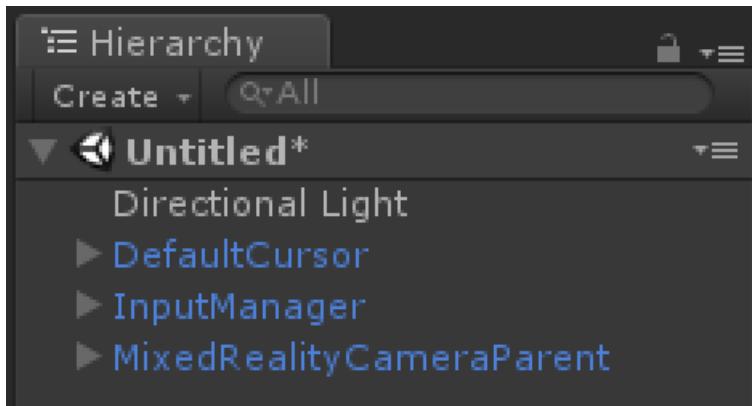
Just like on the desktop where you use a mouse pointer to target and interact with content, you should implement a [cursor](#) that represents the user's gaze. This gives the user confidence in what they're about to interact with.

Gaze in Mixed Reality Toolkit

When you import [MRTK release Unity packages](#) or clone the project from the [GitHub repository](#), you are going to find a new menu 'Mixed Reality Toolkit' in Unity. Under 'Configure' menu, you will see the menu 'Apply Mixed Reality Scene Settings'. When you click it, it removes the default camera and adds foundational components - [InputManager](#), [MixedRealityCameraParent](#), and [DefaultCursor](#).



MRTK Menu for scene setup



Automatic scene setup in MRTK

Gaze related scripts in Mixed Reality Toolkit

Mixed Reality Toolkit's prefab includes [GazeManager.cs](#) and [Gaze Stabilizer](#). Under [SimpleSinglePointerSelector](#), you can assign your custom Cursor. In default, animated [DefaultCursor](#) is assigned.

[Cursor.prefab](#) and [CursorWithFeedback.prefab](#) shows you how to visualize your Gaze using Cursors.

See also

- [Camera](#)
- [Gaze](#)
- [Cursors](#)
- [Gaze targeting](#)

Gestures and motion controllers in Unity

11/6/2018 • 21 minutes to read • [Edit Online](#)

There are two key ways to take action on your [gaze in Unity](#), [hand gestures](#) and [motion controllers](#). You access the data for both sources of spatial input through the same APIs in Unity.

Unity provides two primary ways to access spatial input data for Windows Mixed Reality, the common `Input.GetButton/GetAxis` APIs that work across multiple Unity XR SDKs, and an `InteractionManager/GestureRecognizer` API specific to Windows Mixed Reality that exposes the full set of spatial input data available.

Unity button/axis mapping table

The button and axis IDs in the table below are supported in Unity's Input Manager for Windows Mixed Reality motion controllers through the `Input.GetButton/GetAxis` APIs, while the "Windows MR-specific" column refers to properties available off of the `InteractionSourceState` type. Each of these APIs are described in detail in the sections below.

The button/axis ID mappings for Windows Mixed Reality generally match the Oculus button/axis IDs.

The button/axis ID mappings for Windows Mixed Reality differ from OpenVR's mappings in two ways:

1. The mapping uses touchpad IDs that are distinct from thumbstick, to support controllers with both thumbsticks and touchpads.
2. The mapping avoids overloading the A and X button IDs for the Menu buttons, to leave those available for physical ABXY buttons.

INPUT	COMMON UNITY APIs (INPUT.GETBUTTON/GETAXIS)		WINDOWS MR-SPECIFIC INPUT API (XR.WSA.INPUT)
	LEFT HAND	RIGHT HAND	
Select trigger pressed	Axis 9 = 1.0	Axis 10 = 1.0	selectPressed
Select trigger analog value	Axis 9	Axis 10	selectPressedAmount
Select trigger partially pressed	Button 14 (gamepad compat)	Button 15 (gamepad compat)	selectPressedAmount > 0.0
Menu button pressed	Button 6*	Button 7*	menuPressed
Grip button pressed	Axis 11 = 1.0 (no analog values) Button 4 (gamepad compat)	Axis 12 = 1.0 (no analog values) Button 5 (gamepad compat)	grasped
Thumbstick X (left: -1.0, right: 1.0)	Axis 1	Axis 4	thumbstickPosition.x
Thumbstick Y (top: -1.0, bottom: 1.0)	Axis 2	Axis 5	thumbstickPosition.y

Thumbstick pressed	Button 8	Button 9	thumbstickPressed
Touchpad X (left: -1.0, right: 1.0)	Axis 17*	Axis 19*	touchpadPosition.x
Touchpad Y (top: -1.0, bottom: 1.0)	Axis 18*	Axis 20*	touchpadPosition.y
Touchpad touched	Button 18*	Button 19*	touchpadTouched
Touchpad pressed	Button 16*	Button 17*	touchpadPressed
6DoF grip pose or pointer pose	<i>Grip</i> pose only: XR.InputTracking.GetLocalPosition XR.InputTracking.GetLocalRotation		Pass <i>Grip</i> or <i>Pointer</i> as an argument: sourceState.sourcePose.TryGetPosition sourceState.sourcePose.TryGetRotation
Tracking state	<i>Position accuracy and source loss risk only available through MR-specific API</i>		sourceState.sourcePose.positionAccuracy sourceState.properties.sourceLossRisk

NOTE

These button/axis IDs differ from the IDs that Unity uses for OpenVR due to collisions in the mappings used by gamepads, Oculus Touch and OpenVR.

Grip pose vs. pointing pose

Windows Mixed Reality supports motion controllers in a variety of form factors, with each controller's design differing in its relationship between the user's hand position and the natural "forward" direction that apps should use for pointing when rendering the controller.

To better represent these controllers, there are two kinds of poses you can investigate for each interaction source, the **grip pose** and the **pointer pose**. Both the grip pose and pointer pose coordinates are expressed by all Unity APIs in global Unity world coordinates.

Grip pose

The **grip pose** represents the location of either the palm of a hand detected by a HoloLens, or the palm holding a motion controller.

On immersive headsets, the grip pose is best used to render **the user's hand or an object held in the user's hand**, such as a sword or gun. The grip pose is also used when visualizing a motion controller, as the **renderable model** provided by Windows for a motion controller uses the grip pose as its origin and center of rotation.

The grip pose is defined specifically as follows:

- The **grip position**: The palm centroid when holding the controller naturally, adjusted left or right to center the position within the grip. On the Windows Mixed Reality motion controller, this position generally aligns with the Grasp button.
- The **grip orientation's Right axis**: When you completely open your hand to form a flat 5-finger pose, the ray that is normal to your palm (forward from left palm, backward from right palm)
- The **grip orientation's Forward axis**: When you close your hand partially (as if holding the controller), the

ray that points "forward" through the tube formed by your non-thumb fingers.

- The **grip orientation's Up axis**: The Up axis implied by the Right and Forward definitions.

You can access the grip pose through either Unity's cross-vendor input API ([XR.InputTracking.GetLocalPosition/Rotation](#)) or through the Windows MR-specific API (`sourceState.sourcePose.TryGetPosition/Rotation`, requesting pose data for the **Grip** node).

Pointer pose

The **pointer pose** represents the tip of the controller pointing forward.

The system-provided pointer pose is best used to raycast when you are **rendering the controller model itself**. If you are rendering some other virtual object in place of the controller, such as a virtual gun, you should point with a ray that is most natural for that virtual object, such as a ray that travels along the barrel of the app-defined gun model. Because users can see the virtual object and not the physical controller, pointing with the virtual object will likely be more natural for those using your app.

Currently, the pointer pose is available in Unity only through the Windows MR-specific API, `sourceState.sourcePose.TryGetPosition/Rotation`, passing in `InteractionSourceNode.Pointer` as the argument.

Controller tracking state

Like the headsets, the Windows Mixed Reality motion controller requires no setup of external tracking sensors. Instead, the controllers are tracked by sensors in the headset itself.

If the user moves the controllers out of the headset's field of view, in most cases Windows will continue to infer controller positions and provide them to the app. When the controller has lost visual tracking for long enough, the controller's positions will drop to approximate-accuracy positions.

At this point, the system will body-lock the controller to the user, tracking the user's position as they move around, while still exposing the controller's true orientation using its internal orientation sensors. Many apps that use controllers to point at and activate UI elements can operate normally while in approximate accuracy without the user noticing.

The best way to get a feel for this is to try it yourself. Check out this video with examples of immersive content that works with motion controllers across various tracking states:

Reasoning about tracking state explicitly

Apps that wish to treat positions differently based on tracking state may go further and inspect properties on the controller's state, such as `SourceLossRisk` and `PositionAccuracy`:

TRACKING STATE	SOURCELOSSRISK	POSITIONACCURACY	TRYGETPOSITION
High accuracy	< 1.0	High	true
High accuracy (at risk of losing)	== 1.0	High	true
Approximate accuracy	== 1.0	Approximate	true
No position	== 1.0	Approximate	false

These motion controller tracking states are defined as follows:

- **High accuracy**: While the motion controller is within the headset's field of view, it will generally provide high-

accuracy positions, based on visual tracking. Note that a moving controller that momentarily leaves the field of view or is momentarily obscured from the headset sensors (e.g. by the user's other hand) will continue to return high-accuracy poses for a short time, based on inertial tracking of the controller itself.

- **High accuracy (at risk of losing):** When the user moves the motion controller past the edge of the headset's field of view, the headset will soon be unable to visually track the controller's position. The app knows when the controller has reached this FOV boundary by seeing the **SourceLossRisk** reach 1.0. At that point, the app may choose to pause controller gestures that require a steady stream of very high-quality poses.
- **Approximate accuracy:** When the controller has lost visual tracking for long enough, the controller's positions will drop to approximate-accuracy positions. At this point, the system will body-lock the controller to the user, tracking the user's position as they move around, while still exposing the controller's true orientation using its internal orientation sensors. Many apps that use controllers to point at and activate UI elements can operate as normal while in approximate accuracy without the user noticing. Apps with heavier input requirements may choose to sense this drop from **High** accuracy to **Approximate** accuracy by inspecting the **PositionAccuracy** property, for example to give the user a more generous hitbox on off-screen targets during this time.
- **No position:** While the controller can operate at approximate accuracy for a long time, sometimes the system knows that even a body-locked position is not meaningful at the moment. For example, a controller that was just turned on may have never been observed visually, or a user may put down a controller that's then picked up by someone else. At those times, the system will not provide any position to the app, and *TryGetPosition* will return false.

Common Unity APIs (`Input.GetButton/GetAxis`)

Namespace: `UnityEngine`, `UnityEngine.XR`

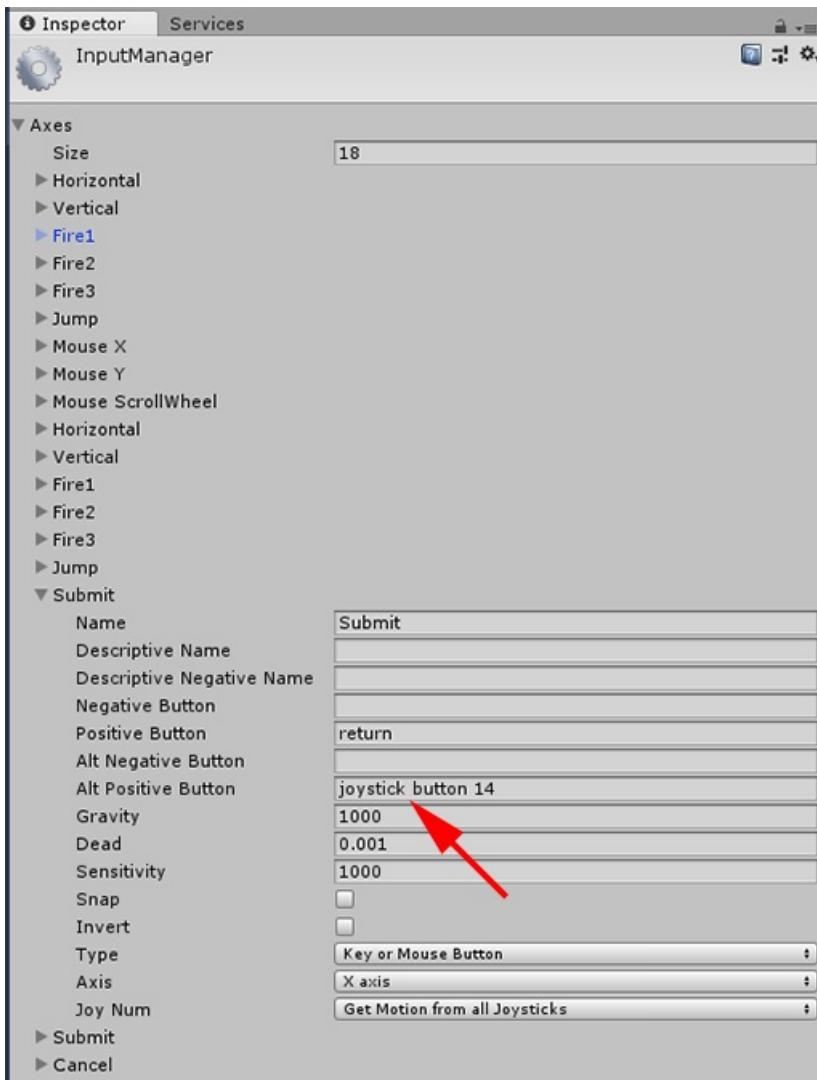
Types: `Input`, `XR.InputTracking`

Unity currently uses its general `Input.GetButton`/`Input.GetAxis` APIs to expose input for [the Oculus SDK](#), [the OpenVR SDK](#) and Windows Mixed Reality, including hands and motion controllers. If your app uses these APIs for input, it can easily support motion controllers across multiple XR SDKs, including Windows Mixed Reality.

Getting a logical button's pressed state

To use the general Unity input APIs, you'll typically start by wiring up buttons and axes to logical names in the [Unity Input Manager](#), binding a button or axis IDs to each name. You can then write code that refers to that logical button/axis name.

For example, to map the left motion controller's trigger button to the Submit action, go to **Edit > Project Settings > Input** within Unity, and expand the properties of the Submit section under Axes. Change the **Positive Button** or **Alt Positive Button** property to read **joystick button 14**, like this:



Unity InputManager

Your script can then check for the Submit action using `Input.GetButton`:

```
if (Input.GetButton("Submit"))
{
    // ...
}
```

You can add more logical buttons by changing the **Size** property under **Axes**.

Getting a physical button's pressed state directly

You can also access buttons manually by their fully-qualified name, using `Input.GetKey`:

```
if (Input.GetKey("joystick button 8"))
{
    // ...
}
```

Getting a hand or motion controller's pose

You can access the position and rotation of the controller, using `XR.InputTracking`:

```
Vector3 leftPosition = InputTracking.GetLocalPosition(XRNode.LeftHand);
Quaternion leftRotation = InputTracking.GetLocalRotation(XRNode.LeftHand);
```

Note that this represents the controller's grip pose (where the user holds the controller), which is useful for

rendering a sword or gun in the user's hand, or a model of the controller itself.

Note that the relationship between this grip pose and the pointer pose (where the tip of the controller is pointing) may differ across controllers. At this moment, accessing the controller's pointer pose is only possible through the MR-specific input API, described in the sections below.

Windows-specific APIs (XR.WSA.Input)

Namespace: *UnityEngine.XR.WSA.Input*

Types: *InteractionManager, InteractionSourceState, InteractionSource, InteractionSourceProperties, InteractionSourceKind, InteractionSourceLocation*

To get at more detailed information about Windows Mixed Reality hand input (for HoloLens) and motion controllers, you can choose to use the Windows-specific spatial input APIs under the *UnityEngine.XR.WSA.Input* namespace. This lets you access additional information, such as position accuracy or the source kind, letting you tell hands and controllers apart.

Polling for the state of hands and motion controllers

You can poll for this frame's state for each interaction source (hand or motion controller) using the *GetCurrentReading* method.

```
var interactionSourceStates = InteractionManager.GetCurrentReading();
foreach (var interactionSourceState in interactionSourceStates) {
    // ...
}
```

Each *InteractionSourceState* you get back represents an interaction source at the current moment in time. The *InteractionSourceState* exposes info such as:

- Which [kinds of presses](#) are occurring (Select/Menu/Grasp/Touchpad/Thumbstick)

```
if (interactionSourceState.selectPressed) {
    // ...
}
```

- Other data specific to motion controllers, such the touchpad and/or thumbstick's XY coordinates and touched state

```
if (interactionSourceState.touchpadTouched && interactionSourceState.touchpadPosition.x > 0.5) {
    // ...
}
```

- The *InteractionSourceKind* to know if the source is a hand or a motion controller

```
if (interactionSourceState.source.kind == InteractionSourceKind.Hand) {
    // ...
}
```

Polling for forward-predicted rendering poses

- When polling for interaction source data from hands and controllers, the poses you get are forward-predicted poses for the moment in time when this frame's photons will reach the user's eyes. These forward-predicted poses are best used for **rendering** the controller or a held object each frame. If you are targeting a given press or release with the controller, that will be most accurate if you use the historical event APIs described below.

```

var sourcePose = interactionSourceState.sourcePose;
Vector3 sourceGripPosition;
Quaternion sourceGripRotation;
if ((sourcePose.TryGetPosition(out sourceGripPosition, InteractionSourceNode.Grip)) &&
    (sourcePose.TryGetRotation(out sourceGripRotation, InteractionSourceNode.Grip))) {
    // ...
}

```

- You can also get the forward-predicted head pose for this current frame. As with the source pose, this is useful for **rendering** a cursor, although targeting a given press or release will be most accurate if you use the historical event APIs described below.

```

var headPose = interactionSourceState.headPose;
var headRay = new Ray(headPose.position, headPose.forward);
RaycastHit raycastHit;
if (Physics.Raycast(headPose.position, headPose.forward, out raycastHit, 10)) {
    var cursorPos = raycastHit.point;
    // ...
}

```

Handling interaction source events

To handle input events as they happen with their accurate historical pose data, you can handle interaction source events instead of polling.

To handle interaction source events:

- Register for a *InteractionManager* input event. For each type of interaction event that you are interested in, you need to subscribe to it.

```
InteractionManager.InteractionSourcePressed += InteractionManager_InteractionSourcePressed;
```

- Handle the event. Once you have subscribed to an interaction event, you will get the callback when appropriate. In the *SourcePressed* example, this will be after the source was detected and before it is released or lost.

```

void InteractionManager_InteractionSourceDetected(InteractionSourceEventArgs args)
    var interactionSourceState = args.state;

    // args.state has information about:
    // targeting head ray at the time when the event was triggered
    // whether the source is pressed or not
    // properties like position, velocity, source loss risk
    // source id (which hand id for example) and source kind like hand, voice, controller or other
}

```

How to stop handling an event

You need to stop handling an event when you are no longer interested in the event or you are destroying the object that has subscribed to the event. To stop handling the event, you unsubscribe from the event.

```
InteractionManager.InteractionSourcePressed -= InteractionManager_InteractionSourcePressed;
```

List of interaction source events

The available interaction source events are:

- *InteractionSourceDetected* (source becomes active)

- *InteractionSourceLost* (becomes inactive)
- *InteractionSourcePressed* (tap, button press, or "Select" uttered)
- *InteractionSourceReleased* (end of a tap, button released, or end of "Select" uttered)
- *InteractionSourceUpdated* (moves or otherwise changes some state)

Events for historical targeting poses that most accurately match a press or release

The polling APIs described earlier give your app forward-predicted poses. While those predicted poses are best for rendering the controller or a virtual handheld object, future poses are not optimal for targeting, for two key reasons:

- When the user presses a button on a controller, there can be about 20ms of wireless latency over Bluetooth before the system receives the press.
- Then, if you are using a forward-predicted pose, there would be another 10-20ms of forward prediction applied to target the time when the current frame's photons will reach the user's eyes.

This means that polling gives you a source pose or head pose that is 30-40ms forward from where the user's head and hands actually were back when the press or release happened. For HoloLens hand input, while there's no wireless transmission delay, there is a similar processing delay to detect the press.

To accurately target based on the user's original intent for a hand or controller press, you should use the historical source pose or head pose from that *InteractionSourcePressed* or *InteractionSourceReleased* input event.

You can target a press or release with historical pose data from the user's head or their controller:

- The head pose at the moment in time when a gesture or controller press occurred, which can be used for **targeting** to determine what the user was [gazing](#) at:

```
void InteractionManager_InteractionSourcePressed(InteractionSourcePressedEventArgs args) {
    var interactionSourceState = args.state;
    var headPose = interactionSourceState.headPose;
    RaycastHit raycastHit;
    if (Physics.Raycast(headPose.position, headPose.forward, out raycastHit, 10)) {
        var targetObject = raycastHit.collider.gameObject;
        // ...
    }
}
```

- The source pose at the moment in time when a motion controller press occurred, which can be used for **targeting** to determine what the user was pointing the controller at. This will be the state of the controller that experienced the press. If you are rendering the controller itself, you can request the pointer pose rather than the grip pose, to shoot the targeting ray from what the user will consider the natural tip of that rendered controller:

```
void InteractionManager_InteractionSourcePressed(InteractionSourcePressedEventArgs args)
{
    var interactionSourceState = args.state;
    var sourcePose = interactionSourceState.sourcePose;
    Vector3 sourceGripPosition;
    Quaternion sourceGripRotation;
    if ((sourcePose.TryGetPosition(out sourceGripPosition, InteractionSourceNode.Pointer)) &&
        (sourcePose.TryGetRotation(out sourceGripRotation, InteractionSourceNode.Pointer))) {
        RaycastHit raycastHit;
        if (Physics.Raycast(sourceGripPosition, sourceGripRotation * Vector3.forward, out raycastHit,
10)) {
            var targetObject = raycastHit.collider.gameObject;
            // ...
        }
    }
}
```

Event handlers example

```

using UnityEngine.XR.WSA.Input;

void Start()
{
    InteractionManager.InteractionSourceDetected += InteractionManager_InteractionSourceDetected;
    InteractionManager.InteractionSourceLost += InteractionManager_InteractionSourceLost;
    InteractionManager.InteractionSourcePressed += InteractionManager_InteractionSourcePressed;
    InteractionManager.InteractionSourceReleased += InteractionManager_InteractionSourceReleased;
    InteractionManager.InteractionSourceUpdated += InteractionManager_InteractionSourceUpdated;
}

void OnDestroy()
{
    InteractionManager.InteractionSourceDetected -= InteractionManager_InteractionSourceDetected;
    InteractionManager.InteractionSourceLost -= InteractionManager_InteractionSourceLost;
    InteractionManager.InteractionSourcePressed -= InteractionManager_InteractionSourcePressed;
    InteractionManager.InteractionSourceReleased -= InteractionManager_InteractionSourceReleased;
    InteractionManager.InteractionSourceUpdated -= InteractionManager_InteractionSourceUpdated;
}

void InteractionManager_InteractionSourceDetected(InteractionSourceDetectedEventArgs args)
{
    // Source was detected
    // args.state has the current state of the source including id, position, kind, etc.
}

void InteractionManager_InteractionSourceLost(InteractionSourceLostEventArgs state)
{
    // Source was lost. This will be after a SourceDetected event and no other events for this
    // source id will occur until it is Detected again
    // args.state has the current state of the source including id, position, kind, etc.
}

void InteractionManager_InteractionSourcePressed(InteractionSourcePressedEventArgs state)
{
    // Source was pressed. This will be after the source was detected and before it is
    // released or lost
    // args.state has the current state of the source including id, position, kind, etc.
}

void InteractionManager_InteractionSourceReleased(InteractionSourceReleasedEventArgs state)
{
    // Source was released. The source would have been detected and pressed before this point.
    // This event will not fire if the source is lost
    // args.state has the current state of the source including id, position, kind, etc.
}

void InteractionManager_InteractionSourceUpdated(InteractionSourceUpdatedEventArgs state)
{
    // Source was updated. The source would have been detected before this point
    // args.state has the current state of the source including id, position, kind, etc.
}

```

High-level composite gesture APIs (GestureRecognizer)

Namespace: `UnityEngine.XR.WSA.Input`

Types: `GestureRecognizer`, `GestureSettings`, `InteractionSourceKind`

Your app can also recognize higher-level composite gestures for spatial input sources, Tap, Hold, Manipulation and Navigation gestures. You can recognize these composite gestures across both [hands](#) and [motion controllers](#) using the `GestureRecognizer`.

Each Gesture event on the `GestureRecognizer` provides the `SourceKind` for the input as well as the targeting head ray at the time of the event. Some events provide additional context specific information.

There are only a few steps required to capture gestures using a Gesture Recognizer:

1. Create a new Gesture Recognizer
2. Specify which gestures to watch for
3. Subscribe to events for those gestures
4. Start capturing gestures

Create a new Gesture Recognizer

To use the *GestureRecognizer*, you must have created a *GestureRecognizer*:

```
GestureRecognizer recognizer = new GestureRecognizer();
```

Specify which gestures to watch for

Specify which gestures you are interested in via *SetRecognizableGestures()*:

```
recognizer.SetRecognizableGestures(GestureSettings.Tap | GestureSettings.Hold);
```

Subscribe to events for those gestures

Subscribe to events for the gestures you are interested in.

```
void Start()
{
    recognizer.Tapped += GestureRecognizer_Tapped;
    recognizer.HoldStarted += GestureRecognizer_HoldStarted;
    recognizer.HoldCompleted += GestureRecognizer_HoldCompleted;
    recognizer.HoldCanceled += GestureRecognizer_HoldCanceled;
}
```

NOTE

Navigation and Manipulation gestures are mutually exclusive on an instance of a *GestureRecognizer*.

Start capturing gestures

By default, a *GestureRecognizer* does not monitor input until *StartCapturingGestures()* is called. It is possible that a gesture event may be generated after *StopCapturingGestures()* is called if input was performed before the frame where *StopCapturingGestures()* was processed. The *GestureRecognizer* will remember whether it was on or off during the previous frame in which the gesture actually occurred, and so it is reliable to start and stop gesture monitoring based on this frame's gaze targeting.

```
recognizer.StartCapturingGestures();
```

Stop capturing gestures

To stop gesture recognition:

```
recognizer.StopCapturingGestures();
```

Removing a gesture recognizer

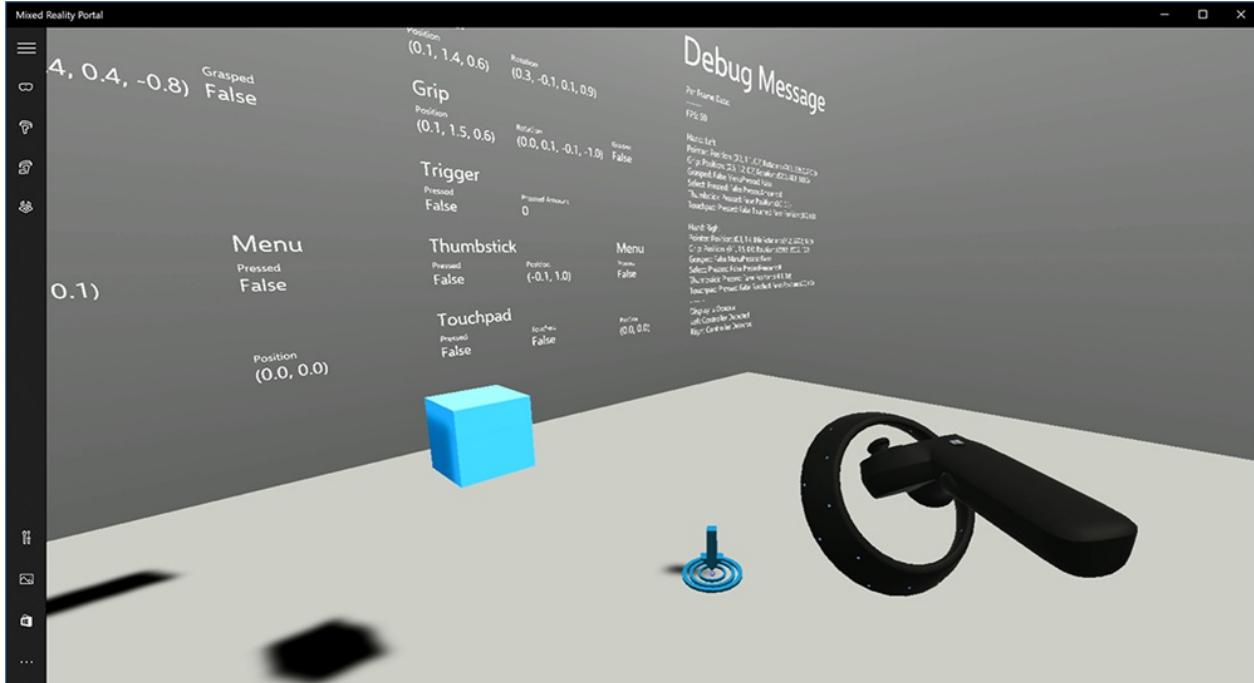
Remember to unsubscribe from subscribed events before destroying a *GestureRecognizer* object.

```

void OnDestroy()
{
    recognizer.Tapped -= GestureRecognizer_Tapped;
    recognizer.HoldStarted -= GestureRecognizer_HoldStarted;
    recognizer.HoldCompleted -= GestureRecognizer_HoldCompleted;
    recognizer.HoldCanceled -= GestureRecognizer_HoldCanceled;
}

```

Rendering the motion controller model in Unity



Motion controller model and teleportation

To render motion controllers in your app that match the physical controllers your users are holding and articulate as various buttons are pressed, you can use the **MotionController prefab** in the [Mixed Reality Toolkit](#). This prefab dynamically loads the correct glTF model at runtime from the system's installed motion controller driver. It's important to load these models dynamically rather than importing them manually in the editor, so that your app will show physically accurate 3D models for any current and future controllers your users may have.

1. Follow the [Getting Started](#) instructions to download the Mixed Reality Toolkit and add it to your Unity project.
2. If you replaced your camera with the *MixedRealityCameraParent* prefab as part of the Getting Started steps, you're good to go! That prefab includes motion controller rendering. Otherwise, add *Assets/HoloToolkit/Input/Prefabs/MotionControllers.prefab* into your scene from the Project pane. You'll want to add that prefab as a child of whatever parent object you use to move the camera around when the user teleports within your scene, so that the controllers come along with the user. If your app does not involve teleporting, just add the prefab at the root of your scene.

Throwing objects

Throwing objects in virtual reality is a harder problem than it may at first seem. As with most physically based interactions, when throwing in game acts in an unexpected way, it is immediately obvious and breaks immersion. We have spent some time thinking deeply about how to represent a physically correct throwing behavior, and have come up with a few guidelines, enabled through updates to our platform, that we would like to share with you.

You can find an example of how we recommend to implement throwing [here](#). This sample follows these four guidelines:

- **Use the controller's velocity instead of position.** In the November update to Windows, we introduced a change in behavior when in the "Approximate" positional tracking state. When in this state, velocity information about the controller will continue to be reported for as long as we believe it is high accuracy, which is often longer than position remains high accuracy.

- **Incorporate the angular velocity of the controller.** This logic is all contained in the `throwing.cs` file in the `GetThrownObjectVelAngVel` static method, within the package linked above:

1. As angular velocity is conserved, the thrown object must maintain the same angular velocity as it had at the moment of the throw: `objectAngularVelocity = throwingControllerAngularVelocity;`
2. As the center of mass of the thrown object is likely not at the origin of the grip pose, it likely has a different velocity than that of the controller in the frame of reference of the user. The portion of the object's velocity contributed in this way is the instantaneous tangential velocity of the center of mass of the thrown object around the controller origin. This tangential velocity is the cross product of the angular velocity of the controller with the vector representing the distance between the controller origin and the center of mass of the thrown object.

```
Vector3 radialVec = thrownObjectCenterOfMass - throwingControllerPos;
Vector3 tangentialVelocity = Vector3.Cross(throwingControllerAngularVelocity, radialVec);
```

3. The total velocity of the thrown object is thus the sum of velocity of the controller and this tangential velocity: `objectVelocity = throwingControllerVelocity + tangentialVelocity;`

- **Pay close attention to the time at which we apply the velocity.** When a button is pressed, it can take up to 20ms for that event to bubble up through Bluetooth to the operating system. This means that if you poll for a controller state change from pressed to not pressed or vice versa, the controller pose information you get with it will actually be ahead of this change in state. Further, the controller pose presented by our polling API is forward predicted to reflect a likely pose at the time the frame will be displayed which could be more than 20ms in the future. This is good for *rendering* held objects, but compounds our time problem for *targeting* the object as we calculate the trajectory for the moment the user released their throw. Fortunately, with the November update, when a Unity event like *InteractionSourcePressed* or *InteractionSourceReleased* is sent, the state includes the historical pose data from back when the button was actually pressed or released. To get the most accurate controller rendering and controller targeting during throws, you must correctly use polling and eventing, as appropriate:

- For **controller rendering** each frame, your app should position the controller's *GameObject* at the forward-predicted controller pose for the current frame's photon time. You get this data from Unity polling APIs like [XR.InputTracking.GetLocalPosition](#) or [XR.WSA.Input.InteractionManager.GetCurrentReading](#).
- For **controller targeting** upon a press or release, your app should raycast and calculate trajectories based on the historical controller pose for that press or release event. You get this data from Unity eventing APIs, like [InteractionManager.InteractionSourcePressed](#).
- **Use the grip pose.** Angular velocity and velocity are reported relative to the grip pose, not pointer pose.

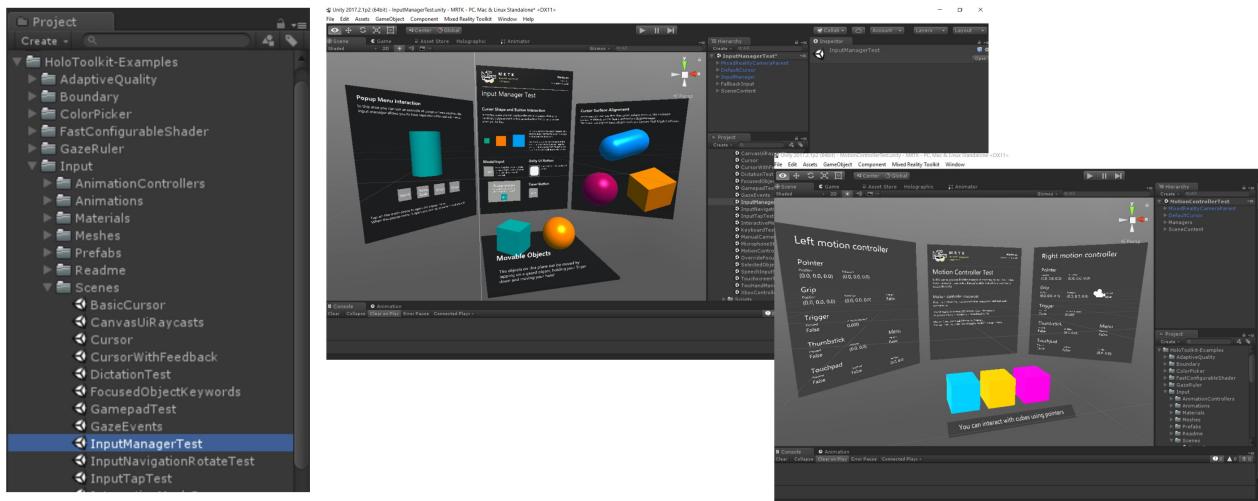
Throwing will continue to improve with future Windows updates, and you can expect to find more information on it here.

Accelerate development with Mixed Reality Toolkit

There are two example scenes about InputManager and MotionController in Unity. Through these scenes, you can learn how to use MRTK's InputManager and access data handle events from the motion controller buttons.

- [HoloToolkit-Examples/Input/Scenes/InputManagerTest.unity](#)

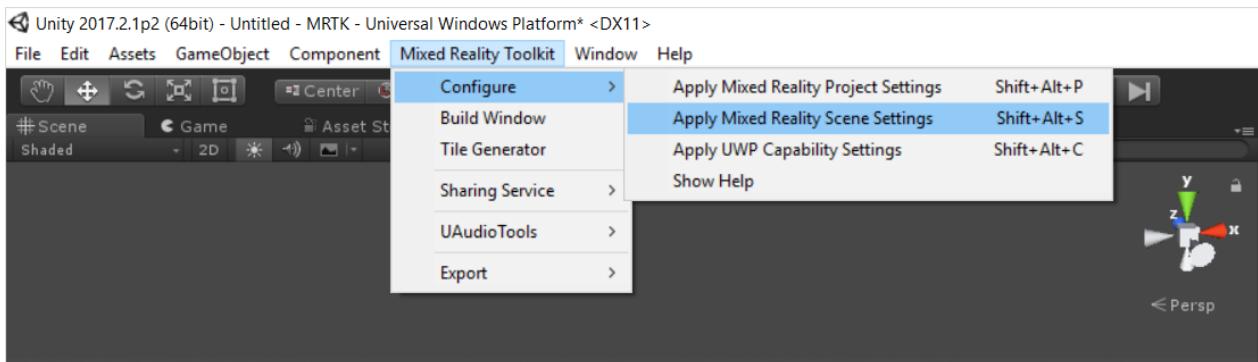
- [HoloToolkit-Examples/Input/Scenes/MotionControllerTest.unity](#)
- [Technical details README File](#)



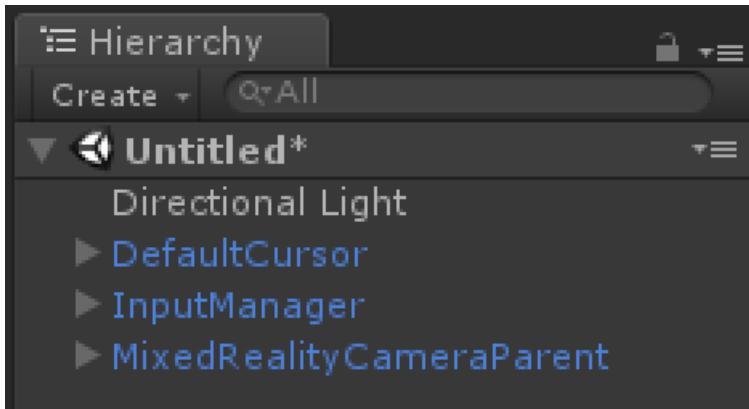
Input example scenes in MRTK

Automatic scene setup

When you import [MRTK release Unity packages](#) or clone the project from the [GitHub repository](#), you are going to find a new menu 'Mixed Reality Toolkit' in Unity. Under 'Configure' menu, you will see the menu 'Apply Mixed Reality Scene Settings'. When you click it, it removes the default camera and adds foundational components - [InputManager](#), [MixedRealityCameraParent](#), and [DefaultCursor](#).



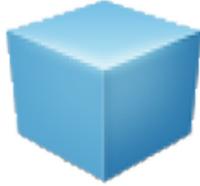
MRTK Menu for scene setup



Automatic scene setup in MRTK

MixedRealityCamera prefab

You can also manually add these from the project panel. You can find these components as prefabs. When you search **MixedRealityCamera**, you will be able to see two different camera prefabs. The difference is, **MixedRealityCamera** is the camera only prefab whereas, **MixedRealityCameraParent** includes additional components for the immersive headsets such as Teleportation, Motion Controller and, Boundary.



MixedRealityCamera.prefab

Camera only



MixedRealityCameraParent.prefab

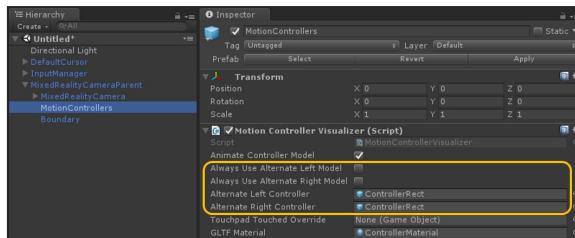
Camera + Motion Controller
+ Teleportation + Boundary



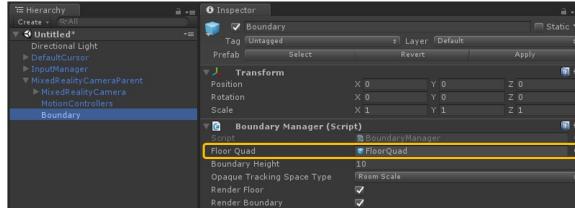
Camera prefabs in MRTK

MixedRealityCamera supports both HoloLens and immersive headset. It detects the device type and optimizes the properties such as clear flags and Skybox. Below you can find some of the useful properties you can customize such as custom Cursor, Motion Controller models, and Floor.

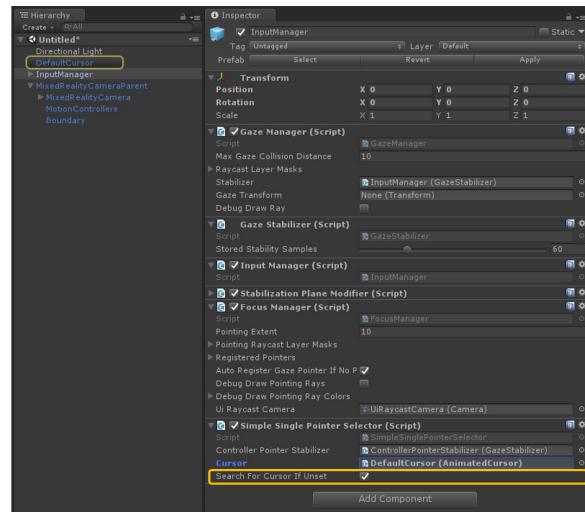
Custom Motion Controller Model



Floor visualization



Cursor

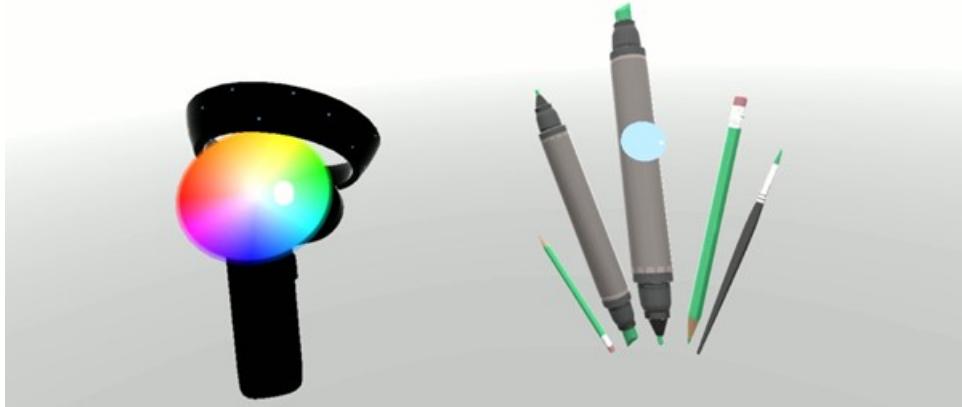


Properties for the Motion controller, Cursor and Floor

Follow along with tutorials

Step-by-step tutorials, with more detailed customization examples, are available in the Mixed Reality Academy:

- [MR Input 211: Gesture](#)
- [MR Input 213: Motion controllers](#)



MR Input 213 - Motion controller

See also

- [Gestures](#)
- [Motion controllers](#)
- [UnityEngine.XR.WSA.Input](#)
- [UnityEngine.XR.InputTracking](#)
- [InteractionInputSource.cs in MixedRealityToolkit-Unity](#)

Voice input in Unity

11/6/2018 • 6 minutes to read • [Edit Online](#)

Unity exposes three ways to add [Voice input](#) to your Unity application.

With the KeywordRecognizer (one of two types of PhraseRecognizers), your app can be given an array of string commands to listen for. With the GrammarRecognizer (the other type of PhraseRecognizer), your app can be given an SRGS file defining a specific grammar to listen for. With the DictationRecognizer, your app can listen for any word and provide the user with a note or other display of their speech.

NOTE

Only dictation or phrase recognition can be handled at once. That means if a GrammarRecognizer or KeywordRecognizer is active, a DictationRecognizer can not be active and vice versa.

Enabling the capability for Voice

The **Microphone** capability must be declared for an app to leverage Voice input.

1. In the Unity Editor, go to the player settings by navigating to "Edit > Project Settings > Player"
2. Click on the "Windows Store" tab
3. In the "Publishing Settings > Capabilities" section, check the **Microphone** capability

Phrase Recognition

To enable your app to listen for specific phrases spoken by the user then take some action, you need to:

1. Specify which phrases to listen for using a KeywordRecognizer or GrammarRecognizer
2. Handle the OnPhraseRecognized event and take action corresponding to the phrase recognized

KeywordRecognizer

Namespace: `UnityEngine.Windows.Speech`

Types: `KeywordRecognizer`, `PhraseRecognizedEventArgs`, `SpeechError`, `SpeechSystemStatus`

We'll need a few using statements to save some keystrokes:

```
using UnityEngine.Windows.Speech;
using System.Collections.Generic;
using System.Linq;
```

Then let's add a few fields to your class to store the recognizer and keyword->action dictionary:

```
KeywordRecognizer keywordRecognizer;
Dictionary<string, System.Action> keywords = new Dictionary<string, System.Action>();
```

Now add a keyword to the dictionary (e.g. inside of a Start() method). We're adding the "activate" keyword in this example:

```
//Create keywords for keyword recognizer
keywords.Add("activate", () =>
{
    // action to be performed when this keyword is spoken
});
```

Create the keyword recognizer and tell it what we want to recognize:

```
keywordRecognizer = new KeywordRecognizer(keywords.Keys.ToArray());
```

Now register for the OnPhraseRecognized event

```
keywordRecognizer.OnPhraseRecognized += KeywordRecognizer_OnPhraseRecognized;
```

An example handler is:

```
private void KeywordRecognizer_OnPhraseRecognized(PhraseRecognizedEventArgs args)
{
    System.Action keywordAction;
    // if the keyword recognized is in our dictionary, call that Action.
    if (keywords.TryGetValue(args.text, out keywordAction))
    {
        keywordAction.Invoke();
    }
}
```

Finally, start recognizing!

```
keywordRecognizer.Start();
```

GrammarRecognizer

Namespace: *UnityEngine.Windows.Speech*

Types: *GrammarRecognizer, PhraseRecognizedEventArgs, SpeechError, SpeechSystemStatus*

The GrammarRecognizer is used if you're specifying your recognition grammar using SRGS. This can be useful if your app has more than just a few keywords, if you want to recognize more complex phrases, or if you want to easily turn on and off sets of commands. See: [Create Grammars Using SRGS XML](#) for file format information.

Once you have your SRGS grammar, and it is in your project in a [StreamingAssets folder](#):

```
<PROJECT_ROOT>/Assets/StreamingAssets/SRGS/myGrammar.xml
```

Create a GrammarRecognizer and pass it the path to your SRGS file:

```
private GrammarRecognizer grammarRecognizer;
grammarRecognizer = new GrammarRecognizer(Application.streamingDataPath + "/SRGS/myGrammar.xml");
```

Now register for the OnPhraseRecognized event

```
grammarRecognizer.OnPhraseRecognized += grammarRecognizer_OnPhraseRecognized;
```

You will get a callback containing information specified in your SRGS grammar which you can handle

appropriately. Most of the important information will be provided in the semanticMeanings array.

```
private void Grammar_OnPhraseRecognized(PhraseRecognizedEventArgs args)
{
    SemanticMeaning[] meanings = args.semanticMeanings;
    // do something
}
```

Finally, start recognizing!

```
grammarRecognizer.Start();
```

Dictation

Namespace: *UnityEngine.Windows.Speech*

Types: *DictationRecognizer, SpeechError, SpeechSystemStatus*

Use the DictationRecognizer to convert the user's speech to text. The DictationRecognizer exposes [dictation](#) functionality and supports registering and listening for hypothesis and phrase completed events, so you can give feedback to your user both while they speak and afterwards. Start() and Stop() methods respectively enable and disable dictation recognition. Once done with the recognizer, it should be disposed using Dispose() method to release the resources it uses. It will release these resources automatically during garbage collection at an additional performance cost if they are not released prior to that.

There are only a few steps needed to get started with dictation:

1. Create a new DictationRecognizer
2. Handle Dictation events
3. Start the DictationRecognizer

Enabling the capability for dictation

The "Internet Client" capability, in addition to the "Microphone" capability mentioned above, must be declared for an app to leverage dictation.

1. In the Unity Editor, go to the player settings by navigating to "Edit > Project Settings > Player" page
2. Click on the "Windows Store" tab
3. In the "Publishing Settings > Capabilities" section, check the **InternetClient** capability

DictationRecognizer

Create a DictationRecognizer like so:

```
dictationRecognizer = new DictationRecognizer();
```

There are four dictation events that can be subscribed to and handled to implement dictation behavior.

1. DictationResult
2. DictationComplete
3. DictationHypothesis
4. DictationError

DictationResult

This event is fired after the user pauses, typically at the end of a sentence. The full recognized string is returned here.

First, subscribe to the DictationResult event:

```
dictationRecognizer.DictationResult += DictationRecognizer_DictationResult;
```

Then handle the DictationResult callback:

```
private void DictationRecognizer_DictationResult(string text, ConfidenceLevel confidence)
{
    // do something
}
```

DictationHypothesis

This event is fired continuously while the user is talking. As the recognizer listens, it provides text of what it's heard so far.

First, subscribe to the DictationHypothesis event:

```
dictationRecognizer.DictationHypothesis += DictationRecognizer_DictationHypothesis;
```

Then handle the DictationHypothesis callback:

```
private void DictationRecognizer_DictationHypothesis(string text)
{
    // do something
}
```

DictationComplete

This event is fired when the recognizer stops, whether from Stop() being called, a timeout occurring, or some other error.

First, subscribe to the DictationComplete event:

```
dictationRecognizer.DictationComplete += DictationRecognizer_DictationComplete;
```

Then handle the DictationComplete callback:

```
private void DictationRecognizer_DictationComplete(DictationCompletionCause cause)
{
    // do something
}
```

DictationError

This event is fired when an error occurs.

First, subscribe to the DictationError event:

```
dictationRecognizer.DictationError += DictationRecognizer_DictationError;
```

Then handle the DictationError callback:

```
private void DictationRecognizer_DictationError(string error, int hresult)
{
    // do something
}
```

Once you have subscribed and handled the dictation events that you care about, start the dictation recognizer to begin receiving events.

```
dictationRecognizer.Start();
```

If you no longer want to keep the DictationRecognizer around, you need to unsubscribe from the events and Dispose the DictationRecognizer.

```
dictationRecognizer.DictationResult -= DictationRecognizer_DictationResult;
dictationRecognizer.DictationComplete -= DictationRecognizer_DictationComplete ;
dictationRecognizer.DictationHypothesis -= DictationRecognizer_DictationHypothesis ;
dictationRecognizer.DictationError -= DictationRecognizer_DictationError ;
dictationRecognizer.Dispose();
```

Tips

- Start() and Stop() methods respectively enable and disable dictation recognition.
- Once done with the recognizer, it must be disposed using Dispose() method to release the resources it uses. It will release these resources automatically during garbage collection at an additional performance cost if they are not released prior to that.
- Timeouts occur after a set period of time. You can check for these timeouts in the DictationComplete event. There are two timeouts to be aware of:
 1. If the recognizer starts and doesn't hear any audio for the first five seconds, it will timeout.
 2. If the recognizer has given a result but then hears silence for twenty seconds, it will timeout.

Using both Phrase Recognition and Dictation

If you want to use both phrase recognition and dictation in your app, you'll need to fully shut one down before you can start the other. If you have multiple KeywordRecognizers running, you can shut them all down at once with:

```
PhraseRecognitionSystem.Shutdown();
```

In order to restore all recognizers to their previous state, after the DictationRecognizer has stopped, you can call:

```
PhraseRecognitionSystem.Restart();
```

You could also just start a KeywordRecognizer, which will restart the PhraseRecognitionSystem as well.

Using the microphone helper

The Mixed Reality Toolkit on GitHub contains a microphone helper class to hint at developers if there is a usable microphone on the system. One use for it is where one would want to check if there is microphone on system before showing any speech interaction hints in the application.

The microphone helper script can be found in the [Input/Scripts/Utilities folder](#). The GitHub repo also contains a [small sample](#) demonstrating how to use the helper.

Voice input in Mixed Reality Toolkit

You can find the examples of the voice input in this scene.

- [HoloToolkit-Examples/Input/Scenes/SpeechInputSource.unity](#)

Spatial mapping in Unity

11/6/2018 • 15 minutes to read • [Edit Online](#)

This topic describes how to use [spatial mapping](#) in your Unity project, retrieving triangle meshes that represent the surfaces in the world around a HoloLens device, for placement, occlusion, room analysis and more.

Unity includes full support for spatial mapping, which is exposed to developers in the following ways:

1. Spatial mapping components available in the MixedRealityToolkit, which provide a convenient and rapid path for getting started with spatial mapping
2. Lower-level spatial mapping APIs, which provide full control and enable more sophisticated application specific customization

To use spatial mapping in your app, the `spatialPerception` capability needs to be set in your AppxManifest.

Setting the `SpatialPerception` capability

In order for an app to consume spatial mapping data, the `SpatialPerception` capability must be enabled.

How to enable the `SpatialPerception` capability:

1. In the Unity Editor, open the "**Player Settings**" pane (Edit > Project Settings > Player)
2. Click on the "**Windows Store**" tab
3. Expand "**Publishing Settings**" and check the "**SpatialPerception**" capability in the "**Capabilities**" list

Note that if you have already exported your Unity project to a Visual Studio solution, you will need to either export to a new folder or manually [set this capability in the AppxManifest in Visual Studio](#).

Spatial mapping also requires a `MaxVersionTested` of at least 10.0.10586.0:

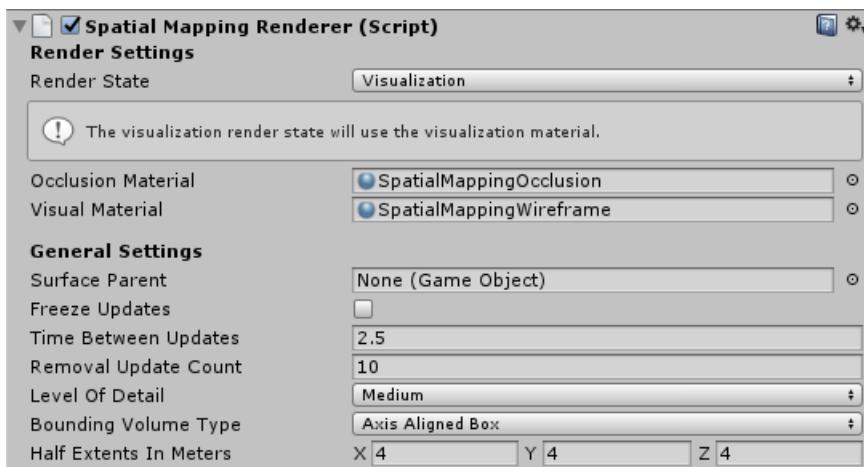
1. In Visual Studio, right click on **Package.appxmanifest** in the Solution Explorer and select **View Code**
2. Find the line specifying **TargetDeviceFamily** and change **MaxVersionTested="10.0.10240.0"** to **MaxVersionTested="10.0.10586.0"**
3. **Save** the Package.appxmanifest.

Getting started with Unity's built-in spatial mapping components

Unity offers 2 components for easily adding spatial mapping to your app, **Spatial Mapping Renderer** and **Spatial Mapping Collider**.

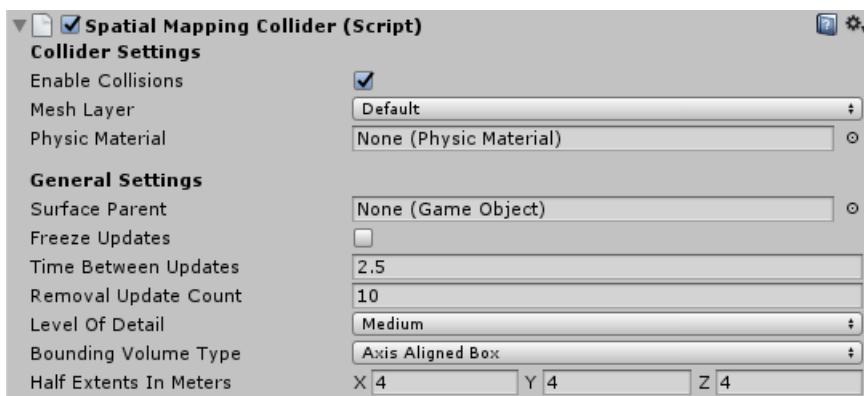
Spatial Mapping Renderer

The Spatial Mapping Renderer allows for visualization of the spatial mapping mesh.



Spatial Mapping Collider

The Spatial Mapping Collider allows for holographic content (or character) interaction, such as physics, with the spatial mapping mesh.



Using the built-in spatial mapping components

You may add both components to your app if you'd like to both visualize and interact with physical surfaces.

To use these two components in your Unity app:

1. Select a GameObject at the center of the area in which you'd like to detect spatial surface meshes.
2. In the Inspector window, **Add Component > XR > Spatial Mapping Collider or Spatial Mapping Renderer**.

You can find more details on how to use these components at the [Unity documentation site](#).

Going beyond the built-in spatial mapping components

These components make it drag-and-drop easy to get started with Spatial Mapping. When you want to go further, there are two main paths to explore:

- To do your own lower-level mesh processing, see the section below about the low-level Spatial Mapping script API.
- To do higher-level mesh analysis, see the section below about the SpatialUnderstanding library in [MixedRealityToolkit](#).

Using the low-level Unity Spatial Mapping API

If you need more control than you get from the Spatial Mapping Renderer and Spatial Mapping Collider components, you can use the low-level Spatial Mapping script APIs.

Namespace: `UnityEngine.XR.WSA`

Types: `SurfaceObserver`, `SurfaceChange`, `SurfaceData`, `SurfaceId`

The following is an outline of the suggested flow for an application that uses the spatial mapping APIs.

Set up the SurfaceObserver(s)

Instantiate one SurfaceObserver object for each application-defined region of space that you need spatial mapping data for.

```
SurfaceObserver surfaceObserver;

void Start () {
    surfaceObserver = new SurfaceObserver();
}
```

Specify the region of space that each SurfaceObserver object will provide data for by calling either SetVolumeAsSphere, SetVolumeAsAxisAlignedBox, SetVolumeAsOrientedBox, or SetVolumeAsFrustum. You can redefine the region of space in the future by simply calling one of these methods again.

```
void Start () {
    ...
    surfaceObserver.SetVolumeAsAxisAlignedBox(Vector3.zero, new Vector3(3, 3, 3));
}
```

When you call SurfaceObserver.Update(), you must provide a handler for each spatial surface in the SurfaceObserver's region of space that the spatial mapping system has new information for. The handler receives, for one spatial surface:

```
private void OnSurfaceChanged(SurfaceId surfaceId, SurfaceChange changeType, Bounds bounds, System.DateTime updateTime)
{
    //see Handling Surface Changes
}
```

Handling Surface Changes

There are several main cases to handle. Added & Updated which can use the same code path and Removed.

- In the Added & Updated cases in the example, we add or get the GameObject representing this mesh from the dictionary, create a SurfaceData struct with the necessary components, then call RequestMeshDataAsync to populate the GameObject with the mesh data and position in the scene.
- In the Removed case, we remove the GameObject representing this mesh from the dictionary and destroy it.

```

System.Collections.Generic.Dictionary<SurfaceId, GameObject> spatialMeshObjects =
    new System.Collections.Generic.Dictionary<SurfaceId, GameObject>();

private void OnSurfaceChanged(SurfaceId surfaceId, SurfaceChange changeType, Bounds bounds, System.DateTime updateTime)
{
    switch (changeType)
    {
        case SurfaceChange.Added:
        case SurfaceChange.Updated:
            if (!spatialMeshObjects.ContainsKey(surfaceId))
            {
                spatialMeshObjects[surfaceId] = new GameObject("spatial-mapping-" + surfaceId);
                spatialMeshObjects[surfaceId].transform.parent = this.transform;
                spatialMeshObjects[surfaceId].AddComponent<MeshRenderer>();
            }
            GameObject target = spatialMeshObjects[surfaceId];
            SurfaceData sd = new SurfaceData(
                //the surface id returned from the system
                surfaceId,
                //the mesh filter that is populated with the spatial mapping data for this mesh
                target.GetComponent<MeshFilter>() ?? target.AddComponent<MeshFilter>(),
                //the world anchor used to position the spatial mapping mesh in the world
                target.GetComponent<WorldAnchor>() ?? target.AddComponent<WorldAnchor>(),
                //the mesh collider that is populated with collider data for this mesh, if true is passed
                to bakeMeshes below
                target.GetComponent<MeshCollider>() ?? target.AddComponent<MeshCollider>(),
                //triangles per cubic meter requested for this mesh
                1000,
                //bakeMeshes - if true, the mesh collider is populated, if false, the mesh collider is
                empty.
                true
            );
            SurfaceObserver.RequestMeshAsync(sd, OnDataReady);
            break;
        case SurfaceChange.Removed:
            var obj = spatialMeshObjects[surfaceId];
            spatialMeshObjects.Remove(surfaceId);
            if (obj != null)
            {
                GameObject.Destroy(obj);
            }
            break;
        default:
            break;
    }
}

```

Handling Data Ready

The OnDataReady handler receives a SurfaceData object. The WorldAnchor, MeshFilter and (optionally) MeshCollider objects it contains reflect the latest state of the associated spatial surface. Optionally perform analysis and/or [processing](#) of the mesh data by accessing the Mesh member of the MeshFilter object. Render the spatial surface with the latest mesh and (optionally) use it for physics collisions and raycasts. It's important to confirm that the contents of the SurfaceData are not null.

Start processing on updates

SurfaceObserver.Update() should be called on a delay, not every frame.

```

void Start () {
    ...
    StartCoroutine(UpdateLoop());
}

IEnumerator UpdateLoop()
{
    var wait = new WaitForSeconds(2.5f);
    while(true)
    {
        surfaceObserver.Update(OnSurfaceChanged);
        yield return wait;
    }
}

```

Higher-level mesh analysis: SpatialUnderstanding

The [MixedRealityToolkit](#) is a collection of helpful utility code for holographic development built upon the holographic Unity APIs.

Spatial Understanding

When placing holograms in the physical world it is often desirable to go beyond spatial mapping's mesh and surface planes. When placement is done procedurally, a higher level of environmental understanding is desirable. This usually requires making decisions about what is floor, ceiling, and walls. In addition, the ability to optimize against a set of placement constraints to determining the most desirable physical locations for holographic objects.

During the development of Young Conker and Fragments, Asobo Studios faced this problem head on, developing a room solver for this purpose. Each of these games had game specific needs, but they shared core spatial understanding technology. The `HoloToolkit.SpatialUnderstanding` library encapsulates this technology, allowing you to quickly find empty spaces on the walls, place objects on the ceiling, identify places for character to sit, and a myriad of other spatial understanding queries.

All of the source code is included, allowing you to customize it to your needs and share your improvements with the community. The code for the C++ solver has been wrapped into a UWP dll and exposed to Unity with a drop in prefab contained within the MixedRealityToolkit.

Understanding Modules

There are three primary interfaces exposed by the module: topology for simple surface and spatial queries, shape for object detection, and the object placement solver for constraint based placement of object sets. Each of these is described below. In addition to the three primary module interfaces, a ray casting interface can be used to retrieve tagged surface types and a custom watertight playspace mesh can be copied out.

Ray Casting

After the room has been scanned and finalized, labels are internally generated for surfaces like the floor, ceiling, and walls. The "PlayspaceRaycast" function takes a ray and returns if the ray collides with a known surface and if so, information about that surface in the form of a "RaycastResult".

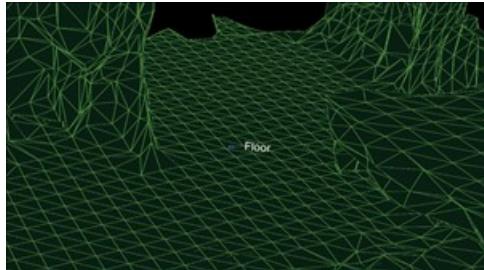
```

struct RaycastResult
{
    enum SurfaceTypes
    {
        Invalid,      // No intersection
        Other,
        Floor,
        FloorLike,   // Not part of the floor topology,
                     // but close to the floor and looks like the floor
        Platform,    // Horizontal platform between the ground and
                     // the ceiling
        Ceiling,
        WallExternal,
        WallLike,    // Not part of the external wall surface,
                     // but vertical surface that looks like a
                     // wall structure
    };
    SurfaceTypes SurfaceType;
    float SurfaceArea; // Zero if unknown
                       // (i.e. if not part of the topology analysis)
    DirectX::XMFLOAT3 IntersectPoint;
    DirectX::XMFLOAT3 IntersectNormal;
};


```

Internally, the raycast is computed against the computed 8cm cubed voxel representation of the playspace. Each voxel contains a set of surface elements with processed topology data (aka surfels). The surfels contained within the intersected voxel cell are compared and the best match used to look up the topology information. This topology data contains the labeling returned in the form of the "SurfaceTypes" enum, as well as the surface area of the intersected surface.

In the Unity sample, the cursor casts a ray each frame. First, against Unity's colliders. Second, against the understanding module's world representation. And finally, again UI elements. In this application, UI gets priority, next the understanding result, and lastly, Unity's colliders. The SurfaceType is reported as text next to the cursor.



Surface type is labeled next to the cursor

Topology Queries

Within the DLL, the topology manager handles labeling of the environment. As mentioned above, much of the data is stored within surfels, contained within a voxel volume. In addition, the "PlaySpaceInfos" structure is used to store information about the playspace, including the world alignment (more details on this below), floor, and ceiling height. Heuristics are used for determining floor, ceiling, and walls. For example, the largest and lowest horizontal surface with greater than 1 m² surface area is considered the floor. Note that the camera path during the scanning process is also used in this process.

A subset of the queries exposed by the Topology manager are exposed out through the dll. The exposed topology queries are as follows.

```
QueryTopology_FindPositionsOnWalls
QueryTopology_FindLargePositionsOnWalls
QueryTopology_FindLargestWall
QueryTopology_FindPositionsOnFloor
QueryTopology_FindLargestPositionsOnFloor
QueryTopology_FindPositionsSittable
```

Each of the queries has a set of parameters, specific to the query type. In the following example, the user specifies the minimum height & width of the desired volume, minimum placement height above the floor, and the minimum amount of clearance in front of the volume. All measurements are in meters.

```
EXTERN_C __declspec(dllexport) int QueryTopology_FindPositionsOnWalls(
    _In_ float minHeightOfWallSpace,
    _In_ float minWidthOfWallSpace,
    _In_ float minHeightAboveFloor,
    _In_ float minFacingClearance,
    _In_ int locationCount,
    _Inout_ Dll_Interface::TopologyResult* locationData)
```

Each of these queries takes a pre-allocated array of "TopologyResult" structures. The "locationCount" parameter specifies the length of the passed in array. The return value reports the number of returned locations. This number is never greater than the passed in "locationCount" parameter.

The "TopologyResult" contains the center position of the returned volume, the facing direction (i.e. normal), and the dimensions of the found space.

```
struct TopologyResult
{
    DirectX::XMFLOAT3 position;
    DirectX::XMFLOAT3 normal;
    float width;
    float length;
};
```

Note that in the Unity sample, each of these queries is linked up to a button in the virtual UI panel. The sample hard codes the parameters for each of these queries to reasonable values. See SpaceVisualizer.cs in the sample code for more examples.

Shape Queries

Inside of the dll, the shape analyzer ("ShapeAnalyzer_W") uses the topology analyzer to match against custom shapes defined by the user. The Unity sample defines a set of shapes and exposes the results out through the in-app query menu, within the shape tab. The intention is that the user can define their own object shape queries and make use of those, as needed by their application.

Note that the shape analysis works on horizontal surfaces only. A couch, for example, is defined by the flat seat surface and the flat top of the couch back. The shape query looks for two surfaces of a specific size, height, and aspect range, with the two surfaces aligned and connected. Using the APIs terminology, the couch seat and back top are shape components and the alignment requirements are shape component constraints.

An example query defined in the Unity sample (ShapeDefinition.cs), for "sittable" objects is as follows.

```

shapeComponents = new List<ShapeComponent>()
{
    new ShapeComponent(
        new List<ShapeComponentConstraint>()
    {
        ShapeComponentConstraint.Create_SurfaceHeight_Between(0.2f, 0.6f),
        ShapeComponentConstraint.Create_SurfaceCount_Min(1),
        ShapeComponentConstraint.Create_SurfaceArea_Min(0.035f),
    }
),
};

AddShape("Sittable", shapeComponents);

```

Each shape query is defined by a set of shape components, each with a set of component constraints and a set of shape constraints which listing dependencies between the components. This example includes three constraints in a single component definition and no shape constraints between components (as there is only one component).

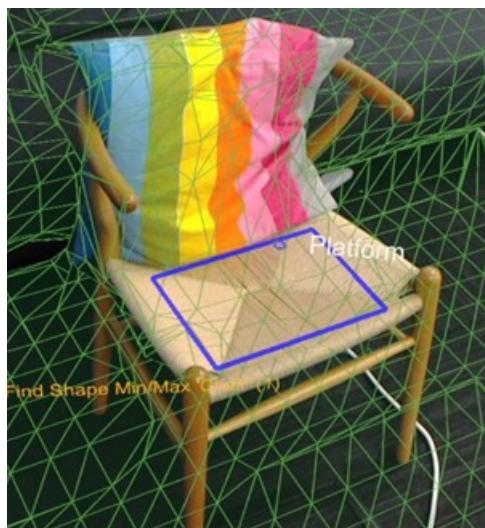
In contrast, the couch shape has two shape components and four shape constraints. Note that components are identified by their index in the user's component list (0 and 1 in this example).

```

shapeConstraints = new List<ShapeConstraint>()
{
    ShapeConstraint.Create_RectanglesSameLength(0, 1, 0.6f),
    ShapeConstraint.Create_RectanglesParallel(0, 1),
    ShapeConstraint.Create_RectanglesAligned(0, 1, 0.3f),
    ShapeConstraint.Create_AtBackOf(1, 0),
};


```

Wrapper functions are provided in the Unity module for easy creation of custom shape definitions. The full list of component and shape constraints can be found in "SpatialUnderstandingDll.cs" within the "ShapeComponentConstraint" and the "ShapeConstraint" structures.



Rectangle shape is found on this surface

Object Placement Solver

The object placement solver can be used to identify ideal locations in the physical room to place your objects. The solver will find the best fit location given the object rules and constraints. In addition, object queries persist until the object is removed with "Solver_RemoveObject" or "Solver_RemoveAllObjects" calls, allowing constrained multi-object placement. Objects placement queries consist of three parts: placement type with parameters, a list of rules, and a list of constraints. To run a query, use the following API.

```

public static int Solver_PlaceObject(
    [In] string objectName,
    [In] IntPtr placementDefinition,           // ObjectPlacementDefinition
    [In] int placementRuleCount,
    [In] IntPtr placementRules,                // ObjectPlacementRule
    [In] int constraintCount,
    [In] IntPtr placementConstraints,         // ObjectPlacementConstraint
    [Out] IntPtr placementResult)

```

This function takes an object name, placement definition, and a list of rules and constraints. The C# wrappers provides construction helper functions to make rule and constraint construction easy. The placement definition contains the query type – that is, one of the following.

```

public enum PlacementType
{
    Place_OnFloor,
    Place_OnWall,
    Place_OnCeiling,
    Place_OnShape,
    Place_OnEdge,
    Place_OnFloorAndCeiling,
    Place_RandomInAir,
    Place_InMidAir,
    Place_UnderFurnitureEdge,
}

```

Each of the placement types has a set of parameters unique to the type. The “ObjectPlacementDefinition” structure contains a set of static helper functions for creating these definitions. For example, to find a place to put an object on the floor, you can use the following function. public static ObjectPlacementDefinition Create_OnFloor(Vector3 halfDims) In addition to the placement type, you can provide a set of rules and constraints. Rules cannot be violated. Possible placement locations that satisfy the type and rules are then optimized against the set of constraints in order to select the optimal placement location. Each of the rules and constraints can be created by the provided static creation functions. An example rule and constraint construction function is provided below.

```

public static ObjectPlacementRule Create_AwayFromPosition(
    Vector3 position, float minDistance)
public static ObjectPlacementConstraint Create_NearPoint(
    Vector3 position, float minDistance = 0.0f, float maxDistance = 0.0f)

```

The below object placement query is looking for a place to put a half meter cube on the edge of a surface, away from other previously placed objects and near the center of the room.

```

List<ObjectPlacementRule> rules =
    new List<ObjectPlacementRule>() {
        ObjectPlacementRule.Create_AwayFromOtherObjects(1.0f),
    };

List<ObjectPlacementConstraint> constraints =
    new List<ObjectPlacementConstraint> {
        ObjectPlacementConstraint.Create_NearCenter(),
    };

Solver_PlaceObject(
    "MyCustomObject",
    new ObjectPlacementDefinition.Create_OnEdge(
        new Vector3(0.25f, 0.25f, 0.25f),
        new Vector3(0.25f, 0.25f, 0.25f)),
    rules.Count,
    UnderstandingDLL.PinObject(rules.ToArray()),
    constraints.Count,
    UnderstandingDLL.PinObject(constraints.ToArray()),
    UnderstandingDLL.GetStaticObjectPlacementResultPtr());

```

If successful, a “ObjectPlacementResult” structure containing the placement position, dimensions and orientation is returned. In addition, the placement is added to the dll’s internal list of placed objects. Subsequent placement queries will take this object into account. The “LevelSolver.cs” file in the Unity sample contains more example queries.

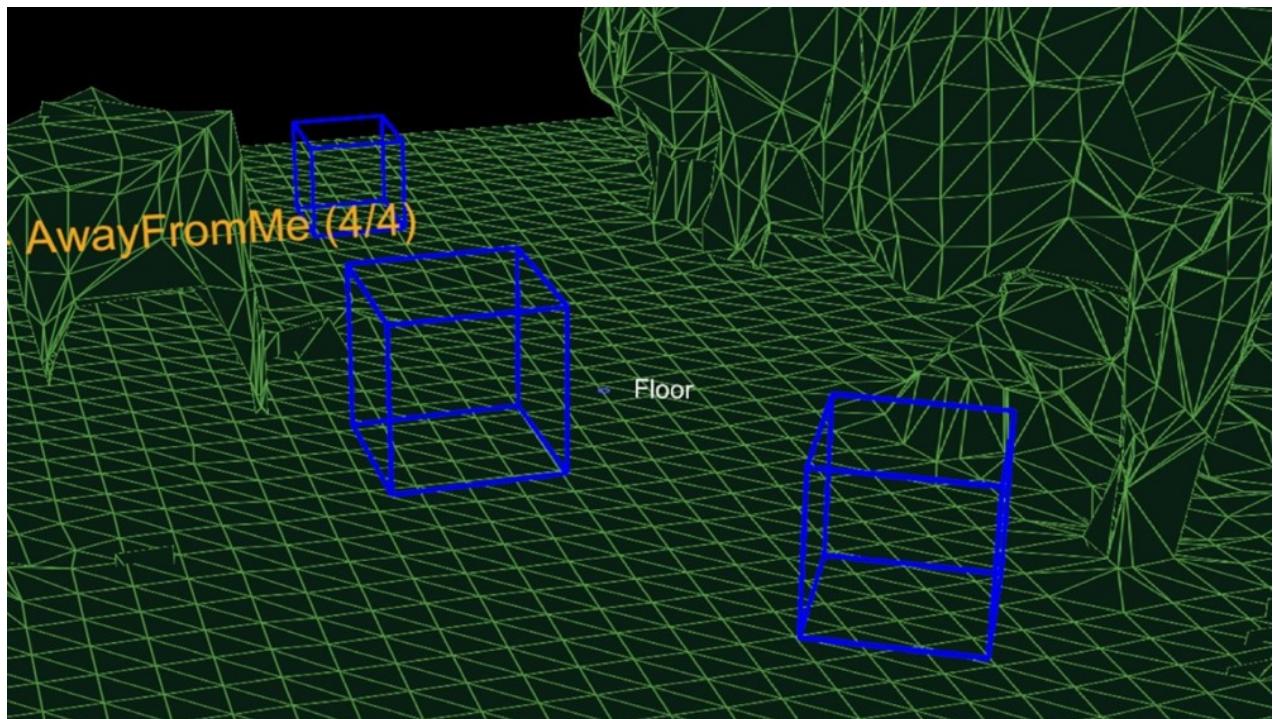


Figure 3: The blue boxes show the result from three place on floor queries with away from camera position rules

When solving for placement location of multiple objects required for a level or application scenario, first solve indispensable and large objects in order to maximizing the probability that a space can be found. Placement order is important. If object placements cannot be found, try less constrained configurations. Having a set of fallback configurations is critical to supporting functionality across many room configurations.

Room Scanning Process

While the spatial mapping solution provided by the HoloLens is designed to be generic enough to meet the needs of the entire gamut of problem spaces, the spatial understanding module was built to support the needs of two specific games. Its solution is structured around a specific process and set of assumptions, summarized below.

Fixed size playspace – The user specifies the maximum playspace size in the init call.

One-time scan process –

The process requires a discrete scanning phase where the user walks around, defining the playspace.

Query functions will not function until after the scan has been finalized.

User driven playspace “painting” – During the scanning phase, the user moves and looks around the playspace, effectively painting the areas which should be included. The generated mesh is important to provide user feedback during this phase. Indoors home or office setup – The query functions are designed around flat surfaces and walls at right angles. This is a soft limitation. However, during the scanning phase, a primary axis analysis is completed to optimize the mesh tessellation along major and minor axis. The included SpatialUnderstanding.cs file manages the scanning phase process. It calls the following functions.

`SpatialUnderstanding_Init` – Called once at the start.

`GeneratePlaySpace_InitScan` – Indicates that the scan phase should begin.

`GeneratePlaySpace_UpdateScan_DynamicScan` –

Called each frame to update the scanning process. The camera position and orientation is passed in and is used for the playspace painting process, described above.

`GeneratePlaySpace_RequestFinish` –

Called to finalize the playspace. This will use the areas “painted” during the scan phase to define and lock the playspace. The application can query statistics during the scanning phase as well as query the custom mesh for providing user feedback.

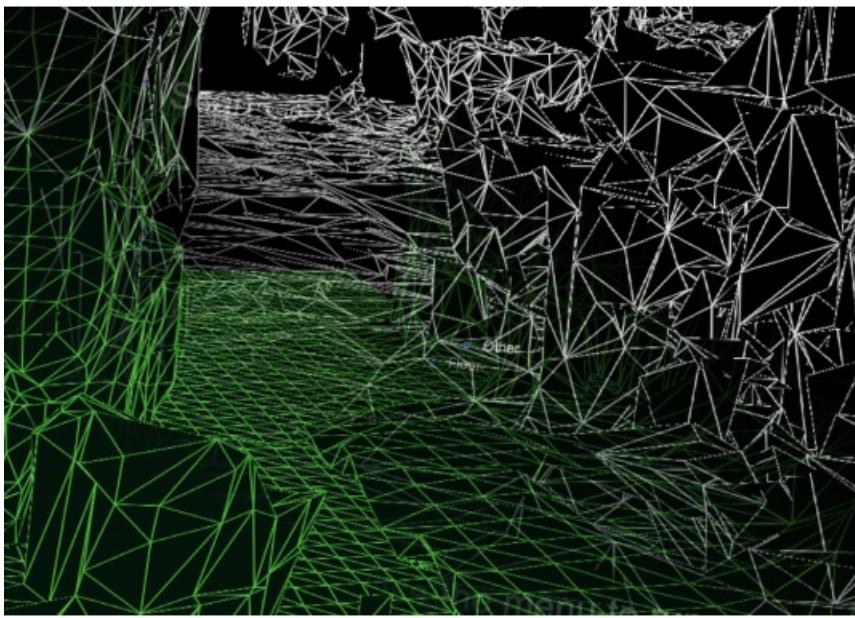
`Import_UnderstandingMesh` –

During scanning, the “SpatialUnderstandingCustomMesh” behavior provided by the module and placed on the understanding prefab will periodically query the custom mesh generated by the process. In addition, this is done once more after scanning has been finalized.

The scanning flow, driven by the “SpatialUnderstanding” behavior calls `InitScan`, then `UpdateScan` each frame. When the statistics query reports reasonable coverage, the user is allowed to airtap to call `RequestFinish` to indicate the end of the scanning phase. `UpdateScan` continues to be called until its return value indicates that the dll has completed processing.

Understanding Mesh

The understanding dll internally stores the playspace as a grid of 8cm sized voxel cubes. During the initial part of scanning, a primary component analysis is completed to determine the axes of the room. Internally, it stores its voxel space aligned to these axes. A mesh is generated approximately every second by extracting the isosurface from the voxel volume.



Generated mesh produced from the voxel volume

Troubleshooting

- Ensure you have set the [SpatialPerception](#) capability
- When tracking is lost, the next OnSurfaceChanged event will remove all meshes.

Spatial Mapping in Mixed Reality Toolkit

There are three example scenes about Spatial mapping in Unity.

- [HoloToolkit-Examples/SpatialMapping/Scenes/SpatialMappingExample.unity](#)
- [HoloToolkit-Examples/SpatialMapping/Scenes/SpatialProcessing.unity](#)
- [HoloToolkit-Examples/SpatialUnderstanding/Scenes/SpatialUnderstandingExample.unity](#)

See also

- [MR Spatial 230: Spatial mapping](#)
- [Coordinate systems](#)
- [Coordinate systems in Unity](#)
- [MixedRealityToolkit](#)
- [UnityEngine.MeshFilter](#)
- [UnityEngine.MeshCollider](#)
- [UnityEngine.Bounds](#)

Spatial sound in Unity

11/6/2018 • 3 minutes to read • [Edit Online](#)

This topic describes how to use Spatial Sound in your Unity projects. It covers the required plugin files as well as the Unity components and properties that enable Spatial Sound.

Enabling Spatial Sound in Unity

Spatial Sound, in Unity, is enabled using an audio spatializer plugin. The plugin files are bundled directly into Unity so enabling spatial sound is as easy as going to **Edit > Project Settings > Audio** and changing the **Spatializer Plugin** in the Inspector to the **MS HRTF Spatializer**. Since the Microsoft spatializer only supports 48000Hz currently, you should also set your System Sample Rate to 48000 to prevent an HRTF failure in the rare case that your system output device is not set to 48000 already:



Inspector for AudioManager

Your Unity project is now configured to use Spatial Sound.

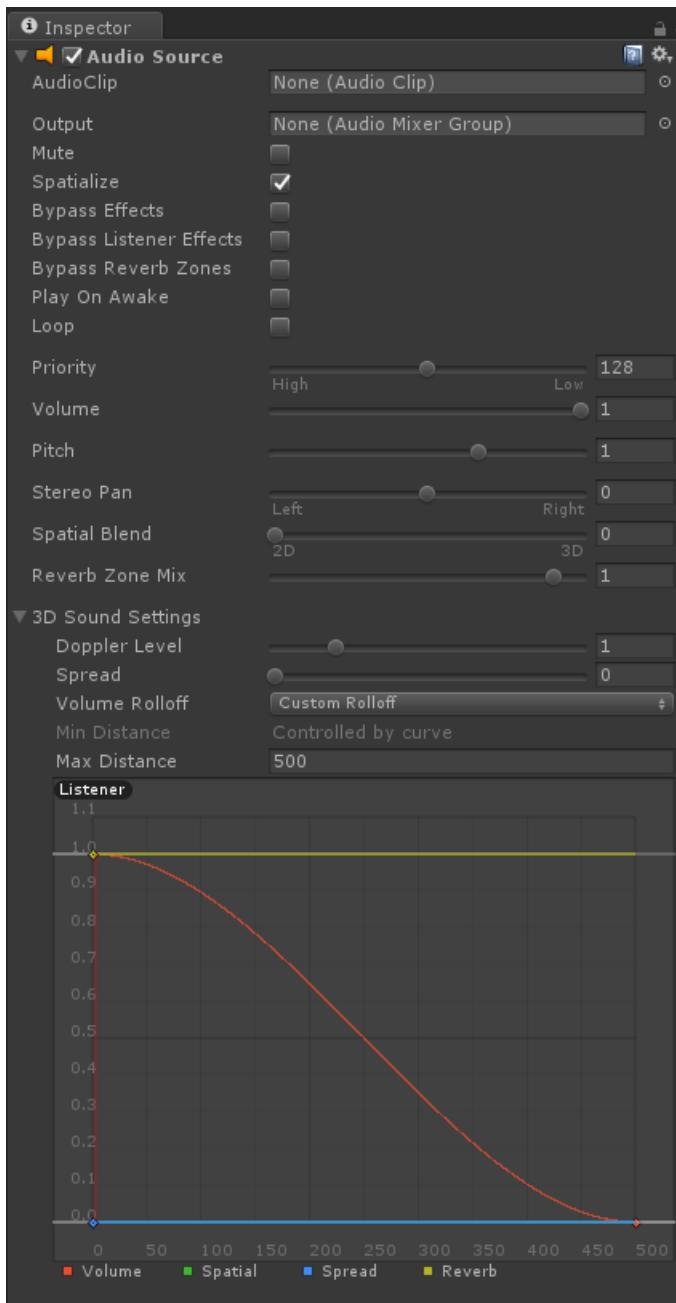
NOTE

If you aren't using a Windows 10 PC for development, you won't get Spatial Sound in the editor nor on the device (even if you're using the Windows 10 SDK).

Using Spatial Sound in Unity

Spatial Sound is used in your Unity project by adjusting three settings on your Audio Source components. The following steps will configure your Audio Source components for Spatial Sound.

- In the **Hierarchy** panel, select the game object that has an attached **Audio Source**.
- In the **Inspector** panel, under the **Audio Source** component
 - Check the **Spatialize** option.
 - Set **Spatial Blend** to **3D** (numeric value 1).
 - For best results, expand **3D Sound Settings** and set **Volume Rolloff** to **Custom Rolloff**.



Inspector panel in Unity showing the Audio Source

Your sounds now realistically exist inside your project's environment!

It is strongly recommended that you become familiar with the [Spatial Sound design guidelines](#). These guidelines help to integrate your audio seamlessly into your project and to further immerse your users into the experience you have created.

Setting Spatial Sound Settings

The Microsoft Spatial Sound plugin provides an additional parameter that can be set, on a per Audio Source basis, to allow additional control of the audio simulation. This parameter is the size of the simulated room.

Room Size

The size of room that is being simulated by Spatial Sound. The approximate sizes of the rooms are; small (an office to a small conference room), medium (a large conference room) and large (an auditorium). You can also specify a room size of none to simulate an outdoor environment. The default room size is small.

Example

The MixedRealityToolkit for Unity provides a static class that makes setting the Spatial Sound settings easy. This class can be found in the [MixedRealityToolkit\SpatialSound folder](#) and can be called from any script in your

project. It is recommended that you set these parameters on each Audio Source component in your project. The following example shows selecting the medium room size for an attached Audio Source.

```
 AudioSource audioSource = gameObject.GetComponent<AudioSource>()

if (audioSource != null) {
    SpatialSoundSettings.SetRoomSize(audioSource, SpatialMappingRoomSizes.Medium);
}
```

Directly Accessing Parameters from Unity

If you don't want to use the excellent Audio tools in the MixedRealityToolkit, here is how you would change HRTF Parameters. You can copy/paste this into a script called *SetHRTF.cs* that you will want to attach to every HRTF AudioSource. It lets you change parameters important to HRTF.

```
using UnityEngine;
using System.Collections;
public class SetHRTF : MonoBehaviour {
    public enum ROOMSIZE { Small, Medium, Large, None };
    public ROOMSIZE room = ROOMSIZE.Small; // Small is regarded as the "most average"
    // defaults and docs from MSDN
    // https://msdn.microsoft.com/library/windows/desktop/mt186602(v=vs.85).aspx
    AudioSource audiosource;
    void Awake()
    {
        audiosource = this.gameObject.GetComponent<AudioSource>();
        if (audiosource == null)
        {
            print("SetHRTFParams needs an audio source to do anything.");
            return;
        }
        audiosource.spatialize = 1; // we DO want spatialized audio
        audiosource.spread = 0; // we dont want to reduce our angle of hearing
        audiosource.spatialBlend = 1; // we do want to hear spatialized audio
        audiosource.SetSpatializerFloat(1, (float)room); // 1 is the roomsize param
    }
}
```

Spatial Sound in Mixed Reality Toolkit

- [HoloToolkit-Examples/SpatialSound/Scenes/UAudioManagerTest.unity](#)

The following examples from the Mixed Reality Toolkit are general audio effect examples that demonstrate ways to make your experiences more immersive by using sound.

- [HoloToolkit-Examples/SpatialSound/Scenes/AudioLoFiTest.unity](#)
- [HoloToolkit-Examples/SpatialSound/Scenes/AudioOcclusionTest.unity](#)

See also

- [Spatial sound](#)
- [Spatial sound design](#)

Shared experiences in Unity

11/6/2018 • 3 minutes to read • [Edit Online](#)

A shared experience is one where multiple users, each with their own HoloLens, collectively view and interact with the same hologram which is positioned at a fixed point in space. This is accomplished through anchor sharing. To use anchor sharing in your app, the *spatialPerception* capability needs to be set in your AppxManifest.

Anchor Sharing

Namespace: `UnityEngine.XR.WSA.Sharing`

Type: `WorldAnchorTransferBatch`

To share a [WorldAnchor](#), one must establish the anchor to be shared. The user of one HoloLens scans their environment and either manually or programmatically chooses a point in space to be the Anchor for the shared experience. The data that represents this point can then be serialized and transmitted to the other devices that are sharing in the experience. Each device then de-serializes the anchor data and attempts to locate that point in space. In order for Anchor Sharing to work, each device must have scanned in enough of the environment such that the point represented by the anchor can be identified.

Setting the SpatialPerception capability

In order for an app to import or export anchor data, the *SpatialPerception* capability must be enabled.

How to enable the *SpatialPerception* capability:

1. In the Unity Editor, open the "**Player Settings**" pane (Edit > Project Settings > Player)
2. Click on the "**Windows Store**" tab
3. Expand "**Publishing Settings**" and check the "**SpatialPerception**" capability in the "**Capabilities**" list

NOTE

If you have already exported your Unity project to a Visual Studio solution, you will need to either export to a new folder or manually [set this capability in the AppxManifest in Visual Studio](#).

Setup

The sample code on this page has a few fields that will need to be initialized:

1. *GameObject rootGameObject* is a *GameObject* in Unity that has a *WorldAnchor* Component on it. One user in the shared experience will place this *GameObject* and export the data to the other users.
2. *WorldAnchor gameRootAnchor* is the `UnityEngine.XR.WSA.WorldAnchor` that is on *rootGameObject*.
3. *byte[] importedData* is a byte array for the serialized anchor each client is receiving over the network.

```

public GameObject rootGameObject;
private UnityEngine.XR.WSA.WorldAnchor gameRootAnchor;

void Start ()
{
    gameRootAnchor = rootGameObject.GetComponent<UnityEngine.XR.WSA.WorldAnchor>();

    if (gameRootAnchor == null)
    {
        gameRootAnchor = rootGameObject.AddComponent<UnityEngine.XR.WSA.WorldAnchor>();
    }
}

```

Exporting

To export, we just need a *WorldAnchor* and to know what we will call it such that it makes sense for the receiving app. One client in the shared experience will perform these steps to export the shared anchor:

1. Create a *WorldAnchorTransferBatch*
2. Add the *WorldAnchors* to transfer
3. Begin the export
4. Handle the *OnExportDataAvailable* event as data becomes available
5. Handle the *OnExportComplete* event

We create a *WorldAnchorTransferBatch* to encapsulate what we will be transferring and then export that into bytes:

```

private void ExportGameRootAnchor()
{
    WorldAnchorTransferBatch transferBatch = new WorldAnchorTransferBatch();
    transferBatch.AddWorldAnchor("gameRoot", this.gameRootAnchor);
    WorldAnchorTransferBatch.ExportAsync(transferBatch, OnExportDataAvailable, OnExportComplete);
}

```

As data becomes available, send the bytes to the client or buffer as segments of data is available and send through whatever means desired:

```

private void OnExportDataAvailable(byte[] data)
{
    TransferDataToClient(data);
}

```

Once the export is complete, if we have been transferring data and serialization failed, tell the client to discard the data. If the serialization succeeded, tell the client that all data has been transferred and importing can start:

```

private void OnExportComplete(SerializationCompletionReason completionReason)
{
    if (completionReason != SerializationCompletionReason.Succeeded)
    {
        SendExportFailedToClient();
    }
    else
    {
        SendExportSucceededToClient();
    }
}

```

Importing

After receiving all of the bytes from the sender, we can import the data back into a `WorldAnchorTransferBatch` and lock our root game object into the same physical location. Note: import will sometimes transiently fail and needs to be retried:

```
// This byte array should have been updated over the network from TransferDataToClient
private byte[] importedData;
private int retryCount = 3;

private void ImportRootGameObject()
{
    WorldAnchorTransferBatch.ImportAsync(importedData, OnImportComplete);
}

private void OnImportComplete(SerializationCompletionReason completionReason, WorldAnchorTransferBatch
deserializedTransferBatch)
{
    if (completionReason != SerializationCompletionReason.Succeeded)
    {
        Debug.Log("Failed to import: " + completionReason.ToString());
        if (retryCount > 0)
        {
            retryCount--;
            WorldAnchorTransferBatch.ImportAsync(importedData, OnImportComplete);
        }
        return;
    }

    this.gameRootAnchor = deserializedTransferBatch.LockObject("gameRoot", this.rootGameObject);
}
```

After a `GameObject` is locked via the `LockObject` call, it will have a `WorldAnchor` which will keep it in the same physical position in the world, but it may be at a different location in the Unity coordinate space than other users.

Example: MixedRealityToolkit Sharing

You can use the [Sharing](#) prefab from the MixedRealityToolkit-Unity repository on GitHub to implement shared experiences in your applications.

See also

- [Shared experiences in mixed reality](#)

Locatable camera in Unity

11/6/2018 • 5 minutes to read • [Edit Online](#)

Enabling the capability for Photo Video Camera

The "WebCam" capability must be declared for an app to use the [camera](#).

1. In the Unity Editor, go to the player settings by navigating to "Edit > Project Settings > Player" page
2. Click on the "Windows Store" tab
3. In the "Publishing Settings > Capabilities" section, check the **WebCam** and **Microphone** capabilities

Only a single operation can occur with the camera at a time. To determine which mode (photo, video, or none) the camera is currently in, you can check `UnityEngine.XR.WSA.WebCam.Mode`.

Photo Capture

Namespace: `UnityEngine.XR.WSA.WebCam`

Type: `PhotoCapture`

The `PhotoCapture` type allows you to take still photographs with the Photo Video Camera. The general pattern for using `PhotoCapture` to take a photo is as follows:

1. Create a `PhotoCapture` object
2. Create a `CameraParameters` object with the settings we want
3. Start Photo Mode via `StartPhotoModeAsync`
4. Take the desired photo
 - (optional) Interact with that picture
5. Stop Photo Mode and clean up resources

Common Set Up for PhotoCapture

For all three uses, we start with the same first 3 steps above

We start by creating a `PhotoCapture` object

```
PhotoCapture photoCaptureObject = null;
void Start()
{
    PhotoCapture.CreateAsync(false, OnPhotoCaptureCreated);
}
```

Next we store our object, set our parameters, and start Photo Mode

```

void OnPhotoCaptureCreated(PhotoCapture captureObject)
{
    photoCaptureObject = captureObject;

    Resolution cameraResolution = PhotoCapture.SupportedResolutions.OrderByDescending((res) => res.width * res.height).First();

    CameraParameters c = new CameraParameters();
    c.hologramOpacity = 0.0f;
    c.cameraResolutionWidth = cameraResolution.width;
    c.cameraResolutionHeight = cameraResolution.height;
    c.pixelFormat = CapturePixelFormat.BGRA32;

    captureObject.StartPhotoModeAsync(c, false, OnPhotoModeStarted);
}

```

In the end, we will also use the same clean up code presented here

```

void OnStoppedPhotoMode(PhotoCapture.PhotoCaptureResult result)
{
    photoCaptureObject.Dispose();
    photoCaptureObject = null;
}

```

After these steps, you can choose which type of photo to capture.

Capture a Photo to a File

The simplest operation is to capture a photo directly to a file. The photo can be saved as a JPG or a PNG.

If we successfully started photo mode, we now will take a photo and store it on disk

```

private void OnPhotoModeStarted(PhotoCapture.PhotoCaptureResult result)
{
    if (result.success)
    {
        string filename = string.Format(@"CapturedImage{0}_n.jpg", Time.time);
        string filePath = System.IO.Path.Combine(Application.persistentDataPath, filename);

        photoCaptureObject.TakePhotoAsync(filePath, PhotoCaptureFileOutputFormat.JPG,
OnCapturedPhotoToDisk);
    }
    else
    {
        Debug.LogError("Unable to start photo mode!");
    }
}

```

After capturing the photo to disk, we will exit photo mode and then clean up our objects

```

void OnCapturedPhotoToDisk(PhotoCapture.PhotoCaptureResult result)
{
    if (result.success)
    {
        Debug.Log("Saved Photo to disk!");
        photoCaptureObject.StopPhotoModeAsync(OnStoppedPhotoMode);
    }
    else
    {
        Debug.Log("Failed to save Photo to disk");
    }
}

```

Capture a Photo to a Texture2D

When capturing data to a Texture2D, the process is extremely similar to capturing to disk.

We will follow the set up process above.

In *OnPhotoModeStarted*, we will capture a frame to memory.

```

private void OnPhotoModeStarted(PhotoCapture.PhotoCaptureResult result)
{
    if (result.success)
    {
        photoCaptureObject.TakePhotoAsync(OnCapturedPhotoToMemory);
    }
    else
    {
        Debug.LogError("Unable to start photo mode!");
    }
}

```

We will then apply our result to a texture and use the common clean up code above.

```

void OnCapturedPhotoToMemory(PhotoCapture.PhotoCaptureResult result, PhotoCaptureFrame photoCaptureFrame)
{
    if (result.success)
    {
        // Create our Texture2D for use and set the correct resolution
        Resolution cameraResolution = PhotoCapture.SupportedResolutions.OrderByDescending((res) =>
res.width * res.height).First();
        Texture2D targetTexture = new Texture2D(cameraResolution.width, cameraResolution.height);
        // Copy the raw image data into our target texture
        photoCaptureFrame.UploadImageDataToTexture(targetTexture);
        // Do as we wish with the texture such as apply it to a material, etc.
    }
    // Clean up
    photoCaptureObject.StopPhotoModeAsync(OnStoppedPhotoMode);
}

```

Capture a Photo and Interact with the Raw bytes

To interact with the raw bytes of an in memory frame, we will follow the same set up steps as above and *OnPhotoModeStarted* as in capturing a photo to a Texture2D. The difference is in *OnCapturedPhotoToMemory* where we can get the raw bytes and interact with them.

In this example, we will create a *List* which could be further processed or applied to a texture via *SetPixels()*

```

void OnCapturedPhotoToMemory(PhotoCapture.PhotoCaptureResult result, PhotoCaptureFrame photoCaptureFrame)
{
    if (result.success)
    {
        List<byte> imageBufferList = new List<byte>();
        // Copy the raw IMFMediaBuffer data into our empty byte list.
        photoCaptureFrame.CopyRawImageDataIntoBuffer(imageBufferList);

        // In this example, we captured the image using the BGRA32 format.
        // So our stride will be 4 since we have a byte for each rgba channel.
        // The raw image data will also be flipped so we access our pixel data
        // in the reverse order.
        int stride = 4;
        float denominator = 1.0f / 255.0f;
        List<Color> colorArray = new List<Color>();
        for (int i = imageBufferList.Count - 1; i >= 0; i -= stride)
        {
            float a = (int)(imageBufferList[i - 0]) * denominator;
            float r = (int)(imageBufferList[i - 1]) * denominator;
            float g = (int)(imageBufferList[i - 2]) * denominator;
            float b = (int)(imageBufferList[i - 3]) * denominator;

            colorArray.Add(new Color(r, g, b, a));
        }
        // Now we could do something with the array such as texture.SetPixels() or run image processing on
        the list
    }
    photoCaptureObject.StopPhotoModeAsync(OnStoppedPhotoMode);
}

```

Video Capture

Namespace: *UnityEngine.XR.WSA.WebCam*

Type: *VideoCapture*

VideoCapture functions very similarly to *PhotoCapture*. The only two differences are that you must specify a Frames Per Second (FPS) value and you can only save directly to disk as an .mp4 file. The steps to use *VideoCapture* are as follows:

1. Create a *VideoCapture* object
2. Create a *CameraParameters* object with the settings we want
3. Start Video Mode via *StartVideoModeAsync*
4. Start recording video
5. Stop recording video
6. Stop Video Mode and clean up resources

We start by creating our *VideoCapture* object *VideoCapture m_VideoCapture = null;*

```

void Start ()
{
    VideoCapture.CreateAsync(false, OnVideoCaptureCreated);
}

```

We then will set up the parameters we will want for the recording and start.

```

void OnVideoCaptureCreated (VideoCapture videoCapture)
{
    if (videoCapture != null)
    {
        m_VideoCapture = videoCapture;

        Resolution cameraResolution = VideoCapture.SupportedResolutions.OrderByDescending((res) =>
res.width * res.height).First();
        float cameraFramerate =
VideoCapture.GetSupportedFrameRatesForResolution(cameraResolution).OrderByDescending((fps) => fps).First();

        CameraParameters cameraParameters = new CameraParameters();
        cameraParameters.hologramOpacity = 0.0f;
        cameraParameters.frameRate = cameraFramerate;
        cameraParameters.cameraResolutionWidth = cameraResolution.width;
        cameraParameters.cameraResolutionHeight = cameraResolution.height;
        cameraParameters.pixelFormat = CapturePixelFormat.BGRA32;

        m_VideoCapture.StartVideoModeAsync(cameraParameters,
                                         VideoCapture.AudioState.None,
                                         OnStartedVideoCaptureMode);
    }
    else
    {
        Debug.LogError("Failed to create VideoCapture Instance!");
    }
}

```

Once started, we will begin the recording

```

void OnStartedVideoCaptureMode(VideoCapture.VideoCaptureResult result)
{
    if (result.success)
    {
        string filename = string.Format("MyVideo_{0}.mp4", Time.time);
        string filepath = System.IO.Path.Combine(Application.persistentDataPath, filename);

        m_VideoCapture.StartRecordingAsync(filepath, OnStartedRecordingVideo);
    }
}

```

After recording has started, you could update your UI or behaviors to enable stopping. Here we just log

```

void OnStartedRecordingVideo(VideoCapture.VideoCaptureResult result)
{
    Debug.Log("Started Recording Video!");
    // We will stop the video from recording via other input such as a timer or a tap, etc.
}

```

At a later point, we will want to stop the recording. This could happen from a timer or user input, for instance.

```

// The user has indicated to stop recording
void StopRecordingVideo()
{
    m_VideoCapture.StopRecordingAsync(OnStoppedRecordingVideo);
}

```

Once the recording has stopped, we will stop video mode and clean up our resources.

```
void OnStoppedRecordingVideo(VideoCapture.VideoCaptureResult result)
{
    Debug.Log("Stopped Recording Video!");
    m_VideoCapture.StopVideoModeAsync(OnStoppedVideoCaptureMode);
}

void OnStoppedVideoCaptureMode(VideoCapture.VideoCaptureResult result)
{
    m_VideoCapture.Dispose();
    m_VideoCapture = null;
}
```

Troubleshooting

- No resolutions are available
 - Ensure the **WebCam** capability is specified in your project.

See Also

- [Locatable camera](#)

Focus point in Unity

11/6/2018 • 2 minutes to read • [Edit Online](#)

Namespace: `UnityEngine.XR.WSA`

Type: `HolographicSettings`

The [focus point](#) can be set to provide HoloLens a hint about how to best perform stabilization on the holograms currently being displayed.

If you want to set the Focus Point in Unity, it needs to be set every frame using `HolographicSettings.SetFocusPointForFrame()`. If the Focus Point is not set for a frame, the default stabilization plane will be used.

NOTE

By default, new Unity projects have the "Enable Depth Buffer Sharing" option set. With this option, a Unity app running on either an immersive desktop headset or a HoloLens running the Windows 10 April 2018 Update (RS4) or later will submit your depth buffer to Windows to optimize hologram stability automatically, without your app specifying a focus point:

- On an immersive desktop headset, this will enable per-pixel depth-based reprojection.
- On a HoloLens running the Windows 10 April 2018 Update or later, this will analyze the depth buffer to pick an optimal stabilization plane automatically.

Either approach should provide even better image quality without explicit work by your app to select a focus point each frame. Note that if you do provide a focus point manually, that will override the automatic behavior described above, and will usually reduce hologram stability. Generally, you should only specify a manual focus point when your app is running on a HoloLens that has not yet been updated to the Windows 10 April 2018 Update.

Example

There are many ways to set the Focus Point, as suggested by the overloads available on the `SetFocusPointForFrame` static function. Presented below is a simple example to set the focus plane to the provided object each frame:

```
public GameObject focusedObject;
void Update()
{
    // Normally the normal is best set to be the opposite of the main camera's
    // forward vector.
    // If the content is actually all on a plane (like text), set the normal to
    // the normal of the plane and ensure the user does not pass through the
    // plane.
    var normal = -Camera.main.transform.forward;
    var position = focusedObject.transform.position;
    UnityEngine.XR.WSA.HolographicSettings.SetFocusPointForFrame(position, normal);
}
```

Note that the simple code above may end up reducing hologram stability if the focused object ends up behind the user. This is why you should generally set "Enable Depth Buffer Sharing" instead of manually specifying a focus point.

See also

- [Stabilization plane](#)

Tracking loss in Unity

11/6/2018 • 2 minutes to read • [Edit Online](#)

When the device cannot locate itself in the world, the app experiences "tracking loss". By default, Unity will pause the update loop and display a splash image to the user. When tracking is regained, the splash image goes away and the update loop continues.

As an alternative, the user can manually handle this transition by opting out of the setting. All content will seem to become body locked during tracking loss if nothing is done to handle it.

Default Handling

By default, the update loop of the app as well as all messages and events will stop for the duration of tracking loss. At that same time, an image will be displayed to the user. You can customize this image by going to **Edit->Settings->Player**, clicking **Splash Image**, and setting the **Holographic Tracking Loss** image.

Manual Handling

To manually handle tracking loss, you need to go to **Edit > Project Settings > Player > Universal Windows Platform settings tab > Splash Image > Windows Holographic** and uncheck "On Tracking Loss Pause and Show Image". After which, you need to handle tracking changes with the APIs specified below.

Namespace: `UnityEngine.XR.WSA`

Type: `WorldManager`

- World Manager exposes an event to detect tracking lost/gained (`WorldManager.OnPositionalLocatorStateChanged`) and a property to query the current state (`WorldManager.state`)
- When the tracking state is not active, the camera will not appear to translate in the virtual world even as the user translates. This means objects will no longer correspond to any physical location and all will appear body locked.

When handling tracking changes on your own you either need to poll for the state property each frame or handle the `OnPositionalLocatorStateChanged` event.

Polling

The most important state is `PositionalLocatorState.Active` which means tracking is fully functional. Any other state will result in only rotational deltas to the main camera. For example:

```

void Update()
{
    switch (UnityEngine.XR.WSA.WorldManager.state)
    {
        case PositionalLocatorState.Active:
            // handle active
            break;
        case PositionalLocatorState.Activating:
        case PositionalLocatorState.Inhibited:
        case PositionalLocatorState.OrientationOnly:
        case PositionalLocatorState.Unavailable:
        default:
            // only rotational information is available
            break;
    }
}

```

Handling the `OnPositionalLocatorStateChanged` event

Alternatively and more conveniently, you can also subscribe to `OnPositionalLocatorStateChanged` to handle the transitions:

```

void Start()
{
    UnityEngine.XR.WSA.WorldManager.OnPositionalLocatorStateChanged += 
    WorldManager_OnPositionalLocatorStateChanged;
}

private void WorldManager_OnPositionalLocatorStateChanged(PositionalLocatorState oldState,
PositionalLocatorState newState)
{
    if (newState == PositionalLocatorState.Active)
    {
        // Handle becoming active
    }
    else
    {
        // Handle becoming rotational only
    }
}

```

See also

- [Handling tracking loss in DirectX](#)

Keyboard input in Unity

11/6/2018 • 3 minutes to read • [Edit Online](#)

Namespace: `UnityEngine`

Type: `TouchScreenKeyboard`

While HoloLens supports many forms of input including Bluetooth keyboards, most applications cannot assume that all users will have a physical keyboard available. If your application requires text input, some form of on screen keyboard should be provided.

Unity provides the `TouchScreenKeyboard` class for accepting keyboard input when there is no physical keyboard available.

HoloLens system keyboard behavior in Unity

On HoloLens, the `TouchScreenKeyboard` leverages the system's on screen keyboard. The system's on screen keyboard is unable to overlay on top of a volumetric view so Unity has to create a secondary 2D XAML view to show the keyboard then return back to the volumetric view once input has been submitted. The user flow goes like this:

1. The user performs an action causing app code to call `TouchScreenKeyboard`
 - The app is responsible for pausing app state before calling `TouchScreenKeyboard`
 - The app may terminate before ever switching back to the volumetric view
2. Unity switches to a 2D XAML view which is auto-placed in the world
3. The user enters text using the system keyboard and submits or cancels
4. Unity switches back to the volumetric view
 - The app is responsible for resuming app state when the `TouchScreenKeyboard` is done
5. Submitted text is available in the `TouchScreenKeyboard`

Available keyboard views

There are six different keyboard views available:

- Single-line textbox
- Single-line textbox with title
- Multi-line textbox
- Multi-line textbox with title
- Single-line password box
- Single-line password box with title

How to enable the system keyboard in Unity

The HoloLens system keyboard is only available to Unity applications that are exported with the "UWP Build Type" set to "XAML". There are tradeoffs you make when you choose "XAML" as the "UWP Build Type" over "D3D". If you aren't comfortable with those tradeoffs, you may wish to explore an [alternative input solution](#) to the system keyboard.

1. Open the **File** menu and select **Build Settings...**
2. Ensure the **Platform** is set to **Windows Store**, the **SDK** is set to **Universal 10**, and set the **UWP Build Type** to **XAML**.
3. In the **Build Settings** dialog, click the **Player Settings...** button

4. Select the **Settings for Windows Store** tab
5. Expand the **Other Settings** group
6. In the **Rendering** section, check the **Virtual Reality Supported** checkbox to add a new **Virtual Reality Devices** list
7. Ensure **Windows Holographic** appears in the list of Virtual Reality SDKs

NOTE

If you don't mark the build as Virtual Reality Supported with the HoloLens device, the project will export as a 2D XAML app.

Using the system keyboard in your Unity app

Declare the keyboard

In the class, declare a variable to store the *TouchScreenKeyboard* and a variable to hold the string the keyboard returns.

```
UnityEngine.TouchScreenKeyboard keyboard;
public static string keyboardText = "";
```

Invoke the keyboard

When an event occurs requesting keyboard input, call one of these functions depending upon the type of input desired. Note that the title is specified in the *textPlaceholder* parameter.

```
// Single-line textbox
keyboard = TouchScreenKeyboard.Open("", TouchScreenKeyboardType.Default, false, false, false, false);

// Single-line textbox with title
keyboard = TouchScreenKeyboard.Open("", TouchScreenKeyboardType.Default, false, false, false, false, "Single-line title");

// Multi-line textbox
keyboard = TouchScreenKeyboard.Open("", TouchScreenKeyboardType.Default, false, true, false, false);

// Multi-line textbox with title
keyboard = TouchScreenKeyboard.Open("", TouchScreenKeyboardType.Default, false, true, false, false, "Multi-line Title");

// Single-line password box
keyboard = TouchScreenKeyboard.Open("", TouchScreenKeyboardType.Default, false, false, true, false);

// Single-line password box with title
keyboard = TouchScreenKeyboard.Open("", TouchScreenKeyboardType.Default, false, false, true, false, "Secure Single-line Title");
```

Retrieve typed contents

In the update loop, check if the keyboard received new input and store it for use elsewhere.

```
if (TouchScreenKeyboard.visible == false && keyboard != null)
{
    if (keyboard.done == true)
    {
        keyboardText = keyboard.text;
        keyboard = null;
    }
}
```

Alternative keyboard options

We understand that switching out of a volumetric view into a 2D view isn't the ideal way to get text input from the user.

The current alternatives to leveraging the system keyboard through Unity include:

- Using speech dictation for input (**Note:** this is often error prone for words not found in the dictionary and is not suitable for password entry)
- Create a keyboard that works in your applications exclusive view

Using the Windows namespace with Unity apps for HoloLens

11/6/2018 • 2 minutes to read • [Edit Online](#)

This page describes how to make use of WinRT APIs in your Unity project for HoloLens.

Conditionally include WinRT API calls

WinRT APIs will only be used in Unity project builds that target Windows 8, Windows 8.1, or the Universal Windows Platform; any code that you write in Unity scripts that targets WinRT APIs must be conditionally included for only those builds. This is done using the NETFX_CORE or WINDOWS_UWP preprocessor definitions. This rule applies to using statements, as well as other code.

The following code snippet is from the Unity manual page for [Universal Windows Platform: WinRT API in C# scripts](#). In this example, an advertising ID is returned, but only on Windows 8.0 or higher target builds:

```
using UnityEngine;
public class WinRTAPI : MonoBehaviour {
    void Update() {
        auto adId = GetAdvertisingId();
        // ...
    }

    string GetAdvertisingId() {
        #if NETFX_CORE
            return Windows.System.UserProfile.AdvertisingManager.AdvertisingId;
        #else
            return "";
        #endif
    }
}
```

Edit your scripts in a Unity C# project

When you double-click a script in the Unity editor, it will by default launch your script in an editor project. The WinRT APIs will appear to be unknown for two reasons: NETFX_CORE is not defined in this environment, and the project does not reference the Windows Runtime. If you use the [recommended export and built settings](#), and edit the scripts in that project instead, it will define NETFX_CORE and also include a reference to the Windows Runtime; with this configuration in place, WinRT APIs will be available for IntelliSense.

Note that your Unity C# project can also be used to debug through your scripts using F5 remote debugging in Visual Studio. If you do not see IntelliSense working the first time that you open your Unity C# project, close the project and re-open it. IntelliSense should start working.

See also

- [Exporting and building a Unity Visual Studio solution](#)

Using Vuforia with Unity

11/6/2018 • 5 minutes to read • [Edit Online](#)

Vuforia brings an important capability to HoloLens – the power to connect AR experiences to specific images and objects in the environment. You can use this capability to overlay guided step by step instructions on top of machinery or to add digital features to a physical product.

Enterprise developers – you can use VuMarks to uniquely identify each piece of machinery on a factory floor – right down to the serial number. VuMarks are scalable into the billions and can be designed to look just like a company logo. They are the ideal solution for adding AR to any product that HoloLens can see.

Existing Vuforia apps that were built for phones and tablets can easily be configured in Unity to run on HoloLens. You can even use Vuforia to take your new HoloLens app to Windows 10 tablets such as the Surface Pro 4 and Surface Book.

Get the tools

Install the recommended versions of Visual Studio and Unity and then configure Unity to use Visual Studio and the preferred IDE and compiler. You will also need to install [Visual Studios tools for Unity](#).

When installing Unity, be sure to install the Windows Store platform, and also the .NET Scripting Backend. Windows Store components can also be installed afterwards from the Build Settings dialog when the Windows Store platform is selected.

Getting started with Vuforia

Vuforia's support for HoloLens is implemented in version 6.1 of the Vuforia Unity extension. The best starting point, to understand the structure and organization of a Vuforia HoloLens project in Unity, is with the [Unity HoloLens sample](#). This provides a complete HoloLens project that includes the Vuforia Unity Extension for Windows 10 as well as a pre-configured scene that can be deployed to a HoloLens.

The scene implements a Vuforia to HoloLens camera binding, along with the build settings necessary to deploy a HoloLens app. It shows how to use image targets and extended tracking to recognize an image and augment it with digital content in a HoloLens experience. You can easily substitute your own content in this scene to begin experimenting with the creation of HoloLens apps that use Vuforia.

Configuring a Vuforia App for HoloLens

Developing a Vuforia app for HoloLens is fundamentally the same as developing Vuforia apps for other devices. You'll simply need to apply a binding between the Vuforia ARCamera and the HoloLens scene camera, and enable extended tracking on your targets. You can then apply the build settings described in the Building and Executing a Vuforia app for HoloLens section. That's all that's needed to enable Vuforia to work with the HoloLens spatial mapping and positional tracking systems.

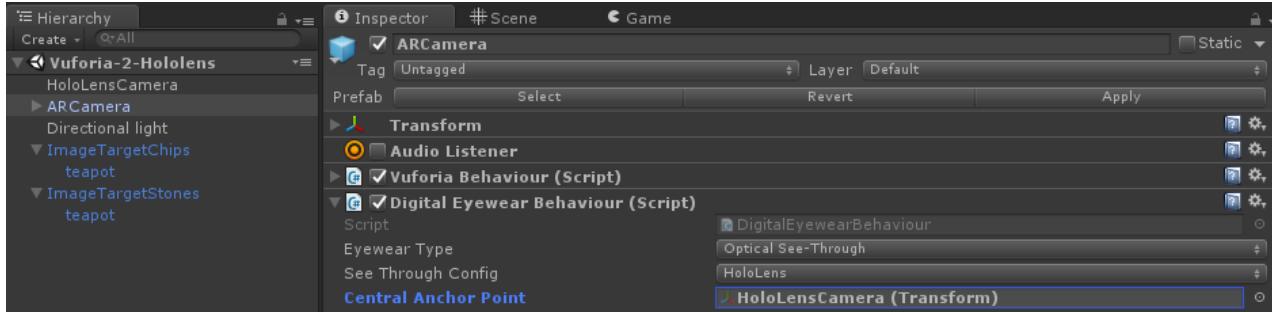
1. Enable **Extended Tracking** on your targets
2. Bind the **ARCamera** to the HoloLens scene camera

Binding the HoloLens scene camera

Vuforia uses the ARCamera prefab as its scene camera in a Unity project. You'll need to bind the scene camera used by HoloLens to the ARCamera to enable Vuforia to interact with HoloLens. The [Vuforia HoloLens sample](#) shows how to configure the **scene camera** used by HoloLens.

Follow these steps to bind the HoloLens scene camera to the Vuforia ARCamera:

1. Add an ARCamera and HoloLens scene camera to your scene Hierarchy
2. Set up the scene for stereo rendering in the ARCamera's digital eyewear behavior component.
 - Eyewear type = Optical See-Through
3. Select HoloLens as the device configuration
 - See Through Config = HoloLens
4. Bind the HoloLens scene camera to the ARCamera
 - Drag the HoloLens Scene camera onto the Central Anchor Point field



Unity editor window showing the HoloLensCamera selected

Building and executing a Vuforia app for HoloLens

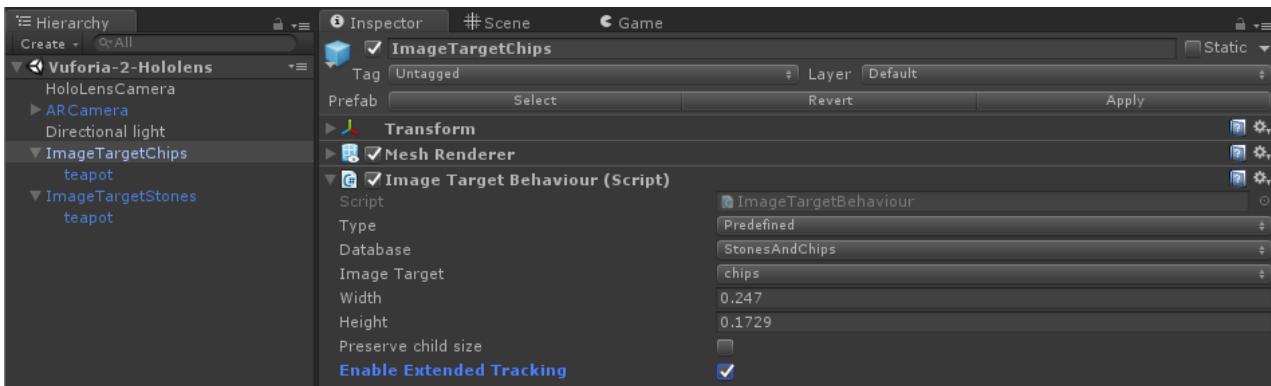
1. [Add an Eyewear App License Key](#) in the ARCamera Inspector
2. Apply the recommended [Unity engine options for power and performance](#)
3. Add the sample scenes to **Scenes in Build**.
4. Set your platform build target for Windows Store in **File > Build Settings**.
5. Select the following platform build configuration settings
 - SDK = Universal10
 - UWP Build Type = D3D
6. Define a unique **Product Name**, in **Player Settings**, to serve as the name of the app when installed on the HoloLens.
7. Select **Landscape Left** as the in **Player Settings > Resolution and Presentation**
8. Check **Virtual Reality Supported + Windows Holographic** in **Player Settings > Other Settings**
9. Check the following Capabilities in **Player Settings > Publish Settings**
 - InternetClient
 - WebCam
 - SpatialPerception - if you intend to use the Surface Observer API
10. Select Build to generate a Visual Studio project
11. Build the executable from Visual Studio and install it on your HoloLens

IMPORTANT

Visual Studio Build Configuration: Be sure to set your build target for x86. Note that the EAP release supports only 32 bit builds.

Extended tracking with Vuforia

[Extended tracking](#) creates a map of the environment to maintain tracking even when a target is no longer in view. It is Vuforia's counterpart to the spatial mapping performed by HoloLens. When you enable extended tracking on a target, you enable the pose of that target to be passed to the spatial mapping system. In this way, targets can exist in both the Vuforia and HoloLens spatial coordinate systems, though not simultaneously.



Unity settings window

Enabling Extended Tracking on a Target

Vuforia will automatically transform the pose of a target that uses extended tracking into the HoloLens spatial coordinate system. This allows HoloLens to take over tracking, and to integrate any content augmenting the target into the spatial map of the target's surroundings. This process occurs between the Vuforia SDK and mixed reality APIs in Unity, and does not require any programming by the developer - it's handled automatically.

Here is what occurs...

1. Vuforia's target Tracker recognizes the target
2. Target tracking is then initialized
3. The position and rotation of the target are analyzed to provide a robust pose estimate for HoloLens to use
4. Vuforia transforms the target's pose into the HoloLens spatial mapping coordinate space
5. HoloLens takes over tracking and the Vuforia tracker is deactivated

The developer can control this process, to return control to Vuforia, by disabling extended tracking on the TargetBehaviour.

See also

- [Install the tools](#)
- [Coordinate systems](#)
- [Spatial mapping](#)
- [Camera in Unity](#)
- [Exporting and building a Unity Visual Studio solution](#)
- [Vuforia documentation: Developing for Windows 10 in Unity](#)
- [Vuforia documentation: How to install the Vuforia Unity extension](#)
- [Vuforia documentation: Working with the HoloLens sample in Unity](#)
- [Vuforia documentation: Extended tracking in Vuforia](#)

DirectX development overview

11/6/2018 • 2 minutes to read • [Edit Online](#)

Windows Mixed Reality apps use the [holographic rendering, gaze, gesture, motion controller, voice](#) and [spatial mapping](#) APIs to build [mixed reality](#) experiences for HoloLens and immersive headsets. You can create mixed reality apps using a 3D engine, such as [Unity](#), or you can use Windows Mixed Reality APIs with DirectX 11. Please note that DirectX 12 is not currently supported. If you are leveraging the platform directly, you'll essentially be building your own middleware or framework. The Windows APIs support apps written in both C++ and C#. If you'd like to use C#, your application can leverage the [SharpDX](#) open source software library.

Windows Mixed Reality supports [two kinds of apps](#):

- **Mixed reality apps** (UWP or Win32), which use the [HolographicSpace API](#) to render an [immersive view](#) to the user that fills the headset display.
- **2D apps** (UWP), which use DirectX, XAML or other frameworks to render [2D views](#) on slates in the Windows Mixed Reality home.

The differences between DirectX development for [2D views](#) and [immersive views](#) are primarily related to holographic rendering and spatial input. Your UWP app's [IFrameworkView](#) or your Win32 app's [HWND](#) are still required and remain largely the same, as do the WinRT APIs available to your app. However, you use a different subset of these APIs to take advantage of holographic features. For example, the swap chain is managed by the system for holographic apps, and you work with the [HolographicSpace API](#) rather than [DXGI](#) to [present frames](#).

To begin developing immersive apps:

- For **UWP apps**, [create a new UWP project using the templates in Visual Studio](#). Based on your language, [Visual C++](#) or [Visual C#](#), you will find the UWP templates under **Windows Universal > Holographic**.
- For **Win32 apps**, [create a new Win32 desktop project](#) and then follow the Win32 instructions on the [Getting a HolographicSpace](#) page to get a [HolographicSpace](#).

This is a great way to get the code you need to add holographic rendering support to an existing app or engine. Code and concepts are presented in the template in a way that's familiar to any developer of real-time interactive software.

Getting started

The following topics discuss the base requirements of adding Windows Mixed Reality support to DirectX-based middleware:

- [Creating a holographic DirectX project](#): The holographic app template coupled with the documentation will show you the differences between what you're used to, and the special requirements introduced by a device that's designed to function while resting on your head.
- [Getting a HolographicSpace](#): You'll first need to create a [HolographicSpace](#), which will provide your app the sequence of [HolographicFrame](#) objects that represent each head position from which you'll render.
- [Rendering in DirectX](#): Since a holographic swap chain has two render targets, you'll likely need to make some changes to the way your application renders.
- [Coordinate systems in DirectX](#): Windows Mixed Reality learns and updates its understanding of the world as the user walks around, providing spatial coordinate systems that apps use to reason about the user's surroundings, including spatial anchors and the user's defined spatial stage.

Adding mixed reality capabilities and inputs

To enable the best possible experience for users of your immersive apps, you'll want to support the following key building blocks:

- [Gaze, gestures, and motion controllers in DirectX](#)
- [Voice input in DirectX](#)
- [Spatial sound in DirectX](#)
- [Spatial mapping in DirectX](#)

There are other key features that many immersive apps will want to use, which are also exposed to DirectX apps:

- [Shared spatial anchors in DirectX](#)
- [Locatable camera in DirectX](#)
- [Keyboard, mouse, and controller input in DirectX](#)

See also

- [App model](#)
- [App views](#)

Creating a holographic DirectX project

11/6/2018 • 5 minutes to read • [Edit Online](#)

A holographic app can be a [Universal Windows Platform app](#) or a Win32 app.

The DirectX 11 holographic UWP app template is much like the DirectX 11 UWP app template; it includes a program loop (or "game loop"), a **DeviceResources** class to manage the Direct3D device and context, and a simplified content renderer class. It also has an [IFrameworkView](#), just like any other UWP app.

The mixed reality app, however, has some additional capabilities that aren't present in a typical D3D11 UWP app. The Windows Holographic app template is able to:

- Handle Direct3D device resources associated with holographic cameras.
- Retrieve camera back buffers from the system.
- Handle [gaze](#) input, and recognize a simple [gesture](#).
- Go into full-screen stereo rendering mode.

How do I get started?

First [install the tools](#), and then follow the instructions on downloading Visual Studio 2017 and the Microsoft HoloLens emulator. The holographic app templates are included in the same installer as the Microsoft HoloLens emulator. Also ensure that the option to install the templates is selected before installing.

Now you can create your DirectX 11 Windows Mixed Reality App! Note, to remove the sample content, comment out the **DRAW_SAMPLE_CONTENT** preprocessor directive in *pch.h*.

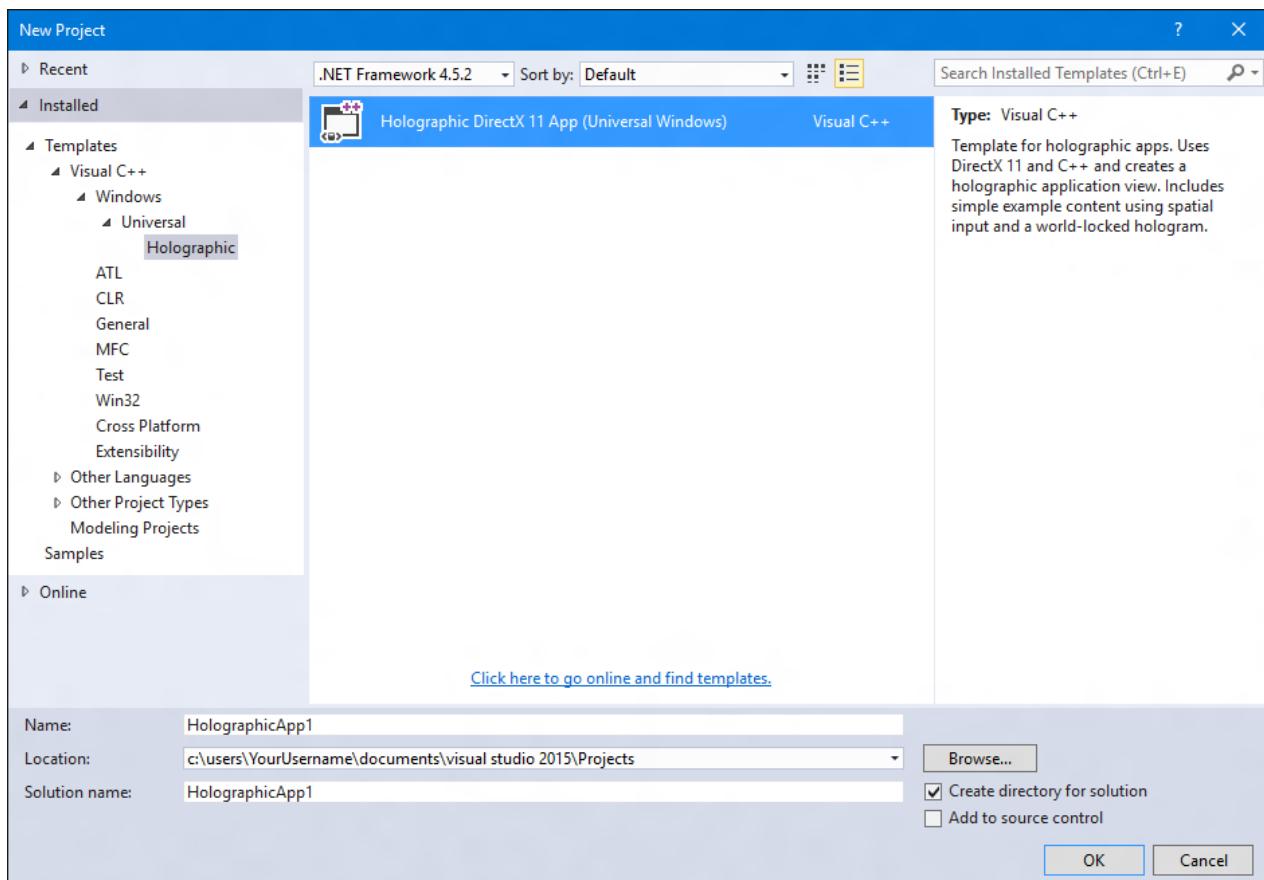
Creating the project

NOTE

The rest of the instructions in this article describe how to create a UWP holographic project using C++/CX. C++/WinRT sample code for both UWP and Win32 projects is coming soon. To create a Win32 holographic project, skip to the [Creating a Win32 project](#) section.

Once the tools are installed you can then create a holographic DirectX project. To create a new project:

1. Start **Visual Studio**.
2. From the **File** menu, point to **New** and select **Project** from the context menu. The **New Project** dialog opens.
3. Expand **Installed** on the left and expand either the **Visual C#** or **Visual C++** language node.
4. Navigate to the **Windows Universal > Holographic** node and select **Holographic DirectX 11 App (Universal Windows)**.
5. Fill in the **Name** and **Location** text boxes, and click or tap **OK**. The holographic app project is created.



Holographic app project template in Visual Studio

Review [Using Visual Studio to deploy and debug](#) for information on how to build and deploy the sample to your HoloLens, PC with immersive device attached, or an emulator.

The project template shows how to create a world-locked cube that's placed two meters from the user. The user can [air-tap](#), or press a button on the controller, to place the cube in a different position that's specified by the user's [gaze](#). You can modify this project to create any mixed reality app.

Note that if your holographic C# project did not start from the Windows Holographic app template, you will need to copy the ms.fxcompile.targets file from a Windows Mixed Reality C# template project and import it in your .csproj file in order to compile HLSL files that you add to your project.

Holographic app entry point

Your holographic app starts in the **main** function in AppView.cpp. The **main** function creates the app's [IFrameworkView](#) and starts the [CoreApplication](#) with it.

From **AppView.cpp**

```
// The main function is only used to initialize our IFrameworkView class.  
// Under most circumstances, you should not need to modify this function.  
[Platform::MTAThread]  
int main(Platform::Array<Platform::String^>^)  
{  
    AppViewSource^ appViewSource = ref new ::AppViewSource();  
    CoreApplication::Run(appViewSource);  
    return 0;  
}
```

From that point on, the AppView class handles interaction with Windows basic input events, CoreWindow events and messaging, and so on. It will also create the HolographicSpace used by your app.

Render holographic content

The project's **Content** folder contains classes for rendering holograms in the [holographic space](#). The default hologram in the template is a spinning cube that's placed two meters away from the user. Drawing this cube is implemented in **SpinningCubeRenderer.cpp**, which has these key methods:

METHOD	EXPLANATION
<code>CreateDeviceDependentResources</code>	Loads shaders and creates the cube mesh.
<code>PositionHologram</code>	Places the hologram at the location specified by the provided SpatialPointerPose .
<code>Update</code>	Rotates the cube, and sets the model matrix.
<code>Render</code>	Renders a frame using the vertex and pixel shaders.

The **Shaders** folder contains four default shader implementations:

SHADER	EXPLANATION
<code>GeometryShader.hlsl</code>	A pass-through that leaves the geometry unmodified.
<code>PixelShader.hlsl</code>	Passes through the color data. The color data is interpolated and assigned to a pixel at the rasterization step.
<code>VertexShader.hlsl</code>	Simple shader to do vertex processing on the GPU.
<code>VPRTVertexShader.hlsl</code>	Simple shader to do vertex processing on the GPU, that is optimized for Windows Mixed Reality stereo rendering.

The shaders are compiled when the project is built, and they're loaded in the

SpinningCubeRenderer::CreateDeviceDependentResources method.

Interact with your holograms

User input is processed in the **SpatialInputHandler** class, which gets a [SpatialInteractionManager](#) instance and subscribes to the [SourcePressed](#) event. This enables detecting the air-tap gesture and other spatial input events.

Update holographic content

Your mixed reality app updates in a game loop, which by default is implemented in the **Update** method in `Main.cpp`. The **Update** method updates scene objects, like the spinning cube, and returns a [HolographicFrame](#) object that is used to get up-to-date view and projection matrices and to present the swap chain.

The **Render** method in `Main.cpp` takes the [HolographicFrame](#) and renders the current frame to each holographic camera, according to the current app and spatial positioning state.

Creating a Win32 project

NOTE

A full Win32 holographic project template is coming soon.

To create a Win32 instead of UWP holographic project, you can adapt the standard Win32 desktop project template by following these steps:

1. Start **Visual Studio**.
2. From the **File** menu, point to **New** and select **Project** from the context menu. The **New Project** dialog opens.
3. Expand **Installed** on the left and expand the **Visual C++** language node.
4. Navigate to the **Windows Desktop** node and select **Windows Desktop Application**.
5. Fill in the **Name** and **Location** text boxes, and click or tap **OK**. The Win32 app project is created.

You can now adapt this Win32 desktop application to make its primary HWND holographic instead. To do so, follow the Win32 instructions on the [Getting a HolographicSpace](#) page to get a HolographicSpace for your main HWND. After that, you'll use the HolographicSpace in the same way in your Win32 project as you would in a UWP project.

See also

- [Getting a HolographicSpace](#)
- [HolographicSpace](#)
- [Rendering in DirectX](#)
- [Using Visual Studio to deploy and debug](#)
- [Using the HoloLens emulator](#)
- [Using XAML with holographic DirectX apps](#)

Getting a HolographicSpace

11/6/2018 • 7 minutes to read • [Edit Online](#)

The [HolographicSpace](#) class is your portal into the holographic world. It controls full-screen rendering, provides camera data, and provides access to spatial reasoning APIs. You will create one for your UWP app's [CoreWindow](#) or your Win32 app's HWND.

Set up the holographic space

Creating the holographic space object is the first step in making your Windows Mixed Reality app. Traditional Windows apps render to a Direct3D swap chain created for the core window of their application view. This swap chain is displayed to a slate in the holographic UI. To make your application view holographic rather than a 2D slate, create a holographic space for its core window instead of a swap chain. Presenting holographic frames that are created by this holographic space puts your app into full-screen rendering mode.

For a **UWP app** starting from the *Holographic DirectX 11 App (Universal Windows)* template, look for this code in the **SetWindow** method in AppView.cpp:

```
m_holographicSpace = HolographicSpace::CreateForCoreWindow(window);
```

The current holographic space is used in multiple places in the DirectX template:

- The **DeviceResources** class needs to get some information from the HolographicSpace object in order to create the Direct3D device. This is the DXGI adapter ID associated with the holographic display. The [HolographicSpace](#) class uses your app's Direct3D 11 device to create and manage device-based resources, like swap chains for holographic cameras. If you're interested in seeing what this function does under the hood, you'll find it in DeviceResources.cpp.
- The function **DeviceResources::InitializeUsingHolographicSpace** demonstrates how to obtain the adapter by looking up the LUID – and how to choose a default adapter when no preferred adapter is specified.
- The main class uses the holographic space in **AppView::SetWindow**, for updates and rendering.

For a **Win32 app** starting from the *Windows Desktop Application* template, add the following code just below the #include directives at the top of the main .cpp file named for your project:

```
#include <..\winrt\WinRTBase.h>;
#include <windows.graphics.holographic.h>;
#include <..\um\HolographicSpaceInterop.h>;
#include <wrl.h>;
```

Then, add the following code block within the **InitInstance** function, just above the call to **ShowWindow**:

```

{
    CoInitializeEx(nullptr, 0);

    using namespace ABI::Windows::Foundation;
    using namespace ABI::Windows::Graphics::Holographic;
    using namespace Microsoft::WRL;
    using namespace Microsoft::WRL::Wrappers;

    ComPtr<IHolographicSpaceStatics> spHolographicSpaceFactory;
    HRESULT hr =
GetActivationFactory(HStringReference(RuntimeClass_Windows_Graphics_HolographicSpace).Get(),
&spHolographicSpaceFactory);

    ComPtr<IHolographicSpaceInterop> spHolographicSpaceInterop;
    if (SUCCEEDED(hr))
    {
        hr = spHolographicSpaceFactory.As(&spHolographicSpaceInterop);
    }

    ComPtr<ABI::Windows::Graphics::Holographic::IHolographicSpace> spHolographicSpace;
    if (SUCCEEDED(hr))
    {
        hr = spHolographicSpaceInterop->CreateForWindow(hWnd, IID_PPV_ARGS(&spHolographicSpace));
    }

    ComPtr<IHolographicFrame> spHolographicFrame;
    if (SUCCEEDED(hr))
    {
        hr = spHolographicSpace->CreateNextFrame(&spHolographicFrame);
    }
}

```

Now that you've obtained a HolographicSpace for either your UWP CoreWindow or Win32 HWND, you'll use that HolographicSpace to handle holographic cameras, create coordinate systems and do holographic rendering. The code you'll write there is the same regardless of whether you're writing a UWP app or Win32 app.

NOTE

While the sections below mention functions like **AppView::SetWindow** that assume that you are using the holographic UWP app template, the code snippets shown apply equally across UWP and Win32 apps.

Next, we'll dive into the setup process that **SetHolographicSpace** is responsible for in the AppMain class.

Subscribe to camera events, create and remove camera resources

Your app's holographic content lives in its holographic space, and is viewed through one or more holographic cameras which represent different perspectives on the scene. Now that you have the holographic space, you can receive data for holographic cameras.

Your app needs to respond to **CameraAdded** events by creating any resources that are specific to that camera, like your back buffer render target view. Register this function before the app creates any holographic frames in **AppView::SetWindow**:

```

m_cameraAddedToken =
    m_holographicSpace->CameraAdded +=
        ref new Windows::Foundation::TypedEventHandler<HolographicSpace^,
HolographicSpaceCameraAddedEventArgs^>(
            std::bind(&AppMain::OnCameraAdded, this, _1, _2)
        );

```

Your app also needs to respond to **CameraRemoved** events by releasing resources that were created for that camera.

From **AppView::SetWindow**:

```
m_cameraRemovedToken =
    m_holographicSpace->CameraRemoved +=  
    ref new Windows::Foundation::TypedEventHandler<HolographicSpace^,  
HolographicSpaceCameraRemovedEventArgs^>(  
        std::bind(&AppMain::OnCameraRemoved, this, _1, _2)  
    );
```

The event handlers must complete some work in order to keep holographic rendering flowing smoothly, and so that your app is able to render at all. Read the code and comments for the details: you can look for **CameraAdded** and **CameraRemoved** in your main class to understand how the **m_cameraResources** map is handled by **DeviceResources**.

Right now, we're focused on AppMain and the setup that it does to enable your app to know about holographic cameras. With this in mind, it's important to take note of the following two requirements:

1. For the **CameraAdded** event handler, the app can work asynchronously to finish creating resources and loading assets for the new holographic camera. Apps that take more than one frame to complete this work should request a deferral, and complete the deferral after loading asynchronously; a **PPL task** can be used to do asynchronous work. Your app must ensure that it's ready to render to that camera right away when it exits the event handler, or when it completes the deferral. Exiting the event handler or completing the deferral tells the system that your app is now ready to receive holographic frames with that camera included.
2. When the app receives a **CameraRemoved** event, it must release all references to the back buffer and exit the function right away. This includes render target views, and any other resource that might hold a reference to the **IDXGIResource**. The app must also ensure that the back buffer is not attached as a render target, as shown in **DeviceResources::ReleaseResourcesForBackBuffer**. To help speed things along, your app can release the back buffer and then launch a task to asynchronously complete any other work that is necessary to tear down that camera. The holographic app template includes a PPL task that you can use for this purpose.

NOTE

If you want to determine when an added or removed camera shows up on the frame, use the **HolographicFrame AddedCameras** and **RemovedCameras** properties.

Create a frame of reference for your holographic content

Your app's content must be positioned in a spatial coordinate system in order to be rendered. The system provides two primary frames of reference which you can use to establish a coordinate system for your holograms.

There are two kinds of reference frames in Windows Holographic: reference frames attached to the device, and reference frames that remain stationary as the device moves through the user's environment. The holographic app template uses a stationary reference frame by default; this is one of the simplest ways to render world-locked holograms.

Stationary reference frames are designed to stabilize positions near the device's current location. This means that **coordinates** further from the device are allowed to drift slightly with respect to the user's environment as the device learns more about the space around it. There are two ways to create a stationary frame of reference:

acquire the coordinate system from the [spatial stage](#), or use the default [SpatialLocator](#). If you are creating a Windows Mixed Reality app for immersive headsets, the recommended starting point is the [spatial stage](#), which also provides information about the capabilities of the immersive headset worn by the player. Here, we show how to use the default [SpatialLocator](#).

The spatial locator represents the Windows Mixed Reality device, and tracks the motion of the device and provides coordinate systems that can be understood relative to its location.

From [AppMain::SetHolographicSpace](#):

```
m_locator = SpatialLocator::GetDefault();
```

Create the stationary reference frame once when the app is launched. This is analogous to defining a world coordinate system, with the origin placed at the device's position when the app is launched. This reference frame doesn't move with the device.

From [AppMain::SetHolographicSpace](#):

```
SpatialStationaryFrameOfReference m_referenceFrame =
    m_locator->CreateStationaryFrameOfReferenceAtCurrentLocation();
```

All reference frames are gravity aligned, meaning that the y axis points "up" with respect to the user's environment. Since Windows uses "right-handed" coordinate systems, the direction of the -z axis coincides with the direction the device is facing when the reference frame is created.

NOTE

When your app requires precise placement of individual holograms, use a [SpatialAnchor](#) to anchor the individual hologram to a position in the real world. For example, use a spatial anchor when the user indicates a point to be of special interest. Anchor positions do not drift, but they can be adjusted. By default, when an anchor is adjusted, it eases its position into place over the next several frames after the correction has occurred. Depending on your application, when this occurs you may want to handle the adjustment in a different manner (e.g. by deferring it until the hologram is out of view). The [RawCoordinateSystem](#) property and [RawCoordinateSystemAdjusted](#) events enable these customizations.

Respond to locatability changed events

Rendering world-locked holograms requires the device to be able to locate itself in the world. This may not always be possible due to environmental conditions, and if so, the user may expect a visual indication of the tracking interruption. This visual indication must be rendered using reference frames attached to the device, instead of stationary to the world.

Your app can request to be notified if tracking is interrupted for any reason. Register for the [LocatablilityChanged](#) event to detect when the device's ability to locate itself in the world changes. From

[AppMain::SetHolographicSpace](#):

```
m_locatabilityChangedToken =
    m_locator->LocatabilityChanged +=
        ref new Windows::Foundation::TypedEventHandler<SpatialLocator^, Object^>(
            std::bind(&AppMain::OnLocatabilityChanged, this, _1, _2)
        );
```

Then use this event to determine when holograms cannot be rendered stationary to the world.

See also

- [Rendering in DirectX](#)
- [Coordinate systems in DirectX](#)

Rendering in DirectX

11/6/2018 • 23 minutes to read • [Edit Online](#)

Windows Mixed Reality is built on DirectX to produce rich, 3D graphical experiences for users. The rendering abstraction sits just above DirectX and lets an app reason about the position and orientation of one or more observers of a holographic scene, as predicted by the system. The developer can then locate their holograms relative to each camera, letting the app render these holograms in various spatial coordinate systems as the user moves around.

Update for the current frame

To update the application state for holograms, once per frame the app will:

- Get a [HolographicFrame](#) from the display management system.
- Update the scene with the current prediction of where the camera view will be when render is completed.
Note, there can be more than one camera for the holographic scene.

To render to holographic camera views, once per frame the app will:

- For each camera, render the scene for the current frame, using the camera view and projection matrices from the system.

Create a new holographic frame and get its prediction

The [HolographicFrame](#) has information that the app needs in order to update and render the current frame. The app begins each new frame by calling the **CreateNextFrame** method. When this method is called, predictions are made using the latest sensor data available, and encapsulated in **CurrentPrediction** object.

A new frame object must be used for each rendered frame as it is only valid for an instant in time. The **CurrentPrediction** property contains information such as the camera position. The information is extrapolated to the exact moment in time when the frame is expected to be visible to the user.

The following code is excerpted from the AppMain::Update() method:

```
HolographicFrame^ holographicFrame = m_holographicSpace->CreateNextFrame();

// Get a prediction of where holographic cameras will be when this frame is presented.
HolographicFramePrediction^ prediction = holographicFrame->CurrentPrediction;
```

Process camera updates

Back buffers can change from frame to frame. Your app needs to validate the back buffer for each camera, and release and recreate resource views and depth buffers as needed. Notice that the set of poses in the prediction is the authoritative list of cameras being used in the current frame. Usually, you use this list to iterate on the set of cameras.

From **AppMain::Update**

```
m_deviceResources->EnsureCameraResources(holographicFrame, prediction);
```

From **DeviceResources::EnsureCameraResources**

```

for (HolographicCameraPose^ pose : prediction->CameraPoses)
{
    HolographicCameraRenderingParameters^ renderingParameters = frame->GetRenderingParameters(pose);
    CameraResources* pCameraResources = cameraResourceMap[pose->HolographicCamera->Id].get();
    pCameraResources->CreateResourcesForBackBuffer(this, renderingParameters);
}

```

Get the coordinate system to use as a basis for rendering

Windows Mixed Reality uses a coordinate system from the attached reference frame that's associated with the current frame to render. Later, this coordinate system is used for creating the stereo view matrices when rendering the sample content.

From **AppMain::Update**

```
SpatialCoordinateSystem^ currentCoordinateSystem = m_referenceFrame->CoordinateSystem;
```

Process gaze and gesture input

Gaze and [gesture](#) input are not time-based and thus do not have to update in the **StepTimer** function. However [this input](#) is something that the app needs to look at each frame.

Process time-based updates

Any real-time rendering app will need some way to process time-based updates; we provide a way to do this in the Windows Holographic app template via a **StepTimer** implementation. This is similar to the StepTimer provided in the DirectX 11 UWP app template, so if you already have looked at that template you should be on familiar ground. This StepTimer sample helper class is able to provide fixed time-step updates, as well as variable time-step updates, and the default mode is variable time steps.

In the case of holographic rendering, we've specifically chosen not to put too much into the timer function. This is because you can configure it to be a fixed time step, in which case it might get called more than once per frame – or not at all, for some frames – and our holographic data updates should happen once per frame.

From **AppMain::Update**

```
m_timer.Tick([&] ())
{
    m_spinningCubeRenderer->Update(m_timer);
});
```

Position and rotate holograms in your coordinate system

If you are operating in a single coordinate system, as the template does with the **SpatialStationaryReferenceFrame**, this process isn't different from what you're used to in 3D graphics. Here, we rotate the cube and set the model matrix relative to the position in the stationary coordinate system.

From **SpinningCubeRenderer::Update**

```

// Rotate the cube.
// Convert degrees to radians, then convert seconds to rotation angle.
const float    radiansPerSecond = XMConvertToRadians(m_degreesPerSecond);
const double   totalRotation   = timer.GetTotalSeconds() * radiansPerSecond;
const float    radians        = static_cast<float>(fmod(totalRotation, XM_2PI));
const XMATRIX modelRotation  = XMMatrixRotationY(-radians);

// Position the cube.
const XMATRIX modelTranslation = XMMatrixTranslationFromVector(XMLoadFloat3(&m_position));

// Multiply to get the transform matrix.
// Note that this transform does not enforce a particular coordinate system. The calling
// class is responsible for rendering this content in a consistent manner.
const XMATRIX modelTransform  = XMMatrixMultiply(modelRotation, modelTranslation);

// The view and projection matrices are provided by the system; they are associated
// with holographic cameras, and updated on a per-camera basis.
// Here, we provide the model transform for the sample hologram. The model transform
// matrix is transposed to prepare it for the shader.
XMStoreFloat4x4(&m_modelConstantBufferData.model, XMMatrixTranspose(modelTransform));

```

Note about advanced scenarios: The spinning cube is a very simple example of how to position a hologram within a single reference frame. It's also possible to [use multiple SpatialCoordinateSystems](#) in the same rendered frame, at the same time.

Update constant buffer data

Model transforms for content are updated as usual. By now, you will have computed valid transforms for the coordinate system you'll be rendering in.

From `SpinningCubeRenderer::Update`

```

// Update the model transform buffer for the hologram.
context->UpdateSubresource(
    m_modelConstantBuffer.Get(),
    0,
    nullptr,
    &m_modelConstantBufferData,
    0,
    0);

```

What about view and projection transforms? For best results, we want to wait until we're almost ready for our draw calls before we get these.

Set the focus point for image stabilization

To keep holograms where a developer or a user puts them in the world, Windows Mixed Reality includes features for [image stabilization](#). Image stabilization uses reduced-latency rendering to ensure the best holographic experiences for users; a focus point may be specified to enhance image stabilization even further, or a depth buffer may be provided to compute optimized image stabilization in real time.

For best results, your app should provide a depth buffer using the [CommitDirect3D11DepthBuffer](#) API. Windows Mixed Reality can then use geometry information from the depth buffer to optimize image stabilization in real time. In the following sample code, we modify the Render method from the Windows Mixed Reality app template to commit the depth buffer. Note that this also required adding an accessor to the CameraResources class, so that we could acquire the depth buffer texture resource for handoff to the API.

Code for `CommitDirect3D11DepthBuffer` example

```

// Up-to-date frame predictions enhance the effectiveness of image stabilization and
// allow more accurate positioning of holograms.

```

```

holographicFrame->UpdateCurrentPrediction();
HolographicFramePrediction^ prediction = holographicFrame->CurrentPrediction;

for (auto cameraPose : prediction->CameraPoses)
{
    // This represents the device-based resources for a HolographicCamera.
    DX::CameraResources* pCameraResources = cameraResourceMap[cameraPose->HolographicCamera->Id].get();

    // Get the device context.
    const auto context = m_deviceResources->GetD3DDeviceContext();
    const auto depthStencilView = pCameraResources->GetDepthStencilView();

    // Set render targets to the current holographic camera.
    ID3D11RenderTargetView *const targets[1] = { pCameraResources->GetBackBufferRenderTargetView() };
    context->OMSetRenderTargets(1, targets, depthStencilView);

    // Clear the back buffer and depth stencil view.
    context->ClearRenderTargetView(targets[0], DirectX::Colors::Transparent);
    context->ClearDepthStencilView(depthStencilView, D3D11_CLEAR_DEPTH | D3D11_CLEAR_STENCIL, 1.0f, 0);

    // The view and projection matrices for each holographic camera will change
    // every frame. This function refreshes the data in the constant buffer for
    // the holographic camera indicated by cameraPose.
    SpatialCoordinateSystem^ currentCoordinateSystem = m_spatialStageManager-
>GetCoordinateSystemForCurrentStage();
    if (currentCoordinateSystem != nullptr)
    {
        pCameraResources->UpdateViewProjectionBuffer(m_deviceResources, cameraPose, currentCoordinateSystem);
    }

    // Attach the view/projection constant buffer for this camera to the graphics pipeline.
    bool cameraActive = pCameraResources->AttachViewProjectionBuffer(m_deviceResources);

    // Only render world-locked content when positional tracking is active.
    if (cameraActive)
    {
        // Draw the sample hologram.
        m_sceneController->Render();

        // We complete the frame by using the depth buffer to optimize image stabilization.
        ComPtr<ID3D11Texture2D> depthBuffer = cameraResourceMap.at(cameraPose->HolographicCamera->Id)->GetDepthBufferTexture2D();

        // The depth buffer might be stereo, so we take the first subresource.
        // This should also work for mono cameras.
        ComPtr<IDXGIResource1> depthStencilResource;
        HRESULT hr = depthBuffer.As(&depthStencilResource);
        ComPtr<IDXGISurface2> depthDxgiSurface;
        if (SUCCEEDED(hr))
        {
            hr = depthStencilResource->CreateSubresourceSurface(0, &depthDxgiSurface);
        }

        if (SUCCEEDED(hr))
        {
            IDirect3DSurface^ depthD3DSurface = CreateDirect3DSurface(depthDxgiSurface.Get());
            auto renderingParameters = holographicFrame->GetRenderingParameters(cameraPose);
            try
            {
                // Provide the depth buffer.
                renderingParameters->CommitDirect3D11DepthBuffer(depthD3DSurface);
            }
            catch (Platform::InvalidArgumentException^ ex)
            {
                OutputDebugStringA("Unsupported depth buffer format, invalid properties, or incorrect D3D
device.\n");
            }
        }
    }
}

```

```
}
```

NOTE

Windows will process your depth texture on the GPU, so it must be possible to use your depth buffer as a shader resource. The ID3D11Texture2D that you create should be in a typeless format and it should be bound as a shader resource view. Here is an example of how to create a depth texture that can be committed for image stabilization.

Code for **Depth buffer resource creation for CommitDirect3D11DepthBuffer**

```
// Create a depth stencil view for use with 3D rendering if needed.  
CD3D11_TEXTURE2D_DESC depthStencilDesc(  
    DXGI_FORMAT_R16_TYPELESS,  
    static_cast<UINT>(m_d3dRenderTargetSize.Width),  
    static_cast<UINT>(m_d3dRenderTargetSize.Height),  
    m_isStereo ? 2 : 1, // Create two textures when rendering in stereo.  
    1, // Use a single mipmap level.  
    D3D11_BIND_DEPTH_STENCIL | D3D11_BIND_SHADER_RESOURCE);  
  
DX::ThrowIfFailed(  
    device->CreateTexture2D(  
        &depthStencilDesc,  
        nullptr,  
        &m_d3dDepthBuffer));  
  
CD3D11_DEPTH_STENCIL_VIEW_DESC depthStencilViewDesc(  
    m_isStereo ? D3D11_DSV_DIMENSION_TEXTURE2DARRAY : D3D11_DSV_DIMENSION_TEXTURE2D,  
    DXGI_FORMAT_D16_UNORM);  
DX::ThrowIfFailed(  
    device->CreateDepthStencilView(  
        m_d3dDepthBuffer.Get(),  
        &depthStencilViewDesc,  
        &m_d3dDepthStencilView));
```

If running on a device such as HoloLens that does not yet support CommitDirect3D11DepthBuffer, your app can also set a manual focus point for image stabilization itself. In the Windows Mixed Reality app template, we check to see if the CommitDirect3D11DepthBuffer method is unavailable, and if so we set the focus point manually to the center of the spinning cube. Note that the focus point (or depth buffer) is provided separately for each camera.

Code for **HoloLens image stabilization (AppMain::Update)**

```

if (!m_canCommitDirect3D11DepthBuffer)
{
    // On versions of the platform that do not support the CommitDirect3D11DepthBuffer API, we can control
    // image stabilization by setting a focus point with optional plane normal and velocity.
    for (auto cameraPose : prediction->CameraPoses)
    {
        // The HolographicCameraRenderingParameters class provides access to set
        // the image stabilization parameters.
        HolographicCameraRenderingParameters^ renderingParameters = holographicFrame-
>GetRenderingParameters(cameraPose);

        // SetFocusPoint informs the system about a specific point in your scene to
        // prioritize for image stabilization. The focus point is set independently
        // for each holographic camera.
        // You should set the focus point near the content that the user is looking at.
        // In this example, we put the focus point at the center of the sample hologram,
        // since that is the only hologram available for the user to focus on.
        // You can also set the relative velocity and facing of that content; the sample
        // hologram is at a fixed point so we only need to indicate its position.
        renderingParameters->SetFocusPoint(
            currentCoordinateSystem,
            m_spinningCubeRenderer->GetPosition()
        );
    }
}

```

When setting a manual focus point on HoloLens, you can also define the image plane by specifying a normal. In this case, the plane passes through the focus point and is perpendicular to the normal you provide. You can also define a velocity for your focus plane to better stabilize moving holograms. For example, [tag-along hologram](#) sample sets the velocity of the hologram as it follows you around:

Code for **Focus point for a tag-along HoloLens hologram**

```

if (!m_canCommitDirect3D11DepthBuffer)
{
    // On versions of the platform that do not support the CommitDirect3D11DepthBuffer API, we can control
    // image stabilization by setting a focus point with optional plane normal and velocity.
    for (auto cameraPose : prediction->CameraPoses)
    {
        // The HolographicCameraRenderingParameters class provides access to set
        // the image stabilization parameters.
        HolographicCameraRenderingParameters^ renderingParameters = holographicFrame-
>GetRenderingParameters(cameraPose);

        // SetFocusPoint informs the system about a specific point in your scene to
        // prioritize for image stabilization. The focus point is set independently
        // for each holographic camera.
        // In this example, we set position, normal, and velocity for a tag-along quad.
        float3& focusPointPosition = m_stationaryQuadRenderer->GetPosition();
        float3 focusPointNormal = -normalize(focusPointPosition);
        float3& focusPointVelocity = m_stationaryQuadRenderer->GetVelocity();
        renderingParameters->SetFocusPoint(
            currentCoordinateSystem,
            focusPointPosition,
            focusPointNormal,
            focusPointVelocity
        );
    }
}

```

As the hologram moves around the user, image stabilization takes into account the relative velocity, avoiding issues such as color separation even though the hologram and the user are both moving at the same time.

Render the current frame

Rendering on Windows Mixed Reality is not much different from rendering on a 2D mono display, but there are some differences you need to be aware of:

- Holographic frame predictions are important. The closer the prediction is to when your frame is presented, the better your holograms will look.
- Windows Mixed Reality controls the camera views. You need to render to each one because the holographic frame will be presenting them for you later.
- Stereo rendering is recommended to be accomplished using instanced drawing to a render target array. The holographic app template uses the recommended approach of instanced drawing to a render target array, which uses a render target view onto a **Texture2DArray**.
- If you want to render without using stereo instancing, you will need to create two non-array `RenderTargetViews` (one for each eye) that each reference one of the two slices in the **Texture2DArray** provided to the app from the system. This is not recommended, as it is typically significantly slower than using instancing.

Get an updated `HolographicFrame` prediction

Updating the frame prediction enhances the effectiveness of image stabilization and allows for more accurate positioning of holograms due to the shorter time between the prediction and when the frame is visible to the user. Ideally update your frame prediction just before rendering.

```
holographicFrame->UpdateCurrentPrediction();
HolographicFramePrediction^ prediction = holographicFrame->CurrentPrediction;
```

Render to each camera

Loop on the set of camera poses in the prediction, and render to each camera in this set.

Set up your rendering pass

Windows Mixed Reality uses stereoscopic rendering to enhance the illusion of depth and to render stereoscopically, so both the left and the right display are active. With stereoscopic rendering there is an offset between the two displays, which the brain can reconcile as actual depth. This section covers stereoscopic rendering using [instancing](#), using code from the Windows Holographic app template.

Each camera has its own render target (back buffer), and view and projection matrices, into the holographic space. Your app will need to create any other camera-based resources - such as the depth buffer - on a per-camera basis. In the Windows Holographic app template, we provide a helper class to bundle these resources together in `DX::CameraResources`. Start by setting up the render target views:

From `AppMain::Render`

```
// This represents the device-based resources for a HolographicCamera.
DX::CameraResources* pCameraResources = cameraResourceMap[cameraPose->HolographicCamera->Id].get();

// Get the device context.
const auto context = m_deviceResources->GetD3DDeviceContext();
const auto depthStencilView = pCameraResources->GetDepthStencilView();

// Set render targets to the current holographic camera.
ID3D11RenderTargetView *const targets[1] = { pCameraResources->GetBackBufferRenderTargetView() };
context->OMSetRenderTargets(1, targets, depthStencilView);

// Clear the back buffer and depth stencil view.
context->ClearRenderTargetView(targets[0], DirectX::Colors::Transparent);
context->ClearDepthStencilView(depthStencilView, D3D11_CLEAR_DEPTH | D3D11_CLEAR_STENCIL, 1.0f, 0);
```

Use the prediction to get the view and projection matrices for the camera

The view and projection matrices for each holographic camera will change with every frame. Refresh the data in the constant buffer for each holographic camera. Do this after you updated the prediction, and before you make any draw calls for that camera.

From AppMain::Render

```
pCameraResources->UpdateViewProjectionBuffer(m_deviceResources, cameraPose, m_referenceFrame->CoordinateSystem);

// Attach the view/projection constant buffer for this camera to the graphics pipeline.
bool cameraActive = pCameraResources->AttachViewProjectionBuffer(m_deviceResources);
```

Here, we show how the matrices are acquired from the camera pose. During this process we also obtain the current viewport for the camera. Note how we provide a coordinate system: this is the same coordinate system we used to understand gaze, and it's the same one we used to position the spinning cube.

From CameraResources::UpdateViewProjectionBuffer

```
// The system changes the viewport on a per-frame basis for system optimizations.
m_d3dViewport = CD3D11_VIEWPORT(
    cameraPose->Viewport.Left,
    cameraPose->Viewport.Top,
    cameraPose->Viewport.Width,
    cameraPose->Viewport.Height
);

// The projection transform for each frame is provided by the HolographicCameraPose.
HolographicStereoTransform cameraProjectionTransform = cameraPose->ProjectionTransform;

// Get a container object with the view and projection matrices for the given
// pose in the given coordinate system.
Platform::IBox<HolographicStereoTransform>^ viewTransformContainer = cameraPose-
    >TryGetViewTransform(coordinateSystem);

// If TryGetViewTransform returns a null pointer, that means the pose and coordinate
// system cannot be understood relative to one another; content cannot be rendered
// in this coordinate system for the duration of the current frame.
// This usually means that positional tracking is not active for the current frame, in
// which case it is possible to use a SpatialLocatorAttachedFrameOfReference to render
// content that is not world-locked instead.
DX::ViewProjectionConstantBufferData viewProjectionConstantBufferData;
bool viewTransformAcquired = viewTransformContainer != nullptr;
if (viewTransformAcquired)
{
    // Otherwise, the set of view transforms can be retrieved.
    HolographicStereoTransform viewCoordinateSystemTransform = viewTransformContainer->Value;

    // Update the view matrices. Holographic cameras (such as Microsoft HoloLens) are
    // constantly moving relative to the world. The view matrices need to be updated
    // every frame.
    XMStoreFloat4x4(
        &viewProjectionConstantBufferData.viewProjection[0],
        XMMatrixTranspose(XMLoadFloat4x4(&viewCoordinateSystemTransform.Left)) *
    XMLoadFloat4x4(&cameraProjectionTransform.Left))
    );
    XMStoreFloat4x4(
        &viewProjectionConstantBufferData.viewProjection[1],
        XMMatrixTranspose(XMLoadFloat4x4(&viewCoordinateSystemTransform.Right)) *
    XMLoadFloat4x4(&cameraProjectionTransform.Right))
    );
}
```

The viewport should be set each frame. Your vertex shader (at least) will generally need access to the view/projection data.

From **CameraResources::AttachViewProjectionBuffer**

```
// Set the viewport for this camera.  
context->RSSetViewports(1, &m_d3dViewport);  
  
// Send the constant buffer to the vertex shader.  
context->VSSetConstantBuffers(  
    1,  
    1,  
    m_viewProjectionConstantBuffer.GetAddressOf()  
);
```

Render to the camera back buffer

It's a good idea to check that **TryGetViewTransform** succeeded before trying to use the view/projection data, because if the coordinate system is not locatable (e.g., tracking was interrupted) your app cannot render with it for that frame. The template only calls **Render** on the spinning cube if the **CameraResources** class indicates a successful update.

From **AppMain::Render**

```
if (cameraActive)  
{  
    // Draw the sample hologram.  
    m_spinningCubeRenderer->Render();  
}
```

Draw holographic content

The Windows Holographic app template renders content in stereo by using the recommended technique of drawing instanced geometry to a Texture2DArray of size 2. Let's look at the instancing part of this, and how it works on Windows Mixed Reality.

From **SpinningCubeRenderer::Render**

```
// Draw the objects.  
context->DrawIndexedInstanced(  
    m_indexCount,    // Index count per instance.  
    2,                // Instance count.  
    0,                // Start index location.  
    0,                // Base vertex location.  
    0                 // Start instance location.  
>;
```

Each instance accesses a different view/projection matrix from the constant buffer. Here's the constant buffer structure, which is just an array of 2 matrices.

From **VPRTVertexShader.hlsl**

```
cbuffer ViewProjectionConstantBuffer : register(b1)  
{  
    float4x4 viewProjection[2];  
};
```

The render target array index must be set for each pixel. In the following snippet, output.rtvid is mapped to the

SV_RenderTargetArrayIndex semantic. Note that this requires support for an optional Direct3D 11.3 feature, which allows the render target array index semantic to be set from any shader stage.

From **VPRTVertexShader.hlsl**

```
// Per-vertex data used as input to the vertex shader.
struct VertexShaderInput
{
    min16float3 pos      : POSITION;
    min16float3 color    : COLOR0;
    uint         instId   : SV_InstanceID;
};

// Per-vertex data passed to the geometry shader.
// Note that the render target array index is set here in the vertex shader.
struct VertexShaderOutput
{
    min16float4 pos      : SV_POSITION;
    min16float3 color    : COLOR0;
    uint         rtvId    : SV_RenderTargetArrayIndex; // SV_InstanceID % 2
};

// ...

int idx = input.instId % 2;
// Set the render target array index.
output.rtvId = idx;
```

If you want to use your existing instanced drawing techniques with this method of drawing to a stereo render target array, all you have to do is draw twice the number of instances you normally have. In the shader, divide **input.instId** by 2 to get the original instance ID, which can be indexed into (for example) a buffer of per-object data: `int actualIdx = input.instId / 2;`

Important note about rendering stereo content on HoloLens

Windows Mixed Reality supports the ability to set the render target array index from any shader stage; normally, this is a task that could only be done in the geometry shader stage due to the way the semantic is defined for Direct3D 11. Here, we show a complete example of how to set up a rendering pipeline with just the vertex and pixel shader stages set. The shader code is as described above.

From **SpinningCubeRenderer::Render**

```

const auto context = m_deviceResources->GetD3DDeviceContext();

// Each vertex is one instance of the VertexPositionColor struct.
const UINT stride = sizeof(VertexPositionColor);
const UINT offset = 0;
context->IASetVertexBuffers(
    0,
    1,
    m_vertexBuffer.GetAddressOf(),
    &stride,
    &offset
);
context->IASetIndexBuffer(
    m_indexBuffer.Get(),
    DXGI_FORMAT_R16_UINT, // Each index is one 16-bit unsigned integer (short).
    0
);
context->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
context->IASetInputLayout(m_inputLayout.Get());

// Attach the vertex shader.
context->VSSetShader(
    m_vertexShader.Get(),
    nullptr,
    0
);
// Apply the model constant buffer to the vertex shader.
context->VSSetConstantBuffers(
    0,
    1,
    m_modelConstantBuffer.GetAddressOf()
);

// Attach the pixel shader.
context->PSSetShader(
    m_pixelShader.Get(),
    nullptr,
    0
);

// Draw the objects.
context->DrawIndexedInstanced(
    m_indexCount, // Index count per instance.
    2,           // Instance count.
    0,           // Start index location.
    0,           // Base vertex location.
    0            // Start instance location.
);

```

Important note about rendering on non-HoloLens devices

Setting the render target array index in the vertex shader requires that the graphics driver supports an optional Direct3D 11.3 feature, which HoloLens does support. Your app may be able to safely implement just that technique for rendering, and all requirements will be met for running on the Microsoft HoloLens.

It may be the case that you want to use the HoloLens emulator as well, which can be a powerful development tool for your holographic app - and support Windows Mixed Reality immersive headset devices that are attached to Windows 10 PCs. Support for the non-HoloLens rendering path - and therefore, for all of Windows Mixed Reality - is also built into the Windows Holographic app template. In the template code, you will find code to enable your holographic app to run on the GPU in your development PC. Here is how the **DeviceResources** class checks for this optional feature support.

From **DeviceResources::CreateDeviceResources**

```

// Check for device support for the optional feature that allows setting the render target array index from
// the vertex shader stage.
D3D11_FEATURE_DATA_D3D11_OPTIONS3 options;
m_d3dDevice->CheckFeatureSupport(D3D11_FEATURE_D3D11_OPTIONS3, &options, sizeof(options));
if (options.VPAndRTArrayIndexFromAnyShaderFeedingRasterizer)
{
    m_supportsVprt = true;
}

```

To support rendering without this optional feature, your app must use a geometry shader to set the render target array index. This snippet would be added *after* **VSSetConstantBuffers**, and *before* **PSSetShader** in the code example shown in the previous section that explains how to render stereo on HoloLens.

From **SpinningCubeRenderer::Render**

```

if (!m_usingVprtShaders)
{
    // On devices that do not support the D3D11_FEATURE_D3D11_OPTIONS3::
    // VPAndRTArrayIndexFromAnyShaderFeedingRasterizer optional feature,
    // a pass-through geometry shader is used to set the render target
    // array index.
    context->GSSetShader(
        m_geometryShader.Get(),
        nullptr,
        0
    );
}

```

HLSL NOTE: In this case, you must also load a slightly modified vertex shader that passes the render target array index to the geometry shader using an always-allowed shader semantic, such as TEXCOORD0. The geometry shader does not have to do any work; the template geometry shader passes through all data, with the exception of the render target array index, which is used to set the SV_RenderTargetArrayIndex semantic.

App template code for **GeometryShader.hlsl**

```

// Per-vertex data from the vertex shader.
struct GeometryShaderInput
{
    min16float4 pos      : SV_POSITION;
    min16float3 color   : COLOR0;
    uint         instId : TEXCOORD0;
};

// Per-vertex data passed to the rasterizer.
struct GeometryShaderOutput
{
    min16float4 pos      : SV_POSITION;
    min16float3 color   : COLOR0;
    uint         rtvId   : SV_RenderTargetArrayIndex;
};

// This geometry shader is a pass-through that leaves the geometry unmodified
// and sets the render target array index.
[maxvertexcount(3)]
void main(triangle GeometryShaderInput input[3], inout TriangleStream<GeometryShaderOutput> outStream)
{
    GeometryShaderOutput output;
    [unroll(3)]
    for (int i = 0; i < 3; ++i)
    {
        output.pos    = input[i].pos;
        output.color = input[i].color;
        output.rtvId = input[i].instId;
        outStream.Append(output);
    }
}

```

Present

Enable the holographic frame to present the swap chain

With Windows Mixed Reality, the system controls the swap chain. The system then manages presenting frames to each holographic camera to ensure a high quality user experience. It also provides a viewport update each frame, for each camera, to optimize aspects of the system such as image stabilization or Mixed Reality Capture. So, a holographic app using DirectX doesn't call **Present** on a DXGI swap chain. Instead, you use the [HolographicFrame](#) class to present all swapchains for a frame once you're done drawing it.

From **DeviceResources::Present**

```

HolographicFramePresentResult presentResult = frame->PresentUsingCurrentPrediction();
HolographicFramePrediction^ prediction = frame->CurrentPrediction;

```

By default, this API waits for the frame to finish before it returns. Holographic apps should wait for the previous frame to finish before starting work on a new frame, because this reduces latency and allows for better results from holographic frame predictions. This isn't a hard rule, and if you have frames that take longer than one screen refresh to render you can disable this wait by passing the [HolographicFramePresentWaitBehavior](#) parameter to [PresentUsingCurrentPrediction](#). In this case, you would likely use an asynchronous rendering thread in order to maintain a continuous load on the GPU. Note that the refresh rate of the HoloLens device is 60hz, where one frame has a duration of approximately 16 ms. Immersive headset devices can range from 60hz to 90hz; when refreshing the display at 90 hz, each frame will have a duration of approximately 11 ms.

In the template, we also discard the render target views for each camera after the frame is presented.

From **DeviceResources::Present**

```

UseHolographicCameraResources<void>([this, prediction](std::map<UINT32, std::unique_ptr<CameraResources>>& cameraResourceMap)
{
    for (auto cameraPose : prediction->CameraPoses)
    {
        DX::CameraResources* pCameraResources = cameraResourceMap[cameraPose->HolographicCamera->Id].get();
        m_d3dContext->DiscardView(pCameraResources->GetBackBufferRenderTargetView());
        m_d3dContext->DiscardView(pCameraResources->GetDepthStencilView());
    }
});

```

Handle DeviceLost scenarios in cooperation with the HolographicFrame

DirectX 11 apps would typically want to check the HRESULT returned by the DXGI swap chain's **Present** function to find out if there was a **DeviceLost** error. The [HolographicFrame](#) class handles this for you. Inspect the [HolographicFramePresentResult](#) it returns to find out if you need to release and recreate the Direct3D device and device-based resources.

```

// The PresentUsingCurrentPrediction API will detect when the graphics device
// changes or becomes invalid. When this happens, it is considered a Direct3D
// device lost scenario.
if (presentResult == HolographicFramePresentResult::DeviceRemoved)
{
    // The Direct3D device, context, and resources should be recreated.
    HandleDeviceLost();
}

```

Note that if the Direct3D device was lost, and you did recreate it, you have to tell the [HolographicSpace](#) to start using the new device. The swap chain will be recreated for this device.

From **DeviceResources::InitializeUsingHolographicSpace**

```
m_holographicSpace->SetDirect3D11Device(m_d3dInteropDevice);
```

Once your frame is presented, you can return back to the main program loop and allow it to continue to the next frame.

Hybrid graphics PCs and mixed reality applications

Windows 10 Creators Update PCs may be configured with **both** discrete and integrated GPUs. With these types of computers, Windows will choose the adapter the headset is connected to. Applications must ensure the DirectX device it creates uses the same adapter.

By default, most sample code demonstrates creating a DirectX device using the default hardware adapter, which on a hybrid system may not be the same as the one used for the headset.

To work around any issues this may cause, use the [HolographicAdapterId](#) from either [HolographicSpace->PrimaryAdapterId](#) property or [HolographicDisplay->AdapterId](#). This adapterId can then be used to enumerate the DXGIAdapters

Code for **GetAdapterFromId** Helper Function

```

IDXGIAAdapter* GetAdapterFromId(Windows::Graphics::Holographic::HolographicAdapterId adapterId)
{
    ComPtr spDXGIFactory;
    HRESULT hr = CreateDXGIFactory1(__uuidof(IDXGIFactory1), (void**)&spDXGIFactory);
    if (FAILED(hr))
        return nullptr;

    ComPtr spDXGIAdapter;
    for (UINT i = 0;
        spDXGIFactory->EnumAdapters(i, &spDXGIAdapter) != DXGI_ERROR_NOT_FOUND;
        ++i)
    {
        DXGI_ADAPTER_DESC desc;
        if (FAILED(spDXGIAdapter->GetDesc(&desc)))
        {
            spDXGIAdapter.Reset();
            continue;
        }

        if (desc.AdapterLuid.HighPart == adapterId.HighPart &&
            desc.AdapterLuid.LowPart == adapterId.LowPart)
        {
            break;
        }
    }

    return spDXGIAdapter.Detach();
}

```

Code to **update DeviceResources::CreateDeviceResources to use IDXGIAdapter**

```

// get adapter from HolographicDisplay
// assumes Windows::Graphics::Holographic::HolographicDisplay^ m_holographicDisplay; was previously
initialized
auto adapterId = m_holographicDisplay->AdapterId;

// use helper function to get the DXGIAdapter that matches
ComPtr spDXGIAdapter = GetAdapterFromId(adapterId);

// if adapter is null, revert to previous behavior
D3D_DRIVER_TYPE driverType = (nullptr != spDXGIAdapter) ? D3D_DRIVER_TYPE_UNKNOWN : D3D_DRIVER_TYPE_HARDWARE;

// Create the Direct3D 11 API device object and a corresponding context.
ComPtr device;
ComPtr context;
HRESULT hr = D3D11CreateDevice(
    spDXGIAdapter.Get(),
    driverType,
    0,      // Should be 0 unless the driver is D3D_DRIVER_TYPE_SOFTWARE.
    creationFlags, // Set debug and Direct2D compatibility flags.
    featureLevels, // List of feature levels this app can support.
    ARRAYSIZE(featureLevels), // Size of the list above.
    D3D11_SDK_VERSION, // Always set this to D3D11_SDK_VERSION for Windows Store apps.
    &device, // Returns the Direct3D device created.
    &m_d3dFeatureLevel, // Returns feature level of device created.
    &context // Returns the device immediate context.
);

```

Hybrid graphics and Media Foundation Using Media Foundation on hybrid systems may cause issues where video will not render or video texture is corrupt. This can occur because Media Foundation is defaulting to a system behavior as mentioned above. In some scenarios, creating a separate ID3D11Device is required to support multi-threading and the correct creation flags are set.

When initializing the ID3D11Device, D3D11_CREATE_DEVICE_VIDEO_SUPPORT flag must be defined as part

of the D3D11_CREATE_DEVICE_FLAG. Once the device and context is created, call [SetMultithreadProtected](#) to enable multithreading. To associate the device with the [IMFDXGIDeviceManager](#), use the [IMFDXGIDeviceManager::ResetDevice](#) function.

Code to **associate a ID3D11Device with IMFDXGIDeviceManager**

```
// create dx device for media pipeline
ComPtr<ID3D11Device> spMediaDevice;
if (FAILED(CreateMediaDevice(spAdapter.Get(), &spMediaDevice))) // see above, also make sure to enable
    return; // D3D11_CREATE_DEVICE_VIDEO_SUPPORT |
D3D11_CREATE_DEVICE_BGRA_SUPPORT;

...
// Turn multithreading on
ComPtr<ID3D10Multithread> spMultithread;
if (SUCCEEDED(spContext.As(&spMultithread)))
{
    spMultithread->SetMultithreadProtected(TRUE);
}

...
// lock the shared dxgi device manager
// call MFUnlockDXGIDeviceManager when no longer needed
UINT uiResetToken;
ComPtr<IMFDXGIDeviceManager> spDeviceManager;
hr = MFLockDXGIDeviceManager(&uiResetToken, &spDeviceManager);
if (FAILED(hr))
    return hr;

// associate the device with the manager
hr = spDeviceManager->ResetDevice(spMediaDevice.Get(), uiResetToken);
if (FAILED(hr))
    return hr;
```

See also

- [Coordinate systems in DirectX](#)
- [Using the HoloLens emulator](#)
- [D3D11_FEATURE_DATA_D3D11_OPTIONS3 structure](#)

Coordinate systems in DirectX

11/6/2018 • 25 minutes to read • [Edit Online](#)

[Coordinate systems](#) form the basis for spatial understanding offered by Windows Mixed Reality APIs.

Today's seated VR or single-room VR devices establish one primary coordinate system to represent their tracked space. Windows Mixed Reality devices such as HoloLens are designed to be used throughout large undefined environments, with the device discovering and learning about its surroundings as the user walks around. This allows the device to adapt to continually-improving knowledge about the user's rooms, but results in coordinate systems that will change their relationship to one another through the lifetime of the app. Windows Mixed Reality supports a wide spectrum of devices, ranging from seated immersive headsets through world-attached reference frames.

Spatial coordinate systems in Windows

The core type used to reason about real-world coordinate systems in Windows is the [SpatialCoordinateSystem](#). An instance of this type represents an arbitrary coordinate system and provides a method to get a transformation matrix that you can use to transform between two coordinate systems without understanding the details of each.

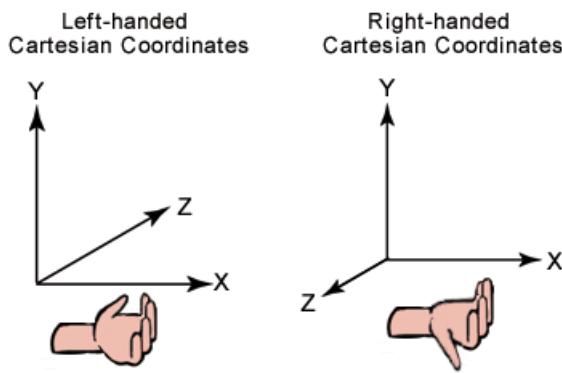
Methods that return spatial information, represented as points, rays, or volumes in the user's surroundings, will accept a [SpatialCoordinateSystem](#) parameter to let you decide the coordinate system in which it's most useful for those coordinates to be returned. The units for these coordinates will always be in meters.

A [SpatialCoordinateSystem](#) has a dynamic relationship with other coordinate systems, including those that represent the device's position. At any point in time, the device may be able to locate some coordinate systems and not others. For most coordinate systems, your app must be ready to handle periods of time during which they cannot be located.

Your application should not create [SpatialCoordinateSystems](#) directly - rather they should be consumed via the Perception APIs. There are three primary sources of coordinate systems in the Perception APIs, each of which map to a concept described on the [Coordinate systems](#) page:

- To get a stationary frame of reference, create a [SpatialStationaryFrameOfReference](#) or obtain one from the current [SpatialStage](#).
- To get a spatial anchor, create a [SpatialAnchor](#).
- To get an attached frame of reference, create a [SpatialLocatorAttachedFrameOfReference](#).

All of the coordinate systems returned by these objects are right-handed, with +y up, +x to the right and +z backwards. You can remember which direction the positive z-axis points by pointing the fingers of either your left or right hand in the positive x direction and curling them into the positive y direction. The direction your thumb points, either toward or away from you, is the direction that the positive z-axis points for that coordinate system. The following illustration shows these two coordinate systems.



Left-hand and right-hand coordinate systems

To bootstrap into a SpatialCoordinateSystem based on the position of a HoloLens, use the [SpatialLocator](#) class to create either an attached or stationary frame of reference, as described in the sections below.

Place holograms in the world using a spatial stage

The coordinate system for opaque Windows Mixed Reality immersive headsets is accessed using the [SpatialStageFrameOfReference::Current](#) property. This API provides a coordinate system, information about whether the player is seated or mobile, the boundary of a safe area for walking around if the player is mobile, and an indication of whether or not the headset is directional. There is also an event handler for updates to the spatial stage.

First, we get the spatial stage and subscribe for updates to it:

Code for **Spatial stage initialization**

```
SpatialStageManager::SpatialStageManager(
    const std::shared_ptr<DX::DeviceResources>& deviceResources,
    const std::shared_ptr<SceneController>& sceneController)
: m_deviceResources(deviceResources), m_sceneController(sceneController)
{
    // Get notified when the stage is updated.
    m_spatialStageChangedEventToken = SpatialStageFrameOfReference::CurrentChanged +=
        ref new EventHandler<Object^>(std::bind(&SpatialStageManager::OnCurrentChanged, this, _1));

    // Make sure to get the current spatial stage.
    OnCurrentChanged(nullptr);
}
```

In the `OnCurrentChanged` method, your app should inspect the spatial stage and update the player experience accordingly. In this example, we provide a visualization of the stage boundary, as well as the start position specified by the user and the stage's range of view and range of movement properties. We also fall back to our own stationary coordinate system, when a stage cannot be provided.

Code for **Spatial stage update**

```
void SpatialStageManager::OnCurrentChanged(Object^ /*o*/)
{
    // The event notifies us that a new stage is available.
    // Get the current stage.
    m_currentStage = SpatialStageFrameOfReference::Current;

    // Clear previous content.
    m_sceneController->ClearSceneObjects();

    if (m_currentStage != nullptr)
    {
        // Obtain stage geometry.
        //
```

```

auto stageCoordinateSystem = m_currentStage->CoordinateSystem;
auto boundsVertexArray = m_currentStage->TryGetMovementBounds(stageCoordinateSystem);

// Visualize the area where the user can move around.
std::vector<float3> boundsVertices;
boundsVertices.resize(boundsVertexArray->Length);
memcpy(boundsVertices.data(), boundsVertexArray->Data, boundsVertexArray->Length * sizeof(float3));
std::vector<unsigned short> indices = TriangulatePoints(boundsVertices);
m_stageBoundsShape =
    std::make_shared<SceneObject>(
        m_deviceResources,
        reinterpret_cast<std::vector<XMFLOAT3>&>(boundsVertices),
        indices,
        XMFLOAT3(DirectX::Colors::SeaGreen),
        stageCoordinateSystem);
m_sceneController->AddSceneObject(m_stageBoundsShape);

// In this sample, we draw a visual indicator for some spatial stage properties.
// If the view is forward-only, the indicator is a half circle pointing forward - otherwise, it
// is a full circle.
// If the user can walk around, the indicator is blue. If the user is seated, it is red.

// The indicator is rendered at the origin - which is where the user declared the center of the
// stage to be during setup - above the plane of the stage bounds object.
float3 visibleAreaCenter = float3(0.f, 0.001f, 0.f);

// Its shape depends on the look direction range.
std::vector<float3> visibleAreaIndicatorVertices;
if (m_currentStage->LookDirectionRange == SpatialLookDirectionRange::ForwardOnly)
{
    // Half circle for forward-only look direction range.
    visibleAreaIndicatorVertices = CreateCircle(visibleAreaCenter, 0.25f, 9, XM_PI);
}
else
{
    // Full circle for omnidirectional look direction range.
    visibleAreaIndicatorVertices = CreateCircle(visibleAreaCenter, 0.25f, 16, XM_2PI);
}

// Its color depends on the movement range.
XMFLOAT3 visibleAreaColor;
if (m_currentStage->MovementRange == SpatialMovementRange::NoMovement)
{
    visibleAreaColor = XMFLOAT3(DirectX::Colors::OrangeRed);
}
else
{
    visibleAreaColor = XMFLOAT3(DirectX::Colors::Aqua);
}

std::vector<unsigned short> visibleAreaIndicatorIndices =
TriangulatePoints(visibleAreaIndicatorVertices);

// Visualize the look direction range.
m_stageVisibleAreaIndicatorShape =
    std::make_shared<SceneObject>(
        m_deviceResources,
        reinterpret_cast<std::vector<XMFLOAT3>&>(visibleAreaIndicatorVertices),
        visibleAreaIndicatorIndices,
        visibleAreaColor,
        stageCoordinateSystem);
m_sceneController->AddSceneObject(m_stageVisibleAreaIndicatorShape);
}

else
{
    // No spatial stage was found.
    // Fall back to a stationary coordinate system.
    auto locator = SpatialLocator::GetDefault();
    if (locator)
}

```

```

    {
        m_stationaryFrameOfReference = locator->CreateStationaryFrameOfReferenceAtCurrentLocation();

        // Render an indicator, so that we know we fell back to a mode without a stage.
        std::vector<float3> visibleAreaIndicatorVertices;
        float3 visibleAreaCenter = float3(0.f, -2.0f, 0.f);
        visibleAreaIndicatorVertices = CreateCircle(visibleAreaCenter, 0.125f, 16, XM_2PI);
        std::vector<unsigned short> visibleAreaIndicatorIndices =
            TriangulatePoints(visibleAreaIndicatorVertices);
        m_stageVisibleAreaIndicatorShape =
            std::make_shared<SceneObject>(
                m_deviceResources,
                reinterpret_cast<std::vector<XMFLOAT3>&>(visibleAreaIndicatorVertices),
                visibleAreaIndicatorIndices,
                XMFLOAT3(DirectX::Colors::LightSlateGray),
                m_stationaryFrameOfReference->CoordinateSystem);
        m_sceneController->AddSceneObject(m_stageVisibleAreaIndicatorShape);
    }
}
}
}

```

The set of vertices that define the stage boundary are provided in clockwise order. The Windows Mixed Reality shell draws a fence at the boundary when the user approaches it; you may wish to triangularize the walkable area for your own purposes. The following algorithm can be used to triangularize the stage.

Code for **Spatial stage triangularization**

```

std::vector<unsigned short> SpatialStageManager::TriangulatePoints(std::vector<float3> const& vertices)
{
    size_t const& vertexCount = vertices.size();

    // Segments of the shape are removed as they are triangularized.
    std::vector<bool> vertexRemoved;
    vertexRemoved.resize(vertexCount, false);
    unsigned int vertexRemovedCount = 0;

    // Indices are used to define triangles.
    std::vector<unsigned short> indices;

    // Decompose into convex segments.
    unsigned short currentVertex = 0;
    while (vertexRemovedCount < (vertexCount - 2))
    {
        // Get next triangle:
        // Start with the current vertex.
        unsigned short index1 = currentVertex;

        // Get the next available vertex.
        unsigned short index2 = index1 + 1;

        // This cycles to the next available index.
        auto CycleIndex = [=](unsigned short indexToCycle, unsigned short stopIndex)
        {
            // Make sure the index does not exceed bounds.
            if (indexToCycle >= unsigned short(vertexCount))
            {
                indexToCycle -= unsigned short(vertexCount);
            }

            while (vertexRemoved[indexToCycle])
            {
                // If the vertex is removed, go to the next available one.
                ++indexToCycle;

                // Make sure the index does not exceed bounds.
                if (indexToCycle >= unsigned short(vertexCount))
                {

```

```

        indexToCycle -= unsigned short(vertexCount);
    }

    // Prevent cycling all the way around.
    // Should not be needed, as we limit with the vertex count.
    if (indexToCycle == stopIndex)
    {
        break;
    }
}

return indexToCycle;
};

index2 = CycleIndex(index2, index1);

// Get the next available vertex after that.
unsigned short index3 = index2 + 1;
index3 = CycleIndex(index3, index1);

// Vertices that may define a triangle inside the 2D shape.
auto& v1 = vertices[index1];
auto& v2 = vertices[index2];
auto& v3 = vertices[index3];

// If the projection of the first segment (in clockwise order) onto the second segment is
// positive, we know that the clockwise angle is less than 180 degrees, which tells us
// that the triangle formed by the two segments is contained within the bounding shape.
auto v2ToV1 = v1 - v2;
auto v2ToV3 = v3 - v2;
float3 normalToV2ToV3 = { -v2ToV3.z, 0.f, v2ToV3.x };
float projectionOntoNormal = dot(v2ToV1, normalToV2ToV3);
if (projectionOntoNormal >= 0)
{
    // Triangle is contained within the 2D shape.

    // Remove peak vertex from the list.
    vertexRemoved[index2] = true;
    ++vertexRemovedCount;

    // Create the triangle.
    indices.push_back(index1);
    indices.push_back(index2);
    indices.push_back(index3);

    // Continue on to the next outer triangle.
    currentVertex = index3;
}
else
{
    // Triangle is a cavity in the 2D shape.
    // The next triangle starts at the inside corner.
    currentVertex = index2;
}
}

indices.shrink_to_fit();
return indices;
}

```

Place holograms in the world using a stationary frame of reference

The [SpatialStationaryFrameOfReference](#) class represents a frame of reference that [remains stationary](#) relative to the user's surroundings as the user moves around. This frame of reference prioritizes keeping coordinates stable near the device. One key use of a [SpatialStationaryFrameOfReference](#) is to act as the underlying world coordinate system within a rendering engine when rendering holograms.

To get a SpatialStationaryFrameOfReference, use the [SpatialLocator](#) class and call [CreateStationaryFrameOfReferenceAtCurrentLocation](#).

From the Windows Holographic app template code:

```
// The simplest way to render world-locked holograms is to create a stationary reference frame
// when the app is launched. This is roughly analogous to creating a "world" coordinate system
// with the origin placed at the device's position as the app is launched.
referenceFrame = locator.CreateStationaryFrameOfReferenceAtCurrentLocation();
```

- Stationary reference frames are designed to provide a best-fit position relative to the overall space. Individual positions within that reference frame are allowed to drift slightly. This is normal as the device learns more about the environment.
- When precise placement of individual holograms is required, a SpatialAnchor should be used to anchor the individual hologram to a position in the real world - for example, a point the user indicates to be of special interest. Anchor positions do not drift, but can be corrected; the anchor will use the corrected position starting in the next frame after the correction has occurred.

Place holograms in the world using spatial anchors

[Spatial anchors](#) are a great way to place holograms at a specific place in the real world, with the system ensuring the anchor stays in place over time. This topic explains how to create and use an anchor, and how to work with anchor data.

You can create a SpatialAnchor at any position and orientation within the SpatialCoordinateSystem of your choosing. The device must be able to locate that coordinate system at the moment, and the system must not have reached its limit of spatial anchors.

Once defined, the coordinate system of a SpatialAnchor adjusts continually to retain the precise position and orientation of its initial location. You can then use this SpatialAnchor to render holograms that will appear fixed in the user's surroundings at that exact location.

The effects of the adjustments that keep the anchor in place are magnified as distance from the anchor increases. Therefore, you should avoid rendering content relative to an anchor that is more than about 3 meters from that anchor's origin.

You can persist a SpatialAnchor using the [SpatialAnchorStore](#) class and then get it back in a future app session.

The [CoordinateSystem](#) property gets a coordinate system that lets you place content relative to the anchor, with easing applied when the device adjusts the anchor's precise location.

Use the [RawCoordinateSystem](#) property and the corresponding [RawCoordinateSystemAdjusted](#) event to manage these adjustments yourself.

Create SpatialAnchors for holographic content

For this code sample, we modified the Windows Holographic app template to create anchors when the **Pressed** gesture is detected. The cube is then placed at the anchor during the render pass.

Since multiple anchors are supported by the helper class, we can place as many cubes as we want using this code sample!

Note that the IDs for anchors are something you control in your app. In this example, we have created a naming scheme that is sequential based on the number of anchors currently stored in the app's collection of anchors.

```

// Check for new input state since the last frame.
SpatialInteractionSourceState^ pointerState = m_spatialInputHandler->CheckForInput();
if (pointerState != nullptr)
{
    // Try to get the pointer pose relative to the SpatialStationaryReferenceFrame.
    SpatialPointerPose^ pointerPose = pointerState->TryGetPointerPose(currentCoordinateSystem);
    if (pointerPose != nullptr)
    {
        // When a Pressed gesture is detected, the anchor will be created two meters in front of the user.

        // Get the gaze direction relative to the given coordinate system.
        const float3 headPosition = pointerPose->Head->Position;
        const float3 headDirection = pointerPose->Head->ForwardDirection;

        // The anchor position in the StationaryReferenceFrame.
        static const float distanceFromUser = 2.0f; // meters
        const float3 gazeAtTwoMeters = headPosition + (distanceFromUser * headDirection);

        // Create the anchor at position.
        SpatialAnchor^ anchor = SpatialAnchor::TryCreateRelativeTo(currentCoordinateSystem,
gazeAtTwoMeters);

        if ((anchor != nullptr) && (m_spatialAnchorHelper != nullptr))
        {
            // In this example, we store the anchor in an IMap.
            auto anchorMap = m_spatialAnchorHelper->GetAnchorMap();

            // Create an identifier for the anchor.
            String^ id = ref new String(L"HolographicSpatialAnchorStoreSample_Anchor") + anchorMap->Size;

            anchorMap->Insert(id->ToString(), anchor);
        }
    }
}

```

Asynchronously load, and cache, the SpatialAnchorStore

Let's see how to write a SampleSpatialAnchorHelper class that helps handle this persistence, including:

- Storing a collection of in-memory anchors, indexed by a Platform::String key.
- Loading anchors from the system's SpatialAnchorStore, which is kept separate from the local in-memory collection.
- Saving the local in-memory collection of anchors to the SpatialAnchorStore when the app chooses to do so.

Here's how to save [SpatialAnchor](#) objects in the [SpatialAnchorStore](#).

When the class starts up, we request the SpatialAnchorStore asynchronously. This involves system I/O as the API loads the anchor store, and this API is made asynchronous so that the I/O is non-blocking.

```

// Request the spatial anchor store, which is the WinRT object that will accept the imported anchor data.
return create_task(SpatialAnchorManager::RequestStoreAsync())
    .then([](task<SpatialAnchorStore^> previousTask)
{
    std::shared_ptr<SampleSpatialAnchorHelper> newHelper = nullptr;

    try
    {
        SpatialAnchorStore^ anchorStore = previousTask.get();

        // Once the SpatialAnchorStore has been loaded by the system, we can create our helper class.

        // Using "new" to access private constructor
        newHelper = std::shared_ptr<SampleSpatialAnchorHelper>(new
SampleSpatialAnchorHelper(anchorStore));

        // Now we can load anchors from the store.
        newHelper->LoadFromAnchorStore();
    }
    catch (Exception^ exception)
    {
        PrintWstringToDebugConsole(
            std::wstring(L"Exception while loading the anchor store: ") +
            exception->Message->Data() +
            L"\n");
    }
}

// Return the initialized class instance.
return newHelper;
});

```

You will be given a SpatialAnchorStore that you can use to save the anchors. This is an IMapView that associates key values that are Strings, with data values that are SpatialAnchors. In our sample code, we store this in a private class member variable that is accessible through a public function of our helper class.

```

SampleSpatialAnchorHelper::SampleSpatialAnchorHelper(SpatialAnchorStore^ anchorStore)
{
    m_anchorStore = anchorStore;
    m_anchorMap = ref new Platform::Collections::Map<String^, SpatialAnchor^>();
}

```

NOTE

Don't forget to hook up the suspend/resume events to save and load the anchor store.

```

void HolographicSpatialAnchorStoreSampleMain::SaveAppState()
{
    // For example, store information in the SpatialAnchorStore.
    if (m_spatialAnchorHelper != nullptr)
    {
        m_spatialAnchorHelper->TrySaveToAnchorStore();
    }
}

```

```

void HolographicSpatialAnchorStoreSampleMain::LoadAppState()
{
    // For example, load information from the SpatialAnchorStore.
    LoadAnchorStore();
}

```

Save content to the anchor store

When the system suspends your app, you need to save your spatial anchors to the anchor store. You may also choose to save anchors to the anchor store at other times, as you find to be necessary for your app's implementation.

When you're ready to try saving the in-memory anchors to the SpatialAnchorStore, you can loop through your collection and try to save each one.

```

// TrySaveToAnchorStore: Stores all anchors from memory into the app's anchor store.
//
// For each anchor in memory, this function tries to store it in the app's AnchorStore. The operation will
fail if
// the anchor store already has an anchor by that name.
//
bool SampleSpatialAnchorHelper::TrySaveToAnchorStore()
{
    // This function returns true if all the anchors in the in-memory collection are saved to the anchor
    // store. If zero anchors are in the in-memory collection, we will still return true because the
    // condition has been met.
    bool success = true;

    // If access is denied, 'anchorStore' will not be obtained.
    if (m_anchorStore != nullptr)
    {
        for each (auto& pair in m_anchorMap)
        {
            auto const& id = pair->Key;
            auto const& anchor = pair->Value;

            // Try to save the anchors.
            if (!m_anchorStore->TrySave(id, anchor))
            {
                // This may indicate the anchor ID is taken, or the anchor limit is reached for the app.
                success=false;
            }
        }
    }

    return success;
}

```

Load content from the anchor store when the app resumes

When your app resumes, or at any other time necessary for your app's implementaiton, you can restore anchors that were previously saved to the AnchorStore by transferring them from the anchor store's IMapView to your own in-memory database of SpatialAnchors.

To restore anchors from the SpatialAnchorStore, restore each one that you are interested in to your own in-memory collection.

You need your own in-memory database of SpatialAnchors; some way to associate Strings with the SpatialAnchors that you create. In our sample code, we choose to use a Windows::Foundation::Collections::IMap to store the anchors, which makes it easy to use the same key and data value for the SpatialAnchorStore.

```
// This is an in-memory anchor list that is separate from the anchor store.  
// These anchors may be used, reasoned about, and so on before committing the collection to the store.  
Windows::Foundation::Collections::IMap<Platform::String^, Windows::Perception::Spatial::SpatialAnchor^>^  
m_anchorMap;
```

NOTE

An anchor that is restored might not be locatable right away. For example, it might be an anchor in a separate room or in a different building altogether. Anchors retrieved from the AnchorStore should be tested for locatability before using them.

NOTE

In this example code, we retrieve all anchors from the AnchorStore. This is not a requirement; your app could just as well pick and choose a certain subset of anchors by using String key values that are meaningful to your implementation.

```
// LoadFromAnchorStore: Loads all anchors from the app's anchor store into memory.  
//  
// The anchors are stored in memory using an IMap, which stores anchors using a string identifier. Any  
string can be used as  
// the identifier; it can have meaning to the app, such as "Game_Level1_CouchAnchor," or it can be a GUID  
that is generated  
// by the app.  
//  
void SampleSpatialAnchorHelper::LoadFromAnchorStore()  
{  
    // If access is denied, 'anchorStore' will not be obtained.  
    if (m_anchorStore != nullptr)  
    {  
        // Get all saved anchors.  
        auto anchorMapView = m_anchorStore->GetAllSavedAnchors();  
        for each (auto const& pair in anchorMapView)  
        {  
            auto const& id = pair->Key;  
            auto const& anchor = pair->Value;  
            m_anchorMap->Insert(id, anchor);  
        }  
    }  
}
```

Clear the anchor store, when needed

Sometimes, you need to clear app state and write new data. Here's how you do that with the [SpatialAnchorStore](#).

Using our helper class, it's almost unnecessary to wrap the Clear function. We choose to do so in our sample implementation, because our helper class is given the responsibility of owning the SpatialAnchorStore instance.

```

// ClearAnchorStore: Clears the AnchorStore for the app.
//
// This function clears the AnchorStore. It has no effect on the anchors stored in memory.
//
void SampleSpatialAnchorHelper::ClearAnchorStore()
{
    // If access is denied, 'anchorStore' will not be obtained.
    if (m_anchorStore != nullptr)
    {
        // Clear all anchors from the store.
        m_anchorStore->Clear();
    }
}

```

Example: Relating anchor coordinate systems to stationary reference frame coordinate systems

Let's say that you have an anchor, and you want to relate something in your anchor's coordinate system to the SpatialStationaryReferenceFrame that you're already using for most of your other content. You can use [TryGetTransformTo](#) to obtain a transform from the anchor's coordinate system to that of the stationary reference frame:

```

// In this code snippet, someAnchor is a SpatialAnchor^ that has been initialized and is valid in the current
environment.
float4x4 anchorSpaceToCurrentCoordinateSystem;
SpatialCoordinateSystem^ anchorSpace = someAnchor->CoordinateSystem;
const auto tryTransform = anchorSpace->TryGetTransformTo(currentCoordinateSystem);
if (tryTransform != nullptr)
{
    anchorSpaceToCurrentCoordinateSystem = tryTransform->Value;
}

```

This process is useful to you in two ways:

1. It tells you if the two reference frames can be understood relative to one another, and;
2. If so, it provides you a transform to go directly from one coordinate system to the other.

With this information, you have an understanding of the spatial relation between objects between the two reference frames.

For rendering, you can often obtain better results by grouping objects according to their original reference frame or anchor. Perform a separate drawing pass for each group. The view matrices are more accurate for objects with model transforms that are created initially using the same coordinate system.

Create holograms using a device-attached frame of reference

There are times when you want to render a hologram that [remains attached](#) to the device's location, for example a panel with debugging information or an informational message when the device is only able to determine its orientation and not its position in space. To accomplish this, we use an attached frame of reference.

The [SpatialLocatorAttachedFrameOfReference](#) class defines coordinate systems which are relative to the device rather than to the real-world. This frame has a fixed heading relative to the user's surroundings that points in the direction the user was facing when the reference frame was created. From then on, all orientations in this frame of reference are relative to that fixed heading, even as the user rotates the device.

For HoloLens, the origin of this frame's coordinate system is located at the center of rotation of the user's head, so that its position is not affected by head rotation. Your app can specify an offset relative to this point to position holograms in front of the user.

To get a [SpatialLocatorAttachedFrameOfReference](#), use the [SpatialLocator](#) class and call

CreateAttachedFrameOfReferenceAtCurrentHeading.

Note that this applies to the entire range of Windows Mixed Reality devices.

Use a reference frame attached to the device

These sections talk about what we changed in the Windows Holographic app template to enable a device-attached frame of reference using this API. Note that this "attached" hologram will work alongside stationary or anchored holograms, and may also be used when the device is temporarily unable to find its position in the world.

First, we changed the template to store a SpatialLocatorAttachedFrameOfReference instead of a SpatialStationaryFrameOfReference:

From **HolographicTagAlongSampleMain.h**:

```
// A reference frame attached to the holographic camera.  
Windows::Perception::Spatial::SpatialLocatorAttachedFrameOfReference^ m_referenceFrame;
```

From **HolographicTagAlongSampleMain.cpp**:

```
// In this example, we create a reference frame attached to the device.  
m_referenceFrame = m_locator->CreateAttachedFrameOfReferenceAtCurrentHeading();
```

During the update, we now obtain the coordinate system at the time stamp obtained from with the frame prediction.

```
// Next, we get a coordinate system from the attached frame of reference that is  
// associated with the current frame. Later, this coordinate system is used for  
// for creating the stereo view matrices when rendering the sample content.  
SpatialCoordinateSystem^ currentCoordinateSystem =  
    m_referenceFrame->GetStationaryCoordinateSystemAtTimestamp(prediction->Timestamp);
```

Get a spatial pointer pose, and follow the user's Gaze

We want our example hologram to follow the user's [gaze](#), similar to how the holographic shell can follow the user's gaze. For this, we need to get the SpatialPointerPose from the same time stamp.

```
SpatialPointerPose^ pose = SpatialPointerPose::TryGetAtTimestamp(currentCoordinateSystem, prediction->Timestamp);
```

This SpatialPointerPose has the information needed to position the hologram according to the [user's current heading](#).

For reasons of user comfort, we use linear interpolation ("lerp") to smooth the change in position such that it occurs over a period of time. This is more comfortable for the user than locking the hologram to their gaze. Lerp the tag-along hologram's position also allows us to stabilize the hologram by dampening the movement; if we did not do this dampening, the user would see the hologram jitter because of what are normally considered to be imperceptible movements of the user's head.

From **StationaryQuadRenderer::PositionHologram**:

```

const float& dtime = static_cast<float>(timer.GetElapsedSeconds());

if (pointerPose != nullptr)
{
    // Get the gaze direction relative to the given coordinate system.
    const float3 headPosition = pointerPose->Head->Position;
    const float3 headDirection = pointerPose->Head->ForwardDirection;

    // The tag-along hologram follows a point 2.0m in front of the user's gaze direction.
    static const float distanceFromUser = 2.0f; // meters
    const float3 gazeAtTwoMeters = headPosition + (distanceFromUser * headDirection);

    // Lerp the position, to keep the hologram comfortably stable.
    auto lerpedPosition = lerp(m_position, gazeAtTwoMeters, dtime * c_lerpRate);

    // This will be used as the translation component of the hologram's
    // model transform.
    SetPosition(lerpedPosition);
}

```

NOTE

In the case of a debugging panel, you might choose to reposition the hologram off to the side a little so that it does not obstruct your view. Here's an example of how you might do that.

For **StationaryQuadRenderer::PositionHologram**:

```

// If you're making a debug view, you might not want the tag-along to be directly in the
// center of your field of view. Use this code to position the hologram to the right of
// the user's gaze direction.
/*
const float3 offset = float3(0.13f, 0.0f, 0.f);
static const float distanceFromUser = 2.2f; // meters
const float3 gazeAtTwoMeters = headPosition + (distanceFromUser * (headDirection + offset));
*/

```

Rotate the hologram to face the camera

It is not enough to simply position the hologram, which in this case is a quad; we must also rotate the object to face the user. Note that this rotation occurs in world space, because this type of billboard allows the hologram to remain a part of the user's environment. View-space billboarding is not as comfortable because the hologram becomes locked to the display orientation; in that case, you would also have to interpolate between the left and right view matrices in order to acquire a view-space billboard transform that does not disrupt stereo rendering. Here, we rotate on the X and Z axes to face the user.

From **StationaryQuadRenderer::Update**:

```

// Seconds elapsed since previous frame.
const float& dTime = static_cast<float>(timer.GetElapsedSeconds());

// Create a direction normal from the hologram's position to the origin of person space.
// This is the z-axis rotation.
XMVECTOR facingNormal = XMVector3Normalize(-XMLoadFloat3(&m_position));

// Rotate the x-axis around the y-axis.
// This is a 90-degree angle from the normal, in the xz-plane.
// This is the x-axis rotation.
XMVECTOR xAxisRotation = XMVector3Normalize(XMVectorSet(XMVectorGetZ(facingNormal), 0.f, -
XMVectorGetX(facingNormal), 0.f));

// Create a third normal to satisfy the conditions of a rotation matrix.
// The cross product of the other two normals is at a 90-degree angle to
// both normals. (Normalize the cross product to avoid floating-point math
// errors.)
// Note how the cross product will never be a zero-matrix because the two normals
// are always at a 90-degree angle from one another.
XMVECTOR yAxisRotation = XMVector3Normalize(XMVector3Cross(facingNormal, xAxisRotation));

// Construct the 4x4 rotation matrix.

// Rotate the quad to face the user.
XMMATRIX rotationMatrix = XMMATRIX(
    xAxisRotation,
    yAxisRotation,
    facingNormal,
    XMVectorSet(0.f, 0.f, 0.f, 1.f)
);

// Position the quad.
const XMMATRIX modelTranslation = XMMatrixTranslationFromVector(XMLoadFloat3(&m_position));

// The view and projection matrices are provided by the system; they are associated
// with holographic cameras, and updated on a per-camera basis.
// Here, we provide the model transform for the sample hologram. The model transform
// matrix is transposed to prepare it for the shader.
XMStoreFloat4x4(&m_modelConstantBufferData.model, XMMatrixTranspose(rotationMatrix * modelTranslation));

```

Set the focus point for image stabilization

WARNING

For Windows Mixed Reality immersive headsets, setting a stabilization plane is usually counter-productive, as it offers less visual quality than providing your app's depth buffer to the system to enable per-pixel depth-based reprojection. Unless running on a HoloLens, you should generally avoid setting the stabilization plane.

On HoloLens, we should also set a manual focus point for [image stabilization](#). For best results with tag-along holograms, we need to use the velocity of the hologram. This is computed as follows.

Note that on immersive desktop headsets, you should instead use the [CommitDirect3D11DepthBuffer](#) API to enable per-pixel depth-based reprojection, as discussed in [Rendering in DirectX](#). That will result in the best visual quality.

From **StationaryQuadRenderer::Update**:

```

// Determine velocity.
// Even though the motion is spherical, the velocity is still linear
// for image stabilization.
auto& deltaX = m_position - m_lastPosition; // meters
m_velocity = deltaX / dTime; // meters per second

```

From **HolographicTagAlongSampleMain::Update:**

```

// SetFocusPoint informs the system about a specific point in your scene to
// prioritize for image stabilization. The focus point is set independently
// for each holographic camera.
// In this example, we set position, normal, and velocity for a tag-along quad.
float3& focusPointPosition = m_stationaryQuadRenderer->GetPosition();
float3 focusPointNormal = -normalize(focusPointPosition);
float3& focusPointVelocity = m_stationaryQuadRenderer->GetVelocity();
renderingParameters->SetFocusPoint(
    currentCoordinateSystem,
    focusPointPosition,
    focusPointNormal,
    focusPointVelocity
);

```

Render the attached hologram

For this example, we also choose to render the hologram in the coordinate system of the SpatialLocatorAttachedReferenceFrame, which is where we positioned the hologram. (If we had decided to render using another coordinate system, we would need to acquire a transform from the device-attached reference frame's coordinate system to that coordinate system.)

From **HolographicTagAlongSampleMain::Render:**

```

// The view and projection matrices for each holographic camera will change
// every frame. This function refreshes the data in the constant buffer for
// the holographic camera indicated by cameraPose.
pCameraResources->UpdateViewProjectionBuffer(
    m_deviceResources,
    cameraPose,
    m_referenceFrame->GetStationaryCoordinateSystemAtTimestamp(prediction->Timestamp)
);

```

That's it! The hologram will now "chase" a position that is 2 meters in front of the user's gaze direction.

NOTE

This example also loads additional content - see StationaryQuadRenderer.cpp.

Handling tracking loss

When the device cannot locate itself in the world, the app experiences "tracking loss". Windows Mixed Reality apps should be able to handle such disruptions to the positional tracking system. These disruptions can be observed, and responses created, by using the LocatabilityChanged event on the default SpatialLocator.

From **AppMain::SetHolographicSpace:**

```
// Be able to respond to changes in the positional tracking state.
m_locatabilityChangedToken =
    m_locator->LocatabilityChanged +=
        ref new Windows::Foundation::TypedEventHandler<SpatialLocator^, Object^>(
            std::bind(&HolographicApp1Main::OnLocatabilityChanged, this, _1, _2)
        );

```

When your app receives a LocatabilityChanged event, it can change behavior as needed. For example, in the PositionalTrackingInhibited state, your app can pause normal operation and render a [tag-along hologram](#) that displays a warning message.

The Windows Holographic app template comes with a LocatabilityChanged handler already created for you. By default, it displays a warning in the debug console when positional tracking is unavailable. You can add code to this handler to provide a response as needed from your app.

From **AppMain.cpp**:

```
void HolographicApp1Main::OnLocatabilityChanged(SpatialLocator^ sender, Object^ args)
{
    switch (sender->Locatability)
    {
        case SpatialLocatability::Unavailable:
            // Holograms cannot be rendered.
            {
                String^ message = L"Warning! Positional tracking is " +
                    sender->Locatability.ToString() + L".\n";
                OutputDebugStringW(message->Data());
            }
            break;

        // In the following three cases, it is still possible to place holograms using a
        // SpatialLocatorAttachedFrameOfReference.
        case SpatialLocatability::PositionalTrackingActivating:
            // The system is preparing to use positional tracking.

        case SpatialLocatability::OrientationOnly:
            // Positional tracking has not been activated.

        case SpatialLocatability::PositionalTrackingInhibited:
            // Positional tracking is temporarily inhibited. User action may be required
            // in order to restore positional tracking.
            break;

        case SpatialLocatability::PositionalTrackingActive:
            // Positional tracking is active. World-locked content can be rendered.
            break;
    }
}
```

Spatial mapping

The [spatial mapping](#) APIs make use of coordinate systems to get model transforms for surface meshes.

See also

- [Coordinate systems](#)
- [Gaze, gestures, and motion controllers in DirectX](#)
- [Spatial mapping in DirectX](#)

Gaze, gestures, and motion controllers in DirectX

11/6/2018 • 9 minutes to read • [Edit Online](#)

If you're going to build directly on top of the platform, you will have to handle input coming from the user - such as where the user is looking via [gaze](#) and what the user has selected with [gestures](#) or [motion controllers](#). Combining these forms of input, you can enable a user to place a [hologram](#) in your app. The [holographic app template](#) has an easy to use example.

Gaze input

To access the user's [gaze](#), you use the [SpatialPointerPose](#) type. The holographic app template includes basic code for understanding gaze. This code provides a vector pointing forward from between the user's eyes, taking into account the device's position and orientation in a given [coordinate system](#).

```
void SpinningCubeRenderer::PositionHologram(SpatialPointerPose^ pointerPose)
{
    if (pointerPose != nullptr)
    {
        // Get the gaze direction relative to the given coordinate system.
        const float3 headPosition      = pointerPose->Head->Position;
        const float3 headDirection    = pointerPose->Head->ForwardDirection;

        // The hologram is positioned two meters along the user's gaze direction.
        static const float distanceFromUser = 2.0f; // meters
        const float3 gazeAtTwoMeters      = headPosition + (distanceFromUser * headDirection);

        // This will be used as the translation component of the hologram's
        // model transform.
        SetPosition(gazeAtTwoMeters);
    }
}
```

You may find yourself asking: "But where does the coordinate system come from?"

Let's answer that question. In our AppMain's **Update** function, we processed a spatial input event by acquiring it relative to the coordinate system for our StationaryFrameOfReference. Recall that the StationaryFrameOfReference was created when we set up the [HolographicSpace](#), and the coordinate system was acquired at the start of [Update](#).

```
// Check for new input state since the last frame.
SpatialInteractionSourceState^ pointerState = m_spatialInputHandler->CheckForInput();
if (pointerState != nullptr)
{
    // When a Pressed gesture is detected, the sample hologram will be repositioned
    // two meters in front of the user.
    m_spinningCubeRenderer->PositionHologram(
        pointerState->TryGetPointerPose(currentCoordinateSystem)
    );
}
```

Note that the data is tied to a pointer state of some kind. We get this from a spatial input event. The event data object includes a coordinate system, so that you can always relate the gaze direction at the time of the event to whatever spatial coordinate system you need. In fact, you must do so in order to get the pointer pose.

Gesture and motion controller input

In Windows Mixed Reality, both hand [gestures](#) and [motion controllers](#) are handled through the same spatial input APIs, found in the [Windows.UI.Input.Spatial](#) namespace. This enables you to easily handle common actions like **Select** presses the same way across both hands and motion controllers.

There are two levels of API you can target when handling gestures or motion controllers in mixed reality:

- [Interactions](#) ([SourcePressed](#), [SourceReleased](#) and [SourceUpdated](#)), accessed using a [SpatialInteractionManager](#)
- [Composite gestures](#) ([Tapped](#), Hold, Manipulation, Navigation), accessed using a [SpatialGestureRecognizer](#)

Select/Menu/Grasp/Touchpad/Thumbstick interactions: SpatialInteractionManager

To detect low-level presses, releases and updates across hands and input devices in Windows Mixed Reality, you start from a [SpatialInteractionManager](#). The [SpatialInteractionManager](#) has an event that informs the app when a hand or motion controller has detected input.

There are three key kinds of [SpatialInteractionSource](#), each represented by a different [SpatialInteractionSourceKind](#) value:

- **Hand** represents a user's detected hand. Hand sources are available only on HoloLens.
- **Controller** represents a paired motion controller. Motion controllers can offer a variety of capabilities, for example: Select triggers, Menu buttons, Grasp buttons, touchpads and thumbsticks.
- **Voice** represents the user's voice speaking system-detected keywords. This will inject a Select press and release whenever the user says "Select".

To detect presses across any of these interaction sources, you can handle the [SourcePressed](#) event on [SpatialInteractionManager](#) in [SpatialInputHandler.cpp](#):

```
m_interactionManager = SpatialInteractionManager::GetForCurrentView();

// Bind a handler to the SourcePressed event.
m_sourcePressedEventToken =
    m_interactionManager->SourcePressed +=
        ref new TypedEventHandler<SpatialInteractionManager^, SpatialInteractionSourceEventArgs^>(
            bind(&SpatialInputHandler::OnSourcePressed, this, _1, _2)
        );
```

This pressed event is sent to your app asynchronously. Your app or game engine may want to perform some processing right away or you may want to queue up the event data in your input processing routine.

The template includes a helper class to get you started. This template forgoes any processing for simplicity of design. The helper class keeps track of whether one or more **Pressed** events occurred since the last **Update** call. From [SpatialInputHandler.cpp](#):

```
void SpatialInputHandler::OnSourcePressed(SpatialInteractionManager^ sender,
    SpatialInteractionSourceEventArgs^ args)
{
    m_sourceState = args->State;

    //
    // TODO: In your app or game engine, rewrite this method to queue
    //        input events in your input class or event handler.
    //
}
```

If so, it returns the [SpatialInteractionSourceState](#) for the most recent input event during the next Update. From [SpatialInputHandler.cpp](#):

```

// Checks if the user performed an input gesture since the last call to this method.
// Allows the main update loop to check for asynchronous changes to the user
// input state.
SpatialInteractionSourceState^ SpatialInputHandler::CheckForInput()
{
    SpatialInteractionSourceState^ sourceState = m_sourceState;
    m_sourceState = nullptr;
    return sourceState;
}

```

Note that the code above will treat all presses the same way, whether the user is performing a primary **Select** press or a secondary **Menu** or **Grasp** press. The **Select** press is a primary form of interaction supported across hands, motion controllers and voice, triggered either by a hand performing an air-tap, a motion controller with its primary trigger/button pressed, or the user's voice saying "Select". Select presses represent the user's intention to activate the hologram they are targeting.

To reason about which kind of press is occurring, we will modify the SpatialInteractionManager::SourceUpdated event handler. Our code will detect Grasp presses (where available) and use them to reposition the cube. If Grasp is not available, we will use Select presses instead.

Add the following private member declarations to SpatialInputHandler.h:

```

void OnSourceUpdated(
    Windows::UI::Input::Spatial::SpatialInteractionManager^ sender,
    Windows::UI::Input::Spatial::SpatialInteractionSourceEventArgs^ args);
Windows::Foundation::EventRegistrationToken m_sourceUpdatedEventToken;

```

Open SpatialInputHandler.cpp. Add the following event registration to the constructor:

```

m_sourceUpdatedEventToken =
    m_interactionManager->SourceUpdated +=
    ref new TypedEventHandler<SpatialInteractionManager^, SpatialInteractionSourceEventArgs^>(
        bind(&SpatialInputHandler::OnSourceUpdated, this, _1, _2)
    );

```

This is the event handler code. If the input source is experiencing a Grasp, the pointer pose will be stored for the next update loop. Otherwise, it will check for a Select press instead. From SpatialInputHandler.cpp:

```

void SpatialInputHandler::OnSourceUpdated(SpatialInteractionManager^ sender,
    SpatialInteractionSourceEventArgs^ args)
{
    if (args->State->Source->IsGraspSupported)
    {
        if (args->State->IsGrasped)
        {
            m_sourceState = args->State;
        }
    }
    else
    {
        if (args->State->IsSelectPressed)
        {
            m_sourceState = args->State;
        }
    }
}

```

Make sure to unregister the event handler in the destructor. From SpatialInputHandler.cpp:

```
m_interactionManager->SourceUpdated -= m_sourcePressedEventToken;
```

Recompile, and then redeploy. Your template project should now be able to recognize Grasp interactions to reposition the spinning cube.

The SpatialInteractionSource API supports controllers with a wide range of capabilities. In the example shown above, we check to see if Grasp is supported before trying to use it. The SpatialInteractionSource supports the following optional features beyond the common **Select** press:

- **Menu button:** Check support by inspecting the `IsMenuSupported` property. When supported, check the `SpatialInteractionSourceState::IsMenuPressed` property to find out if the menu button is pressed.
- **Grasp button:** Check support by inspecting the `IsGraspSupported` property. When supported, check the `SpatialInteractionSourceState::IsGrasped` property to find out if grasp is activated.

For controllers, the SpatialInteractionSource has a `Controller` property with additional capabilities.

- **HasThumbstick:** If true, the controller has a thumbstick. Inspect the `ControllerProperties` property of the `SpatialInteractionSourceState` to acquire the thumbstick x and y values (`ThumbstickX` and `ThumbstickY`), as well as its pressed state (`IsThumbstickPressed`).
- **HasTouchpad:** If true, the controller has a touchpad. Inspect the `ControllerProperties` property of the `SpatialInteractionSourceState` to acquire the touchpad x and y values (`TouchpadX` and `TouchpadY`), and to know if the user is touching the pad (`IsTouchpadTouched`) and if they are pressing the touchpad down (`IsTouchpadPressed`).
- **SimpleHapticsController:** The SimpleHapticsController API for the controller allows you to inspect the haptics capabilities of the controller, and it also allows you to control haptic feedback.

Note that the range for touchpad and thumbstick from -1 to 1 for both axes (from bottom to top, and from left to right). The range for the analog trigger, which is accessed using the `SpatialInteractionSourceState::SelectPressedValue` property, has a range of 0 to 1. A value of 1 correlates with `IsSelectPressed` being equal to true; any other value correlates with `IsSelectPressed` being equal to false.

Note that for both a hand and a controller, using `SpatialInteractionSourceState::Properties::TryGetLocation` will provide the user's hand position - this is distinct from the pointer pose representing the controller's pointing ray. If you want to draw something at the hand location, use `TryGetLocation`. If you want to raycast from the tip of the controller, get the pointing ray from `TryGetInteractionSourcePose` on the `SpatialPointerPose`.

You can also use the other events on `SpatialInteractionManager`, such as `SourceDetected` and `SourceLost`, to react when hands enter or leave the device's view or when they move in or out of the ready position (index finger raised with palm forward), or when motion controllers are turned on/off or are paired/unpaired.

Grip pose vs. pointing pose

Windows Mixed Reality supports motion controllers in a variety of form factors, with each controller's design differing in its relationship between the user's hand position and the natural "forward" direction that apps should use for pointing when rendering the controller.

To better represent these controllers, there are two kinds of poses you can investigate for each interaction source:

- The **grip pose**, representing the location of either the palm of a hand detected by a HoloLens, or the palm holding a motion controller.
 - On immersive headsets, this pose is best used to render **the user's hand or an object held in the user's hand**, such as a sword or gun.
 - The **grip position**: The palm centroid when holding the controller naturally, adjusted left or right to center the position within the grip.
 - The **grip orientation's Right axis**: When you completely open your hand to form a flat 5-finger pose,

- the ray that is normal to your palm (forward from left palm, backward from right palm)
- The **grip orientation's Forward axis**: When you close your hand partially (as if holding the controller), the ray that points "forward" through the tube formed by your non-thumb fingers.
- The **grip orientation's Up axis**: The Up axis implied by the Right and Forward definitions.
- You can access the grip pose through [SpatialInteractionSourceState.Properties.TryGetLocation\(...\)](#).
- The **pointer pose**, representing the tip of the controller pointing forward.
 - This pose is best used to raycast when **pointing at UI** when you are rendering the controller model itself.
 - You can access the pointer pose through [SpatialInteractionSourceState.Properties.TryGetLocation\(...\).SourcePointerPose](#) or [SpatialInteractionSourceState.TryGetPointerPose\(...\).TryGetInteractionSourcePose](#).

Composite gestures: SpatialGestureRecognizer

A [SpatialGestureRecognizer](#) interprets user interactions from hands, motion controllers, and the "Select" voice command to surface spatial gesture events, which users target using their gaze.

Spatial gestures are a key form of input for Windows Mixed Reality apps. By routing interactions from the [SpatialInteractionManager](#) to a hologram's [SpatialGestureRecognizer](#), apps can detect Tap, Hold, Manipulation, and Navigation events uniformly across hands, voice, and spatial input devices, without having to handle presses and releases manually.

[SpatialGestureRecognizer](#) performs only the minimal disambiguation between the set of gestures that you request. For example, if you request just Tap, the user may hold their finger down as long as they like and a Tap will still occur. If you request both Tap and Hold, after about a second of holding down their finger, the gesture will promote to a Hold and a Tap will no longer occur.

To use [SpatialGestureRecognizer](#), handle the [SpatialInteractionManager](#)'s [InteractionDetected](#) event and grab the [SpatialPointerPose](#) exposed there. Use the user's gaze ray from this pose to intersect with the holograms and surface meshes in the user's surroundings, in order to determine what the user is intending to interact with. Then, route the [SpatialInteraction](#) in the event arguments to the target hologram's [SpatialGestureRecognizer](#), using its [CaptureInteraction](#) method. This starts interpreting that interaction according to the [SpatialGestureSettings](#) set on that recognizer at creation time - or by [TrySetGestureSettings](#).

On HoloLens, interactions and gestures should generally derive their targeting from the user's gaze, rather than trying to render or interact at the hand's location directly. Once an interaction has started, relative motions of the hand may be used to control the gesture, as with the Manipulation or Navigation gesture.

See also

- [Gaze](#)
- [Gestures](#)
- [Motion controllers](#)

Voice input in DirectX

11/6/2018 • 8 minutes to read • [Edit Online](#)

This topic explains how to implement [voice commands](#), and small phrase and sentence recognition in a DirectX app for Windows Mixed Reality.

Use a SpeechRecognizer for continuous recognition of voice commands

In this section, we describe how to use continuous speech recognition to enable voice commands in your app. This walkthrough uses code from the [HolographicVoiceInput Sample](#). When the sample is running, speak the name of one of the registered color commands to change the color of the spinning cube.

First, create a new **Windows::Media::SpeechRecognition::SpeechRecognizer** instance.

From *HolographicVoiceInputSampleMain::CreateSpeechConstraintsForCurrentState*:

```
m_speechRecognizer = ref new SpeechRecognizer();
```

You'll need to create a list of speech commands for the recognizer to listen for. Here, we construct a set of commands to change the color of a hologram. For the sake of convenience, we also create the data that we'll use for the commands later on.

```
m_speechCommandList = ref new Platform::Collections::Vector<String^>();
m_speechCommandData.clear();
m_speechCommandList->Append(StringReference(L"white"));
m_speechCommandData.push_back(float4(1.f, 1.f, 1.f, 1.f));
m_speechCommandList->Append(StringReference(L"grey"));
m_speechCommandData.push_back(float4(0.5f, 0.5f, 0.5f, 1.f));
m_speechCommandList->Append(StringReference(L"green"));
m_speechCommandData.push_back(float4(0.f, 1.f, 0.f, 1.f));
m_speechCommandList->Append(StringReference(L"black"));
m_speechCommandData.push_back(float4(0.1f, 0.1f, 0.1f, 1.f));
m_speechCommandList->Append(StringReference(L"red"));
m_speechCommandData.push_back(float4(1.f, 0.f, 0.f, 1.f));
m_speechCommandList->Append(StringReference(L"yellow"));
m_speechCommandData.push_back(float4(1.f, 1.f, 0.f, 1.f));
m_speechCommandList->Append(StringReference(L"aquamarine"));
m_speechCommandData.push_back(float4(0.f, 1.f, 1.f, 1.f));
m_speechCommandList->Append(StringReference(L"blue"));
m_speechCommandData.push_back(float4(0.f, 0.f, 1.f, 1.f));
m_speechCommandList->Append(StringReference(L"purple"));
m_speechCommandData.push_back(float4(1.f, 0.f, 1.f, 1.f));
```

Commands can be specified using phonetic words that might not be in a dictionary:

```
m_speechCommandList->Append(StringReference(L"SpeechRecognizer"));
m_speechCommandData.push_back(float4(0.5f, 0.1f, 1.f, 1.f));
```

The list of commands is loaded into the list of constraints for the speech recognizer. This is done by using a [SpeechRecognitionListConstraint](#) object.

```

SpeechRecognitionListConstraint^ spConstraint = ref new SpeechRecognitionListConstraint(m_speechCommandList);
m_speechRecognizer->Constraints->Clear();
m_speechRecognizer->Constraints->Append(spConstraint);
create_task(m_speechRecognizer->CompileConstraintsAsync()).then([this](SpeechRecognitionCompilationResult^ compilationResult)
{
    if (compilationResult->Status == SpeechRecognitionResultStatus::Success)
    {
        m_speechRecognizer->ContinuousRecognitionSession->StartAsync();
    }
    else
    {
        // Handle errors here.
    }
});

```

Subscribe to the [ResultGenerated](#) event on the speech recognizer's [SpeechContinuousRecognitionSession](#). This event notifies your app when one of your commands has been recognized.

```

m_speechRecognizer->ContinuousRecognitionSession->ResultGenerated +=
    ref new TypedEventHandler<SpeechContinuousRecognitionSession^,
SpeechContinuousRecognitionResultGeneratedEventArgs^>(
    std::bind(&HolographicVoiceInputSampleMain::OnResultGenerated, this, _1, _2)
);

```

Your **OnResultGenerated** event handler receives event data in a

[SpeechContinuousRecognitionResultGeneratedEventArgs](#) instance. If the confidence is greater than the threshold you have defined, your app should note that the event happened. Save the event data so that you can make use of it in a subsequent update loop.

From *HolographicVoiceInputSampleMain.cpp*:

```

// Change the cube color, if we get a valid result.
void HolographicVoiceInputSampleMain::OnResultGenerated(SpeechContinuousRecognitionSession ^sender,
SpeechContinuousRecognitionResultGeneratedEventArgs ^args)
{
    if (args->Result->RawConfidence > 0.5f)
    {
        m_lastCommand = args->Result->Text;
    }
}

```

Make use of the data however applicable to your app scenario. In our example code, we change the color of the spinning hologram cube according to the user's command.

From *HolographicVoiceInputSampleMain::Update*:

```

// Check for new speech input since the last frame.
if (m_lastCommand != nullptr)
{
    auto command = m_lastCommand;
    m_lastCommand = nullptr;

    int i = 0;
    for each (auto& iter in m_speechCommandList)
    {
        if (iter == command)
        {
            m_spinningCubeRenderer->SetColor(m_speechCommandData[i]);
            break;
        }

        ++i;
    }
}

```

Use dictation for one-shot recognition of speech phrases and sentences

You can configure a speech recognizer to listen for phrases or sentences spoken by the user. In this case, we apply a SpeechRecognitionTopicConstraint that tells the speech recognizer what type of input to expect. The app workflow is as follows, for this type of use case:

1. Your app creates the SpeechRecognizer, provides UI prompts, and starts listening for a command to be spoken immediately.
2. The user speaks a phrase, or sentence.
3. Recognition of the user's speech is performed, and a result is returned to the app. At this point, your app should provide a UI prompt indicating that recognition has occurred.
4. Depending on the confidence level you want to respond to and the confidence level of the speech recognition result, your app can process the result and respond as appropriate.

This section describes how to create a SpeechRecognizer, compile the constraint, and listen for speech input.

The following code compiles the topic constraint, which in this case is optimized for Web search.

```

auto constraint = ref new SpeechRecognitionTopicConstraint(SpeechRecognitionScenario::WebSearch,
L"webSearch");
m_speechRecognizer->Constraints->Clear();
m_speechRecognizer->Constraints->Append(constraint);
return create_task(m_speechRecognizer->CompileConstraintsAsync())
    .then([this](task<SpeechRecognitionCompilationResult^> previousTask)
{

```

If compilation succeeds, we can proceed with speech recognition.

```

try
{
    SpeechRecognitionCompilationResult^ compilationResult = previousTask.get();

    // Check to make sure that the constraints were in a proper format and the recognizer was able to
    // compile it.
    if (compilationResult->Status == SpeechRecognitionResultStatus::Success)
    {
        // If the compilation succeeded, we can start listening for the user's spoken phrase or
        sentence.
        create_task(m_speechRecognizer->RecognizeAsync()).then([this](task<SpeechRecognitionResult^>&
previousTask)
    {

```

The result is then returned to the app. If we are confident enough in the result, we can process the command. This code example processes results with at least Medium confidence.

```

try
{
    auto result = previousTask.get();

    if (result->Status != SpeechRecognitionResultStatus::Success)
    {
        PrintWstringToDebugConsole(
            std::wstring(L"Speech recognition was not successful: ") +
            result->Status.ToString()->Data() +
            L"\n"
        );
    }

    // In this example, we look for at least medium confidence in the speech result.
    if ((result->Confidence == SpeechRecognitionConfidence::High) ||
        (result->Confidence == SpeechRecognitionConfidence::Medium))
    {
        // If the user said a color name anywhere in their phrase, it will be recognized in
        the
        // Update loop; then, the cube will change color.
        m_lastCommand = result->Text;

        PrintWstringToDebugConsole(
            std::wstring(L"Speech phrase was: ") +
            m_lastCommand->Data() +
            L"\n"
        );
    }
    else
    {
        PrintWstringToDebugConsole(
            std::wstring(L"Recognition confidence not high enough: ") +
            result->Confidence.ToString()->Data() +
            L"\n"
        );
    }
}

```

Whenever you use speech recognition, you should watch for exceptions that could indicate the user has turned off the microphone in the system privacy settings. This can happen during initialization, or during recognition.

```

        catch (Exception^ exception)
        {
            // Note that if you get an "Access is denied" exception, you might need to enable the
            microphone
            // privacy setting on the device and/or add the microphone capability to your app
            manifest.

            PrintWstringToDebugConsole(
                std::wstring(L"Speech recognizer error: ") +
                exception->ToString()->Data() +
                L"\n"
            );
        }
    });

    return true;
}
else
{
    OutputDebugStringW(L"Could not initialize predefined grammar speech engine!\n");

    // Handle errors here.
    return false;
}
}

catch (Exception^ exception)
{
    // Note that if you get an "Access is denied" exception, you might need to enable the microphone
    // privacy setting on the device and/or add the microphone capability to your app manifest.

    PrintWstringToDebugConsole(
        std::wstring(L"Exception while trying to initialize predefined grammar speech engine:") +
        exception->Message->Data() +
        L"\n"
    );

    // Handle exceptions here.
    return false;
}
);
}

```

NOTE: There are several predefined [SpeechRecognitionScenarios](#) available for optimizing speech recognition.

- If you want to optimize for dictation, use the Dictation scenario:

```

// Compile the dictation topic constraint, which optimizes for speech dictation.
auto dictationConstraint = ref new SpeechRecognitionTopicConstraint(SpeechRecognitionScenario::Dictation,
"dictation");
_m_speechRecognizer->Constraints->Append(dictationConstraint);

```

- When using speech to perform a Web search, you can use a Web-specific scenario constraint as follows:

```

// Add a web search topic constraint to the recognizer.
auto webSearchConstraint = ref new SpeechRecognitionTopicConstraint(SpeechRecognitionScenario::WebSearch,
"webSearch");
speechRecognizer->Constraints->Append(webSearchConstraint);

```

- Use the form constraint to fill out forms. In this case, it is best to apply your own grammar that is optimized for filling out your form.

```
// Add a form constraint to the recognizer.  
auto formConstraint = ref new SpeechRecognitionTopicConstraint(SpeechRecognitionScenario::FormFilling,  
"formFilling");  
speechRecognizer->Constraints->Append(formConstraint );
```

- You can provide your own grammar using the SRGS format.

Use continuous, freeform speech dictation

See the Windows 10 UWP speech code sample for the continuous dictation scenario [here](#).

Handle degradation in quality

Conditions in the environment can sometimes prevent speech recognition from working. For example, the room might be too noisy or the user might speak at too high a volume. The speech recognition API provides info, where possible, about conditions that have caused a degradation in quality.

This information is pushed to your app using a WinRT event. Here is an example of how to subscribe to this event.

```
m_speechRecognizer->RecognitionQualityDegrading +=  
ref new TypedEventHandler<SpeechRecognizer^, SpeechRecognitionQualityDegradingEventArgs^>(  
    std::bind(&HolographicVoiceInputSampleMain::OnSpeechQualityDegraded, this, _1, _2)  
>;
```

In our code sample, we choose to write the conditions info to the debug console. An app might want to provide feedback to the user via UI, speech synthesis, and so on, or it might need to behave differently when speech is interrupted by a temporary reduction in quality.

```

void HolographicSpeechPromptSampleMain::OnSpeechQualityDegraded(SpeechRecognizer^ recognizer,
SpeechRecognitionQualityDegradingEventArgs^ args)
{
    switch (args->Problem)
    {
        case SpeechRecognitionAudioProblem::TooFast:
            OutputDebugStringW(L"The user spoke too quickly.\n");
            break;

        case SpeechRecognitionAudioProblem::TooSlow:
            OutputDebugStringW(L"The user spoke too slowly.\n");
            break;

        case SpeechRecognitionAudioProblem::TooQuiet:
            OutputDebugStringW(L"The user spoke too softly.\n");
            break;

        case SpeechRecognitionAudioProblem::TooLoud:
            OutputDebugStringW(L"The user spoke too loudly.\n");
            break;

        case SpeechRecognitionAudioProblem::TooNoisy:
            OutputDebugStringW(L"There is too much noise in the signal.\n");
            break;

        case SpeechRecognitionAudioProblem::NoSignal:
            OutputDebugStringW(L"There is no signal.\n");
            break;

        case SpeechRecognitionAudioProblem::None:
        default:
            OutputDebugStringW(L"An error was reported with no information.\n");
            break;
    }
}

```

If you are not using ref classes to create your DirectX app, you must unsubscribe from the event before releasing or recreating your speech recognizer. The `HolographicSpeechPromptSample` has a routine to stop recognition, and unsubscribe from events like so:

```

Concurrency::task<void> HolographicSpeechPromptSampleMain::StopCurrentRecognizerIfExists()
{
    return create_task([this]()
    {
        if (m_speechRecognizer != nullptr)
        {
            return create_task(m_speechRecognizer->StopRecognitionAsync()).then([this]()
            {
                m_speechRecognizer->RecognitionQualityDegrading -= m_speechRecognitionQualityDegradedToken;

                if (m_speechRecognizer->ContinuousRecognitionSession != nullptr)
                {
                    m_speechRecognizer->ContinuousRecognitionSession->ResultGenerated -=
m_speechRecognizerResultEventToken;
                }
            });
        }
        else
        {
            return create_task([this]() { m_speechRecognizer = nullptr; });
        }
    });
}

```

Use speech synthesis to provide audible voice prompts

The holographic speech samples use speech synthesis to provide audible instructions to the user. This topic walks through the process of creating a synthesized voice sample, and playing it back using the HRTF audio APIs.

You should provide your own speech prompts when requesting phrase input. This can also be helpful for indicating when speech commands can be spoken, for a continuous recognition scenario. Here is an example of how to do that with a speech synthesizer; note that you could also use a pre-recorded voice clip, a visual UI, or other indicator of what to say, for example in scenarios where the prompt is not dynamic.

First, create the SpeechSynthesizer object:

```
auto speechSynthesizer = ref new Windows::Media::SpeechSynthesis::SpeechSynthesizer();
```

You also need a string with the text to be synthesized:

```
// Phrase recognition works best when requesting a phrase or sentence.  
StringReference voicePrompt = L"At the prompt: Say a phrase, asking me to change the cube to a specific  
color.;"
```

Speech is synthesized asynchronously using SynthesizeTextToStreamAsync. Here, we kick off an async task to synthesize the speech.

```
create_task(speechSynthesizer->SynthesizeTextToStreamAsync(voicePrompt),  
task_continuation_context::use_current())  
.then([this, speechSynthesizer](task<Windows::Media::SpeechSynthesis::SpeechSynthesisStream^>  
synthesisStreamTask)  
{  
    try  
    {
```

The speech synthesis is sent as a byte stream. We can initialize an XAudio2 voice using that byte stream; for our holographic code samples, we play it back as an HRTF audio effect.

```
Windows::Media::SpeechSynthesis::SpeechSynthesisStream^ stream = synthesisStreamTask.get();  
  
auto hr = m_speechSynthesisSound.Initialize(stream, 0);  
if (SUCCEEDED(hr))  
{  
    m_speechSynthesisSound.SetEnvironment(HrtfEnvironment::Small);  
    m_speechSynthesisSound.Start();  
  
    // Amount of time to pause after the audio prompt is complete, before listening  
    // for speech input.  
    static const float bufferTime = 0.15f;  
  
    // Wait until the prompt is done before listening.  
    m_secondsUntilSoundIsComplete = m_speechSynthesisSound.GetDuration() + bufferTime;  
    m_waitingForSpeechPrompt = true;  
}
```

As with speech recognition, speech synthesis will throw an exception if something goes wrong.

```
catch (Exception^ exception)
{
    PrintWstringToDebugConsole(
        std::wstring(L"Exception while trying to synthesize speech: ") +
        exception->Message->Data() +
        L"\n"
    );
}

// Handle exceptions here.
}
});
```

See also

- [Speech app design](#)
- [Spatial sound in DirectX](#)
- [SpeechRecognitionAndSynthesis sample](#)

Spatial sound in DirectX

11/6/2018 • 8 minutes to read • [Edit Online](#)

Add spatial sound to your Windows Mixed Reality apps based on DirectX by using the [XAudio2](#) and [xAPO](#) audio libraries.

This topic uses sample code from the [HolographicHrtfAudioSample](#).

Overview of Head Relative Spatial Sound

Spatial sound is implemented as an **audio processing object (APO)** that uses a **head related transfer function (HRTF)** filter to *spatialize* an ordinary audio stream.

Include these header files in pch.h to access the audio APIs:

- XAudio2.h
- xapo.h
- hrtfapoapi.h

To set up spatial sound:

1. Call [CreateHrtfApo](#) to initialize a new APO for HRTF audio.
2. Assign the [HRTF parameters](#) and [HRTF environment](#) to define the acoustic characteristics of the spatial sound APO.
3. Set up the XAudio2 engine for HRTF processing.
4. Create an [IXAudio2SourceVoice](#) object and call [Start](#).

Implementing HRTF and spatial sound in your DirectX app

You can achieve a variety of effects by configuring the HRTF APO with different parameters and environments. Use the following code to explore the possibilities. Download the Universal Windows Platform code sample from here: [Spatial sound sample](#)

Helper types are available in these files:

- [AudioFileReader.cpp](#): Loads audio files by using Media Foundation and the [IMFSourceReader interface](#).
- [XAudio2Helpers.h](#): Implements the [SetupXAudio2](#) function to initialize XAudio2 for HRTF processing.

Add spatial sound for an omnidirectional source

Some holograms in the user's surroundings emit sound equally in all directions. The following code shows how to initialize an APO to emit omnidirectional sound. In this example, we apply this concept to the spinning cube from the Windows Holographic app template. For the complete code listing, see [OmnidirectionalSound.cpp](#).

Window's spatial sound engine only supports 48k samplerate for playback. Most middleware programs, like Unity, will automatically convert sound files into the desired format, but if you start tinkering at lower levels in the audio system or making your own, this is very important to remember to prevent crashes or undesired behaviour like HRTF system failure.

First, we need to initialize the APO. In our holographic sample app, we choose to do this once we have the [HolographicSpace](#).

From [HolographicHrtfAudioSampleMain::SetHolographicSpace\(\)](#):

```
// Spatial sound
auto hr = m_omnidirectionalSound.Initialize(L"assets//MonoSound.wav");
```

The implementation of Initialize, from *OmnidirectionalSound.cpp*:

```
// Initializes an APO that emits sound equally in all directions.
HRESULT OmnidirectionalSound::Initialize( LPCWSTR filename )
{
    // _audioFile is of type AudioFileReader, which is defined in AudioFileReader.cpp.
    auto hr = _audioFile.Initialize( filename );

    ComPtr<IXAPO> xapo;
    if ( SUCCEEDED( hr ) )
    {
        // Passing in nullptr as the first arg for HrtfApoInit initializes the APO with defaults of
        // omnidirectional sound with natural distance decay behavior.
        // CreateHrtfApo fails with E_NOTIMPL on unsupported platforms.
        hr = CreateHrtfApo( nullptr, &xapo );
    }

    if ( SUCCEEDED( hr ) )
    {
        // _hrtfParams is of type ComPtr<IXAPOHrtfParameters>.
        hr = xapo.As( &_hrtfParams );
    }

    // Set the default environment.
    if ( SUCCEEDED( hr ) )
    {
        hr = _hrtfParams->SetEnvironment( HrtfEnvironment::Outdoors );
    }

    // Initialize an XAudio2 graph that hosts the HRTF xAPO.
    // The source voice is used to submit audio data and control playback.
    if ( SUCCEEDED( hr ) )
    {
        hr = SetupXAudio2( _audioFile.GetFormat(), xapo.Get(), &_xaudio2, &_sourceVoice );
    }

    // Submit audio data to the source voice.
    if ( SUCCEEDED( hr ) )
    {
        XAUDIO2_BUFFER buffer{ };
        buffer.AudioBytes = static_cast<UINT32>( _audioFile.GetSize() );
        buffer.pAudioData = _audioFile.GetData();
        buffer.LoopCount = XAUDIO2_LOOP_INFINITE;

        // _sourceVoice is of type IXAudio2SourceVoice*.
        hr = _sourceVoice->SubmitSourceBuffer( &buffer );
    }

    return hr;
}
```

After the APO is configured for HRTF, you call [Start](#) on the source voice to play the audio. In our sample app, we choose to put it on a loop so that you can continue to hear the sound coming from the cube.

From *HolographicHrtfAudioSampleMain::SetHolographicSpace()*:

```

if (SUCCEEDED(hr))
{
    m_omnidirectionalSound.SetEnvironment(HrtfEnvironment::Small);
    m_omnidirectionalSound.OnUpdate(m_spinningCubeRenderer->GetPosition());
    m_omnidirectionalSound.Start();
}

```

From *OmnidirectionalSound.cpp*:

```

HRESULT OmnidirectionalSound::Start()
{
    _lastTick = GetTickCount64();
    return _sourceVoice->Start();
}

```

Now, whenever we update the frame, we need to update the hologram's position relative to the device itself. This is because HRTF positions are always expressed relative to the user's head, including the head position and orientation.

To do this in a *HolographicSpace*, we need to construct a transform matrix from our *SpatialStationaryFrameOfReference* coordinate system to a coordinate system that is fixed to the device itself.

From *HolographicHrtfAudioSampleMain::Update()*:

```

m_spinningCubeRenderer->Update(m_timer);

SpatialPointerPose^ currentPose = SpatialPointerPose::TryGetAtTimestamp(currentCoordinateSystem, prediction-
>Timestamp);
if (currentPose != nullptr)
{
    // Use a coordinate system built from a pointer pose.
    SpatialPointerPose^ pose = SpatialPointerPose::TryGetAtTimestamp(currentCoordinateSystem, prediction-
>Timestamp);
    if (pose != nullptr)
    {
        float3 headPosition = pose->Head->Position;
        float3 headUp = pose->Head->UpDirection;
        float3 headDirection = pose->Head->ForwardDirection;

        // To construct a rotation matrix, we need three vectors that are mutually orthogonal.
        // The first vector is the gaze vector.
        float3 negativeZAxis = normalize(headDirection);

        // The second vector should end up pointing away from the horizontal plane of the device.
        // We first guess by using the head "up" direction.
        float3 positiveYAxisGuess = normalize(headUp);

        // The third vector completes the set by being orthogonal to the other two.
        float3 positiveXAxis = normalize(cross(negativeZAxis, positiveYAxisGuess));

        // Now, we can correct our "up" vector guess by redetermining orthogonality.
        float3 positiveYAxis = normalize(cross(negativeZAxis, positiveXAxis));

        // The rotation matrix is formed as a standard basis rotation.
        float4x4 rotationTransform =
        {
            positiveXAxis.x, positiveYAxis.x, negativeZAxis.x, 0.f,
            positiveXAxis.y, positiveYAxis.y, negativeZAxis.y, 0.f,
            positiveXAxis.z, positiveYAxis.z, negativeZAxis.z, 0.f,
            0.f, 0.f, 0.f, 1.f,
        };

        // The translate transform can be constructed using the Windows::Foundation::Numerics API.
    }
}

```

```

float4x4 translationTransform = make_float4x4_translation(-headPosition);

// Now, we have a basis transform from our spatial coordinate system to a device-relative
// coordinate system.
float4x4 coordinateSystemTransform = translationTransform * rotationTransform;

// Reinterpret the cube position in the device's coordinate system.
float3 cubeRelativeToHead = transform(m_spinningCubeRenderer->GetPosition(),
coordinateSystemTransform);

// Note that at (0, 0, 0) exactly, the HRTF audio will simply pass through audio. We can use a minimal
offset
// to simulate a zero distance when the hologram position vector is exactly at the device origin in
order to
// allow HRTF to continue functioning in this edge case.
float distanceFromHologramToHead = length(cubeRelativeToHead);
static const float distanceMin = 0.00001f;
if (distanceFromHologramToHead < distanceMin)
{
    cubeRelativeToHead = float3(0.f, distanceMin, 0.f);
}

// Position the spatial sound source on the hologram.
m_omnidirectionalSound.OnUpdate(cubeRelativeToHead);

// For debugging, it can be interesting to observe the distance in the debugger.
/*
std::wstring distanceString = L"Distance from hologram to head: ";
distanceString += std::to_wstring(distanceFromHologramToHead);
distanceString += L"\n";
OutputDebugStringW(distanceString.c_str());
*/
}
}

```

The HRTF position is applied directly to the sound APO by the OmnidirectionalSound helper class.

From *OmnidirectionalSound::OnUpdate*:

```

HRESULT OmnidirectionalSound::OnUpdate(_In_ Numerics::float3 position)
{
    auto hrtfPosition = HrtfPosition{ position.x, position.y, position.z };
    return _hrtfParams->SetSourcePosition(&hrtfPosition);
}

```

That's it! Continue reading to learn more about what you can do with HRTF audio and Windows Holographic.

Initialize spatial sound for a directional source

Some holograms in the user's surroundings emit sound mostly in one direction. This sound pattern is named *cardioid* because it looks like a cartoon heart. The following code shows how to initialize an APO to emit directional sound. For the complete code listing, see [CardioidSound.cpp](#).

After the APO is configured for HRTF, call [Start](#) on the source voice to play the audio.

```

// Initializes an APO that emits directional sound.
HRESULT CardioidSound::Initialize( LPCWSTR filename )
{
    // _audioFile is of type AudioFileReader, which is defined in AudioFileReader.cpp.
    auto hr = _audioFile.Initialize( filename );
    if ( SUCCEEDED( hr ) )
    {
        // Initialize with "Scaling" fully directional and "Order" with broad radiation pattern.
        // As the order goes higher, the cardioid directivity region becomes narrower.
        // Any direct path signal outside of the directivity region will be attenuated based on the scaling
}

```

```

factor.

    // For example, if scaling is set to 1 (fully directional) the direct path signal outside of the
    // directivity
    // region will be fully attenuated and only the reflections from the environment will be audible.
    hr = ConfigureApo( 1.0f, 4.0f );
}
return hr;
}

HRESULT CardioidSound::ConfigureApo( float scaling, float order )
{
    // Cardioid directivity configuration:
    // Directivity is specified at xAPO instance initialization and can't be changed per frame.
    // To change directivity, stop audio processing and reinitialize another APO instance with the new
    // directivity.
    HrtfDirectivityCardioid cardioid;
    cardioid.directivity.type = HrtfDirectivityType::Cardioid;
    cardioid.directivity.scaling = scaling;
    cardioid.order = order;

    // APO initialization
    HrtfApoInit apoInit;
    apoInit.directivity = &cardioid.directivity;
    apoInit.distanceDecay = nullptr; // nullptr specifies natural distance decay behavior (simulates real
    world)

    // CreateHrtfApo fails with E_NOTIMPL on unsupported platforms.
    ComPtr<IXAPO> xapo;
    auto hr = CreateHrtfApo( &apoInit, &xapo );

    if ( SUCCEEDED( hr ) )
    {
        hr = xapo.As( &_hrtfParams );
    }

    // Set the initial environment.
    // Environment settings configure the "distance cues" used to compute the early and late reverberations.
    if ( SUCCEEDED( hr ) )
    {
        hr = _hrtfParams->SetEnvironment( _HrtfEnvironment::Outdoors );
    }

    // Initialize an XAudio2 graph that hosts the HRTF xAPO.
    // The source voice is used to submit audio data and control playback.
    if ( SUCCEEDED( hr ) )
    {
        hr = SetupXAudio2( _audioFile.GetFormat(), xapo.Get(), &_xaudio2, &_sourceVoice );
    }

    // Submit audio data to the source voice
    if ( SUCCEEDED( hr ) )
    {
        XAUDIO2_BUFFER buffer{ };
        buffer.AudioBytes = static_cast<UINT32>( _audioFile.GetSize() );
        buffer.pAudioData = _audioFile.GetData();
        buffer.LoopCount = XAUDIO2_LOOP_INFINITE;
        hr = _sourceVoice->SubmitSourceBuffer( &buffer );
    }

    return hr;
}

```

Implement custom decay

You can override the rate at which a spatial sound falls off with distance and/or at what distance it cuts off completely. To implement custom decay behavior on a spatial sound, populate an [HrtfDistanceDecay struct](#) and assign it to the **distanceDecay** field in an [HrtfApolnit struct](#) before passing it to the [CreateHrtfApo](#) function.

Add the following code to the **Initialize** method shown previously to specify custom decay behavior. For the complete code listing, see [CustomDecay.cpp](#).

```
HRESULT CustomDecaySound::Initialize( LPCWSTR filename )
{
    auto hr = _audioFile.Initialize( filename );

    ComPtr<IXAPO> xapo;
    if ( SUCCEEDED( hr ) )
    {
        HrtfDistanceDecay customDecay;
        customDecay.type = HrtfDistanceDecayType::CustomDecay;           // Custom decay behavior, we'll
pass in the gain value on every frame.
        customDecay.maxGain = 0;                                         // 0dB max gain
        customDecay.minGain = -96.0f;                                      // -96dB min gain
        customDecay.unityGainDistance = HRTF_DEFAULT_UNITY_GAIN_DISTANCE; // Default unity gain distance
        customDecay.cutoffDistance = HRTF_DEFAULT_CUTOFF_DISTANCE;       // Default cutoff distance

        // Setting the directivity to nullptr specifies omnidirectional sound.
        HrtfApoInit init;
        init.directivity = nullptr;
        init.distanceDecay = &customDecay;

        // CreateHrtfApo will fail with E_NOTIMPL on unsupported platforms.
        hr = CreateHrtfApo( &init, &xapo );
    }
    ...
}
```

Spatial mapping in DirectX

11/6/2018 • 16 minutes to read • [Edit Online](#)

This topic describes how to implement [spatial mapping](#) in your DirectX app. This includes a detailed explanation of the spatial mapping sample application that is included with the Universal Windows Platform SDK.

This topic uses code from the [HolographicSpatialMapping](#) UWP code sample.

DirectX development overview

Native application development for spatial mapping uses the APIs under the [Windows.Perception.Spatial](#) namespace. These APIs provide full control of spatial mapping functionality, in a manner directly analogous to the spatial mapping APIs exposed by [Unity](#).

Perception APIs

The primary types provided for spatial mapping development are as follows:

- [SpatialSurfaceObserver](#) provides information about surfaces in application-specified regions of space near the user, in the form of [SpatialSurfaceInfo](#) objects.
- [SpatialSurfaceInfo](#) describes a single extant spatial surface, including a unique ID, bounding volume and time of last change. It will provide a [SpatialSurfaceMesh](#) asynchronously upon request.
- [SpatialSurfaceMeshOptions](#) contains parameters used to customize the [SpatialSurfaceMesh](#) objects requested from [SpatialSurfaceInfo](#).
- [SpatialSurfaceMesh](#) represents the mesh data for a single spatial surface. The data for vertex positions, vertex normals and triangle indices is contained in member [SpatialSurfaceMeshBuffer](#) objects.
- [SpatialSurfaceMeshBuffer](#) wraps a single type of mesh data.

When developing an application using these APIs, your basic program flow will look like this (as demonstrated in the sample application described below):

- **Set up your [SpatialSurfaceObserver](#)**
 - Call [RequestAccessAsync](#), to ensure that the user has given permission for your application to use the device's spatial mapping capabilities.
 - Instantiate a [SpatialSurfaceObserver](#) object.
 - Call [SetBoundingVolumes](#) to specify the regions of space in which you want information about spatial surfaces. You may modify these regions in the future by simply calling this function again. Each region is specified using a [SpatialBoundingVolume](#).
 - Register for the [ObservedSurfacesChanged](#) event, which will fire whenever new information is available about the spatial surfaces in the regions of space you have specified.
- **Process [ObservedSurfacesChanged](#) events**
 - In your event handler, call [GetObservedSurfaces](#) to receive a map of [SpatialSurfaceInfo](#) objects. Using this map, you can update your records of which spatial surfaces [exist in the user's environment](#).
 - For each [SpatialSurfaceInfo](#) object, you may query [TryGetBounds](#) to determine the spatial extents of the surface, expressed in a [spatial coordinate system](#) of your choosing.
 - If you decide to request mesh for a spatial surface, call [TryComputeLatestMeshAsync](#). You may provide options specifying the desired density of triangles, and the format of the returned mesh data.
- **Receive and process mesh**
 - Each call to [TryComputeLatestMeshAsync](#) will asynchronously return one [SpatialSurfaceMesh](#) object.
 - From this object you can access the contained [SpatialSurfaceMeshBuffer](#) objects in order to access the

triangle indices, vertex positions and (if requested) vertex normals of the mesh. This data will be in a format directly compatible with the [Direct3D 11 APIs](#) used for rendering meshes.

- From here your application can optionally perform analysis or [processing](#) of the mesh data, and use it for [rendering](#) and physics [raycasting and collision](#).
- One important detail to note is that you must apply a scale to the mesh vertex positions (for example in the vertex shader used for rendering the meshes), to convert them from the optimized integer units in which they are stored in the buffer, to meters. You can retrieve this scale by calling [VertexPositionScale](#).

Troubleshooting

- Don't forget to scale mesh vertex positions in your vertex shader, using the scale returned by [SpatialSurfaceMesh.VertexPositionScale](#)

Spatial Mapping code sample walkthrough

The [Holographic Spatial Mapping](#) code sample includes code that you can use to get started loading surface meshes into your app, including infrastructure for managing and rendering surface meshes.

Now, we walk through how to add surface mapping capability to your DirectX app. You can add this code to your [Windows Holographic app template](#) project, or you can follow along by browsing through the code sample mentioned above. This code sample is based on the Windows Holographic app template.

Set up your app to use the spatialPerception capability

Your app must be able to use the spatial mapping capability. This is necessary because the spatial mesh is a representation of the user's environment, which may be considered private data. Declare this capability in the package.appxmanifest file for your app. Here's an example:

```
<Capabilities>
  <uap2:Capability Name="spatialPerception" />
</Capabilities>
```

The capability comes from the **uap2** namespace. To get access to this namespace in your manifest, include it as an *xmns* attribute in the `<Package>` element. Here's an example:

```
<Package
  xmlns="http://schemas.microsoft.com/appx/manifest/foundation/windows10"
  xmlns:mp="http://schemas.microsoft.com/appx/2014/phone/manifest"
  xmlns:uap="http://schemas.microsoft.com/appx/manifest/uap/windows10"
  xmlns:uap2="http://schemas.microsoft.com/appx/manifest/uap/windows10/2"
  IgnorableNamespaces="uap uap2 mp"
  >
```

Check for spatial mapping feature support

Windows Mixed Reality supports a wide range of devices, including devices which do not support spatial mapping. If your app can use spatial mapping, or must use spatial mapping, to provide functionality, it should check to make sure that spatial mapping is supported before trying to use it. For example, if spatial mapping is required by your mixed reality app, it should display a message to that effect if a user tries running it on a device without spatial mapping. Or, your app may be able to render its own virtual environment in place of the user's environment, providing an experience that is similar to what would happen if spatial mapping were available. In any event, this API allows your app to be aware of when it will not get spatial mapping data and respond in the appropriate way.

To check the current device for spatial mapping support, first make sure the UWP contract is at level 4 or greater and then call `SpatialSurfaceObserver::IsSupported()`. Here's how to do so in the context of the [Holographic Spatial Mapping](#) code sample. Support is checked just before requesting access.

The `SpatialSurfaceObserver::IsSupported()` API is available starting in SDK version 15063. If necessary, retarget your project to platform version 15063 before using this API.

```
if (_surfaceObserver == nullptr)
{
    using namespace Windows::Foundation::Metadata;
    if (ApiInformation::IsApiContractPresent(L"Windows.Foundation.UniversalApiContract", 4))
    {
        if (!SpatialSurfaceObserver::IsSupported())
        {
            // The current system does not have spatial mapping capability.
            // Turn off spatial mapping.
            m_spatialPerceptionAccessRequested = true;
            m_surfaceAccessAllowed = false;
        }
    }

    if (!m_spatialPerceptionAccessRequested)
    {
        /// etc ...
    }
}
```

Note that when the UWP contract is less than level 4, the app should proceed as though the device is capable of doing spatial mapping.

Request access to spatial mapping data

Your app needs to request permission to access spatial mapping data before trying to create any surface observers. Here's an example based upon our Surface Mapping code sample, with more details provided later on this page:

```
auto initSurfaceObserverTask = create_task(SpatialSurfaceObserver::RequestAccessAsync());
initSurfaceObserverTask.then([this, coordinateSystem]
(Windows::Perception::Spatial::SpatialPerceptionAccessStatus status)
{
    if (status == SpatialPerceptionAccessStatus::Allowed)
    {
        // Create a surface observer.
    }
    else
    {
        // Handle spatial mapping unavailable.
    }
})
```

Create a surface observer

The **Windows::Perception::Spatial::Surfaces** namespace includes the [SpatialSurfaceObserver](#) class, which observes one or more volumes that you specify in a [SpatialCoordinateSystem](#). Use a [SpatialSurfaceObserver](#) instance to access surface mesh data in real time.

From **AppMain.h**:

```
// Obtains surface mapping data from the device in real time.
Windows::Perception::Spatial::Surfaces::SpatialSurfaceObserver^     m_surfaceObserver;
Windows::Perception::Spatial::Surfaces::SpatialSurfaceMeshOptions^   m_surfaceMeshOptions;
```

As noted in the previous section, you must request access to spatial mapping data before your app can use it. This access is granted automatically on the HoloLens.

```

// The surface mapping API reads information about the user's environment. The user must
// grant permission to the app to use this capability of the Windows Mixed Reality device.
auto initSurfaceObserverTask = create_task(SpatialSurfaceObserver::RequestAccessAsync());
initSurfaceObserverTask.then([this, coordinateSystem]
(Windows::Perception::Spatial::SpatialPerceptionAccessStatus status)
{
    if (status == SpatialPerceptionAccessStatus::Allowed)
    {
        // If status is allowed, we can create the surface observer.
        m_surfaceObserver = ref new SpatialSurfaceObserver();
    }
}

```

Next, you need to configure the surface observer to observe a specific bounding volume. Here, we observe a box that is 20x20x5 meters, centered at the origin of the coordinate system.

```

// The surface observer can now be configured as needed.

// In this example, we specify one area to be observed using an axis-aligned
// bounding box 20 meters in width and 5 meters in height and centered at the
// origin.
SpatialBoundingBox aabb =
{
    { 0.f, 0.f, 0.f },
    {20.f, 20.f, 5.f },
};

SpatialBoundingVolume^ bounds = SpatialBoundingVolume::FromBox(coordinateSystem, aabb);
m_surfaceObserver->SetBoundingVolume(bounds);

```

Note that you can set multiple bounding volumes instead.

This is pseudocode:

```
m_surfaceObserver->SetBoundingVolumes(/* iterable collection of bounding volumes*/);
```

It is also possible to use other bounding shapes - such as a view frustum, or a bounding box that is not axis aligned.

This is pseudocode:

```
m_surfaceObserver->SetBoundingVolume(
    SpatialBoundingVolume::FromFrustum(/*SpatialCoordinateSystem*/, /*SpatialBoundingFrustum*/)
);
```

If your app needs to do anything differently when surface mapping data is not available, you can write code to respond to the case where the [SpatialPerceptionAccessStatus](#) is not **Allowed** - for example, it will not be allowed on PCs with immersive devices attached because those devices don't have hardware for spatial mapping. For these devices, you should instead rely on the spatial stage for information about the user's environment and device configuration.

Initialize and update the surface mesh collection

If the surface observer was successfully created, we can proceed to initialize our surface mesh collection. Here, we use the pull model API to get the current set of observed surfaces right away:

```

auto mapContainingSurfaceCollection = m_surfaceObserver->GetObservedSurfaces();
for (auto& pair : mapContainingSurfaceCollection)
{
    // Store the ID and metadata for each surface.
    auto const& id = pair->Key;
    auto const& surfaceInfo = pair->Value;
    m_meshCollection->AddOrUpdateSurface(id, surfaceInfo);
}

```

There is also a push model available to get surface mesh data. You are free to design your app to use only the pull model if you choose, in which case you'll poll for data every so often - say, once per frame - or during a specific time period, such as during game setup. If so, the above code is what you need.

In our code sample, we chose to demonstrate the use of both models for pedagogical purposes. Here, we subscribe to an event to receive up-to-date surface mesh data whenever the system recognizes a change.

```

m_surfaceObserver->ObservedSurfacesChanged += ref new TypedEventHandler<SpatialSurfaceObserver^,
Platform::Object^>(
    bind(&HolographicDesktopAppMain::OnSurfacesChanged, this, _1, _2)
);

```

Our code sample is also configured to respond to these events. Let's walk through how we do this.

NOTE: This might not be the most efficient way for your app to handle mesh data. This code is written for clarity and is not optimized.

The surface mesh data is provided in a read-only map that stores [SpatialSurfaceInfo](#) objects using [Platform::Guids](#) as key values.

```
IMapView<Guid, SpatialSurfaceInfo^>^ const& surfaceCollection = sender->GetObservedSurfaces();
```

To process this data, we look first for key values that aren't in our collection. Details on how the data is stored in our sample app will be presented later in this topic.

```

// Process surface adds and updates.
for (const auto& pair : surfaceCollection)
{
    auto id = pair->Key;
    auto surfaceInfo = pair->Value;

    if (m_meshCollection->HasSurface(id))
    {
        // Update existing surface.
        m_meshCollection->AddOrUpdateSurface(id, surfaceInfo);
    }
    else
    {
        // New surface.
        m_meshCollection->AddOrUpdateSurface(id, surfaceInfo);
    }
}

```

We also have to remove surface meshes that are in our surface mesh collection, but that aren't in the system collection anymore. To do so, we need to do something akin to the opposite of what we just showed for adding and updating meshes; we loop on our app's collection, and check to see if the **Guid** we have is in the system collection. If it's not in the system collection, we remove it from ours.

From our event handler in AppMain.cpp:

```
m_meshCollection->PruneMeshCollection(surfaceCollection);
```

The implementation of mesh pruning in `RealtimeSurfaceMeshRenderer.cpp`:

```
void RealtimeSurfaceMeshRenderer::PruneMeshCollection(IMapView<Guid, SpatialSurfaceInfo^>^ const&
surfaceCollection)
{
    std::lock_guard<std::mutex> guard(m_meshCollectionLock);
    std::vector<Guid> idsToRemove;

    // Remove surfaces that moved out of the culling frustum or no longer exist.
    for (const auto& pair : m_meshCollection)
    {
        const auto& id = pair.first;
        if (!surfaceCollection->HasKey(id))
        {
            idsToRemove.push_back(id);
        }
    }

    for (const auto& id : idsToRemove)
    {
        m_meshCollection.erase(id);
    }
}
```

Acquire and use surface mesh data buffers

Getting the surface mesh information was as easy as pulling a data collection and processing updates to that collection. Now, we'll go into detail on how you can use the data.

In our code example, we chose to use the surface meshes for rendering. This is a common scenario for occluding holograms behind real-world surfaces. You can also render the meshes, or render processed versions of them, to show the user what areas of the room are scanned before you start providing app or game functionality.

The code sample starts the process when it receives surface mesh updates from the event handler that we described in the previous section. The important line of code in this function is the call to update the surface *mesh*: by this time we have already processed the mesh info, and we are about to get the vertex and index data for use as we see fit.

From `RealtimeSurfaceMeshRenderer.cpp`:

```
void RealtimeSurfaceMeshRenderer::AddOrUpdateSurface(Guid id, SpatialSurfaceInfo^ newSurface)
{
    auto options = ref new SpatialSurfaceMeshOptions();
    options->IncludeVertexNormals = true;

    auto createMeshTask = create_task(newSurface->TryComputeLatestMeshAsync(1000, options));
    createMeshTask.then([this, id](SpatialSurfaceMesh^ mesh)
    {
        if (mesh != nullptr)
        {
            std::lock_guard<std::mutex> guard(m_meshCollectionLock);
            '''m_meshCollection[id].UpdateSurface(mesh);'''
        }
    }, task_continuation_context::use_current());
}
```

Our sample code is designed so that a data class, **SurfaceMesh**, handles mesh data processing and rendering. These meshes are what the **RealtimeSurfaceMeshRenderer** actually keeps a map of. Each one has a reference

to the SpatialSurfaceMesh it came from, and we use it any time that we need to access the mesh vertex or index buffers, or get a transform for the mesh. For now, we flag the mesh as needing an update.

From SurfaceMesh.cpp:

```
void SurfaceMesh::UpdateSurface(SpatialSurfaceMesh^ surfaceMesh)
{
    m_surfaceMesh = surfaceMesh;
    m_updateNeeded = true;
}
```

Next time the mesh is asked to draw itself, it will check the flag first. If an update is needed, the vertex and index buffers will be updated on the GPU.

```
void SurfaceMesh::CreateDeviceDependentResources(ID3D11Device* device)
{
    m_indexCount = m_surfaceMesh->TriangleIndices->ElementCount;
    if (m_indexCount < 3)
    {
        // Not enough indices to draw a triangle.
        return;
    }
}
```

First, we acquire the raw data buffers:

```
Windows::Storage::Streams::IBuffer^ positions = m_surfaceMesh->VertexPositions->Data;
Windows::Storage::Streams::IBuffer^ normals   = m_surfaceMesh->VertexNormals->Data;
Windows::Storage::Streams::IBuffer^ indices   = m_surfaceMesh->TriangleIndices->Data;
```

Then, we create Direct3D device buffers with the mesh data provided by the HoloLens:

```
CreateDirectXBuffer(device, D3D11_BIND_VERTEX_BUFFER, positions, m_vertexPositions.GetAddressOf());
CreateDirectXBuffer(device, D3D11_BIND_VERTEX_BUFFER, normals,   m_vertexNormals.GetAddressOf());
CreateDirectXBuffer(device, D3D11_BIND_INDEX_BUFFER,  indices,   m_triangleIndices.GetAddressOf());

// Create a constant buffer to control mesh position.
CD3D11_BUFFER_DESC constantBufferDesc(sizeof(SurfaceTransforms), D3D11_BIND_CONSTANT_BUFFER);
DX::ThrowIfFailed(
    device->CreateBuffer(
        &constantBufferDesc,
        nullptr,
        &m_modelTransformBuffer
    )
);

m_loadingComplete = true;
}
```

NOTE: For the CreateDirectXBuffer helper function used in the previous snippet, see the Surface Mapping code sample: SurfaceMesh.cpp, GetDataFromIBuffer.h. Now the device resource creation is complete, and the mesh is considered to be loaded and ready for update and render.

Update and render surface meshes

Our SurfaceMesh class has a specialized update function. Each [SpatialSurfaceMesh](#) has its own transform, and our sample uses the current coordinate system for our **SpatialStationaryReferenceFrame** to acquire the transform. Then it updates the model constant buffer on the GPU.

```

void SurfaceMesh::UpdateTransform(
    ID3D11DeviceContext* context,
    SpatialCoordinateSystem^ baseCoordinateSystem
)
{
    if (_indexCount < 3)
    {
        // Not enough indices to draw a triangle.
        return;
    }

    XMATRIX transform = XMMatrixIdentity();

    auto tryTransform = m_surfaceMesh->CoordinateSystem->TryGetTransformTo(baseCoordinateSystem);
    if (tryTransform != nullptr)
    {
        transform = XMLoadFloat4x4(&tryTransform->Value);
    }

    XMATRIX scaleTransform = XMMatrixScalingFromVector(XMLoadFloat3(&m_surfaceMesh->VertexPositionScale));

    XMStoreFloat4x4(
        &m_constantBufferData.vertexWorldTransform,
        XMMatrixTranspose(
            scaleTransform * transform
        )
    );
}

// Normals don't need to be translated.
XMATRIX normalTransform = transform;
normalTransform.r[3] = XMVectorSet(0.f, 0.f, 0.f, XMVectorGetW(normalTransform.r[3]));
XMStoreFloat4x4(
    &m_constantBufferData.normalWorldTransform,
    XMMatrixTranspose(
        normalTransform
    )
);

if (!m_loadingComplete)
{
    return;
}

context->UpdateSubresource(
    m_modelTransformBuffer.Get(),
    0,
    NULL,
    &m_constantBufferData,
    0,
    0
);
}

```

When it's time to render surface meshes, we do some prep work before rendering the collection. We set up the shader pipeline for the current rendering configuration, and we set up the input assembler stage. Note that the holographic camera helper class **CameraResources.cpp** already has set up the view/projection constant buffer by now.

From **RealtimeSurfaceMeshRenderer::Render**:

```

auto context = m_deviceResources->GetD3DDeviceContext();

context->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
context->IASetInputLayout(m_inputLayout.Get());

// Attach our vertex shader.
context->VSSetShader(
    m_vertexShader.Get(),
    nullptr,
    0
);

// The constant buffer is per-mesh, and will be set as such.

if (depthOnly)
{
    // Explicitly detach the later shader stages.
    context->GSSetShader(nullptr, nullptr, 0);
    context->PSSetShader(nullptr, nullptr, 0);
}
else
{
    if (!m_usingVprtShaders)
    {
        // Attach the passthrough geometry shader.
        context->GSSetShader(
            m_geometryShader.Get(),
            nullptr,
            0
        );
    }

    // Attach our pixel shader.
    context->PSSetShader(
        m_pixelShader.Get(),
        nullptr,
        0
    );
}

```

Once this is done, we loop on our meshes and tell each one to draw itself. **NOTE:** This sample code is not optimized to use any sort of frustum culling, but you should include this feature in your app.

```

std::lock_guard<std::mutex> guard(m_meshCollectionLock);

auto device = m_deviceResources->GetD3DDevice();

// Draw the meshes.
for (auto& pair : m_meshCollection)
{
    auto& id = pair.first;
    auto& surfaceMesh = pair.second;

    surfaceMesh.Draw(device, context, m_usingVprtShaders, isStereo);
}

```

The individual meshes are responsible for setting up the vertex and index buffer, stride, and model transform constant buffer. As with the spinning cube in the Windows Holographic app template, we render to stereoscopic buffers using instancing.

From **SurfaceMesh::Draw:**

```

// The vertices are provided in {vertex, normal} format

const auto& vertexStride = m_surfaceMesh->VertexPositions->Stride;
const auto& normalStride = m_surfaceMesh->VertexNormals->Stride;

UINT strides [] = { vertexStride, normalStride };
UINT offsets [] = { 0, 0 };
ID3D11Buffer* buffers [] = { m_vertexPositions.Get(), m_vertexNormals.Get() };

context->IASetVertexBuffers(
    0,
    ARRAYSIZE(buffers),
    buffers,
    strides,
    offsets
);

const auto& indexFormat = static_cast<DXGI_FORMAT>(m_surfaceMesh->TriangleIndices->Format);

context->IASetIndexBuffer(
    m_triangleIndices.Get(),
    indexFormat,
    0
);

context->VSSetConstantBuffers(
    0,
    1,
    m_modelTransformBuffer.GetAddressOf()
);

if (!usingVprtShaders)
{
    context->GSSetConstantBuffers(
        0,
        1,
        m_modelTransformBuffer.GetAddressOf()
    );
}

context->PSSetConstantBuffers(
    0,
    1,
    m_modelTransformBuffer.GetAddressOf()
);

context->DrawIndexedInstanced(
    m_indexCount,           // Index count per instance.
    isStereo ? 2 : 1,       // Instance count.
    0,                      // Start index location.
    0,                      // Base vertex location.
    0                       // Start instance location.
);

```

Rendering choices with Surface Mapping

The Surface Mapping code sample offers code for occlusion-only rendering of surface mesh data, and for on-screen rendering of surface mesh data. Which path you choose - or both - depends on your application. We'll walk through both configurations in this document.

Rendering occlusion buffers for holographic effect

Start by clearing the render target view for the current virtual camera.

From AppMain.cpp:

```
context->ClearRenderTargetView(pCameraResources->GetBackBufferRenderTargetView(),
DirectX::Colors::Transparent);
```

This is a "pre-rendering" pass. Here, we create an occlusion buffer by asking the mesh renderer to render only depth. In this configuration, we don't attach a render target view, and the mesh renderer sets the pixel shader stage to **nullptr** so that the GPU doesn't bother to draw pixels. The geometry will be rasterized to the depth buffer, and the graphics pipeline will stop there.

```
// Pre-pass rendering: Create occlusion buffer from Surface Mapping data.
context->ClearDepthStencilView(pCameraResources->GetSurfaceDepthStencilView(), D3D11_CLEAR_DEPTH | D3D11_CLEAR_STENCIL, 1.0f, 0);

// Set the render target to null, and set the depth target occlusion buffer.
// We will use this same buffer as a shader resource when drawing holograms.
context->OMSetRenderTargets(0, nullptr, pCameraResources->GetSurfaceOcclusionDepthStencilView());

// The first pass is a depth-only pass that generates an occlusion buffer we can use to know which
// hologram pixels are hidden behind surfaces in the environment.
m_meshCollection->Render(pCameraResources->IsRenderingStereoscopic(), true);
```

We can draw holograms with an extra depth test against the Surface Mapping occlusion buffer. In this code sample, we render pixels on the cube a different color if they are behind a surface.

From AppMain.cpp:

```
// Hologram rendering pass: Draw holographic content.
context->ClearDepthStencilView(pCameraResources->GetHologramDepthStencilView(), D3D11_CLEAR_DEPTH | D3D11_CLEAR_STENCIL, 1.0f, 0);

// Set the render target, and set the depth target drawing buffer.
ID3D11RenderTargetView *const targets[1] = { pCameraResources->GetBackBufferRenderTargetView() };
context->OMSetRenderTargets(1, targets, pCameraResources->GetHologramDepthStencilView());

// Render the scene objects.
// In this example, we draw a special effect that uses the occlusion buffer we generated in the
// Pre-Pass step to render holograms using X-Ray Vision when they are behind physical objects.
m_xrayCubeRenderer->Render(
    pCameraResources->IsRenderingStereoscopic(),
    pCameraResources->GetSurfaceOcclusionShaderResourceView(),
    pCameraResources->GetHologramOcclusionShaderResourceView(),
    pCameraResources->GetDepthTextureSamplerState()
);
```

Based on code from SpecialEffectPixelShader.hlsl:

```

// Draw boundaries
min16int surfaceSum = GatherDepthLess(envDepthTex, uniSamp, input.pos.xy, pixelDepth, input.idx.x);

if (surfaceSum <= -maxSum)
{
    // The pixel and its neighbors are behind the surface.
    // Return the occluded 'X-ray' color.
    return min16float4(0.67f, 0.f, 0.f, 1.0f);
}
else if (surfaceSum < maxSum)
{
    // The pixel and its neighbors are a mix of in front of and behind the surface.
    // Return the silhouette edge color.
    return min16float4(1.f, 1.f, 1.f, 1.0f);
}
else
{
    // The pixel and its neighbors are all in front of the surface.
    // Return the color of the hologram.
    return min16float4(input.color, 1.0f);
}

```

Note: For our **GatherDepthLess** routine, see the Surface Mapping code sample: SpecialEffectPixelShader.hlsl.

Rendering surface mesh data to the display

We can also just draw the surface meshes to the stereo display buffers. We chose to draw full faces with lighting, but you're free to draw wireframe, process meshes before rendering, apply a texture map, and so on.

Here, our code sample tells the mesh renderer to draw the collection. This time we don't specify a depth-only pass, so it will attach a pixel shader and complete the rendering pipeline using the targets that we specified for the current virtual camera.

```

// SR mesh rendering pass: Draw SR mesh over the world.
context->ClearDepthStencilView(pCameraResources->GetSurfaceOcclusionDepthStencilView(), D3D11_CLEAR_DEPTH | D3D11_CLEAR_STENCIL, 1.0f, 0);

// Set the render target to the current holographic camera's back buffer, and set the depth buffer.
ID3D11RenderTargetView *const targets[1] = { pCameraResources->GetBackBufferRenderTargetView() };
context->OMSetRenderTargets(1, targets, pCameraResources->GetSurfaceDepthStencilView());

// This drawing pass renders the surface meshes to the stereoscopic display. The user will be
// able to see them while wearing the device.
m_meshCollection->Render(pCameraResources->IsRenderingStereoscopic(), false);

```

See also

- [Creating a holographic DirectX project](#)
- [Windows.Perception.Spatial API](#)

Shared spatial anchors in DirectX

11/6/2018 • 16 minutes to read • [Edit Online](#)

You can transfer spatial anchors between Windows Mixed Reality devices by using the [SpatialAnchorTransferManager](#). This API lets you bundle up an anchor with all the supporting sensor data needed to find that exact place in the world, and then import that bundle on another device. Once the app on the second device has imported that anchor, each app can render holograms using that shared spatial anchor's coordinate system, which will then appear in the same place in the real world.

Note that spatial anchors are not currently able to transfer between different device types, for example a HoloLens spatial anchor may not be locatable using an immersive headset.

Set up your app to use the spatialPerception capability

Your app must be granted permission to use the spatialPerception capability before it can use the [SpatialAnchorTransferManager](#). This is necessary because transferring a spatial anchor involves sharing sensor images gathered over time in vicinity of that anchor, which might include sensitive information.

Declare this capability in the package.appxmanifest file for your app. Here's an example:

```
<Capabilities>
    <uap2:Capability Name="spatialPerception" />
</Capabilities>
```

The capability comes from the **uap2** namespace. To get access to this namespace in your manifest, include it as an *xlmns* attribute in the `<Package>` element. Here's an example:

```
<Package
    xmlns="http://schemas.microsoft.com/appx/manifest/foundation/windows10"
    xmlns:mp="http://schemas.microsoft.com/appx/2014/phone/manifest"
    xmlns:uap="http://schemas.microsoft.com/appx/manifest/uap/windows10"
    xmlns:uap2="http://schemas.microsoft.com/appx/manifest/uap/windows10/2"
    IgnorableNamespaces="uap mp"
    >
```

NOTE: Your app will need to request the capability at runtime before it can access SpatialAnchor export/import APIs. See [RequestAccessAsync](#) in the examples below.

Serialize anchor data by exporting it with the SpatialAnchorTransferManager

A helper function is included in the code sample to export (serialize) [SpatialAnchor](#) data. This export API serializes all anchors in a collection of key-value pairs associating strings with anchors.

```

// ExportAnchorDataAsync: Exports a byte buffer containing all of the anchors in the given collection.
//
// This function will place data in a buffer using a std::vector<byte>. The data buffer contains one or
more
// Anchors if one or more Anchors were successfully imported; otherwise, it is not modified.
//
task<bool> SpatialAnchorImportExportHelper::ExportAnchorDataAsync(
    vector<byte>* anchorByteDataOut,
    IMap<String^, SpatialAnchor^>^ anchorsToExport
)
{

```

First, we need to set up the data stream. This will allow us to 1.) use TryExportAnchorsAsync to put the data in a buffer owned by the app, and 2.) read data from the exported byte buffer stream - which is a WinRT data stream - into our own memory buffer, which is a std::vector<byte>.

```

// Create a random access stream to process the anchor byte data.
InMemoryRandomAccessStream^ stream = ref new InMemoryRandomAccessStream();

// Get an output stream for the anchor byte stream.
IOutputStream^ outputStream = stream->GetOutputStreamAt(0);

```

We need to ask permission to access spatial data, including anchors that are exported by the system.

```

// Request access to spatial data.
auto accessRequestedTask =
create_task(SpatialAnchorTransferManager::RequestAccessAsync()).then([anchorsToExport, outputStream]
(SpatialPerceptionAccessStatus status)
{
    if (status == SpatialPerceptionAccessStatus::Allowed)
    {
        // Access is allowed.

        // Export the indicated set of anchors.
        return create_task(SpatialAnchorTransferManager::TryExportAnchorsAsync(
            anchorsToExport,
            outputStream
        ));
    }
    else
    {
        // Access is denied.

        return task_from_result<bool>(false);
    }
});

```

If we do get permission and anchors are exported, we can read the data stream. Here, we also show how to create the DataReader and InputStream we will use to read the data.

```

// Get the input stream for the anchor byte stream.
IInputStream^ inputStream = stream->GetInputStreamAt(0);
// Create a DataReader, to get bytes from the anchor byte stream.
DataReader^ reader = ref new DataReader(inputStream);
return accessRequestedTask.then([anchorByteDataOut, stream, reader](bool anchorsExported)
{
    if (anchorsExported)
    {
        // Get the size of the exported anchor byte stream.
        size_t bufferSize = static_cast<size_t>(stream->Size);

        // Resize the output buffer to accept the data from the stream.
        anchorByteDataOut->reserve(bufferSize);
        anchorByteDataOut->resize(bufferSize);

        // Read the exported anchor store into the stream.
        return create_task(reader->LoadAsync(bufferSize));
    }
    else
    {
        return task_from_result<size_t>(0);
    }
}

```

After we read bytes from the stream, we can save them to our own data buffer like so.

```

}).then([anchorByteDataOut, reader](size_t bytesRead)
{
    if (bytesRead > 0)
    {
        // Read the bytes from the stream, into our data output buffer.
        reader->ReadBytes(Platform::ArrayReference<byte>(&(*anchorByteDataOut)[0], bytesRead));

        return true;
    }
    else
    {
        return false;
    }
});
};
}

```

Deserialize anchor data by importing it into the system using the SpatialAnchorTransferManager

A helper function is included in the code sample to load previously exported data. This deserialization function provides a collection of key-value pairs, similar to what the SpatialAnchorStore provides - except that we got this data from another source, such as a network socket. You can process and reason about this data before storing it offline, using in-app memory, or (if applicable) your app's SpatialAnchorStore.

```

// ImportAnchorDataAsync: Imports anchors from a byte buffer that was previously exported.
//
// This function will import all anchors from a data buffer into an in-memory collection of key, value
// pairs that maps String objects to SpatialAnchor objects. The Spatial AnchorStore is not affected by
// this function unless you provide it as the target collection for import.
//
task<bool> SpatialAnchorImportExportHelper::ImportAnchorDataAsync(
    std::vector<byte>& anchorByteDataIn,
    IMap<String^, SpatialAnchor^>^ anchorMapOut
)
{

```

First, we need to create stream objects to access the anchor data. We will be writing the data from our buffer to a system buffer, so we will create a DataWriter that writes to an in-memory data stream in order to accomplish our goal of getting anchors from a byte buffer into the system as SpatialAnchors.

```
// Create a random access stream for the anchor data.  
InMemoryRandomAccessStream^ stream = ref new InMemoryRandomAccessStream();  
  
// Get an output stream for the anchor data.  
IOutputStream^ outputStream = stream->GetOutputStreamAt(0);  
  
// Create a writer, to put the bytes in the stream.  
DataWriter^ writer = ref new DataWriter(outputStream);
```

Once again, we need to ensure the app has permission to export spatial anchor data, which could include private information about the user's environment.

```
// Request access to transfer spatial anchors.  
return create_task(SpatialAnchorTransferManager::RequestAccessAsync()).then(  
    [&anchorByteDataIn, writer](SpatialPerceptionAccessStatus status)  
{  
    if (status == SpatialPerceptionAccessStatus::Allowed)  
    {  
        // Access is allowed.
```

If access is allowed, we can write bytes from the buffer to a system data stream.

```
// Write the bytes to the stream.  
byte* anchorDataFirst = &anchorByteDataIn[0];  
size_t anchorContentSize = anchorByteDataIn.size();  
writer->WriteBytes(Platform::ArrayReference<byte>(anchorDataFirst, anchorContentSize));  
  
// Store the stream.  
return create_task(writer->StoreAsync());  
}  
else  
{  
    // Access is denied.  
    return task_from_result<size_t>(0);  
}
```

If we were successful in storing bytes in the data stream, we can try to import that data using the SpatialAnchorTransferManager.

```
}).then([writer, stream](unsigned int bytesWritten)
{
    if (bytesWritten > 0)
    {
        // Try to import anchors from the byte stream.
        return create_task(writer->FlushAsync())
            .then([stream](bool dataWasFlushed)
{
            if (dataWasFlushed)
            {
                // Get the input stream for the anchor data.
                IIInputStream^ inputStream = stream->GetInputStreamAt(0);

                return create_task(SpatialAnchorTransferManager::TryImportAnchorsAsync(inputStream));
            }
            else
            {
                return task_from_result<IMapView<String^, SpatialAnchor^>>(nullptr);
            }
        });
    }
    else
    {
        return task_from_result<IMapView<String^, SpatialAnchor^>>(nullptr);
    }
}
```

If the data is able to be imported, we get a map view of key-value pairs associating strings with anchors. We can load this into our own in-memory data collection, and use that collection to look for anchors that we are interested in using.

```

}).then([anchorMapOut](task<Windows::Foundation::Collections::IMMapView<String^, SpatialAnchor^>>
previousTask)
{
    try
    {
        auto importedAnchorsMap = previousTask.get();

        // If the operation was successful, we get a set of imported anchors.
        if (importedAnchorsMap != nullptr)
        {
            for each (auto& pair in importedAnchorsMap)
            {
                // Note that you could look for specific anchors here, if you know their key values.

                auto const& id = pair->Key;
                auto const& anchor = pair->Value;

                // Append "Remote" to the end of the anchor name for disambiguation.
                std::wstring idRemote(id->Data());
                idRemote += L"Remote";
                String^ idRemoteConst = ref new String (idRemote.c_str());

                // Store the anchor in the current in-memory anchor map.
                anchorMapOut->Insert(idRemoteConst, anchor);
            }

            return true;
        }
    }
    catch (Exception^ exception)
    {
        OutputDebugString(L"Error: Unable to import the anchor data buffer bytes into the in-memory anchor
collection.\n");
    }

    return false;
});
}

```

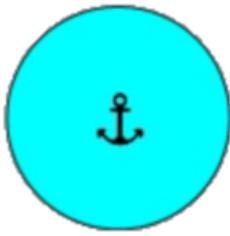
NOTE: Just because you can import an anchor, doesn't necessarily mean that you can use it right away. The anchor might be in a different room, or another physical location entirely; the anchor won't be locatable until the device that received it has enough visual information about the environment the anchor was created in, to restore the anchor's position relative to the known current environment. The client implementation should try locating the anchor relative to your local coordinate system or reference frame before continuing on to try to use it for live content. For example, try locating the anchor relative to a current coordinate system periodically until the anchor begins to be locatable.

Special Considerations

The [TryExportAnchorsAsync](#) API allows multiple [SpatialAnchors](#) to be exported into the same opaque binary blob. However, there is a subtle difference in what data the blob will include, depending on whether a single [SpatialAnchor](#) or multiple [SpatialAnchors](#) are exported in a single call.

Export of a single [SpatialAnchor](#)

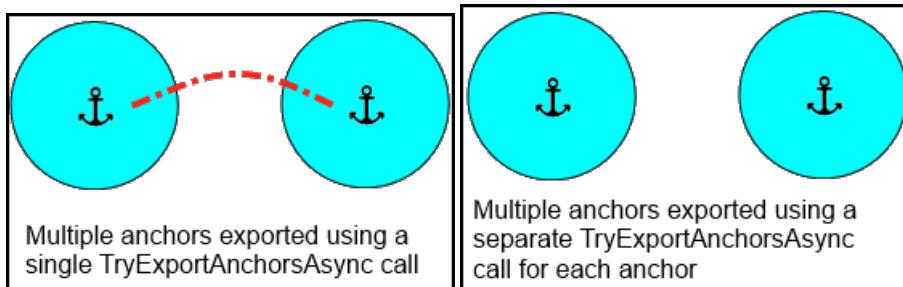
The blob contains a representation of the environment in the vicinity of the [SpatialAnchor](#) so that the environment can be recognized on the device that imports the [SpatialAnchor](#). After the import completes, the new [SpatialAnchor](#) will be available to the device. Assuming the user has recently been in vicinity of the anchor, it will be locatable and holograms attached to the [SpatialAnchor](#) can be rendered. These holograms will show up in the same physical location that they did on the original device which exported the [SpatialAnchor](#).



Export of a single SpatialAnchor

Export of multiple SpatialAnchors

Like the export of a single SpatialAnchor, the blob contains a representation of the environment in the vicinity of all the specified SpatialAnchors. In addition, the blob contains information about the connections between the included SpatialAnchors, if they are located in the same physical space. This means that if two nearby SpatialAnchors are imported, then a hologram attached to the *second* SpatialAnchor would be locatable even if the device only recognizes the environment around the *first* SpatialAnchor, because enough data to compute transform between the two SpatialAnchors was included in the blob. If the two SpatialAnchors were exported individually (two separate calls to TryExportSpatialAnchors) then there may not be enough data included in the blob for holograms attached to the second SpatialAnchor to be locatable when the first one is located.



Example: Send anchor data using a Windows::Networking::StreamSocket

Here, we provide an example of how to use exported anchor data by sending it across a TCP network. This is from the HolographicSpatialAnchorTransferSample.

The WinRT StreamSocket class uses the PPL task library. In the case of network errors, the error is returned to the next task in the chain using an exception that is re-thrown. The exception contains an HRESULT indicating the error status.

Use a Windows::Networking::StreamSocketListener with TCP to send exported anchor data

Create a server instance that listens for a connection.

```

void SampleAnchorTcpServer::ListenForConnection()
{
    // Make a local copy to avoid races with Closed events.
    StreamSocketListener^ streamSocketListener = m_socketServer;

    if (streamSocketListener == nullptr)
    {
        OutputDebugString(L"Server listening for client.\n");

        // Create the web socket connection.
        streamSocketListener = ref new StreamSocketListener();
        streamSocketListener->Control->KeepAlive = true;
        streamSocketListener->BindEndpointAsync(
            SampleAnchorTcpCommon::m_serverHost,
            SampleAnchorTcpCommon::m_tcpPort
        );

        streamSocketListener->ConnectionReceived +=
            ref new Windows::Foundation::TypedEventHandler<StreamSocketListener^,
            StreamSocketListenerConnectionReceivedEventArgs^>(
                std::bind(&SampleAnchorTcpServer::OnConnectionReceived, this, _1, _2)
            );
    }

    m_socketServer = streamSocketListener;
}
else
{
    OutputDebugString(L"Error: Stream socket listener not created.\n");
}
}

```

When a connection is received, use the client socket connection to send anchor data.

```

void SampleAnchorTcpServer::OnConnectionReceived(StreamSocketListener^ listener,
StreamSocketListenerConnectionReceivedEventArgs^ args)
{
    m_socketForClient = args->Socket;

    if (m_socketForClient != nullptr)
    {
        // In this example, when the client first connects, we catch it up to the current state of our
        // anchor set.
        OutputToClientSocket(m_spatialAnchorHelper->GetAnchorMap());
    }
}

```

Now, we can begin to send a data stream that contains the exported anchor data.

```

void SampleAnchorTcpServer::OutputToClientSocket(IMap<String^, SpatialAnchor^>^ anchorsToSend)
{
    m_anchorTcpSocketStreamWriter = ref new DataWriter(m_socketForClient->OutputStream);

    OutputDebugString(L"Sending stream to client.\n");
    SendAnchorDataStream(anchorsToSend).then([this](task<bool> previousTask)
    {
        try
        {
            bool success = previousTask.get();
            if (success)
            {
                OutputDebugString(L"Anchor data sent!\n");
            }
            else
            {
                OutputDebugString(L"Error: Anchor data not sent.\n");
            }
        }
        catch (Exception^ exception)
        {
            HandleException(exception);
            OutputDebugString(L"Error: Anchor data was not sent.\n");
        }
    });
}

```

Before we can send the stream itself, we must first send a header packet. This header packet must be of fixed length, and it must also indicate the length of the variable array of bytes that is the anchor data stream; in the case of this example we have no other header data to send, so our header is 4 bytes long and contains a 32-bit unsigned integer.

```

Concurrency::task<bool> SampleAnchorTcpServer::SendAnchorDataLengthMessage(size_t dataStreamLength)
{
    unsigned int arrayLength = dataStreamLength;
    byte* data = reinterpret_cast<byte*>(&arrayLength);

    m_anchorTcpSocketStreamWriter->WriteBytes(Platform::ArrayReference<byte>(data,
SampleAnchorTcpCommon::c_streamHeaderByteArrayLength));
    return create_task(m_anchorTcpSocketStreamWriter->StoreAsync()).then([this](unsigned int bytesStored)
{
    if (bytesStored > 0)
    {
        OutputDebugString(L"Anchor data length stored in stream; Flushing stream.\n");
        return create_task(m_anchorTcpSocketStreamWriter->FlushAsync());
    }
    else
    {
        OutputDebugString(L"Error: Anchor data length not stored in stream.\n");
        return task_from_result<bool>(false);
    }
});
}

Concurrency::task<bool> SampleAnchorTcpServer::SendAnchorDataStream(IMap<String^, SpatialAnchor^>^
anchorsToSend)
{
    return SpatialAnchorImportExportHelper::ExportAnchorDataAsync(
        &m_exportedAnchorStoreBytes,
        anchorsToSend
    ).then([this](bool anchorDataExported)
{
    if (anchorDataExported)
    {
        const size_t arrayLength = m_exportedAnchorStoreBytes.size();
        if (arrayLength > 0)
        {
            OutputDebugString(L"Anchor data was exported; sending data stream length message.\n");
            return SendAnchorDataLengthMessage(arrayLength);
        }
    }
    OutputDebugString(L"Error: Anchor data was not exported.\n");

    // No data to send.
    return task_from_result<bool>(false);
}

```

Once the stream length, in bytes, has been sent to the client, we can proceed to write the data stream itself to the socket stream. This will cause the anchor store bytes to get sent to the client.

```

}).then([this](bool dataLengthSent)
{
    if (dataLengthSent)
    {
        OutputDebugString(L"Data stream length message sent; writing exported anchor store bytes to
stream.\n");
        m_anchorTcpSocketStreamWriter->WriteBytes(Platform::ArrayReference<byte>
(&m_exportedAnchorStoreBytes[0], m_exportedAnchorStoreBytes.size()));
        return create_task(m_anchorTcpSocketStreamWriter->StoreAsync());
    }
    else
    {
        OutputDebugString(L"Error: Data stream length message not sent.\n");
        return task_from_result<size_t>(0);
    }
}

).then([this](unsigned int bytesStored)
{
    if (bytesStored > 0)
    {
        PrintWstringToDebugConsole(
            std::to_wstring(bytesStored) +
            L" bytes of anchor data written and stored to stream; flushing stream.\n"
        );
    }
    else
    {
        OutputDebugString(L"Error: No anchor data bytes were written to the stream.\n");
    }

    return task_from_result<bool>(false);
});
}

```

As noted earlier in this topic, we must be prepared to handle exceptions containing network error status messages. For errors that are not expected, we can write the exception info to the debug console like so. This will give us a clue as to what happened if our code sample is unable to complete the connection, or if it is unable to finish sending the anchor data.

```

void SampleAnchorTcpServer::HandleException(Exception^ exception)
{
    PrintWstringToDebugConsole(
        std::wstring(L"Connection error: ") +
        exception->ToString()->Data() +
        L"\n"
    );
}

```

Use a Windows::Networking::StreamSocket with TCP to receive exported anchor data

First, we have to connect to the server. This code sample shows how to create and configure a StreamSocket, and create a DataReader that you can use to acquire network data using the socket connection.

NOTE: If you run this sample code, ensure that you configure and launch the server before starting the client.

```

task<bool> SampleAnchorTcpClient::ConnectToServer()
{
    // Make a local copy to avoid races with Closed events.
    StreamSocket^ streamSocket = m_socketClient;

    // Have we connected yet?
    if (m_socketClient == nullptr)
    {

```

```

        OutputDebugString(L"Client is attempting to connect to server.\n");

        EndpointPair^ endpointPair = ref new EndpointPair(
            SampleAnchorTcpCommon::m_clientHost,
            SampleAnchorTcpCommon::m_tcpPort,
            SampleAnchorTcpCommon::m_serverHost,
            SampleAnchorTcpCommon::m_tcpPort
        );

        // Create the web socket connection.
        m_socketClient = ref new StreamSocket();

        // The client connects to the server.
        return create_task(m_socketClient->ConnectAsync(endpointPair,
    SocketProtectionLevel::PlainSocket)).then([this](task<void> previousTask)
    {
        try
        {
            // Try getting all exceptions from the continuation chain above this point.
            previousTask.get();

            m_anchorTcpSocketStreamReader = ref new DataReader(m_socketClient->InputStream);

            OutputDebugString(L"Client connected!\n");

            m_anchorTcpSocketStreamReader->InputStreamOptions = InputStreamOptions::ReadAhead;

            WaitForAnchorDataStream();

            return true;
        }
        catch (Exception^ exception)
        {
            if (exception->HRESULT == 0x80072741)
            {
                // This code sample includes a very simple implementation of client/server
                // endpoint detection: if the current instance tries to connect to itself,
                // it is determined to be the server.
                OutputDebugString(L"Starting up the server instance.\n");

                // When we return false, we'll start up the server instead.
                return false;
            }
            else if ((exception->HRESULT == 0x8007274c) || // connection timed out
                (exception->HRESULT == 0x80072740)) // connection maxed at server end
            {
                // If the connection timed out, try again.
                ConnectToServer();
            }
            else if (exception->HRESULT == 0x80072741)
            {
                // No connection is possible.
            }

            HandleException(exception);
            return true;
        }
    });
});

else
{
    OutputDebugString(L"A StreamSocket connection to a server already exists.\n");
    return task_from_result<bool>(true);
}
}

```

Once we have a connection, we can wait for the server to send data. We do this by calling LoadAsync on the

stream data reader.

The first set of bytes we receive should always be the header packet, which indicates the anchor data stream byte length as described in the previous section.

```
void SampleAnchorTcpClient::WaitForAnchorDataStream()
{
    if (m_anchorTcpSocketStreamReader == nullptr)
    {
        // We have not connected yet.
        return;
    }

    OutputDebugString(L"Waiting for server message.\n");

    // Wait for the first message, which specifies the byte length of the string data.
    create_task(m_anchorTcpSocketStreamReader-
>LoadAsync(SampleAnchorTcpCommon::c_streamHeaderByteArrayLength)).then([this](unsigned int numberOfBytes)
    {
        if (numberOfBytes > 0)
        {
            OutputDebugString(L"Server message incoming.\n");
            return ReceiveAnchorDataLengthMessage();
        }
        else
        {
            OutputDebugString(L"0-byte async task received, awaiting server message again.\n");
            WaitForAnchorDataStream();
            return task_from_result<size_t>(0);
        }
    }
}
```

...

```
task<size_t> SampleAnchorTcpClient::ReceiveAnchorDataLengthMessage()
{
    byte data[4];
    m_anchorTcpSocketStreamReader->ReadBytes(Platform::ArrayReference<byte>(data,
SampleAnchorTcpCommon::c_streamHeaderByteArrayLength));
    unsigned int lengthMessageSize = *reinterpret_cast<unsigned int*>(data);

    if (lengthMessageSize > 0)
    {
        OutputDebugString(L"One or more anchors to be received.\n");
        return task_from_result<size_t>(lengthMessageSize);
    }
    else
    {
        OutputDebugString(L"No anchors to be received.\n");
        ConnectToServer();
    }

    return task_from_result<size_t>(0);
}
```

After we have received the header packet, we know how many bytes of anchor data we should expect. We can proceed to read those bytes from the stream.

```

}).then([this](size_t dataStreamLength)
{
    if (dataStreamLength > 0)
    {
        std::wstring debugMessage = std::to_wstring(dataStreamLength);
        debugMessage += L" bytes of anchor data incoming.\n";
        OutputDebugString(debugMessage.c_str());

        // Prepare to receive the data stream in one or more pieces.
        m_anchorStreamLength = dataStreamLength;
        m_exportedAnchorStoreBytes.clear();
        m_exportedAnchorStoreBytes.resize(m_anchorStreamLength);

        OutputDebugString(L"Loading byte stream.\n");
        return ReceiveAnchorDataStream();
    }
    else
    {
        OutputDebugString(L"Error: Anchor data size not received.\n");
        ConnectToServer();
        return task_from_result<bool>(false);
    }
});
}

```

Here's our code for receiving the anchor data stream. Again, we will first load the bytes from the stream; this operation may take some time to complete as the StreamSocket waits to receive that amount of bytes from the network.

When the loading operation is complete, we can read that number of bytes. If we received the number of bytes that we expect for the anchor data stream, we can go ahead and import the anchor data; if not, there must have been some sort of error. For example, this can happen when the server instance terminates before it can finish sending the data stream, or the network goes down before the entire data stream can be received by the client.

```

task<bool> SampleAnchorTcpClient::ReceiveAnchorDataStream()
{
    if (m_anchorStreamLength > 0)
    {
        // First, we load the bytes from the network socket.
        return create_task(m_anchorTcpSocketStreamReader->LoadAsync(m_anchorStreamLength)).then([this]
(size_t bytesLoadedByStreamReader)
    {
        if (bytesLoadedByStreamReader > 0)
        {
            // Once the bytes are loaded, we can read them from the stream.
            m_anchorTcpSocketStreamReader->ReadBytes(Platform::ArrayReference<byte>
(&m_exportedAnchorStoreBytes[0],
            bytesLoadedByStreamReader));

            // Check status.
            if (bytesLoadedByStreamReader == m_anchorStreamLength)
            {
                // The whole stream has arrived. We can process the data.

                // Informational message of progress complete.
                std::wstring infoMessage = std::to_wstring(bytesLoadedByStreamReader);
                infoMessage += L" bytes read out of ";
                infoMessage += std::to_wstring(m_anchorStreamLength);
                infoMessage += L" total bytes; importing the data.\n";
                OutputDebugStringW(infoMessage.c_str());

                // Kick off a thread to wait for a new message indicating another incoming anchor data
stream.
                WaitForAnchorDataStream();

                // Process the data for the stream we just received.
                return
SpatialAnchorImportExportHelper::ImportAnchorDataAsync(m_exportedAnchorStoreBytes, m_spatialAnchorHelper-
>GetAnchorMap());
            }
            else
            {
                OutputDebugString(L"Error: Fewer than expected anchor data bytes were received.\n");
            }
        }
        else
        {
            OutputDebugString(L"Error: No anchor bytes were received.\n");
        }

        return task_from_result<bool>(false);
    });
}

else
{
    OutputDebugString(L"Warning: A zero-length data buffer was sent.\n");
    return task_from_result<bool>(false);
}
}

```

Again, we must be prepared to handle unknown network errors.

```
void SampleAnchorTcpClient::HandleException(Exception^ exception)
{
    std::wstring error = L"Connection error: ";
    error += exception->ToString()->Data();
    error += L"\n";
    OutputDebugString(error.c_str());
}
```

That's it! Now, you should have enough information to try locating the anchors received over the network. Again, note that the client must have enough visual tracking data for the space to successfully locate the anchor; if it doesn't work right away, try walking around for a while. If it still doesn't work, have the server send more anchors, and use network communications to agree on one that works for the client. You can try this out by downloading the `HolographicSpatialAnchorTransferSample`, configuring your client and server IPs, and deploying it to client and server HoloLens devices.

See also

- [Parallel Patterns Library \(PPL\)](#)
- [Windows.Networking.StreamSocket](#)
- [Windows.Networking.StreamSocketListener](#)

Locatable camera in DirectX

11/6/2018 • 2 minutes to read • [Edit Online](#)

This topic describes how to set up a [Media Foundation](#) pipeline to access the [camera](#) in a DirectX app, including the frame metadata that will allow you to locate the images produced in the real world.

Windows Media Capture and Media Foundation Development: IMFAttributes

Each image frame [includes a coordinate system](#), as well as two important transforms. The "view" transform maps from the provided coordinate system to the camera, and the "projection" maps from the camera to pixels in the image. The coordinate system and the 2 transforms are embedded as metadata in every image frame via Media Foundation's [IMFAttributes](#).

Sample usage of reading attributes with MF custom sink and doing projection

In your custom MF Sink stream ([IMFStreamSink](#)), you will get [IMFSample](#) with sample attributes:

The following MediaExtensions must be defined for WinRT based code:

```
EXTERN_GUID(MFSampleExtension_Spatial_CameraViewTransform, 0x4e251fa4, 0x830f, 0x4770, 0x85, 0x9a, 0x4b, 0x8d,  
0x99, 0xaa, 0x80, 0x9b);  
EXTERN_GUID(MFSampleExtension_Spatial_CameraCoordinateSystem, 0x9d13c82f, 0x2199, 0x4e67, 0x91, 0xcd, 0xd1,  
0xa4, 0x18, 0x1f, 0x25, 0x34);  
EXTERN_GUID(MFSampleExtension_Spatial_CameraProjectionTransform, 0x47f9fcb5, 0x2a02, 0x4f26, 0xa4, 0x77, 0x79,  
0x2f, 0xdf, 0x95, 0x88, 0x6a);
```

You can't access these attributes from WinRT APIs, but requires Media Extension implementation of [IMFT Transform](#) (for Effect) or [IMFMediaSink](#) and [IMFStreamSink](#) (for Custom Sink). When you process the sample in this extension either in [IMFT Transform::ProcessInput\(\)](#)/[IMFT Transform::ProcessOutput\(\)](#) or [IMFStreamSink::ProcessSample\(\)](#), you can query attributes like this sample.

```

ComPtr<IUnknown> spUnknown;
ComPtr<Windows::Perception::Spatial::ISpatialCoordinateSystem> spSpatialCoordinateSystem;
ComPtr<Windows::Foundation::IReference<Windows::Foundation::Numerics::Matrix4x4>> spTransformRef;
Windows::Foundation::Numerics::Matrix4x4 worldToCameraMatrix;
Windows::Foundation::Numerics::Matrix4x4 viewMatrix;
Windows::Foundation::Numerics::Matrix4x4 projectionMatrix;
UINT32 cbBlobSize = 0;
hr = pSample->GetUnknown(MFSampleExtension_Spatial_CameraCoordinateSystem, IID_PPV_ARGS(&spUnknown));
if (SUCCEEDED(hr))
{
    hr = spUnknown.As(&spSpatialCoordinateSystem);
    if (FAILED(hr))
    {
        return hr;
    }
    hr = pSample->GetBlob(MFSampleExtension_Spatial_CameraViewTransform,
                           (UINT8*)(&viewMatrix),
                           sizeof(viewMatrix),
                           &cbBlobSize);
    if (SUCCEEDED(hr) && cbBlobSize == sizeof(Windows::Foundation::Numerics::Matrix4x4))
    {
        hr = pSample->GetBlob(MFSampleExtension_Spatial_CameraProjectionTransform,
                               (UINT8*)(&projectionMatrix),
                               sizeof(projectionMatrix),
                               &cbBlobSize);
        if (SUCCEEDED(hr) && cbBlobSize == sizeof(Windows::Foundation::Numerics::Matrix4x4))
        {
            // Assume m_spCurrentCoordinateSystem is representing
            // world coordinate system application is using
            hr = m_spCurrentCoordinateSystem->TryGetTransformTo(spSpatialCoordinateSystem.Get(),
                                                               &spTransformRef);
            if (FAILED(hr))
            {
                return hr;
            }
            hr = spTransformRef->get_Value(&worldToCameraMatrix);
            if (FAILED(hr))
            {
                return hr;
            }
        }
    }
}
}

```

To access the texture from the camera, you need same D3D device which creates camera frame texture. This D3D device is in [IMFDXGIDeviceManager](#) in the capture pipeline. To get the DXGI Device manager from Media Capture you can use [IAdvancedMediaCapture](#) and [IAdvancedMediaCaptureSettings](#) interfaces.

```

Microsoft::WRL::ComPtr<IAdvancedMediaCapture> spAdvancedMediaCapture;
Microsoft::WRL::ComPtr<IAdvancedMediaCaptureSettings> spAdvancedMediaCaptureSettings;
Microsoft::WRL::ComPtr<IMFDXGIDeviceManager> spDXGIDeviceManager;
// Assume mediaCapture is Windows::Media::Capture::MediaCapture and initialized
if (SUCCEEDED((IUnknown *)(mediaCapture))->QueryInterface(IID_PPV_ARGS(&spAdvancedMediaCapture)))
{
    if (SUCCEEDED(spAdvancedMediaCapture->GetAdvancedMediaCaptureSettings(&spAdvancedMediaCaptureSettings)))
    {
        spAdvancedMediaCaptureSettings->GetDirectXDeviceManager(&spDXGIDeviceManager);
    }
}

```

You can also enable mouse and keyboard input as optional input methods for your Windows Mixed Reality app. This can also be a great debugging feature for devices such as HoloLens, and may be desirable for user input in mixed reality apps running in immersive headsets attached to PCs.

Keyboard, mouse, and controller input in DirectX

11/6/2018 • 6 minutes to read • [Edit Online](#)

Keyboards, mice, and game controllers can all be useful forms of input for Windows Mixed Reality devices.

Bluetooth keyboards and mice are both supported on HoloLens, for use with debugging your app or as an alternate form of input. Windows Mixed Reality also supports immersive headsets attached to PCs - where mice, keyboards, and game controllers have historically been the norm.

To use keyboard input on HoloLens, pair a Bluetooth keyboard to your device or use virtual input via the Windows Device Portal. To use keyboard input while wearing a Windows Mixed Reality immersive headset, assign input focus to mixed reality by putting on the device or using the Windows Key + Y keyboard combination. Keep in mind that apps intended for HoloLens must provide functionality without these devices attached.

Subscribe for CoreWindow input events

Keyboard input

In the Windows Holographic app template, we include an event handler for keyboard input just like any other UWP app. Your app consumes keyboard input data the same way in Windows Mixed Reality.

From AppView.cpp:

```
// Register for keypress notifications.  
window->KeyDown +=  
    ref new TypedEventHandler<CoreWindow^, KeyEventArgs^>(this, &AppView::OnKeyPressed);  
  
...  
  
// Input event handlers  
  
void AppView::OnKeyPressed(CoreWindow^ sender, KeyEventArgs^ args)  
{  
    //  
    // TODO: Respond to keyboard input here.  
    //  
}
```

Virtual keyboard input

For immersive desktop headsets, you can also support virtual keyboards rendered by Windows over your immersive view. To support this, your app can implement **CoreTextEditContext**. This lets Windows understand the state of your own app-rendered text boxes, so the virtual keyboard can correctly contribute to the text there.

For more information on implementing CoreTextEditContext support, see the [CoreTextEditContext sample](#).

Mouse Input

You can also use mouse input, again via the UWP CoreWindow input event handlers. Here's how to modify the Windows Holographic app template to support mouse clicks in the same way as pressed gestures. After making this modification, a mouse click while wearing an immersive headset device will reposition the cube.

Note that UWP apps can also get raw XY data for the mouse by using the [MouseDevice API](#).

Start by declaring a new OnPointerPressed handler in AppView.h:

```
protected:  
    void OnPointerPressed(Windows::UI::Core::CoreWindow^ sender, Windows::UI::Core::PointerEventArgs^  
args);
```

In AppView.cpp, add this code to SetWindow:

```
// Register for pointer pressed notifications.  
window->PointerPressed +=  
    ref new TypedEventHandler<CoreWindow^, PointerEventArgs^>(this, &AppView::OnPointerPressed);
```

Then put this definition for OnPointerPressed at the bottom of the file:

```
void AppView::OnPointerPressed(CoreWindow^ sender, PointerEventArgs^ args)  
{  
    // Allow the user to interact with the holographic world using the mouse.  
    if (m_main != nullptr)  
    {  
        m_main->OnPointerPressed();  
    }  
}
```

The event handler we just added is a pass-through to the template main class. Let's modify the main class to support this pass-through. Add this public method declaration to the header file:

```
// Handle mouse input.  
void OnPointerPressed();
```

You'll need this private member variable, as well:

```
// Keep track of mouse input.  
bool m_pointerPressed = false;
```

Finally, we will update the main class with new logic to support mouse clicks. Start by adding this event handler. Make sure to update the class name:

```
void MyHolographicAppMain::OnPointerPressed()  
{  
    m_pointerPressed = true;  
}
```

Now, in the Update method, replace the existing logic for getting a pointer pose with this:

```
SpatialInteractionSourceState^ pointerState = m_spatialInputHandler->CheckForInput();  
SpatialPointerPose^ pose = nullptr;  
if (pointerState != nullptr)  
{  
    pose = pointerState->TryGetPointerPose(currentCoordinateSystem);  
}  
else if (m_pointerPressed)  
{  
    pose = SpatialPointerPose::TryGetAtTimestamp(currentCoordinateSystem, prediction->Timestamp);  
}  
m_pointerPressed = false;
```

Recompile and redeploy. You should notice that the mouse click will now reposition the cube in your immersive

headset - or HoloLens with bluetooth mouse attached.

Game controller support

Game controllers can be a fun and convenient way of allowing the user to control an immersive Windows Mixed Reality experience.

The first step in adding support for game controllers to the Windows Holographic app template, is to add the following private member declarations to the header class for your main file:

```
// Recognize gamepads that are plugged in after the app starts.  
void OnGamepadAdded(Platform::Object^, Windows::Gaming::Input::Gamepad^ args);
```

```
// Stop looking for gamepads that are unplugged.  
void OnGamepadRemoved(Platform::Object^, Windows::Gaming::Input::Gamepad^ args);
```

```
Windows::Foundation::EventRegistrationToken  
Windows::Foundation::EventRegistrationToken  
m_gamepadAddedEventToken;  
m_gamepadRemovedEventToken;
```

```
// Keeps track of a gamepad and the state of its A button.  
struct GamepadWithButtonState  
{  
    Windows::Gaming::Input::Gamepad^ gamepad;  
    bool buttonAWasPressedLastFrame = false;  
};  
std::vector<GamepadWithButtonState>  
m_gamepads;
```

Initialize gamepad events, and any gamepads that are currently attached, in the constructor for your main class:

```
// If connected, a game controller can also be used for input.  
m_gamepadAddedEventToken = Gamepad::GamepadAdded +=  
ref new EventHandler<Gamepad^>(  
    bind(&$safeprjectname$Main::OnGamepadAdded, this, _1, _2)  
>;
```

```
m_gamepadRemovedEventToken = Gamepad::GamepadRemoved +=  
ref new EventHandler<Gamepad^>(  
    bind(&$safeprjectname$Main::OnGamepadRemoved, this, _1, _2)  
>;
```

```
for (auto const& gamepad : Gamepad::Gamepads)  
{  
    OnGamepadAdded(nullptr, gamepad);  
}
```

Add these event handlers to your main class. Make sure to update the class name:

```

void MyHolographicAppMain::OnGamepadAdded(Object^, Gamepad^ args)
{
    for (auto const& gamepadWithButtonState : m_gamepads)
    {
        if (args == gamepadWithButtonState.gamepad)
        {
            // This gamepad is already in the list.
            return;
        }
    }
    m_gamepads.push_back({ args, false });
}

```

```

void MyHolographicAppMain::OnGamepadRemoved(Object^, Gamepad^ args)
{
    m_gamepads.erase(
        std::remove_if(m_gamepads.begin(), m_gamepads.end(), [&](GamepadWithButtonState& gamepadWithState)
        {
            return gamepadWithState.gamepad == args;
        }),
        m_gamepads.end());
}

```

Finally, update the input logic to recognize changes in controller state. Here, we use the same `m_pointerPressed` variable discussed in the section above for adding mouse events. Add this to the `Update` method, just before where it checks for the `SpatialPointerPose`:

```

// Check for new input state since the last frame.
for (auto& gamepadWithButtonState : m_gamepads)
{
    bool buttonDownThisUpdate = ((gamepadWithButtonState.gamepad->GetCurrentReading().Buttons &
GamepadButtons::A) == GamepadButtons::A);
    if (buttonDownThisUpdate && !gamepadWithButtonState.buttonAWasPressedLastFrame)
    {
        m_pointerPressed = true;
    }
    gamepadWithButtonState.buttonAWasPressedLastFrame = buttonDownThisUpdate;
}

```

```

// For context.
SpatialInteractionSourceState^ pointerState = m_spatialInputHandler->CheckForInput();
SpatialPointerPose^ pose = nullptr;
if (pointerState != nullptr)
{
    pose = pointerState->TryGetPointerPose(currentCoordinateSystem);
}
else if (m_pointerPressed)
{
    pose = SpatialPointerPose::TryGetAtTimestamp(currentCoordinateSystem, prediction->Timestamp);
}
m_pointerPressed = false;

```

Don't forget to unregister the events when cleaning up the main class:

```
if (m_gamepadAddedEventToken.Value != 0)
{
    Gamepad::GamepadAdded -= m_gamepadAddedEventToken;
}
if (m_gamepadRemovedEventToken.Value != 0)
{
    Gamepad::GamepadRemoved -= m_gamepadRemovedEventToken;
}
```

Recompile, and redeploy. You can now attach, or pair, a game controller and use it to reposition the spinning cube.

Important guidelines for keyboard and mouse input

There are some key differences in how this code can be used on Microsoft HoloLens – which is a device that relies primarily on natural user input – versus what is available on a Windows Mixed Reality-enabled PC.

- You can't rely on keyboard or mouse input to be present. All of your app's functionality must work with gaze, gesture, and speech input.
- When a Bluetooth keyboard is attached, it can be helpful to enable keyboard input for any text that your app might ask for. This can be a great supplement for dictation, for example.
- When it comes to designing your app, don't rely on (for example) WASD and mouse look controls for your game. HoloLens is designed for the user to walk around the room. In this case, the user controls the camera directly. An interface for driving the camera around the room with move/look controls won't provide the same experience.
- Keyboard input can be an excellent way to control the debugging aspects of your app or game engine, especially since the user will not be required to use the keyboard. Wiring it up is the same as you're used to, with CoreWindow event APIs. In this scenario, you might choose to implement a way to configure your app to route keyboard events to a "debug input only" mode during your debug sessions.
- Bluetooth controllers work as well.

See also

- [Hardware accessories](#)

Using XAML with holographic DirectX apps

11/6/2018 • 2 minutes to read • [Edit Online](#)

This topic explains the impact of switching between [2D XAML views and immersive views](#) in your DirectX app, and how to make efficient use of both a XAML view and immersive view.

XAML view switching overview

On HoloLens, a generally immersive app that may later display a 2D XAML view must initialize that XAML view first and immediately switch to an immersive view from there. This means XAML will load before your app can do anything. As a result, there will be a small increase in your startup time, and XAML will continue to occupy memory space in your app process while it resides in the background; the amount of startup delay and memory usage depends on what your app does with XAML before switching to the native view. If you do nothing in your XAML start up code at first except start your immersive view, the impact should be minor. Also, because your holographic rendering is done directly to the immersive view, you will avoid any XAML-related restrictions on that rendering.

Note that the memory usage counts for both CPU and GPU. Direct3D 11 is able to swap virtual graphics memory, but it might not be able to swap out some or all of the XAML GPU resources, and there might be a noticeable performance hit. Either way, not loading any XAML features you don't need will leave more room for your app and provide a better experience.

XAML view switching workflow

The workflow for an app that goes directly from XAML to immersive mode is like so:

- The app starts up in the 2D XAML view.
- The app's XAML startup sequence detects if the current system supports holographic rendering:
- If so, the app creates the immersive view and brings it to the foreground right away. XAML loading is skipped for anything not necessary on Windows Mixed Reality devices, including any rendering classes and asset loading in the XAML view. If the app is using XAML for keyboard input, that input page should still be created.
- If not, the XAML view can proceed with business as usual.

Tip for rendering graphics across both views

If your app needs to implement some amount of rendering in DirectX for the XAML view in Windows Mixed Reality, your best bet is to create one renderer that can work with both views. The renderer should be one instance that can be accessed from both views, and it should be able to switch between 2D and holographic rendering. That way GPU assets are only loaded once - this reduces load times, memory impact, and the amount of resources to be swapped when switching views.

Add holographic remoting

11/6/2018 • 6 minutes to read • [Edit Online](#)

Add holographic remoting to your desktop or UWP app

This page describes how to add Holographic Remoting to a desktop or UWP app.

Holographic remoting allows your app to target a HoloLens with holographic content hosted on a desktop PC or on a UWP device such as the Xbox One, allowing access to more system resources and making it possible to integrate remote [immersive views](#) into existing desktop PC software. A remoting host app receives an input data stream from a HoloLens, renders content in a virtual immersive view, and streams content frames back to HoloLens. The connection is made using standard Wi-Fi. To use remoting, you will use a NuGet package to add holographic remoting to your desktop or UWP app, and write code to handle the connection and to render in an immersive view. Helper libraries are included in the code sample that simplify the task of handling the device connection.

A typical remoting connection will have as low as 50 ms of latency. The player app can report the latency in real-time.

Get the remoting NuGet packages

Follow these steps to get the NuGet package for holographic remoting, and add a reference from your project:

1. Go to your project in Visual Studio.
2. Right-click on the project node and select **Manage NuGet Packages...**
3. In the panel that appears, click **Browse** and then search for "Holographic Remoting".
4. Select **Microsoft.Holographic.Remoting** and click **Install**.
5. If the **Preview** dialog appears, click **OK**.
6. The next dialog that appears is the license agreement. Click on **I Accept** to accept the license agreement.

Create the HolographicStreamerHelpers

First, we need an instance of HolographicStreamerHelpers. Add this to the class that will be handling remoting.

```
#include <HolographicStreamerHelpers.h>

private:
    Microsoft::Holographic::HolographicStreamerHelpers^ m_streamerHelpers;
```

You'll also need to track connection state. If you want to render the preview, you need to have a texture to copy it to. You also need a few things like a connection state lock, some way of storing the IP address of HoloLens, and so on.

```
private:
    Microsoft::Holographic::HolographicStreamerHelpers^ m_streamerHelpers;

    Microsoft::WRL::Wrappers::SRWLock           m_connectionStateLock;

    RemotingHostSample::AppView^                  m_appView;
    Platform::String^                           m_ipAddress;
    Microsoft::Holographic::HolographicStreamerHelpers^ m_streamerHelpers;

    Microsoft::WRL::Wrappers::CriticalSection   m_deviceLock;
    Microsoft::WRL::ComPtr<IDXGISwapChain1>     m_swapChain;
    Microsoft::WRL::ComPtr<ID3D11Texture2D>      m_spTexture;
```

Initialize HolographicStreamerHelpers and connect to HoloLens

To connect to a HoloLens device, create an instance of HolographicStreamerHelpers and connect to the target IP address. You will need to set the video frame size to match the HoloLens display width and height, because the Holographic Remoting library expects the encoder and decoder resolutions to match exactly.

```
m_streamerHelpers = ref new HolographicStreamerHelpers();
m_streamerHelpers->CreateStreamer(m_d3dDevice);

// We currently need to stream at 720p because that's the resolution of our remote display.
// There is a check in the holographic streamer that makes sure the remote and local
// resolutions match. The default streamer resolution is 1080p.
m_streamerHelpers->SetVideoFrameSize(1280, 720);

try
{
    m_streamerHelpers->Connect(m_ipAddress->Data(), 8001);
}
catch (Platform::Exception^ ex)
{
    DebugLog(L"Connect failed with hr = 0x%08X", ex->HResult);
}
```

The device connection is asynchronous. Your app needs to provide event handlers for connect, disconnect, and frame send events.

The OnConnected event can update the UI, start rendering, and so on. In our desktop code sample, we update the window title with a "connected" message.

```
m_streamerHelpers->OnConnected += ref new ConnectedEvent(
    [this]()
{
    UpdateWindowTitle();
});
```

The OnDisconnected event can handle reconnection, UI updates, and so on. In this example, we reconnect if there is a transient failure.

```

Platform::WeakReference streamerHelpersWeakRef = Platform::WeakReference(m_streamerHelpers);
m_streamerHelpers->OnDisconnected += ref new DisconnectedEvent(
    [this, streamerHelpersWeakRef](_In_ HolographicStreamerConnectionFailureReason failureReason)
{
    DebugLog(L"Disconnected with reason %d", failureReason);
    UpdateWindowTitle();

    // Reconnect if this is a transient failure.
    if (failureReason == HolographicStreamerConnectionFailureReason::Unreachable ||
        failureReason == HolographicStreamerConnectionFailureReason::ConnectionLost)
    {
        DebugLog(L"Reconnecting...");
    }

    try
    {
        auto helpersResolved = streamerHelpersWeakRef.Resolve<HolographicStreamerHelpers>();
        if (helpersResolved)
        {
            helpersResolved->Connect(m_ipAddress->Data(), 8001);
        }
        else
        {
            DebugLog(L"Failed to reconnect because a disconnect has already occurred.\n");
        }
    }
    catch (Platform::Exception^ ex)
    {
        DebugLog(L"Connect failed with hr = 0x%08X", ex->HRESULT);
    }
}
else
{
    DebugLog(L"Disconnected with unrecoverable error, not attempting to reconnect.");
}
));

```

When the remoting component is ready to send a frame, your app is provided an opportunity to make a copy of it in the SendFrameEvent. Here, we copy the frame to a swap chain so that we can display it in a preview window.

```

m_streamerHelpers->OnSendFrame += ref new SendFrameEvent(
    [this](_In_ const ComPtr<ID3D11Texture2D>& spTexture, _In_ FrameMetadata metadata)
{
    if (m_showPreview)
    {
        ComPtr<ID3D11Device1> spDevice = m_appView->GetDeviceResources()->GetD3DDevice();
        ComPtr<ID3D11DeviceContext> spContext = m_appView->GetDeviceResources()-
>GetD3DDeviceContext();

        ComPtr<ID3D11Texture2D> spBackBuffer;
        ThrowIfFailed(m_swapChain->GetBuffer(0, IID_PPV_ARGS(&spBackBuffer)));

        spContext->CopySubresourceRegion(
            spBackBuffer.Get(), // dest
            0,                 // dest subresource
            0, 0, 0,           // dest x, y, z
            spTexture.Get(),  // source
            0,                 // source subresource
            nullptr);         // source box, null means the entire resource

        DXGI_PRESENT_PARAMETERS parameters = { 0 };
        ThrowIfFailed(m_swapChain->Present1(1, 0, &parameters));
    }
});

```

Render holographic content

To render content using remoting, you set up a virtual IFrameworkView within your desktop or UWP app and process holographic frames from remoting. All of the Windows Holographic APIs are used the same way by this view, but it is set up slightly differently.

Instead of creating them yourself, the holographic space and speech components come from your HolographicRemotingHelpers class:

```
m_appView->Initialize(m_streamerHelpers->HolographicSpace, m_streamerHelpers->RemoteSpeech);
```

Instead of using an update loop inside of a Run method, you provide tick updates from the main loop of your desktop or UWP app. This allows your desktop or UWP app to remain in control of message processing.

```
void DesktopWindow::Tick()
{
    auto lock = m_deviceLock.Lock();
    m_appView->Tick();

    // display preview, etc.
}
```

The holographic app view's Tick() method completes one iteration of the update, draw, present loop.

```
void AppView::Tick()
{
    if (m_main)
    {
        // When running on Windows Holographic, we can use the holographic rendering system.
        HolographicFrame^ holographicFrame = m_main->Update();

        if (holographicFrame && m_main->Render(holographicFrame))
        {
            // The holographic frame has an API that presents the swap chain for each
            // holographic camera.
            m_deviceResources->Present(holographicFrame);
        }
    }
}
```

The holographic app view update, render, and present loop is exactly the same as it is when running on HoloLens - except that you have access to a much greater amount of system resources on your desktop PC. You can render many more triangles, have more drawing passes, do more physics, and use x64 processes to load content that requires more than 2 GB of RAM.

Disconnect and end the remote session

To disconnect - for example, when the user clicks a UI button to disconnect - call Disconnect() on the HolographicStreamerHelpers, and then release the object.

```

void DesktopWindow::DisconnectFromRemoteDevice()
{
    // Disconnecting from the remote device can change the connection state.
    auto exclusiveLock = m_connectionStateLock.LockExclusive();

    if (m_steamerHelpers != nullptr)
    {
        m_steamerHelpers->Disconnect();

        // Reset state
        m_steamerHelpers = nullptr;
    }
}

```

Get the remoting player

The Windows Holographic remoting player is offered in the Windows app store as an endpoint for remoting host apps to connect to. To get the Windows Holographic remoting player, visit the Windows app store from your HoloLens, search for Remoting, and download the app. The remoting player includes a feature to display statistics on-screen, which can be useful when debugging remoting host apps.

Notes and resources

The holographic app view will need a way to provide your app with the Direct3D device, which must be used to initialize the holographic space. Your app should use this Direct3D device to copy and display the preview frame.

```

internal:
    const std::shared_ptr<DX::DeviceResources>& GetDeviceResources()
    {
        return m_deviceResources;
    }

```

Code sample: A complete Holographic Remoting code sample is available, which includes a holographic application view that is compatible with remoting and remoting host projects for desktop Win32, UWP DirectX, and UWP with XAML. To get it, go here:

- [Windows Holographic Code Sample for Remoting](#)

Debugging note: The Holographic Remoting library can throw first-chance exceptions. These exceptions may be visible in debugging sessions, depending on the Visual Studio exception settings that are active at the time. These exceptions are caught internally by the Holographic Remoting library and can be ignored.

Using WebVR in Microsoft Edge with Windows Mixed Reality

11/6/2018 • 2 minutes to read • [Edit Online](#)

Creating WebVR content for Windows Mixed reality immersive headsets

WebVR 1.1 is available in Microsoft Edge beginning with the Windows 10 Fall Creators Update. Developers can now use this API to create immersive VR experiences on the web.

The WebVR API provides HMD pose data to the page which can be used to render a stereo WebGL scene back to the HMD. Details on API support is available on the [Microsoft Edge Platform Status page](#). The WebVR API surface area is present at all times within Microsoft Edge. However, a call to `getVRDisplays()` will only return a headset if either a headset is plugged in or the simulator has been turned on.

Viewing WebVR content in Windows Mixed Reality immersive headsets

Instructions for accessing WebVR content in your immersive headset can be found in the [Enthusiast's Guide](#).

See Also

- [WebVR information](#)
- [WebVR specification](#)
- [WebVR API](#)
- [WebGL API](#)
- [Gamepad API and Gamepad Extensions](#)
- [Handling Lost Context in WebGL](#)
- [Pointerlock](#)
- [glTF](#)
- [Using Babylon.js to enable WebVR](#)

Porting guides

11/6/2018 • 9 minutes to read • [Edit Online](#)

Windows 10 includes support for immersive and holographic headsets directly. If you have built content for another device such as the Oculus Rift or HTC Vive, these have dependencies on libraries that exist above the operating system's platform API. Bringing existing content over to Windows Mixed Reality involves retargeting usage of these other SDKs to the Windows APIs. The [Windows platform APIs for mixed reality](#) only work in the Universal Windows Platform (UWP) app model. So if your app is not already built for UWP, porting to UWP will be part of the porting experience.

Porting overview

At a high-level, these are the steps involved in porting existing content:

1. **Make sure your PC is running the Windows 10 Fall Creators Update (16299).** We no longer recommend receiving preview builds from the Insider Skip Ahead ring, as those builds won't be the most stable for mixed reality development.
2. **Upgrade to the latest version of your graphics or game engine.** Game engines will need to support the Windows 10 SDK version 10.0.15063.0 (released in April 2017) or higher.
3. **Upgrade any middleware, plug-ins or components.** If your app contains any components, it's a good idea to upgrade to the latest version. Newer versions of most common plug-ins have support for UWP.
4. **Remove dependencies on duplicate SDKs.** Depending on which device your content was targeting, you'll need to remove or conditionally compile out that SDK (e.g. SteamVR) so you can target the Windows APIs instead.
5. **Work through build issues.** At this point, the porting exercise is specific to your app, your engine, and the component dependencies you have.

Common porting steps

Common step 1: Make sure you have the right development hardware

The [install the tools](#) page lists the recommended development hardware.

Common step 2: Upgrade to the latest flight of Windows 10

The Windows Mixed Reality platform is still under active development, and to be most effective, we recommend being on the "Windows Insider Fast" flight. In order to have access to windows flights, you will need to [join the Windows Insider Program](#).

1. Install the [Windows 10 Creators Update](#)
2. [Join](#) the Windows Insider Program.
3. Enable [Developer Mode](#)
4. Switch to the [Windows Insider Fast flights](#) through Settings --> Update & Security Section

Common step 3: Upgrade to the most recent build of Visual Studio

- Please see [Install the tools](#) page under Visual Studio 2017

Common step 4: Be Ready for The Store

- Use [Windows App Certification Kit](#) (aka WACK) early and often!
- Use [Portability Analyzer](#) ([Download](#))

Common step 5: Choose the correct Adapter

- In systems like notebooks with two GPUs, [target the correct adapter](#). This applies to Unity apps, in addition to native DirectX apps, where a ID3D11Device is created, either explicitly or implicitly (Media Foundation), for its functionality.

Unity porting guidance

Unity step 1: Follow the common porting steps

Follow all of the common steps. When in step #3, select the **Game Development with Unity** workload. You may deselect the Unity Editor optional component since you'll be installing a newer version of Unity from the instructions below.

Unity step 2: Upgrade to the latest public build of Unity with Windows MR Support

1. Download the latest [recommended public build of Unity](#) with mixed reality support.
2. Save a copy of your project before you get started
3. Review the [documentation](#) available from Unity on porting.
4. Follow the [instructions](#) on Unity's site for using their automatic API updater
5. Check and see if there are additional changes that you need to make to get your project running, and work through any remaining errors and warnings. Note: If you have middleware that you depend on, you may need to update that middleware to get going (more details in step 3 below).

Unity step 3: Upgrade your middleware to the latest versions

With any Unity update, there is a good chance that you need to update one or more middleware packages that your game or application depends on. Additionally, being on the latest version of all of your middleware will increase your likelihood of success throughout the rest of the porting process. Many middleware packages have recently added support for Universal Windows Platform (UWP), and upgrading to the most recent versions will let you leverage that work.

Unity step 4: Target your application to run on Universal Windows Platform (UWP)

After installing the tools, you need to get your app running as a Universal Windows app.

- Follow the [detailed step by step walk through](#) provided by Unity. Please note that you should stay on the latest LTS release (any 20xx.4 release) for Windows MR.
- For more UWP development resources, take a look at the [Windows 10 game development guide](#).
- Please note that Unity continues to improve IL2CPP support; IL2CPP makes some UWP ports significantly easier. If you are currently targeting the .Net scripting backend, you should consider converting to leverage the IL2CPP backend instead.

Note: If your application has any dependencies on device specific services, such as match making from Steam, you will need to disable them at this step. At a later time, you can hook up to the equivalent services that Windows provides.

Unity step 5: (Deprecated)

Step 5 is no longer required. We are leaving it here so that the indexing of steps remains the same.

Unity step 6: Get your Windows Mixed Reality hardware set up

1. Review steps in [Immersive headset setup](#)
2. Learn about [Using the Windows Mixed Reality simulator](#) and [Navigating the Windows Mixed Reality home](#)

Unity step 7: Target your application to run on Windows Mixed Reality

1. First, you must remove or conditionally compile out any other library support specific to a particular VR SDK. Those assets frequently change settings and properties on your project in ways that are incompatible with other VR SDKs, such as Windows Mixed Reality.
 - For example, if your project references the SteamVR SDK, you will need to update your project to

exclude those prefabs and script API calls when exporting for the Windows Store build target.

- Specific steps for conditionally excluding other VR SDKs is coming soon.

2. In your Unity project, [target the Windows 10 SDK](#)

3. For each scene, [setup the camera](#)

Unity step 8: Use the stage to place content on the floor

You can build Mixed Reality experiences across a wide range of [experience scales](#).

If you're porting a **seated-scale experience**, you must ensure Unity is set to the **Stationary** tracking space type:

```
XRDevice.SetTrackingSpaceType(TrackingSpaceType.Stationary);
```

This sets Unity's world coordinate system to track the [stationary frame of reference](#). In the Stationary tracking mode, content placed in the editor just in front of the camera's default location (forward is -Z) will appear in front of the user when the app launches. To recenter the user's seated origin, you can call Unity's [XR.InputTracking.Recenter](#) method.

If you're porting a **standing-scale experience** or **room-scale experience**, you'll be placing content relative to the floor. You reason about the user's floor using the **spatial stage**, which represents the user's defined floor-level origin and optional room boundary, set up during first run. For these experiences, you must ensure Unity is set to the **RoomScale** tracking space type. While RoomScale is the default, you'll want to set it explicitly and ensure you get back true, to catch situations where the user has moved their computer away from the room they calibrated:

```
if (XRDevice.SetTrackingSpaceType(TrackingSpaceType.RoomScale))
{
    // RoomScale mode was set successfully. App can now assume that y=0 in Unity world coordinate represents
    // the floor.
}
else
{
    // RoomScale mode was not set successfully. App cannot make assumptions about where the floor plane is.
}
```

Once your app successfully sets the RoomScale tracking space type, content placed on the y=0 plane will appear on the floor. The origin at (0, 0, 0) will be the specific place on the floor where the user stood during room setup, with -Z representing the forward direction they were facing during setup.

In script code, you can then call the TryGetGeometry method on your the UnityEngine.Experimental.XR.Boundary type to get a boundary polygon, specifying a boundary type of TrackedArea. If the user defined a boundary (you get back a list of vertices), you know it is safe to deliver a **room-scale experience** to the user, where they can walk around the scene you create.

Note that the system will automatically render the boundary when the user approaches it. Your app does not need to use this polygon to render the boundary itself.

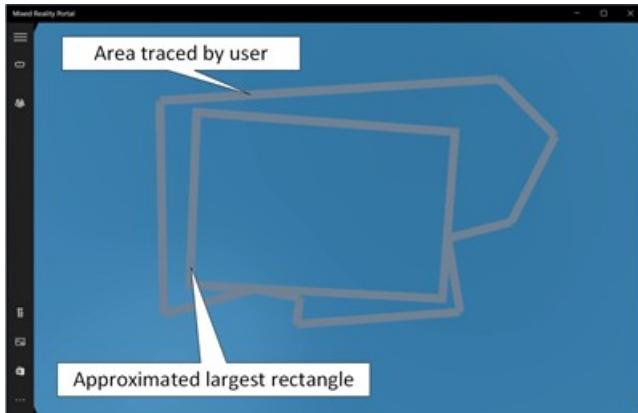
For more details, see the [Coordinate systems in Unity](#) page.

Some applications use a rectangle to constrain their interaction. Retrieving the largest inscribed rectangle is not directly supported in the UWP API or Unity. The example code linked to below shows how to find a rectangle within the traced bounds. It is heuristic based so may not find the optimal solution, however, results are generally consistent with expectations. Parameters in the algorithm can be tuned to find more precise results at the cost of processing time. The algorithm is in a fork of the Mixed Reality Toolkit that uses the 5.6 preview MRTP version of Unity. This is not publically available. The code should be directly usable in 2017.2 and higher versions of Unity. The code will be ported to the current MRTK in the near future.

[zip file of code on GitHub](#) Important files:

- Assets/HoloToolkit/Stage/Scripts/StageManager.cs - example of usage
- Assets/HoloToolkit/Stage/Scripts/LargestRectangle.cs - implementation of the algorithm
- Assets/HoloToolkit-UnitTests/Editor/Stage/LargestRectangleTest.cs - trivial test of the algorithm

Example of results:



Algorithm is based on a blog by Daniel Smilkov: [Largest rectangle in a polygon](#)

Unity step 9: Work through your input model

Each game or application targeting an existing HMD will have a set of inputs that it handles, types of inputs that it needs for the experience, and specific APIs that it calls to get those inputs. We have invested in trying to make it as simple and straightforward as possible to take advantage of the inputs available in Windows Mixed Reality.

1. Read through the [Input porting guide for Unity](#) for details of how Windows Mixed Reality exposes input, and how that maps to what your application may do today.
2. Choose whether you are going to leverage Unity's cross-VR-SDK input API, or the MR-specific input API. The general Input.GetButton/Input.GetAxis APIs are used by Unity VR apps today for [Oculus input](#) and [OpenVR input](#). If your apps are already using these APIs for motion controllers, this is the easiest path - you should just need to remap buttons and axes in the Input Manager.
 - You can access motion controller data in Unity using either the general cross-VR-SDK Input.GetButton/Input.GetAxis APIs or the MR-specific UnityEngine.XR.WSA.Input APIs. (previously in the UnityEngine.XR.WSA.Input namespace in Unity 5.6)
 - See the [example in Toolkit](#) that combines gamepad and motion controllers.

Unity step 10: Performance testing and tuning

Windows Mixed Reality will be available on a broad class of devices, ranging from high end gaming PCs, down to broad market mainstream PCs. Depending on what market you are targeting, there is a significant difference in the available compute and graphics budgets for your application. During this porting exercise, you are likely leveraging a premium PC, and have had significant compute and graphics budgets available to your app. If you wish to make your app available to a broader audience, you should test and profile your app on [the representative hardware that you wish to target](#).

Both [Unity](#) and [Visual Studio](#) include performance profilers, and both [Microsoft](#) and [Intel](#) publish guidelines on performance profiling and optimization. There is an extensive discussion of performance available at [Performance recommendations for HoloLens apps](#).

See also

- [Input porting guide for Unity](#)
- [Windows Mixed Reality minimum PC hardware compatibility guidelines](#)
- [Performance recommendations for immersive headset apps](#)
- [Add Xbox Live support to Unity for UWP](#)

Input porting guide for Unity

11/6/2018 • 2 minutes to read • [Edit Online](#)

You can port your input logic to Windows Mixed Reality using one of two approaches, Unity's general `Input.GetButton`/`GetAxis` APIs that span across multiple platforms, or the Windows-specific `XR.WSA.Input` APIs that offer richer data specifically for motion controllers and HoloLens hands.

General `Input.GetButton`/`GetAxis` APIs

Unity currently uses its general `Input.GetButton`/`Input.GetAxis` APIs to expose input for [the Oculus SDK](#) and [the OpenVR SDK](#). If your apps are already using these APIs for input, this is the easiest path for supporting motion controllers in Windows Mixed Reality: you should just need to remap buttons and axes in the Input Manager.

For more information, see the [Unity button/axis mapping table](#) and the [overview of the common Unity APIs](#).

Windows-specific `XR.WSA.Input` APIs

If your app already builds custom input logic for each platform, you can choose to use the Windows-specific spatial input APIs under the **UnityEngine.XR.WSA.Input** namespace. This lets you access additional information, such as position accuracy or the source kind, letting you tell hands and controllers apart on HoloLens.

For more information, see the [overview of the UnityEngine.XR.WSA.Input APIs](#).

Grip pose vs. pointing pose

Windows Mixed Reality supports motion controllers in a variety of form factors, with each controller's design differing in its relationship between the user's hand position and the natural "forward" direction that apps should use for pointing when rendering the controller.

To better represent these controllers, there are two kinds of poses you can investigate for each interaction source:

- The **grip pose**, representing the location of either the palm of a hand detected by a HoloLens, or the palm holding a motion controller.
 - On immersive headsets, this pose is best used to render **the user's hand or an object held in the user's hand**, such as a sword or gun.
 - The **grip position**: The palm centroid when holding the controller naturally, adjusted left or right to center the position within the grip.
 - The **grip orientation's Right axis**: When you completely open your hand to form a flat 5-finger pose, the ray that is normal to your palm (forward from left palm, backward from right palm)
 - The **grip orientation's Forward axis**: When you close your hand partially (as if holding the controller), the ray that points "forward" through the tube formed by your non-thumb fingers.
 - The **grip orientation's Up axis**: The Up axis implied by the Right and Forward definitions.
 - You can access the grip pose through either Unity's cross-vendor input API ([XR.InputTracking.GetLocalPosition/Rotation](#)) or through the Windows-specific API (`sourceState.sourcePose.TryGetPosition/Rotation`, requesting the Grip pose).
- The **pointer pose**, representing the tip of the controller pointing forward.
 - This pose is best used to raycast when **pointing at UI** when you are rendering the controller model itself.
 - Currently, the pointer pose is available only through the Windows-specific API (`sourceState.sourcePose.TryGetPosition/Rotation`, requesting the Pointer pose).

These pose coordinates are all expressed in Unity world coordinates.

See also

- [Motion controllers](#)
- [Gestures and motion controllers in Unity](#)
- [UnityEngine.XR.WSA.Input](#)
- [UnityEngine.XR.InputTracking](#)
- [Porting guides](#)

Updating your SteamVR application for Windows Mixed Reality

11/6/2018 • 3 minutes to read • [Edit Online](#)

We encourage developers to test and optimize their SteamVR experiences to run on Windows Mixed Reality headsets. This documentation covers common improvements developers can make to ensure that their experience runs great on Windows Mixed Reality.

Initial setup instructions

To start testing out your game or app on Windows Mixed Reality make sure to first follow our [getting started guide](#).

Controller Models

1. If your app renders controller models:
 - Use the [Windows Mixed Reality motion controller models](#)
 - Use IVRRenderModel::GetComponentState to get local transforms to component parts (eg. Pointer pose)
2. Experiences that have a notion of handedness should get hints from the input APIs to differentiate controllers ([Unity example](#))

Controls

When designing or adjusting your control layout keep in mind the following set of reserved commands:

1. Clicking down the **left and right analog thumbstick** is reserved for the **Steam Dashboard**.
2. The **Windows button** will always return users to the Windows Mixed Reality home.

If possible, default to thumb stick based teleportation to match the [Windows Mixed Reality home](#) teleportation behavior

Tooltips and UI

Many VR games take advantage of motion controller tooltips and overlays to teach users the most important commands for their game or app. When tuning your application for Windows Mixed reality we recommend reviewing this part of your experience to make sure the tooltips map to the Windows controller models.

Additionally if there are any points in your experience where you display images of the controllers make sure to provide updated images using the Windows Mixed Reality motion controllers.

Haptics

Beginning with the [Windows 10 April 2018 Update](#), haptics are now supported for SteamVR experiences on Windows Mixed Reality. If your SteamVR app or game already includes support for haptics, it should now work (with no additional work) with [Windows Mixed Reality motion controllers](#).

Windows Mixed Reality motion controllers use a standard haptics motor, as opposed to the linear actuators found in some other SteamVR motion controllers, which can lead to a slightly different-than-expected user experience. So, we recommend testing and tuning your haptics design with Windows Mixed Reality motion controllers. For

example, sometimes short haptic pulses (5-10 ms) are less noticeable on Windows Mixed Reality motion controllers. To produce a more noticeable pulse, experiment with sending a longer "click" (40-70ms) to give the motor more time to spin up before being told to power off again.

Launching SteamVR apps from Windows Mixed Reality Start menu

For VR experiences distributed through Steam, we've [updated the Windows Mixed Reality for SteamVR Beta](#) along with the latest [Windows Insider RS5](#) flights so that SteamVR titles now show up in the Windows Mixed Reality Start menu in the "All apps" list automatically. With these software versions installed, your customers can now launch SteamVR titles directly from within the Windows Mixed Reality home without removing their headsets.

Windows Mixed Reality logo

To display Windows Mixed Reality support for your title, go to the "Edit Store Page" link on your App Landing Page, click the "Basic Info" tab, and scroll down to "Virtual Reality." Uncheck the "Hide Windows Mixed Reality" and then publish to the store.

Bugs and feedback

Your feedback is invaluable when it comes to improving the Windows Mixed Reality SteamVR experience. Please submit all feedback and bugs through the [Windows Feedback Hub](#). Here are some [tips on how to make your SteamVR feedback as helpful as possible](#).

If you have questions or comments to share, you can also reach us on our [Steam forum](#).

FAQs and troubleshooting

If you're running into general issues setting up or playing your experience, [check out the latest troubleshooting steps](#).

See also

- [Install the tools](#)
- [Headset and motion controller driver history](#)
- [Windows Mixed Reality minimum PC hardware compatibility guidelines](#)

Updating 2D UWP apps for mixed reality

11/6/2018 • 9 minutes to read • [Edit Online](#)

Windows Mixed Reality enables a user to see holograms as if they are right around you, in your physical or digital world. At its core, both HoloLens and the Desktop PCs you attach immersive headset accessories to are Windows 10 devices; this means that you're able to run almost all of the Universal Windows Platform (UWP) apps in the Store as 2D apps.

Creating a 2D UWP app for mixed reality

The first step to bringing a 2D app to mixed reality headsets is to get your app running as a standard 2D app on your desktop monitor.

Building a new 2D UWP app

To build a new 2D app for mixed reality, you simply build a standard 2D Universal Windows Platform (UWP) app. No other app changes are required for that app to then run as a slate in mixed reality.

To get started building a 2D UWP app, check out the [Create your first app](#) article.

Bringing an existing 2D Store app to UWP

If you already have a 2D Windows app in the Store, you must first ensure it is targeting the Windows 10 Universal Windows Platform (UWP). Here are all the potential starting points you may have with your Store app today:

STARTING POINT	APPX MANIFEST PLATFORM TARGET	HOW TO MAKE THIS UNIVERSAL?
Windows Phone (Silverlight)	Silverlight App Manifest	Migrate to WinRT
Windows Phone 8.1 Universal	8.1 AppX Manifest that Doesn't Include Platform Target	Migrate your app to the Universal Windows Platform
Windows Store 8	8 AppX Manifest that Doesn't Include Platform Target	Migrate your app to the Universal Windows Platform
Windows Store 8.1 Universal	8.1 AppX Manifest that Doesn't Include Platform Target	Migrate your app to the Universal Windows Platform

If you have a 2D Unity app today built as a Win32 app (the "PC, Mac & Linux Standalone" build target), you can target mixed reality by switching Unity to the "Universal Windows Platform" build target instead.

We'll talk about ways that you can restrict your app specifically to HoloLens using the Windows.Holographic device family [below](#).

Run your 2D app in a Windows Mixed Reality immersive headset

If you've deployed your 2D app to the desktop machine where you are developing and tried it out on your monitor, you're already ready to try it out in an immersive desktop headset!

Just go to the Start menu within the mixed reality headset and launch the app from there. The desktop shell and the holographic shell both share the same set of UWP apps, and so the app should already be present once you've deployed from Visual Studio.

Targeting both immersive headsets and HoloLens

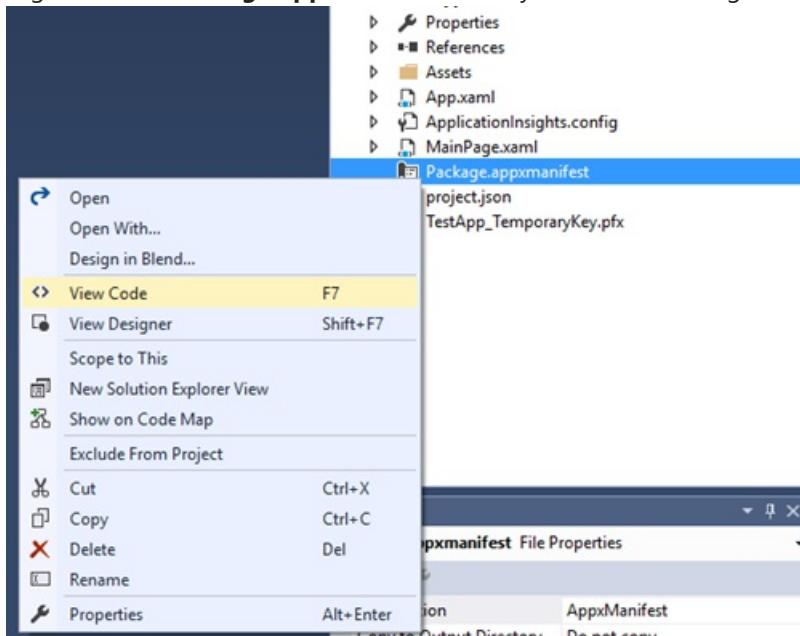
Congratulations! Your app is now using the Windows 10 Universal Windows Platform (UWP).

Your app is now capable of running on today's Windows devices like Desktop, Mobile, Xbox, Windows Mixed Reality immersive headsets, and HoloLens, as well as future Windows devices. However, to actually target all of those devices, you will need to ensure your app is targeting the Windows.Universal device family.

Change your device family to Windows.Universal

Now let's jump into your AppX manifest to ensure your Windows 10 UWP app can run on HoloLens:

- Open your app's solution file with **Visual Studio** and navigate to the app package manifest
- Right click the **Package.appxmanifest** file in your Solution and go to **View Code**



- Ensure your Target Platform is Windows.Universal in the dependencies section

```
<Dependencies>
  <TargetDeviceFamily Name="Windows.Universal" MinVersion="10.0.10240.0"
    MaxVersionTested="10.0.10586.0" />
</Dependencies>
```

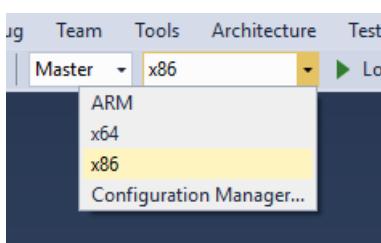
- Save!

If you do not use Visual Studio for your development environment, you can open **AppXManifest.xml** in the text editor of your choice to ensure you're targeting the **Windows.Universal** *TargetDeviceFamily*.

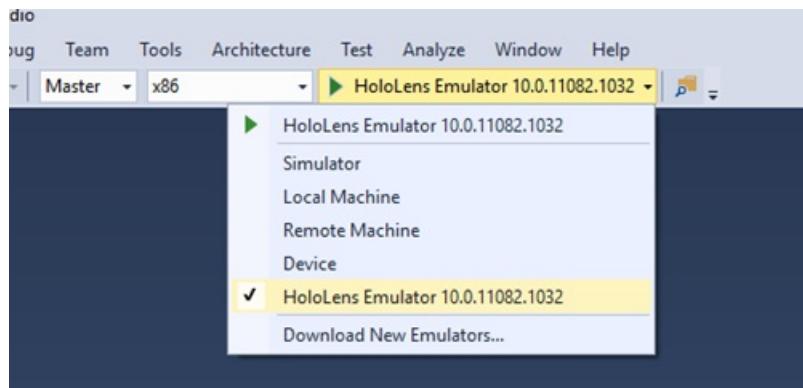
Run in the HoloLens Emulator

Now that your UWP app targets "Windows.Universal", let's build your app and run it in the [HoloLens Emulator](#).

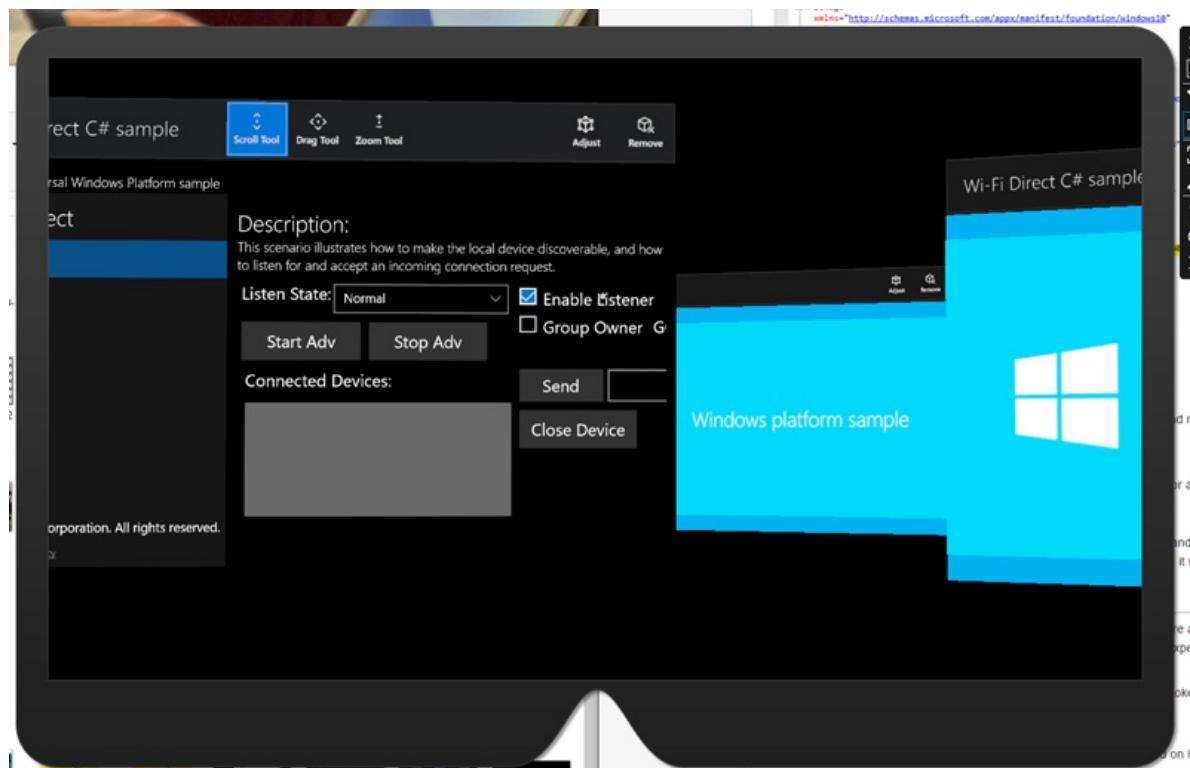
- Make sure you have [installed the HoloLens Emulator](#).
- In Visual Studio, select the **x86** build configuration for your app



- Select **HoloLens Emulator** in the deployment target drop-down menu



- Select **Debug > Start Debugging** to deploy your app and start debugging.
- The emulator will start and run your app.
- With a keyboard, mouse, and/or an Xbox controller, place your app in the world to launch it.



Next steps

At this point, one of two things can happen:

1. Your app will show its splash and start running after it is placed in the Emulator! Awesome!
2. Or after you see a loading animation for a 2D hologram, loading will stop and you will just see your app at its splash screen. This means that something went wrong and it will take more investigation to understand how to bring your app to life in Mixed Reality.

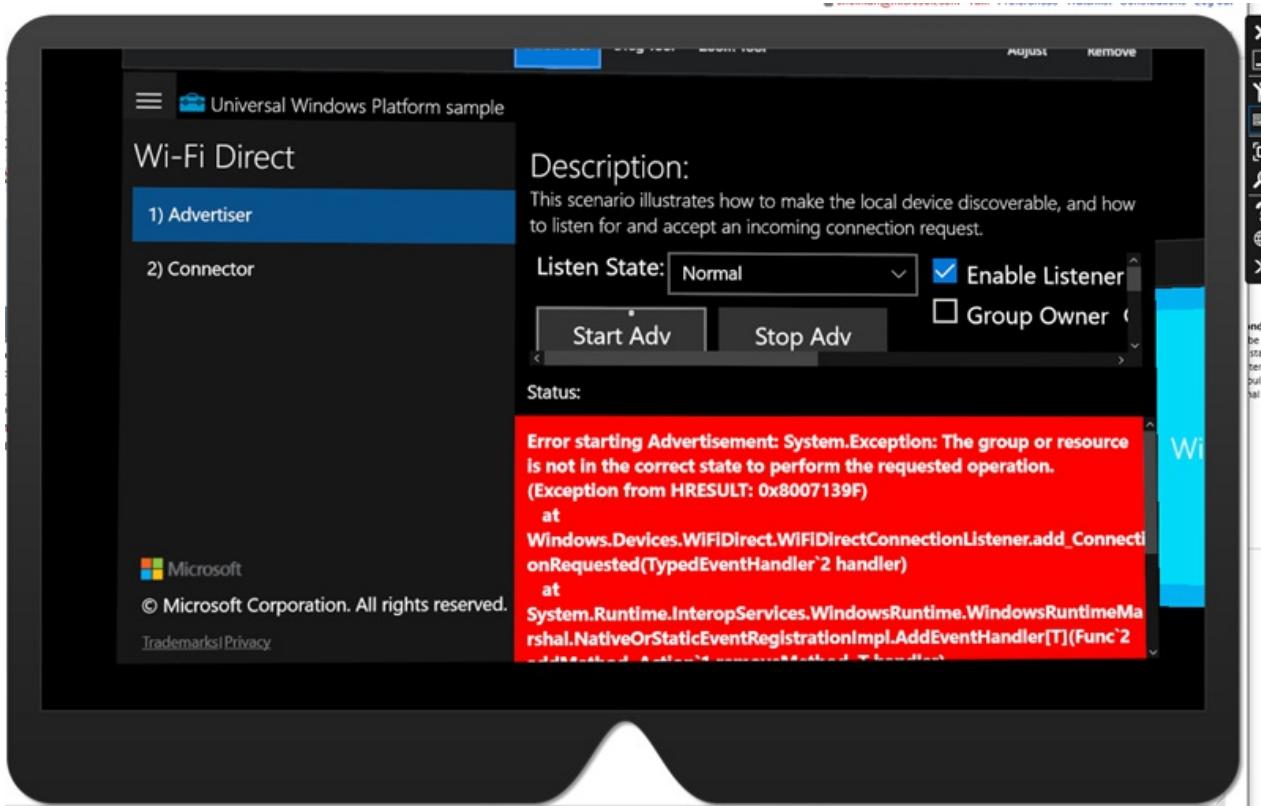
To get to the bottom of what may be causing your UWP app not to start on HoloLens, you'll have to debug.

Running your UWP app in the debugger

These steps will walk you through debugging your UWP app using the Visual Studio debugger.

- If you haven't already done so, open your solution in Visual Studio. Change the target to the **HoloLens Emulator** and the build configuration to **x86**.
- Select **Debug > Start Debugging** to deploy your app and start debugging.
- Place the app in the world with your mouse, keyboard, or Xbox controller.

- Visual Studio should now break somewhere in your app code.
 - If your app doesn't immediately crash or break into the debugger because of an unhandled error, then go through a test pass of the core features of your app to make sure everything is running and functional. You may see errors like pictured below (internal exceptions that are being handled). To ensure you don't miss internal errors that impact the experience of your app, run through your automated tests and unit tests to make sure everything behaves as expected.

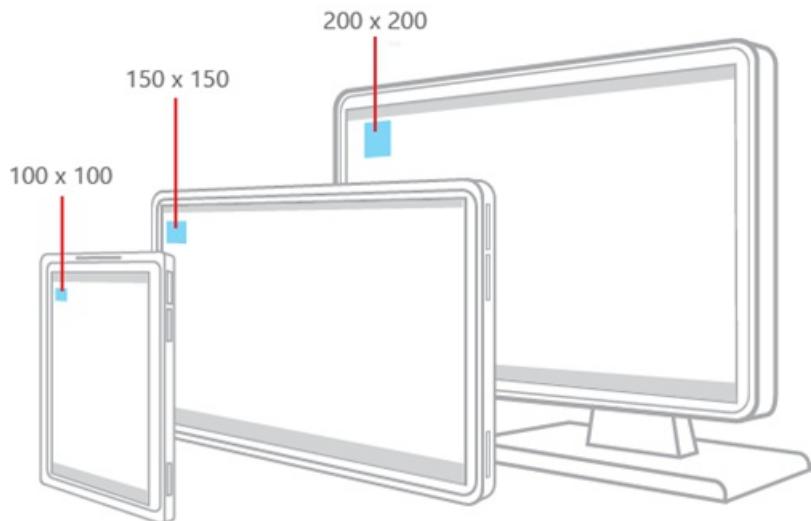


Update your UI

Now that your UWP app is running on immersive headsets and/or HoloLens as a 2D hologram, next we'll make sure it looks beautiful. Here are some things to consider:

- Windows Mixed Reality will run all 2D apps at a fixed resolution and DPI that equates to 853x480 effective pixels. Consider if your design needs refinement at this scale and review the design guidance below to improve your experience on HoloLens and immersive headsets.
- Windows Mixed Reality [does not support](#) 2D live tiles. If your core functionality is showing information on a live tile, consider moving that information back into your app or explore [3D app launchers](#).

2D app view resolution and scale factor



Windows 10 moves all visual design from real screen pixels to **effective pixels**. That means, developers design their UI following the Windows 10 Human Interface Guidelines for effective pixels, and Windows scaling ensures those effective pixels are the right size for usability across devices, resolutions, DPI, etc. See this [great read on MSDN](#) to learn more as well as this [BUILD presentation](#).

Even with the unique ability to place apps in your world at a range of distances, TV-like viewing distances are recommended to produce the best readability and interaction with gaze/gesture. Because of that, a virtual slate in the Mixed Reality Home will display your flat UWP view at:

1280x720, 150%DPI (853x480 effective pixels)

This resolution has several advantages:

- This effective pixel layout will have about the same information density as a tablet or small desktop.
- It matches the fixed DPI and effective pixels for UWP apps running on Xbox One, enabling seamless experiences across devices.
- This size looks good when scaled across our range of operating distances for apps in the world.

2D app view interface design best practices

Do:

- Follow the [Windows 10 Human Interface Guidelines \(HIG\)](#) for styles, font sizes and button sizes. HoloLens will do the work to ensure your app will have compatible app patterns, readable text sizes, and appropriate hit target sizing.
- Ensure your UI follows best practices for [responsive design](#) to look best at HoloLen's unique resolution and DPI.
- Use the "light" color theme recommendations from Windows.

Don't:

- Change your UI too drastically when in mixed reality, to ensure users have a familiar experience in and out of the headset.

Understand the app model

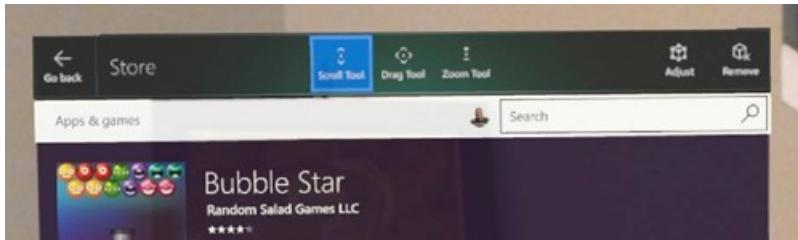
The [app model](#) for mixed reality is designed to use the Mixed Reality Home, where many apps live together. Think of this as the mixed reality equivalent of the desktop, where you run many 2D apps at once. This has implications on app life cycle, Tiles, and other key features of your app.

App bar and back button

2D views are decorated with a app bar above their content. The app bar has two points of app-specific personalization:

Title: displays the *displayname* of the Tile associated with the app instance

Back Button: raises the *BackRequested* event when pressed. Back Button visibility is controlled by *SystemNavigationManager.AppViewBackButtonVisibility*.



App bar UI in 2D app view

Test your 2D app's design

It is important to test your app to make sure the text is readable, the buttons are targetable, and the overall app looks correct. You can [test](#) on a desktop headset, a HoloLens, an emulator, or a touch device with resolution set to 1280x720 @150%.

New input possibilities

HoloLens uses advanced depth sensors to see the world and see users. This enables advanced hand gestures like [bloom](#) and [air-tap](#). Powerful microphones also enable [voice experiences](#).

With Desktop headsets, users can use motion controllers to point at apps and take action. They can also use a gamepad, targeting objects with their gaze.

Windows takes care of all of this complexity for UWP apps, translating your [gaze](#), gestures, voice and motion controller input to [pointer events](#) that abstract away the input mechanism. For example, a user may have done an air-tap with their hand or pulled the Select trigger on a motion controller, but 2D applications don't need to know where the input came from - they just see a 2D touch press, as if on a touchscreen.

Here are the high level concepts/scenarios you should understand for input when bringing your UWP app to HoloLens:

- [Gaze](#) turns into hover events, which can unexpectedly trigger menus, flyouts or other user interface elements to pop up just by gazing around your app.
- Gaze is not as precise as mouse input. Use appropriately sized hit targets for HoloLens, similar to touch-friendly mobile applications. Small elements near the edges of the app are especially hard to interact with.
- Users must switch input modes to go from scrolling to dragging to two finger panning. If your app was designed for touch input, consider ensuring that no major functionality is locked behind two finger panning. If so, consider having alternative input mechanisms like buttons that can initiate two finger panning. For example, the Maps app can zoom with two finger panning but has a plus, minus, and rotate button to simulate the same zoom interactions with single clicks.

[Voice input](#) is a critical part of the mixed reality experience. We've enabled all of the speech APIs that are in Windows 10 powering Cortana when using a headset.

Publish and Maintain your Universal app

Once your app is up and running, package your app to [submit it to the Microsoft Store](#).

See also

- [App model](#)
- [Gaze](#)
- [Gesture](#)
- [Motion controllers](#)
- [Voice](#)
- [Submitting an app to the Microsoft Store](#)
- [Using the HoloLens emulator](#)

Rendering

11/6/2018 • 5 minutes to read • [Edit Online](#)

Holographic rendering enables your app to draw a hologram in a precise location in the world around the user, whether it's precisely placed in the physical world or within a virtual realm you've created. [Holograms](#) are objects made of sound and light, and rendering enables your app to add the light.

Holographic rendering

Key to holographic rendering is knowing whether you are rendering to a see-through display like HoloLens, which lets the user see both the physical world and your holograms together, or an opaque display like a Windows Mixed Reality immersive headset, which blocks out the world.

Devices with **see-through displays**, like [HoloLens](#), add light to the world. Black pixels will be fully transparent, while brighter pixels will be increasingly opaque. Because the light from the displays is added to the light from the real world, even white pixels are somewhat translucent.

While stereoscopic rendering provides one depth cue for your holograms, adding [grounding effects](#) can help users see more easily what surface a hologram is near. One grounding technique is to add a glow around a hologram on the nearby surface and then render a shadow against this glow. In this way, your shadow will appear to subtract light from the environment. [Spatial sound](#) can be another extremely important depth cue, letting users reason about the distance and relative location of a hologram.

Devices with **opaque displays**, like [Windows Mixed Reality immersive headsets](#), block out the world. Black pixels will be solid black, and any other color will appear as that color to the user. Your app is responsible for rendering everything the user will see, so it's even more important to maintain a constant refresh rate so that users have a comfortable experience.

Predicted rendering parameters

Mixed reality headsets (both HoloLens and immersive headsets) continually track the position and orientation of the user's head relative to their surroundings. As your app begins preparing its next frame, the system predicts where the user's head will be in the future at the exact moment that the frame will show up on the displays. Based on this prediction, the system calculates the view and projection transforms to use for that frame. Your application **absolutely has to use these transforms to produce correct results**; if system-supplied transforms are not used, the resulting image will not align with the real world, leading to user discomfort.

Note that to accurately predict when a new frame will reach the displays, the system is constantly measuring the effective end-to-end latency of your app's rendering pipeline. While the system will adjust to the length of your rendering pipeline, you can further improve hologram stability by keeping that pipeline as short as possible.

Other rendering parameters

When rendering a frame, the system will specify the back-buffer viewport in which your application should draw. This viewport will often be smaller than the full size of the frame buffer. Regardless of the viewport size, once the frame has been rendered by the application, the system will upscale the image to fill the entirety of the displays.

For applications that find themselves unable to render at the required refresh rate, [reducing the viewport size](#) can be used as a mechanism to reduce rendering cost at the cost of increased pixel aliasing.

The rendering frustum, resolution, and framerate in which your app is asked to render may also change from frame to frame and may differ across the left and right eye. For example, when [mixed reality capture](#) (MRC) is

active, one eye may be rendered with a larger FOV or resolution.

For any given frame, your app *must* render using the view transform, projection transform, and viewport resolution provided by the system. Additionally, your application must never assume that any rendering/view parameter remains fixed from frame-to-frame. Engines like Unity handle all these transforms for you in their own Camera objects, so that the physical movement of your users and the state of the system is always respected. If your app further allows for virtual movement of the user through the world (e.g. using the thumbstick on a gamepad), you can add a parent "rig" object above the camera that moves it around, causing the camera to reflect both the user's virtual and physical motion.

To enhance the stability of your holographic rendering, your app should provide to Windows each frame the depth buffer it used for rendering. If your app does provide a depth buffer, it should have coherent depth values, with depth expressed in meters from the camera. This enables the system to use your per-pixel depth data to better stabilize content if the user's head ends up slightly offset from the predicted location. If you are not able to provide your depth buffer, you should instead provide a focus point and normal, defining a plane that cuts through most of your content. If provided, this plane will be used instead of the depth buffer to do a simpler stabilization of content along that plane.

For all the low-level details here, check out the [Rendering in DirectX](#) article.

Holographic cameras

Windows Mixed Reality introduces the concept of a **holographic camera**. Holographic cameras are similar to the traditional camera found in 3D graphics texts: they define both the extrinsic (position and orientation) and intrinsic camera properties (ex: field-of-view) used to view a virtual 3D scene. Unlike traditional 3D cameras, the application is not in control of the position, orientation, and intrinsic properties of the camera. Rather, the position and orientation of the holographic camera is implicitly controlled by the user's movement. The user's movement is relayed to the application on a frame-by-frame basis via a view transform. Likewise, the camera's intrinsic properties are defined by the device's calibrated optics and relayed frame-by-frame via the projection transform.

In general your app will be rendering for a single stereo camera. However, a robust rendering loop should support multiple cameras and should support both mono and stereo cameras. For example, the system may ask your app to render from an alternate perspective when the user activates a feature like [mixed reality capture](#) (MRC), depending on the shape of the headset in question.

Volume rendering

When rendering medical MRI or engineering volumes in 3D, [volume rendering](#) techniques are often used. These techniques can be particularly interesting in mixed reality, where users can naturally view such a volume from key angles, simply by moving their head.

Supported resolutions on HoloLens

- The default and maximum supported resolution is 720p (1268x720).
- The lowest supported viewport size is 50% of 720p (a ViewportScaleFactor of 0.5), which is 360p (634x360).
- Anything lower than 540p is **not recommended** due to visual degradation but can be used to identify bottle necks in pixel fill rate.

See also

- [Hologram stability](#)
- [Rendering in DirectX](#)

Volume rendering

11/6/2018 • 5 minutes to read • [Edit Online](#)

For medical MRI or engineering volumes, see [Volume Rendering on Wikipedia](#). These 'volumetric images' contain rich information with opacity and color throughout the volume that cannot be easily expressed as surfaces such as [polygonal meshes](#).

Key solutions to improve performance

1. BAD: Naïve Approach: Show Whole Volume, generally runs too slowly
2. GOOD: Cutting Plane: Show only a single slice of the volume
3. GOOD: Cutting Sub-Volume: Show only a few layers of the volume
4. GOOD: Lower the resolution of the volume rendering (see 'Mixed Resolution Scene Rendering')

There is only a certain amount of information that can be transferred from the application to the screen in any particular frame, this is the total memory bandwidth. Also any processing (or 'shading') required to transform that data for presentation also requires time. The primary considerations when doing volume rendering are as such:

- Screen-Width * Screen-Height * Screen-Count * Volume-Layers-On-That-Pixel = Total-Volume-Samples-Per-Frame
- $1028 * 720 * 2 * 256 = 378961920$ (100%) (full res volume: too many samples)
- $1028 * 720 * 2 * 1 = 1480320$ (0.3% of full) (thin slice: 1 sample per pixel, runs smoothly)
- $1028 * 720 * 2 * 10 = 14803200$ (3.9% of full) (sub-volume slice: 10 samples per pixel, runs fairly smoothly, looks 3d)
- $200 * 200 * 2 * 256 = 20480000$ (5% of full) (lower res volume: fewer pixels, full volume, looks 3d but a bit blurry)

Representing 3D Textures

On the CPU:

```

public struct Int3 { public int X, Y, Z; /* ... */ }
public class VolumeHeader {
    public readonly Int3 Size;
    public VolumeHeader(Int3 size) { this.Size = size; }
    public int CubicToLinearIndex(Int3 index) {
        return index.X + (index.Y * (Size.X)) + (index.Z * (Size.X * Size.Y));
    }
    public Int3 LinearToCubicIndex(int linearIndex)
    {
        return new Int3((linearIndex / 1) % Size.X,
            (linearIndex / Size.X) % Size.Y,
            (linearIndex / (Size.X * Size.Y)) % Size.Z);
    }
    /* ... */
}
public class VolumeBuffer<T> {
    public readonly VolumeHeader Header;
    public readonly T[] DataArray;
    public T GetVoxel(Int3 pos) {
        return this.DataArray[this.Header.CubicToLinearIndex(pos)];
    }
    public void SetVoxel(Int3 pos, T val) {
        this.DataArray[this.Header.CubicToLinearIndex(pos)] = val;
    }
    public T this[Int3 pos] {
        get { return this.GetVoxel(pos); }
        set { this.SetVoxel(pos, value); }
    }
    /* ... */
}

```

On the GPU:

```

float3 _VolBufferSize;
int3 UnitVolumeToIntVolume(float3 coord) {
    return (int3)( coord * _VolBufferSize.xyz );
}
int IntVolumeToLinearIndex(int3 coord, int3 size) {
    return coord.x + ( coord.y * size.x ) + ( coord.z * ( size.x * size.y ) );
}
uniform StructuredBuffer<float> _VolBuffer;
float SampleVol(float3 coord3) {
    int3 intIndex3 = UnitVolumeToIntVolume( coord3 );
    int index1D = IntVolumeToLinearIndex( intIndex3, _VolBufferSize.xyz );
    return __VolBuffer[index1D];
}

```

Shading and Gradients

How to shade an volume, such as MRI, for useful visualization. The primary method is to have an 'intensity window' (a min and max) that you want to see intensities within, and simply scale into that space to see the black and white intensity. A 'color ramp' can then be applied to the values within that range, and stored as a texture, so that different parts of the intensity spectrum can be shaded different colors:

```

float4 ShadeVol( float intensity ) {
    float unitIntensity = saturate( intensity - IntensityMin / ( IntensityMax - IntensityMin ) );
    // Simple two point black and white intensity:
    color.rgba = unitIntensity;
    // Color ramp method:
    color.rgba = tex2d( ColorRampTexture, float2( unitIntensity, 0 ) );
}

```

In many of our applications we store in our volume both a raw intensity value and a 'segmentation index' (to segment different parts such as skin and bone, these segments are generally created by experts in dedicated tools). This can be combined with the approach above to put a different color, or even different color ramp for each segment index:

```
// Change color to match segment index (fade each segment towards black):
color.rgb = SegmentColors[ segment_index ] * color.a; // brighter alpha gives brighter color
```

Volume Slicing in a Shader

A great first step is to create a "slicing plane" that can move through the volume, 'slicing it', and how the scan values at each point. This assumes that there is a 'VolumeSpace' cube, which represents where the volume is in world space, that can be used as a reference for placing the points:

```
// In the vertex shader:
float4 worldPos = mul(_Object2World, float4(input.vertex.xyz, 1));
float4 volSpace = mul(_WorldToVolume, float4(worldPos, 1));
```

```
// In the pixel shader:
float4 color = ShadeVol( SampleVol( volSpace ) );
```

Volume Tracing in Shaders

How to use the GPU to do sub-volume tracing (walks a few voxels deep then layers on the data from back to front):

```
float4 AlphaBlend(float4 dst, float4 src) {
    float4 res = (src * src.a) + (dst - dst * src.a);
    res.a = src.a + (dst.a - dst.a*src.a);
    return res;
}
float4 volTraceSubVolume(float3 objPosStart, float3 cameraPosVolSpace) {
    float maxDepth = 0.15; // depth in volume space, customize!!!
    float numLoops = 10; // can be 400 on nice PC
    float4 curColor = float4(0, 0, 0, 0);
    // Figure out front and back volume coords to walk through:
    float3 frontCoord = objPosStart;
    float3 backCoord = frontCoord + (normalize(cameraPosVolSpace - objPosStart) * maxDepth);
    float3 stepCoord = (frontCoord - backCoord) / numLoops;
    float3 curCoord = backCoord;
    // Add per-pixel random offset, avoids layer aliasing:
    curCoord += stepCoord * RandomFromPositionFast(objPosStart);
    // Walk from back to front (to make front appear in-front of back):
    for (float i = 0; i < numLoops; i++) {
        float intensity = SampleVol(curCoord);
        float4 shaded = ShadeVol(intensity);
        curColor = AlphaBlend(curColor, shaded);
        curCoord += stepCoord;
    }
    return curColor;
}
```

```
// In the vertex shader:
float4 worldPos = mul(_Object2World, float4(input.vertex.xyz, 1));
float4 volSpace = mul(_WorldToVolume, float4(worldPos.xyz, 1));
float4 cameraInVolSpace = mul(_WorldToVolume, float4(_WorldSpaceCameraPos.xyz, 1));
```

```
// In the pixel shader:  
float4 color = volTraceSubVolume( volSpace, cameraInVolSpace );
```

Whole Volume Rendering

Modifying the sub-volume code above we get:

```
float4 volTraceSubVolume(float3 objPosStart, float3 cameraPosVolSpace) {  
    float maxDepth = 1.73; // sqrt(3), max distance from point on cube to any other point on cube  
    int maxSamples = 400; // just in case, keep this value within bounds  
    // not shown: trim front and back positions to both be within the cube  
    int distanceInVoxels = length(UnitVolumeToIntVolume(frontPos - backPos)); // measure distance in voxels  
    int numLoops = min( distanceInVoxels, maxSamples ); // put a min on the voxels to sample
```

Mixed Resolution Scene Rendering

How to render a part of the scene with a low resolution and put it back in place:

1. Setup two off-screen cameras, one to follow each eye that update each frame
2. Setup two low-resolution render targets (say 200x200 each), that the cameras render into
3. Setup a quad that moves in front of the user

Each Frame:

1. Draw the render targets for each eye at low-resolution (volume data, expensive shaders, etc.)
2. Draw the scene normally as full resolution (meshes, UI, etc.)
3. Draw a quad in front of the user, over the scene, and project the low-res renders onto that.
4. Result: visual combination of full-resolution elements with low-resolution but high-density volume data.

Hologram stability

11/6/2018 • 13 minutes to read • [Edit Online](#)

To achieve stable holograms, HoloLens has a built-in image stabilization pipeline. The stabilization pipeline works automatically in the background, so there are no extra steps required to enable it. However, developers should exercise techniques that improve hologram stability and avoid scenarios that reduce stability.

Hologram quality terminology

The quality of holograms is a result of good environment and good app development. Apps that hit a constant 60 frames-per-second in an environment where HoloLens can track the surroundings will ensure the hologram and the matching coordinate system are in sync. From a user's perspective, holograms that are meant to be stationary will not move relative to the environment.

When environment issues, inconsistent or low rendering rates, or other app problems show up, the following terminology is helpful in identifying the problem.

- **Accuracy.** Once the hologram is world-locked and placed in the real world, it should stay where it was placed, relative to the surrounding environment, independent of user motion or small and sparse environment changes. If a hologram later appears in an unexpected location, it is an *accuracy* problem. Such scenarios can happen if two distinct rooms look identical.
- **Jitter.** Users observe this as high frequency shaking of a hologram. This can happen when tracking of the environment degrades. For users, the solution is running [sensor tuning](#).
- **Judder.** Low rendering frequencies result in uneven motion and double images of holograms. This is especially noticeable in holograms with motion. Developers need to maintain a [constant 60 FPS](#).
- **Drift.** Users see this as hologram appears to move away from where it was originally placed. This happens when holograms are placed far away from [spatial anchors](#), particularly in parts of the environment that have not been fully mapped. Creating holograms close to spatial anchors lowers the likelihood of drift.
- **Jumpiness.** When a hologram "pops" or "jumps" away from its location occasionally. This can occur as tracking adjusts holograms to match updated understanding of your environment.
- **Swim.** When a hologram appears to sway corresponding to the motion of the user's head. This occurs when holograms are not on the [stabilization plane](#), and if the HoloLens is not [calibrated](#) for the current user. The user can rerun the [calibration](#) application to fix this. Developers can update the stabilization plane to further enhance stability.
- **Color separation.** The displays in HoloLens are a color sequential display, which flash color channels of red-green-blue-green at 60Hz (individual color fields are shown at 240Hz). Whenever a user tracks a moving hologram with his or her eyes, that hologram's leading and trailing edges separate in their constituent colors, producing a rainbow effect. The degree of separation is dependent upon the speed of the hologram. In some rarer cases, moving ones head rapidly while looking at a stationary hologram can also result in a rainbow effect. This is called [color separation](#).

Frame rate

Frame rate is the first pillar of hologram stability. For holograms to appear stable in the world, each image presented to the user must have the holograms drawn in the correct spot. The displays on HoloLens refresh 240 times a second, showing four separate color fields for each newly rendered image, resulting in a user experience of 60 FPS (frames per second). To provide the best experience possible, application developers must maintain 60 FPS, which translates to consistently providing a new image to the operating system every 16 milliseconds.

60 FPS To draw holograms to look like they're sitting in the real world, HoloLens needs to render images from the user's position. Since image rendering takes time, HoloLens predicts where a user's head will be when the images are shown in the displays. This prediction algorithm is an approximation. HoloLens has hardware that adjusts the rendered image to account for the discrepancy between the predicted head position and the actual head position. This makes the image the user sees appear as if it was rendered from the correct location, and holograms feel stable. The image updates work best with small changes, and it can't completely fix certain things in the rendered image like motion-parallax.

By rendering at 60 FPS, you are doing three things to help make stable holograms:

1. Minimizing the overall latency between rendering an image and that image being seen by the user. In an engine with a game thread and a render thread running in lockstep, running at 30FPS can add 33.3ms of extra latency. By reducing latency, this decreases prediction error, and increases hologram stability.
2. Making it so every image reaching the user's eyes have a consistent amount of latency. If you render at 30fps, the display still displays images at 60 FPS. This means the same image will be displayed twice in a row. The second frame will have 16.6ms more latency than the first frame and will have to correct a more pronounced amount of error. This inconsistency in error magnitude can cause unwanted 60hz judder.
3. Reducing the appearance of judder, which is characterized by uneven motion and double images. Faster hologram motion and lower render rates are associated with more pronounced judder. Therefore, striving to maintain 60 FPS at all times will help avoid judder for a given moving hologram.

Frame-rate consistency Frame rate consistency is as important as a high frames-per-second. Occasionally dropped frames are inevitable for any content-rich application, and the HoloLens implements some sophisticated algorithms to recover from occasional glitches. However, a constantly fluctuating framerate is a lot more noticeable to a user than running consistently at lower frame rates. For example, an application that renders smoothly for 5 frames (60 FPS for the duration of these 5 frames) and then drops every other frame for the next 10 frames (30 FPS for the duration of these 10 frames) will appear more unstable than an application that consistently renders at 30 FPS.

On a related note, the operating system will throttle applications down to 30 FPS when [mixed reality capture](#) is running.

Performance analysis There are a variety of tools that can be used to benchmark your application frame rate such as:

- GPUView
- Visual Studio Graphics Debugger
- Profilers built into 3D engines such as Unity

Hologram render distances

The human visual system integrates multiple distance-dependent signals when it fixates and focuses on an object.

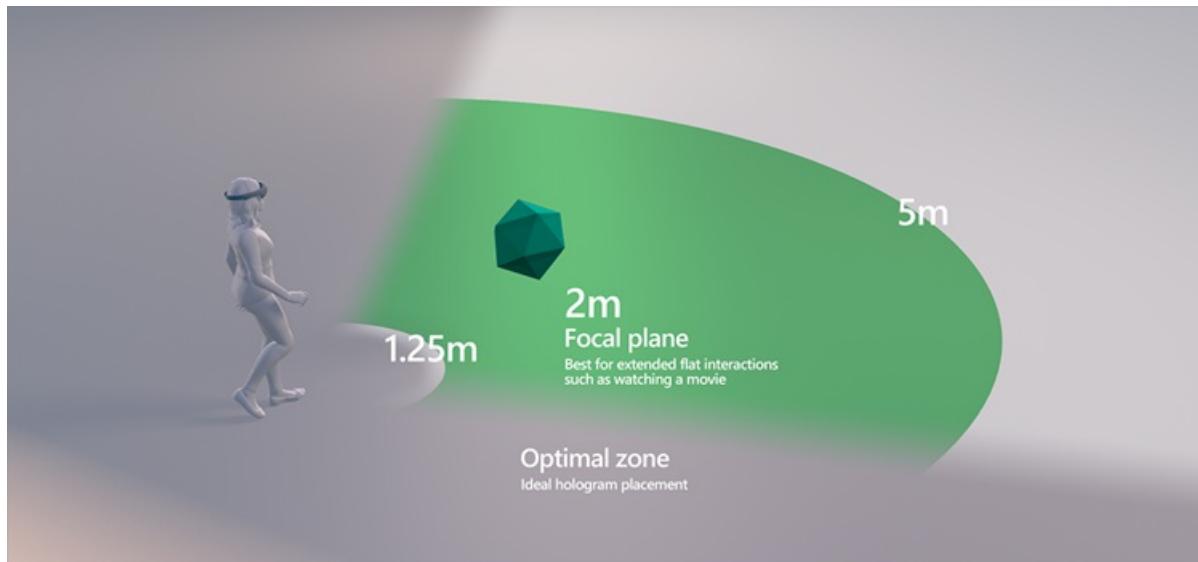
- [Accommodation](#) - The focus of an individual eye.
- [Convergence](#) - Two eyes moving inward or outward to center on an object.
- [Binocular vision](#) - Disparities between the left- and right-eye images that are dependent on an object's distance away from your fixation point.
- Shading, relative angular size, and other monocular (single eye) cues.

Convergence and accommodation are unique because they are extra-retinal cues related to how the eyes change to perceive objects at different distances. In natural viewing, convergence and accommodation are linked. When the eyes view something near (e.g. your nose) the eyes cross and accommodate to a near point. When the eyes

view something at infinity the eyes become parallel and the eye accommodates to infinity. Users wearing HoloLens will always accommodate to 2.0m to maintain a clear image because the HoloLens displays are fixed at an optical distance approximately 2.0m away from the user. App developers control where users' eyes converge by placing content and holograms at various depths. When users accommodate and converge to different distances, the natural link between the two cues are broken and this can lead to visual discomfort or fatigue, especially when the magnitude of the conflict is large. Discomfort from the vergence-accommodation conflict can be avoided or minimized by keeping content that users converge to as close to 2.0m as possible (i.e. in a scene with lots of depth place the areas of interest near 2.0m when possible). When content cannot be placed near 2.0m discomfort from the vergence-accommodation conflict is greatest when user's gaze back and forth between different distances. In other words, it is much more comfortable to look at a stationary hologram that stays 50cm away than to look at a hologram 50cm away that moves toward and away from you over time.

Placing content at 2.0m is also advantageous because the two displays are designed to fully overlap at this distance. For images placed off this plane, as they move off the side of the holographic frame they will disappear from one display while still being visible on the other. This binocular rivalry can be disruptive to the depth perception of the hologram.

Optimal distance for placing holograms from the user



Clip Planes For maximum comfort we recommend clipping render distance at 85cm with fadeout of content starting at 1m. In applications where holograms and users are both stationary holograms can be viewed comfortably as near as 50cm. In those cases, applications should place a clip plane no closer than 30cm and fade out should start at least 10cm away from the clip plane. Whenever content is closer than 85cm it is important to ensure that users do not frequently move closer or farther from holograms or that holograms do not frequently move closer to or farther from the user as these situations are most likely to cause discomfort from the vergence-accommodation conflict. Content should be designed to minimize the need for interaction closer than 85cm from the user, but when content must be rendered closer than 85cm a good rule of thumb for developers is to design scenarios where users and/or holograms do not move in depth more than 25% of the time.

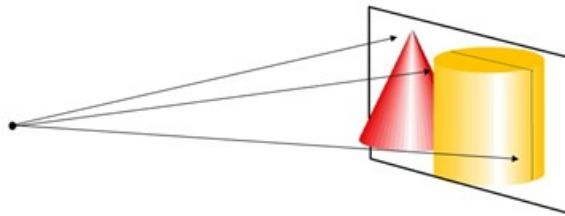
Best practices When holograms cannot be placed at 2m and conflicts between convergence and accommodation cannot be avoided, the optimal zone for hologram placement is between 1.25m and 5m. In every case, designers should structure content to encourage users to interact 1+ m away (e.g. adjust content size and default placement parameters).

Stabilization plane

NOTE

For desktop immersive headsets, setting a stabilization plane is usually counter-productive, as it offers less visual quality than providing your app's depth buffer to the system to enable per-pixel depth-based reprojection. Unless running on a HoloLens, you should generally avoid setting the stabilization plane.

HoloLens performs a sophisticated hardware-assisted holographic stabilization technique. This is largely automatic and has to do with motion and change of the point of view (CameraPose) as the scene animates and the user moves their head. A single plane, called the stabilization plane, is chosen to maximize this stabilization. While all holograms in the scene receive some stabilization, holograms in the stabilization plane receive the maximum hardware stabilization.



Stabilization Plane for 3-D Scene Objects

The device will automatically attempt to choose this plane, but the application can assist in this process by selecting the focus point in the scene. Unity apps running on a HoloLens should choose the best focus point based on your scene and pass this into [SetFocusPoint\(\)](#). An example of setting the focus point in DirectX is included in the default spinning cube template.

Note that when your Unity app runs on an immersive headset connected to a desktop PC, Unity will submit your depth buffer to Windows to enable per-pixel reprojection, which will usually provide even better image quality without explicit work by the app. If you provide a Focus Point, that will override the per-pixel reprojection, so you should only do so when your app is running on a HoloLens.

```
// SetFocusPoint informs the system about a specific point in your scene to
// prioritize for image stabilization. The focus point is set independently
// for each holographic camera.
// You should set the focus point near the content that the user is looking at.
// In this example, we put the focus point at the center of the sample hologram,
// since that is the only hologram available for the user to focus on.
// You can also set the relative velocity and facing of that content; the sample
// hologram is at a fixed point so we only need to indicate its position.
renderingParameters.SetFocusPoint(
    currentCoordinateSystem,
    spinningCubeRenderer.Position
);
```

Placement of the focus point largely depends on the hologram is looking at. The app has the gaze vector for reference and the app designer knows what content they want the user to observe.

The single most important thing a developer can do to stabilize holograms is to render at 60 FPS. Dropping below 60 FPS will dramatically reduce hologram stability, regardless of the stabilization plane optimization.

Best practices There is no universal way to set up the stabilization plane and it is app-specific, so the main recommendation is to experiment and see what works best for your scenarios. However, try to align the stabilization plane with as much content as possible because all the content on this plane is perfectly stabilized.

For example:

- If you have only planar content (reading app, video playback app), align the stabilization plane with the plane that has your content.

- If there are 3 small spheres that are world-locked, make the stabilization plane "cut" though the centers of all the spheres that are currently in the user's view.
- If your scene has content at substantially different depths, favor further objects.
- Make sure to adjust the stabilization point every frame to coincide with the hologram the user is looking at

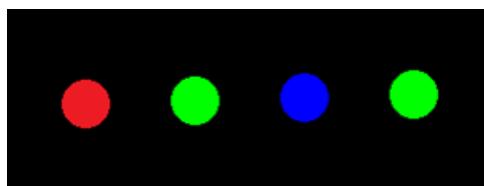
Things to Avoid The stabilization plane is a great tool to achieve stable holograms, but if misused it can result in severe image instability.

- Don't "fire and forget", since you can end up with the stabilization plane behind the user or attached to an object that is no longer in the user's view. Ensure the stabilization plane normal is set opposite camera-forward (e.g. -camera.forward)
- Don't rapidly change the stabilization plane back and forth between extremes
- Don't leave the stabilization plane set to a fixed distance/orientation
- Don't let the stabilization plane cut through the user
- Don't set the focus point when running on a desktop PC rather than a HoloLens, and instead rely on per-pixel depth-based reprojection.

Color separation

Due to the nature of HoloLens displays, an artifact called "color-separation" can sometimes be perceived. It manifests as the image separating into individual base colors - red, green and blue. The artifact can be especially visible when displaying white objects, since they have large amounts of red, green and blue. It is most pronounced when a user visually tracks a hologram that is moving across the holographic frame at high speed. Another way the artifact can manifest is warping/deformation of objects. If an object has high contrast and/or pure colors (red, green, blue), color-separation will be perceived as warping of different parts of the object.

Example of what the color separation of a head-locked white round cursor could look like as a user rotates their head to the side:



Though it's difficult to completely avoid color separation, there are several techniques available to mitigate it.

Color-separation can be seen on:

- Objects that are moving quickly, including head-locked objects such as the [cursor](#).
- Objects that are substantially far from the [stabilization plane](#).

To attenuate the effects of color-separation:

- Make the object lag the user's gaze. It should appear as if it has some inertia and is attached to the gaze "on springs". This slows the cursor (reducing separation distance) and puts it behind the user's likely gaze point. So long as it quickly catches up when the user stops shifting their gaze it feels quite natural.
- If you do want to move a hologram, try to keep its movement speed below 5 degrees/second if you anticipate that the user will follow it with their eyes.
- Use *light* instead of *geometry* for the cursor. A source of virtual illumination attached to the gaze will be perceived as an interactive pointer but will not cause color-separation.
- Adjust the stabilization plane to match the holograms the user is gazing at.
- Make the object red, green or blue.
- Switch to a blurred version of the content. For example, a round white cursor could be change to a slightly blurred line oriented in the direction of motion.

As before, rendering at 60 FPS and setting the stabilization plane are the most important techniques for hologram stability. If facing noticeable color separation, first make sure the frame rate meets expectations.

See also

- [Performance recommendations for HoloLens apps](#)
- [Color, light and materials](#)
- [Interaction fundamentals](#)

Performance recommendations for HoloLens apps

11/6/2018 • 10 minutes to read • [Edit Online](#)

HoloLens is capable of presenting an immersive holographic experience to the user without being tethered to a PC, phone or external cameras. To accomplish this, HoloLens has more compute power than the average laptop with passive cooling. The passive cooling means that there is no fan and no noise that disturbs the user's experience.

Sensor processing is offloaded to the holographic processing unit (HPU) specifically designed to process camera data captured by HoloLens to ensure stable hologram presentation and real time tracking of gaze and gesture inputs. Off-loading sensor fusion to the HPU frees up the host processor for applications while also ensuring the real time requirements of the tracking infrastructure.

Developing applications for HoloLens is different from developing typical desktop applications because the user's view has to be quickly updated as she moves around in her world. Each eye has to be presented with independent visual frames to simulate the appearance of holographic objects in accordance with the laws of physics. As the user moves around, the visual representation has to be updated with minimal latency to avoid objects shifting relative to their real world counterparts. Because of this, the rendering pipeline of a holographic application is very similar to the rendering pipeline of a 3D game from the first person point of view. Note that forward rendering is cheaper on mobile chip sets when compared to the deferred pipeline.

Performance guidelines

Holographic applications need to balance visual complexity, presentation frame rate, rendering latency, input latency, power consumption for thermals, and battery life to achieve an optimal immersive experience. For user comfort, it is important to achieve consistent and high frame rates with minimal latency. Applications should target 60 frames per second (fps) with 1 frame latency. The HoloLens display pipeline is able to upscale the application frame rate to match the display's native frame rate by making small corrections to the presented frames based on very high frequency tracking information. This is critical to achieving stable holograms that behave like real world objects in the user's holographic environment.

Once the application achieves its target display rate, it is important to monitor power consumption and to ensure that the application operates within the designed thermal and power envelope of HoloLens.

Both frame rate and power consumption can be observed in the HoloLens Device Management System Performance view.

Essential Performance Targets

METRIC	TARGET
Frame Rate	60 fps
Power consumption	1-minute average in orange and green area. See System Performance Tool information.
Memory	< 900 MB Total Commit

Tip: If an application is not achieving close to 60 fps, power consumption can be misleadingly low. Therefore, ensure that the application achieves close to target frame rates before taking power measurements.

Development process guidance

Because of the real time and low latency rendering requirements of HoloLens it is critical to track performance from the very beginning of the development process. The general guidance is to achieve [60 fps](#) as early in the development cycle as possible. Once the application achieves 60 fps, the focus shifts to monitoring power consumption while maintaining frame rate.

When measuring and optimizing power, it can be very helpful if the application supports a scripted walk through that can be used to produce repeatable power and performance measurements.

Content guidance

This section provides broad recommendations for application, especially visual complexity. They are meant to be a set of guidelines and not hard limits. In many cases it is possible to trade off performance between different effects or features.

General

- Rendering pixels is often the bottleneck, so avoid drawing too many pixels or using shaders that are too expensive.
 - Limit overdraw. Aim for no more than 1x - 1.5x overdraw.
 - Limit memory bandwidth wherever possible by reducing overdraw, geometry and texture samples. Use mip-maps wherever possible.
 - Consider trading off resolution to allow for more complex visuals and geometry.
 - Avoid full screen effects such as FXAA and SSAO.
- Batch draw calls and use instancing to minimize DirectX overhead.
- Race to sleep on CPU/GPU by aligning tasks with the Present intervals.
- Use 16bit depth buffers.
- Turn off physics if your project doesn't use it.

Geometry

- Perform frustum culling against the combined left and right eye frustum.
- Relax your culling frustum by about 5 degrees to account for some head movement.
- Use last frame's head pose for the culling and grab a fresh [HolographicFrame](#) as late as possible for actual rendering.
- Tune your model complexity and create simplified meshes with lower Level Of Detail for holograms that are far away.
- Measure your scene using the Visual Studio graphics tools to see if your bottleneck is the Vertex Shader or the Pixel Shader and optimize as appropriate.
- Sub-pixel triangles are very expensive.
- Draw static geometry front to back to reject occluded pixels.
- Avoid Geometry, Hull and Compute Shaders.
- Use the SpatialMapping mesh to occlude holograms, cull this mesh to the part that is actually in view.
- Tune the number of triangles you want to receive from SpatialMapping against the holographic content you want to render.
- If you prefill your scene with the occlusion mesh, only render your occlusion mesh to the depth buffer, do not bind it to your color buffer to avoid writes.
- If you want to post process your scene and 'kill' pixels which should be occluded, set a stencil bit to mark the pixels holograms are rendered to and honor this stencil when rendering your occlusion mesh.
- Use render queries to cull objects which are fully occluded by other holograms or the spatial map of the world.

- Set your far plane to the furthest point of the furthest in view hologram to reject as much of your occlusion geometry as possible.
- Reducing the number of influences on a skeletal vert to 1 or 2 can greatly reduce skeletal deformation overhead and save some frame rate/power.

Shaders

- Tools often create generic Shaders that are too complex. This problem can be avoided by hand-writing Shaders in HLSL.
- If your GPU usage is high and the Visual Studio graphics tools indicate your draw calls are expensive there are several steps to reduce their cost.
- To easily distinguish a vertex bottleneck from a pixel bottleneck, use the ViewportScaleFactor to reduce your output resolution. If you see a performance gain, you have a pixel bottleneck, if you do not see a gain, you likely have a vertex shader bottleneck.
- To distinguish a vertex shader bottleneck from a CPU bottleneck, create a minimal vertex shader that still performs all the positional calculations but simply emits a white color. If you see a performance gain, you have a vertex shader bottle neck, if performance remains the same, your problem is likely on the CPU and probably related to having too many DirectX calls.
- Shader switching between holograms can be expensive, consider sorting your draw calls such that you submit all objects with the same shader in sequence.
- Use full floats for vertex position calculations but use [min16float](#) for all other calculations. Check the shader assembly to ensure that it is using 16-bit. It is easy to make simple mistakes that force 32-bit mode. HLSL compiler will give you a warning if it up-converts to 32-bit.
- Leverage "low level" HLSL intrinsic instructions to maximize ALU throughput. [HLSL intrinsic reference](#).
- Combine separate MULTIPLY and ADD instructions into a single [MAD](#) instruction when possible.

Vertex Shader

- Only perform vertex specific calculations in the vertex shader. Move other computations into the constant buffer or additional vertex attributes.
- You can trade off vertex shader instruction count against the total number of vertices on screen.

Pixel Shader

- For objects which cover a large part of the view, shade each pixel only once when possible.
- Limit to approximately 48 Pixel Shader arithmetic operations based on compiler output.
- Shader cost is often directly proportional to the screen space an object takes up. E.g. one wouldn't want to put a 30 instruction shader on SR, which could take up the full screen.
- Move work from Pixel Shader to Vertex Shader when possible. E.g. move lighting into Vertex Shader and interpolate.
- Limit to 1 texture sample.
- Static branching can be helpful. Check compiler output that it is being picked up.
- Dynamic branching can be useful if you can reject a large number of complex pixels. E.g. shading objects where most pixels are transparent.

Texture sampling

- Use bilinear whenever possible.
- Use trilinear sparingly.
- Anisotropic filtering should only be used when absolutely necessary.
- Use DXT compression and mip-maps when possible.
- Use 16-bit texture bandwidth.
- If Pixel Shader is texture read bound, ensure texture read instructions are at beginning of shader code. The

GPU does not support prefetching of texture data.

Tools

TOOL	METRICS
HoloLens Device Portal Performance Tools	Power, Memory, CPU, GPU, FPS
Visual Studio Graphics Debugger	GPU, Shaders, Graphics Performance
Visual Studio Diagnostic Tools	Memory, CPU
Windows Performance Analyzer	Memory, CPU, GPU, FPS

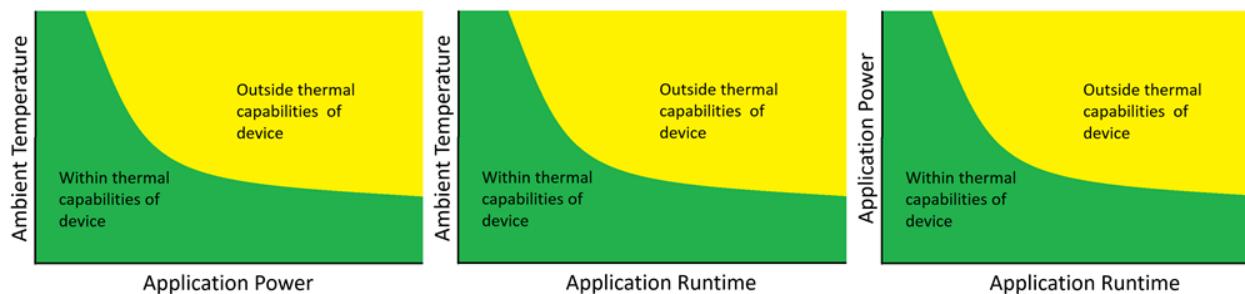
Windows Device Portal - Performance

The HoloLens Device Portal offers the following performance tools.

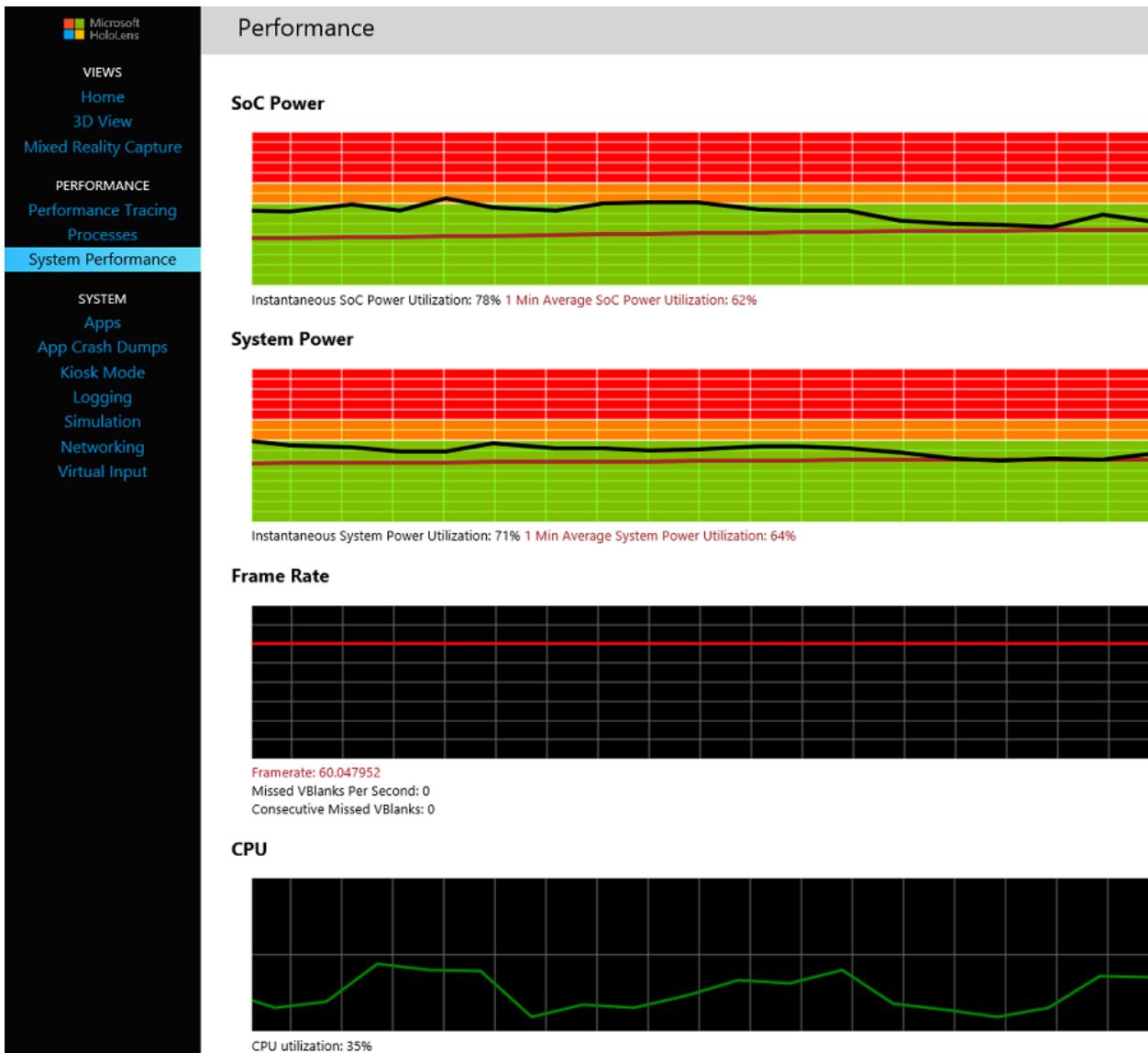
- Performance Tracing - this page offers the ability to capture performance data and analyze it with the Windows Performance Analyzer.
- Processes - shows currently running processes and related process information.
- System Performance - for real time tracking of power, cpu utilization, gpu utilization, disk I/O, memory, frame rate, and several other metrics.

System performance

HoloLens is designed to work across a range of ambient temperature environments. If HoloLens exceeds its thermal capabilities, the foreground application will be shut down to allow the device to cool off. Thermals are dependent on three variables: ambient temperature, power consumption of the experience, and the amount of time the experience is in use. The relationship between the three variables and HoloLens' thermal capabilities is shown below.



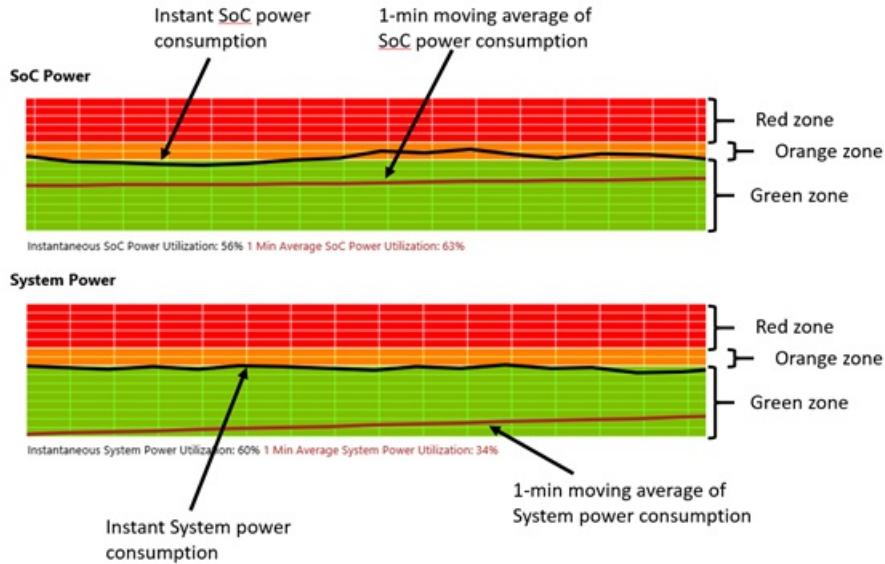
The System Performance page provides real time performance data for SoC power consumption, system power consumption, device frame rate, CPU and GPU utilization, disk I/O, networking bandwidth, and memory utilization.



System power System power shows the total power consumption for HoloLens. This includes active sensors, displays, speakers, GPU, CPU, memory, and all other components. HoloLens cannot measure the system power consumption when it is being charged. System power consumption will therefore drop to zero while charging.

SoC power SoC power is the combined power consumption of CPU, GPU, and memory. CPU and GPU utilization can be used to further understand which of the two is a major power contributor. It should also be kept in mind that memory transfers require significant amounts of power, hence optimizing memory bandwidth can be used to further reduce SoC power consumption.

Each graph, SoC and System power, show current power consumption and power consumption averaged over 1 minute. Brief high power loads are acceptable, e.g. during application launch.



Green zone: Application power consumption is well within the device's capabilities. **Orange zone:** Applications operating in the orange zone is acceptable even in elevated ambient temperature environments. **Red zone:** Applications operating in the red zone might be acceptable, especially in cooler environments such as Offices or typical living environments.

Frame Rate

Shows the frame rate of the rendering or composition layer. When a holographic exclusive application is running, it is in full control of the display stack, and the frame rate shown, is the frame rate of the application. This counter cannot be used to measure the frame rate of traditional 2D store applications that are pinned in the Holographic shell. To measure the frame rate of your traditional 2D application in the Holographic shell, the app needs to display the frame rate itself.

CPU

The percentage of time the CPU cores are running a thread other than the Idle thread. 100% represents full utilization of all processor cores.

GPU

The percentage of time the GPU is active with graphics or compute tasks.

I/O

Read and write disk bandwidth for all processes.

Network

Network send and receive bandwidth for all processes.

Memory

The memory limit on HoloLens for an application is 900MB. If an application exceeds that limit, resource manager will terminate the application.

The memory graph in the System Performance section shows the total memory committed by all processes by the device. For individual process memory information, see the Processes tab.

See also

- [Performance recommendations for immersive headset apps](#)

Performance recommendations for immersive headset apps

11/6/2018 • 18 minutes to read • [Edit Online](#)

Hardware targets

Windows Mixed Reality Ultra PCs will consist of [desktops and laptops with discrete graphics \(plus additional requirements\)](#) and will support experiences at 90Hz.

Windows Mixed Reality PCs will consist of [desktops and laptops with integrated graphics \(plus additional requirements\)](#) and will support experiences at 60Hz. To more easily distinguish the two PC targets, we'll refer to these PCs as "core" (differentiated from "ultra") through the rest of this article.

If you target just Windows Mixed Reality Ultra PCs for your experience, it will have more power at its disposal, but you'll also be limiting your audience. Conversely, if you target Windows Mixed Reality "core" PCs for your experience, you'll have a much larger audience, but won't be offering unique performance value to customers with higher-end Windows Mixed Reality Ultra PCs. Thus, a hybrid approach for your VR experience may be the best of both worlds.

We recommend testing your app on the lowest-end hardware in each category you intend to support. When targeting Windows Mixed Reality Ultra PC's, that would be a PC with an Nvidia GTX 1050 or Radeon RX 460 GPU. Since laptops often have additional performance constraints, we recommend testing with a laptop with one of those GPUs. A list of Windows Mixed Reality PCs and Windows Mixed Reality Ultra PCs you can purchase for testing purposes will be coming soon.

Performance targets

Framerate targets

The target framerate for your VR experience on Windows Mixed Reality immersive headsets will be either 60Hz or 90Hz depending on which [Windows Mixed Reality compatible PCs](#) you wish to support.

For the PC you're currently using, you can determine its target framerate by checking the [holographic frame duration](#), or, in Unity, checking the device's [refresh rate](#).

Optimizing performance for Unity apps

Unity has many resources available with guidance on how to optimize your app:

- [Optimisation for VR in Unity](#)
- [Performance Optimization Tutorial](#)
- [Optimizing Graphics Rendering in Unity Games](#)
- [Optimizing Graphics Performance](#)

Hitting performance targets on Windows Mixed Reality PCs usually has two phases. First, improvement of the overall performance of the app without downgrading the experience. Second, finding performance/design tradeoffs that can be made on lower end machines to hit framerate. These tradeoffs include things like modifying assets, behavior, and quality settings.

Performance toolkit

We have provided a series of tools in the Mixed Reality Toolkit to help find and evaluate these tradeoffs, as well as

provided the scaffolding to allow apps to only make these tradeoffs on machines where they are necessary. You may find these as a [Unity package on GitHub](#). The tools in this package are:

- **Performance Display:** This will overlay the app's current and target framerates, as well as any other information you choose to show. Use this tool to identify scene and computers where your app does not hit framerate and evaluate the performance impact of design tradeoffs. To use this package, find the `PerformanceDisplay` prefab in the prefabs folder and drag it onto your main camera.
- **Adaptive Performance:** This tool is the heart of the toolkit. Use it to define different levels of performance settings and switch between them. In this script you may define performance buckets that may include changes to quality settings, viewport scale, as well as triggers to change things like materials on objects. To see the effect a design tradeoff has on the performance of your app, apply in one of these buckets and use `FPSDisplay` to compare frame rate with and without it.
- **Viewport Scale Manager:** This tool manages the [viewport scale](#) rendered.
- **Quality Manager:** This tool makes it easy to change the [unity quality settings](#) while running the game.
- **Shader Control:** This tool enables apps to use simpler shaders on lower end machines. This script subscribes the `OnPerformanceBucketChanged` event in `AdaptivePerformance.cs` and looks for a bucket value called `ShaderLevel` to determine which material to use. The script must be configured for each object that needs a material change.
- **GPU Whitelist:** This class identifies the GPU manufacturer and type and groups them into loose performance buckets. This can help you to pick a starting performance bucket that is most likely to work out well for users. It also contains a function for reporting machine configuration through unity analytics. We recommend whenever your app switches buckets, it sends this information along with the bucket you are switching into in order to track performance on different systems.
- **Performance Counter:** This tool contains logic to measure the framerate of your app and adapt your settings accordingly. It relies on the adaptive performance tool and requires defined performance buckets. More information is at the very end of [Adapting app quality to each machine](#).

Performance analysis

To do an initial performance test, we recommend having at least the `PerformanceDisplay` and `ViewportScaleManager` imported and working in your project. Then build a version of your app in Unity that is not a developer build. Target 'Master' and 'x64' in your Visual Studio configuration and run the built app on a minimum spec computer without debugging attached. With this build look at how the actual framerate compares with the target framerate. Run through your whole experience this way to see where the biggest changes need to be made. If your framerate is consistently below the target, try to figure out which of the following three places your bottleneck is:

1. **GPU.** The rendering of objects.
2. **App thread on the CPU.** Unity's game logic, including everything called in 'Update', physics and animation.
3. **Render thread on the CPU.** This thread does all the background work queuing work for the GPU.

To figure this out, you may try decreasing the viewport scaling factor. If that improves your framerate, then you are likely GPU bound. Conversely, if decreasing the viewport does not improve your FPS, then you are likely CPU bound. You may use the [Unity Profiler](#) to help understand if you are CPU bound on the render or app thread. The [Windows Performance Analyzer](#) is another tool that is quite good at identifying this which of the three places you are bound on.

GPU bound scenes

There are quite a few things you can do on a GPU bound game. The first step is to start identifying issues with a profiler. Either the [Unity frame debugger](#) or Intel GPA (especially if you have an intel GPU to test on) will work well for this. Some easy things to look for here are:

1. **Reduce the number of full screen passes.**

- **Use single pass instanced rendering.** This might involve modifying some shaders, but should give you a performance boost without reducing graphic quality.
 - **Turn off HDR.** HDR requires an extra texture copy per frame. Disabling this should get you a performance win.
 - **Remove, replace or reduce MSAA.** Anti-aliasing is quite expensive and hardware MSAA can be quite slow on lower end machines. Unity allows you to turn this off or reduce the number of passes in its [quality settings](#) or you can try some software ones such as FXAA in Unity's post effect pipeline.
 - **Use Unity's post effects pipeline.** If you are using more than one post processing effect, you may want to consider removing some or combining them, either manually or with this pipeline.
2. **Reduce poly count.** You may want to create lower poly models for many of your most complicated assets. [Simplygon](#) can automate this. Unity's [LOD System](#) is useful here and can be configured in your [Unity Settings](#)
 3. **Reduce number of particles.** Particle effects can be quite expensive, and sometimes having fewer can still give you the effect you want without reducing framerate.
 4. **Reduce the resolution.** You can take advantage of [dynamic resolution scaling](#) or decide to reduce the [default render target size](#).
 5. **Improve shader performance.** If you find an object with a high render time on its own, you can try swapping out the shader for some of the fast shaders found in the [Mixed Reality Toolkit](#). You can also optimize your shaders yourself. Unity can compile your shaders for you and show you how many operations they call, which can help compare relative speed. To see this, navigate to a shader in your project menu, and in the inspector menu click the "compile and show code" button. A Visual Studio window should open with stats like these:

Shader stats output:

```
// Stats for Vertex shader:  
//      d3d11: 39 math  
// Stats for Fragment shader:  
//      d3d11: 4 math, 1 texture
```

App thread CPU bound scenes

For reducing app thread time, the [Unity Profiler](#) is your friend. Strategies we've found useful are:

1. **Cache data, and avoid expensive calls:**
 - GetComponent
 - FindObjectOfType
 - RaycastAll
 - Anything that traverses the scene graph.
 - Even static accessors, like Vector3.zero, are faster when cached as class variables.
2. **Simplify animations.** Often similar effects can be achieved by simplifying animations, or removing extra ones.
3. **Pool your objects.** [Object pooling](#) can be extremely effective especially if allocations are costing you performance.
4. **Simplify physics.** Reducing the number of iterations can improve performance significantly. Unfortunately, this usually has behavior consequences, so make sure to *test these changes*.
5. **Reduce complexity of tools.** Inverse kinematics and path finding can both be quite expensive, if you are using a complex algorithm like this, try to find simpler algorithms or adjust their settings to improve performance.

Render thread CPU bound scenes

If your game is render thread bound, as shown by either the [Unity Profiler](#) or the [Windows Performance Analyzer](#), try:

1. **Use single pass instanced rendering.** This might involve modifying some shaders, but can roughly cut your render thread time in half, as you no longer need to do two scene traversals.
2. **Follow Unity's excellent guide** on the topic. Starting from the 'Graphics jobs' section, this article has an extensive list of suggestions for this problem.

Once you've improved performance across your app, you can use the same tool to set up adaptive quality and viewport adjustment for your app.

Adapting app quality to each machine

After adding this package to your project, you'll need to predefine a set of performance buckets corresponding to your different quality levels. The set must be ordered in a list starting with the highest performance settings (lower quality, small viewport) and going up to the lowest performance settings (high quality, full viewport). The set of buckets is hard coded as an array field in `AdaptivePerformance.cs` script. You should see something like this:

Adaptive Performance Bucket Set:

```
private PerformanceBucket[] perfBucketList=
{
    new PerformanceBucket()
    {
        QualityLevel = 0,
        ViewportScale = 0.5f,
        ShaderLevel = 1
    },
    new PerformanceBucket()
    {
        QualityLevel = 5,
        ViewportScale = 1f,
        ShaderLevel = 0
    }
};
```

These buckets should correspond to the settings you test and feel work well together, giving you better performance at potentially worse visual quality. Aside from the [viewport scale](#) and the [Unity quality level](#), there is also a 'shader level' example. This demonstrates how the adaptive performance toolkit may be extended beyond the parameters we provide by default.

To use the shader level, you need to add the `ShaderControl` component to the object who's material you want to manage dynamically. In the inspector, add the default material as first entry in the material list. Then add more materials as needed. The first (the default) material is expected to be the most expensive. Materials in the following entries should be computationally less expensive than the preceding materials. The `ShaderControl` subscribes to get callbacks from `AdaptivePerformance` when a performance bucket changes. When a bucket change occurs, if the component has a material with the same level as indicated by the new performance bucket, the said material will be used to render the object, improving performance.

You may add other values to buckets to tune other settings that you find improve performance. For example, you may want a parameter modifying the number of particles generated by particle effects. You could also add a "graphics tier" manager, similar to the quality manager already in the tool that allows you to change to different [graphics tiers](#) dynamically.

Choosing the correct bucket for a machine

Choosing the correct bucket for a given machine has a few steps:

1. **Make an educated guess.** You may use the `GPUWhitelist` tool as a starting place to guess at the best

settings when a user first runs your app.

2. **Adapt to a better bucket.** This tool contains logic for changing buckets automatically if you are missing your framerate target, or if your performing extremely well.
3. **Override with user setting.** The user knows what they want best! We recommend you expose your buckets in your UI and allow the user to pick between them if they desire.
4. **Improve your starting guess.** In the GPUWhitelist class, we provide the `LogSystemInfo` function for sending back telemetry for which bucket each configuration ends up on. We recommend calling this on `AdaptivePerformance.cs`'s `ApplyBucketSettings` event to collect data on the best bucket for a given configuration. You may then use that collected data to improve your initial guess.

By default, the 'adapt to a better bucket' part of this is not running and can be turned on in `AdaptivePerformance.cs`. See [the readme] for more details.

When the adaptive manager is running it will analyze the performance over a period of time before making decision whether to move to a different quality bucket. Currently the adaptive manager supports 2 analyzers:

Frame rate analyzer

The frame rate analyzer measures the FPS over a time sample of half a second. When it accumulates samples over a period of 1 minute, the analyzer will check if at least 80% of the samples meet the target frame rate. If less than that meet the target frame rate, the adaptive manager switches to a lower quality bucket. If we are meeting the target frame rate consistently over 3 minutes, the adaptive manager will try a higher quality bucket. Note that if we had previously switched from a higher quality bucket to a lower quality bucket, the adaptive manager will not attempt a higher quality bucket if frame rate is consistently on target. All of these numbers are tunable constants defined at the top of `FrameRateAdaptiveAnalyzer` in `PerformanceAnalyzers.cs`

GPU render time analyzer

The GPU render time analyzer measures render time on the GPU. If that time exceeds 95% of the target time consistently for 5 frames, the adaptive manager will switch to a lower quality bucket. If the time is consistently less than 75% of the target time over the course of 5 frames, the adaptive manager will try a higher quality bucket. These numbers are tunable constants at the top of the `GpuTimeAdaptiveAnalyzer` class within `PerformanceAnalyzers.cs`.

Note: Using this method has an impact on app performance, so we recommend going with the frame rate based method.

Content guidance

Considerations for Windows Mixed Reality "core" PCs

In order to hit performance goals on [Windows Mixed Reality "core" PCs](#), you may need to reduce your quality settings in Unity, and/or reduce the viewport for those devices. Both of these modifications will have visual fidelity implications, however, low framerate can induce motion sickness in users, and we **strongly** recommend considering hitting the target framerate as a **requirement** for running your game. If you decide that the loss of visual fidelity in your game would be too great on lower spec machines, update your Windows Store description to discourage users with lower specifications from buying your game.

Default render target size

[Windows Mixed Reality immersive headsets](#) contain lenses which distort the presented image to give higher pixel density in the center of view, and lower pixel density in the periphery. In order to have the highest visual fidelity on Windows Mixed Reality Ultra devices, we set the render target's pixel density to match the highly-dense center of the lens area. As this high pixel density is constant across the whole render target, we end up with a higher resolution than the headset's display. By contrast, other VR platforms may default to the render size of the display, which would require you to increase this size to get the correct pixel density in the center of the lensed image. This means that if you keep the default settings, your app may be rendering more pixels compared to

other VR platforms, which might decrease performance but increase visual fidelity. If you have found on other platforms that you need to increase your render scale in order to achieve this high pixel density (in Unity 2017 the line would be something like `UnityEngine.XR.XRSettings.renderScale = 1.0`), you likely will want to remove this logic for our platform as it won't gain you any added visual fidelity, and will cost you performance.

In order to hit the more difficult performance target of Windows Mixed Reality "core" PCs, we also lower the resolution target.

For either sort of device you may want to scale the default resolution target smaller in order to get back some GPU memory and reduce the number of rendered pixels. You may do this by setting the **"Windows.Graphics.Holographic.RenderTargetSizeScaleFactorRequest"** key in the property bag on the **CoreApplication**, however, it needs to be done before you create your holographic space and cannot be changed once you create your holographic space. In order to help you determine what systems might need such a change, we have provided a sample project you may use to get information about the system you are running on [here](#).

If you are running in C++ or with the IL2CPP backend of unity, add the following code to the Initialize function in your app.cpp file:

Sample C++ code for setting render scale with SystemInfoHelper Project:

```
auto holographicDisplay = Windows::Graphics::Holographic::HolographicDisplay::GetDefault();
if (nullptr != holographicDisplay)
{
    double targetRenderScale = 1.0; auto systemInfo = ref new SystemInfoHelper::SystemInfo(holographicDisplay->AdapterId);
    SystemInfoHelper::RenderScaleOverride^ renderScaleOverride = systemInfo->ReadRenderScaleSpinLockSync();
    if (renderScaleOverride == nullptr || renderScaleOverride->MaxVerticalResolution != (int)holographicDisplay->MaxViewportSize.Height)
    {
        if (renderScaleOverride != nullptr)
        {
            // this deletes the file where we read the override values
            // it is async but we shouldn't have to wait for it to finish
            systemInfo->InvalidateRenderScaleAsync();
        }
        /// You may insert logic here to help you determine what your resolution
        /// should be if you don't have one saved. SystemInfoHelper has some
        /// functions that may be useful for this
        if (holographicDisplay->MaxViewportSize.Height < 1300.0)
            // Performance constrained systems that are throttled by the OS will have a
            // max resolution of 1280x1280
        {
            // Set default render scale for performance constrained systems here
            targetRenderScale = 0.8;
        }
    }
    else
    {
        targetRenderScale = renderScaleOverride->RenderScaleValue;
    }
    CoreApplication::Properties->Insert("Windows.Graphics.Holographic.RenderTargetSizeScaleFactorRequest",
    targetRenderScale);
}
```

As this value must be set before you can actually run your program and do any performance evaluation, you may find that you need to adjust its value for the next startup. SystemInfoHelper has the ability to save and load a different value that you find might better suit how your app actually runs on the hardware.

Dynamic resolution scaling

Viewport scaling (dynamic resolution scaling) is the practice of rendering your image to a smaller render target

then your output device can display, and sampling from those pixels to display your final image. It trades visual fidelity for speed. Windows Mixed Reality devices support viewport scaling at a platform level. This means if you set the viewport to be smaller (in Unity: `UnityEngine.XR.XRSettings.renderViewportScale = 0.7f`) Unity will inform the platform it is rendering to a smaller section of the render target, and the platform will composite its display from that smaller section of the render target.

Performance Tools

Visual Studio

[Visual Studio Graphics Diagnostics](#) can debug immersive applications running on Windows Mixed Reality. Please note that GPU Usage is not supported for Window Mixed Reality.

Windows Performance Analyzer

The Windows Performance Analyzer is particularly useful for identifying where you bottleneck is, as well as ensuring that you are taking full advantage of your PC's resources. There are usually three places where your app can be performance constrained: App thread or render thread on the CPU, or GPU. This tool is great for finding out which ones are causing you the biggest issue. There is a great tutorial for using this [here](#).

Unity Performance Profiler

The [Unity Profiler](#) is particularly useful if you are CPU bound, as it will show you how long you are spending in each update function. The most accurate performance measurements will come from profiling a deployed UWP app. To profile on a built UWP app, make sure you have turned on the InternetClient capability build with the developer build checkbox marked. To turn on InternetClient capability, go to Edit > Project Settings > Player, select "Publisher Settings" and under "Capabilities" check "InternetClient". If you already know that you need to improve performance in a given scene, you may use play mode to iterate quickly, and you will likely see proportionate improvements in your UWP solution. If your bottleneck is in the GPU, you can still start with the Unity Profiler and make significant process. You may, for example, isolate which object seems to be causing you the most render issues by turning off all object in the unity hierarchy and turning them on selectively until you find one that takes a particularly ling time to render. Once you have discoved that, you can either try to simplify the object, or improve performance in its shader. The [Mixed Reality Toolkit](#) has some excellent, fast shaders that might be helpful.

Unity Frame Debugger

The [Unity Frame Debugger](#) is useful if you are GPU bound. Pay attention to the draw order and the number of passes your app makes.

Windows Device Portal

The [Windows Device Portal](#) lets you configure and manage your device remotely over a network or USB connection. It also provides [advanced diagnostic tools](#) to help you troubleshoot and view the real time performance of your Windows device.

Intel® Graphics Performance Analyzers (Intel® GPA)

[Intel® Graphics Performance Analyzers \(Intel® GPA\)](#) added support for Windows Mixed Reality on [December 21st 2017](#). The [Getting Started for Windows* Host](#) documentation has more details.

Intel® Power Gadget

[Intel® Power Gadget](#) is a software-based power usage monitoring tool enabled for Intel® Core™ processors (from 2nd Generation up to 6th Generation Intel® Core™ processors), Intel® Atom™ processors not supported. It includes an application, driver, and libraries to monitor and estimate real-time processor package power information in watts using the energy counters in the processor.

See also

- Intel

- VR content developer guide
- How to plan optimizations with Unity
- Render queue ordering in Unity
- Optimizing VR hit game *Space Pirate Trainer*
- Unity
 - Analyzing your game performance using Event Tracing for Windows
- Performance recommendations for HoloLens apps
- Case study - Scaling Datascape across devices with different performance

Performance recommendations for Unity

11/6/2018 • 10 minutes to read • [Edit Online](#)

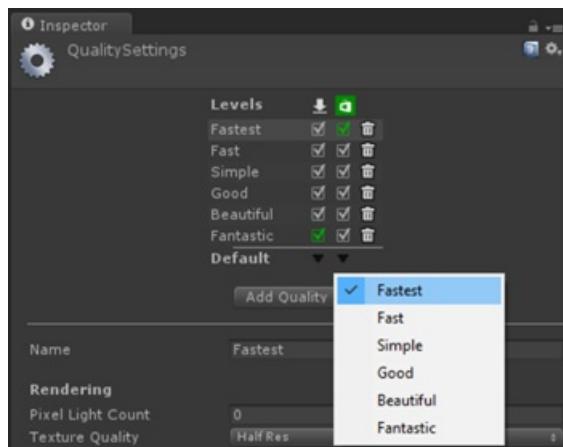
For the most part, the general [performance recommendations for HoloLens apps](#) apply to Unity as well, however there are a few Unity specific things you can do as well.

Unity engine options for power and performance

Unity's defaults lean towards the average case for all platforms, including desktops. Some of the settings need to be tweaked for maximum performance on a device like HoloLens.

Use the fastest quality settings

Unity quality settings



- On the "Edit > Project Settings > Quality" page, select the dropdown under the Windows Store logo and select "Fastest". This ensures that most tunable quality options are set to maximize for performance.

Enable player options to maximize performance

Unity player settings



Go to the player settings by navigating to "Edit > Project Settings > Player" page, click on the "Windows Store", then consider the settings below:

- Use *Shader preloading* and other tricks to optimize [shader load time](#). In particular shader preloading means you won't see any hitches due to runtime shader compilation.
- Make sure "*Rendering > Rendering Path*" is set to *Forward* (this is the default). While deferred rendering is an excellent rendering technique for other platforms, it does eat up a lot of memory bandwidth (and thus power) which makes it unsuitable for mobile devices such as HoloLens.
- The "Use 16-bit Depth Buffers" setting allows you to enable 16-bit depth buffers, which drastically reduces the bandwidth (and thus power) associated with depth buffer traffic. This can be a big power win, but is only applicable for experiences with a small depth range. You should carefully tune your near/far planes to tightly encapsulate your experience so as to not waste any precision here.

Set up the camera for holograms

The Unity Main Camera will have a skybox by default, which is not suitable for HoloLens applications. To set up the camera properly for HoloLens, select the Main Camera then look at the Camera component. Change "Clear Flags" to "Solid Color" then change the background to transparent black (0,0,0,0).

Write efficient shaders

You should minimize the size and number of textures that your shader uses. Especially problematic are so-called "dependent texture reads", where the data in one texture is used to calculate the texture coordinates for another texture.

Unity supports using [min16float](#) for HLSL using the *fixed* alias. On HoloLens hardware this can give a 2x speedup in ALU performance. D3D does not support using min16float for shader constants, so use regular floats there. You *can* (and should) use them in vertex shader outputs and pixel shader inputs. Note, *fixed* works for vector types as well, e.g. *fixed3*. Be careful to not introduce any unintentional conversions to 32 bit floats - these will happen implicitly if you mix and match data types. In particular, be sure to cast shader constants to *fixed* before using them in expressions, and avoid using the float suffix on literals (e.g. write *1.3* instead of *1.3f*) when used in expressions with *fixed* data types.

You should also use regular HLSL optimization techniques, such as trying to rearrange expressions to use the `mad` intrinsics in order to do a multiply and an add at the same time. And of course, precalculate as much as you can on the CPU and pass as constants to the material. Whenever that isn't possible, do as much as you can in the vertex shader. Even for things that vary per-pixel you can sometimes get away with doing parts of the calculation in the vertex shader.

CPU performance

In order to maintain a steady 60 frames per second and keep power utilization down, you have to be careful to write your logic in an efficient way. The biggest factors for CPU performance are:

- Too many objects being rendered (try to keep this under 100 unique *Renderers* or UI elements)
- Expensive updates or too many object updates
- Hitches due to garbage collection
- Expensive graphics settings and shaders (shadows, reflection probes, etc.)

Garbage collections

When writing managed code, thinking about the performance impact of the garbage collector is important. The fact that you have a garbage collector does not mean you don't have to worry about memory management, it just changes how you think about it.

While the GC doesn't typically contribute all that much to overall cycles spent or power usage, it does cause unpredictable spikes that can make you drop several frames in a row which leads to holograms looking "glitchy" or unstable. This is true even for [Generation 0 collection](#). So while guidance for other kinds of apps might be that short-lived allocations are cheap, this is not true for applications that need to update at 60 frames per second.

A key aspect of managing the GC is to plan for it up front. It's usually *much* harder to fix high GC usage after-the-fact than it is to take some extra care up front.

Tips for avoiding allocations. The ideal way to avoid allocations (and thus collections) is to allocate everything you need at the startup of the application and just reuse that data while the app runs. There are plenty of cases where it's not possible to avoid allocations (for example APIs that return allocated objects), so this isn't actually possible in practice, but it's a good goal to shoot for.

There are also some unintuitive reasons why allocations may occur even when you're not directly allocating something, here are some tips to avoid that:

- Do not use LINQ, as it causes heavy allocations.
- Do not use lambdas, as they cause allocations.
- Beware of boxing! A common case for that is passing structs to a method that takes an interface as a parameter. Instead, make the method take the concrete type (by ref) so that it can be passed without allocation.
- Prefer structs to classes whenever you can.
- Default implementations for value equality and GetHashcode uses reflection *in some cases*, which is not only slow but also performs a lot of allocations. Make sure to debug to find out which ones are causing allocations.
- Avoid foreach loops on everything except raw arrays and List. Each call potentially allocates an Enumerator. Prefer regular for loops whenever possible. (See <https://jacksondunstan.com/articles/3805> for more info)

Other garbage collections concerns. Another key concept to be aware of is that GC time is largely proportional to the number of references in the heap. Thus, it's preferable to store data as structs instead of objects. For example, instead of referring to an object by reference, you might allocate a bunch of those types of objects as a shared array of structs and then refer to them by index. This is not as important for long-lived types (e.g. the ones you allocate at startup and keep around for the duration of the application), since they will rapidly move into the oldest generation and stay there (where you hopefully don't have many collections at all), but worth keeping in mind whenever it's easy to do.

Memory utilization is also a key factor for garbage collection performance. Generally speaking, garbage collections gets significantly costlier the less free space you have available. Typically if you're doing a lot of allocations (and thus garbage collecting a lot) you'd want to keep at least half the heap free. One consequence of this is that you might think you're not doing so badly with respect to garbage collection early on, but then as more and more content gets added to your application and free memory gets used up, you could start experiencing significant GC issues all of a sudden.

Consider enabling [Sustained Low Latency](#) mode in the .NET garbage collector. This will cause the GC to try much harder to avoid stopping your application threads, which can reduce the number of pauses. Be aware that this can only take you so far, and will work best if you have a low rate of allocations to start with.

Startup performance

You should consider starting your app with a smaller scene, then using [SceneManager.LoadSceneAsync](#) to load the rest of the scene. This allows your app to get to an interactive state as fast as possible. Be aware that there may be a large CPU spike while the new scene is being activated and that any rendered content might stutter or hitch. One way to work around this is to set the [AsyncOperation.allowSceneActivation](#) property to false on the scene being loaded, wait for the scene to load, clear the screen to black, and then set back to true to complete the scene activation.

Remember that while the startup scene is loading the [holographic splash screen](#) will be displayed to the user.

General CPU performance tips

Aside from Garbage Collection, you also need to be aware of the general CPU cost of updating your scene. Here are a few things that you may consider to keep things efficient:

Basics

- If you have a lot of objects in the scene and/or scripts that do heavy processing, avoid having *Update* functions on every object in your scene. Instead, have just a few higher level "manager" objects with *Update* functions that calls into any other objects that need attention. The specifics of using this approach is highly application-dependent, but you can often skip large numbers of objects at once using higher level logic. For example, AI logic code probably doesn't need to update every single frame, so you could instead store them all in an array and have a higher level manager object update only a small number of them per frame.
- Remember that *FixedUpdate* can be called multiple times per frame. If you use it, make sure to set the physics timestep to be equal to the refresh rate of the application each frame, or use either *Update* or your own update manager instead.
- Avoid any synchronous loading code or other long running operations. On HoloLens it's critical to always update the rendering at 60 frames per second, or you might risk causing comfort issues for the user. For this reason you should make sure that any long running operation is asynchronous.
- Consider caching often-used components. For example, if you often need to access the Rigid Body of an object, just grab it once and reference it with a private variable rather than looking it up each time.
- Avoid the *foreach* construct (except for arrays and List). This will sometimes allocate an *IEnumerable*, and just generally introduce iteration overhead. It's usually much faster to explicitly iterate over a concrete collection type.
- Avoid deep object hierarchies for moving objects. When moving a transform, all of the parent and child transforms also get recomputed. If content moves in the scene in each frame, this cost will add up.
- Disable idle animations by disabling the Animator component (disabling the game object won't have the same effect). Avoid design patterns where an animator sits in a loop setting a value to the same thing. There is considerable overhead for this technique, with no effect on the application.

Advanced Topics

- Optimize for cache coherency. Cache misses are orders of magnitude more expensive than most CPU instructions, so avoiding random jumping through memory can have a *huge* impact on performance. Flat

arrays of compact structs that are processed in order is ideal.

- Avoid interfaces and virtual methods in hot code such as inner loops. Interfaces in particular are *much* slower than a direct call. Virtual functions are not as bad, but still significantly slower than a direct call.
- Looping over a native array with direct indexing is by far the fastest way to process a list of items. List is significantly slower, even worse if you use *foreach*, and even worse if you cast to *IList* first. Note, looping over a collection isn't *that* slow in the first place, so this mainly matters in hot spots of the code.

Performance tools

Other than the tools listed in [Performance recommendations for HoloLens apps](#), check out

- The [Unity Frame Debugger](#)
- The [Unity Profiler](#)
 - **Note:** The Unity profiler will disable some asynchronous rendering resulting in about half of the normal allowed time for CPU and GPU work to maintain framerate. This will appear in the profiler as `Device.Present` taking a long time. Additionally, not all CPU work is shown in the profile such as `WorldAnchor` update calculations.

See also

- [Unity development overview](#)
- [Performance recommendations for HoloLens apps](#)

Using Visual Studio to deploy and debug

11/6/2018 • 5 minutes to read • [Edit Online](#)

Whether you want to use DirectX or Unity to develop your mixed reality app, you will use Visual Studio for debugging and deploying. In this section, you will learn:

- How to deploy applications to your HoloLens or Windows Mixed Reality immersive headset through Visual Studio.
- How to use the HoloLens emulator built in to Visual Studio.
- How to debug mixed reality apps.

Prerequisites

1. See [Install the Tools](#) for installation instructions.
2. Create a new Universal Windows app project in Visual Studio 2015 Update 1 or Visual Studio 2017. C#, C++, and JavaScript projects are all supported. (Or follow the instructions to [create and build an app in Unity](#).)

Enabling Developer Mode

Start by enabling **Developer Mode** on your device so Visual Studio can connect to it.

HoloLens

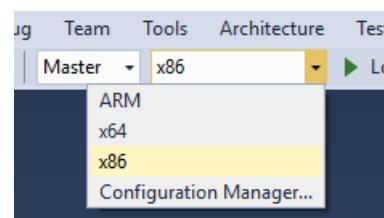
1. Turn on your HoloLens and put on the device.
2. Perform the [bloom](#) gesture to launch the main menu.
3. Gaze at the **Settings** tile and perform the [air-tap](#) gesture. Perform a second air tap to place the app in your environment. The Settings app will launch after you place it.
4. Select the **Update** menu item.
5. Select the **For developers** menu item.
6. Enable **Developer Mode**. This will allow you to [deploy apps from Visual Studio](#) to your HoloLens.
7. Optional: Scroll down and also enable **Device Portal**. This will also allow you to connect to the [Windows Device Portal](#) on your HoloLens from a web browser.

Windows PC

If you are working with a Windows Mixed Reality headset connected to your PC, you must enable **Developer Mode** on the PC.

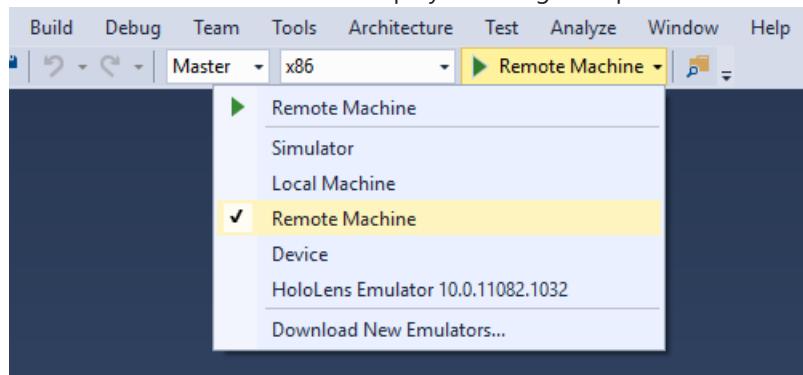
1. Go to **Settings**
2. Select **Update and Security**
3. Select **For developers**
4. Enable **Developer Mode**, read the disclaimer for the setting you chose, then click Yes to accept the change.

Deploying an app over Wi-Fi (HoloLens)



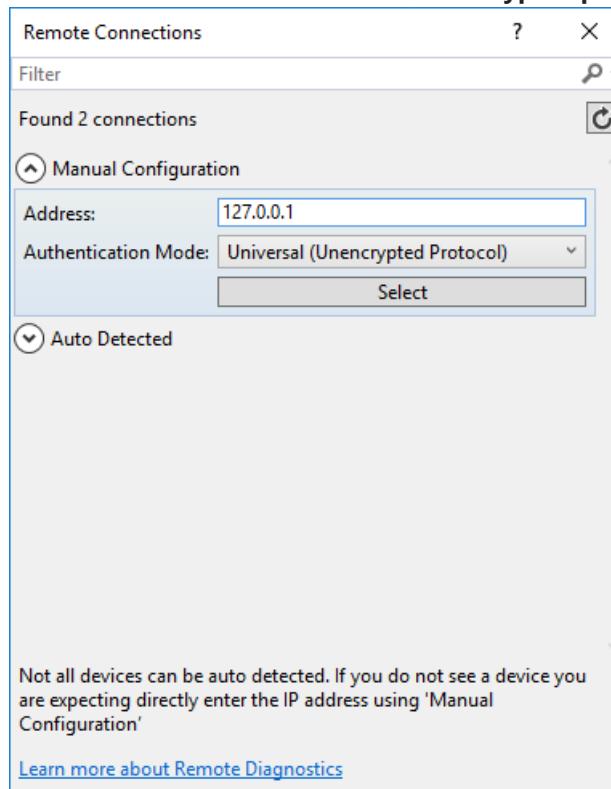
1. Select an **x86** build configuration for your app

2. Select **Remote Machine** in the deployment target drop-down menu

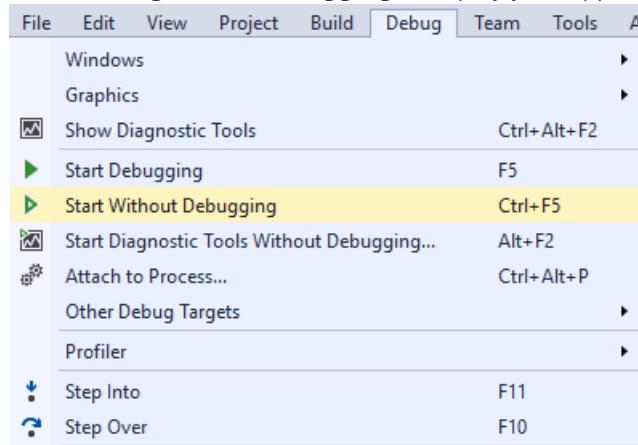


3. For C++ and JavaScript projects, go to **Project > Properties > Configuration Properties > Debugging**.

For C# projects, a dialog will automatically pop-up to configure your connection. a. Enter the IP address of your device in the **Address** or **Machine Name** field. Find the IP address on your HoloLens under **Settings > Network & Internet > Advanced Options**, or you can ask Cortana "What is my IP address?" b. Set Authentication Mode to **Universal (Unencrypted protocol)**



4. Select **Debug > Start debugging** to deploy your app and start debugging

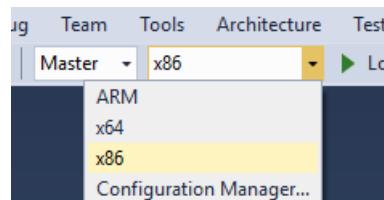


5. The first time you deploy an app to your HoloLens from your PC, you will be prompted for a PIN. Follow the **Pairing your device** instructions below.

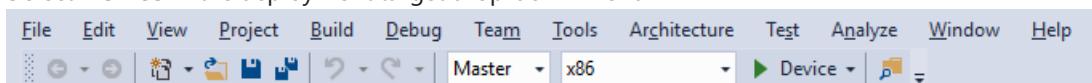
If your HoloLens IP address changes, you can change the IP address of the target machine by going to **Project >**

Properties > Configuration Properties > Debugging

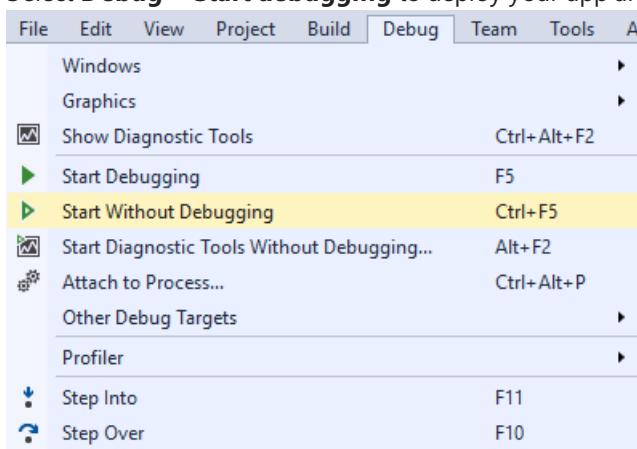
Deploying an app over USB (HoloLens)



1. Select an **x86** build configuration for your app
2. Select **Device** in the deployment target drop-down menu



3. Select **Debug > Start debugging** to deploy your app and start debugging



4. The first time you deploy an app to your HoloLens from your PC, you will be prompted for a PIN. Follow the **Pairing your device** instructions below.

Deploying an app to your Local PC (immersive headset)

Follow these instructions when using a Windows Mixed Reality immersive headset that connects to your PC or the [Mixed Reality simulator](#). In these cases, simply deploy and run your app on the local PC.

1. Select an **x86** or **x64** build configuration for your app
2. Select **Local Machine** in the deployment target drop-down menu
3. Select **Debug > Start debugging** to deploy your app and start debugging

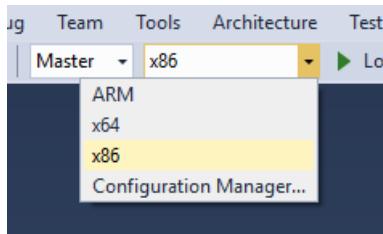
Pairing your device (HoloLens)

The first time you deploy an app from Visual Studio to your HoloLens, you will be prompted for a PIN. On the HoloLens, generate a PIN by launching the Settings app, go to **Update > For Developers** and tap on **Pair**. A PIN will be displayed on your HoloLens; type this PIN in Visual Studio. After pairing is complete, tap **Done** on your HoloLens to dismiss the dialog. This PC is now paired with the HoloLens and you will be able to deploy apps automatically. Repeat these steps for every subsequent PC that is used to deploy apps to your HoloLens.

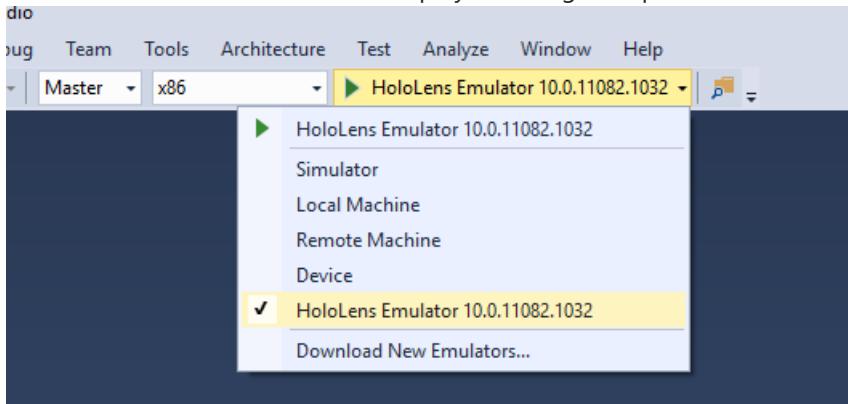
To un-pair your HoloLens from all computers it was paired with, launch the **Settings** app, go to **Update > For Developers** and tap on **Clear**.

Deploying an app to the HoloLens Emulator

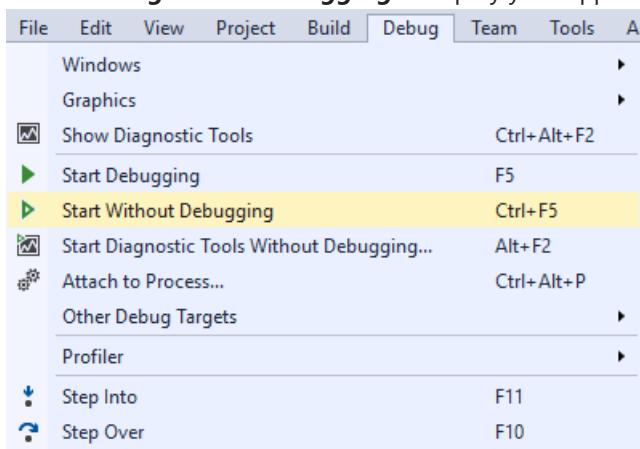
1. Make sure you have [installed the HoloLens Emulator](#).



2. Select an **x86** build configuration for your app.
3. Select **HoloLens Emulator** in the deployment target drop-down menu



4. Select **Debug > Start debugging** to deploy your app and start debugging



Graphics Debugger

The Visual Studio Graphics Diagnostics tools are very helpful when writing and optimizing a Holographic app. See [Visual Studio Graphics Diagnostics on MSDN](#) for full details.

To Start the Graphics Debugger

1. Follow the instructions above to target a device or emulator
2. Go to **Debug > Graphics > Start Diagnostics**
3. The first time you do this with a HoloLens, you may get an "access denied" error. Reboot your HoloLens to allow updated permissions to take effect and try again.

Profiling

The Visual Studio profiling tools allow you to analyze your app's performance and resource use. This includes tools to optimize CPU, memory, graphics, and network use. See [Run diagnostic tools without debugging on MSDN](#) for full details.

To Start the Profiling Tools with HoloLens

1. Follow the instructions above to target a device or emulator
2. Go to **Debug > Start Diagnostic Tools Without Debugging...**
3. Select the tools you want to use

4. Click **Start**
5. The first time you do this with a HoloLens, you may get an "access denied" error. Reboot your HoloLens to allow updated permissions to take effect and try again.

Debugging an installed or running app

You can use Visual Studio to debug a Universal Windows app that's installed without deploying from a Visual Studio project. This is useful if you want to debug an installed app package, or if you want to debug an app that's already running.

1. Go to **Debug -> Other Debug Targets -> Debug Installed App Package**
2. Select the **Remote Machine** target for HoloLens or **Local Machine** for immersive headsets.
3. Enter your device's **IP address**
4. Choose the **Universal** Authentication Mode
5. The window shows both running and inactive apps. Pick the one what you'd like to debug.
6. Choose the type of code to debug (Managed, Native, Mixed)
7. Click **Attach** or **Start**

See also

- [Install the tools](#)
- [Using the HoloLens emulator](#)
- [Deploying and debugging Universal Windows Platform \(UWP\) apps](#)
- [Enable your device for development](#)

Testing your app on HoloLens

11/6/2018 • 7 minutes to read • [Edit Online](#)

Testing HoloLens applications are very similar to testing Windows applications. All the usual areas should be considered (functionality, interoperability, performance, security, reliability, etc.). There are, however, some areas that require special handling or attention to details that are not usually observed in PC or phone apps.

Holographic apps need to run smoothly in a diverse set of environments. They also need to maintain performance and user comfort at all times. Guidance for testing these areas is detailed in this topic.

Performance

Holographic apps need to run smoothly in a diverse set of environments. They also need to maintain performance and user comfort at all times. Performance is so important to the user's experience with a Holographic app that we have an entire topic devoted to it. Please make sure you read and follow the [performance recommendations for HoloLens apps](#).

Testing 3D in 3D

1. **Test your app in as many different spaces as possible.** Try in big rooms, small rooms, bathrooms, kitchens, bedrooms, offices, etc. Also take into consideration rooms with non standard features such as non vertical walls, curved walls, non-horizontal ceilings. Does it work well when transitioning between rooms, floors, going through hallways or stairs?
2. **Test your app in different lighting conditions.** Does it respond properly to different environmental conditions such as lighting, black surfaces, transparent/reflective surfaces such as mirrors, glass walls, etc.
3. **Test your app in different motion conditions.** Put on device and try your scenarios in various states of motion. Does it respond properly to different movement or steady state?
4. **Test how your app works from different angles.** If you have a world locked hologram, what happens if your user walks behind it? What happens if something comes between the user and the hologram? What if the user looks at the hologram from above or below?
5. **Use spatial and audio cues.** Make sure your app uses these to prevent the user from getting lost.
6. **Test your app at different levels of ambient noise.** If you've implemented voice commands, try invoking them with varying levels of ambient noise.
7. **Test your app seated and standing.** Make sure to test from both seating and standing positions.
8. **Test your app from different distances.** Can UI elements be read and interacted with from far away? Does your app react to users getting too close to your holograms?
9. **Test your app against common app bar interactions.** All app tiles and 2D universal apps have a [app bar](#) that allows you to control how the app is positioned in the Mixed World. Make sure clicking Remove terminates your app process gracefully and that the Back button is supported within the context of your 2D universal app. Try scaling and moving your app in [Adjust mode](#) both while it is active, and while it is a suspended app tile.

Environmental Test Matrix

Category	Sub-category	Example
Light	Brightness	Dimly lit room
	Color	Room lit with red light
	Dynamic lighting	Bright flashing light
	Hologram lighting	Spotlight shining on a 3D ball in digital space
Sound	Ambient noise	Loud bus outside on the street
	Ambient conversation	Movie on Xbox in the background
	Spatial audio	Holovideo of a singer walking across a table
User Movement	Head movement	User tilting head and leaning toward a hologram
	Full body movement	User walking from living room to kitchen
	Entering holograms	User sticking head into a hologram
Physical Environment	Clutter	Floor covered with stuffed animals
	Moving items	Door opening
	Distance	User close to or far away from shared world video
	Incomplete scans	Unknown space/objects behind a sofa
	Surface types	Glass coffee table

Comfort

1. **Clip planes.** Be attentive to where [holograms are rendered](#).
2. **Avoid virtual movement inconsistent with actual head movement.** Avoid moving the camera in a way that is not representative of the user's actual motion. If your app requires moving the user through a scene, make the motion predictable, minimize acceleration, and let the user control the movement.
3. **Follow the hologram quality guidelines.** Performant apps that implement the [hologram quality guidance](#) are less likely to result in user discomfort.
4. **Distribute holograms horizontally rather than vertically.** Forcing the user to spend extended periods of time looking up or down can lead to fatigue in the neck.

Input

Gaze and Gestures

[Gaze](#) is a basic form of input on HoloLens that enable users to aim at holograms and the environment. You can visually see where your gaze is targeting based on the cursor position. It's common to associate the gaze cursor with a mouse cursor.

[Gestures](#) are how you interact with holograms, like a mouse click. Most of the time the mouse and touch behaviors are the same, but it's important to understand and validate when they differ.

Validate when your app has a different behavior with mouse and touch. This will identify inconsistencies and help with design decisions to make the experience more natural for users. For example, triggering an action based on hover.

Custom Voice Commands

[Voice input](#) is a natural form of interaction. The user experience can be magical or confusing depending on your choice of commands and how you expose them. As a rule, you should not use system voice commands such as "Select" or "Hey Cortana" as custom commands. Here are a few points to consider:

1. **Avoid using commands that sound similar.** This can potentially trigger the incorrect command.
2. **Choose phonetically rich words when possible.** This will minimize and/or avoid false activations.

Peripherals

Users can interact with your App through [peripherals](#). Apps don't need to do anything special to take advantage of that capability, however there are a couple things worth checking.

1. **Validate custom interactions.** Things like custom keyboard shortcuts for your app.
2. **Validate switching input types.** Attempting to use multiple input methods to complete a task, such as voice, gesture, mouse, and keyboard all in the same scenario.

System Integration

Battery

Test your application without a power source connected to understand how quickly it drains the battery. One can easily understand the battery state by looking at Power LED readings.

Power level	LED state	Legend
<= 20%	○●●●●	○ LED at medium brightness ● LED off
21% - 40%	○○●●●	
41% - 60%	○○○●●	
61% - 80%	○○○○●	
81% - 100%	○○○○○	

Power State Transitions

Validate key scenarios work as expected when transitioning between power states. For example, does the application remain at its original position? Does it correctly persist its state? Does it continue to function as expected?

1. **Stand-by / Resume.** To enter standby, one can press and release the power button immediately. The device also will enter standby automatically after 3 minutes of inactivity. To resume from standby, one can press and release the power button immediately. The device will also resume if you connect or disconnect it from a power source.
2. **Shutdown / Restart.** To shutdown, press and hold the power button continuously for 6 seconds. To restart, press the power button.

Multi-App Scenarios

Validate core app functionality when switching between apps, especially if you've implemented a background task. Copy/Paste and Cortana integration are also worth checking where applicable.

Telemetry

Use telemetry and analytics to guide you. Integrating analytics into your app will help you get insights about your app from your Beta testers and end-users. This data can be used to help optimize your app before submission to the Store and for future updates. There are many analytics options out there. If you're not sure where to start, check out [App Insights](#).

Questions to consider:

1. How are users using the space?
2. How is the app placing objects in the world - can you detect problems?
3. How much time do they spend on different stages of the application?
4. How much time do they spend in the app?
5. What are the most common usage paths the users are trying?
6. Are users hitting unexpected states and/or errors?

Emulator and Simulated Input

The [HoloLens emulator](#) is a great way to efficiently test your Holographic app with a variety of simulated user characteristics and spaces. Here are some suggestions for effectively using the emulator to test your app:

1. **Use the emulator's virtual rooms to expand your testing.** The emulator comes with a set of virtual rooms

that you can use to test your app in even more environments.

2. **Use the emulator to look at your app from all angles.** The PageUp/PageDn keys will make your simulated user taller or shorter.
3. **Test your app with a real HoloLens.** The HoloLens Emulator is a great tool to help you quickly iterate on an app and catch new bugs, but make sure you also test on a physical HoloLens before submitting to the Windows Store. This is important to ensure that the performance and experience are great on real hardware.

Automated testing with Perception Simulation

Some app developers might want to automate testing of their apps. Beyond simple unit tests, you can use the [perception simulation](#) stack in HoloLens to automate human and world input to your app. The perception simulation API can send simulated input to either the HoloLens emulator or a physical HoloLens.

Windows App Certification Kit

To give your app the best chance of being [published on the Windows Store](#), validate and test it locally before you submit it for certification. If your app targets the Windows.Holographic device family, the [Windows App Certification Kit](#) will only run local static analysis tests on your PC. No tests will be run on your HoloLens.

See also

- [Submitting an app to the Windows Store](#)

Using the HoloLens emulator

11/6/2018 • 6 minutes to read • [Edit Online](#)

The HoloLens emulator allows you to test holographic apps on your PC without a physical HoloLens and comes with the HoloLens development toolset. The emulator uses a Hyper-V virtual machine. The human and environmental inputs that would usually be read by the sensors on the HoloLens are instead simulated using your keyboard, mouse, or Xbox controller. Apps don't need to be modified to run on the emulator and don't know that they aren't running on a real HoloLens.

If you're looking to develop Windows Mixed Reality immersive (VR) headset apps or games for desktop PCs, check out the [Windows Mixed Reality simulator](#), which lets you simulate desktop headsets instead.

Installing the HoloLens emulator

[Click here to download the latest build of the HoloLens emulator](#)

You can find older builds of the HoloLens emulator on the [HoloLens emulator archive](#) page.

HoloLens emulator system requirements

The HoloLens emulator is based on Hyper-V and uses RemoteFx for hardware accelerated graphics. To use the emulator, make sure your PC meets these hardware requirements:

- 64-bit Windows 10 Pro, Enterprise, or Education

NOTE

Windows 10 Home edition does not support Hyper-V or the HoloLens emulator

- 64-bit CPU
- CPU with 4 cores (or multiple CPUs with a total of 4 cores)
- 8 GB of RAM or more
- In the BIOS, the following features must be [supported and enabled](#):
 - Hardware-assisted virtualization
 - Second Level Address Translation (SLAT)
 - Hardware-based Data Execution Prevention (DEP)
- GPU requirements (the emulator might work with an unsupported GPU, but will be significantly slower)
 - DirectX 11.0 or later
 - WDDM 1.2 driver or later

If your system meets the above requirements, **please ensure that the "Hyper-V" feature has been enabled on your system** through Control Panel -> Programs -> Programs and Features -> Turn Windows Features on or off -> ensure that "Hyper-V" is selected for the Emulator installation to be successful.

Installation troubleshooting

You may see an error while installing the emulator that you need "*Visual Studio 2015 Update 1 and UWP tools version 1.2*". There are three possible causes of this error:

- You do not have a recent enough version of Visual Studio (Visual Studio 2017 or Visual Studio 2015 Update 1 or later). To correct this, install the latest release of Visual Studio.
- You have a recent enough version of Visual Studio, but you do not have the Universal Windows Platform

(UWP) tools installed. This is an optional feature for Visual Studio.

You may also see an error installing the emulator on a non-PRO/Enterprise/Education SKU of Windows or if you do not have Hyper-V feature enabled.

- Please read the [system requirements](#) section above for a complete set of requirements.
- Please also ensure that Hyper-V feature has been enabled on your system.

Deploying apps to the HoloLens emulator

1. Load your app solution in Visual Studio 2015.

NOTE

When using Unity, build your project from Unity and then load the built solution into Visual Studio as usual.

2. Ensure the Platform is set to **x86**.
3. Select the **HoloLens Emulator** as the target device for debugging.
4. Go to **Debug > Start Debugging** or press **F5** to launch the emulator and deploy your app for debugging.

The emulator may take a minute or more to boot when you first start it. We recommend that you keep the emulator open during your debugging session so you can quickly deploy apps to the running emulator.

Basic emulator input

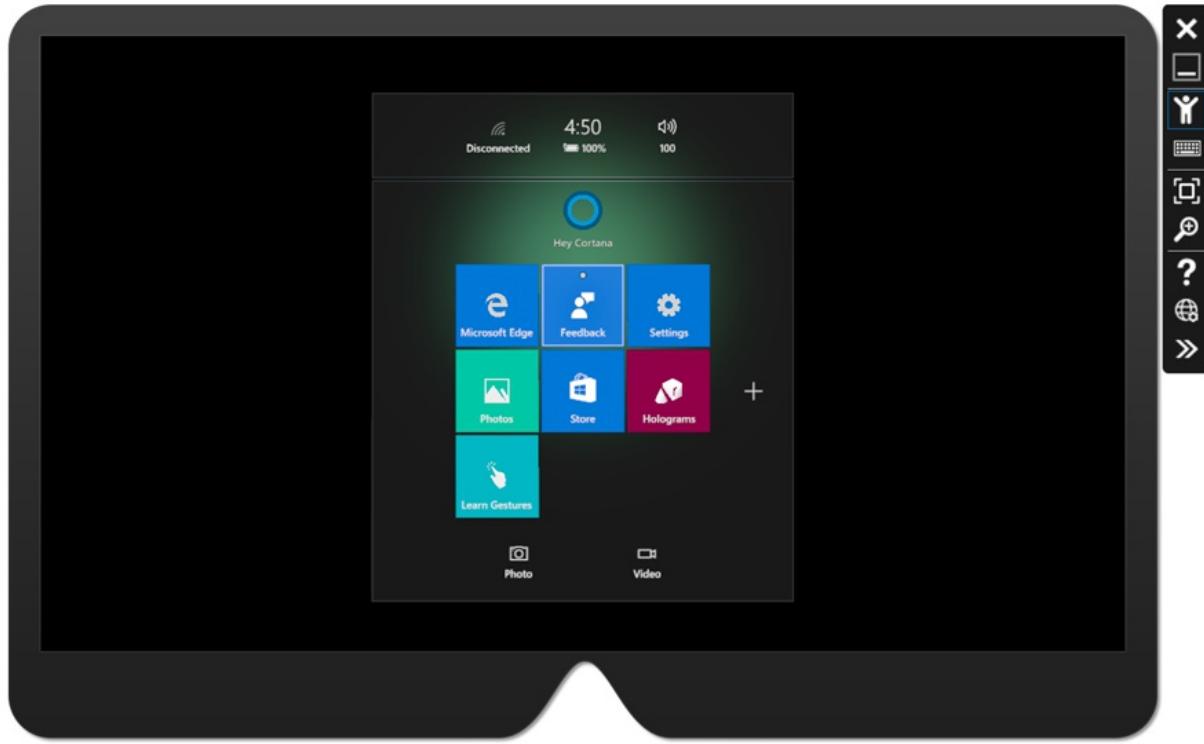
Controlling the emulator is very similar to many common 3D video games. There are input options available using the keyboard, mouse, or Xbox controller. You control the emulator by directing the actions of a simulated user wearing a HoloLens. Your actions move that simulated user around and apps running in the emulator respond like they would on a real device.

- **Walk forward, back, left, and right** - Use the W,A,S, and D keys on your keyboard, or the left stick on an Xbox controller.
- **Look up, down, left, and right** - Click and drag the mouse, use the arrow keys on your keyboard, or the right stick on an Xbox controller.
- **Air tap gesture** - Right-click the mouse, press the Enter key on your keyboard, or use the A button on an Xbox controller.
- **Bloom gesture** - Press the Windows key or F2 key on your keyboard, or press the B button on an Xbox controller.
- **Hand movement for scrolling** - Hold the Alt key, hold the right mouse button, and drag the mouse up / down, or in an Xbox controller hold the right trigger and A button down and move the right stick up and down.

Anatomy of the HoloLens emulator

Main window

When the emulator launches, you will see a window which displays the HoloLens OS.



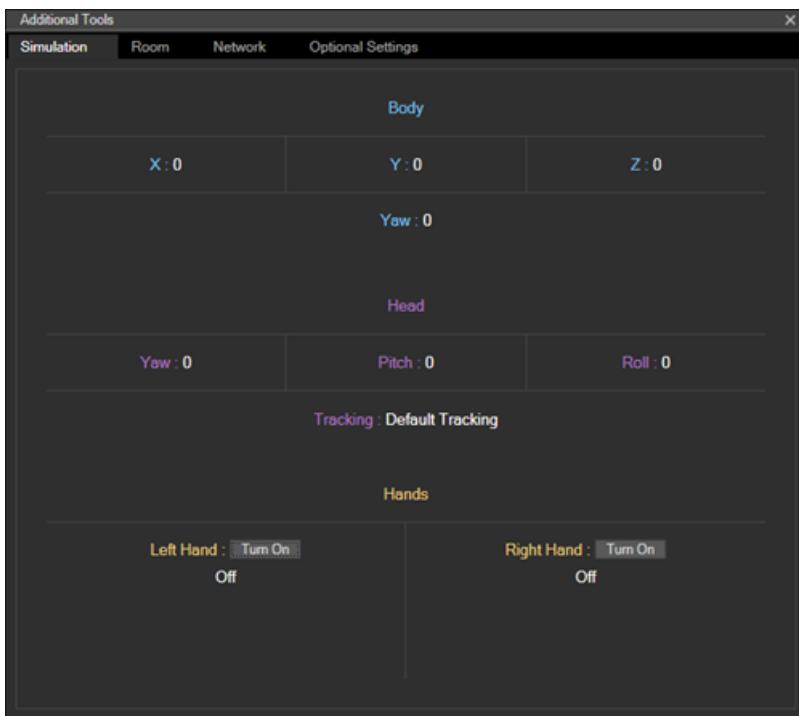
Toolbar

To the right of the main window, you will find the emulator toolbar. The toolbar contains the following buttons:

- **Close**: Closes the emulator.
- **Minimize**: Minimizes the emulator window.
- **Human Input**: Mouse and Keyboard are used to simulate human [input to the emulator](#).
- **Keyboard and Mouse Input**: Keyboard and mouse input are passed directly to the HoloLens OS as keyboard and mouse events as if you connected a Bluetooth keyboard and mouse.
- **Fit to Screen**: Fits the emulator to screen.
- **Zoom**: Make the emulator larger and smaller.
- **Help**: Open emulator help.
- **Open Device Portal**: Open the Windows Device Portal for the HoloLens OS in the emulator.
- **Tools**: Open the **Additional Tools** pane.

Simulation tab

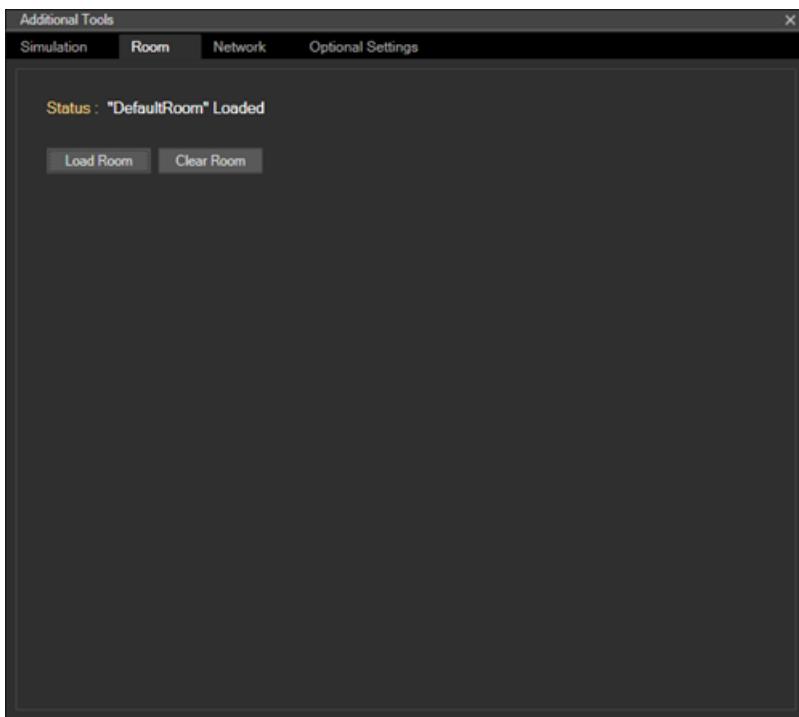
The default tab within the **Additional Tools** pane is the **Simulation** tab.



The Simulation tab shows the current state of the simulated sensors used to drive the HoloLens OS within the emulator. Hovering over any value in the Simulation tab will provide a tooltip describing how to control that value.

Room tab

The emulator simulates world input in the form of the spatial mapping mesh from simulated "rooms". This tab lets you pick which room to load instead of the default room.



Simulated rooms are useful for testing your app in multiple environments. Several rooms are shipped with the emulator and once you install the emulation, you will find them in %ProgramFiles(x86)%\Program Files (x86)\Microsoft XDE\10.0.11082.0\Plugins\Rooms. All of these rooms were captured in real environments using a HoloLens:

- **DefaultRoom.xef** - A small living room with a TV, coffee table, and two sofas. Loaded by default when you start the emulator.
- **Bedroom1.xef** - A small bedroom with a desk.

- **Bedroom2.xef** - A bedroom with a queen size bed, dresser, nightstands, and walk-in closet.
- **GreatRoom.xef** - A large open space great room with living room, dining table, and kitchen.
- **LivingRoom.xef** - A living room with a fireplace, sofa, armchairs, and a coffee table with a vase.

You can also record your own rooms to use in the emulator using the Simulation page of the [Windows Device Portal](#) on your HoloLens.

On the emulator, you will only see holograms that you render and you will not see the simulated room behind the holograms. This is in contrast to the real HoloLens where you see both blended together. If you want to see the simulated room in the HoloLens emulator, you will need to update your app to render the spatial mapping mesh in the scene.

Account Tab

The Account tab allows you to configure the emulator to sign-in with a Microsoft Account. This is useful for testing API's that require the user to be signed-in with an account. After checking the box on this page, subsequent launches of the emulator will ask you to sign-in, just like a user would the first time the HoloLens is started.

See also

- [Advanced HoloLens Emulator and Mixed Reality Simulator input](#)
- [HoloLens emulator software history](#)
- [Spatial mapping in Unity](#)
- [Spatial mapping in DirectX](#)

Using the Windows Mixed Reality simulator

11/6/2018 • 2 minutes to read • [Edit Online](#)

The Windows Mixed Reality simulator allows you to test mixed reality apps on your PC without a Windows Mixed Reality immersive headset. It is available beginning with the Windows 10 Creators Update. The simulator is similar to the [HoloLens emulator](#), though the simulator does not use a virtual machine. Apps running in the simulator run in your Windows 10 desktop user session, just like they would if you were using an immersive headset. The human and environmental input that would usually be read by the sensors on an immersive headset are instead simulated using your keyboard, mouse, or Xbox controller. Apps don't need to be modified to run in the simulator, and don't know that they aren't running on an immersive headset.

Enabling the Windows Mixed Reality simulator

1. **Install the latest Windows Insider Preview Build** - 64-bit version is required. See [here](#) for more details.
2. **Enable Developer mode** from Settings -> Update and Security -> For developers
3. Launch the **Mixed Reality Portal** from the desktop
4. If this is your first time launching the portal, you'll need to go through the setup experience a. Click **Get started** b. Click **I agree** to accept the agreement c. Click **Set up for simulation (for developers)** to proceed through setup without a physical device d. Click **Set up** to confirm your choice
5. Click the **For developers** button on the left side of the Mixed Reality Portal
6. Turn the Simulation toggle switch to **On** a. This requires admin permissions and you must accept the User Account Control dialog that appears

You should now be running with simulation!

Deploying apps to the Mixed Reality simulator

Since the simulator runs on your local PC without a Virtual Machine, you can simply deploy your Universal Windows apps to the **Local Machine** when debugging.

Basic simulator input

Controlling the simulator is very similar to many common 3D video games and the [HoloLens emulator](#). There are input options available using the keyboard, mouse, or Xbox controller.

You control the simulator by directing the actions of a simulated user wearing an immersive headset. Your actions move the simulated user and cause interactions with apps that respond as they would on an immersive headset.

- **Walk forward, back, left, and right** - Use the W,A,S, and D keys on your keyboard, or the left stick on an Xbox controller.
- **Look up, down, left, and right** - Click and drag the mouse, use the arrow keys on your keyboard, or the right stick on an Xbox controller.
- **Action button press on controller** - Right-click the mouse, press the Enter key on your keyboard, or use the A button on an Xbox controller.
- **Home button press on controller** - Press the Windows key or F2 key on your keyboard, or press the B button on an Xbox controller.
- **Controller movement for scrolling** - Hold the Alt key, hold the right mouse button, and drag the mouse up / down, or in an Xbox controller hold the right trigger and A button down and move the right stick up and

down.

Tracked controllers

The Mixed Reality simulator can simulate up to two hand-held tracked motion controllers. Enable them using the toggle switches in the Mixed Reality Portal. Each simulated controller has:

- Position in space
- Home button
- Menu button
- Grip button
- Touchpad

See also

- [Advanced Mixed Reality Simulator Input](#)
- [Spatial mapping in Unity](#)
- [Spatial mapping in DirectX](#)

Advanced HoloLens Emulator and Mixed Reality Simulator input

11/6/2018 • 5 minutes to read • [Edit Online](#)

Most emulator users will only need to use the basic input controls for the [HoloLens emulator](#) or the [Windows Mixed Reality simulator](#). The details below are for advanced users who have found a need to simulate more complex types of input.

Concepts

To get started controlling the virtual input to the HoloLens emulator and Windows Mixed Reality simulator, you should first understand a few concepts.

Motion is controlled with both rotation and translation (movement) along three axes.

- **Yaw:** Turn left or right.
- **Pitch:** Turn up or down.
- **Roll:** Roll side-to-side.
- **X:** Move left or right.
- **Y:** Move up or down.
- **Z:** Move forward or backward.

[Gesture](#) and motion controller input are mapped closely to how they physical devices:

- **Action:** This simulates the action of pressing the forefinger to the thumb or pulling the action button on a controller. For example, the Action input can be used to simulate the air-tap gesture, to scroll through content, and to press-and-hold.
- **Bloom or Home:** The HoloLens bloom gesture or a controller's Home button is used to return to the shell and to perform system actions.

You can also control the state of simulated sensor input:

- **Reset:** This will return all simulated sensors to their default values.
- **Tracking:** Cycles through the positional tracking modes. This includes:
 - **Default:** The OS chooses the best tracking mode based upon the requests made of the system.
 - **Orientation:** Forces Orientation-only tracking, regardless of the requests made of the system.
 - **Positional:** Forces Positional tracking, regardless of the requests made of the system.

Types of input

The following table shows how each type of input maps to the keyboard, mouse, and Xbox controller. Each type has a different mapping depending on the input control mode; more information on input control modes is provided later in this document.

	KEYBOARD	MOUSE	XBOX CONTROLLER
Yaw	Left / right arrows	Drag Left / Right	Right thumbstick left / right
Pitch	Up / down arrows	Drag up / down	Right thumbstick up / down

	KEYBOARD	MOUSE	XBOX CONTROLLER
Roll	Q / E		DPad left / right
X	A / D		Left thumbstick left / right
Y	Page up / page down		DPad up / down
Z	W / S		Left thumbstick up / down
Action	Enter or space	Right button	A button or either trigger
Bloom	F2 or Windows key (Windows key only works with the HoloLens emulator)		B button
Controller grip button	G (Windows Mixed Reality simulator-only)		
Controller menu button	M (Windows Mixed Reality simulator-only)		
Controller touchpad touch	U (Windows Mixed Reality simulator-only)		
Controller touchpad press	P (Windows Mixed Reality simulator-only)		
Reset	Escape key		Start button
Tracking	T or F3		X button

Input control modes

The emulator can be controlled in multiple modes, which impact how the controls are interpreted. The input modes are:

- **Default mode:** The default mode combines the most common operations for ease of use. This is the most commonly used mode.
- **Hands or controller mode:** The HoloLens emulator simulates gesture input with hands, while the Windows Mixed Reality simulator simulates tracked controllers. To enter this mode, press and hold an alt key on the keyboard: use left alt for the left hand/controller, and/or use right alt for the right hand/controller. You can also press and hold a shoulder button on the Xbox controller to enter this mode: press the left shoulder for the left hand/controller, and/or press the right shoulder for the right hand/controller.
 - Hands are typically not visible to the HoloLens emulator - they are made visible briefly when performing gestures such as [air-tap](#) and bloom using the default input mode. This is a difference from tracked controllers in the Mixed Reality simulator. The corresponding Hand is also made visible when you enter hands mode, or when you click "Turn On" in the **Simulation** tab, which is located in the **Additional Tools** pane.
 - * **Head mode:** The head mode applies controls, where appropriate, exclusively to the head. To enter head mode, press and hold the H key on the keyboard.

The following table shows how each input mode maps each type of input:

	DEFAULT	HAND/CONTROLLER (HOLD ALT / SHOULDER)	HEAD (HOLD H)
Yaw	Turn body left / right	Move hand left / right	Turn head left / right
Pitch	Turn head up / down	Move hand up / down	Turn head Up / down
Roll	Roll head left / right		Roll head left / right
X	Slide body left / right	Move hand/controller left / right	Turn head left / right
Y	Move body up / down	Move hand/controller up / down	Turn head up / down
Z	Move body forward / backward	Move hand/controller forward / backward	Turn head up / down
Action	Perform action	Perform action	
Bloom / Home	Perform bloom gesture or Home button press	Perform bloom gesture or Home button press	
Reset	Reset to defaults	Reset to defaults	Reset to defaults
Tracking	Cycle tracking	Cycle tracking	Cycle tracking

Controlling an app

This article has described the complete set of input types and input modes that are available in the HoloLens emulator and Windows Mixed Reality simulator. The following set of controls is suggested for day-to-day use:

OPERATION	KEYBOARD AND MOUSE	CONTROLLER
Body X	A / D	Left thumbstick left / right
Body Y	Page up / page down	DPad up / down
Body Z	W / S	Left thumbstick up / down
Body Yaw	Drag mouse left / right	Right thumbstick left / right
Head Yaw	H + drag mouse left / right	H (on Keyboard) + right thumbstick left / right
Head Pitch	Drag mouse up / down	Right thumbstick up / down
Head Roll	Q / E	DPad left / right
Hand X	Alt + drag mouse left / right	Shoulder + right thumbstick left / right
Hand Y	Alt + drag mouse up / down	Shoulder + right thumbstick up / down

OPERATION	KEYBOARD AND MOUSE	CONTROLLER
Hand Z	Alt + W / S	Shoulder + left thumbstick up / down
Action	Right mouse button	Trigger
Bloom / Home	F2 or Windows key (Windows key is only for the HoloLens emulator)	B button
Reset	Escape	Start button
Tracking	T	X button
Scrolling	Alt + right mouse button + drag mouse up / down	Shoulder + trigger + right thumbstick up / down

See also

- [Install the tools](#)
- [Using the HoloLens emulator](#)
- [Using the Windows Mixed Reality simulator](#)

HoloLens emulator archive

11/6/2018 • 2 minutes to read • [Edit Online](#)

HoloLens emulator builds install side-by-side. We generally recommend using the latest available build, but there might be cases where you want or need to test an app against an old emulator. This page includes links to released versions.

BUILD	RELEASE DATE	NOTES
HoloLens Emulator build 10.0.17134.80	May 21, 2018	Latest build. Windows 10 April 2018 Update.
HoloLens Emulator build 10.0.14393.1358	July 7, 2017	
HoloLens Emulator build 10.0.14393.0	August 2, 2016	
HoloLens Emulator build 10.0.14342.1018	May 31, 2016	
HoloLens Emulator build 10.0.11082.1033	March 30, 2016	

See also

- [Install the tools](#)
- [Using the HoloLens emulator](#)
- [Advanced HoloLens emulator and Mixed reality simulator input](#)

Perception simulation

11/6/2018 • 14 minutes to read • [Edit Online](#)

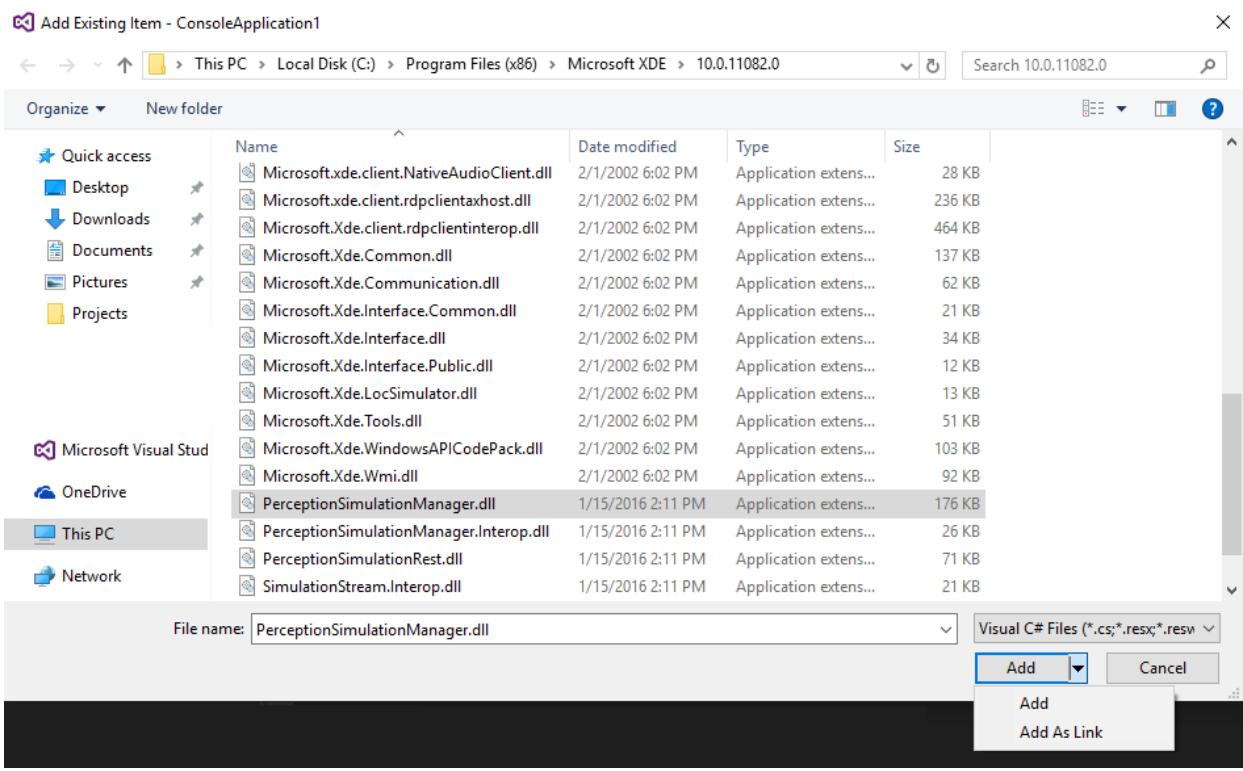
Do you want to build an automated test for your app? Do you want your tests to go beyond component-level unit testing and really exercise your app end-to-end? Perception Simulation is what you're looking for. The Perception Simulation library sends fake human and world input data to your app so you can automate your tests. For example, you can simulate the input of a human looking left and right and then performing a gesture.

Perception Simulation can send simulated input like this to a physical HoloLens, the HoloLens emulator, or a PC with Mixed Reality Portal installed. Perception Simulation bypasses the live sensors on a Mixed Reality device and sends simulated input to applications running on the device. Applications receive these fake input events through the same APIs they always use, and can't tell the difference between running with real sensors versus running with Perception Simulation. Perception Simulation is the same technology used by the HoloLens emulator to send simulated input to the HoloLens Virtual Machine.

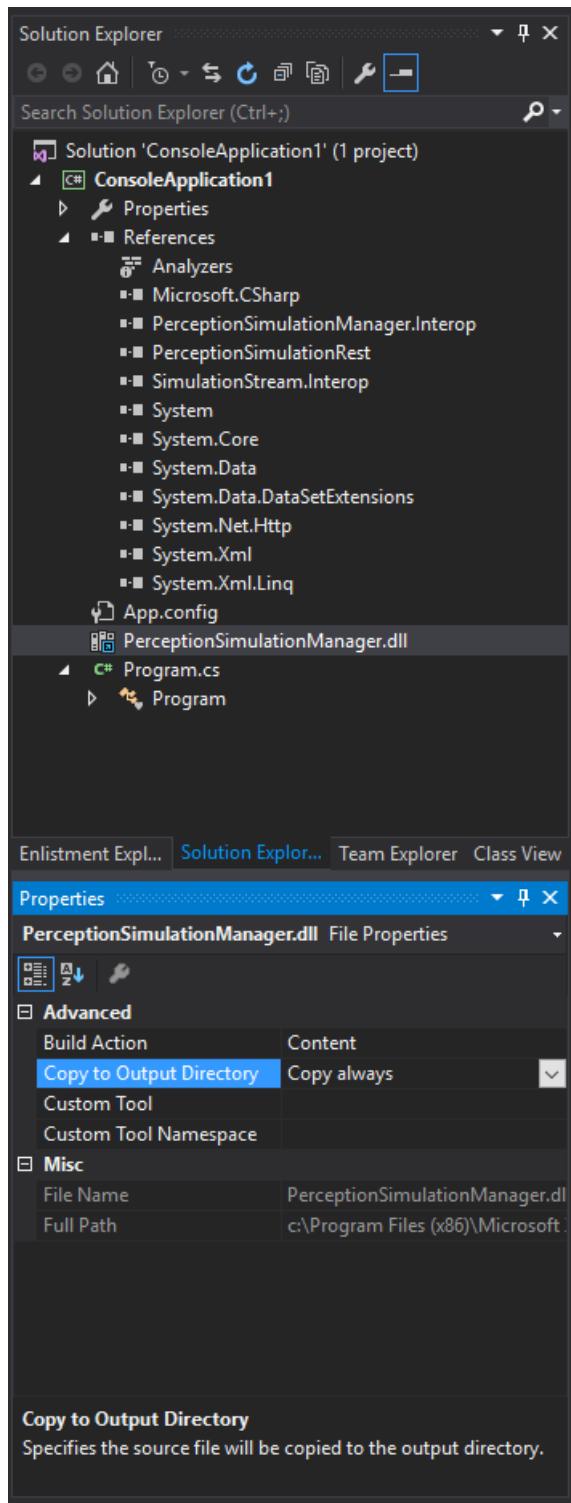
Simulation starts by creating an `IPerceptionSimulationManager` object. From that object, you can issue commands to control properties of a simulated "human", including head position, hand position, and gestures.

Setting Up a Visual Studio Project for Perception Simulation

1. [Install the HoloLens emulator](#) on your development PC. The emulator includes the libraries you will use for Perception Simulation.
2. Create a new Visual Studio C# desktop project (a Console Project works great to get started).
3. Add the following binaries to your project as references (Project->Add->Reference...). You can find them in **%ProgramFiles(x86)%\Microsoft XDE\10.0.11082.0** a. `PerceptionSimulationManager.Interop.dll` - Managed C# wrapper for Perception Simulation. b. `PerceptionSimulationRest.dll` - Library for setting up a web-socket communication channel to the HoloLens or emulator. c. `SimulationStream.Interop.dll` - Shared types for simulation.
4. Add the implementation binary `PerceptionSimulationManager.dll` to your project a. First add it as a binary to the project (Project->Add->Existing Item...). Save it as a link so that it doesn't copy it to your project source folder.



b. Then make sure that it get's copied to your output folder on build. This is in the property sheet for the binary .



Creating an `IPerceptionSimulationManager` Object

To control simulation, you'll issue updates to objects retrieved from an `IPerceptionSimulationManager` object. The first step is to get that object and connect it to your target device or emulator. You can get the IP address of your emulator by clicking on the Device Portal button in the [toolbar](#)

 **Open Device Portal:** Open the Windows Device Portal for the HoloLens OS in the emulator.

First, you'll call `RestSimulationStreamSink.Create` to get a `RestSimulationStreamSink` object. This is the target device or emulator that you will control over an http connection. Your commands will be passed to and handled by the [Windows Device Portal](#) running on the device or emulator. The four parameters you'll need to create an object are:

- Uri uri - IP address of the target device (e.g., "http://123.123.123.123")

- System.Net.NetworkCredential credentials - Username/password for connecting to the [Windows Device Portal](#) on the target device or emulator. If you are connecting to the emulator via its local 168...* address on the same PC, any credentials will be accepted.
- bool normal - True for normal priority, false for low priority. You generally want to set this to *true* for test scenarios.
- System.Threading.CancellationToken token - Token to cancel the async operation.

Second you will create the IPerceptionSimulationManager. This is the object you use to control simulation. Note that this must also be done in an async method.

Control the simulated Human

An IPerceptionSimulationManager has a Human property that returns an ISimulatedHuman object. To control the simulated human, perform operations on this object. For example:

```
manager.Human.Move(new Vector3(0.1f, 0.0f, 0.0f))
```

Basic Sample C# console application

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.PerceptionSimulation;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Task.Run(async () =>
            {
                try
                {
                    RestSimulationStreamSink sink = await RestSimulationStreamSink.Create(
                        // use the IP address for your device/emulator
                        new Uri("http://169.254.227.115"),
                        // no credentials are needed for the emulator
                        new System.Net.NetworkCredential("", ""),
                        // normal priority
                        true,
                        // cancel token
                        new System.Threading.CancellationToken());

                    IPerceptionSimulationManager manager =
                    PerceptionSimulationManager.CreatePerceptionSimulationManager(sink);
                }
                catch (Exception e)
                {
                    Console.WriteLine(e);
                }
            });

            // If main exits, the process exits.
            Console.WriteLine("Press any key to exit...");
            Console.ReadLine();
        }
    }
}

```

Extended Sample C# console application

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.PerceptionSimulation;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Task.Run(async () =>
            {
                try
                {
                    RestSimulationStreamSink sink = await RestSimulationStreamSink.Create(

```

```

        // use the IP address for your device/emulator
        new Uri("http://169.254.227.115"),
        // no credentials are needed for the emulator
        new System.Net.NetworkCredential("", ""),
        // normal priority
        true,
        // cancel token
        new System.Threading.CancellationToken());

    IPerceptionSimulationManager manager =
    PerceptionSimulationManager.CreatePerceptionSimulationManager(sink);

    // Now, we'll simulate a sequence of actions.
    // Sleeps in-between each action give time to the system
    // to be able to properly react.
    // This is just an example. A proper automated test should verify
    // that the app has behaved correctly
    // before proceeding to the next step, instead of using Sleeps.

    // Simulate Bloom gesture, which should cause Shell to disappear
    manager.Human.RightHand.PerformGesture(SimulatedGesture.Home);
    Thread.Sleep(2000);

    // Simulate Bloom gesture again... this time, Shell should reappear
    manager.Human.RightHand.PerformGesture(SimulatedGesture.Home);
    Thread.Sleep(2000);

    // Simulate a Head rotation down around the X axis
    // This should cause gaze to aim about the center of the screen
    manager.Human.Head.Rotate(new Rotation3(0.04f, 0.0f, 0.0f));
    Thread.Sleep(300);

    // Simulate a finger press & release
    // Should cause a tap on the center tile, thus launching it
    manager.Human.RightHand.PerformGesture(SimulatedGesture.FingerPressed);
    Thread.Sleep(300);
    manager.Human.RightHand.PerformGesture(SimulatedGesture.FingerReleased);
    Thread.Sleep(2000);

    // Simulate a second finger press & release
    // Should activate the app that was launched when the center tile was clicked
    manager.Human.RightHand.PerformGesture(SimulatedGesture.FingerPressed);
    Thread.Sleep(300);
    manager.Human.RightHand.PerformGesture(SimulatedGesture.FingerReleased);
    Thread.Sleep(5000);

    // Simulate a Head rotation towards the upper right corner
    manager.Human.Head.Rotate(new Rotation3(-0.14f, 0.17f, 0.0f));
    Thread.Sleep(300);

    // Simulate a third finger press & release
    // Should press the Remove button on the app
    manager.Human.RightHand.PerformGesture(SimulatedGesture.FingerPressed);
    Thread.Sleep(300);
    manager.Human.RightHand.PerformGesture(SimulatedGesture.FingerReleased);
    Thread.Sleep(2000);

    // Simulate Bloom gesture again... bringing the Shell back once more
    manager.Human.RightHand.PerformGesture(SimulatedGesture.Home);
    Thread.Sleep(2000);
}

catch (Exception e)
{
    Console.WriteLine(e);
}

});

// If main exits, the process exits.
Console.WriteLine("Press any key to exit...");

```

```
        Console.ReadLine();
    }
}
```

API Reference

Microsoft.PerceptionSimulation.SimulatedDeviceType

Describes a simulated device type

```
public enum SimulatedDeviceType
{
    Reference = 0
}
```

Microsoft.PerceptionSimulation.SimulatedDeviceType.Reference

A fictitious reference device, the default for PerceptionSimulationManager

Microsoft.PerceptionSimulation.HeadTrackerMode

Describes a head tracker mode

```
public enum HeadTrackerMode
{
    Default = 0,
    Orientation = 1,
    Position = 2
}
```

Microsoft.PerceptionSimulation.HeadTrackerMode.Default

Default Head Tracking. This means the system may select the best head tracking mode based upon runtime conditions.

Microsoft.PerceptionSimulation.HeadTrackerMode.Orientation

Orientation Only Head Tracking. This means that the tracked position may not be reliable, and some functionality dependent on head position may not be available.

Microsoft.PerceptionSimulation.HeadTrackerMode.Position

Positional Head Tracking. This means that the tracked head position and orientation are both reliable

Microsoft.PerceptionSimulation.SimulatedGesture

Describes a simulated gesture

```
public enum SimulatedGesture
{
    None = 0,
    FingerPressed = 1,
    FingerReleased = 2,
    Home = 4,
    Max = Home
}
```

Microsoft.PerceptionSimulation.SimulatedGesture.None

A sentinel value used to indicate no gestures

Microsoft.PerceptionSimulation.SimulatedGesture.FingerPressed

A finger pressed gesture

Microsoft.PerceptionSimulation.SimulatedGesture.FingerReleased

A finger released gesture

Microsoft.PerceptionSimulation.SimulatedGesture.Home

The home gesture

Microsoft.PerceptionSimulation.SimulatedGesture.Max

The maximum valid gesture

Microsoft.PerceptionSimulation.PlaybackState

Describes the state of a playback

```
public enum PlaybackState
{
    Stopped = 0,
    Playing = 1,
    Paused = 2,
    End = 3,
}
```

Microsoft.PerceptionSimulation.PlaybackState.Stopped

The recording is currently stopped and ready for playback

Microsoft.PerceptionSimulation.PlaybackState.Playing

The recording is currently playing

Microsoft.PerceptionSimulation.PlaybackState.Paused

The recording is currently paused

Microsoft.PerceptionSimulation.PlaybackState.End

The recording has reached the end

Microsoft.PerceptionSimulation.Vector3

Describes a 3 components vector, which might describe a point or a vector in 3D space

```
public struct Vector3
{
    public float X;
    public float Y;
    public float Z;
    public Vector3(float x, float y, float z);
}
```

Microsoft.PerceptionSimulation.Vector3.X

The X component of the vector

Microsoft.PerceptionSimulation.Vector3.Y

The Y component of the vector

Microsoft.PerceptionSimulation.Vector3.Z

The Z component of the vector

Microsoft.PerceptionSimulation.Vector3.#ctor(System.Single,System.Single,System.Single)

Construct a new Vector3

Parameters

- x - The x component of the vector
- y - The y component of the vector
- z - The z component of the vector

Microsoft.PerceptionSimulation.Rotation3

Describes a 3 components rotation

```
public struct Rotation3
{
    public float Pitch;
    public float Yaw;
    public float Roll;
    public Rotation3(float pitch, float yaw, float roll);
}
```

Microsoft.PerceptionSimulation.Rotation3.Pitch

The Pitch component of the Rotation, down around the X axis

Microsoft.PerceptionSimulation.Rotation3.Yaw

The Yaw component of the Rotation, right around the Y axis

Microsoft.PerceptionSimulation.Rotation3.Roll

The Roll component of the Rotation, right around the Z axis

Microsoft.PerceptionSimulation.Rotation3.#ctor(System.Single,System.Single,System.Single)

Construct a new Rotation3

Parameters

- pitch - The pitch component of the Rotation
- yaw - The yaw component of the Rotation
- roll - The roll component of the Rotation

Microsoft.PerceptionSimulation.Frustum

Describes a view frustum, as typically used by a camera

```
public struct Frustum
{
    float Near;
    float Far;
    float FieldOfView;
    float AspectRatio;
}
```

Microsoft.PerceptionSimulation.Frustum.Near

The minimum distance that is contained in the frustum

Microsoft.PerceptionSimulation.Frustum.Far

The maximum distance that is contained in the frustum

Microsoft.PerceptionSimulation.Frustum.FieldOfView

The horizontal field of view of the frustum, in radians (less than PI)

Microsoft.PerceptionSimulation.Frustum.AspectRatio

The ratio of horizontal field of view to vertical field of view

Microsoft.PerceptionSimulation.IPerceptionSimulationManager

Root for generating the packets used to control a device

```
public interface IPerceptionSimulationManager
{
    ISimulatedDevice Device { get; }
    ISimulatedHuman Human { get; }
    void Reset();
}
```

Microsoft.PerceptionSimulation.IPerceptionSimulationManager.Device

Retrieve the simulated device object that interprets the simulated human and the simulated world.

Microsoft.PerceptionSimulation.IPerceptionSimulationManager.Human

Retrieve the object that controls the simulated human.

Microsoft.PerceptionSimulation.IPerceptionSimulationManager.Reset

Resets the simulation to its default state.

Microsoft.PerceptionSimulation.ISimulatedDevice

Interface describing the device which interprets the simulated world and the simulated human

```
public interface ISimulatedDevice
{
    ISimulatedHeadTracker HeadTracker { get; }
    ISimulatedHandTracker HandTracker { get; }
    void SetSimulatedDeviceType(SimulatedDeviceType type);
}
```

Microsoft.PerceptionSimulation.ISimulatedDevice.HeadTracker

Retrieve the Head Tracker from the Simulated Device.

Microsoft.PerceptionSimulation.ISimulatedDevice.HandTracker

Retrieve the Hand Tracker from the Simulated Device.

Microsoft.PerceptionSimulation.ISimulatedDevice.SetSimulatedDeviceType([Microsoft.PerceptionSimulation.SimulatedDeviceType](#))

Set the properties of the simulated device to match the provided device type.

Parameters

- type - The new type of Simulated Device

Microsoft.PerceptionSimulation.ISimulatedHeadTracker

Interface describing the portion of the simulated device that tracks the head of the simulated human

```
public interface ISimulatedHeadTracker
{
    HeadTrackerMode HeadTrackerMode { get; set; }
};
```

Microsoft.PerceptionSimulation.ISimulatedHeadTracker.HeadTrackerMode

Retrieves and sets the current head tracker mode.

Microsoft.PerceptionSimulation.ISimulatedHandTracker

Interface describing the portion of the simulated device that tracks the hands of the simulated human

```
public interface ISimulatedHandTracker
{
    Vector3 WorldPosition { get; }
    Vector3 Position { get; set; }
    float Pitch { get; set; }
    bool FrustumIgnored { [return: MarshalAs(UnmanagedType.Bool)] get; [param: MarshalAs(UnmanagedType.Bool)] set; }
    Frustum Frustum { get; set; }
}
```

Microsoft.PerceptionSimulation.ISimulatedHandTracker.WorldPosition

Retrieve the position of the node with relation to the world, in meters

Microsoft.PerceptionSimulation.ISimulatedHandTracker.Position

Retrieve and set the position of the simulated hand tracker, relative to the center of the head.

Microsoft.PerceptionSimulation.ISimulatedHandTracker.Pitch

Retrieve and set the downward pitch of the simulated hand tracker

Microsoft.PerceptionSimulation.ISimulatedHandTracker.FrustumIgnored

Retrieve and set whether the frustum of the simulated hand tracker is ignored. When ignored, both hands are always visible. When not ignored (the default) hands are only visible when they are within the frustum of the hand tracker.

Microsoft.PerceptionSimulation.ISimulatedHandTracker.Frustum

Retrieve and set the frustum properties used to determine if hands are visible to the simulated hand tracker.

Microsoft.PerceptionSimulation.ISimulatedHuman

Top level interface for controlling the simulated human

```
public interface ISimulatedHuman
{
    Vector3 WorldPosition { get; set; }
    float Direction { get; set; }
    float Height { get; set; }
    ISimulatedHand LeftHand { get; }
    ISimulatedHand RightHand { get; }
    ISimulatedHead Head { get; }
    void Move(Vector3 translation);
    void Rotate(float radians);
}
```

Microsoft.PerceptionSimulation.ISimulatedHuman.WorldPosition

Retrieve and set the position of the node with relation to the world, in meters. The position corresponds to a point at the center of the human's feet.

Microsoft.PerceptionSimulation.ISimulatedHuman.Direction

Retrieve and set the direction the simulated human faces in the world. 0 radians faces down the negative Z axis. Positive radians rotate clockwise about the Y axis.

Microsoft.PerceptionSimulation.ISimulatedHuman.Height

Retrieve and set the height of the simulated human, in meters

Microsoft.PerceptionSimulation.ISimulatedHuman.LeftHand

Retrieve the left hand of the simulated human

Microsoft.PerceptionSimulation.ISimulatedHuman.RightHand

Retrieve the right hand of the simulated human

Microsoft.PerceptionSimulation.ISimulatedHuman.Head

Retrieve the head of the simulated human

Microsoft.PerceptionSimulation.ISimulatedHuman.Move(Microsoft.PerceptionSimulation.Vector3)

Move the simulated human relative to its current position, in meters

Parameters

- translation - The translation to move, relative to current position

Microsoft.PerceptionSimulation.ISimulatedHuman.Rotate(System.Single)

Rotate the simulated human relative to its current direction, clockwise about the Y axis

Parameters

- radians - The amount to rotate around the Y axis

Microsoft.PerceptionSimulation.ISimulatedHand

Interface describing a hand of the simulated human

```
public interface ISimulatedHand
{
    Vector3 WorldPosition { get; }
    Vector3 Position { get; set; }
    bool Activated { [return: MarshalAs(UnmanagedType.Bool)] get; [param: MarshalAs(UnmanagedType.Bool)] set; }
    bool Visible { [return: MarshalAs(UnmanagedType.Bool)] get; }
    void EnsureVisible();
    void Move(Vector3 translation);
    void PerformGesture(SimulatedGesture gesture);
}
```

Microsoft.PerceptionSimulation.ISimulatedHand.WorldPosition

Retrieve the position of the node with relation to the world, in meters

Microsoft.PerceptionSimulation.ISimulatedHand.Position

Retrieve and set the position of the simulated hand relative to the human, in meters.

Microsoft.PerceptionSimulation.ISimulatedHand.Activated

Retrieve and set whether the hand is currently activated.

Microsoft.PerceptionSimulation.ISimulatedHand.Visible

Retrieve whether the hand is currently visible to the SimulatedDevice (ie, whether it's in a position to be detected by the HandTracker).

Microsoft.PerceptionSimulation.ISimulatedHand.EnsureVisible

Move the hand such that it is visible to the SimulatedDevice.

Microsoft.PerceptionSimulation.ISimulatedHand.Move(Microsoft.PerceptionSimulation.Vector3)

Move the position of the simulated hand relative to its current position, in meters.

Parameters

- translation - The amount to translate the simulated hand

Microsoft.PerceptionSimulation.ISimulatedHand.PerformGesture(Microsoft.PerceptionSimulation.SimulatedGesture)

Perform a gesture using the simulated hand (it will only be detected by the system if the hand is enabled).

Parameters

- gesture - The gesture to perform

Microsoft.PerceptionSimulation.ISimulatedHead

Interface describing the head of the simulated human

```
public interface ISimulatedHead
{
    Vector3 WorldPosition { get; }
    Rotation3 Rotation { get; set; }
    float Diameter { get; set; }
    void Rotate(Rotation3 rotation);
}
```

Microsoft.PerceptionSimulation.ISimulatedHead.WorldPosition

Retrieve the position of the node with relation to the world, in meters

Microsoft.PerceptionSimulation.ISimulatedHead.Rotation

Retrieve the rotation of the simulated head. Positive radians rotate clockwise when looking along the axis.

Microsoft.PerceptionSimulation.ISimulatedHead.Diameter

Retrieve the simulated head's diameter. This value is used to determine the head's center (point of rotation).

Microsoft.PerceptionSimulation.ISimulatedHead.Rotate(Microsoft.PerceptionSimulation.Rotation3)

Rotate the simulated head relative to its current rotation. Positive radians rotate clockwise when looking along the axis.

Parameters

- rotation - The amount to rotate

Microsoft.PerceptionSimulation.ISimulationRecording

Interface for interacting with a single recording loaded for playback.

```
public interface ISimulationRecording
{
    StreamDataTypes DataTypes { get; }
    PlaybackState State { get; }
    void Play();
    void Pause();
    void Seek(UInt64 ticks);
    void Stop();
};
```

Microsoft.PerceptionSimulation.ISimulationRecording.DataTypes

Retrieves the list of data types in the recording.

Microsoft.PerceptionSimulation.ISimulationRecording.State

Retrieves the current state of the recording.

Microsoft.PerceptionSimulation.ISimulationRecording.Play

Starts the playback. If the recording is paused, playback will resume from the paused location; if stopped, playback will start at the beginning. If already playing, this call is ignored.

Microsoft.PerceptionSimulation.ISimulationRecording.Pause

Pauses the playback at its current location. If the recording is stopped, the call is ignored.

Microsoft.PerceptionSimulation.ISimulationRecording.Seek(System.UInt64)

Seeks the recording to the specified time (in 100 nanoseconds intervals from the beginning) and pauses at that location. If the time is beyond the end of the recording, it is paused at the last frame.

Parameters

- ticks - The time to which to seek

Microsoft.PerceptionSimulation.ISimulationRecording.Stop

Stops the playback and resets the position to the beginning

Microsoft.PerceptionSimulation.ISimulationRecordingCallback

Interface for receiving state changes during playback

```
public interface ISimulationRecordingCallback
{
    void PlaybackStateChanged(PlaybackState newState);
};
```

Microsoft.PerceptionSimulation.ISimulationRecordingCallback.PlaybackStateChanged(Microsoft.PerceptionSimulation.PlaybackState)

Called when an ISimulationRecording's playback state has changed

Parameters

- newState - The new state of the recording

Microsoft.PerceptionSimulation.PerceptionSimulationManager

Root object for creating Perception Simulation objects

```
public static class PerceptionSimulationManager
{
    public static IPerceptionSimulationManager CreatePerceptionSimulationManager(ISimulationStreamSink sink);
    public static ISimulationStreamSink CreatePerceptionSimulationRecording(string path);
    public static ISimulationRecording LoadPerceptionSimulationRecording(string path,
    ISimulationStreamSinkFactory factory);
    public static ISimulationRecording LoadPerceptionSimulationRecording(string path,
    ISimulationStreamSinkFactory factory, ISimulationRecordingCallback callback);
}
```

Microsoft.PerceptionSimulation.PerceptionSimulationManager.CreatePerceptionSimulationManager(Microsoft.PerceptionSimulation.ISimulationStreamSink)

Create an object for generating simulated packets and delivering them to the provided sink.

Parameters

- sink - The sink that will receive all generated packets

Return value

The created Manager

Microsoft.PerceptionSimulation.PerceptionSimulationManager.CreatePerceptionSimulationRecording(System.String)

Create a sink which stores all received packets in a file at the specified path.

Parameters

- path - The path of the file to create

Return value

The created sink

Microsoft.PerceptionSimulation.PerceptionSimulationManager.LoadPerceptionSimulationRecording(S ystem.String,Microsoft.PerceptionSimulation.ISimulationStreamSinkFactory)

Load a recording from the specified file

Parameters

- path - The path of the file to load
- factory - A factory used by the recording for creating an ISimulationStreamSink when required

Return value

The loaded recording

Microsoft.PerceptionSimulation.PerceptionSimulationManager.LoadPerceptionSimulationRecording(S ystem.String,Microsoft.PerceptionSimulation.ISimulationStreamSinkFactory,Microsoft.PerceptionSimul ation.ISimulationRecordingCallback)

Load a recording from the specified file

Parameters

- path - The path of the file to load
- factory - A factory used by the recording for creating an ISimulationStreamSink when required
- callback - A callback which receives updates regarding the recording's status

Return value

The loaded recording

Microsoft.PerceptionSimulation.StreamDataTypes

Describes the different types of stream data

```
public enum StreamDataTypes
{
    None = 0x00,
    Head = 0x01,
    Hands = 0x02,
    SpatialMapping = 0x08,
    Calibration = 0x10,
    Environment = 0x20,
    All = None | Head | Hands | SpatialMapping | Calibration | Environment
}
```

Microsoft.PerceptionSimulation.StreamDataTypes.None

A sentinel value used to indicate no stream data types

Microsoft.PerceptionSimulation.StreamDataTypes.Head

Stream of data regarding the position and orientation of the head

Microsoft.PerceptionSimulation.StreamDataTypes.Hands

Stream of data regarding the position and gestures of hands

Microsoft.PerceptionSimulation.StreamDataTypes.SpatialMapping

Stream of data regarding spatial mapping of the environment

Microsoft.PerceptionSimulation.StreamDataTypes.Calibration

Stream of data regarding calibration of the device. Calibration packets are only accepted by a system in Remote Mode.

Microsoft.PerceptionSimulation.StreamDataTypes.Environment

Stream of data regarding the environment of the device

Microsoft.PerceptionSimulation.StreamDataTypes.All

A sentinel value used to indicate all recorded data types

Microsoft.PerceptionSimulation.ISimulationStreamSink

An object that receives data packets from a simulation stream

```
public interface ISimulationStreamSink
{
    void OnPacketReceived(uint length, byte[] packet);
}
```

Microsoft.PerceptionSimulation.ISimulationStreamSink.OnPacketReceived(uint length, byte[] packet)

Receives a single packet, which is internally typed and versioned

Parameters

- length - The length of the packet
- packet - The data of the packet

Microsoft.PerceptionSimulation.ISimulationStreamSinkFactory

An object that creates ISimulationStreamSink

```
public interface ISimulationStreamSinkFactory
{
    ISimulationStreamSink CreateSimulationStreamSink();
}
```

Microsoft.PerceptionSimulation.ISimulationStreamSinkFactory.CreateSimulationStreamSink()

Creates a single instance of ISimulationStreamSink

Return value

The created sink

Implement 3D app launchers (UWP apps)

11/6/2018 • 7 minutes to read • [Edit Online](#)

NOTE

This feature was added as part of the 2017 Fall Creators Update (RS3) for immersive headsets and is supported by HoloLens with the Windows 10 April 2018 Update. Make sure your application is targeting a version of the Windows SDK greater than or equal to 10.0.16299 on immersive Headsets and 10.0.17125 on HoloLens. You can find the latest Windows SDK [here](#).

The [Windows Mixed Reality home](#) is the starting point where users land before launching applications. When creating a UWP application for Windows Mixed Reality, by default, apps are launched as 2D slates with their app's logo. When developing experiences for Windows Mixed Reality, a 3D launcher can optionally be defined to override the default 2D launcher for your application. In general, 3D launchers are recommended for launching immersive applications that take users out of the Windows Mixed Reality home whereas the default 2D launcher is preferred when the app is activated in place. You can also create a [3D deep link \(secondaryTile\)](#) as a 3D launcher to content within a 2D UWP app.

3D app launcher creation process

There are 3 steps to creating a 3D app launcher:

1. [Designing and conceiving](#)
2. [Modeling and exporting](#)
3. Integrating it into your application (this article)

3D assets to be used as launchers for your application should be authored using the [Windows Mixed Reality authoring guidelines](#) to ensure compatibility. Assets that fail to meet this authoring specification will not be rendered in the Windows Mixed Reality home.

Configuring the 3D launcher

When you create a new project in Visual Studio, it creates a simple default tile that displays your app's name and logo. To replace this 2D representation with a custom 3D model edit the app manifest of your application to include the "MixedRealityModel" element as part of your default tile definition. To revert to the 2D launcher just remove the MixedRealityModel definition from the manifest.

XML

First, locate the app package manifest in your current project. By default, the manifest will be named Package.appxmanifest. If you're using Visual Studio, then right-click the manifest in your solution viewer and select **View source** to open the xml for editing.

At the top of the manifest, add the uap5 schema and include it as an ignorable namespace:

```

<Package xmlns:mp="http://schemas.microsoft.com/appx/2014/phone/manifest"
    xmlns:uap="http://schemas.microsoft.com/appx/manifest/uap/windows10"
    xmlns:uap2="http://schemas.microsoft.com/appx/manifest/uap/windows10/2"
    xmlns:uap5="http://schemas.microsoft.com/appx/manifest/uap/windows10/5"
    IgnorableNamespaces="uap uap2 uap5 mp"
    xmlns="http://schemas.microsoft.com/appx/manifest/foundation/windows10">

```

Next specify the "MixedRealityModel" in the default tile for your application:

```

<Applications>
    <Application Id="App"
        Executable="$targetnametoken$.exe"
        EntryPoint="ExampleApp.App">
        <uap:VisualElements
            DisplayName="ExampleApp"
            Square150x150Logo="Assets\Logo.png"
            Square44x44Logo="Assets\SmallLogo.png"
            Description="ExampleApp"
            BackgroundColor="#464646">
            <uap:DefaultTile Wide310x150Logo="Assets\WideLogo.png" >
                <uap5:MixedRealityModel Path="Assets\My3DTile.glb" />
            </uap:DefaultTile>
            <uap:SplashScreen Image="Assets\SplashScreen.png" />
        </uap:VisualElements>
    </Application>
</Applications>

```

The MixedRealityModel elements accepts a file path pointing to a 3D asset stored in your app package. Currently only 3D models delivered using the .glb file format and authored against the [Windows Mixed Reality 3D asset authoring instructions](#) are supported. Assets must be stored in the app package and animation is not currently supported. If the "Path" parameter is left blank Windows will show the 2D slate instead of the 3D launcher. **Note:** the .glb asset must be marked as "Content" in your build settings before building and running your app.



Select the .glb in your solution explorer and use the properties section to mark it as "Content" in the build settings

Bounding box

A bounding box can be used to optionally add an additional buffer region around the object. The bounding box is specified using a center point and extents which indicate the distance from the center of the bounding box to its edges along each axis. Units for the bounding box can be mapped to 1 unit = 1 meter. If a bounding box is not provided then one will be automatically fitted to the mesh of the object. If the provided bounding box is smaller than the model then it will be resized to fit the mesh.

Support for the bounding box attribute will come with the Windows RS4 update as a property on the MixedRealityModel element. To define a bounding box first at the top of the app manifest add the uap6 schema and include it them as ignorable namespaces:

```
<Package xmlns:mp="http://schemas.microsoft.com/appx/2014/phone/manifest"
    xmlns:uap="http://schemas.microsoft.com/appx/manifest/uap/windows10"
    xmlns:uap2="http://schemas.microsoft.com/appx/manifest/uap/windows10/2"
    xmlns:uap5="http://schemas.microsoft.com/appx/manifest/uap/windows10/5"
    xmlns:uap6="http://schemas.microsoft.com/appx/manifest/uap/windows10/6"
    IgnorableNamespaces="uap uap2 uap5 uap6 mp"
    xmlns="http://schemas.microsoft.com/appx/manifest/foundation/windows10">
```

Next, on the MixedRealityModel set the SpatialBoundingBox property to define the bounding box:

```
<uap:DefaultTile Wide310x150Logo="Assets\WideLogo.png" >
    <uap5:MixedRealityModel Path="Assets\My3DTile.glb">
        <uap6:SpatialBoundingBox Center="1,-2,3" Extents="1,2,3" />
    </uap5:MixedRealityModel>
</uap:DefaultTile>
```

Using Unity

When working with Unity the project must be built and opened in Visual Studio before the App Manifest can be edited.

NOTE

The 3D launcher must be redefined in the manifest when building and deploying a new Visual Studio solution from Unity.

3D deep links (secondaryTiles)

NOTE

This feature was added as part of the 2017 Fall Creators Update (RS3) for immersive (VR) headsets and as part of the April 2018 Update (RS4) for HoloLens. Make sure your application is targeting a version of the Windows SDK greater than or equal to 10.0.16299 on immersive (VR) headsets and 10.0.17125 on HoloLens. You can find the latest Windows SDK [here](#).

IMPORTANT

3D deep links (secondaryTiles) only work with 2D UWP apps. You can, however, create a [3D app launcher](#) to launch an exclusive app from the Windows Mixed Reality home.

Your 2D applications can be enhanced for Windows Mixed Reality by adding the ability to place 3D models from your app into the [Windows Mixed Reality home](#) as deep links to content within your 2D app, just like [2D secondary tiles](#) on the Windows Start menu. For example, you can create 360° photospheres that link directly into a 360° photo viewer app, or let users place 3D content from a collection of assets that opens a details page about the author. These are just a couple ways to expand the functionality of your 2D application with 3D content.

Creating a 3D “secondaryTile”

You can place 3D content from your application using “secondaryTiles” by defining a mixed reality model at creation time. Mixed reality models are created by referencing a 3D asset in your app package and optionally defining a bounding box.

NOTE

Creating “secondaryTiles” from within an exclusive view is not currently supported.

```

using Windows.UI.StartScreen;
using Windows.Foundation.Numerics;
using Windows.Perception.Spatial;

// Initialize the tile
SecondaryTile tile = new SecondaryTile("myTileId")
{
    DisplayName = "My Tile",
    Arguments = "myArgs"
};

tile.VisualElements.Square150x150Logo = new Uri("ms-appx:///Assets/MyTile/Square150x150Logo.png");

//Assign 3D model (only ms-appx and ms-appdata are allowed)
TileMixedRealityModel model = tile.VisualElements.MixedRealityModel;
model.Uri = new Uri("ms-appx:///Assets/MyTile/MixedRealityModel.glb");
model.ActivationBehavior = TileMixedRealityModelActivationBehavior.Default;
model.BoundingBox = new SpatialBoundingBox
{
    Center = new Vector3 { X = 1, Y = 0, Z = 0 },
    Extents = new Vector3 { X = 3, Y = 5, Z = 4 }
};

// And place it
await tile.RequestCreateAsync();

```

Bounding box

A bounding box can be used to add an additional buffer region around the object. The bounding box is specified using a center point and extents which indicate the distance from the center of the bounding box to its edges along each axis. Units for the bounding box can be mapped to 1 unit = 1 meter. If a bounding box is not provided then one will be automatically fitted to the mesh of the object. If the provided bounding box is smaller than the model then it will be resized to fit the mesh.

Activation behavior

NOTE

This feature will be supported as of the Windows RS4 update. Make sure your application is targeting a version of the Windows SDK greater than or equal to 10.0.17125 if you plan to use this feature

You can define the activation behavior for a 3D secondaryTile to control how it reacts when a user selects it. This can be used to place 3D objects in the Mixed Reality home that are purely informative or decorative. The following activation behavior types are supported:

1. Default: When a user selects the 3D secondaryTile the app is activated
2. None: When the user selects the 3D secondaryTile nothing happens and the app is not activated.

Obtaining and updating an existing “secondaryTile”

Developers can get back a list of their existing secondary tiles, which includes the properties that they previously specified. They can also update the properties by changing the value and then calling UpdateAsync().

```
// Grab the existing secondary tile
SecondaryTile tile = (await SecondaryTile.FindAllAsync()).First();

Uri updatedUri = new Uri("ms-appdata:///local/MixedRealityUpdated.glb");

// See if the model needs updating
if (!tile.VisualElements.MixedRealityModel.Uri.Equals(updatedUri))
{
    // Update it
    tile.VisualElements.MixedRealityModel.Uri = updatedUri;

    // And apply the changes
    await tile.UpdateAsync();
}
```

Checking that the user is in Windows Mixed Reality

3D deep links (secondaryTiles) can only be created while the view is being displayed in a Windows Mixed Reality headset. When your view isn't being presented in a Windows Mixed Reality headset we recommend gracefully handling this by either hiding the entry point or showing an error message. You can check this by querying [IsCurrentViewPresentedOnHolographic\(\)](#).

Tile notifications

Tile notifications do not currently support sending an update with a 3D asset. This means that developers will not be able to do the following

- Push Notifications
- Periodic Polling
- Scheduled Notifications

For more information on the other tiles features and attributes and how they are used for 2D tiles refer to the [Tiles for UWP Apps documentation](#).

See also

- [Mixed reality model sample](#) containing a 3D app launcher.
- [3D app launcher design guidance](#)
- [Creating 3D models for use in the Windows Mixed Reality home](#)
- [Implementing 3D app launchers \(Win32 apps\)](#)
- [Navigating the Windows Mixed Reality home](#)

Implement 3D app launchers (Win32 apps)

11/6/2018 • 5 minutes to read • [Edit Online](#)

NOTE

This feature is only available to PCs running the latest [Windows Insider](#) flights (RS5), build 17704 and newer.

The [Windows Mixed Reality home](#) is the starting point where users land before launching applications. By default, immersive Win32 VR apps and games have to be launched from outside the headset and won't appear in the "All apps" list on the Windows Mixed Reality Start menu. However, by following the instructions in this article to implement a 3D app launcher, your immersive Win32 VR experience can be launched from within the Windows Mixed Reality Start menu and home environment.

This is only true for immersive Win32 VR experiences distributed outside of Steam. For VR experiences [distributed through Steam](#), we've [updated the Windows Mixed Reality for SteamVR Beta](#) along with the latest Windows Insider RS5 flights so that SteamVR titles show up in the Windows Mixed Reality Start menu in the "All apps" list automatically using a default launcher. In other words, the method described in this article is unnecessary for SteamVR titles and will be overridden by the Windows Mixed Reality for SteamVR Beta functionality.

3D app launcher creation process

There are 3 steps to creating a 3D app launcher:

1. [Designing and conceiving](#)
2. [Modeling and exporting](#)
3. Integrating it into your application (this article)

3D assets to be used as launchers for your application should be authored using the [Windows Mixed Reality authoring guidelines](#) to ensure compatibility. Assets that fail to meet this authoring specification will not be rendered in the Windows Mixed Reality home.

Configuring the 3D launcher

Win32 applications will appear in the "All apps" list on the Windows Mixed Reality Start menu if you create a 3D app launcher for them. To do that, create a [Visual Elements Manifest](#) XML file referencing the 3D App Launcher by following these steps:

1. Create a **3D App Launcher asset GLB file** (See [Modeling and exporting](#)).
2. Create a **Visual Elements Manifest** for your application.
 - a. You can start with the [sample below](#). See the full [Visual Elements Manifest](#) documentation for more details.
 - b. Update **Square150x150Logo** and **Square70x70Logo** with a PNG/JPG/GIF for your app.
 - These will be used for the app's 2D logo in the Windows Mixed Reality All Apps list and for the Start Menu on desktop.
 - The file path is relative to the folder containing the Visual Elements Manifest.
 - You still need to provide a desktop Start Menu icon for your app through the standard mechanisms. This can either be directly in the executable or in the shortcut you create (e.g. via `IShellLink::SetIconLocation`).

- *Optional:* You can use a resources.pri file if you would like for MRT to provide multiple asset sizes for different resolution scales and high contrast themes.
- Update the **MixedRealityModel Path** to point to the GLB for your 3D App Launcher
 - Save the file with the same name as your executable file, with an extension of ".VisualElementsManifest.xml" and save it in the same directory. For example, for the executable file "contoso.exe", the accompanying XML file is named "contoso.visualelementsmanifest.xml".
- Add a shortcut** to your application to the desktop Windows Start Menu. See the [sample below](#) for an example C++ implementation.
 - Create it in %ALLUSERSPROFILE%\Microsoft\Windows\Start Menu\Programs (machine) or %APPDATA%\Microsoft\Windows\Start Menu\Programs (user)
 - If an update changes your visual elements manifest or the assets referenced by it, the updater or installer should update the shortcut such that the manifest is reparsed and cached assets are updated.
 - Optional:* If your desktop shortcut does not point directly to your application's EXE (e.g., if it invokes a custom protocol handler like "myapp://"), the Start Menu won't automatically find the app's VisualElementsManifest.xml file. To resolve this, the shortcut should specify the file path of the Visual Elements Manifest using System.AppUserModel.VisualElementsManifestHintPath(). This can be set in the shortcut using the same techniques as System.AppUserModel.ID. You are not required to use System.AppUserModel.ID but you may do so if you wish for the shortcut to match the explicit Application User Model ID of the application if one is used. See the [sample app launcher shortcut creation](#) section below for a C++ sample.

Sample Visual Elements Manifest

```
<Application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <VisualElements>
    ShowNameOnSquare150x150Logo="on"
    Square150x150Logo="YOUR_APP_LOGO_150X150.png"
    Square70x70Logo=" YOUR_APP_LOGO_70X70.png"
    ForegroundColor="#000000">
      <MixedRealityModel Path="YOUR_3D_APP_LAUNCHER_ASSET.glb">
        <SpatialBoundingBox Center="0,0,0" Extents="Auto" />
      </MixedRealityModel>
    </VisualElements>
  </Application>
```

Sample app launcher shortcut creation

The sample code below shows how you can create a shortcut in C++, including overriding the path to the Visual Elements Manifest XML file. Note the override is only required in cases where your shortcut does not point directly to the EXE associated with the manifest (eg. your shortcut uses a custom protocol handler like "myapp://").

Sample .LNK shortcut creation (C++)

```
#include <windows.h>
#include <propkey.h>
#include <shlobj_core.h>
#include <shlwapi.h>
#include <propvarutil.h>
#include <wrl.h>

#include <memory>

using namespace Microsoft::WRL;

#define RETURN_IF_FAILED(x) do { HRESULT hr = x; if (FAILED(hr)) { return hr; } } while(0)
#define RETURN_IF_WIN32_BOOL_FALSE(x) do { DWORD res = x; if (res == 0) { return
HRESULT_FROM_WIN32(GetLastError()); } } while(0)

int wmain()
{
```

```

    RETURN_IF_FAILED(CoInitializeEx(nullptr, COINIT_APARTMENTTHREADED));

    ComPtr shellLink;
    RETURN_IF_FAILED(CoCreateInstance(__uuidof(ShellLink), nullptr, CLSCTX_INPROC_SERVER,
IID_PPV_ARGS(&shellLink)));
    RETURN_IF_FAILED(shellLink->SetPath(L"MyLauncher://launch/app-identifier"));

    // It is also possible to use an icon file in another location. For example, "C:\Program Files
    // (x86)\MyLauncher\assets\app-identifier.ico".
    RETURN_IF_FAILED(shellLink->SetIconLocation(L"C:\\\\Program Files (x86)\\\\MyLauncher\\\\apps\\\\app-
    identifier\\\\game.exe", 0 /*iIcon*/));

    ComPtr propStore;
    RETURN_IF_FAILED(shellLink.As(&propStore));

    {
        // Optional: If the application has an explicit Application User Model ID, then you should usually
        // specify it in the shortcut.
        PROPVARIANT propVar;
        RETURN_IF_FAILED(InitPropVariantFromString(L"ExplicitAppUserModelID", &propVar));
        RETURN_IF_FAILED(propStore->SetValue(PKEY_AppUserModel_ID, propVar));
        PropVariantClear(&propVar);
    }

    {
        // A hint path to the manifest is only necessary if the target path of the shortcut is not a file path
        // to the executable.
        // By convention the manifest is named <executable name>.VisualElementsManifest.xml and is in the same
        // folder as the executable
        // (and resources.pri if applicable). Assets referenced by the manifest are relative to the folder
        // containing the manifest.

        //
        // PropKey.h
        //
        // Name:      System.AppUserModel.VisualElementsManifestHintPath --
PKEY_AppUserModel_VisualElementsManifestHintPath
        // Type:      String -- VT_LPWSTR (For variants: VT_BSTR)
        // FormatID: {9F4C2855-9F79-4B39-A8D0-E1D42DE1D5F3}, 31
        //
        // Suggests where to look for the VisualElementsManifest for a Win32 app
        //
        // DEFINE_PROPERTYKEY(PKEY_AppUserModel_VisualElementsManifestHintPath, 0x9F4C2855, 0x9F79, 0x4B39,
0xA8, 0xD0, 0xE1, 0xD4, 0x2D, 0xE1, 0xD5, 0xF3, 31);
        // #define INIT_PKEY_AppUserModel_VisualElementsManifestHintPath { { 0x9F4C2855, 0x9F79, 0x4B39, 0xA8,
0xD0, 0xE1, 0xD4, 0x2D, 0xE1, 0xD5, 0xF3 }, 31 }

        PROPVARIANT propVar;
        RETURN_IF_FAILED(InitPropVariantFromString(L"C:\\\\Program Files (x86)\\\\MyLauncher\\\\apps\\\\app-
        identifier\\\\game.visualelementsmanifest.xml", &propVar));
        RETURN_IF_FAILED(propStore->SetValue(PKEY_AppUserModel_VisualElementsManifestHintPath, propVar));
        PropVariantClear(&propVar);
    }

    constexpr PCWSTR shortcutPath = L"%APPDATA%\\Microsoft\\Windows\\Start Menu\\Programs\\game.lnk";
    const DWORD requiredBufferLength = ExpandEnvironmentStrings(shortcutPath, nullptr, 0);
    RETURN_IF_WIN32_BOOL_FALSE(requiredBufferLength);

    const auto expandedShortcutPath = std::make_unique<wchar_t[]>(requiredBufferLength);
    RETURN_IF_WIN32_BOOL_FALSE(ExpandEnvironmentStrings(shortcutPath, expandedShortcutPath.get(),
requiredBufferLength));

    ComPtr persistFile;
    RETURN_IF_FAILED(shellLink.As(&persistentFile));
    RETURN_IF_FAILED(persistentFile->Save(expandedShortcutPath.get(), FALSE));

    return 0;
}

```

Sample .URL launcher shortcut

```
[{9F4C2855-9F79-4B39-A8D0-E1D42DE1D5F3}]
Prop31=C:\Program Files (x86)\MyLauncher\apps\app-identifier\game.visualelementsmanifest.xml
Prop5=ExplicitAppUserModelID

[{000214A0-0000-0000-C000-000000000046}]
Prop3=19,0

[InternetShortcut]
IDList=
URL=MyLauncher://launch/app-identifier
IconFile=C:\Program Files (x86)\MyLauncher\apps\app-identifier\game.exe
IconIndex=0
```

See also

- [Mixed reality model sample](#) containing a 3D app launcher.
- [3D app launcher design guidance](#)
- [Creating 3D models for use in the Windows Mixed Reality home](#)
- [Implementing 3D app launchers \(UWP apps\)](#)
- [Navigating the Windows Mixed Reality home](#)

Enable placement of 3D models in the mixed reality home

11/6/2018 • 2 minutes to read • [Edit Online](#)

NOTE

This feature was added as part of the [Windows 10 April 2018 Update](#). Older versions of Windows are not compatible with this feature.

The [Windows Mixed Reality home](#) is the starting point where users land before launching applications. In some scenarios, 2D apps (like the Holograms app) enable placement of 3D models directly into the mixed reality home as decorations or for further inspection in full 3D. The *add model protocol* allows you to send a 3D model from your website or application directly into the Windows Mixed Reality home, where it will persist like [3D app launchers](#), 2D apps, and holograms.

For example, if you're developing an application that surfaces a catalog of 3D furniture for designing a space, you can use the *add model protocol* to allow users to place those 3D furniture models from the catalog. Once placed in the world, users can move, resize, and remove these 3D models just like other holograms in the home. This article provides an overview of implementing the *add model protocol* so that you can start enabling users to decorate their world with 3D objects from your app or the web.

Device support

FEATURE	HOLOLENS	IMMERSIVE HEADSETS
Add model protocol	✓ <input type="checkbox"/>	✓ <input type="checkbox"/>

Overview

There are 2 steps to enabling the placement of 3D models in the Windows Mixed Reality home:

1. [Ensure your 3D model is compatible with the Windows Mixed Reality home](#).
2. Implement the *add model protocol* in your application or webpage (this article).

Implementing the *add model protocol*

Once you have a [compatible 3D model](#), you can implement the *add model protocol* by activating the following URI from any webpage or application:

```
ms-mixedreality:addmodel?uri=<Path to a .glb 3D model either local or remote>
```

If the URI points to a remote resource, then it will automatically be downloaded and placed in the home. Local resources will be copied to the mixed reality home's app data folder before being placed in the home. We recommend designing your experience to account for scenarios where the user might be running an older version of Windows that doesn't support this feature by hiding the button or disabling it if possible.

Invoking the *add model protocol* from a Universal Windows Platform app:

```
private async void launchURI_Click(object sender, RoutedEventArgs e)
{
    // Define the add model URI
    var uriAddModel = new Uri(@"ms-mixedreality:addModel?uri=sample.glb");

    // Launch the URI to invoke the placement
    var success = await Windows.System.Launcher.LaunchUriAsync(uriAddModel);

    if (success)
    {
        // URI launched
    }
    else
    {
        // URI launch failed
    }
}
```

Invoking the *add model protocol* from a webpage:

```
<a class="btn btn-default" href="ms-mixedreality:addModel?uri=sample.glb"> Place 3D Model </a>
```

Considerations for immersive (VR) headsets

- For immersive (VR) headsets, the Mixed Reality Portal does not have to be running before invoking the *add model protocol*. In this case, the *add model protocol* will launch the Mixed Reality Portal and place the object directly where the headset is looking once you arrive in the mixed reality home.
- When invoking the *add model protocol* from the desktop with the Mixed Reality Portal already running, ensure that the headset is "awake". If not, the placement will not succeed.

See also

- [Creating 3D models for use in the Windows Mixed Reality home](#)
- [Navigating the Windows Mixed Reality home](#)

In-app purchases

11/6/2018 • 2 minutes to read • [Edit Online](#)

In-app purchases are supported in HoloLens, but there is some work to set them up.

In order to leverage the in app-purchase functionality, you must:

- Create a XAML [2D view](#) to appear as a slate
- Switch to it to activate placement, which leaves the immersive view
- Call the API: await `CurrentApp.RequestProductPurchaseAsync("DurableItemIAPName");`

This API will bring up the stock Windows OS popup that shows the in-app purchase name, description, and price.

The user can then choose purchase options. Once the action is completed, the app will need to present UI which allows the user to switch back to its [immersive view](#).

For apps targeting desktop-based Windows Mixed Reality immersive headsets, it is not required to manually switch to a XAML view before calling the RequestProductPurchaseAsync API.

Submitting an app to the Microsoft Store

11/9/2018 • 13 minutes to read • [Edit Online](#)

Both [HoloLens](#) and the Windows 10 PC powering your [immersive headset](#) run Universal Windows Platform apps. Whether you're submitting an app that supports HoloLens or PC (or both), you'll submit your app to the Microsoft Store through [Partner Center](#).

If you don't already have a Partner Center developer account, you can [sign up today](#).

Packaging a mixed reality app

Prepare image assets included in the appx

There are several image assets required by the appx building tools to build your application into an appx package to submit to the Store. You can learn more about [guidelines for tile and icon assets](#) on MSDN.

REQUIRED ASSET	RECOMMENDED SCALE	IMAGE FORMAT	WHERE IS THIS DISPLAYED?
Square 71x71 Logo	Any	PNG	N/A
Square 150x150 Logo	150x150 (100% scale) or 225x225 (150% scale)	PNG	Start pins and All Apps (if 310x310 isn't provided), Store Search Suggestions, Store Listing Page, Store Browse, Store Search
Wide 310x150 Logo	Any	PNG	N/A
Store Logo	75x75 (150% scale)	PNG	Partner Center, Report App, Write a Review, My Library
Splash Screen	930x450 (150% scale)	PNG	2D app launcher (slate)

There are also some recommended assets which HoloLens can take advantage of.

RECOMMENDED ASSETS	RECOMMENDED SCALE	WHERE IS THIS DISPLAYED?
Square 310x310 Logo	310x310 (150% scale)	Start pins and All Apps

Live Tile requirements

The Start menu on HoloLens will use the largest included square tile image.

You may see that some apps published by Microsoft have a 3D launcher for their application. Developers can add a 3D launcher for their app using [these instructions](#).

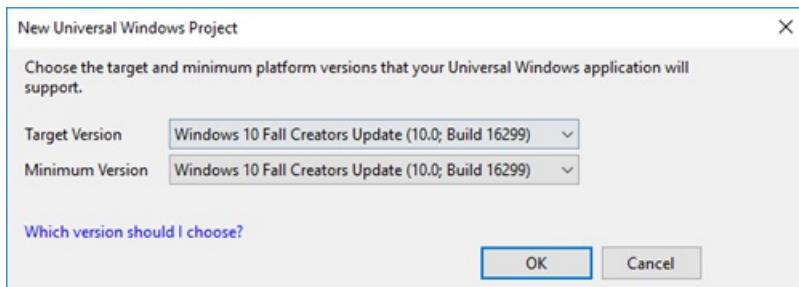
Specifying target and minimum version of Windows

If your mixed reality app includes features that are specific to a certain version of Windows, it's important to specify the target and minimum platform versions that your Universal Windows application will support.

This is especially true for apps targeting [Windows Mixed Reality immersive headsets](#), which require at least the Windows 10 Fall Creators Update (10.0; Build 16299) to function properly.

You will be prompted to set target and minimum version of Windows when you create a new Universal Windows

Project in Visual Studio. You can also change this setting for an existing project in the "Project" menu, then "<Your app name's> Properties" at the bottom of the drop-down menu.



Set minimum and target platform versions in Visual Studio

Specifying target device families

Windows Mixed Reality applications (for both **HoloLens** and **immersive headsets**) are part of the Universal Windows Platform, so any app package with a **target device family** of "Windows.Universal" is capable of running on HoloLens or Windows 10 PCs with immersive headsets. That being said, if you do not specify a target device family in your app manifest you may inadvertently open your app up to unintended Windows 10 devices. Follow the steps below to specify the intended Windows 10 device family, and then **double-check that the correct device families are selected when you upload your app package in Partner Center to submit to the Store**.

To set this field in Visual Studio, right click on the Package.appxmanifest and select "View Code" then find the TargetDeviceFamily Name field. By default, it might look like the following:

```
<Dependencies>
  <TargetDeviceFamily Name="Windows.Universal" MinVersion="10.0.10240.0" MaxVersionTested="10.0.10586.0" />
</Dependencies>
```

If your app is created for **HoloLens**, then you can ensure that it is only installed on HoloLens by specifying a target device family of "Windows.Holographic".

```
<Dependencies>
  <TargetDeviceFamily Name="Windows.Holographic" MinVersion="10.0.10240.0" MaxVersionTested="10.0.10586.0"
  />
</Dependencies>
```

If your app is created for **Windows Mixed Reality immersive headsets**, then you can ensure that it is only installed on Windows 10 PCs with the Windows 10 Fall Creators Update (necessary for Windows Mixed Reality) by specifying a target device family of "Windows.Desktop" and MinVersion of "10.0.16299.0".

```
<Dependencies>
  <TargetDeviceFamily Name="Windows.Desktop" MinVersion="10.0.16299.0" MaxVersionTested="10.0.16299.0" />
</Dependencies>
```

Finally, if your app is intended to run on both **HoloLens and Windows Mixed Reality immersive headsets**, you can ensure the app is only made available to those two device families and simultaneously ensure each targets the correct minimum Windows version by including a line for each target device family with its respective MinVersion.

```
<Dependencies>
  <TargetDeviceFamily Name="Windows.Desktop" MinVersion="10.0.16299.0" MaxVersionTested="10.0.16299.0" />
  <TargetDeviceFamily Name="Windows.Holographic" MinVersion="10.0.10240.0" MaxVersionTested="10.0.10586.0"
  />
</Dependencies>
```

You can learn more about targeting device families by reading the [TargetDeviceFamily UWP documentation](#).

Associate app with the Store

From the Project menu in your Visual Studio solution, choose "Store > Associate App with the Store". If you do this, you can test purchase and notification scenarios in your app. When you associate your app with the Store, these values are downloaded to the app manifest file for the current project on your local machine:

- Package Display Name
- Package Name
- Publisher ID
- Publisher Display Name
- Version

If you override the default package.appxmanifest file by creating a custom .xml file for the manifest, you can't associate your app with the Store. If you try to associate a custom manifest file with the Store, you will see an error message.

Creating an upload package

Follow guidelines at [Packaging Universal Windows apps for Windows 10](#).

The final step of creating an upload package is validating the package using the [Windows App Certification Kit](#).

If you'll be adding a package specifically for HoloLens to an existing product that is available on other Windows 10 device families, you will also want to learn about [how version numbers may impact which packages are delivered to specific customers](#), and [how packages are distributed to different operating systems](#).

The general guidance is that the highest version number package that is applicable to a device will be the one distributed by the Store.

If there is a Windows.Universal package and a Windows.Holographic package and the Windows.Universal package has a higher version number, a HoloLens user will download the higher version number Windows.Universal package instead of the Windows.Holographic package. There are several solutions to this problem:

1. Ensure your platform specific packages such as Windows.Holographic always have a higher version number than your platform agnostic packages such as Windows.Universal
2. Do not package apps as Windows.Universal if you also have platform specific packages - instead package the Windows.Universal package for the specific platforms you want it available on

NOTE

You can declare a single package to be applicable to multiple target device families

3. Create a single Windows.Universal package that works across all platforms. Support for this isn't great right now so the above solutions are recommended.

Testing your app

Windows App Certification Kit

When you create app packages to submit to Partner Center through Visual Studio, the Create App Packages wizard will prompt you to run the Windows App Certification Kit against the packages that get created. In order to have a smooth submission process to the Store, it's best to verify that the [Windows App Certification Kit tests](#) pass against your app on your local machine before submitting them to the Store. Running the Windows App Certification Kit on a remote HoloLens is not currently supported.

Run on all targeted device families

The Windows Universal Platform allows you to create a single application that runs across all of the Windows 10 device families. However, it doesn't guarantee that Universal Windows apps will just work on all device families. Before you choose to make your app available on HoloLens or any other Windows 10 target device family, it's important that you [test the app](#) on each of those device families to ensure a good experience.

Submitting your mixed reality app to the Store

If you are submitting a mixed reality app that is based on a Unity project, please see this [video](#) first.

In general, submitting a Windows Mixed Reality app that works on HoloLens and/or immersive headsets is just like submitting any UWP app to the Microsoft Store. Once you've [created your app by reserving its name](#), you should follow the [UWP submission checklist](#).

One of the first things you'll do is [select a category and sub-category](#) for your mixed reality experience. It's important that you **choose the most accurate category for your app** so that we can merchandise your application in the right Store categories and ensure it shows up using relevant search queries. **Listing your VR title as a game will not result in better exposure for your app**, and may prevent it from showing up in categories that are more fitting and less crowded.

However, there are four key areas in the submission process where you'll want to make mixed reality-specific selections:

1. In the [Product declarations](#) section under [Properties](#).
2. In the [System requirements](#) section under [Properties](#).
3. In the [Device family availability](#) section under [Packages](#).
4. In several of the [Store listing page](#) fields.

Mixed reality product declarations

On the [Properties](#) page of the app submission process, you'll find several options related to mixed reality in the [Product declarations](#) section.

The screenshot shows the 'Product declarations' section of the Windows App Submission interface. On the left is a sidebar with navigation links: Submissions, Submission 7, Pricing and availability, Properties, Age ratings, Packages, Store listings, Manage languages, Notes for certification, and Add-ons. The main area is titled 'System requirements'. It contains several checkboxes and dropdown menus:

- Customers can use Windows 10 features to record and broadcast clips of this game.
- This app sends Kinect data to external services.
- This product supports Cortana.
- This product supports 4K resolution video output. PC Xbox
- This product supports High Dynamic Range (HDR) video PC Xbox
- This experience is designed for Windows Mixed Reality on:
 - PC HoloLens
 - Seated + standing - This app will be used in a seated or standing position, does not require moving around, and can be used without a user-created boundary.
 - All experiences - This app allows or requires the user to move around, so they will need to create a boundary during setup.
- Mixed reality setup

Mixed reality product declarations

First, you'll want to identify the device types for which your app offers a mixed reality experience. This ensures that your app is included in Windows Mixed Reality collections in the Store, and that it's surfaced to users browsing the Store after connecting an immersive headset (or when browsing the Store on HoloLens).

Next to "This experience is designed for Windows Mixed Reality on:"

- Check the **PC** box only if your app offers a VR experience when an immersive headset is connected to the user's PC. You should check this box whether your app is designed exclusively to run on an immersive headset or if it is a standard PC game/app that offers a mixed reality mode and/or bonus content when a headset is connected.
- Check the **HoloLens** box only if your app offers a holographic experience when it's run on HoloLens.

- Check **both** boxes if your app offers a mixed reality experience on both device types, like the [Mixed Reality Academy "Project Island" app](#) from Build 2017.

If you selected "PC" above, you'll want to set the "mixed reality setup" (activity level). This only applies to mixed reality experiences that run on PCs connected to immersive headsets, as mixed reality apps on HoloLens are world-scale and the user doesn't define a boundary during setup.

- Choose **Seated + standing** if your app is designed with the intention that the user stays in one position (an example would be a game where you're seated in a cockpit of an aircraft).
- Choose **All experiences** if your app is designed with the intention that the user walks around within the boundary he or she defined during setup (an example might be a game where you side-step and duck to dodge attacks).

Mixed reality system requirements

On the [Properties](#) page of the app submission process, you'll find several options related to mixed reality in the **System requirements** section.

Microphone	<input type="checkbox"/>	<input type="checkbox"/>
Xbox controller or gamepad	<input type="checkbox"/>	<input type="checkbox"/>
Windows Mixed Reality motion controllers	<input type="checkbox"/>	<input type="checkbox"/>
Windows Mixed Reality immersive headset	<input type="checkbox"/>	<input type="checkbox"/>
Memory	Not Specified ▾	Not Specified ▾
DIRECTX	Not Specified ▾	Not Specified ▾
Video memory	Not Specified ▾	Not Specified ▾
Processor	<input type="text"/>	<input type="text"/>
Graphics	<input type="text"/>	<input type="text"/>

System requirements

In this section, you'll identify minimum (required) hardware and recommended (optional) hardware for your mixed reality app.

Input hardware:

Use the checkboxes to tell potential customers if your app supports **microphone** (for [voice input](#)), **Xbox controller or gamepad**, and/or **Windows Mixed Reality motion controllers**. This information will be surfaced on your app's product detail page in the Store and will help your app get included in the appropriate app/game collections (for example, a collection may exist for all games that support motion controllers).

Be thoughtful about selecting checkboxes for "minimum hardware" or "recommended hardware" for input types.

For example:

- If your game requires motion controllers, but accepts voice input via microphone, select the "minimum hardware" checkbox next to "Windows Mixed Reality motion controllers," but the "recommended hardware" checkbox next to "Microphone."
- If your game can be played with either an Xbox controller/gamepad or motion controllers, you might select the "minimum hardware" checkbox next to "Xbox controller or gamepad" and select the "recommended hardware" checkbox next to "Windows Mixed Reality motion controllers" as motion controllers will likely offer

a step-up in experience from the gamepad.

Windows Mixed Reality immersive headset:

Indicating whether an immersive headset is required to use your app, or is optional, is critical to customer satisfaction and education.

If your app can *only* be used through an immersive headset, select the "minimum hardware" checkbox next to "Windows Mixed Reality immersive headset." This will be surfaced on your app's product detail page in Store as a warning above the purchase button so customers don't think they're purchasing an app that will function on their PC like a traditional desktop app.

If your app runs on the desktop like a traditional PC app, but offers a VR experience when an immersive headset is connected (whether the full content of your app is available, or only a portion), select the "recommended hardware" checkbox next to "Windows Mixed Reality immersive headset." No warning will be surfaced above the purchase button on your app's product detail page if your app functions as a traditional desktop app without an immersive headset connected.

PC specifications:

If you want your app to reach as many Windows Mixed Reality immersive headset users as possible, you'll want to [target the PC specifications for Windows Mixed Reality PCs with integrated graphics](#).

Whether your mixed reality app targets the minimum Windows Mixed Reality PC requirements, or requires a specific PC configuration (like the dedicated GPU of a [Windows Mixed Reality Ultra PC](#)), you should indicate that with the relevant PC specifications in the "minimum hardware" column.

If your mixed reality app is designed to perform better, or offer higher-resolution graphics, on a particular PC configuration or graphics card, you should indicate that with the relevant PC specifications in the "recommended hardware" column.

This only applies if your mixed reality app uses an immersive headset connected to a PC. If your mixed reality app only runs on HoloLens, you won't need to indicate PC specifications as HoloLens has only one hardware configuration.

Device family availability

If you've [packaged your app correctly](#) in Visual Studio, uploading it on the Packages page of the app submission process should produce a table identifying which device families your app will be available to.

Packages	Windows 10 Desktop	Windows 10 Mobile	Windows 10 Xbox	Windows 10 Team	Windows 10 Holographic	Windows 8/8.1	Windows Phone 8x and earlier
Mixed Reality Academy Project Island_1.2.8.0_x86_bundle_Master.appupload v1.2.8.0, Neutral	1	1		1	1		

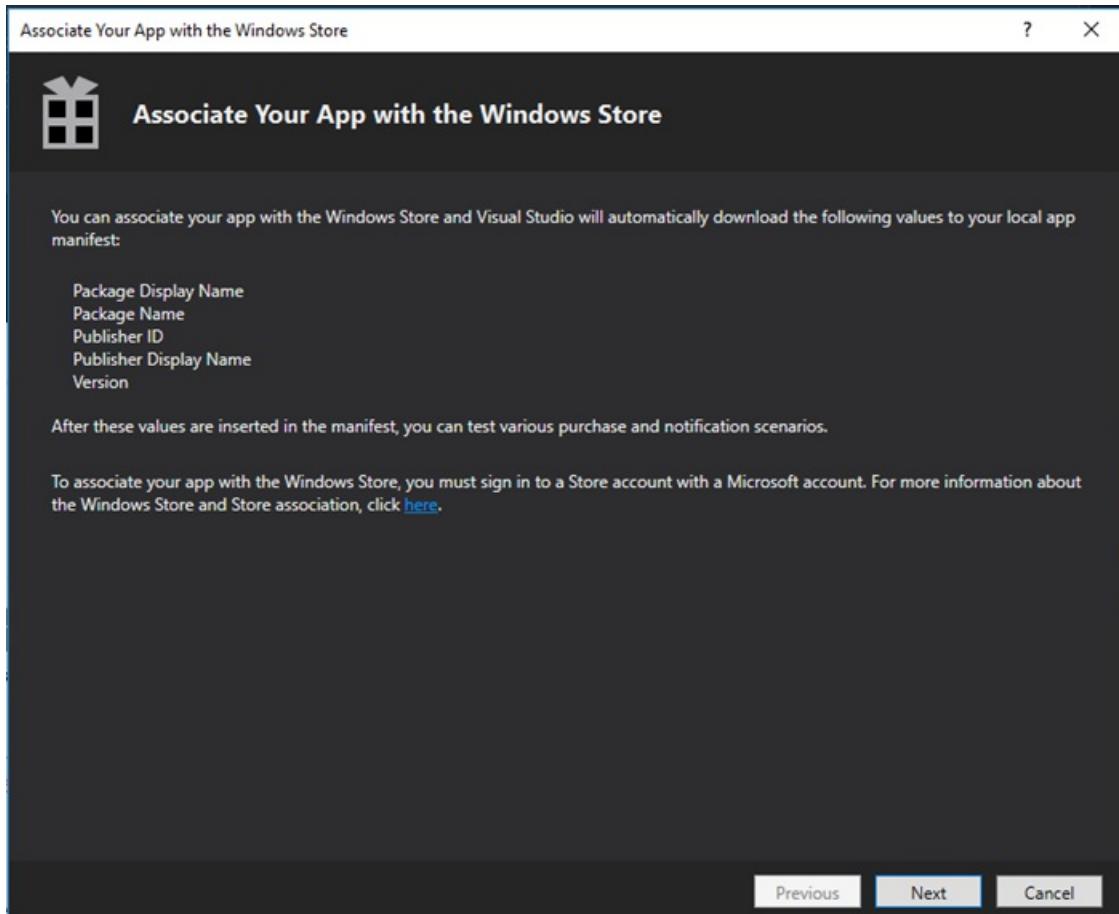
1 Offered to this device family first.
2 Offered if a device can't support a higher-ranked package.
To make packages available to this device family, check its box.

Device family availability table

If your mixed reality app works on immersive headsets, then at least "Windows 10 Desktop" should be selected in the table. If your mixed reality app works on HoloLens, then at least "Windows 10 Holographic" should be selected. If your app runs on both Windows Mixed Reality headset types, like the [Mixed Reality Academy "Project Island" app](#), both "Windows 10 Desktop" and "Windows 10 Holographic" should be selected.

TIP

Many developers run into errors when uploading their app's package related to mismatches between the package manifest and your app/publisher account information in Partner Center. These errors can often be avoided by signing into Visual Studio with the same account associated with your Windows developer account (the one you use to sign into Partner Center). If you use the same account, you'll be able to associate your app with its identity in the Microsoft Store before you package it.



Associate your app with the Microsoft Store in Visual Studio

Store listing page

On the [Store listing](#) page of the app submission process, there are several places you can add useful information about your mixed reality app.

IMPORTANT

To ensure your app is correctly categorized by the Store and made discoverable to Windows Mixed Reality customers, you should add "**Windows Mixed Reality**" as one of your "Search terms" for the app (you can find search terms by expanding the "Shared fields" section).

Shared fields [Hide details](#)

Support contact info

 2048

• Must be a valid URL or email address

Search term

VR	43		MR	43	
----	----	--	----	----	--

• Up to 7 search terms
• 45 character limit for each search term

virtual reality	30		mixed reality	32	
-----------------	----	--	---------------	----	--

Windows Mixed Reality	24	
-----------------------	----	--

[Add more](#)

Add "Windows Mixed Reality" to search terms

Offering a free trial for your game or app

Many consumers will have limited to no experience with virtual reality before buying a Windows Mixed Reality immersive headset. They may not know what to expect from intense games, and may not be familiar with their own comfort threshold in immersive experiences. Many customers may also try a Windows Mixed Reality immersive headset on PCs that aren't badged as [Windows Mixed Reality PCs](#). Because of these considerations, we strongly recommend you consider offering a [free trial](#) for your paid mixed reality app or game.

See also

- [Mixed reality](#)
- [Development overview](#)
- [App views](#)
- [Performance recommendations for immersive headset apps](#)
- [Testing your app on HoloLens](#)
- [Windows Mixed Reality minimum PC hardware compatibility guidelines](#)

Using the Windows Device Portal

11/6/2018 • 12 minutes to read • [Edit Online](#)

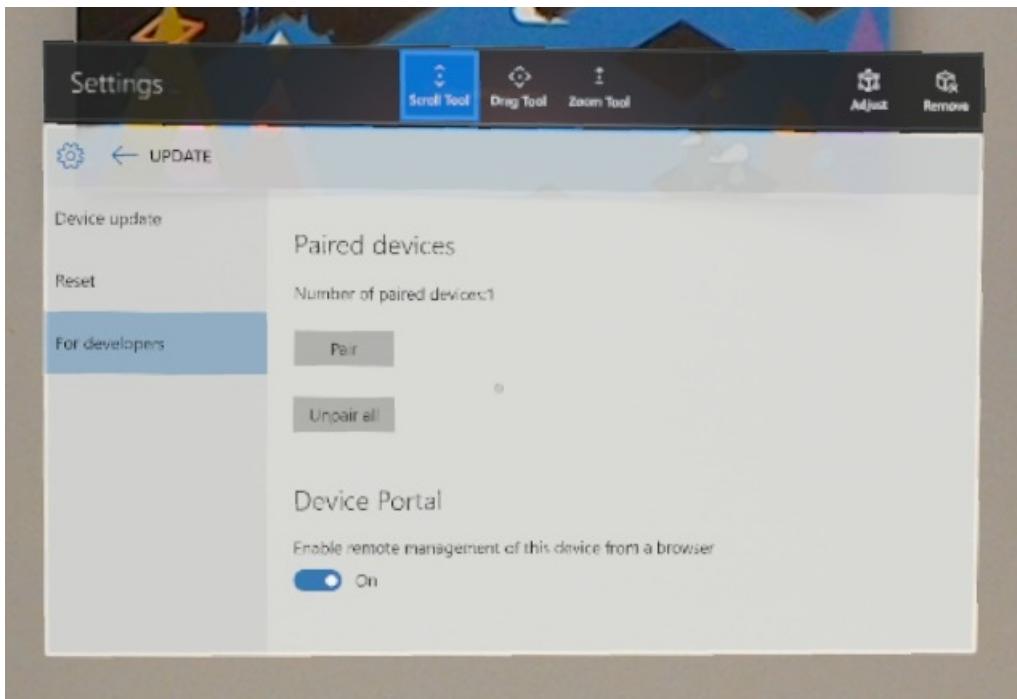
FEATURE	HOLOLENS	IMMERSIVE HEADSETS
Windows Device Portal	✓ <input type="checkbox"/>	

The Windows Device Portal for HoloLens lets you configure and manage your device remotely over Wi-Fi or USB. The Device Portal is a web server on your HoloLens that you can connect to from a web browser on your PC. The Device Portal includes many tools that will help you manage your HoloLens and debug and optimize your apps.

This documentation is specifically about the Windows Device Portal for HoloLens. To use the Windows Device portal for desktop (including for Windows Mixed Reality), see [Windows Device Portal overview](#)

Setting up HoloLens to use Windows Device Portal

1. Power on your HoloLens and put on the device.
2. Perform the [bloom](#) gesture to launch the main menu.
3. Gaze at the **Settings** tile and perform the [air-tap](#) gesture. Perform a second air-tap to place the app in your environment. The Settings app will launch after you place it.
4. Select the **Update** menu item.
5. Select the **For developers** menu item.
6. Enable **Developer Mode**.
7. [Scroll down](#) and enable **Device Portal**.
8. If you are setting up Windows Device Portal so you can deploy apps to this HoloLens over USB or Wi-Fi, click **Pair** to [generate a pairing PIN](#). Leave the Settings app at the PIN popup until you enter the PIN into Visual Studio during your first deployment.



Connecting over Wi-Fi

1. [Connect your HoloLens to Wi-Fi.](#)
2. Look up your device's IP address.
 - Find the IP address on the device under **Settings > Network & Internet > Wi-Fi > Advanced Options.**
3. From a web browser on your PC, go to https://<YOUR_HOLOLENS_IP_ADDRESS>
 - The browser will display the following message: "There's a problem with this website's security certificate". This happens because the certificate which is issued to the Device Portal is a test certificate. You can ignore this certificate error for now and proceed.

Connecting over USB

1. [Install the tools](#) to make sure you have Visual Studio Update 1 with the Windows 10 developer tools installed on your PC. This enables USB connectivity.
2. Connect your HoloLens to your PC with a micro-USB cable.
3. From a web browser on your PC, go to <http://127.0.0.1:10080>.

Connecting to an emulator

You can also use the Device Portal with your emulator. To connect to the Device Portal, use the [toolbar](#). Click on this icon:  **Open Device Portal:** Open the Windows Device Portal for the HoloLens OS in the emulator.

Creating a Username and Password



Set up access

[« Credentials reset](#)

PIN displayed on your device:

New user name:

New password:

Confirm password:

Pair

Set up access to Windows Device Portal

The first time you connect to the Device Portal on your HoloLens, you will need to create a username and password.

1. In a web browser on your PC, enter the IP address of the HoloLens. The Set up access page opens.
2. Click or tap **Request pin** and look at the HoloLens display to get the generated PIN.
3. Enter the PIN in the **PIN displayed on your device** textbox.
4. Enter the user name you will use to connect to the Device Portal. It doesn't need to be a Microsoft Account (MSA) name or a domain name.
5. Enter a password and confirm it. The password must be at least seven characters in length. It doesn't need to be an MSA or domain password.
6. Click **Pair** to connect to Windows Device Portal on the HoloLens.

If you wish to change this username or password at any time, you can repeat this process by visiting the device security page by either clicking the **Security** link along the top right, or navigating to:
https://<YOUR_HOLOLENS_IP_ADDRESS>/devicesecurity.htm.

Security certificate

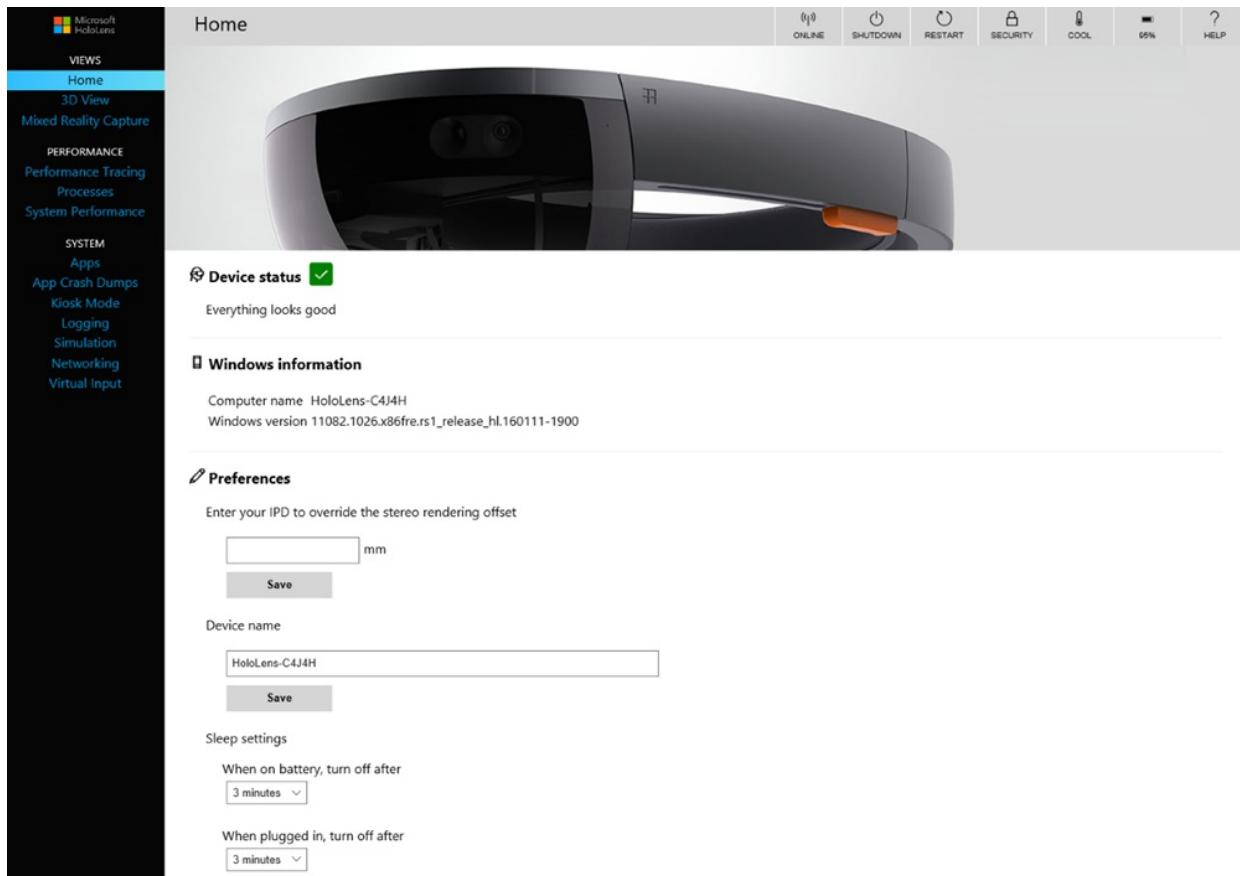
If you see a "certificate error" in your browser, you can fix it by creating a trust relationship with the device.

Each HoloLens generates a unique self-signed certificate for its SSL connection. By default, this certificate is not trusted by your PC's web browser and you may get a "certificate error". By downloading this certificate from your HoloLens (over USB or a Wi-Fi network you trust) and trusting it on your PC, you can securely connect to your device.

1. **Make sure you are on a secure network (USB or a Wi-Fi network you trust).**
2. Download this device's certificate from the "Security" page on the Device Portal.
 - Either click the **Security** link from the top right list of icons or navigate to:
https://<YOUR_HOLOLENS_IP_ADDRESS>/devicesecurity.htm
3. Install the certificate in the "Trusted Root Certification Authorities" store on your PC.
 - From the Windows menu, type: Manage Computer Certificates and start the applet.
 - Expand the **Trusted Root Certification Authority** folder.
 - Click on the **Certificates** folder.
 - From the Action menu, select: All Tasks > Import...
 - Complete the Certificate Import Wizard, using the certificate file you downloaded from the Device Portal.
4. Restart the browser.

Device Portal Pages

Home



The screenshot shows the Microsoft HoloLens Device Portal Home page. On the left, a vertical navigation bar lists categories: VIEWS (Home, 3D View, Mixed Reality Capture), PERFORMANCE (Performance Tracing, Processes, System Performance), and SYSTEM (Apps, App Crash Dumps, Kiosk Mode, Logging, Simulation, Networking, Virtual Input). The main content area features a large image of the Microsoft HoloLens device. At the top right is a toolbar with icons for ONLINE (status: ONLINE), SHUTDOWN, RESTART, SECURITY, COOL, and HELP. Below the toolbar, the status section says "Device status: Everything looks good". The Windows information section displays the computer name as "HoloLens-C4J4H" and the Windows version as "Windows version 11082.1026.x86fre.rs1_release_hl.160111-1900". The Preferences section contains fields for "IPD" (set to 65 mm) and "Device name" (set to "HoloLens-C4J4H"), both with "Save" buttons. Sleep settings dropdowns show "When on battery, turn off after: 3 minutes" and "When plugged in, turn off after: 3 minutes".

Windows Device Portal home page on Microsoft HoloLens

Your Device Portal session starts at the Home page. Access other pages from the navigation bar along the left side of the home page.

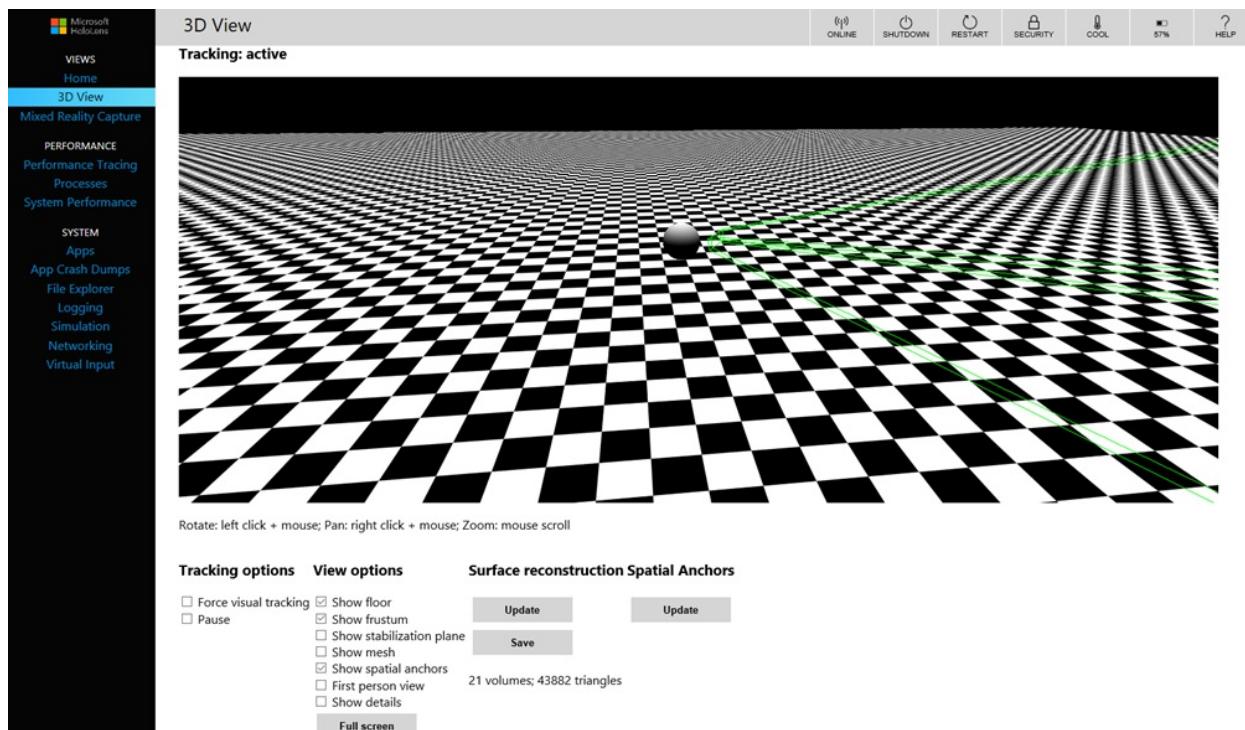
The toolbar at the top of the page provides access to commonly used status and features.

- **Online:** Indicates whether the device is connected to Wi-Fi.
- **Shutdown:** Turns off the device.
- **Restart:** Cycles power on the device.
- **Security:** Opens the Device Security page.
- **Cool:** Indicates the temperature of the device.
- **A/C:** Indicates whether the device is plugged in and charging.
- **Help:** Opens the REST interface documentation page.

The home page shows the following info:

- **Device Status:** monitors the health of your device and reports critical errors.
- **Windows information:** shows the name of the HoloLens and the currently installed version of Windows.
- **Preferences** section contains the following settings:
 - **IPD:** Sets the interpupillary distance (IPD), which is the distance, in millimeters, between the center of the user's pupils when looking straight ahead. The setting takes effect immediately. The default value was calculated automatically when you set up your device.
 - **Device name:** Assign a name to the HoloLens. You must reboot the device after changing this value for it to take effect. After clicking **Save**, a dialog will ask if you want to reboot the device immediately or reboot later.
 - **Sleep settings:** Sets the length of time to wait before the device goes to sleep when it's plugged in and when it's on battery.

3D View

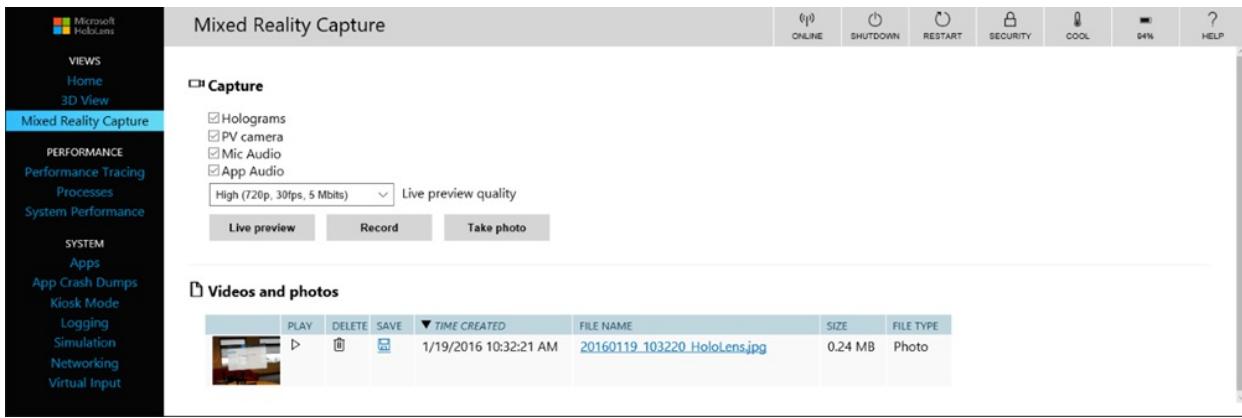


3D View page in Windows Device Portal on Microsoft HoloLens

Use the 3D View page to see how HoloLens interprets your surroundings. Navigate the view by using the mouse:

- Rotate: left click + mouse;
- Pan: right click + mouse;
- Zoom: mouse scroll.
- **Tracking options**
 - Turn on continuous visual tracking by checking **Force visual tracking**.
 - **Pause** stops visual tracking.
- **View options:** Set options on the 3D view:
 - **Tracking:** Indicates whether visual tracking is active.
 - **Show floor:** Displays a checkered floor plane.
 - **Show frustum:** Displays the view frustum.
 - **Show stabilization plane:** Displays the plane that HoloLens uses for stabilizing motion.
 - **Show mesh:** Displays the spatial mapping mesh that represents your surroundings.
 - **Show spatial anchors:** Displays spatial anchors for the active app. You must click the **Update** button to get and refresh the anchors.
 - **Show details:** Displays hand positions, head rotation quaternions, and the device origin vector as they change in real time.
 - **Full screen button:** Shows the 3D View in full screen mode. Press ESC to exit full screen view.
- **Surface reconstruction:** Click or tap **Update** to display the latest spatial mapping mesh from the device. A full pass may take some time to complete, up to a few seconds. The mesh does not update automatically in the 3D view, and you must manually click **Update** to get the latest mesh from the device. Click **Save** to save the current spatial mapping mesh as an obj file on your PC.
- **Spatial anchors:** Click **Update** to display or update the spatial anchors for the active app.

Mixed Reality Capture



Mixed Reality Capture page in Windows Device Portal on Microsoft HoloLens

Use the [Mixed Reality Capture](#) page to save media streams from the HoloLens.

- **Settings:** Control the media streams that are captured by checking the following settings:
 - **Holograms:** Captures the holographic content in the video stream. Holograms are rendered in mono, not stereo.
 - **PV camera:** Captures the video stream from the photo/video camera.
 - **Mic Audio:** Captures audio from the microphone array.
 - **App Audio:** Captures audio from the currently running app.
 - **Live preview quality:** Select the screen resolution, frame rate, and streaming rate for the live preview.
- Click or tap the **Live preview** button to show the capture stream. **Stop live preview** stops the capture stream.
- Click or tap **Record** to start recording the mixed-reality stream, using the specified settings. **Stop recording** ends the recording and saves it.
- Click or tap **Take photo** to take a still image from the capture stream.
- **Videos and photos:** Shows a list of video and photo captures taken on the device.

Note that HoloLens apps will not be able to capture an MRC photo or video while you are recording or streaming a live preview from the Device Portal.

Performance Tracing

Performance Tracing page in Windows Device Portal on Microsoft HoloLens

Capture [Windows Performance Recorder](#) (WPR) traces from your HoloLens.

- **Available profiles:** Select the WPR profile from the dropdown, and click or tap **Start** to start tracing.
- **Custom profiles:** Click or tap **Browse** to choose a WPR profile from your PC. Click or tap **Upload and start** to start tracing.

To stop the trace click on the stop link. Stay on this page until the trace file has completed downloading.

Captured ETL files can be opened for analysis in [Windows Performance Analyzer](#).

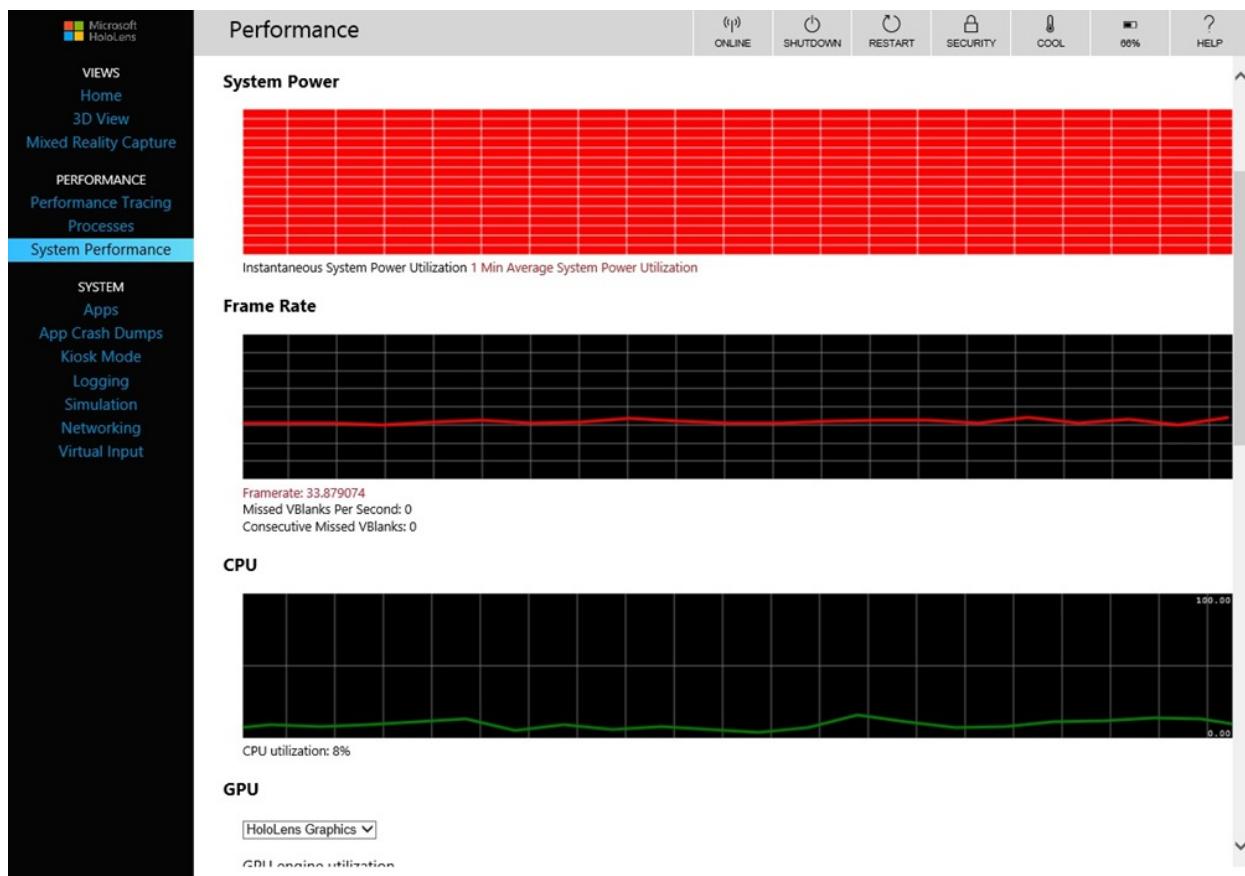
Processes

Running Processes							ONLINE	SHUTDOWN	RESTART	SECURITY	COOL	A/C	? HELP
▲ PID	NAME	USER NAME	SESSION ID	CPU	PRIVATE WORKIN...	WORKING SET	COMMIT SIZE						
0	System Idle Process	NT AUTHORITY\SYSTEM	0	97.25%	8.0 KB	8.0 KB	N/A						
4	System	NT AUTHORITY\SYSTEM	0	0.00%	8.0 KB	60.0 KB	N/A						
788	smss.exe	NT AUTHORITY\SYSTEM	0	0.00%	136.0 KB	920.0 KB	N/A						
872	svchost.exe	NT AUTHORITY\SYSTEM	0	0.00%	3.0 MB	14.1 MB	8.7 MB						
896	crss.exe	NT AUTHORITY\SYSTEM	0	0.00%	296.0 KB	1.9 MB	N/A						
972	wininit.exe	NT AUTHORITY\SYSTEM	0	0.00%	416.0 KB	3.3 MB	N/A						
1008	svchost.exe	NT AUTHORITY\SYSTEM	0	0.00%	1.1 MB	7.3 MB	1.8 MB						
1012	services.exe	NT AUTHORITY\SYSTEM	0	0.00%	1.3 MB	4.3 MB	N/A						
1020	lsass.exe	NT AUTHORITY\SYSTEM	0	0.00%	3.4 MB	12.7 MB	4.1 MB						
1128	svchost.exe	NT AUTHORITY\SYSTEM	0	0.00%	2.3 MB	12.0 MB	3.8 MB						
1148	dwm.exe	Window Manager\DWDM-0	0	1.95%	105.0 MB	140.1 MB	282.5 MB						
1196	svchost.exe	NT AUTHORITY\NETWO...	0	0.00%	1.8 MB	6.0 MB	2.4 MB						
1312	svchost.exe	NT AUTHORITY\SYSTEM	0	0.00%	3.4 MB	17.4 MB	4.8 MB						
1320	svchost.exe	NT AUTHORITY\LOCAL S...	0	0.00%	4.0 MB	12.9 MB	5.1 MB						
1336	svchost.exe	NT AUTHORITY\LOCAL S...	0	0.00%	4.0 MB	17.4 MB	5.8 MB						
1456	svchost.exe	NT AUTHORITY\LOCAL S...	0	0.00%	688.0 KB	4.5 MB	1.1 MB						
1488	svchost.exe	NT AUTHORITY\SYSTEM	0	0.00%	752.0 KB	4.9 MB	1.2 MB						
1496	svchost.exe	NT AUTHORITY\SYSTEM	0	0.00%	3.3 MB	14.2 MB	4.3 MB						
1508	svchost.exe	NT AUTHORITY\LOCAL S...	0	0.00%	2.2 MB	10.1 MB	3.5 MB						
1792	svchost.exe	NT AUTHORITY\SYSTEM	0	0.00%	4.5 MB	17.4 MB	6.2 MB						
1808	svchost.exe	NT AUTHORITY\NETWO...	0	0.00%	2.8 MB	9.8 MB	3.8 MB						
2076	MixedRealityCapture.exe	NT AUTHORITY\SYSTEM	0	0.00%	772.0 KB	4.8 MB	2.5 MB						
2084	XdeSvc.exe	NT AUTHORITY\SYSTEM	0	0.00%	1.1 MB	5.4 MB	2.0 MB						
2104	WebManagement.exe	NT AUTHORITY\SYSTEM	0	0.00%	5.2 MB	16.1 MB	7.7 MB						
2112	Spectrum.exe	NT AUTHORITY\LOCAL S...	0	0.38%	3.1 MB	40.9 MB	134.2 MB						
2232	msvmon.exe	MINWINPC\DefaultAcco...	0	0.00%	568.0 KB	4.3 MB	1.1 MB						
2304	sihost.exe	MINWINPC\DefaultAcco...	0	0.00%	2.9 MB	16.4 MB	4.4 MB						

Processes page in Windows Device Portal on Microsoft HoloLens

Shows details about currently running processes. This includes both apps and system processes.

System Performance



System Performance page in Windows Device Portal on Microsoft HoloLens

Shows real-time graphs of system diagnostic info, like power usage, frame rate, and CPU load.

These are the available metrics:

- **SoC power:** Instantaneous system-on-chip power utilization, averaged over one minute
- **System power:** Instantaneous system power utilization, averaged over one minute
- **Frame rate:** Frames per second, missed VBlanks per second, and consecutive missed VBlanks

- **GPU:** GPU engine utilization, percent of total available
- **CPU:** percent of total available
- **I/O:** Reads and writes
- **Network:** Received and sent
- **Memory:** Total, in use, committed, paged, and non-paged

Apps

Apps page in Windows Device Portal on Microsoft HoloLens

Manages the apps that are installed on the HoloLens.

- **Installed apps:** Remove and start apps.
- **Running apps:** Lists apps that are running currently.
- **Install app:** Select app packages for installation from a folder on your computer/network.
- **Dependency:** Add dependencies for the app you are going to install.
- **Deploy:** Deploy the selected app + dependencies to the HoloLens.

App Crash Dumps

App Crash Dumps page in Windows Device Portal on Microsoft HoloLens

This page allows you to collect crash dumps for your side-loaded apps. Check the **Crash Dumps Enabled** checkbox for each app for which you want to collect crash dumps. Return to this page to collect crash dumps. Dump files can be [opened in Visual Studio for debugging](#).

File Explorer

File Explorer page in Windows Device Portal on Microsoft HoloLens

Use the file explorer to browse, upload, and download files. You can work with files in the Documents folder, Pictures folder, and in the local storage folders for apps that you deployed from Visual Studio or the Device Portal.

Kiosk Mode

NOTE

Kiosk mode is only available with the Microsoft HoloLens Commercial Suite.

Please check the [Set up HoloLens in kiosk mode](#) article in Windows IT Pro Center for up-to-date instructions on enabling kiosk mode via Windows Device Portal.

Logging

Logging page in Windows Device Portal on Microsoft HoloLens

Manages realtime Event Tracing for Windows (ETW) on the HoloLens.

Check **Hide providers** to show the **Events** list only.

- **Registered providers:** Select the ETW provider and the tracing level. Tracing level is one of these values:

1. Abnormal exit or termination
2. Severe errors
3. Warnings
4. Non-error warnings

Click or tap **Enable** to start tracing. The provider is added to the **Enabled Providers** dropdown.

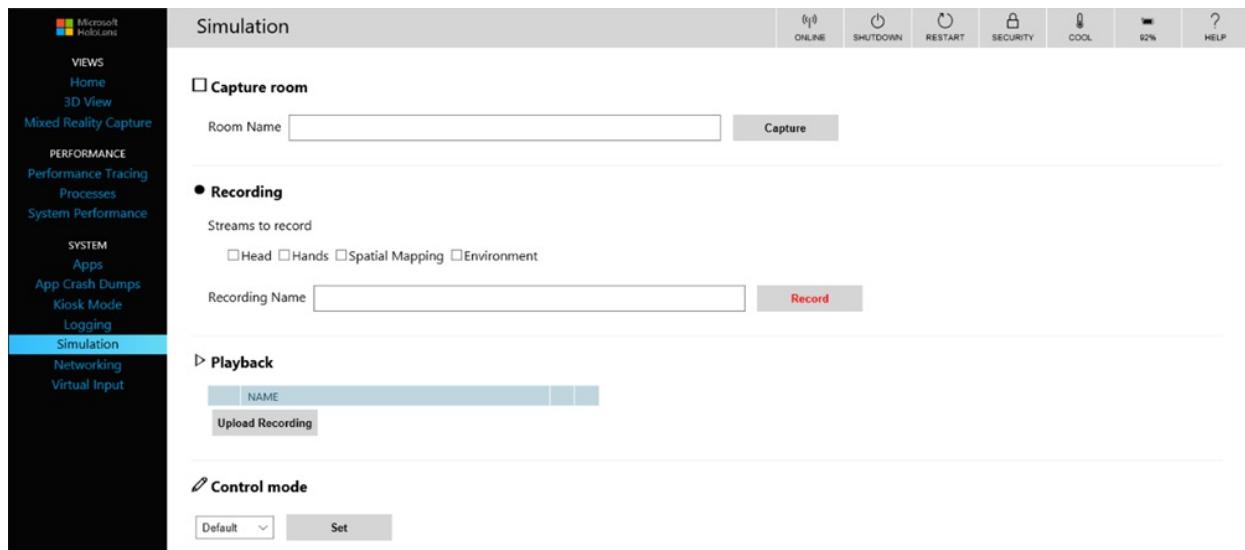
- **Custom providers:** Select a custom ETW provider and the tracing level. Identify the provider by its GUID.

Don't include brackets in the GUID.

- **Enabled providers:** Lists the enabled providers. Select a provider from the dropdown and click or tap **Disable** to stop tracing. Click or tap **Stop all** to suspend all tracing.
- **Providers history:** Shows the ETW providers that were enabled during the current session. Click or tap **Enable** to activate a provider that was disabled. Click or tap **Clear** to clear the history.
- **Events:** Lists ETW events from the selected providers in table format. This table is updated in real time. Beneath the table, click on the **Clear** button to delete all ETW events from the table. This does not disable any providers. You can click **Save to file** to export the currently collected ETW events to a CSV file locally.
- **Filters:** Allow you to filter the ETW events collected by ID, Keyword, Level, Provider Name, Task Name, or Text. You can combine several criteria together:
 1. For criteria applying to the same property - events can satisfy any one of these criteria are shown.
 2. For criteria applying to different property - events must satisfy all of the criteria

For example, you can specify the criteria (*Task Name contains 'Foo' or 'Bar'*) AND (*Text contains 'error' or 'warning'*)

Simulation



Simulation page in Windows Device Portal on Microsoft HoloLens

Allows you to record and play back input data for testing.

- **Capture room:** Used to download a simulated room file that contains the spatial mapping mesh for the user's surroundings. Name the room and then click **Capture** to save the data as a .xef file on your PC. This room file can be loaded into the HoloLens emulator.
- **Recording:** Check the streams to record, name the recording, and click or tap **Record** to start recording. Perform actions with your HoloLens and then click **Stop** to save the data as a .xef file on your PC. This file can be loaded on the HoloLens emulator or device.
- **Playback:** Click or tap **Upload recording** to select a xef file from your PC and send the data to the HoloLens.
- **Control mode:** Select **Default** or **Simulation** from the dropdown, and click or tap the **Set** button to select the mode on the HoloLens. Choosing "Simulation" disables the real sensors on your HoloLens and uses uploaded simulated data instead. If you switch to "Simulation", your HoloLens will not respond to the real user until you switch back to "Default".

Networking

WiFi adapters

Broadcom 802.11ac Wireless PCIE Full Dongle Adapter	▼
---	---

Available networks

SSID	PROFILE	INFRA	SIGNAL	SECURITY	ENCRYPTION	CHANNEL
WirelessNetwork-24		✓	■■■	WPA2_PSK	AES	
WirelessNetwork-5	WirelessNetwork-5	✓	■■■	WPA2_PSK	AES	52

IP configuration

Description:	Broadcom 802.11ac Wireless PCIE Full Dongle Adapter
Type:	IEEE 802.11
Physical address:	b4-ae-2b-be-c9-ea
IPv4 address:	10.10.0.10
Subnet mask:	255.255.254.0
Gateway address:	10.10.0.1
DHCP server:	10.10.0.1

Networking page in Windows Device Portal on Microsoft HoloLens

Manages Wi-Fi connections on the HoloLens.

- WiFi adapters:** Select a Wi-Fi adapter and profile by using the dropdown controls. Click or tap **Connect** to use the selected adapter.
- Available networks:** Lists the Wi-Fi networks that the HoloLens can connect to. Click or tap **Refresh** to update the list.
- IP configuration:** Shows the IP address and other details of the network connection.

Virtual Input

Virtual Input

Virtual keyboard

Select this control to send keyboard input to the device

Input text

Send

Virtual Input page in Windows Device Portal on Microsoft HoloLens

Sends keyboard input from the remote machine to the HoloLens.

Click or tap the region under **Virtual keyboard** to enable sending keystrokes to the HoloLens. Type in the **Input text** textbox and click or tap **Send** to send the keystrokes to the active app.

Device Portal REST API's

Everything in the device portal is built on top of [REST API's](#) that you can optionally use to access the data and

control your device programmatically.

Device portal API reference

11/6/2018 • 6 minutes to read • [Edit Online](#)

Everything in the [Windows Device Portal](#) is built on top of REST API's that you can use to access the data and control your device programmatically.

App deployment

/api/app/packagemanager/package (DELETE)

Uninstalls an app

Parameters

- package : File name of the package to be uninstalled.

/api/app/packagemanager/package (POST)

Installs an App

Parameters

- package : File name of the package to be installed.

Payload

- multi-part conforming http body

/api/app/packagemanager/packages (GET)

Retrieves the list of installed apps on the system, with details

Return data

- List of installed packages with details

/api/app/packagemanager/state (GET)

Gets the status of in progress app installation

Dump collection

/api/debug/dump/usermode/crashcontrol (DELETE)

Disables crash dump collection for a sideloaded app

Parameters

- packageFullscreen : package name

/api/debug/dump/usermode/crashcontrol (GET)

Gets settings for sideloaded apps crash dump collection

Parameters

- packageFullscreen : package name

/api/debug/dump/usermode/crashcontrol (POST)

Enables and sets dump control settings for a sideloaded app

Parameters

- packageFullscreen : package name

/api/debug/dump/usermode/crashdump (DELETE)

Deletes a crash dump for a sideloaded app

Parameters

- packageFullscreen : package name
- fileName : dump file name

/api/debug/dump/usermode/crashdump (GET)

Retrieves a crash dump for a sideloaded app

Parameters

- packageFullscreen : package name
- fileName : dump file name

Return data

- Dump file. Inspect with WinDbg or Visual Studio

/api/debug/dump/usermode/dumps (GET)

Returns list of all crash dumps for sideloaded apps

Return data

- List of crash dumps per side loaded app

ETW

/api/etw/providers (GET)

Enumerates registered providers

Return data

- List of providers, friendly name and GUID

/api/etw/session/realtime (GET/WebSocket)

Creates a realtime ETW session; managed over a websocket.

Return data

- ETW events from the enabled providers

Holographic OS

/api/holographic/os/etw/customproviders (GET)

Returns a list of HoloLens specific ETW providers that are not registered with the system

/api/holographic/os/services (GET)

Returns the states of all services running.

/api/holographic/os/settings/ipd (GET)

Gets the stored IPD (Interpupillary distance) in millimeters

/api/holographic/os/settings/ipd (POST)

Sets the IPD

Parameters

- ipd : New IPD value to be set in millimeters

/api/holographic/os/webmanagement/settings/https (GET)

Get HTTPS requirements for the Device Portal

/api/holographic/os/webmanagement/settings/https (POST)

Sets HTTPS requirements for the Device Portal

Parameters

- required : yes, no or default

Holographic Perception

/api/holographic/perception/client (GET/WebSocket)

Accepts websocket upgrades and runs a perception client that sends updates at 30 fps.

Parameters

- clientmode: "active" forces visual tracking mode when it can't be established passively

Holographic Thermal

/api/holographic/thermal/stage (GET)

Get the thermal stage of the device (0 normal, 1 warm, 2 critical)

Perception Simulation Control

/api/holographic/simulation/control mode (GET)

Get the simulation mode

/api/holographic/simulation/control mode (POST)

Set the simulation mode

Parameters

- mode : simulation mode: default, simulation, remote, legacy

/api/holographic/simulation/control/stream (DELETE)

Delete a control stream.

/api/holographic/simulation/control/stream (GET/WebSocket)

Open a web socket connection for a control stream.

/api/holographic/simulation/control/stream (POST)

Create a control stream (priority is required) or post data to a created stream (streamId required). Posted data is expected to be of type 'application/octet-stream'.

Perception Simulation Playback

/api/holographic/simulation/playback/file (DELETE)

Delete a recording.

Parameters

- recording : Name of recording to delete.

/api/holographic/simulation/playback/file (POST)

Upload a recording.

/api/holographic/simulation/playback/files (GET)

Get all recordings.

/api/holographic/simulation/playback/session (GET)

Get the current playback state of a recording.

Parameters

- recording : Name of recording.

/api/holographic/simulation/playback/session/file (DELETE)

Unload a recording.

Parameters

- recording : Name of recording to unload.

/api/holographic/simulation/playback/session/file (POST)

Load a recording.

Parameters

- recording : Name of recording to load.

/api/holographic/simulation/playback/session/files (GET)

Get all loaded recordings.

/api/holographic/simulation/playback/session/pause (POST)

Pause a recording.

Parameters

- recording : Name of recording.

/api/holographic/simulation/playback/session/play (POST)

Play a recording.

Parameters

- recording : Name of recording.

/api/holographic/simulation/playback/session/stop (POST)

Stop a recording.

Parameters

- recording : Name of recording.

/api/holographic/simulation/playback/session/types (GET)

Get the types of data in a loaded recording.

Parameters

- recording : Name of recording.

Perception Simulation Recording

/api/holographic/simulation/recording/start (POST)

Start a recording. Only a single recording can be active at once. One of head, hands, spatialMapping or environment must be set.

Parameters

- head : Set to 1 to record head data.
- hands : Set to 1 to record hand data.
- spatialMapping : Set to 1 to record spatial mapping.
- environment : Set to 1 to record environment data.
- name : Name of the recording.
- singleSpatialMappingFrame : Set to 1 to record only a single spatial mapping frame.

/api/holographic/simulation/recording/status (GET)

Get recording state.

/api/holographic/simulation/recording/stop (GET)

Stop the current recording. Recording will be returned as a file.

Mixed Reality Capture

/api/holographic/mrc/file (DELETE)

Deletes a mixed reality recording from the device.

Parameters

- filename : Name, hex64 encoded, of the file to delete

/api/holographic/mrc/settings (GET)

Gets the default mixed reality capture settings

/api/holographic/mrc/file (GET)

Downloads a mixed reality file from the device. Use op=stream query parameter for streaming.

Parameters

- filename : Name, hex64 encoded, of the video file to get
- op : stream

/api/holographic/mrc/thumbnail (GET)

Gets the thumbnail image for the specified file.

Parameters

- filename: Name, hex64 encoded, of the file for which the thumbnail is being requested

/api/holographic/mrc/status (GET)

Gets the status of the mixed reality recorded (running, stopped)

/api/holographic/mrc/files (GET)

Returns the list of mixed reality files stored on the device

/api/holographic/mrc/settings (POST)

Sets the default mixed reality capture settings

/api/holographic/mrc/video/control/start (POST)

Starts a mixed reality recording

Parameters

- holo : capture holograms: true or false
- pv : capture PV camera: true or false
- mic : capture microphone: true or false
- loopback : capture app audio: true or false

/api/holographic/mrc/video/control/stop (POST)

Stops the current mixed reality recording

/api/holographic/mrc/photo (POST)

Takes a mixed reality photo and creates a file on the device

Parameters

- holo : capture holograms: true or false
- pv : capture PV camera: true or false

Mixed Reality Streaming

/api/holographic/stream/live.mp4 (GET)

Initiates a chunked download of a fragmented mp4

Parameters

- holo : capture holograms: true or false
- pv : capture PV camera: true or false
- mic : capture microphone: true or false
- loopback : capture app audio: true or false

/api/holographic/stream/live_high.mp4 (GET)

Initiates a chunked download of a fragmented mp4

Parameters

- holo : capture holograms: true or false
- pv : capture PV camera: true or false
- mic : capture microphone: true or false
- loopback : capture app audio: true or false

/api/holographic/stream/live_low.mp4 (GET)

Initiates a chunked download of a fragmented mp4

Parameters

- holo : capture holograms: true or false
- pv : capture PV camera: true or false
- mic : capture microphone: true or false
- loopback : capture app audio: true or false

/api/holographic/stream/live_med.mp4 (GET)

Initiates a chunked download of a fragmented mp4

Parameters

- holo : capture holograms: true or false
- pv : capture PV camera: true or false
- mic : capture microphone: true or false
- loopback : capture app audio: true or false

Networking

/api/networking/ipconfig (GET)

Gets the current ip configuration

OS Information

/api/os/info (GET)

Gets operating system information

/api/os/machinename (GET)

Gets the machine name

/api/os/machinename (POST)

Sets the machine name

Parameters

- name : New machine name, hex64 encoded, to set to

Performance data

/api/resourcemanager/processes (GET)

Returns the list of running processes with details

Return data

- JSON with list of processes and details for each process

/api/resourcemanager/systemperf (GET)

Returns system perf statistics (I/O read/write, memory stats etc.

Return data

- JSON with system information: CPU, GPU, Memory, Network, IO

Power

/api/power/battery (GET)

Gets the current battery state

/api/power/state (GET)

Checks if the system is in a low power state

Remote Control

/api/control/restart (POST)

Restarts the target device

/api/control/shutdown (POST)

Shuts down the target device

Task Manager

/api/taskmanager/app (DELETE)

Stops a modern app

Parameters

- package : Full name of the app package, hex64 encoded
- forcestop : Force all processes to stop (=yes)

/api/taskmanager/app (POST)

Starts a modern app

Parameters

- appid : PRAID of app to start, hex64 encoded
- package : Full name of the app package, hex64 encoded

WiFi Management

/api/wifi/interfaces (GET)

Enumerates wireless network interfaces

Return data

- List of wireless interfaces with details (GUID, description etc.)

/api/wifi/network (DELETE)

Deletes a profile associated with a network on a specified interface

Parameters

- interface : network interface guid
- profile : profile name

/api/wifi/networks (GET)

Enumerates wireless networks on the specified network interface

Parameters

- interface : network interface guid

Return data

- List of wireless networks found on the network interface with details

/api/wifi/network (POST)

Connects or disconnects to a network on the specified interface

Parameters

- interface : network interface guid
- ssid : ssid, hex64 encoded, to connect to
- op : connect or disconnect
- createprofile : yes or no
- key : shared key, hex64 encoded

Windows Performance Recorder

/api/wpr/customtrace (POST)

Uploads a WPR profile and starts tracing using the uploaded profile.

Payload

- multi-part conforming http body

Return data

- Returns the WPR session status.

/api/wpr/status (GET)

Retrieves the status of the WPR session

Return data

- WPR session status.

/api/wpr/trace (GET)

Stops a WPR (performance) tracing session

Return data

- Returns the trace ETL file

/api/wpr/trace (POST)

Starts a WPR (performance) tracing sessions

Parameters

- profile : Profile name. Available profiles are stored in perfprofiles/profiles.json

Return data

- On start, returns the WPR session status.

See also

- [Using the Windows Device Portal](#)
- [Device Portal core API reference \(UWP\)](#)

Holographic Remoting Player

11/6/2018 • 2 minutes to read • [Edit Online](#)

The Holographic Remoting Player is a companion app that connects to PC apps and games that support Holographic Remoting. Holographic Remoting streams holographic content from a PC to your Microsoft HoloLens in real-time, using a Wi-Fi connection.

The Holographic Remoting Player can only be used with PC apps that are specifically designed to support Holographic Remoting.

Connecting to the Holographic Remoting Player

Follow your app's instructions to connect to the Holographic Remoting Player. You will need to enter the IP address of your HoloLens device, which you can see on the Remoting Player's main screen as follows:



Whenever you see the main screen, you will know that you do not have an app connected.

Note that the holographic remoting connection is **not encrypted**. You should always use Holographic Remoting over a secure Wi-Fi connection that you trust.

Quality and Performance

The quality and performance of your experience will vary based on three factors:

- **The holographic experience you're running** - Apps that render high-resolution or highly-detailed content may require a faster PC or faster wireless connection.
- **Your PC's hardware** - Your PC needs to be able to run and encode your holographic experience at 60 frames

per second. For a graphics card, we generally recommend a GeForce GTX 970 or AMD Radeon R9 290 or better. Again, your particular experience may require a higher or lower-end card.

- **Your Wi-Fi connection** - Your holographic experience is streamed over Wi-Fi. Use a fast network with low congestion to maximize quality. Using a PC that is connected over an Ethernet cable, rather than Wi-Fi, may also improve quality.

Diagnostics

To measure the quality of your connection, say "**enable diagnostics**" while on the main screen of the Holographic Remoting Player. When diagnostics are enabled, the app will show you:

- **FPS** - The average number of rendered frames the remoting player is receiving and rendering per second. The ideal is 60 FPS.
- **Latency** - The average amount of time it takes for a frame to go from your PC to the HoloLens. The lower the better. This is largely dependent on your Wi-Fi network.

While on the main screen, you can say "**disable diagnostics**" to turn off diagnostics.

PC System Requirements

- Your PC **must** be running the Windows 10 Anniversary Update.
- We recommend a GeForce GTX 970 or AMD Radeon R9 290 or better graphics card.
- We recommend you connect your PC to your network via ethernet to reduce the number of Wireless hops.

See Also

- [Holographic remoting software license terms](#)
- [Microsoft Privacy Statement](#)

QR code tracking

11/6/2018 • 8 minutes to read • [Edit Online](#)

QR code tracking is implemented in the Windows Mixed Reality driver for immersive (VR) headsets. By enabling the QR code tracker in the headset driver, the headset scans for QR codes and they are reported to interested apps. This feature is only available as of the [Windows 10 October 2018 Update \(also known as RS5\)](#).

Device support

FEATURE	HOLOLENS	IMMERSIVE HEADSETS
QR code tracking		✓ <input type="checkbox"/>

Enabling and disabling QR code tracking for your headset

Whether you're developing a mixed reality app that will leverage QR code tracking, or you're a customer of one of these apps, you'll need to manually turn on QR code tracking in your headset's driver.

In order to **turn on QR code tracking** for your immersive (VR) headset:

1. Close the Mixed Reality Portal app on your PC.
2. Unplug the headset from your PC.
3. Run the following script in the Command Prompt:

```
reg add "HKLM\SOFTWARE\Microsoft\HoloLensSensors" /v EnableQRTrackerDefault /t REG_DWORD /d 1 /F
```

4. Reconnect your headset to your PC.

In order to **turn off QR code tracking** for your immersive (VR) headset:

1. Close the Mixed Reality Portal app on your PC.
2. Unplug the headset from your PC.
3. Run the following script in the Command Prompt:

```
reg add "HKLM\SOFTWARE\Microsoft\HoloLensSensors" /v EnableQRTrackerDefault /t REG_DWORD /d 0 /F
```

4. Reconnect your headset to your PC. This will make any discovered QR codes "non-locatable."

QRTracking API

The QRTracking plugin exposes the APIs for QR code tracking. To use the plugin, you will need to use the following types from the *QRCodeTrackerPlugin* namespace.

```
// QRTracker plugin namespace
namespace QRCodeTrackerPlugin
{
    // Encapsulates information about a labeled QR code element.
    public class QRCode
    {
        // Unique id that identifies this QR code for this session.
        public Guid Id { get; private set; }

        // Version of this QR code.
        public Int32 Version { get; private set; }

        // PhysicalSizeMeters of this QR code.
    }
}
```

```

        public float PhysicalSizeMeters { get; private set; }

        // QR code Data.
        public string Code { get; private set; }

        // QR code DataStream this is the error corrected data stream
        public Byte[] CodeStream { get; private set; }

        // QR code last detected QPC ticks.
        public Int64 LastDetectedQPCTicks { get; private set; }
    };

    // The type of a QR Code added event.
    public class QRCodeAddedEventArgs
    {
        // Gets the QR Code that was added
        public QRCode Code { get; private set; }
    };

    // The type of a QR Code removed event.
    public class QRCodeRemovedEventArgs
    {
        // Gets the QR Code that was removed.
        public QRCode Code { get; private set; }
    };

    // The type of a QR Code updated event.
    public class QRCodeUpdatedEventArgs
    {
        // Gets the QR Code that was updated.
        public QRCode Code { get; private set; }
    };

    // A callback for handling the QR Code added event.
    public delegate void QRCodeAddedHandler(QRCodeAddedEventArgs args);

    // A callback for handling the QR Code removed event.
    public delegate void QRCodeRemovedHandler(QRCodeRemovedEventArgs args);

    // A callback for handling the QR Code updated event.
    public delegate void QRCodeUpdatedHandler(QRCodeUpdatedEventArgs args);

    // Enumerates the possible results of a start of QRTracker.
    public enum QRTrackerStartResult
    {
        // The start has succeeded.
        Success = 0,

        // The currently no device is connected.
        DeviceNotConnected = 1,

        // The QRTracking Feature is not supported by the current HMD driver
        // systems
        FeatureNotSupported = 2,

        // The access is denied. Administrator needs to enable QR tracker feature.
        AccessDenied = 3,
    };
}

// QRTracker
public class QRTracker
{
    // Constructs a new QRTracker.
    public QRTracker(){}
}

// Gets the QRTracker.
public static QRTracker Get()
{
    return new QRTracker();
}

```

```

        }

        // Start the QRTracker. Returns QRTrackerStartResult.
        public QRTrackerStartResult Start()
        {
            throw new NotImplementedException();
        }

        // Stops tracking QR codes.
        public void Stop() {}

        // Not Implemented
        Windows.Foundation.Collections.IVector<QRCode> GetList() { throw new NotImplementedException(); }

        // Event representing the addition of a QR Code.
        public event QRCodeAddedHandler Added = delegate { };

        // Event representing the removal of a QR Code.
        public event QRCodeRemovedHandler Removed = delegate { };

        // Event representing the update of a QR Code.
        public event QRCodeUpdatedHandler Updated = delegate { };
    };
}

```

Implementing QR code tracking in Unity

Sample Unity scenes in MRTK (Mixed Reality Toolkit)

You can find an example for how to use the QR Tracking API in the Mixed Reality Toolkit [GitHub site](#).

MRTK has implemented the needed scripts to simplify the QR tracking usage. All necessary assets to develop QR tracking apps are in the "QRTracker" folder. There are two scenes: the first is a sample to simply show details of the QR codes as they are detected, and the second demonstrates how to use the coordinate system attached to the QR code to display holograms. There is a prefab "QRScanner" which adds all the necessary scripts to the scenes to use QR Codes. The script QRCodeManager is a singleton class that implements the QRCode API you can add it to your scene. The script "AttachToQRCode" is used to attach holograms to the QR Code coordinate systems, this script can be added to any of your holograms. The "SpatialGraphCoordinateSystem" shows how to use the QRCode coordinate system. These scripts can be used as is in your project scenes or you can write your own directly using the plugin as described above.

Implementing QR code tracking in Unity without MRTK

You can also use the QR Tracking API in Unity without taking a dependency on MRTK. In order to use the API, you will need to prepare your project with the following instruction:

1. Create a new folder in the assets folder of your unity project with the name: "Plugins".
2. Copy all the required files from [this folder](#) into the local "Plugins" folder you just created.
3. You can use the QR tracking scripts in the [MRTK scripts folder](#) or write your own.

Implementing QR code tracking in DirectX

To use the QRTrackingPlugin in Visual Studio, you will need to add reference of the QRTrackingPlugin to the .winmd. You can find the [required files for supported platforms here](#).

```
// MyClass.h
public ref class MyClass
{
    private:
        QRCodeScannerPlugin::QRScanner^ m_scanner;
        // Handlers
        void OnAddedQRCode(QRCodeScannerPlugin::QRCodeEventArgs ^args);
        void OnUpdatedQRCode(QRCodeScannerPlugin::QRCodeEventArgs ^args);
        void OnRemovedQRCode(QRCodeScannerPlugin::QRCodeEventArgs ^args);
    ..
};


```

```
// MyClass.cpp
MyClass::MyClass()
{
    // Create the scanner and register the callbacks
    m_scanner = ref new QRCodeScannerPlugin::QRScanner();
    m_scanner->Added += ref new QRCodeScannerPlugin::QRCodeAddedHandler(this, &OnAddedQRCode);
    m_scanner->Updated += ref new QRCodeScannerPlugin::QRCodeUpdatedHandler(this, &OnUpdatedQRCode);
    m_scanner->Removed += ref new QRCodeScannerPlugin::QRCodeRemovedHandler(this, &OnRemovedQRCode);

    // Start the scanner
    if (m_scanner->Start() != QRCodeScannerPlugin::QRScannerStartResult::Success)
    {
        // Handle the failure
        // It can fail for multiple reasons and can be handled properly
    }
}

void MyClass::OnAddedQRCode(QRCodeScannerPlugin::QRCodeEventArgs ^args)
{
    // use args->Code add to own list
}

void MyClass::OnUpdatedQRCode(QRCodeScannerPlugin::QRCodeEventArgs ^args)
{
    // use args->Code update the existing one with the new one in own list
}

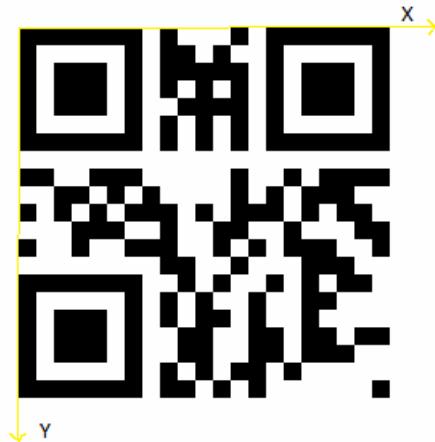
void MyClass::OnRemovedQRCode(QRCodeScannerPlugin::QRCodeEventArgs ^args)
{
    // use args->Code remove from own list.
}
```

Getting a coordinate system

We define a right-hand coordinate system aligned with the QR code at the top left corner of the fast detection square in the top left. The coordinate system is shown below. The Z-axis is pointing into the paper (not shown), but in Unity the z-axis is out of the paper and left-handed.

A SpatialCoordinateSystem is defined that is aligned as shown. This coordinate system can be obtained from the platform using the API

Windows::Perception::Spatial::Preview::SpatialGraphInteropPreview::CreateCoordinateSystemForNode.



From QRCode[^] Code object, the following code shows how to create a rectangle and put it in QR coordinate system:

```
// Creates a 2D rectangle in the x-y plane, with the specified properties.
std::vector<float3> SpatialStageManager::CreateRectangle(float width, float height)
{
    std::vector<float3> vertices(4);

    vertices[0] = { 0, 0, 0 };
    vertices[1] = { width, 0, 0 };
    vertices[2] = { width, height, 0 };
    vertices[3] = { 0, height, 0 };

    return vertices;
}
```

You can use the physical size to create the QR rectangle:

```
std::vector<float3> qrVertices = CreateRectangle(Code->PhysicalSizeMeters, Code->PhysicalSizeMeters);
```

The coordinate system can be used to draw the QR code or attach holograms to the location:

```
Windows::Perception::Spatial::SpatialCoordinateSystem^ qrCoordinateSystem =
Windows::Perception::Spatial::Preview::SpatialGraphInteropPreview::CreateCoordinateSystemForNode(Code->Id);
```

Altogether, your QRCodeTrackerPlugin::QRCodeAddedHandler may look something like this:

```

void MyClass::OnAddedQRCode(QRCodesTrackerPlugin::QRCodeAddedEventArgs ^args)
{
    std::vector<float3> qrVertices = CreateRectangle(args->Code->PhysicalSizeMeters, args->Code-
>PhysicalSizeMeters);
    std::vector<unsigned short> qrCodeIndices = TriangulatePoints(qrVertices);
    XMFLOAT3 qrAreaColor = XMFLOAT3(DirectX::Colors::Aqua);

    Windows::Perception::Spatial::SpatialCoordinateSystem^ qrCoordinateSystem =
Windows::Perception::Spatial::Preview::SpatialGraphInteropPreview::CreateCoordinateSystemForNode(args->Code-
>Id);
    std::shared_ptr<SceneObject> m_qrShape =
    std::make_shared<SceneObject>(
        m_deviceResources,
        reinterpret_cast<std::vector<XMFLOAT3>&>(qrVertices),
        qrCodeIndices,
        qrAreaColor,
        qrCoordinateSystem);

    m_sceneController->AddSceneObject(m_qrShape);
}

```

Troubleshooting and FAQ

General troubleshooting

- Is your PC running the Windows 10 October 2018 Update?
- Have you set the reg key? Restarted the device afterwards?
- Is the QR code version a supported version? Current API supports up to QR Code Version 20. We recommend using version 5 for general usage.
- Are you close enough to the QR code? The closer the camera is to the QR code, the higher the QR code version the API can support.

How close do I need to be to the QR code to detect it?

This will depend on the size of the QR code, and also what version it is. For a version 1 QR code varying from 5 cm sides to 25 cm sides, the minimum detection distance ranges from 0.25 meters to 0.5 meters. The farthest away these can be detected from goes from about 0.5 meters for the smaller QR code targets to 2 meters for the bigger. For QR codes bigger than that, you can estimate; the detection distance for size increases linearly. Our tracker does not work with QR codes with sides smaller than 5 cm.

Do QR codes with logos work?

QR codes with logos have not been tested and are currently unsupported.

How do I clear QR codes from my app so they don't persist?

- QR codes are only persisted in the boot session. Once you reboot (or restart the driver), they will be gone and detected as new objects next time.
- QR code history is saved at the system level in the driver session, but you can configure your app to ignore QR codes older than a specific timestamp if you want. Currently the API does support clearing QR code history, as multiple apps might be interested in the data.

Shared experiences in mixed reality

11/6/2018 • 8 minutes to read • [Edit Online](#)

Holograms don't need to stay private to just one user. Holographic apps may share [spatial anchors](#) from one HoloLens to another, enabling users to render a hologram at the same place in the real world across multiple devices.

Six questions to define shared scenarios

Before you begin designing for shared experiences, it's important to define the target scenarios. These scenarios help clarify what you're designing and establish a common vocabulary to help compare and contrast features required in your experience. Understanding the core problem, and the different avenues for solutions, is key to uncovering opportunities inherent in this new medium.

Through internal prototypes and explorations from our HoloLens partner agencies, we created six questions to help you define shared scenarios. These questions form a framework, not intended to be exhaustive, to help distill the important attributes of your scenarios.

1. How are they sharing?

A presentation might be led by a single virtual user, while multiple users can collaborate, or a teacher might provide guidance to virtual students working with virtual materials — the complexity of the experiences increases based on the level of agency a user has or can have in a scenario.



There are many ways to share, but we've found that most of them fall into three categories:

- **Presentation:** When the same content is being shown to several users. For example: A professor is giving out a lecture to several students using the same holographic material being presented to everyone. The professor however could have his/her own hints and notes that may not be visible to others.
- **Collaboration:** When people are working together to achieve some common goals. For example: The professor gave out a project to learn about performing a heart surgery. Students pair up and create a shared skills lab experience which allows medical students to collaborate on the heart model and learn.
- **Guidance:** When one person is helping someone to solve a problem in a more one-one style interaction. For example: The professor giving guidance to a student when he/she is performing the heart surgery skills lab in the shared experience.

2. What is the group size?

One-to-one sharing experiences can provide a strong baseline and ideally your proofs of concept can be created at this level. But be aware that sharing with large groups (beyond 6 people) can lead to difficulties from technical (data and networking) to social (the impact of being in a room with [several avatars](#)). Complexity increases exponentially as you go from **small** to **large groups**.

We have found that the needs of groups can fall into three size categories:

- 1:1
- Small < 7
- Large >= 7

Group size makes for an important question because it influences:

- Representations of people in holographic space
- Scale of objects
- Scale of environment

3. Where is everyone?

The strength of mixed reality comes into play when a shared experience can take place in the same location. We call that **co-located**. Conversely, when the group is distributed and at least one participant is not in the same physical space (as is often the case with VR) we call that a remote experience. Often it's the case that your group has **both** co-located and remote participants (e.g., two groups in conference rooms).



Following categories help convey where users are located:

- Co-located: All your users will be in the same physical space.
- Remote: All your users will be in separate physical spaces.
- Both: Your users will be a mix of co-located and remote spaces.

Some reasons why this question is crucial because it influences:

- How people communicate?
- For example: Whether they should have avatars?
- What objects they see. Are all objects shared?
- Whether we need to adapt to their environment?

4. When are they sharing?

We typically think of **synchronous** experiences when shared experiences come to mind: We're all doing it together. But include a single, virtual element that was added by someone else and you have an **asynchronous** scenario. Imagine a note, or voice memo, left in a virtual environment. How do you handle 100 virtual memos left on your design? What if they're from dozens of people with different levels of privacy?

Consider your experiences as one of these categories of time:

- **Synchronously:** Sharing the holographic experience at the same time. For example: Two students performing the skills lab at the same time.
- **Asynchronously:** Sharing the holographic experience at different times. For example: Two students performing the skills lab but working on separate sections at different times.
- **Both:** Your users will sometimes be sharing synchronously but other times asynchronously. For example: Professor grading the assignment performed by the students at a later time and leaving notes for student for the next day.

Some reasons why this question is important because it influences:

- Object and environment persistence. For example: Storing the states so they can be retrieved.
- User perspective. For example: Perhaps remembering what the user was looking at when leaving notes.

5. How similar are their physical environments?

The likelihood of two identical real-life environments, outside of co-located experiences, is slim unless those environments have been designed to be identical. You're more likely to have **similar** environments. For example, conference rooms are similar — they typically have a centrally located table surrounded by chairs. Living rooms, on the other hand, are usually **dissimilar** and can include any number of pieces of furniture in an infinite array of layouts.



Consider your sharing experiences fitting into one of these two categories:

- **Similar:** Environments that tend to have similar furniture, ambient light and sound, physical room size. For example: Professor is in lecture hall A and students are in lecture hall B. Lecture hall A might have fewer chairs than B but they both may have a physical desk to place holograms on.
- **Dissimilar:** Environments that are quite different in furniture settings, room sizes, light and sound considerations. For example: Professor is in a focus room whereas students are in a large lecture hall filled with students and teachers.

It's important to think about the environment as it will influence:

- How will people experience these objects. For example: If your experience works best on a table and the user has no table? Or on a flat floor surface but the user has a cluttered space.
- Scale of the objects. For example: Placing a 6 feet human model on a table could be challenging but a heart model would work great.

6. What devices are they using?

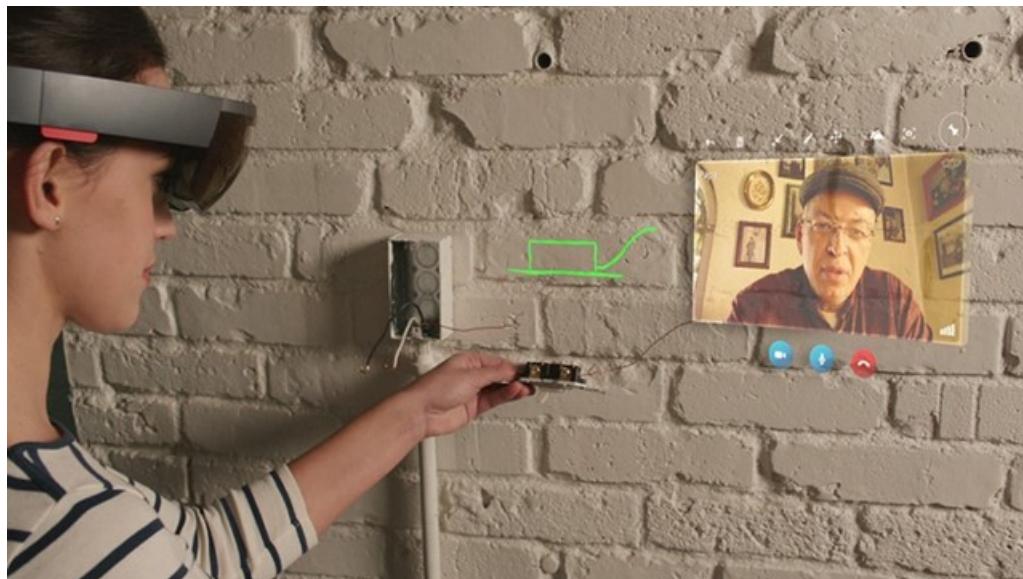
Today you're often likely to see shared experiences between two **immersive devices** (those devices might differ slightly in terms of buttons and relative capability, but not greatly) or two **holographic devices** given the solutions being targeted at these devices. But consider that **2D devices** (a mobile/desktop participant or observer) will be a necessary consideration, especially in situations of **mixed 2D and 3D devices**. Understanding the types of devices your participants will be using is important, not only because they come with different fidelity

and data constraints and opportunities, but because users have unique expectations for each platform.

Exploring the potential of shared experiences

Answers to the questions above can be combined to better understand your shared scenario, crystallizing the challenges as you expand the experience. For the team at Microsoft, this helped establish a road map for improving the experiences we use today, understanding the nuance of these complex problems and how to take advantage of shared experiences in mixed reality.

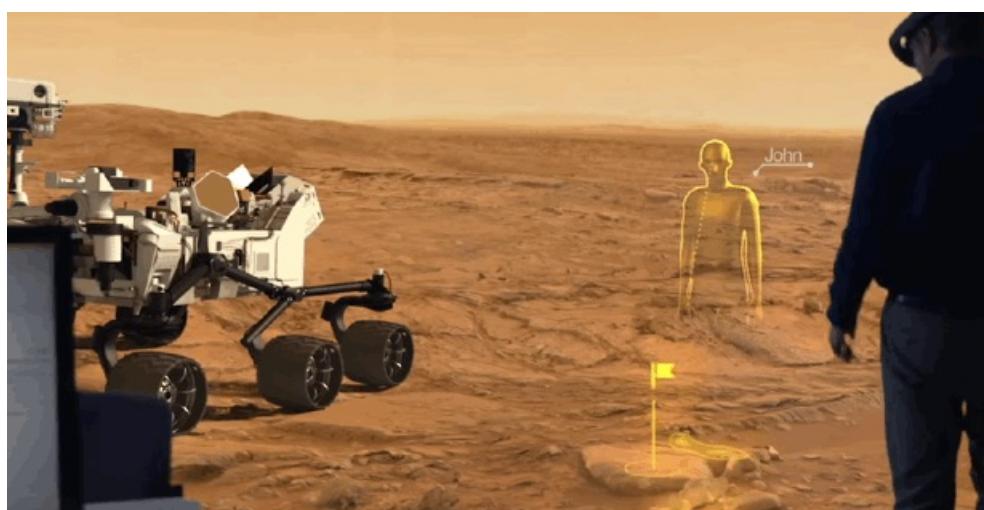
For example, consider one of Skype's scenarios from the HoloLens launch: a user worked through [how to fix a broken light switch](#) with help from a remotely-located expert.



An expert provides **1:1** guidance from his **2D**, desktop computer to a user of a **3D, mixed-reality** device. The **guidance is synchronous** and the physical environments are **dissimilar**.

An experience like this is a step-change from our current experience — applying the paradigm of video and voice to a new medium. But as we look to the future, we must better define the opportunity of our scenarios and build experiences that reflect the strength of mixed reality.

Consider the [OnSight collaboration tool](#) developed by NASA's Jet Propulsion Laboratory. Scientists working on data from the Mars rover missions can collaborate with colleagues in real-time within the data from the Martian landscape.



A scientist explores an environment using a **3D, mixed-reality** device with a **small** group of **remote** colleagues using **3D and 2D** devices. The **collaboration is synchronous** (but can be revisited asynchronously) and the physical environments are (virtually) **similar**.

Experiences like OnSight present new opportunities to collaborate. From physically pointing out elements in the virtual environment to standing next to a colleague and sharing their perspective as they explain their findings. OnSight uses the lens of immersion and presence to rethink sharing experiences in mixed reality.

Intuitive collaboration is the bedrock of conversation and working together and understanding how we can apply this intuition to the complexity of mixed reality is crucial. If we can not only recreate sharing experiences in mixed reality but supercharge them, it will be a paradigm shift for the future of work. Designing for shared experiences in mixed reality is new and exciting space — and we're only at the beginning.

Get started sharing experiences

The key to shared experiences is multiple users seeing the same holograms in the world on their own device. One of the most common methods to do this is to use spatial anchors in your app. One way to achieve shared anchors would be:

- First the user places the hologram.
- App creates a [spatial anchor](#) to pin that hologram precisely in the world.
- App can then export this anchor and supporting tracking information to another HoloLens using a networking solution.
- That shared anchor can now be imported by another HoloLens along with the supporting tracking data that lets that HoloLens locate the shared anchor in their world space.

With a shared spatial anchor, the app on each device now has a common coordinate system in which they can place content. Now the app can ensure to position and orient the hologram at the same location.

See also

- [Shared spatial anchors in DirectX](#)
- [Shared experiences in Unity](#)
- [Spectator view](#)

Locatable camera

11/6/2018 • 11 minutes to read • [Edit Online](#)

HoloLens includes a world-facing camera mounted on the front of the device which enables apps to see what the user sees. Developers have access to and control of the camera just as they would for color cameras on smartphones, portables, or desktops. The same universal windows [media capture](#) and windows media foundation APIs that work on mobile and desktop work on HoloLens. Unity [has also wrapped these windows APIs](#) to abstract simple usage of the camera on HoloLens for tasks such as taking regular photos and videos (with or without holograms) and locating the camera's position in and perspective on the scene.

Device Camera Information

- Fixed focus with auto white balance and auto exposure and full image processing pipe
- A white Privacy LED facing the world will illuminate whenever the camera is active
- The camera supports the following modes (all modes are 16:9 aspect ratio) at 30, 24, 20, 15, and 5 fps:

VIDEO	PREVIEW	STILL	HORIZONTAL FIELD OF VIEW (H-FOV)	SUGGESTED USAGE
1280x720	1280x720	1280x720	45deg	(default mode)
N/A	N/A	2048x1152	67deg	Highest resolution still image
1408x792	1408x792	1408x792	48deg	Overscan (padding) resolution for video stabilization
1344x756	1344x756	1344x756	67deg	Large FOV video mode with overscan
896x504	896x504	896x504	48deg	Low power / Low resolution mode for image processing tasks

Locating the Device Camera in the World

When HoloLens takes photos and videos, the captured frames include the location of the camera in the world, as well as the perspective projection of the camera. This allows applications to reason about the position of the camera in the real world for augmented imaging scenarios. Developers can creatively roll their own scenarios using their favorite image processing or custom computer vision libraries.

"Camera" elsewhere in HoloLens documentation may refer to the "virtual game camera" (the frustum the app renders to). Unless denoted otherwise, "camera" on this page refers to the real-world RGB color camera.

The details on this page cover [Media Foundation Attributes](#), however there are also APIs to pull camera intrinsics using [WinRT APIs](#).

Images with Coordinate Systems

Each image frame (whether photo or video) includes a coordinate system, as well as two important transforms.

The "view" transform maps from the provided coordinate system to the camera, and the "projection" maps from the camera to pixels in the image. Together, these transforms define for each pixel a ray in 3D space representing the path taken by the photons that produced the pixel. These rays can be related to other content in the app by obtaining the transform from the frame's coordinate system to some other coordinate system (e.g. from a [stationary frame of reference](#)). To summarize, each image frame provides the following:

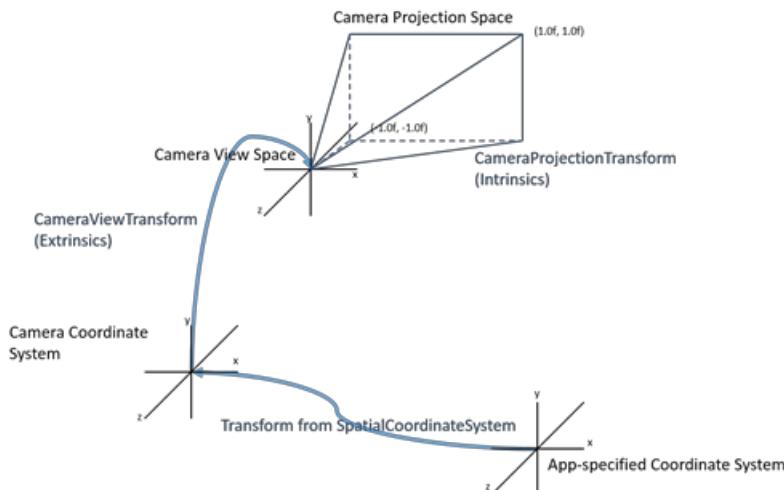
- Pixel Data (in RGB/NV12/JPEG/etc. format)
- 3 pieces of metadata (stored as [IMFAttributes](#)) that make each frame "locatable":

ATTRIBUTE NAME	TYPE	GUID	DESCRIPTION
MFSampleExtension_Spatial_CameraCoordinateSystem	IUnknown (SpatialCoordinateSystem)	{9D13C82F-2199-4E67-91CD-D1A4181F2534}	Stores the coordinate system of the captured frame
MFSampleExtension_Spatial_CameraViewTransform	Blob (Matrix4x4)	{4E251FA4-830F-4770-859A-4B8D99AA809B}	Stores the camera's extrinsic transform in the coordinate system
MFSampleExtension_Spatial_CameraProjectionTransform	Blob (Matrix4x4)	{47F9FCB5-2A02-4F26-A477-792FDF95886A}	Stores the camera's projection transform

The projection transform represents the intrinsic properties (focal length, center of projection, skew) of the lens mapped onto an image plane that extends from -1 to +1 in both the X and Y axis.

Matrix4x4 format	Terms
m11 m12 m13 m14	fx 0 0 0
m21 m22 m23 m24	skew fy 0 0
m31 m32 m33 m34	cx cy A -1
m41 m42 m43 m44	0 0 B 0

Different applications will have different coordinate systems. Here's an overview of the flow to locate a camera pixel for a single application:



Camera to Application-specified Coordinate System

To go from the 'CameraView' and 'CameraCoordinateSystem' to your application/world coordinate system, you'll need the following:

Locatable camera in Unity: CameraToWorldMatrix is automatically provided by PhotoCaptureFrame class(so you don't need to worry about the CameraCoordinateSystem transforms).

Locatable camera in DirectX: Shows the fairly straightforward way to query for the transform between the

camera's coordinate system and your own application coordinate system(s).

Application-specified Coordinate System to Pixel Coordinates

Let's say you wanted to find or draw at a specific 3d location on a camera image:

The view and projection transforms, while both 4x4 matrices, need to be utilized slightly differently. Namely after performing the Projection, one would 'normalize by w', this extra step in the projection simulates how multiple different 3d locations can end up as the same 2d location on a screen (i.e. anything along a certain ray will show up on the same pixel). So key points (in shader code):

```
// Usual 3d math:  
float4x4 WorldToCamera = inverse( CameraToWorld );  
float4 CameraSpacePos = mul( WorldToCamera, float4( WorldSpacePos.xyz, 1 ) ); // use 1 as the W component  
// Projection math:  
float4 ImagePosUnnormalized = mul( CameraProjection, float4( CameraSpacePos.xyz, 1 ) ); // use 1 as the W  
component  
float2 ImagePosProjected = ImagePosUnnormalized.xy / ImagePosUnnormalized.w; // normalize by W, gives -1 to 1  
space  
float2 ImagePosZeroToOne = ( ImagePosProjected * 0.5 ) + float2( 0.5, 0.5 ); // good for GPU textures  
int2 PixelPos = int2( ImagePosZeroToOne.x * ImageWidth, ( 1 - ImagePosZeroToOne.y ) * ImageHeight ); // good  
for CPU textures
```

Pixel to Application-specified Coordinate System

Going from pixel to world coordinates is a little trickier:

```
float2 ImagePosZeroToOne = float2( PixelPos.x / ImageWidth, 1.0 - (PixelPos.y / ImageHeight) );  
float2 ImagePosProjected = ( ( ImagePosZeroToOne * 2.0 ) - float2( 1, 1 ) ); // -1 to 1 space  
float3 CameraSpacePos = UnProjectVector( Projection, float3( ImagePosProjected, 1 ) );  
float3 WorldSpaceRayPoint1 = mul( CameraToWorld, float4( 0, 0, 0, 1 ) ); // camera location in world space  
float3 WorldSpaceRayPoint2 = mul( CameraToWorld, CameraSpacePos ); // ray point in world space
```

Where we define UnProject as:

```
public static Vector3 UnProjectVector(Matrix4x4 proj, Vector3 to)  
{  
    Vector3 from = new Vector3(0, 0, 0);  
    var axsX = proj.GetRow(0);  
    var axsY = proj.GetRow(1);  
    var axsZ = proj.GetRow(2);  
    from.z = to.z / axsZ.z;  
    from.y = (to.y - (from.z * axsY.z)) / axsY.y;  
    from.x = (to.x - (from.z * axsX.z)) / axsX.x;  
    return from;  
}
```

To find the actual world location of a point, you'll need either: two world rays and find their intersection, or a known size of the points.

Distortion Error

On HoloLens, the video and still image streams are undistorted in the system's image processing pipeline before the frames are made available to the application (the preview stream contains the original distorted frames).

Because only the projection matrix is made available, applications must assume image frames represent a perfect pinhole camera, however the undistortion function in the image processor may still leave an error of up to 10 pixels when using the projection matrix in the frame metadata. In many use cases, this error will not matter, but if you are aligning holograms to real world posters/markers, for example, and you notice a <10px offset (roughly 11mm for holograms positioned 2 meters away) this distortion error could be the cause.

Locatable Camera Usage Scenarios

Show a photo or video in the world where it was captured

The Device Camera frames come with a "Camera To World" transform, that can be used to show exactly where the device was when the image was taken. For example you could position a small holographic icon at this location (`CameraToWorld.MultiplyPoint(Vector3.zero)`) and even draw a little arrow in the direction that the camera was facing (`CameraToWorld.MultiplyVector(Vector3.forward)`).

Painting the world using a camera shader

In this section we'll create a material 'shader' that colors the world based on where it showed up in a device camera's view. Effectively what we'll do is that every vertex will figure out its location relative to the camera, and then every pixel will utilize the 'projection matrix' to figure out which image texel it is associated with. Lastly, and optionally, we'll fade out the corners of the image to make it appear more as a dream-like memory:

```
// In the vertex shader:  
float4 worldSpace = mul( ObjectToWorld, float4( vertexPos.xyz, 1));  
float4 cameraSpace = mul( CameraWorldToLocal, float4(worldSpace.xyz, 1));  
  
// In the pixel shader:  
float4 unprojectedTex = mul( CameraProjection, float4( cameraSpace .xyz, 1));  
float2 projectedTex = (unprojectedTex.xy / unprojectedTex.w);  
float2 unitTexcoord = ((projectedTex * 0.5) + float4(0.5, 0.5, 0, 0));  
float4 cameraTextureColor = tex2D(_CameraTex, unitTexcoord);  
// Fade out edges for better look:  
float pctInView = saturate((1.0 - length(projectedTex.xy)) * 3.0);  
float4 finalColor = float4( cameraTextureColor.rgb, pctInView );
```

Tag / Pattern / Poster / Object Tracking

Many mixed reality applications use a recognizable image or visual pattern to create a trackable point in space. This is then used to render objects relative to that point or create a known location. Some uses for HoloLens include finding a real world object tagged with fiducials (e.g. a TV monitor with a QR code), placing holograms over fiducials, and visually pairing with non-HoloLens devices like tablets that have been setup to communicate with HoloLens via Wi-Fi.

To recognize a visual pattern, and then place that object in the applications world space, you'll need a few things:

1. An image pattern recognition toolkit, such as QR code, AR tags, face finder, circle trackers, OCR etc.
2. Collect image frames at runtime, and pass them to the recognition layer
3. Unproject their image locations back into world positions, or likely world rays. See
4. Position your virtual models over these world locations

Some important image processing links:

- [OpenCV](#)
- [QR Tags](#)
- [FaceSDK](#)
- [Microsoft Translator](#)

Keeping an interactive application frame-rate is critical, especially when dealing with long-running image recognition algorithms. For this reason we commonly use the following pattern:

1. Main Thread: manages the camera object
2. Main Thread: requests new frames (async)
3. Main Thread: pass new frames to tracking thread
4. Tracking Thread: processes image to collect key points

5. Main Thread: moves virtual model to match found key points
6. Main Thread: repeat from step 2

Some image marker systems only provide a single pixel location (others provide the full transform in which case this section will not be needed), which equates to a ray of possible locations. To get to a single 3d location we can then leverage multiple rays and find the final result by their approximate intersection. To do this you'll need to:

1. Get a loop going collecting multiple camera images
2. Find the [associated feature points](#), and their world rays
3. When you have a dictionary of features, each with multiple world rays, you can use the following code to solve for the intersection of those rays:

```
public static Vector3 ClosestPointBetweenRays(
    Vector3 point1, Vector3 normalizedDirection1,
    Vector3 point2, Vector3 normalizedDirection2) {
    float directionProjection = Vector3.Dot(normalizedDirection1, normalizedDirection2);
    if (directionProjection == 1) {
        return point1; // parallel lines
    }
    float projection1 = Vector3.Dot(point2 - point1, normalizedDirection1);
    float projection2 = Vector3.Dot(point2 - point1, normalizedDirection2);
    float distanceAlongLine1 = (projection1 - directionProjection * projection2) / (1 - directionProjection * directionProjection);
    float distanceAlongLine2 = (projection2 - directionProjection * projection1) / (directionProjection * directionProjection - 1);
    Vector3 pointOnLine1 = point1 + distanceAlongLine1 * normalizedDirection1;
    Vector3 pointOnLine2 = point2 + distanceAlongLine2 * normalizedDirection2;
    return Vector3.Lerp(pointOnLine2, pointOnLine1, 0.5f);
}
```

Given two or more tracked tag locations, you can position a modelled scene to fit the users current scenario. If you can't assume gravity, then you'll need three tag locations. In many cases we use a simple color scheme where white spheres represent real-time tracked tag locations, and blue spheres represent modelled tag locations, this allows the user to visually gauge the alignment quality. We assume the following setup in all our applications:

- Two or more modelled tag locations
- One 'calibration space' which in the scene is the parent of the tags
- Camera feature identifier
- Behavior which moves the calibration space to align the modelled tags with the real-time tags (we are careful to move the parent space, not the modelled markers themselves, because other connect is positions relative to them).

```
// In the two tags case:
Vector3 idealDelta = (realTags[1].EstimatedWorldPos - realTags[0].EstimatedWorldPos);
Vector3 curDelta = (modelledTags[1].transform.position - modelledTags[0].transform.position);
if (IsAssumeGravity) {
    idealDelta.y = 0;
    curDelta.y = 0;
}
Quaternion deltaRot = Quaternion.FromToRotation(curDelta, idealDelta);
trans.rotation = Quaternion.LookRotation(deltaRot * trans.forward, trans.up);
trans.position += realTags[0].EstimatedWorldPos - modelledTags[0].transform.position;
```

Render holograms from the Camera's position

Note: If you are trying to create your own [Mixed reality capture \(MRC\)](#), which blends holograms with the Camera stream, you can use the [MRC effects](#) or enable the `showHolograms` property in [Locatable camera in Unity](#).

If you'd like to do a special render directly on the RGB Camera stream, it's possible to render holograms in space

from the Camera's position in sync with a video feed in order to provide a custom hologram recording/live preview.

In Skype, we do this to show the remote client what the HoloLens user is seeing and allow them to interact with the same holograms. Before sending over each video frame through the Skype service, we grab each frame's corresponding camera data. We then package the camera's extrinsic and intrinsic metadata with the video frame and then send it over the Skype service.

On the receiving side, using Unity, we've already synced all of the holograms in the HoloLens user's space using the same coordinate system. This allows us to use the camera's extrinsic metadata to place the Unity camera in the exact place in the world (relative to the rest of the holograms) that the HoloLens user was standing when that video frame was captured, and use the camera intrinsic information to ensure the view is the same.

Once we have the camera set up properly, we combine what holograms the camera sees onto the frame we received from Skype, creating a mixed reality view of what the HoloLens user sees using `Graphics.Blit`.

```
private void OnFrameReceived(Texture frameTexture, Vector3 cameraPosition, Quaternion cameraRotation,
Matrix4x4 cameraProjectionMatrix)
{
    //set material that will be blitted onto the RenderTexture
    this.compositeMaterial.SetTexture(CompositeRenderer.CameraTextureMaterialProperty, frameTexture);
    //set the camera to be that of the HoloLens's device camera
    this.Camera.transform.position = cameraPosition;
    this.Camera.transform.rotation = cameraRotation;
    this.Camera.projectionMatrix = cameraProjectionMatrix;
    //trigger the Graphics's Blit now that the frame and camera are set up
    this.TextureReady = false;
}
private void OnRenderImage(RenderTexture source, RenderTexture destination)
{
    if (!this.TextureReady)
    {
        Graphics.Blit(source, destination, this.compositeMaterial);
        this.TextureReady = true;
    }
}
```

Track or Identify Tagged Stationary or Moving real-world objects/faces using LEDs or other recognizer libraries

Examples:

- Industrial robots with LEDs (or QR codes for slower moving objects)
- Identify and recognize objects in the room
- Identify and recognize people in the room (e.g. place holographic contact cards over faces)

See also

- [Locatable camera in DirectX](#)
- [Locatable camera in Unity](#)
- [Mixed reality capture](#)
- [Mixed reality capture for developers](#)
- [Media capture introduction](#)

Mixed reality capture for developers

11/6/2018 • 7 minutes to read • [Edit Online](#)

Since a user could take a [mixed reality capture](#) (MRC) photo or video at any time, there are a few things that you should keep in mind when developing your application. This includes best practices for MRC visual quality and being responsive to system changes while MRCs are being captured.

Developers can also seamlessly integrate mixed reality capture and insertion into their apps.

The importance of quality MRC

Mixed reality captured photos and videos are likely the first exposure a user will have of your app. Whether as mixed reality screenshots on your Microsoft Store page or from other users sharing MRCs on social networks. You can use MRC to demo your app, educate users, encourage users to share their mixed world interactions, and for user research and problem solving.

How MRC impacts your app

Enabling MRC in your app

By default, an app does not have to do anything to enable users to take mixed reality captures.

Disabling MRC in your app

When a 2D app uses [DXGI_PRESENT_RESTRICT_TO_OUTPUT](#) or [DXGI_SWAP_CHAIN_FLAG_HW_PROTECTED](#) to show protected content with a properly-configured swap chain, the app's visual content will be automatically obscured while mixed reality capture is running.

Knowing when MRC is active

The [AppCapture](#) class can be used by an app to know when system mixed reality capture is running (for either audio or video).

NOTE

AppCapture's [GetForCurrentView](#) API can return null if mixed reality capture isn't available on the device. It's also important to de-register the CapturingChanged event when your app is suspended, otherwise MRC can get into a blocked state.

Best practices (HoloLens-specific)

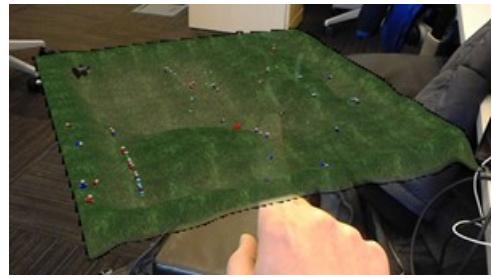
MRC is expected to work without additional work from developers, but there are a few things to be aware of to provide the best mixed reality capture experience of your app.

MRC uses the hologram's alpha channel to blend with the camera imagery

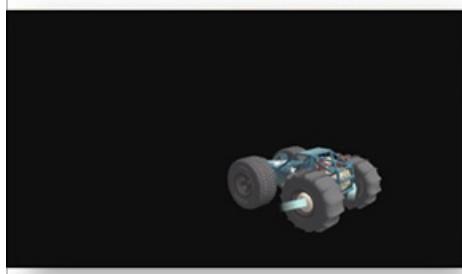
The most important step is to make sure your app is clearing to transparent black instead of clearing to opaque black. In Unity, this is done by default with the MixedRealityToolkit but if you are developing in non-Unity, you may need to make a one line change.

Here are some of the artifacts you might see in MRC if your app is not clearing to transparent black:

Example Failures: Black edges around the content (failing to clear to transparent black)



Example Failures: The entire background scene of the hologram appears black. Setting a background alpha value of 1 results in a black background



Expected Result: Holograms appear properly blended with the real-world (expected result if clearing to transparent black)



Solution:

- Change any content that is showing up as opaque black to have an alpha value of 0.
- Ensure that the app is clearing to transparent black.
- Unity defaults to clear to clear automatically with the MixedRealityToolkit, but if it's a non-Unity app you should modify the color used with ID3D11DeviceContext::ClearRenderTargetView(). You want to ensure you clear to transparent black (0,0,0,0) instead of opaque black (0,0,0,1).

You can now tune the alpha values of your assets if you'd like, but typically don't need to. Most of the time, MRCs will look good out of the box. MRC assumes pre-multiplied alpha. The alpha values will only affect the MRC capture.

What to expect when MRC is enabled (HoloLens-specific)

- The system will throttle the application to 30Hz rendering. This creates some headroom for MRC to run so the app doesn't need to keep a constant budget reserve and also matches the MRC video record framerate of 30fps
- Hologram content in the right eye of the device may appear to "sparkle" when recording/streaming MRC: text may become more difficult to read and hologram edges may appear more jaggy
- MRC photos and videos will respect the application's [focus point](#) if the application has enabled it, which will help ensure holograms are accurately positioned. For videos, the Focus Point is smoothed so holograms may appear to slowly drift into place if the Focus Point depth changes significantly. Holograms that are at different depths from the focus point may appear offset from the real world (see example below where Focus Point is set at 2 meters but hologram is positioned at 1 meter).

	In-Device View	MRC Video
2 meters		
1 meter		

Integrating MRC functionality from within your app

Your mixed reality app can initiate MRC photo or video capture from within the app, and the content captured is made available to your app without being stored to the device's "Camera roll." You can create a custom MRC recorder or take advantage of built-in camera capture UI.

MRC with built-in camera UI

Developers can use the [Camera Capture UI API](#) to get a user-captured mixed reality photo or video with just a few lines of code.

This API launches the built-in MRC camera UI, from which the user can take a photo or video, and returns the resulting capture to your app. If you want to create your own camera UI, or need lower-level access to the capture stream, you can create a custom Mixed Reality Capture recorder.

Creating a custom MRC recorder

While the user can always trigger a photo or video using the system MRC capture service, an application may want to build a custom camera app but include holograms in the camera stream just like MRC. This allows the application to kick off captures on behalf of the user, build custom recording UI, or customize MRC settings to name a few examples.

HoloStudio adds a custom MRC camera using MRC effects



Unity Applications should see [Locatable_camera_in_Unity](#) for the property to enable holograms.

Other applications can do this by using the [Windows Media Capture APIs](#) to control the Camera and add an MRC Video and Audio effect to include virtual holograms and application audio in stills and videos.

Applications have two options to add the effect:

- The older API: [Windows.Media.Capture.MediaCapture.AddEffectAsync\(\)](#)
- The new Microsoft recommended API (returns an object, making it possible to manipulate dynamic properties): [Windows.Media.Capture.MediaCapture.AddVideoEffectAsync\(\)](#) / [Windows.Media.Capture.MediaCapture.AddAudioEffectAsync\(\)](#) which require the app create its own implementation of [IVideoEffectDefinition](#) and [IAudioEffectDefinition](#). Please see the MRC effect sample for sample usage.

(Note that these namespaces will not be recognized by Visual Studio, but the strings are still valid)

MRC Video Effect (**Windows.Media.MixedRealityCapture.MixedRealityCaptureVideoEffect**)

PROPERTY NAME	TYPE	DEFAULT VALUE	DESCRIPTION
StreamType	UINT32 (MediaStreamType)	1 (VideoRecord)	Describe which capture stream this effect is used for. Audio is not available.
HologramCompositionEnabled	boolean	TRUE	Flag to enable or disable holograms in video capture.
RecordingIndicatorEnabled	boolean	TRUE	Flag to enable or disable recording indicator on screen during hologram capturing.
VideoStabilizationEnabled	boolean	FALSE	Flag to enable or disable video stabilization powered by the HoloLens tracker.
VideoStabilizationBufferLength	UINT32	0	Set how many historical frames are used for video stabilization. 0 is 0-latency and nearly "free" from a power and performance perspective. 15 is recommended for highest quality (at the cost of 15 frames of latency and memory).
GlobalOpacityCoefficient	float	0.9 (HoloLens) 1.0 (Immersive headset)	Set global opacity coefficient of hologram in range from 0.0 (fully transparent) to 1.0 (fully opaque).
BlankOnProtectedContent	boolean	FALSE	Flag to enable or disable returning an empty frame if there is a 2d UWP app showing protected content. If this flag is false and a 2d UWP app is showing protected content, the 2d UWP app will be replaced by a protected content texture in both the headset and in the mixed reality capture.

PROPERTY NAME	TYPE	DEFAULT VALUE	DESCRIPTION
ShowHiddenMesh	boolean	FALSE	Flag to enable or disable showing the holographic camera's hidden area mesh and neighboring content.

MRC Audio Effect (**Windows.Media.MixedRealityCapture.MixedRealityCaptureAudioEffect**)

PROPERTY NAME	TYPE	DEFAULT VALUE	DESCRIPTION
MixerMode	UINT32	2	<ul style="list-style-type: none"> • 0 : Mic audio only • 1 : System audio only • 2 : Mic and System audio

Simultaneous MRC limitations

There are certain limitations around multiple apps accessing MRC at the same time.

Photo/video camera access

The photo/video camera is limited to the number of processes that can access it at the same time. While a process is recording video or taking a photo any other process will fail to acquire the photo/video camera. (this applies to both Mixed Reality Capture and standard photo/video capture)

With the Windows 10 April 2018 Update, this restriction does not apply if the built-in MRC camera UI is used to take a photo or a video after an app has started using the photo/video camera. When this happens, the resolution and framerate of the built-in MRC camera UI might be reduced from its normal values.

With the Windows 10 October 2018 Update, this restriction does not apply to streaming MRC over Miracast.

MRC access

With the Windows 10 April 2018 Update, there is no longer a limitation around multiple apps accessing the MRC stream (however, the access to the photo/video camera still has limitations).

Previous to the Windows 10 April 2018 Update, an app's custom MRC recorder was mutually exclusive with system MRC (capturing photos, capturing videos, or streaming from the Windows Device Portal).

See also

- [Mixed reality capture](#)
- [Spectator view](#)

HoloLens Research mode

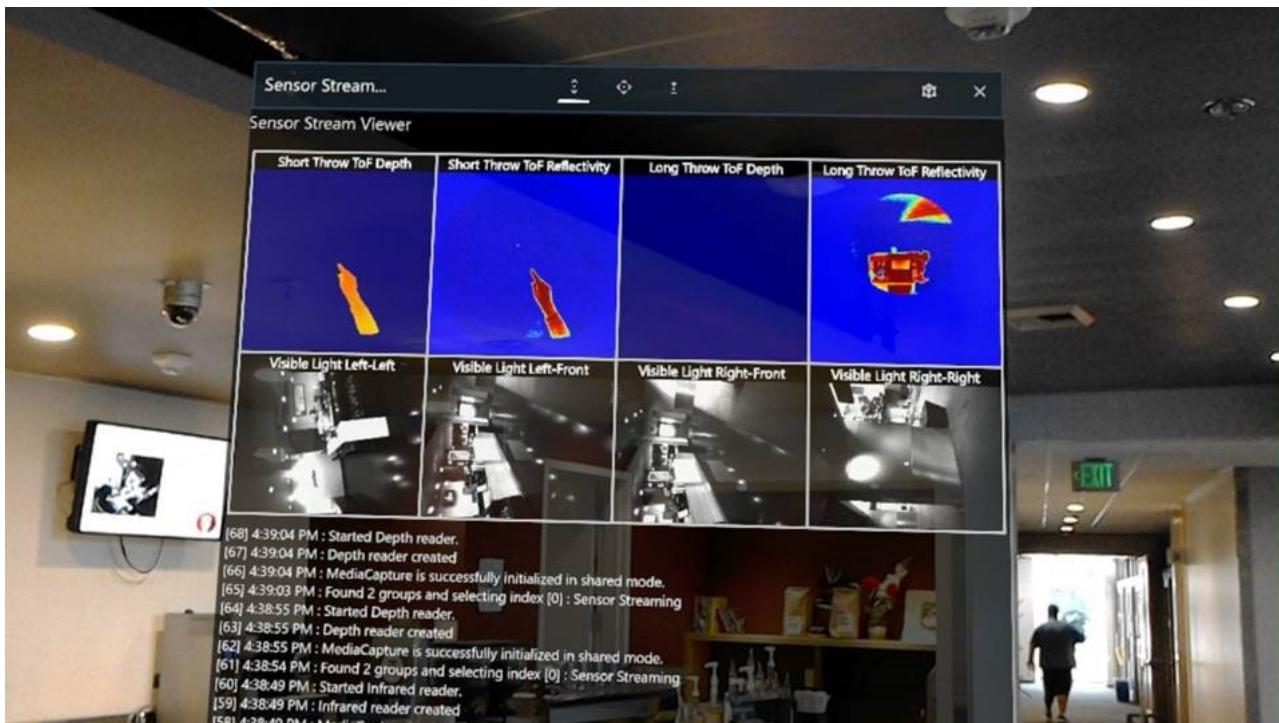
11/6/2018 • 2 minutes to read • [Edit Online](#)

NOTE

This feature was added as part of the [Windows 10 April 2018 Update](#) for HoloLens, and is not available on earlier releases.

Research mode is a new capability of HoloLens that provides application access to the key sensors on the device. These include:

- The four environment tracking cameras used by the system for map building and head tracking.
- Two versions of the depth camera data – one for high-frequency (30 FPS) near-depth sensing, commonly used in hand tracking, and the other for lower-frequency (1 FPS) far-depth sensing, currently used by Spatial Mapping,
- Two versions of an IR-reflectivity stream, used by the HoloLens to compute depth, but valuable in its own right as these images are illuminated from the HoloLens and reasonably unaffected by ambient light.



A mixed reality capture of a test application that displays the eight sensor streams available in Research mode

Device support

FEATURE	HOLOLENS	IMMERSIVE HEADSETS
Research mode	✓ <input type="checkbox"/>	

Before using Research mode

Research mode is well named: it is intended for academic and industrial researchers trying out new ideas in the fields of Computer Vision and Robotics. Research mode is not intended for applications that will be deployed

across an enterprise or made available in the Microsoft Store. The reason for this is that Research mode lowers the security of your device and consumes significantly more battery power than normal operation. Microsoft is not committing to supporting this mode on any future devices. Thus, we recommend you use it to develop and test new ideas; however, you will not be able to widely deploy applications that use Research mode or have any assurance that it will continue to work on future hardware.

Enabling Research mode

Research mode is a sub-mode of developer mode. You first need to enable developer mode in the Settings app (**Settings > Update & Security > For developers**):

1. Set "Use developer features" to **On**
2. Set "Enable Device Portal" to **On**

Then using a web browser that is connected to the same Wi-Fi network as your HoloLens, navigate to the IP address of your HoloLens (obtained through **Settings > Network & Internet > Wi-Fi > Hardware properties**). This is the [Device Portal](#), and you will find a "Research mode" page in the "System" section of the portal:

The screenshot shows the HoloLens Device Portal interface. The top navigation bar includes links for Feedback, Online, Cool, 100% Power, Help, and a menu icon. On the left, a sidebar lists various sections: Views, Performance, System (with App Crash Dumps and Device manager), File explorer, Logging, Networking, Preferences, Research mode (which is selected and highlighted in blue), Simulation, Virtual Input, and Scratch. The main content area is titled "Research mode". It contains a checkbox labeled "Allow access to sensor streams". Below the checkbox, a note states: "When enabled, applications will be able to access low-level HoloLens sensor streams in addition to the environment data that all apps can access. These low-level streams are for research purposes only and are not available to apps that are in the Windows Store." A link "Learn more about HoloLens Research Mode and get sample applications here." is provided. A "Warnings" section follows, enclosed in a red-bordered box: "A malicious application may misuse the sensor data or potentially compromise the integrity of the system. For this reason research mode is recommended only for devices that are being actively used for research tasks." Another note in the box says: "While enabled, research mode affects performance and battery life regardless of whether or not any applications are running." A final note at the bottom of the box states: "Changes to this setting require a reboot to take effect."

Research mode in the HoloLens Device Portal

After selecting **Allow access to sensor streams**, you will need to reboot HoloLens. You can do this from the Device Portal, under the "Power" menu item at the top of the page.

Once your device has rebooted, applications that have been loaded through Device Portal should be able to access Research mode streams.

Using sensor data in your apps

Applications can access sensor stream data by opening [Media Foundation](#) streams in exactly the same way they access the photo/video camera stream.

All APIs that work for HoloLens development are also available when in Research mode. In particular, the application can know precisely where HoloLens is in 6DoF space at each sensor frame capture time.

Sample applications showing how you access the various Research mode streams, how to use the intrinsics and extrinsics, and how to record streams are available in the [HoloLensForCV GitHub repo](#).

Known issues

See the [issue tracker](#) in the HoloLensForCV repository.

See also

- [Microsoft Media Foundation](#)
- [HoloLensForCV GitHub repo](#)
- [Using the Windows Device Portal](#)

Open source projects

11/6/2018 • 2 minutes to read • [Edit Online](#)

Tools



Mixed Reality Toolkit

The Mixed Reality Toolkit is a collection of scripts and components intended to accelerate development of applications targeting Microsoft HoloLens and Windows Mixed Reality headsets. The project is aimed at reducing barriers to entry to create mixed reality applications and contribute back to the community as we all grow.

Mixed Reality Toolkit - Unity (MRTK)

Mixed Reality Toolkit - Unity uses code from the base Mixed Reality Toolkit and makes it easier to consume in Unity.

Mixed Reality Companion Kit

This is a Mixed Reality Toolkit-style repository for code bits and components that may not run directly on Microsoft HoloLens or immersive headsets but instead pair with them to build experiences targeting Windows Mixed Reality.

Windows Device Portal Wrapper

A client library that wraps the Windows Device Portal REST APIs.



Mixed Reality Design Labs (MRDL)

The Mixed Reality Design Labs (MRDL) is a collection of well-documented, open-source samples, based on the foundation of [Mixed Reality Toolkit - Unity \(MRTK\)](#). The goal is to inspire creators and help them build compelling, efficient Mixed Reality experiences.

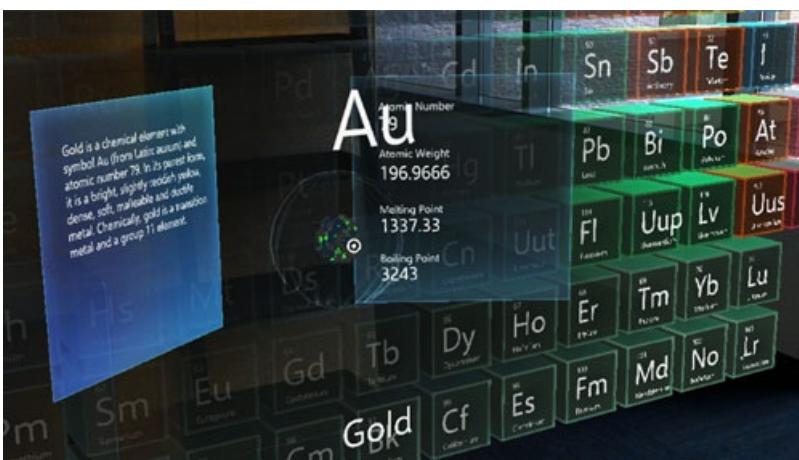
MRTK offers building-block components, and MRDL leverages them to offer more complete experiences and samples. As the name suggests, these samples are experimental/'works-in-progress', driven by experience design, which provide creators with concrete examples of best practices for app experiences, UX, and MRTK implementation. By 'experimental', we mean MRDL is not officially supported/maintained by Microsoft (e.g. updated to latest versions of Unity), whereas MRTK is officially supported/maintained.

Sample apps



Galaxy Explorer

The Galaxy Explorer Project is ready. You shared your ideas with the community, chose an app, watched a team build it, and can now get the source code.



Periodic Table of the Elements

Learn how to lay out an array of objects in 3D space with various surface types using an Object collection.



Lunar Module

Learn how to extend HoloLens base gestures with two-handed tracking and Xbox controller input.

Galaxy Explorer

11/6/2018 • 3 minutes to read • [Edit Online](#)

You shared your ideas. We're sharing the code.

The Galaxy Explorer Project is ready. You shared your ideas with the community, chose an app, watched a team build it, and can now get the source code. If you have a device, Galaxy Explorer Project is also available for download from the Windows Store for Microsoft HoloLens.

TIP

[Get the code on GitHub](#)

Our HoloLens [development team](#) of designers, artists, and developers built Galaxy Explorer and invited all of you to be part of this journey with them. After six weeks of core development and two weeks of refinement, this app is now ready for you! You can also follow along our whole journey through the video series below.

Share your idea

The Galaxy Explorer journey begins with the "Share your idea" campaign.

The Microsoft HoloLens community is bursting with spectacular ideas for how holographic computing will transform our world. We believe the most incredible HoloLens apps will come out of ideas you imagine together.

You shared over 5000 amazing ideas throughout those few weeks! Our development team reviewed the most successful and viable ideas and offered to build one of the top three ideas.

After a 24-hour Twitter poll, Galaxy Explorer was the winning idea! Our HoloLens development team of designers, artists, and developers built Galaxy Explorer and invited all of you to be part of this journey with them. You can follow the development process in the videos below.

Ep 1: Trust the Process

In Episode 1, the development team begins the creative process: brainstorming, conceiving, and deciding what to prototype.

Ep 2: Let's Do This

In Episode 2, the development team completes the prototyping phase – tackling hard problems and figuring out which ideas to pursue further.

Ep 3: Laying Foundations

In Episode 3, the team starts the first week of development – creating a plan, writing production code, creating art assets, and figuring out the user interface.

Ep 4: Make It Real

In Episode 4, the team dives deeper into development – bringing in scientific data, optimizing the rendering process, and incorporating spatial mapping.

Ep 5: See What Happens

In Episode 5, the development team tests the app, searches for bugs that need to be fixed, and refines the experience.

Ep 6: Coming to Life

In Episode 6, the team finishes the last week of development, prepares for two weeks of polish work, and reflects on the progress they've made

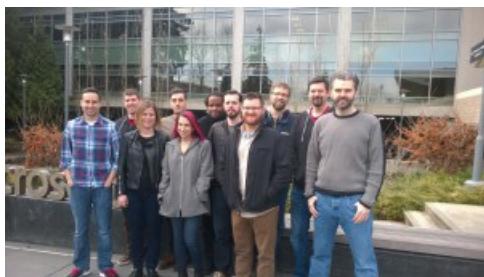
Ep 7: The Final Product

In Episode 7, the team completes the project and shares their code.

Case study

You can find even more insights and lessons from developing Galaxy Explorer by reading the "[Creating a galaxy in mixed reality](#)" case study.

Meet the team



Galaxy Explorer development team

We learned the building the right team is one of the most important investments we could make. We decided to organize similarly to a game studio for those of you familiar with that development model. We chose to have eleven core team members to control scope, since we had a fixed timeframe (create something cool before Build on March 30).

For this project, we started with a producer, Jessica who conducted planning, reviewing progress, and keeping things running day to day. She's the one with pink hair. We had a design director (Jon) and a senior designer (Peter). They held the creative vision for Galaxy Explorer. Jon is the one in glasses in the front row, and Peter is the second from the right in the back.

We had three developers – BJ (between Jon and Jessica), Mike (second row on the left), and Karim (second row middle, next to BJ). They figured out the technical solutions needed to realize that creative vision.

We started out with four artists full-time – a concept artist (Jedd, second from left in the back), a modeler (Andy, third from right in the back), a technical artist (Alex (right-most person)) and an animator (Steve (left-most person)). Each of them does more than that, too – but those are their primary responsibilities.

We'll had one full-time tester – Lena – who tested our builds every day, set up our build reviews, and reviewed features as they come online. Everyone tested constantly though, as we were always looking at our builds. Lena's the one rocking the leather jacket.

We are all a part of a larger studio here at Microsoft (think team in non-game development). There were a bunch of other people involved as well – we called on the talents of our art director, audio engineer and studio leadership frequently throughout the project, but those folks were shared resources with other projects our broader team has.

See also

- [Case study - Creating a galaxy in mixed reality](#)
- [Galaxy Explorer GitHub repo](#)

Periodic Table of the Elements

11/6/2018 • 3 minutes to read • [Edit Online](#)

NOTE

This article discusses an exploratory sample we've created in the [Mixed Reality Design Labs](#), a place where we share our learnings about and suggestions for mixed reality app development. Our design-related articles and code will evolve as we make new discoveries.

[Periodic Table of the Elements](#) is an open-source sample app from Microsoft's Mixed Reality Design Labs. With this project, you can learn how to lay out an array of objects in 3D space with various surface types using an [Object collection](#). Also learn how to create interactable objects that respond to standard inputs from HoloLens. You can use this project's components to create your own mixed reality app experience.



About the app

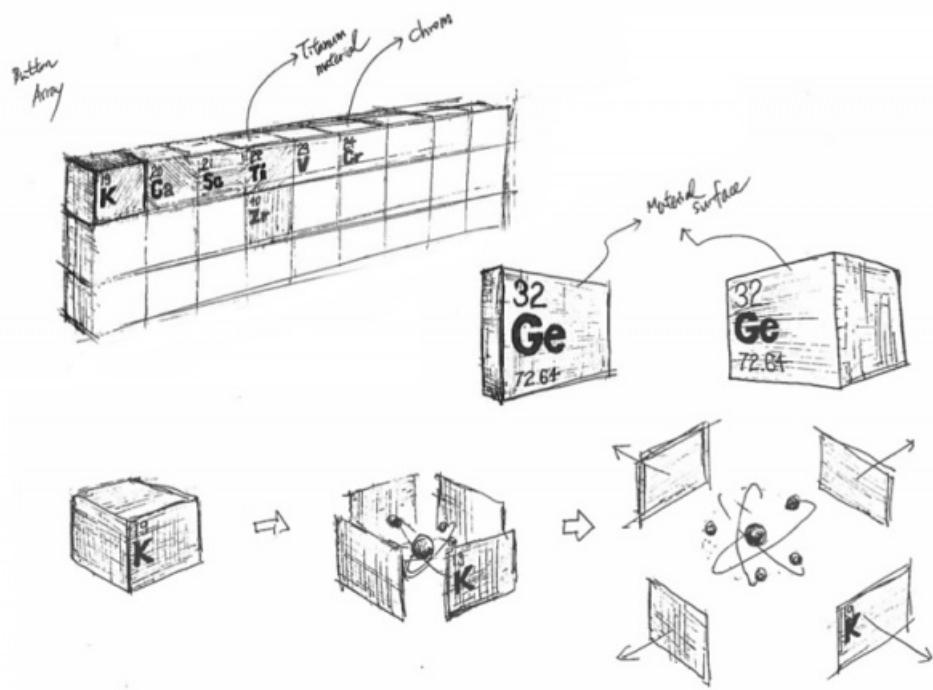
Periodic Table of the Elements visualizes the chemical elements and each of their properties in a 3D space. It incorporates the basic interactions of HoloLens such as gaze and air tap. Users can learn about the elements with animated 3D models. They can visually understand an element's electron shell and its nucleus - which is composed of protons and neutrons.

Background

After I first experienced HoloLens, a periodic table app was an idea I knew that I wanted to experiment with in mixed reality. Since each element has many data points that are displayed with text, I thought it would be great subject matter for exploring typographic composition in a 3D space. Being able to visualize the element's electron model was another interesting part of this project.

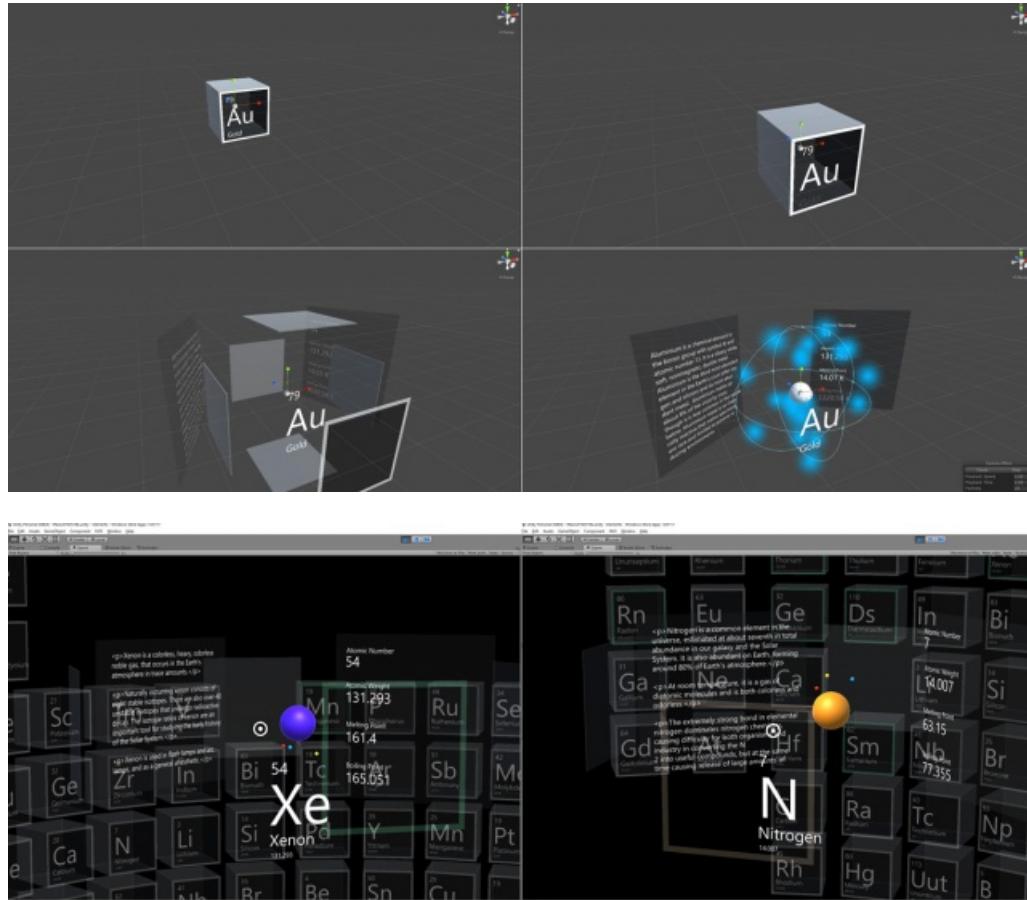
Design

For the default view of the periodic table, I imagined three-dimensional boxes that would contain the electron model of each element. The surface of each box would be translucent so that the user could get a rough idea of the element's volume. With gaze and air tap, the user could open up a detailed view of each element. To make the transition between table view and detail view smooth and natural, I made it similar to the physical interaction of a box opening in real life.



Design sketches

In detail view, I wanted to visualize the information of each element with beautifully rendered text in 3D space. The animated 3D electron model is displayed in the center area and can be viewed from different angles.



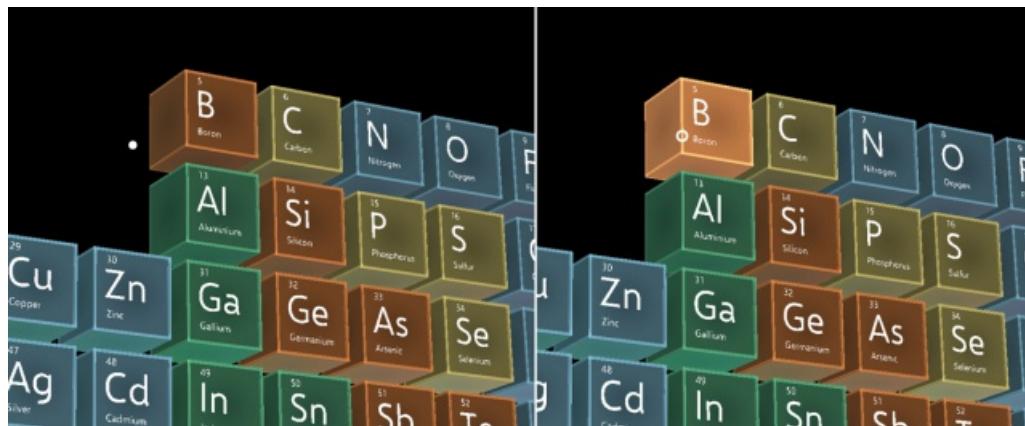
Interaction prototypes

The user can change the surface type by air tapping the buttons on the bottom of the table - they can switch between plane, cylinder, sphere and scatter.

Common controls and patterns used in this app

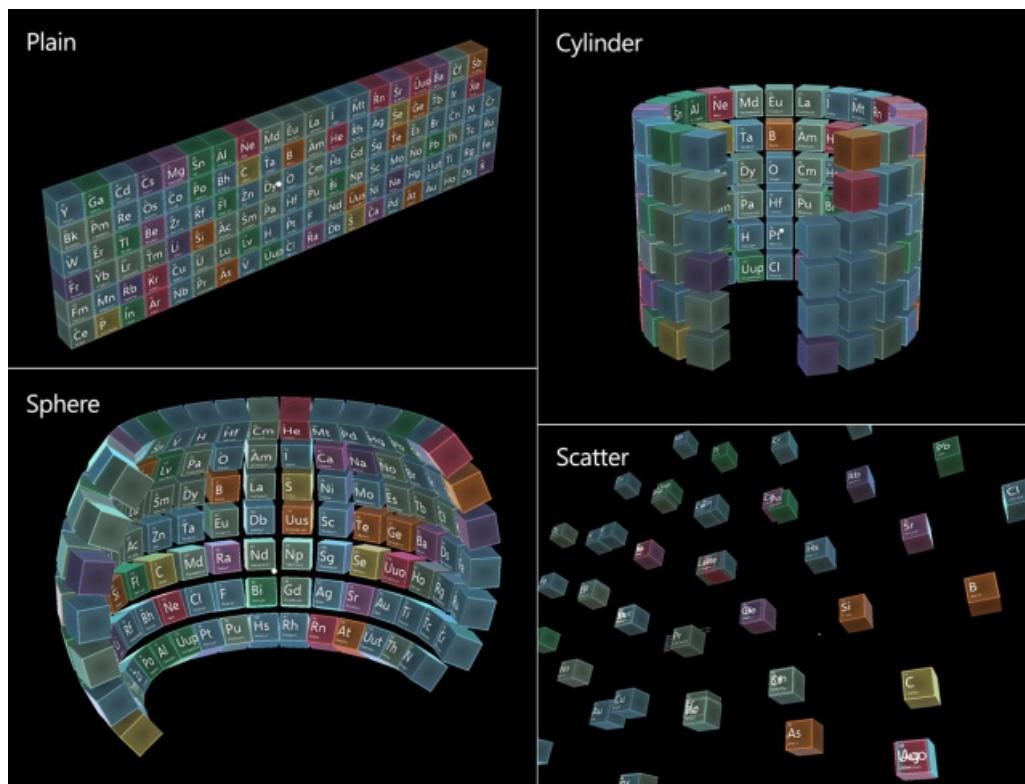
Interactable object (button)

[Interactable object](#) is an object which can respond to basic HoloLens inputs. It is provided as a prefab/script which you can easily apply to any object. For example, you can make a coffee cup in your scene interactable and respond to inputs such as gaze, air tap, navigation and manipulation gestures. [Learn more](#)



Object collection

[Object collection](#) is an object which helps you lay out multiple objects in various shapes. It supports plane, cylinder, sphere and scatter. You can configure additional properties such as radius, number of rows and the spacing. [Learn more](#)



Fitbox

By default, holograms will be placed in the location where the user is gazing at the moment the application is launched. This sometimes leads to unwanted result such as holograms being placed behind a wall or in the middle of a table. A fitbox allows a user to use gaze to determine the location where the hologram will be placed. It is made with a simple PNG image texture which can be easily customized with your own images or 3D objects.



Technical details

You can find scripts and prefabs for the Periodic Table of the Elements app on the [Mixed Reality Design Labs GitHub](#).

Application examples

Here are some ideas for what you could create by leveraging the components in this project.

Stock data visualization app

Using the same controls and interaction model as the Periodic Table of the Elements sample, you could build an app which visualizes stock market data. This example uses the Object collection control to lay out stock data in a spherical shape. You can imagine a detail view where additional information about each stock could be displayed in an interesting way.

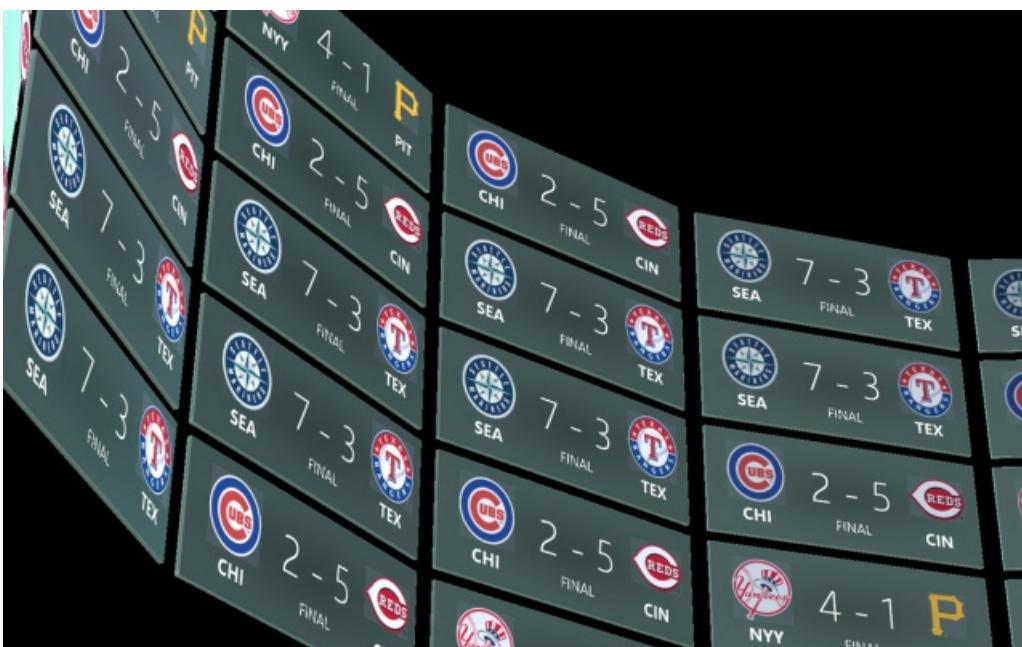




An example of how the Object collection used in the Periodic Table of the Elements sample app could be used in a finance app

Sports app

This is an example of visualizing sports data using Object collection and other components from the Periodic Table of the Elements sample app.



An example of how the Object collection used in the Periodic Table of the Elements sample app could be used in a sports app

About the author



Dong Yoon Park
UX Designer @Microsoft

See also

- [Interactable object](#)
- [Object collection](#)

Lunar Module

11/6/2018 • 8 minutes to read • [Edit Online](#)

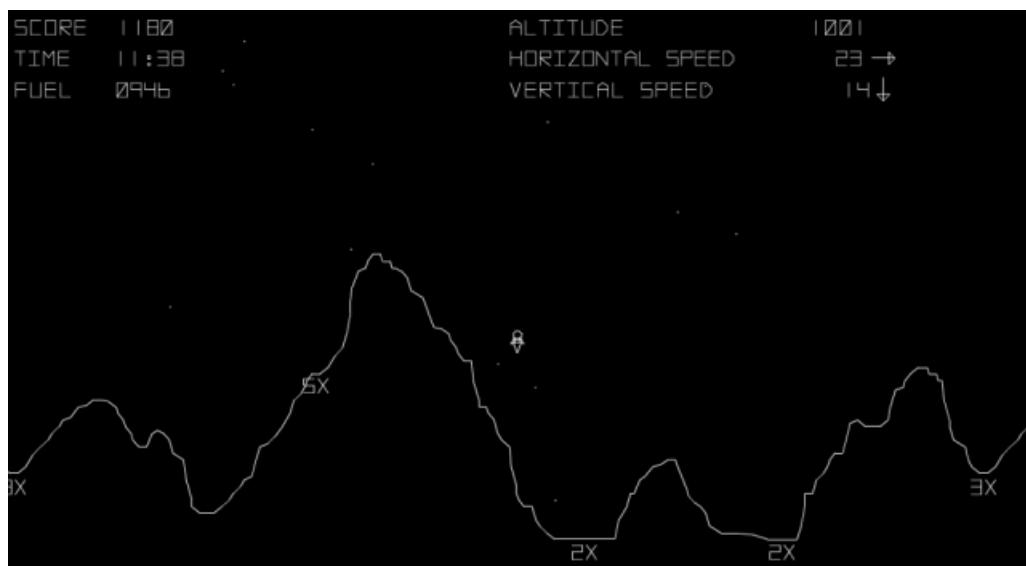
NOTE

This article discusses an exploratory sample we've created in the [Mixed Reality Design Labs](#), a place where we share our learnings about and suggestions for mixed reality app development. Our design-related articles and code will evolve as we make new discoveries.

[Lunar Module](#) is an open-source sample app from Microsoft's Mixed Reality Design Labs. With this project, you can learn how to extend Hololens' base gestures with two-handed tracking and Xbox controller input, create objects that are reactive to surface mapping and plane finding and implement simple menu systems. All of the project's components are available for use in your own mixed reality app experiences.

Rethinking classic experiences for Windows Mixed Reality

High up in the atmosphere, a small ship reminiscent of the Apollo module methodically surveys jagged terrain below. Our fearless pilot spots a suitable landing area. The descent is arduous but thankfully, this journey has been made many times before...



Original interface from Atari's 1979 Lunar Lander

[Lunar Lander](#) is an arcade classic where players attempt to pilot a moon lander onto a flat spot of lunar terrain. Anyone born in the 1970s has most likely spent hours in an arcade with their eyes glued to this vector ship plummeting from the sky. As a player navigates their ship toward a desired landing area the terrain scales to reveal progressively more detail. Success means landing within the safe threshold of horizontal and vertical speed. Points are awarded for time spent landing and remaining fuel, with a multiplier based on the size of the landing area.

Aside from the gameplay, the arcade era of games brought constant innovation of control schemes. From the simplest 4-way joystick and button configurations (seen in the iconic [Pac-Man](#)) to the highly specific and complicated schemes seen in the late 90s and 00s (like those in golf simulators and rail shooters). The input scheme used in the Lunar Lander machine is particularly intriguing for two reasons: curb appeal and immersion.



Atari's Lunar Lander arcade console

Why did Atari and so many other game companies decide to rethink input?

A kid walking through an arcade will naturally be intrigued by the newest, flashiest machine. But Lunar Lander features a novel input mechanic that stood out from the crowd.

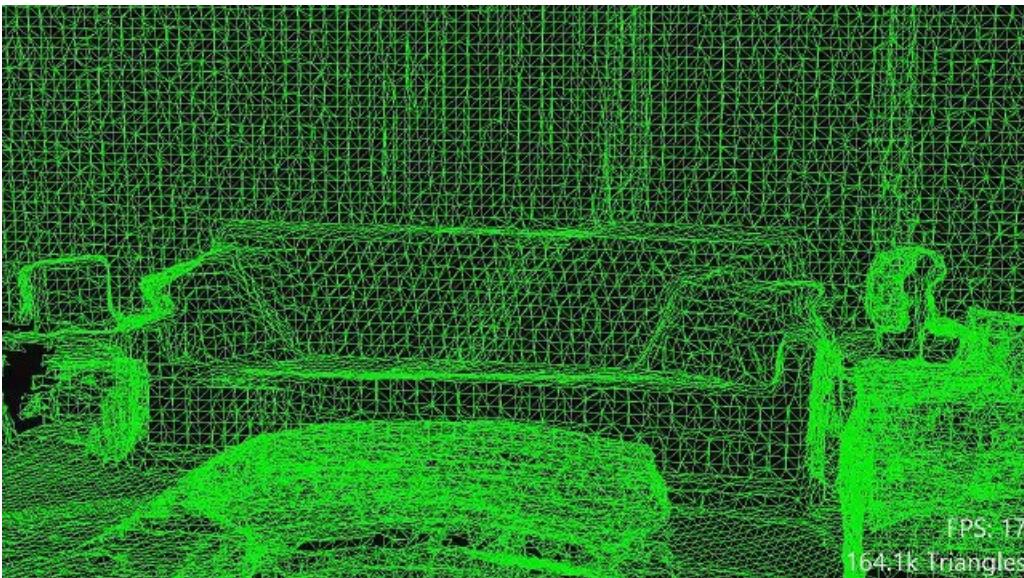
Lunar Lander uses two buttons for rotating the ship left and right and a **thrust lever** to control the amount of thrust the ship produces. This lever gives users a certain level of finesse a regular joystick can't provide. It is also happens to be a component common to modern aviation cockpits. Atari wanted Lunar Lander to immerse the user in the feeling that they were in fact piloting a lunar module. This concept is known as **tactile immersion**.

Tactile immersion is the experience of sensory feedback from performing repetitious actions. In this case, the repetitive action of adjusting the throttle lever and rotation which our eyes see and our ears hear, helps connect the player to the act of landing a ship on the moon's surface. This concept can be tied to the psychological concept of "flow." Where a user is fully absorbed in a task that has the right mixture of challenge and reward, or put more simply, they're "in the zone."

Arguably, the most prominent type of immersion in mixed reality is spatial immersion. The whole point of mixed reality is to fool ourselves into believing these digital objects exist in the real world. We're synthesizing holograms in our surroundings, spatially immersed in entire environments and experiences. This doesn't mean we can't still employ other types of immersion in our experiences just as Atari did with tactile immersion in Lunar Lander.

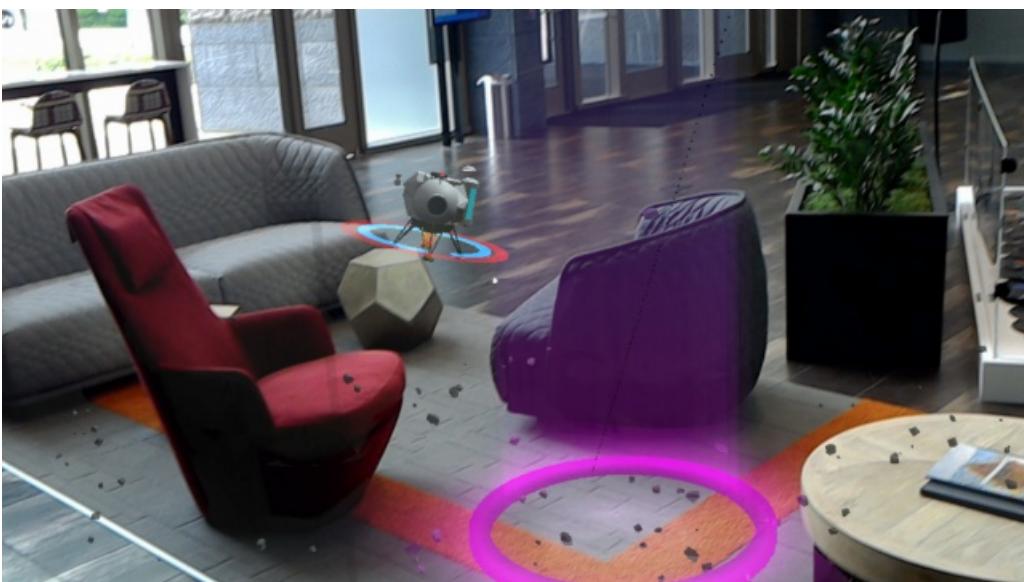
Designing with immersion

How might we apply tactile immersion to an updated, volumetric sequel to the Atari classic? Before tackling the input scheme, the game construct for 3-dimensional space needs to be addressed.



Visualizing spatial mapping in HoloLens

By leveraging a user's surroundings, we effectively have infinite terrain options for landing our lunar module. To make the game most like the original title, a user could potentially manipulate and place landing pads of varying difficulties in their environment.



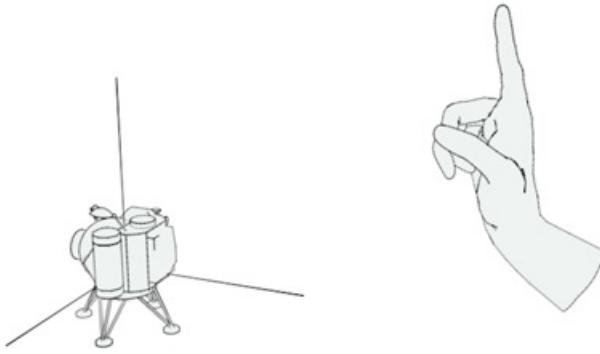
Flying the Lunar Module

Requiring the user to learn the input scheme, control the ship, and have a small target to land on is a lot to ask. A successful game experience features the right mix of challenge and reward. The user should be able to choose a level of difficulty, with the easiest mode simply requiring the user to successfully land in a user-defined area on a surface scanned by the HoloLens. Once a user gets the hang of the game, they can then crank up the difficulty as they see fit.

Adding input for hand gestures

HoloLens base input has only two gestures - [Air Tap and Bloom](#). Users don't need to remember contextual nuances or a laundry list of specific gestures which makes the platform's interface both versatile and easy to learn. While the system may only expose these two gestures, HoloLens as a device is capable of tracking two hands at once. Our ode to Lunar Lander is an [immersive app](#) which means we have the ability to extend the base set of gestures to leverage two hands and add our own delightfully tactile means for lunar module navigation.

Looking back at the original control scheme, **we needed to solve for thrust and rotation**. The caveat is rotation in the new context adds an additional axis (technically two, but the Y axis is less important for landing). The two distinct ship movements naturally lend themselves to be mapped to each hand:



Tap and drag gesture to rotate lander on all three axes

Thrust

The lever on the original arcade machine mapped to a scale of values, the higher the lever was moved the more thrust was applied to the ship. An important nuance to point out here is the user can take their hand off of the control and maintain a desired value. We can effectively use tap-and-drag behavior to achieve the same result. The thrust value starts at zero. The user taps and drags to increase the value. At that point they could let go to maintain it. Any tap-and-drag gesture value change would be the delta from the original value.

Rotation

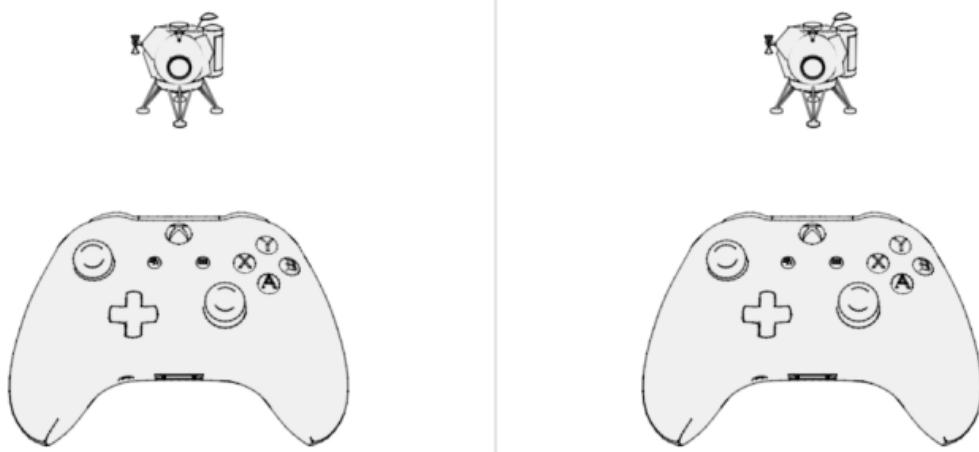
This is a little more tricky. Having holographic "rotate" buttons to tap makes for a terrible experience. There isn't a physical control to leverage, so the behavior must come from manipulation of an object representing the lander, or with the lander itself. We came up with a method using tap-and-drag which enables a user to effectively "push and pull" it in the direction they want it to face. Any time a user taps and holds, the point in space where the gesture was initiated becomes the origin for rotation. Dragging from the origin converts the delta of the hand's translation (X,Y,Z) and applies it to the delta of the lander's rotation values. Or more simply, *dragging left <-> right, up <-> down, forward <-> back in spaces rotates the ship accordingly*.

Since the HoloLens can track two hands, rotation can be assigned to the right hand while thrust is controlled by the left. Finesse is the driving factor for success in this game. The *feel* of these interactions is the absolute highest priority. Especially in context of tactile immersion. A ship that reacts too quickly would be unnecessarily difficult to steer, while one too slow would require the user to "push and pull" on the ship for an awkwardly long amount of time.

Adding input for game controllers

While hand gestures on the HoloLens provide a novel method of fine-grain control, there is still a certain lack of 'true' tactile feedback that you get from analog controls. Connecting an Xbox game controller allows to bring back this sense of physicality while leveraging the control sticks to retain fine-grain control.

There are multiple ways to apply the relatively straight-forward control scheme to the Xbox controller. Since we're trying to stay as close to the original arcade set up as possible, **Thrust** maps best to the trigger button. These buttons are analog controls, meaning they have more than simple *on and off* states, they actually respond to the degree of pressure put on them. This gives us a similar construct as the **thrust lever**. Unlike the original game and the hand gesture, this control will cut the ship's thrust once a user stops putting pressure on the trigger. It still gives the user the same degree of finesse as the original arcade game did.



Left thumbstick is mapped to yaw and roll; right thumbstick is mapped to pitch and roll

The dual thumbsticks naturally lend themselves to controlling ship rotation. Unfortunately, there are 3 axes on which the ship can rotate and two thumbsticks which both support two axes. This mismatch means either one thumbstick controls one axis; or there is overlap of axes for the thumbsticks. The former solution ended up feeling "broken" since thumbsticks inherently blend their local X and Y values. The latter solution required some testing to find which redundant axes feel the most natural. The final sample uses *yaw* and *roll* (Y and X axes) for the left thumbstick, and *pitch* and *roll* (Z and X axes) for the right thumbstick. This felt the most natural as *roll* seems to independently pair well with *yaw* and *pitch*. As a side note, using both thumbsticks for *roll* also happens to double the rotation value; it's pretty fun to have the lander do loops.

This sample app demonstrates how spatial recognition and tactile immersion can significantly change an experience thanks to Windows Mixed Reality's extensible input modalities. While Lunar Lander may be nearing 40 years in age, the concepts exposed with that little octagon-with-legs will live on forever. When imagining the future, why not look at the past?

Technical details

You can find scripts and prefabs for the Lunar Module sample app on the [Mixed Reality Design Labs GitHub](#).

About the author



Addison Linville
UX Designer @Microsoft

See also

- [Motion controllers](#)
- [Gestures](#)
- [Types of mixed reality apps](#)

Case study - AfterNow's process - envisioning, prototyping, building

11/6/2018 • 6 minutes to read • [Edit Online](#)

At [AfterNow](#), we work with you to turn your ideas and aspirations into concrete, fully operational products and experiences ready for the market. But before we write a single line of code, we create a blueprint through a process called envisioning.



What is Envisioning?

Envisioning is an ideation process used for product design. We streamline a large pool of ideas into one -- or a few -- calculated, realistic concepts to execute.

This process will bring to light many problems and blockers that would have eventually come up in the product. Failing early and fast ultimately saves a lot of time and money.

Throughout this process, we'll provide you with notes on the group's ideas and findings. When we're done, you'll receive a document of a script and a video of our "act it out" step, a simulation of the proposed product in action. These are concrete blueprints to your product -- a framework you can keep. Ideally, the next step is for us to build the product, which lets us test and experience the real deal.

Here's a daily play-by-play of the process, how long each step takes, and who you'll need on your team.



Envisioning participants

It's important to select the right participants for this process. The number can vary, but we recommend keeping it under seven people for the sake of efficiency.

Decider (critical)- The Decider is an important person to have in the group. He or she is the one who will be making the final decision or tipping the scale if there is an indecision during the process (typically the CEO, founder, product manager, or head of design). If they aren't able to participate for the entire process, appoint someone most fit to take on the role.

Finance (optional)- Someone who can explain how the project is funded.

Marketing (optional)- Someone who crafts the company's messages.

Customer (important)- Someone who talks to customers frequently and can be their advocate.

Tech/logistics (important)- Someone who understands what the company can build and deliver.

Design (important)- Someone who designs the product your company makes.

DAY 1 - Map

List goals & constraints; set a long-term goal - 5 min

If we were being completely optimistic, what is our long-term goal? Why are we building this solution in mixed reality and what problems are we trying to solve?

List sprint questions - 5 min

Get a bit pragmatic and list out fears in question format. How could we fail? To meet the long-term goal, what has to be true? If the project failed, what may have caused that?

Make a flow map - Up to 1 hour

This is a rough, functional map (not detailed). We write all of the user types on the left and the endings on the right. Then we connect them in the middle with words and arrows showing how users interact with the product. This is done in 5-15 steps.

Interview experts & take notes

The goal of this exercise is to enhance our collective understanding of the problem. Here we can add things to the map and fix any mistakes on it. If part of the flow can be expanded in more detail, we talk to the expert.

Our best practice for this process happens when we take individual notes in How might we (HMW) format. HMW is a technique developed by P&G in the 70s. It's a method of note taking in the form of a question which ultimately results in organized and prioritized notes. We will jot down one idea per sticky note which starts with the phrase, "How might we." For example, how might we communicate the values of [your company] in mixed reality?



Organize and decide

Organize HMW notes

We take all the "How might we" notes and stick them onto a wall. We will then categorize them into noticeable themes as they emerge.

Vote on HMW notes

Once all the notes are in their appropriate categories, we will vote on our favorite questions. Everyone gets two votes; the Decider gets four votes.

Pick a target

Circle the most important customer and one target moment on the map and look over the initial questions for the sprint. The Decider has the task of making the call.

DAY 2 - Sketch

Existing product demos

Everyone in the sprint group will research existing products they like that solve similar problems to the ones we're tackling. We'll then demo our findings to the rest of the group. This gives us a good idea of the features we find interesting.

Notes

We spend 20 minutes silently walking around the room to gather notes.

Ideas

We spend 20 minutes privately jotting down some rough ideas.

Crazy 8s

We spend eight minutes creating eight frames. This is where we rapidly sketch a variation of one of our best ideas in each frame.

Solution

We spend 30 minutes sketching storyboards. We take the best variations of ideas from Crazy 8s and create a three-panel storyboard of the idea.



Sketching storyboards

DAY 3 - Decide

This day involves a lot of critiquing and voting. Here's the breakdown:

Quiet critiquing

We tape solutions onto the wall and take time to look at them silently. After everyone has gotten time to look at all the sketches, we will collectively create a heat map. Each person puts marks on the sketches they like. This can be the entire flow or an area of a flow.

Discussion & explanations

We will begin to see a heat map of ideas people find interesting, and spend three minutes on each sketch that got the most attention. People can openly discuss why they like certain sketches and the artist can correctly convey their ideas when there is a misunderstanding. This can surface new ideas at times.

In this step we also want to give people opportunities to explain ideas that didn't get a lot of attention. Sometimes you'll find a gem in an idea people didn't really understand previously.

Straw poll & supervote

Each person will silently choose their favorite idea and, when ready, people will place a mark on their favorite one

all at once. The Decider will then get three large dots for the idea he/she liked the most. The ones marked by the Decider are the ideas that are going to be prototyped.

DAY 4 - Prototype

Preparation

In preparation for the prototyping phase, we will gather an assortment of physical tools -- much like a child's play kit -- to construct 3D objects that will represent holograms. We will also have physical tools that represent the hardware devices people are using to access these holograms.

Prototype!

Prepare a script

Script out the entire user flow for this prototype and decide on the actors that will be playing a part in the skit.

Act it out

Physically act out the user flow which involves all the user parties in the order they will be using it. This gives us a good idea of how viable this product will be when we build it out. It will also bring to light some major kinks we may have overlooked when coming up with the idea.

About the author



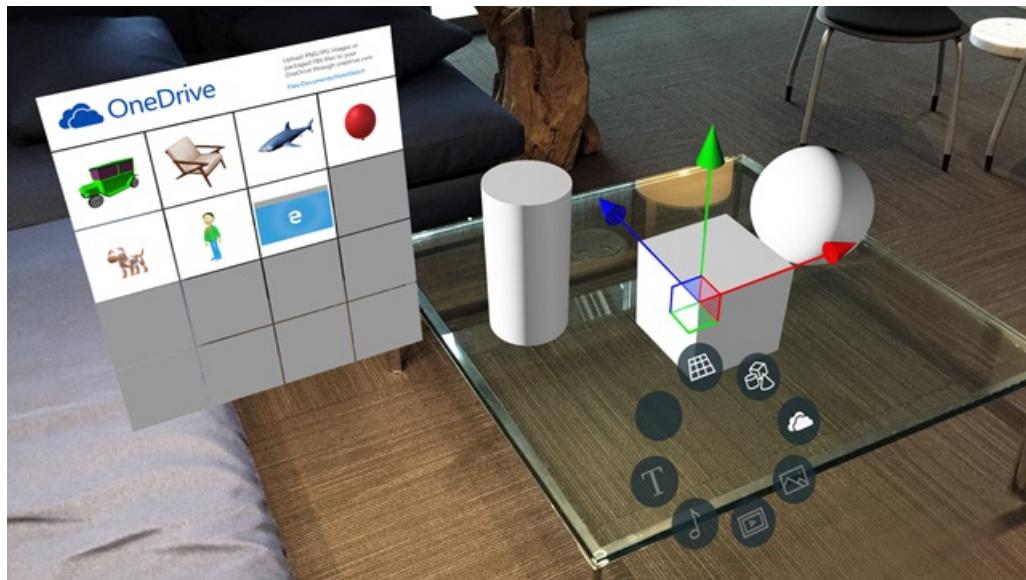
Rafai Eddy

AfterNow

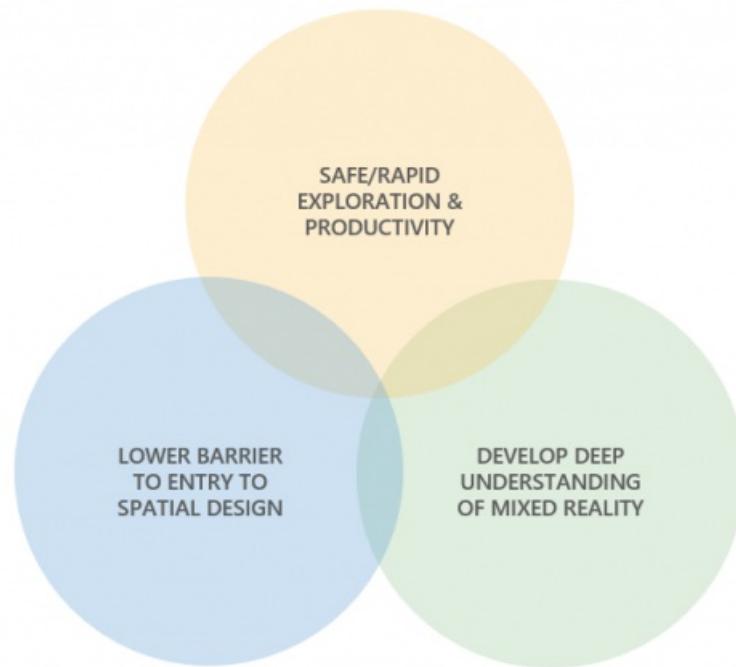
Case study - Building HoloSketch, a spatial layout and UX sketching app for HoloLens

11/6/2018 • 4 minutes to read • [Edit Online](#)

HoloSketch is an on-device spatial layout and UX sketching tool for HoloLens to help build holographic experiences. HoloSketch works with a paired Bluetooth keyboard and mouse as well as gesture and voice commands. The purpose of HoloSketch is to provide a simple UX layout tool for quick visualization and iteration.



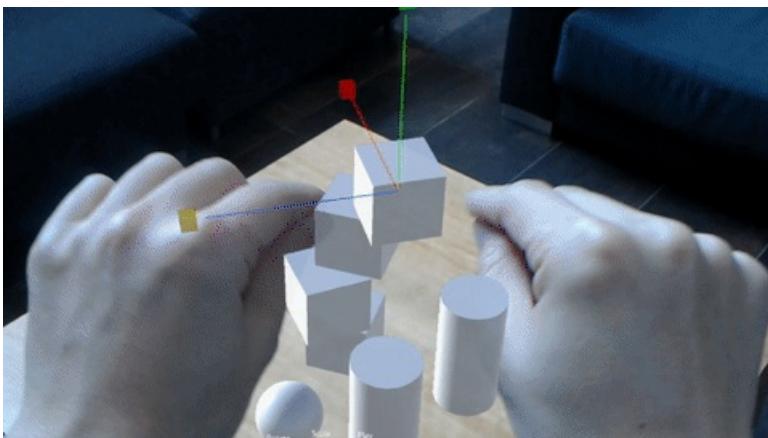
HoloSketch: spatial layout and UX sketching app for HoloLens



A simple UX layout tool for quick visualization and iteration

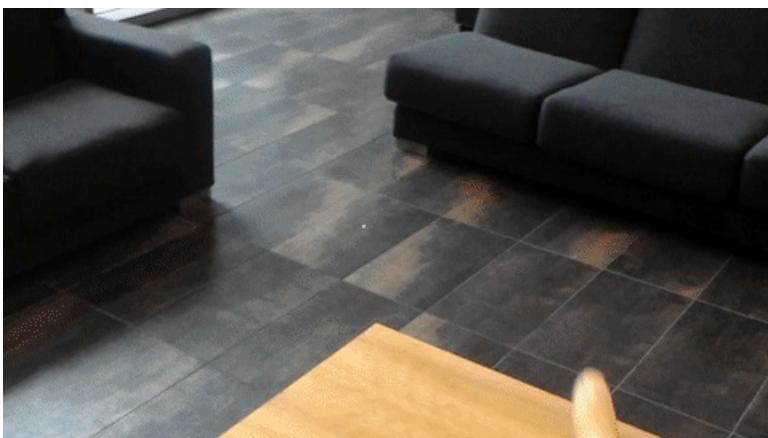
Features

Primitives for quick studies and scale-prototyping



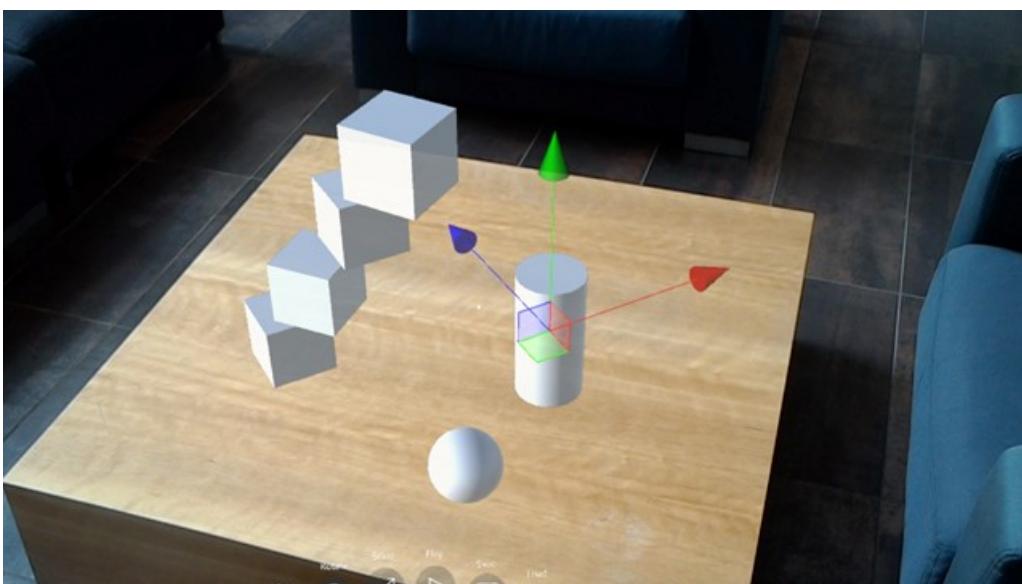
Use primitive shapes for quick massing studies and scale-prototyping.

Import objects through OneDrive



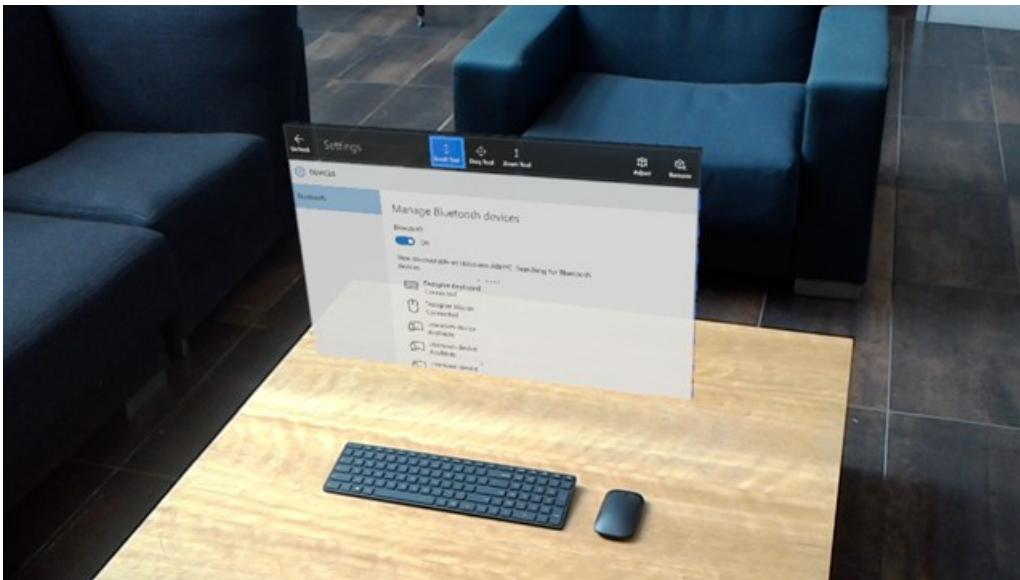
Import PNG/JPG images or 3D FBX objects (requires packaging in Unity) into the mixed reality space through OneDrive.

Manipulate objects



Manipulate objects (move/rotate/scale) with a familiar 3D object interface.

Bluetooth, mouse/keyboard, gestures and voice commands



HoloSketch supports Bluetooth mouse/keyboard, gestures and voice commands.

Background

Importance of experiencing your design in the device

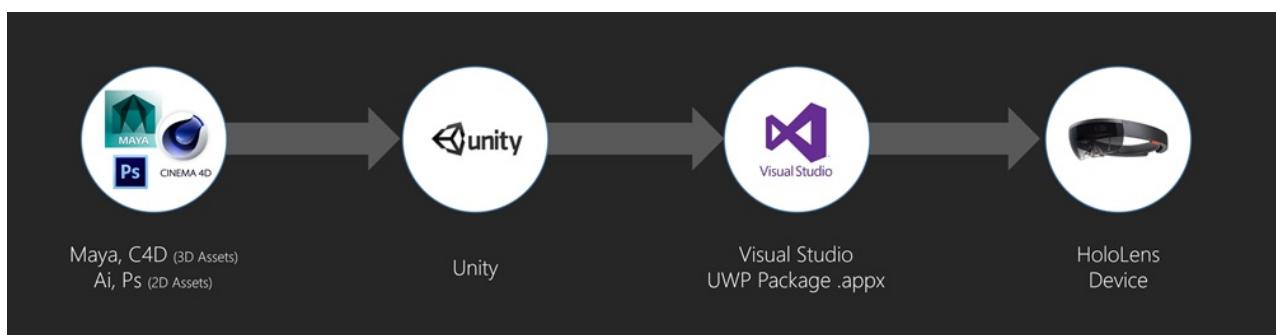
When you design something for HoloLens, it is important to experience your design in the device. One of the biggest challenges in mixed reality app design is that it is hard to get the sense of scale, position and depth, especially through traditional 2D sketches.

Cost of 2D based communication

To effectively communicate UX flows and scenarios to others, a designer may end up spending a lot of time creating assets using traditional 2D tools such as Illustrator, Photoshop and PowerPoint. These 2D designs often require additional effort to convert them into 3D. Some ideas are lost in this translation from 2D to 3D.

Complex deployment process

Since mixed reality is a new canvas for us, it involves a lot of design iteration and trial and error by its nature. For designer who are not familiar with tools such as Unity and Visual Studio, it is not easy to put something together in HoloLens. Typically you have to go through the process below to see your 2D/3D artwork in the device. This was a big barrier for designers iterating on ideas and scenarios quickly.



Deployment process

Simplified process with HoloSketch

With HoloSketch, we wanted to simplify this process without involving development tools and device portal pairing. Using OneDrive, users can easily put 2D/3D assets into HoloLens.



Simplified process with HoloSketch

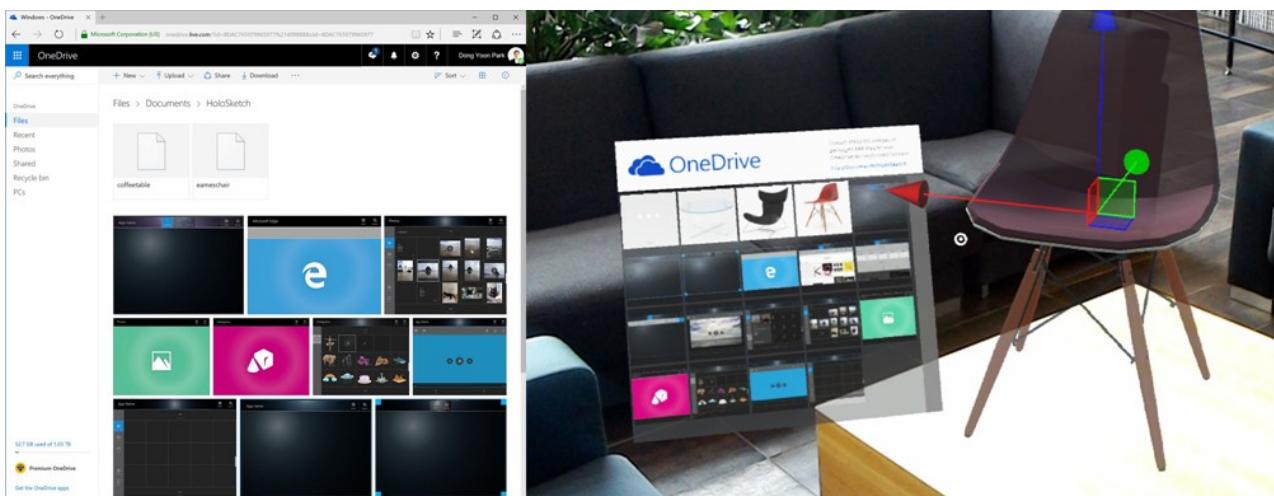
Encouraging three-dimensional design thinking and solutions

We thought this tool would give designers an opportunity to explore solutions in a truly three-dimensional space and not be stuck in 2D. They don't have to think about creating a 3D background for their UI since the background is the real world in the case of Hololens. HoloSketch opens up a way for designers to easily explore 3D design on Hololens.

Get Started

How to import 2D images (JPG/PNG) into HoloSketch

- Upload JPG/PNG images to your OneDrive folder 'Documents/HoloSketch'.
- From the OneDrive menu in HoloSketch, you will be able to select and place the image in the environment.



Importing images and 3D objects through OneDrive

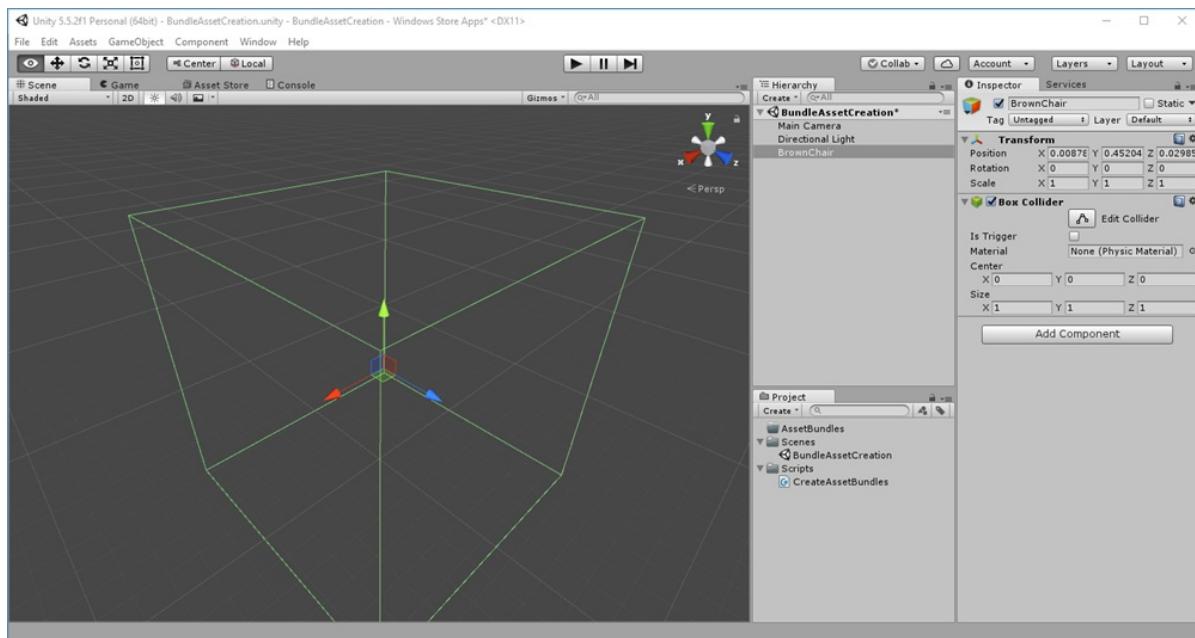
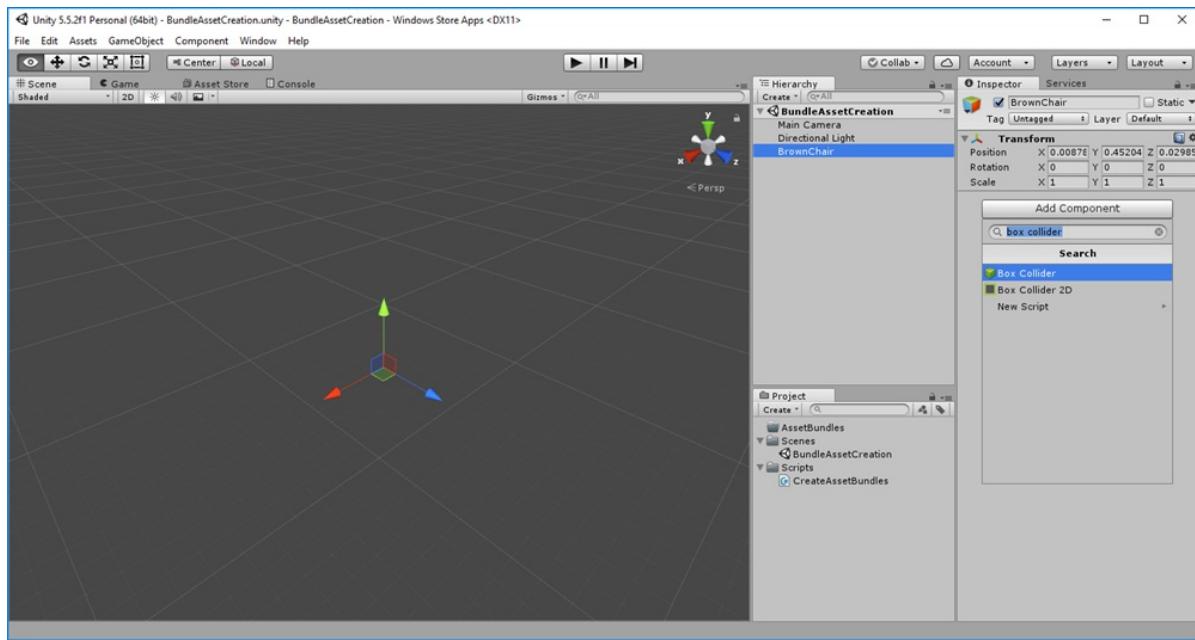
How to import 3D objects into HoloSketch

Before uploading to your OneDrive folder, please follow the steps below to package your 3D objects into a Unity asset bundle. Using this process you can import your FBX/OBJ files from 3D software such as Maya, Cinema 4D and Microsoft Paint 3D.

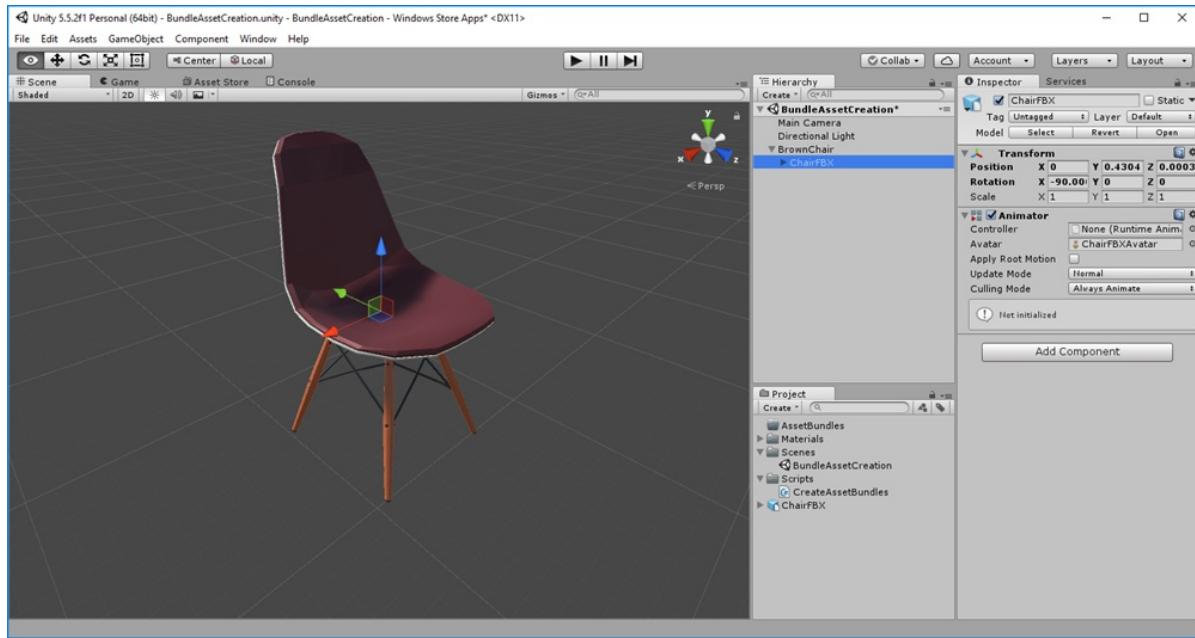
IMPORTANT

Currently, asset bundle creation is supported with Unity version 5.4.5f1.

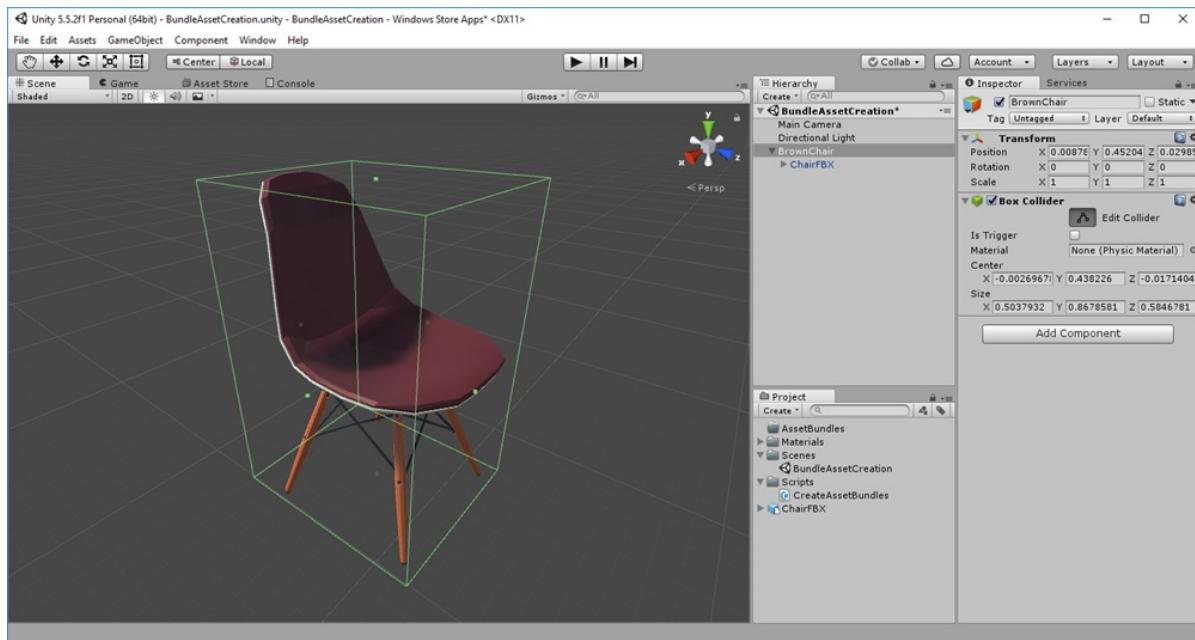
1. Download and open the Unity project '[AssetBundler_Unity](#)'. This Unity project includes the script for the bundle asset generation.
2. Create a new GameObject.
3. Name the GameObject based on the contents.
4. In the Inspector panel, click 'Add Component' and add 'Box Collider'.



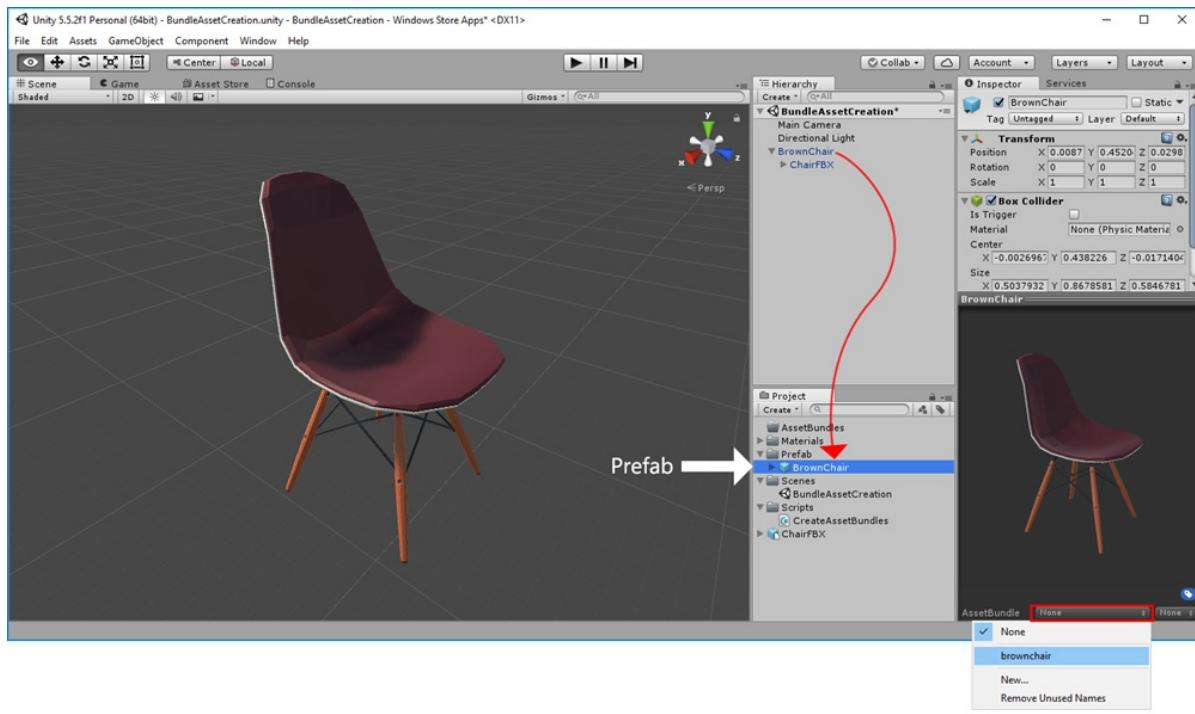
5. Import the 3D FBX file by dragging it into the project panel.
6. Drag the object into the Hierarchy panel **under your new GameObject**.



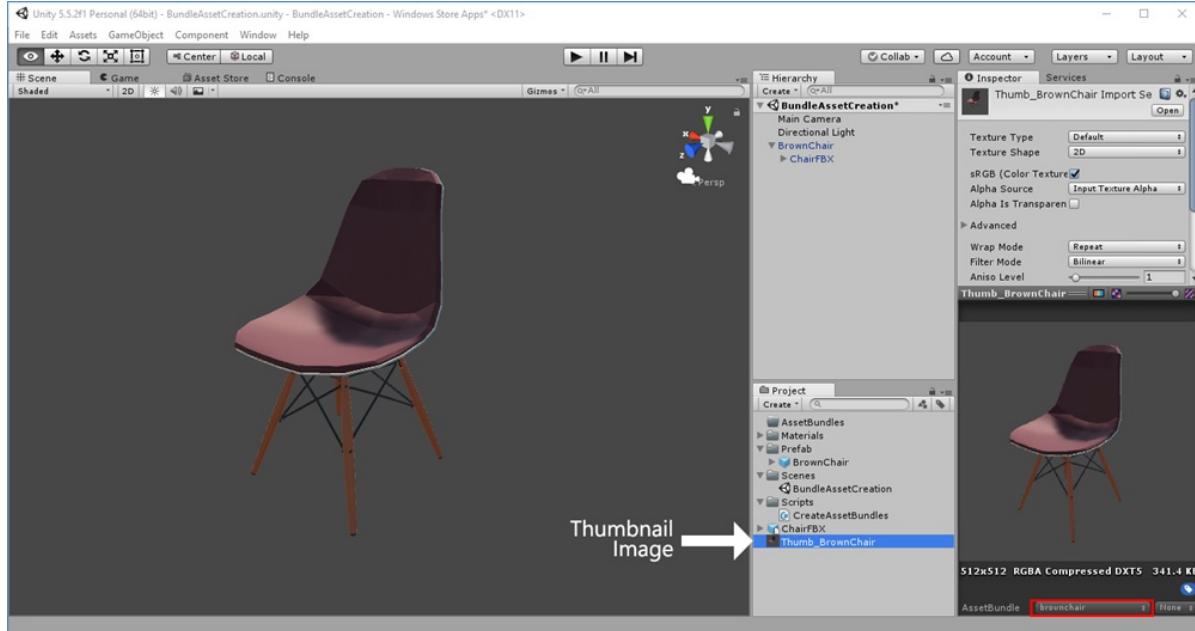
7. Adjust the collider dimension if it does not match the object. Rotate the object to face **Z-axis**.



8. Drag the object from the Hierarchy panel to the Project panel to **make it prefab**.
9. On the bottom of the inspector panel, click the dropdown, create and assign a new unique name. Below example shows adding and assigning 'brownchair' for the AssetBundle Name.

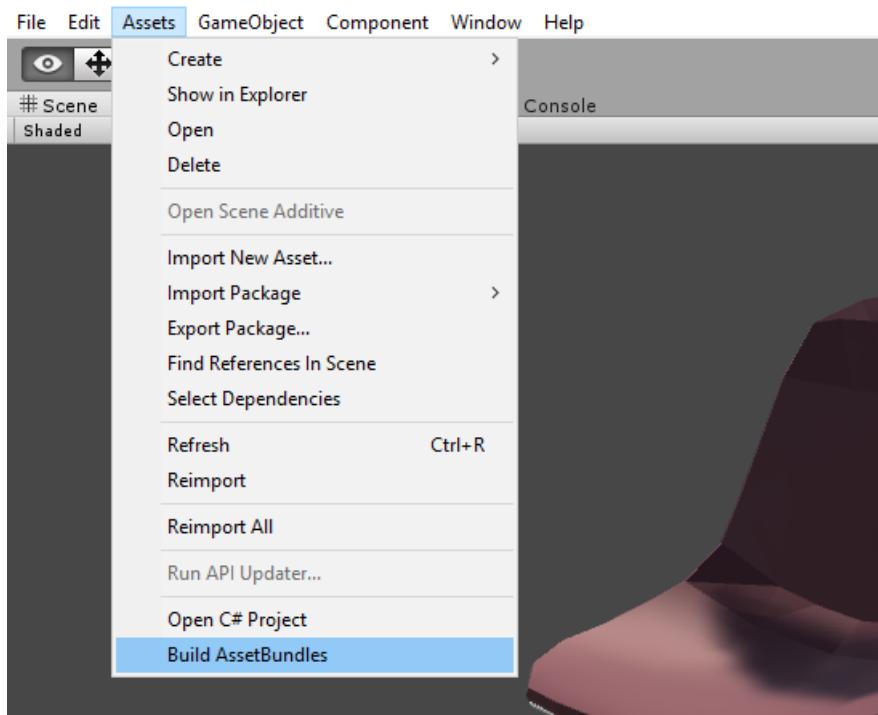


10. Prepare a thumbnail image for the model object.

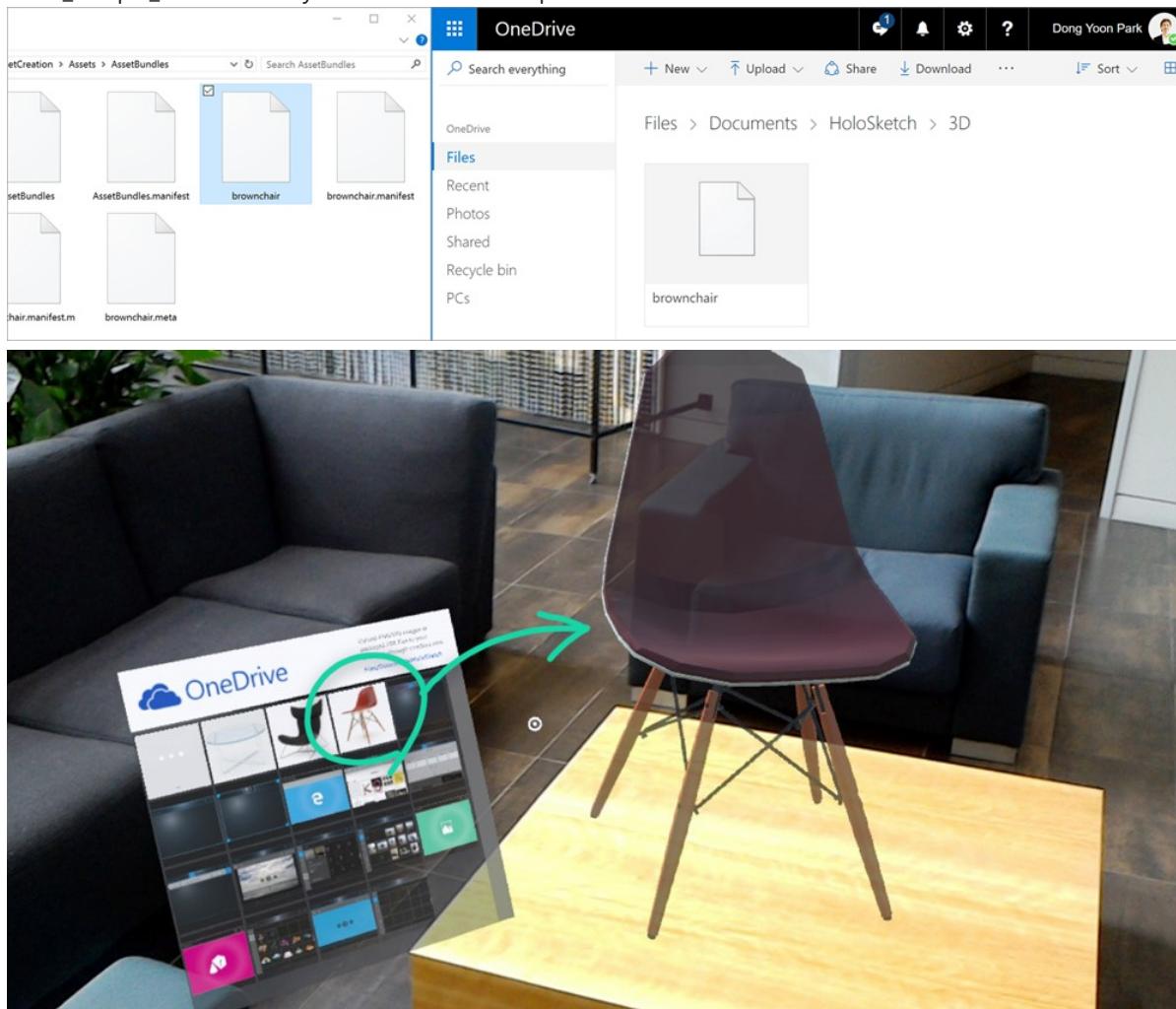


11. Create a folder named 'Assetbundles' under the Unity project's 'Asset' folder.

12. From the Assets menu, select 'Build AssetBundles' to generate file.



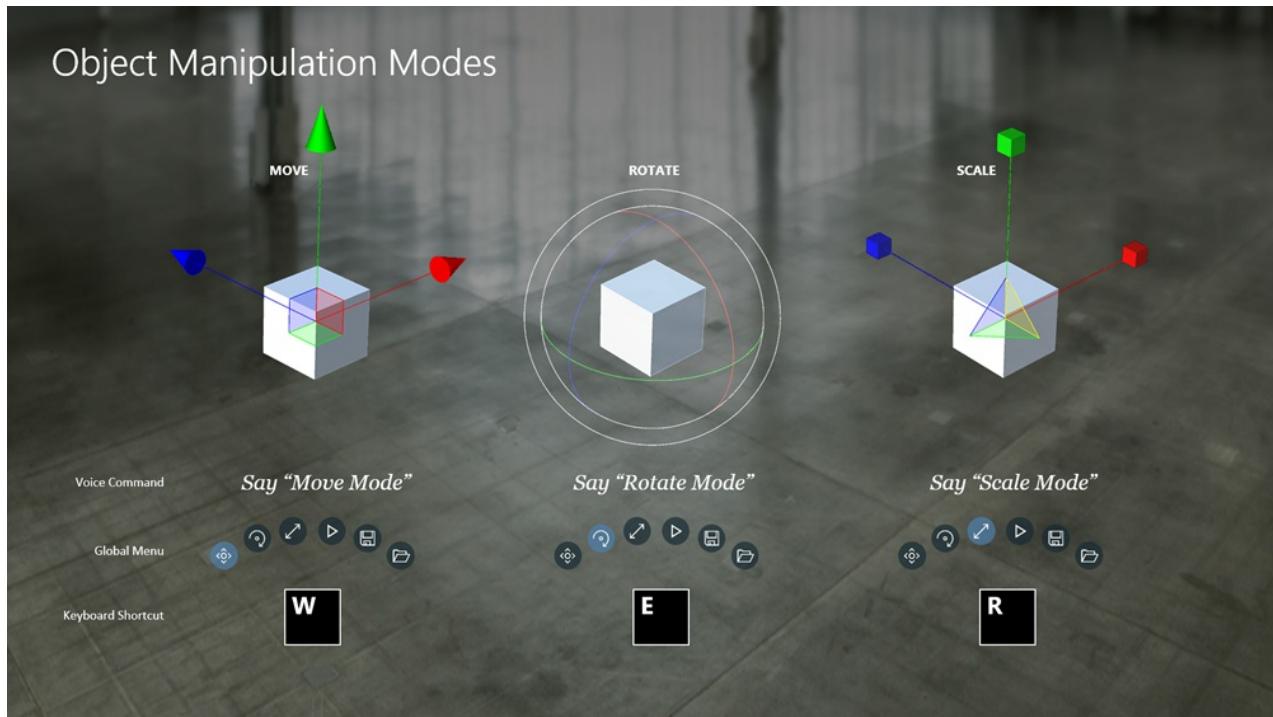
13. **Upload the generated file to the /Files/Documents/HoloSketch folder on OneDrive.** Upload the asset_unique_name file only. You don't need to upload manifest files or AssetBundles file.



How to manipulate the objects

HoloSketch supports the traditional interface that is widely used in 3D software. You can use move, rotate, scale the objects with gestures and a mouse. You can quickly switch between different modes with voice or keyboard.

Object manipulation modes



How to manipulate the objects

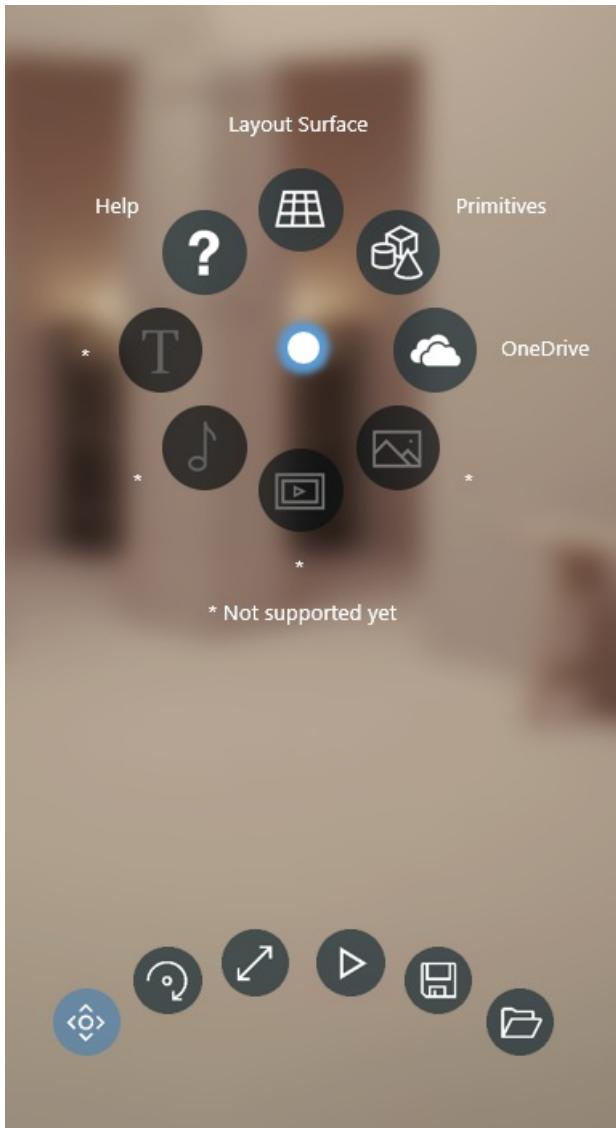
Contextual and Tool Belt menus

Using the Contextual Menu

Double air tap to open the Contextual Menu.

Menu items:

- **Layout Surface:** This is a 3D grid system where you can layout multiple objects and manage them as a group.
Double air-tap on the Layout Surface to add objects to it.
- **Primitives:** Use cubes, spheres, cylinders and cones for massing studies.
- **OneDrive:** Open the OneDrive menu to import objects.
- **Help:** Displays help screen.



Contextual menu

Using the Tool Belt Menu

Move, Rotate, Scale, Save, and Load Scene are available from the Tool Belt Menu.

Using keyboard, gestures and voice commands

⌨️ Keyboard shortcuts

W	Move Mode
E	Rotate Mode
R	Scale Mode
H	Show Help
J	Hide Help

CTRL + C	Copy
CTRL + V	Paste
CTRL + U	Undo
CTRL + Y	Redo
DEL	Delete Object

🎙️ Voice commands

“move mode” “rotate mode” “scale mode”

↗️ Gestures

Double tap to open the main menu

In scale mode use two hands for uniform scaling

“delete object”

“turn to me”

“snap to surface”

“copy object”
“paste object”

“add cube”

“add sphere”

“add plane”

“snap to grid”
“snap to object”

Keyboard, gestures, and voice commands

Download the app



[Download and install the HoloSketch app for free from the Microsoft Store](#)

Known issues

- Currently asset bundle creation is supported with **Unity version 5.4.5f1**.
- Depending on the amount of data in your OneDrive, the app might appear as if it has stopped while loading OneDrive contents
- Currently, Save and Load feature only supports primitive objects
- Text, Sound, Video and Photo menus are disabled on the contextual menu
- The Play button on the Tool Belt menu clears the manipulation gizmos

Sharing your sketches

You can use the video recording feature in HoloLens by saying 'Hey Cortana, Start/Stop recording'. Press the volume up/down key together to take a picture of your sketch.

About the authors



Dong Yoon Park

UX Designer @Microsoft



Patrick Sebring

Developer @Microsoft

Case study - Capturing and creating content for HoloTour

11/6/2018 • 12 minutes to read • [Edit Online](#)

HoloTour for Microsoft HoloLens provides immersive 3D personal tours of iconic locations around the world. As the designers, artists, producers, audio designers, and developers working on this project found out, creating a convincingly real 3D rendering of a well-known location takes a unique blend of creative and technological wizardry.

The tech

With HoloTour, we wanted to make it possible for people to visit some of the world's most amazing destinations—like the [ruins of Machu Picchu](#) in Peru or the modern day [Piazza Navona](#) in Italy—right from their own living rooms. Our team made it the goal of HoloTour to "make you feel like you're really there." The experience needed to be more than just pictures or videos. By leveraging the unique display, tracking, and audio technology of HoloLens we believed we could virtually transport you to another place. We would need to capture the sights, sounds, and three-dimensional geometry of every location we visited and then recreate that within our app.

In order to do this, we needed a 360° camera rig with directional audio capture. It needed to capture at extremely high resolution, so that the footage would look crisp when played back on a HoloLens and the cameras would need to be positioned close together in order to minimize stitching artifacts. We wanted full spherical coverage, not just along the horizon, but above and below you as well. The rig also needed to be portable so that we could take it all over the world. We evaluated available off-the-shelf options and realized they simply weren't good enough to realize our vision – either because of resolution, cost, or size. If we couldn't find a camera rig that satisfied our needs, we would have to build one ourselves.

Building the rig

The first version—made from cardboard, Velcro, duct tape, and 14 GoPro cameras—was something MacGyver would have been proud of. After reviewing everything from low-end solutions to custom manufactured rigs, GoPro cameras were ultimately the best option for us because they were small, affordable, and had easy-to-use memory storage. The small form factor was especially important because it allowed us to place cameras fairly close together—the smaller the distance between cameras, the smaller the stitching artifacts will be. Our unique camera arrangement allowed us to get full sphere coverage *plus* enough overlap to intelligently align cameras and smooth out some artifacts during the stitching process.

Taking advantage of the [spatial sound](#) capabilities on HoloLens is critical to creating a convincingly real immersive experience. We used a four-microphone array situated beneath the cameras on the tripod, which would capture sound from the location of our camera in four directions, giving us enough information to create spatial sounds in our scenes.



Our 360° camera rig set up for filming outside the Pantheon.

We tested out our homemade rig by taking it up to Rattlesnake Ridge near Seattle, capturing the scenery at the top of the hike. The result, though significantly less polished than the locations you see in HoloTour today, gave us confidence that our rig design was good enough to make you feel like you're really there.

We upgraded our rig from Velcro and cardboard to a 3D-printed camera housing and bought external battery packs for the GoPro cameras to simplify battery management. We then did a more extensive test, traveling down to San Francisco to create a miniature tour of the city's coast and the iconic Golden Gate bridge. This camera rig is what we used for most of our captures of the locations you visit in HoloTour.



The 360° camera rig filming in Machu Picchu.

Behind the scenes

Before filming, we needed to figure out which locations we wanted to include on our virtual tour. Rome was the first location we intended to ship and we wanted to get it right, so we decided to do a scouting trip in advance. We sent a team of six people—including artists, designers, and producers—for an in-person visit to the sites we were considering. The trip took about 9 days – 2.5 for travel, the rest for filming. (For Machu Picchu we opted not to do a scout trip, researching in advance and booking a few days of buffer for filming.)

While in Rome, the team took photos of each area and noted interesting facts as well as practical considerations, such as how hard it is to travel to each spot and how difficult it would be to film because of crowds or restrictions. This may sound like a vacation, but it's a lot of work. Days started early in the morning and would go non-stop until evening. Each night, footage was uploaded for the team back in Seattle to review.



Our capture crew in Rome.

After the scout trip was completed, a final plan was made for actual filming. This required a detailed list of where we were going to film, on what day, and at what time. Every day overseas is expensive, so these trips needed to be efficient. We booked guides and handlers in Rome to help us and fully used every day from before sunrise to after sunset. We need to get the best footage possible in order to make you feel like you're really there.

Capturing the video

Doing a few simple things during capture can make post-processing much easier. For example, whenever you stitch together images from multiple cameras, you end up with visual artifacts because each camera has a slightly different view. The closer objects are to the camera, the larger the difference between the views, and the larger the stitching artifacts will be. Here's an easy way to visualize the problem: hold your thumb up in front of your face and look at it with only one eye. Now switch eyes. You'll see that your thumb appears to move relative to the background. If you hold your thumb further away from your face

and repeat the experiment, your thumb will appear to move less. That apparent movement is similar to the stitching problem we faced: Your eyes, like our cameras, don't see the exact same image because they are separated by a little distance.

Because it's much easier to prevent the worst artifacts while filming than it is to correct them in post-processing, we tried to keep people and things far away from the camera in the hopes we could eliminate the need to stitch close-up objects.

Maintaining a large clearing around our camera was probably one of the biggest challenges we had during shooting and we had to get creative to make it work. Working with local guides was a huge help in managing crowds, but we also found that using signs—and sometimes small cones or bean bags—to mark our filming space was reasonably effective, especially since we only needed to get a short amount of footage at each location. Often the best way to get a good capture was to just to arrive very early in the morning, before most people showed up.

Some other useful capture techniques come straight from traditional film practices. For example, we used a color correction card on all of our cameras and captured reference photos of textures and objects we might need later.



A rough cut of Pantheon footage before stitching.

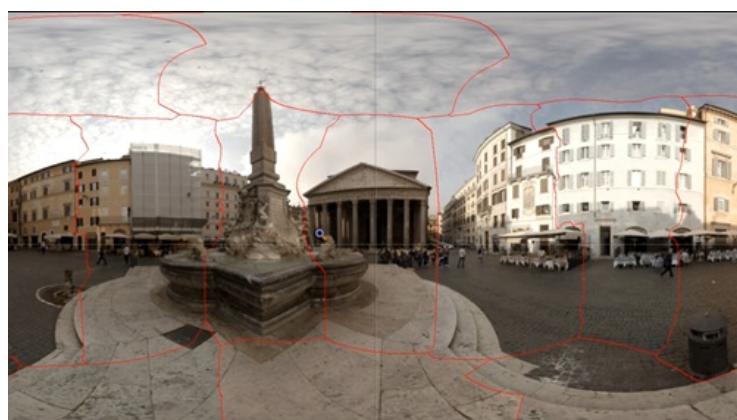
Processing the video

Capturing 360° content is only the first step—a lot of processing is needed to convert the raw camera footage we captured into the final assets you see in HoloTour. Once we were back home we needed to take the video from 14 different camera feeds and turn it into a single continuous video with minimal artifacts. Our art team used a number of tools to combine and polish the captured footage and we developed a pipeline to optimize the processing as much as possible. The footage had to be stitched together, color corrected, and then composited to remove distracting elements and artifacts or to add supplemental pockets of life and motion, all with the goal to enhance that feeling of actually being there.



A rough cut of Pantheon footage before stitching.

To stitch the videos together, we used a tool called [PTGui](#) and integrated it into our processing pipeline. As part of post-processing we extracted still frames from our videos and found a stitching pattern that looked good for one of those frames. We then applied that pattern to a custom plugin we wrote that allowed our video artists to fine tune and tweak the stitching pattern directly while compositing in After Effects.



Screenshot of PTGui showing the stitched Pantheon footage.

Video playback

After processing of the footage is completed, we have a seamless video but it's extraordinarily large—around 8K resolution. Decoding video is expensive and there are very few computers that can handle an 8K video so the next challenge was finding a way to play this video back on HoloLens. We developed a number of strategies to avoid the cost of decoding while still making the user feel like they were viewing the entire video.

The easiest optimization is to avoid decoding parts of the video that don't change much. We wrote a tool to identify areas in each scene that have little or no motion. For those regions we show a static image rather than decoding a video each frame. To make this possible, we divided up the massive video into much smaller chunks.

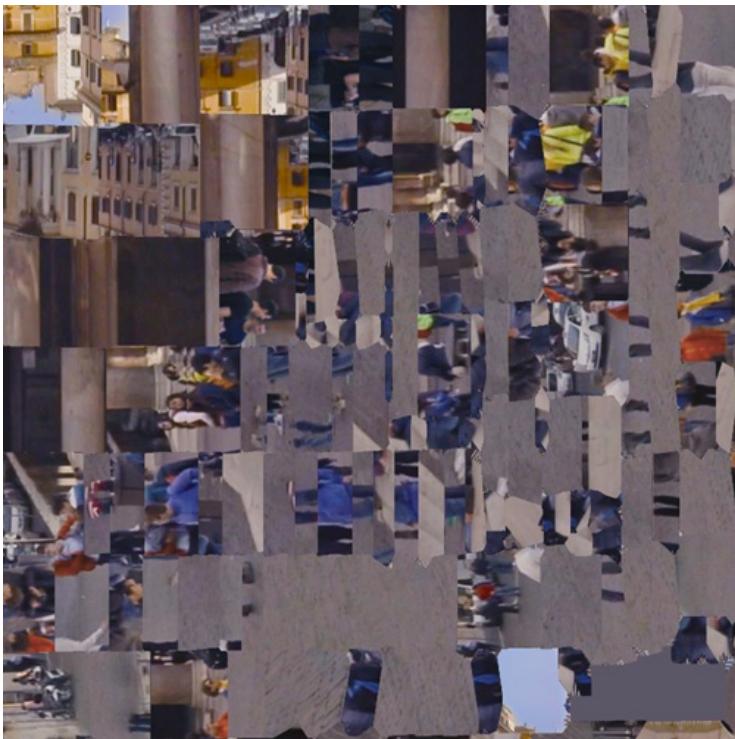
We also made sure that every pixel we decoded was used most effectively. We experimented with compression techniques to lower the size of the video; we split the video regions according to the polygons of the geometry it would be projected onto; we adjusted UVs and repacked the videos based on how much detail different polygons included. The result of this work is that what started as a single 8k video turned into many chunks that look almost unintelligible until they are properly re-projected into the scene. For a game developer who understands texture mapping and UV packing, this will probably look familiar.



A full view of the Pantheon before optimizations.



The right half of the Pantheon, processed for video playback.



Example of a single video region after optimization and packing.

Another trick we used was to avoid decoding video you aren't actively viewing. While in HoloTour, you can only see part of the full scene at any given moment. We only decode videos within or shortly outside of your field of view (FOV). As you rotate your head, we start playing the regions of the video that are now in your FOV and stop playing ones that are no longer within it. Most people won't even notice this is happening, but if you turn around rapidly, you'll see the video takes a second to start—in the meantime you'll see a static image which then fades back to video once it's ready.

To make this strategy work we developed an extensive video playback system. We optimized the low level playback code in order to make video switching extremely efficient. Additionally, we had to encode our videos in a special way to make it possible to rapidly switch to any video at any time. This playback pipeline took a significant amount of development time so we implemented it in stages. We started with a simpler system that was less efficient, but allowed designers and artists to work on the experience, and slowly improved it to a more robust playback system that allowed us to ship at the final quality bar. This final system had custom tools we created within Unity to set up the video within the scene and monitor the playback engine.

Recreating near-space objects in 3D

Videos make up the majority of what you see in HoloTour, but there are a number of 3D objects that appear close to you, such as the painting in Piazza Navona, the fountain outside the Pantheon, or the hot air balloon you stand in for aerial scenes. These 3D objects are important because human depth perception is very good up close, but not very good far away. We can get away with video in the distance, but to allow users to walk around their space and feel like they're really there, nearby objects need depth. This technique is similar to the sort of thing you might see in a natural history museum—picture a diorama that has physical landscaping, plants, and animal specimens in the foreground, but recedes into a cleverly masked matte painting in the background.

Some objects are simply 3D assets we created and added to the scene to enhance the experience. The painting and the hot air balloon fall into this category because they weren't present when we filmed. Similar to game assets, they were created by a 3D artist on our team and textured appropriately. We place these in our scenes near where you stand, and the game engine can render them to the two HoloLens displays so that they appear as a 3D object.

Other assets, like the fountain outside the Pantheon, are real objects that exist in the locations where we're shooting video, but to bring these objects out of the video and into 3D, we have to do a number of things.

First, we need additional information about each object. While on location for filming, our team captured a lot of reference footage of these objects so that we would have enough detailed images to accurately recreate the textures. The team also performed a [photogrammetry](#) scan, which constructs a 3D model from dozens of 2D images, giving us a rough model of the object at perfect scale.

As we process our footage, objects that will later be replaced with a 3D representation are removed from the video. The 3D

asset is based on the photogrammetry model but cleaned up and simplified by our artists. For some objects, we can use parts of the video—such as the water texture on the fountain—but most of the fountain is now a 3D object, which allows users to perceive depth and walk around it in a limited space in the experience. Having near-space objects like this greatly adds to the sense of realism and helps to ground the users in the virtual location.



Pantheon footage with the fountain removed. It will be replaced with a 3D asset.

Final thoughts

Obviously, there was more to creating this content than what we've discussed here. There are a few scenes—we like to call them "impossible perspectives"—including the hot air balloon ride and the gladiator fight in the Colosseum, which took a more creative approach. We'll address these in a future case study.

We hope that sharing solutions to some of the bigger challenges we had during production is helpful to other developers and that you're inspired to use some of these techniques to create your own immersive experiences for HoloLens. (And if you do, please make sure to share it with us on the [HoloLens App Development forum!](#))

About the authors



David Haley is a Senior Developer who learned more about camera rigs and video playback than he thought possible from working on HoloTour.



Danny Askew is a Video Artist who made sure your journey through Rome was as flawless as possible.



Jason Syltebo is an Audio Designer who made sure you could experience the soundscape of every destination you visit, even when you go back in time.



Travis Steiner is a Design Director who researched and scouted locations, created trip plans, and directed filming on site.

See also

- [Video: Microsoft HoloLens: HoloTour](#)

Case study - Creating a galaxy in mixed reality

11/6/2018 • 9 minutes to read • [Edit Online](#)

Before Microsoft HoloLens shipped, we asked our developer community what kind of app they'd like to see an experienced internal team build for the new device. More than 5000 ideas were shared, and after a 24-hour Twitter poll, the winner was an idea called [Galaxy Explorer](#).

Andy Zubits, the art lead on the project, and Karim Luccin, the team's graphics engineer, talk about the collaborative effort between art and engineering that led to the creation of an accurate, interactive representation of the Milky Way galaxy in Galaxy Explorer.

The Tech

[Our team](#) - made up of two designers, three developers, four artists, a producer, and one tester — had six weeks to build a fully functional app which would allow people to learn about and explore the vastness and beauty of our Milky Way Galaxy.

We wanted to take full advantage of the ability of HoloLens to render 3D objects directly in your living space, so we decided we wanted to create a realistic looking galaxy where people would be able to zoom in close and see individual stars, each on their own trajectories.

In the first week of development, we came up with a few goals for our representation of the Milky Way Galaxy: It needed to have depth, movement, and feel volumetric—full of stars that would help create the shape of the galaxy.

The problem with creating an animated galaxy that had billions of stars was that the sheer number of single elements that need updating would be too big per frame for HoloLens to animate using the CPU. Our solution involved a complex mix of art and science.

Behind the scenes

To allow people to explore individual stars, our first step was to figure out how many particles we could render at once.

Rendering particles

Current CPUs are great for processing serial tasks and up to a few parallel tasks at once (depending on how many cores they have), but GPUs are much more effective at processing thousands of operations in parallel. However, because they don't usually share the same memory as the CPU, exchanging data between CPU <> GPU can quickly become a bottleneck. Our solution was to make a galaxy on the GPU, and it had to live completely on the GPU.

We started stress tests with thousands of point particles in various patterns. This allowed us to get the galaxy on HoloLens to see what worked and what didn't.

Creating the position of the stars

One of our team members had already written the C# code that would generate stars at their initial position. The stars are on an ellipse and their position can be described by (**curveOffset**, **ellipseSize**, **elevation**) where **curveOffset** is the angle of the star along the ellipse, **ellipseSize** is the dimension of the ellipse along X and Z, and elevation the proper elevation of the star within the galaxy. Thus, we can create a buffer ([Unity's ComputeBuffer](#)) that would be initialized with each star attribute and send it on the GPU where it would live for the rest of the experience. To draw this buffer, we use [Unity's DrawProcedural](#) which allows running a shader (code on a GPU) on an arbitrary set of points without having an actual mesh that represents the galaxy:

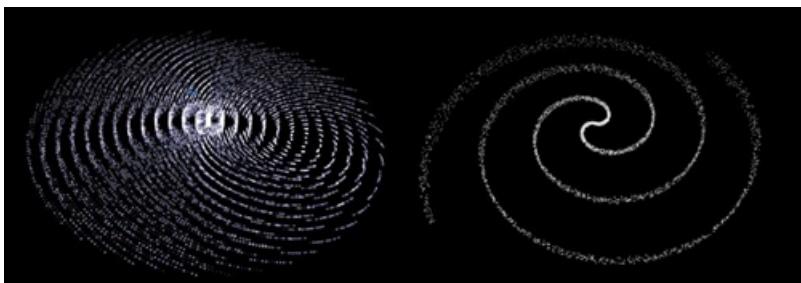
CPU:

```
GraphicsDrawProcedural(MeshTopology.Points, starCount, 1);
```

GPU:

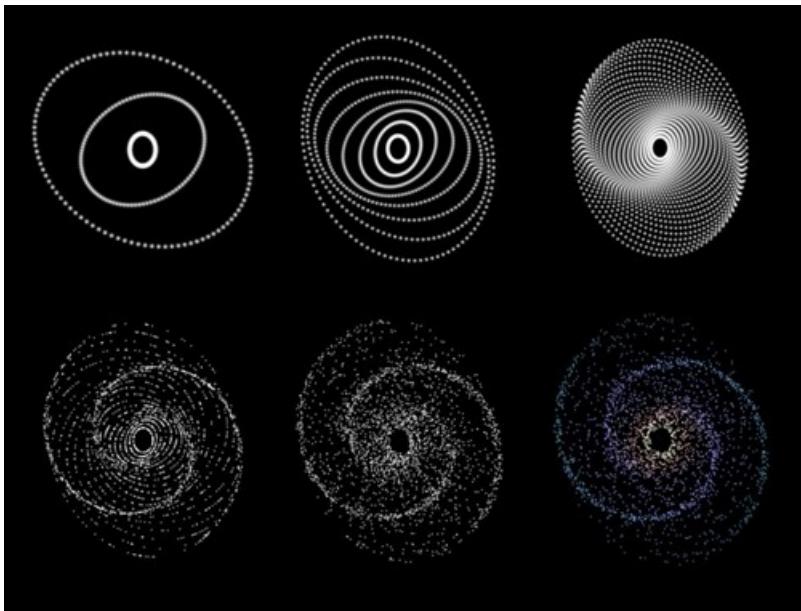
```
v2g vert (uint index : SV_VertexID)
{
    // _Stars is the buffer we created that contains the initial state of the system
    StarDescriptor star = _Stars[index];
    ...
}
```

We started with raw circular patterns with thousands of particles. This gave us the proof we needed that we could manage many particles AND run it at performant speeds, but we weren't satisfied with the overall shape of the galaxy. To improve the shape, we attempted various patterns and particle systems with rotation. These were initially promising because the number of particles and performance stayed consistent, but the shape broke down near the center and the stars were emitting outwardly which wasn't realistic. We needed an emission that would allow us to manipulate time and have the particles move realistically, looping ever closer to the center of the galaxy.



We attempted various patterns and particle systems that rotated, like these.

Our team did some research about the way galaxies function and we made a custom particle system specifically for the galaxy so that we could move the particles on ellipses based on "[density wave theory](#)," which theorizes that the arms of a galaxy are areas of higher density but in constant flux, like a traffic jam. It appears stable and solid, but the stars are actually moving in and out of the arms as they move along their respective ellipses. In our system, the particles never exist on the CPU—we generate the cards and orient them all on the GPU, so the whole system is simply initial state + time. It progressed like this:

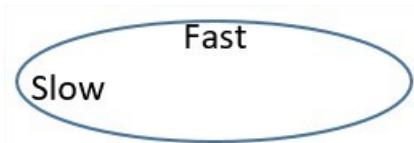


Progression of particle system with GPU rendering

Once enough ellipses are added and are set to rotate, the galaxies began to form “arms” where the movement of stars converge. The spacing of the stars along each elliptical path was given some randomness, and each star got a bit of positional randomness added. This created a much more natural looking distribution of star movement and arm shape. Finally, we added the ability to drive color based on distance from center.

Creating the motion of the stars

To animate the general star motion, we needed to add a constant angle for each frame and to get stars moving along their ellipses at a constant radial velocity. This is the primary reason for using **curveOffset**. This isn’t technically correct as stars will move faster along the long sides of the ellipses, but the general motion felt good.



Stars move faster on the long arc, slower on the edges.

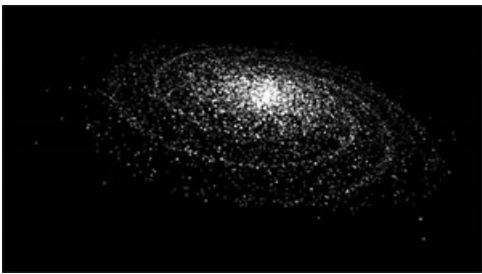
With that, each star is fully described by (**curveOffset**, **ellipseSize**, **elevation**, **Age**) where **Age** is an accumulation of the total time that has passed since the scene was loaded.

```
float3 ComputeStarPosition(StarDescriptor star)
{
    float curveOffset = star.curveOffset + Age;

    // this will be coded as a "sincos" on the hardware which will compute both sides
    float x = cos(curveOffset) * star.xRadii;
    float z = sin(curveOffset) * star.zRadii;

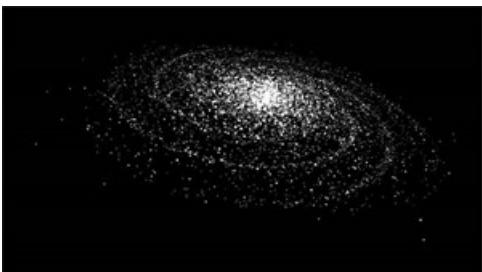
    return float3(x, star.elevation, z);
}
```

This allowed us to generate tens of thousands of stars once at the start of the application, then we animated a singled set of stars along the established curves. Since everything is on the GPU, the system can animate all the stars in parallel at no cost to the CPU.



Here's what it looks like when drawing white quads.

To make each quad face the camera, we used a geometry shader to transform each star position to a 2D rectangle on the screen that will contain our star texture.



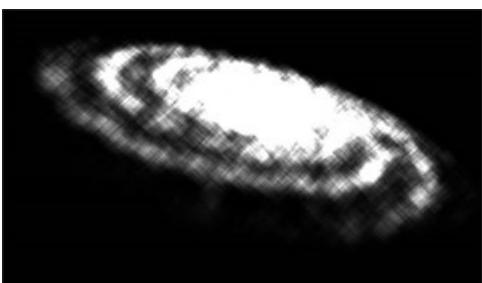
Diamonds instead of quads.

Because we wanted to limit the overdraw (number of times a pixel will be processed) as much as possible, we rotated our quads so that they would have less overlap.

Adding clouds

There are many ways to get a volumetric feeling with particles—from ray marching inside of a volume to drawing as many particles as possible to simulate a cloud. Real-time ray marching was going to be too expensive and hard to author, so we first tried building an imposter system using a method for rendering forests in games—with a lot of 2D images of trees facing the camera. When we do this in a game, we can have textures of trees rendered from a camera that rotates around, save all those images, and at runtime for each billboard card, select the image that matches the view direction. This doesn't work as well when the images are holograms. The difference between the left eye and the right eye make it so that we need a much higher resolution, or else it just looks flat, aliased, or repetitive.

On our second attempt, we tried having as many particles as possible. The best visuals were achieved when we additively drew particles and blurred them before adding them to the scene. The typical problems with that approach were related to how many particles we could draw at a single time and how much screen area they covered while still maintaining 60fps. Blurring the resulting image to get this cloud feeling was usually a very costly operation.



Without texture, this is what the clouds would look like with 2% opacity.

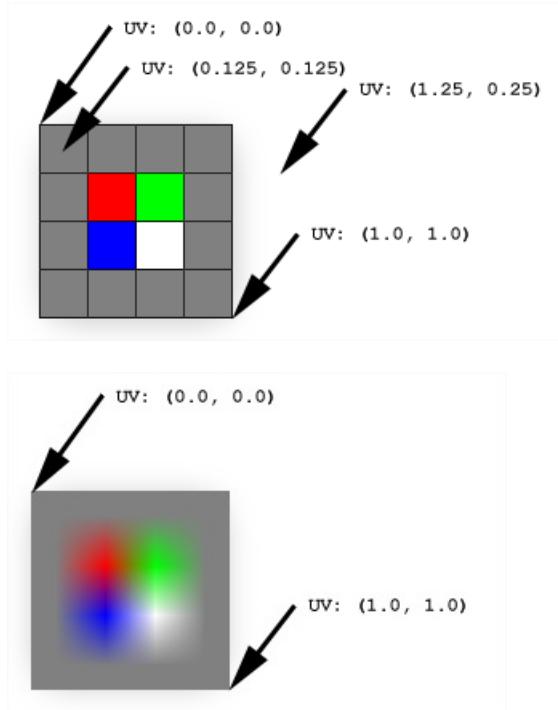
Being additive and having a lot of them means that we would have several quads on top of each other, repeatedly shading the same pixel. In the center of the galaxy, the same pixel has hundreds of quads on top of each other and this had a huge cost when being done full screen.

Doing full screen clouds and trying to blur them would have been a bad idea, so instead we decided to let the

hardware do the work for us.

A bit of context first

When using textures in a game the texture size will rarely match the area we want to use it in, but we can use different kind of texture filtering to get the graphic card to interpolate the color we want from the pixels of the texture ([Texture Filtering](#)). The filtering that interests us is [bilinear filtering](#) which will compute the value of any pixel using the 4 nearest neighbors.



Using this property, we see that each time we try to draw a texture into an area twice as big, it blurs the result.

Instead of rendering to a full screen and losing those precious milliseconds we could be spending on something else, we render to a tiny version of the screen. Then, by copying this texture and stretching it by a factor of 2 several times, we get back to full screen while blurring the content in the process.



x3 upscale back to full resolution.

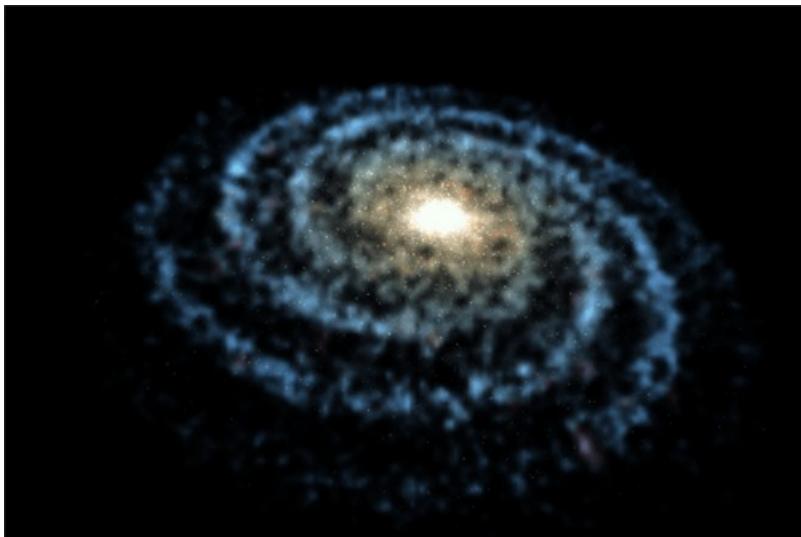
This allowed us to get the cloud part with only a fraction of the original cost. Instead of adding clouds on the full resolution, we only paint 1/64th of the pixels and just stretch the texture back to full resolution.



Left, with an upscale from 1/8th to full resolution; and right, with 3 upscale using power of 2.

Note that trying to go from 1/64th of the size to the full size in one go would look completely different, as the graphic card would still use 4 pixels in our setup to shade a bigger area and artifacts start to appear.

Then, if we add full resolution stars with smaller cards, we get the full galaxy:



Once we were on the right track with the shape, we added a layer of clouds, swapped out the temporary dots with ones we painted in Photoshop, and added some additional color. The result was a Milky Way Galaxy our art and engineering teams both felt good about and it met our goals of having depth, volume, and motion—all without taxing the CPU.



Our final Milky Way Galaxy in 3D.

More to explore

We've open-sourced the code for the Galaxy Explorer app and made it available on [GitHub](#) for developers to build on.

Interested in finding out more about the development process for Galaxy Explorer? Check out all our past project updates on the [Microsoft HoloLens YouTube channel](#).

About the authors



Karim Luccin is a Software Engineer and fancy visuals enthusiast. He was the Graphics Engineer for Galaxy Explorer.



Andy Zibits is an Art Lead and space enthusiast who managed the 3D modeling team for Galaxy Explorer and fought for even more particles.

See also

- [Galaxy Explorer on GitHub](#)
- [Galaxy Explorer project updates on YouTube](#)

Case study - Creating an immersive experience in Fragments

11/6/2018 • 2 minutes to read • [Edit Online](#)

Fragments is an interactive crime drama for Microsoft Hololens, where you follow the clues to solve a mystery that takes place in your real-world space. Find out how the Fragments development team used the unique features of HoloLens to immerse players in the action, putting them directly in the center of the story.

See also

- [Spatial mapping](#)
- [Fragments for Microsoft HoloLens](#)

Case study - Creating impossible perspectives for HoloTour

11/6/2018 • 7 minutes to read • [Edit Online](#)

We wanted your experiences in HoloTour for Microsoft HoloLens to be unforgettable. In addition to the traditional tourist stops, we planned out some "impossible perspectives" – moments that would be impossible to experience on any tour but which, via the technology in HoloLens, we could bring directly to your living room. Creating the content for these experiences required some different techniques than our standard capture process.

The content challenge

There are certain scenes in the HoloTour experience—such as the hot air balloon ride over modern-day Rome and the gladiatorial fight at the Colosseum in ancient Rome—that provide unique views you won't see anywhere else. These moments are meant to excite and amaze you, making your trip through HoloTour more than just an educational experience. They are the moments we want you to remember and get excited to tell other people about. Since we couldn't take our camera rig up into the sky and we haven't (yet) mastered time travel, each of these "impossible perspectives" called for a special approach to creating content.

Behind the scenes

Creating these unique moments and perspectives required more than just filming and editing. It took a tremendous amount of time, people with many different skills, and a little bit of Hollywood magic.

Viewing Rome from a hot air balloon

From our early planning days, we knew we wanted to do aerial views in HoloTour. Seeing Rome from the sky gives you a perspective that most people never get to see and a sense of how popular landmarks are spatially located. Trying to capture this ourselves with our existing camera and microphone rig would have been tremendously difficult, but luckily we didn't have to.

First, it's important to explain that all of the locations you visit in HoloTour have movement in them. Our goal was to make you "feel like you're really there" and since you're surrounded by movement everywhere you go in real life, our virtual destinations needed to convey ambient movement as well. For instance, when you visit the Pantheon on your trip, you'll see people wandering throughout the plaza and congregating on the steps. The background motion helps to make you feel like you're really standing in the location, rather than in a staged, static environment.

To create the aerial views for the balloon ride, we worked with other teams at Microsoft to get access to aerial panoramic imagery of Rome. The quality of these images was excellent and the view was stunning, but when we used them in the scenes without modification, they felt lifeless compared to the other parts of the tour and the lack of motion was distracting.



The hot air balloon basket, floating over Rome

To ensure the aerial locations met the same quality bar as other destinations, we decided to transform the static photographs into living, moving scenes. The first step was to edit the image and composite motion into it. We contracted a visual effects artist to help us with this. Editing was done to show clouds slowly drifting, birds flying by, and an occasional plane or helicopter traversing the skyline. On the ground, a number of cars were made to drive the streets. If you've been on the tour of Rome in HoloTour, it's unlikely that you were explicitly aware of any of this movement. That's actually great! The subtle motion isn't meant to catch your eye, but without these little touches, people noticed immediately that it was a static image in

the scene.

The second thing we did was give you a vantage point from which to view the scene. You won't feel like you're really there if it looks like you're simply floating in midair, so we created a 3D model of a balloon and placed you inside of it. This allows you to walk around the balloon and look over the edge to get a better viewpoint. We found this to be a natural and fun way to experience the aerial imagery.

The hot air balloon experience presented unique challenges for our audio team, as logistics prevented us from having microphones hovering thousands of feet over Rome. Luckily, we had a large amount of ambient audio capture from all over the city that we were able to use during post-production. We placed audio emitters at their relative locations from where they were captured from the ground. The audio was then filtered to sound distant, as if you were hearing it from the perspective of someone riding in the hot air balloon, providing an authentic, directional soundscape for the scene.

Time traveling to ancient Rome

The remnants of monuments and buildings throughout Rome are impressive even two thousand years after their construction, but we knew we had a unique opportunity to show you what it would be like to go back in time and see these structures as they appeared in ancient Rome.

Naturally, there isn't any video footage or static imagery) of the Colosseum as it appeared when it was built, so we needed to create our own. We had to do a lot of research to learn as much about the structure as we could; understanding the materials it was made from, reviewing architectural diagrams, and reading historical descriptions to get enough information to be able to make a virtual recreation.



The modern day ruins of the Colosseum with an overlay showing the arena floor as it would have looked in ancient Rome

The first thing we wanted to do was enhance traditional tours with educational overlays. In HoloTour when you visit the ruins of the Colosseum as it stands today, the arena floor is transformed to show you how it would have looked during use, including the elaborate underground staging areas. On a normal tour you might have this information described to you and you could try to imagine it, but in HoloTour you can actually see it.

For an overlay like this we had our artists match the viewpoint of our capture footage and create the overlay image by hand. The perspective needs to match so that when we replace the video with our image both will align properly.

Staging the gladiator fight

While overlays are an engaging way to teach people about history, what we were most excited about was transporting you back in time. The overlay was just a still image from a particular viewpoint, but time traveling would require the entire Colosseum to be modeled, and, as discussed above, we needed to have motion in the scene to make it feel alive. Achieving that took a considerable amount of effort.

This undertaking was too large for our team to do alone, so our art team worked with Whiskytree, an external effects company that typically works on visual effects for Hollywood movies. Whiskytree helped us recreate the Colosseum in its heyday, allowing us to teach you about the structure while standing on the arena floor and to create a view of a gladiator fight from the emperor's box. The cheering crowds and waving banners add the subtle motion necessary to make it feel like these are real places and not just images.



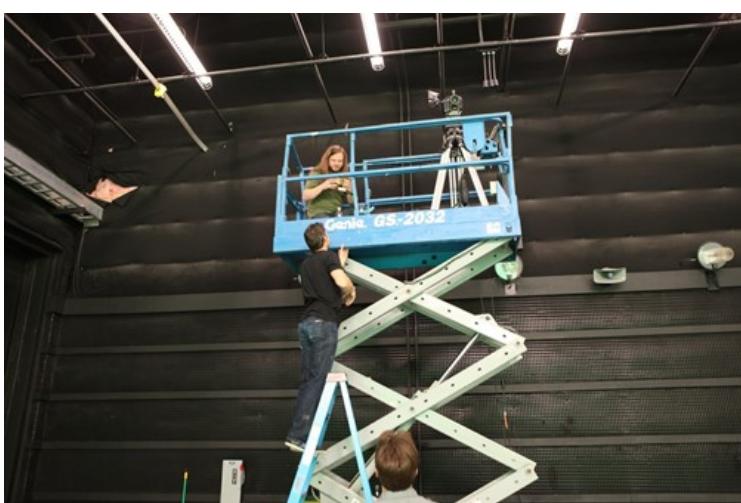
The recreated Colosseum as seen from the arena floor. When viewed in HoloTour, the banners flutter in the breeze, giving a feeling of motion.

The tour of Rome culminates with the gladiator fight. Whiskytree provided us with the arena and 3D crowd simulations rendered as video, but we needed to add in the gladiators on the arena floor. This part of our process looked more like a Hollywood video production than a project from an incubation game studio. Members of our team mapped out a rough fight sequence and then refined it with a choreographer. We hired actors to stage our mock battle and purchased armor so they would look the part. Finally, we filmed the whole scene against a green screen.



Our gladiators, getting instructions between takes

This scene places you in the Emperor's box, which meant that all the footage needed to be from that perspective. If we filmed from where the gladiators were fighting on the arena floor, we wouldn't have been able to correctly composite the fight sequence in later, so we put our camera operator up in a very tall scissor lift, looking down on the fight sequence for filming.



Getting the right perspective: filming from a scissor lift

In post-production, the gladiators were composited onto the arena floor and the perspective was correct, but one issue remained: the shadows of the gladiators on the green screen were removed as part of the compositing process. Without shadows, it looked like the gladiators were floating in the air. Luckily, Whiskytree is great at solving just this kind of problem

and they used a bit of technical wizardry to add shadows back into our scene. The result is what you see in the tour today.

About the authors



David Haley is a Senior Developer who learned more about camera rigs and video playback than he thought possible from working on HoloTour.



Jason Syltebo is an Audio Designer who made sure you could experience the soundscape of every destination you visit, even when you go back in time.



Danny Askew is a Video Artist who made sure your journey through Rome was as flawless as possible.

See also

- [Video: Microsoft HoloLens: HoloTour](#)

Case study: Expanding the design process for mixed reality

11/6/2018 • 11 minutes to read • [Edit Online](#)

As Microsoft launched the HoloLens to an audience of eager developers in 2016, the team had already partnered with studios inside and outside of Microsoft to build the device's launch experiences. These teams learned by doing, finding both opportunities and challenges in the new field of mixed reality design.

To help new teams and partners innovate more effectively, we turned their methods and insights into a curriculum of design and development lessons that we teach to developers in our Mixed Reality Academy (including week-long design workshops we offer to our enterprise partners).

Below is a snapshot of these learnings, part of a larger process we use to help our enterprise partners prepare their teams for mixed reality development. While many of these methods were originally targeted for HoloLens development, ideating and communicating in 3D are critical to the full spectrum of mixed reality experiences.

Thinking spatially during the design process

Any design process or design framework is meant to iterate thinking: To approach a problem broadly, to share ideas with others effectively, to evaluate those ideas and reach a solution. Today, we have well-established design and development methods for building experiences on desktops, phones, and tablets. Teams have clear expectations of what is necessary to iterate an idea and turn it into a product for users.

Often teams are a mix of development, design, research, and management, but all roles can participate in the design process. The barrier to entry to contribute an idea for a mobile app can be as simple as drawing a rectangle for the device's screen. While sketching boxes and lines for the UI elements, with arrows to indicate motion or interactions, can be enough to establish technical requirements or define potential user behavior.

With mixed reality, the traditional 2D design process begins to break down: sketching in 3D is difficult for most people and using 2D tools, like pen and paper or whiteboards, can often limit ideas to those dimensions.

Meanwhile 3D tools, built for gaming or engineering, require a high degree of proficiency to quickly flesh out ideas. The lack of lightweight tools is compounded by the technical uncertainty inherent with new devices, where foundational interaction methods are still being established. These challenges can potentially limit the design contributions of your team to only those with 3D development backgrounds—drastically reducing the team's ability to iteration.



Teams from the Mixed Reality Partner Program in our workshop

When we work with external partners, we hear stories of teams ‘waiting for the developer to finish the prototype’ before they can continue with their design process. This often means the rest of the team is blocked in making meaningful progress on the product, overloading the developer to solve both the technical implementation as well as major components of the user experience (as they attempt to put a rough idea into code).

Techniques for expanding the design process

Our teams at Microsoft have a set of techniques to more effectively include their team and quickly iterate through complex design problems. While not a formal process, these techniques are intended to supplement rather than replace your workflow. These methods allow those without specialized 3D skills to offer ideas before diving into the prototyping phase. Different roles and disciplines (not just 3D designers and developers) can be part of the design process, uncovering opportunities and identifying possible challenges that might occur later in development.

Generating ideas with bodystorming

Getting your team to think about events occurring in the real world, beyond the traditional world of 2D devices, is key to developing innovative mixed reality experiences. At Microsoft, we have found the best way to do this is to encourage interaction with physical props in a real-world space. Using simple, cheap crafting materials we build physical props to represent digital objects, user interfaces, and animations in a proposed experience. This technique is called bodystorming and has been a staple of product ideation within industrial design for decades.



Simple, cheap art supplies used in bodystorming

Simple, physical props level the playing field for participants, allowing individuals with different skill sets and backgrounds to contribute ideas and uncover opportunities inherent to mixed reality experiences instead of being locked into the paradigm of 2D thinking. While technical prototyping or high-fidelity storyboarding requires a skilled 3D developer or artist, a few Styrofoam balls and cardboard can be enough to showcase how an interface might unfold in physical space. These techniques apply to both mixed reality development with HoloLens and the immersive headsets. For example, a set of plastic connectors might roughly illustrate the size of holograms that appear in a HoloLens experience or as props to act out interactable elements or motion designs in a virtual world.

Bodystorming is a technique to quickly generate ideas and evaluate ideas that are too nebulous to prototype. At Microsoft, bodystorming is most commonly used to quickly vet an idea, although it can be helpful to host a more in-depth session if you involve outside stakeholders who are not familiar with mixed reality development or need to distill very broad scenarios. Remember that the purpose of bodystorming is to ideate quickly and efficiently by encouraging participants to think spatially. Detailed artwork or precise measurements are not important at this stage. Physical props need only meet the minimum requirements to explore or communicate an idea. Ideas presented through bodystorming are not expected to be fully vetted, but the process can help narrow down possibilities to test later during the in-device prototyping phase. As such, bodystorming is not intended to replace technical prototyping, but rather offset the burden of solving both technical and design challenges during the prototyping phase.

Acting and expert feedback

Following the bodystorming process of ideating with physical objects in the real world, the next step is to walk through an experience with these objects. We call this phase of the process acting and it often involves staging how a user would move through the experience or a specific interaction.



Teams acting out a scenario during a workshop

Acting with physical props allows participants to not only experience the thinking through the perspective of the user but allow outside observers to see how events play out. This presents an ideal time to include a wider audience of team members or stakeholders who are able to provide specific 'expert' feedback. For example, if you are exploring a mixed reality experience designed for hospitals, acting out your thinking to a medical professional can provide invaluable feedback. Another example is when you might have a specific challenge you are trying to understand, like spatial audio or budgeting for asset quality vs. performance. In these cases, acting gives experts a quick, rough concept of how the experience might unfold, without the need for a device-based prototype.

This sort of acting is not formalized, with no need to bring in professional talent, but in situations where we want to share the thinking with others who are not present we will video record a 'scene' of interaction or include a storyboard artist to observe and sketch key moments. Acting can also be a very lightweight activity, often happening in-situ during the bodystorming phase. Deciding which to use depends on the audience (and fidelity needed to elicit the necessary type of feedback) but ultimately comes down to whatever will most effectively capture your team's thinking.

Capturing ideas with storyboards

The best method for conveying the ideas and concepts of your proposed experience depends on your intended audience, as well as the type of feedback your next iteration requires. When presenting new ideas to team members, low fidelity re-enactments of bodystorming can be enough to bring someone up to speed. When introducing your experience concepts to new stakeholders or potential users, the best method is often storyboarding. Storyboarding is a technique common to the entertainment industry, usually found behind the scenes in movies and video game development, and helps convey both the overall flow of an experience (at low fidelities) as well as the aesthetic look and feel (at high fidelities). Just as with prototyping, understanding the fidelity needs of your storyboard is key to gathering the right kind of feedback and avoiding counter-productive discussions.



Example of a low-fidelity storyboard

Low-fidelity storyboards are the right fidelity for quick discussions, especially when conveying high-level ideas. These can be as simple as stick-figure drawings and primitive shapes to denote virtual elements in a scene or the proximity of interactive components (both physical and virtual). While these are useful given the relative ease and low skill barrier to execute, remember the lesson from bodystorming: Not everyone can see 2D depictions of an experience and understand the 3D implications.



Example of a high-fidelity storyboard

High-fidelity storyboards are a powerful tool when bringing in new stakeholders or when combining insights from a bodystorming session with the proposed aesthetic direction of your experience. Storyboards can build off mood boards to illustrate the final appearance of a virtual experience, as well as capture key moments that may be pivotal to the final product. Keep in mind that high-fidelity storyboards often require an artist, especially one embedded within the team, who can capture difficult-to-describe ideas. At Microsoft we have added storyboard artists, often with backgrounds in game development, to our mixed reality teams who attend meetings prepared to quickly capture sketches that will later be expanded into higher fidelity mockups. These individuals work closely with technical artists, helping to convey the artistic direction of assets used in the final experience.

Expanding your team's skills

In addition to the techniques described, altering your process to better accommodate mixed reality experiences depends on more closely connecting technical and design roles. Working with partners at Microsoft, we have witnessed several teams successfully transition into 3D development, and find the biggest advantage comes from team members stepping out of their comfort zone and developing skills that give them more involvement throughout the development process (not just limiting their skills to design or development).

Feedback and iteration is key to any successful design, and with mixed reality experiences this sort of feedback often skews to more technical concepts, especially given the relative nascence of mixed reality's technology and tools. Building your design team's skills in technical areas, including a proficiency with tools like Unity or Unreal, helps those individuals better understand the approach of developers and provide more effective feedback during prototyping. Similarly, developers who understand the fundamental UX concepts of mixed reality experiences (and potential design pitfalls) can help them offer implementation insights during the early phases of planning.

We usually get questions from partners working in mixed reality about how best to grow their team and skill sets with 3D tools. This is a difficult question to answer given the state of the industry, as teams are faced with building

tomorrow's experiences with yesterday's tools. Many teams are gathering individuals with backgrounds in gaming and entertainment, where 3D development has been a staple for decades, or encouraging their existing teams to pick up tools like Unity and 3D modeling software. While these are both essential needs for meeting the baseline of mixed reality experiences today, there will very likely be a great number of specialized tools and skills to help build tomorrow's ground-breaking applications.

The field of mixed reality design will change dramatically in the near future, with more focus on data and cloud services in holographic experiences. This next generation of experiences will take advantage of advanced natural language processing and real-time computer vision, as well as more inherently social scenarios within immersive experiences. For example, experiences in virtual reality allows designers to build imaginative environments and spaces, requiring a skill set more akin to architecture or psychology rather than traditional interface design. Or consider experiences in the real world with HoloLens, where the use cases often involve highly specialized fields such as medicine, engineering, or manufacturing, where specific environments and real-time data are essential elements of the experience. Working across roles, leveraging both design and specialized knowledge, and having a willingness to learn new tools are invaluable skill for teams working in mixed reality.

Helping your team explore mixed reality quickly and effectively

In the early days of HoloLens, these techniques were borne out of necessity as the device's early prototype hardware proved to be a less than ideal format for quickly iterating through design ideas. Interaction methods, including fundamental UI in the operating system were explored with bodystorming, while acting helped our usability team understand fundamental user behaviors for scenarios in the workplace. In the end, the team was able to establish a strong baseline for the core UX of HoloLens, communicating concepts across team roles, allowing OS development to move quickly in parallel with hardware development.



Whether it is bodystorming, acting, reviewing with experts, or storyboarding, these techniques are intended to save you time and effort. While we are still very much in the early days of HoloLens and virtual reality development, expanding your design process for mixed reality will help your team spend their energy exploring new and challenging design problems rather than overcoming difficulties communicating ideas.

Sample list of workshop supplies

Simple, cheap art supplies are key to providing team members with the tools necessary to explain ideas without requiring advanced artistic skills. Here is a sample of what our team commonly uses during bodystorming:

- Styrofoam discs
- Styrofoam cubes
- Styrofoam cones
- Styrofoam spheres
- Cardboard boxes
- Wooden dowels
- Lollipop sticks
- Cardstock
- Paper cups
- Duct tape
- Masking tape
- Scissors
- Paperclips
- Filing clamps
- Post-Its
- Twine
- Pencils
- Sharpies

See also

- [Case study - My first year on the HoloLens design team](#)
- [Case study - AfterNow's process - envisioning, prototyping, building](#)

Case study - Expanding the spatial mapping capabilities of HoloLens

11/6/2018 • 12 minutes to read • [Edit Online](#)

When creating our first apps for Microsoft HoloLens, we were eager to see just how far we could push the boundaries of spatial mapping on the device. Jeff Everett, a software engineer at Microsoft Studios, explains how a new technology was developed out of the need for more control over how holograms are placed in a user's real-world environment.

Watch the video

Beyond spatial mapping

While we were working on [Fragments](#) and [Young Conker](#), two of the first games for HoloLens, we found that when we were doing procedural placement of holograms in the physical world, we needed a higher level of understanding about the user's environment. Each game had its own specific placement needs: In [Fragments](#), for example, we wanted to be able to distinguish between different surfaces—such as the floor or a table—to place clues in relevant locations. We also wanted to be able to identify surfaces that life-size holographic characters could sit on, such as a couch or a chair. In [Young Conker](#), we wanted Conker and his opponents to be able to use raised surfaces in a player's room as platforms.

[Asobo Studios](#), our development partner for these games, faced this problem head-on and created a technology that extends the spatial mapping capabilities of HoloLens. Using this, we could analyze a player's room and identify surfaces such as walls, tables, chairs, and floors. It also gave us the ability to optimize against a set of constraints to determine the best placement for holographic objects.

The spatial understanding code

We took Asobo's original code and created a library that encapsulates this technology. Microsoft and Asobo have now open-sourced this code and made it available on [MixedRealityToolkit](#) for you to use in your own projects. All the source code is included, allowing you to customize it to your needs and share your improvements with the community. The code for the C++ solver has been wrapped into a UWP DLL and exposed to Unity with a [drop-in prefab contained within MixedRealityToolkit](#).

There are many useful queries included in the Unity sample that will allow you to find empty spaces on walls, place objects on the ceiling or on large spaces on the floor, identify places for characters to sit, and a myriad of other spatial understanding queries.

While the spatial mapping solution provided by HoloLens is designed to be generic enough to meet the needs of the entire gamut of problem spaces, the spatial understanding module was built to support the needs of two specific games. As such, its solution is structured around a specific process and set of assumptions:

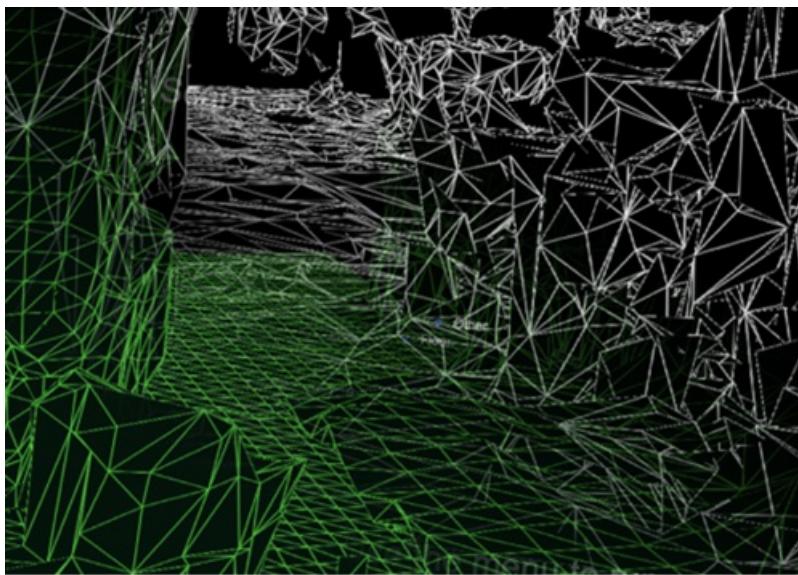
- **Fixed size playspace:** The user specifies the maximum playspace size in the init call.
- **One-time scan process:** The process requires a discrete scanning phase where the user walks around, defining the playspace. Query functions will not function until after the scan has been finalized.
- **User driven playspace “painting”:** During the scanning phase, the user moves and looks around the playspace, effectively painting the areas which should be included. The generated mesh is important to provide user feedback during this phase.

- **Indoors home or office setup:** The query functions are designed around flat surfaces and walls at right angles. This is a soft limitation. However, during the scanning phase, a primary axis analysis is completed to optimize the mesh tessellation along major and minor axis.

Room Scanning Process

When you load the spatial understanding module, the first thing you'll do is scan your space, so all the usable surfaces—such as the floor, ceiling, and walls—are identified and labeled. During the scanning process, you look around your room and "paint" the areas that should be included in the scan.

The mesh seen during this phase is an important piece of visual feedback that lets users know what parts of the room are being scanned. The DLL for the spatial understanding module internally stores the playspace as a grid of 8cm sized voxel cubes. During the initial part of scanning, a primary component analysis is completed to determine the axes of the room. Internally, it stores its voxel space aligned to these axes. A mesh is generated approximately every second by extracting the isosurface from the voxel volume.



Spatial mapping mesh in white and understanding playspace mesh in green

The included SpatialUnderstanding.cs file manages the scanning phase process. It calls the following functions:

- **SpatialUnderstanding_Init:** Called once at the start.
- **GeneratePlayspace_InitScan:** Indicates that the scan phase should begin.
- **GeneratePlayspace_UpdateScan_DynamicScan:** Called each frame to update the scanning process. The camera position and orientation is passed in and is used for the playspace painting process, described above.
- **GeneratePlayspace_RequestFinish:** Called to finalize the playspace. This will use the areas "painted" during the scan phase to define and lock the playspace. The application can query statistics during the scanning phase as well as query the custom mesh for providing user feedback.
- **Import_UnderstandingMesh:** During scanning, the **SpatialUnderstandingCustomMesh** behavior provided by the module and placed on the understanding prefab will periodically query the custom mesh generated by the process. In addition, this is done once more after scanning has been finalized.

The scanning flow, driven by the **SpatialUnderstanding** behavior calls **InitScan**, then **UpdateScan** each frame. When the statistics query reports reasonable coverage, the user can airtap to call **RequestFinish** to indicate the end of the scanning phase. **UpdateScan** continues to be called until its return value indicates that the DLL has completed processing.

The queries

Once the scan is complete, you'll be able to access three different types of queries in the interface:

- **Topology queries:** These are fast queries that are based on the topology of the scanned room.
- **Shape queries:** These utilize the results of your topology queries to find horizontal surfaces that are a good match to custom shapes that you define.
- **Object placement queries:** These are more complex queries that find the best-fit location based on a set of rules and constraints for the object.

In addition to the three primary queries, there is a raycasting interface which can be used to retrieve tagged surface types and a custom watertight room mesh can be copied out.

Topology queries

Within the DLL, the topology manager handles labeling of the environment. As mentioned above, much of the data is stored within surfels, which are contained within a voxel volume. In addition, the **PlaySpaceInfos** structure is used to store information about the playspace, including the world alignment (more details on this below), floor, and ceiling height.

Heuristics are used for determining floor, ceiling, and walls. For example, the largest and lowest horizontal surface with greater than 1 m² surface area is considered the floor. Note that the camera path during the scanning process is also used in this process.

A subset of the queries exposed by the Topology manager are exposed out through the DLL. The exposed topology queries are as follows:

- QueryTopology_FindPositionsOnWalls
- QueryTopology_FindLargePositionsOnWalls
- QueryTopology_FindLargestWall
- QueryTopology_FindPositionsOnFloor
- QueryTopology_FindLargestPositionsOnFloor
- QueryTopology_FindPositionsSittable

Each of the queries has a set of parameters, specific to the query type. In the following example, the user specifies the minimum height & width of the desired volume, minimum placement height above the floor, and the minimum amount of clearance in front of the volume. All measurements are in meters.

```
EXTERN_C __declspec(dllexport) int QueryTopology_FindPositionsOnWalls(
    _In_ float minHeightOfWallSpace,
    _In_ float minWidthOfWallSpace,
    _In_ float minHeightAboveFloor,
    _In_ float minFacingClearance,
    _In_ int locationCount,
    _Inout_ Dll_Interface::TopologyResult* locationData)
```

Each of these queries takes a pre-allocated array of **TopologyResult** structures. The **locationCount** parameter specifies the length of the passed-in array. The return value reports the number of returned locations. This number is never greater than the passed-in **locationCount** parameter.

The **TopologyResult** contains the center position of the returned volume, the facing direction (i.e. normal), and the dimensions of the found space.

```
struct TopologyResult
{
    DirectX::XMFLOAT3 position;
    DirectX::XMFLOAT3 normal;
    float width;
    float length;
};
```

Note that in the Unity sample, each of these queries is linked up to a button in the virtual UI panel. The sample hard codes the parameters for each of these queries to reasonable values. See *SpaceVisualizer.cs* in the sample code for more examples.

Shape queries

Inside of the DLL, the shape analyzer (**ShapeAnalyzer_W**) uses the topology analyzer to match against custom shapes defined by the user. The Unity sample has a pre-defined set of shapes which are shown in the query menu, on the shape tab.

Note that the shape analysis works on horizontal surfaces only. A couch, for example, is defined by the flat seat surface and the flat top of the couch back. The shape query looks for two surfaces of a specific size, height, and aspect range, with the two surfaces aligned and connected. Using the APIs terminology, the couch seat and the top of the back of the couch are shape components and the alignment requirements are shape component constraints.

An example query defined in the Unity sample (**ShapeDefinition.cs**), for "sittable" objects is as follows:

```
shapeComponents = new List<ShapeComponent>()
{
    new ShapeComponent(
        new List<ShapeComponentConstraint>()
    {
        ShapeComponentConstraint.Create_SurfaceHeight_Between(0.2f, 0.6f),
        ShapeComponentConstraint.Create_SurfaceCount_Min(1),
        ShapeComponentConstraint.Create_SurfaceArea_Min(0.035f),
    }),
};

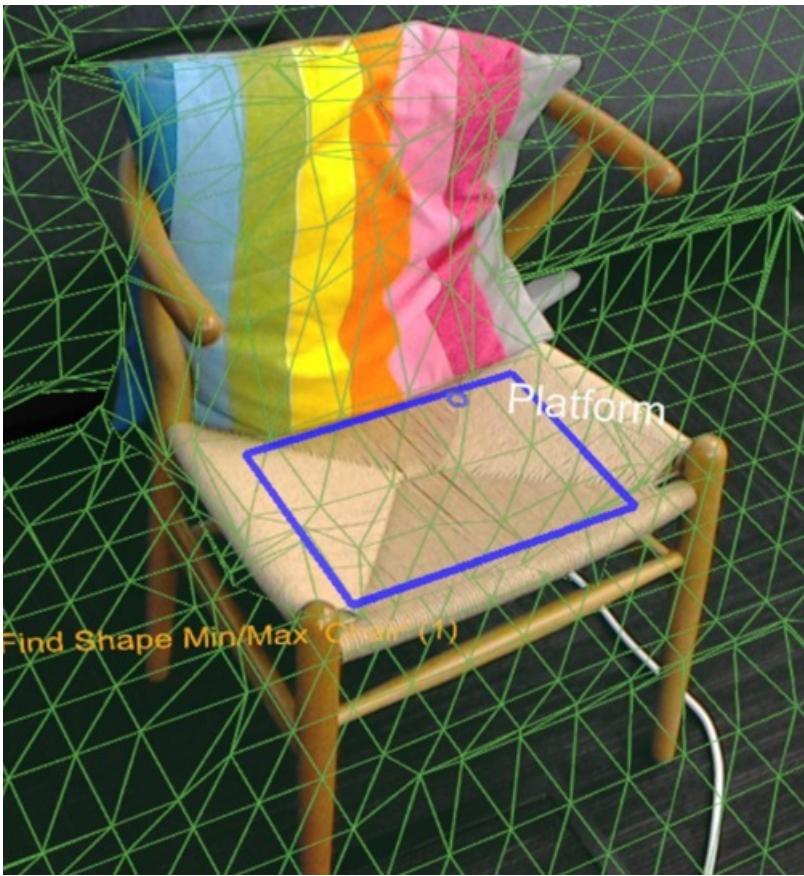
AddShape("Sittable", shapeComponents);
```

Each shape query is defined by a set of shape components, each with a set of component constraints and a set of shape constraints which lists dependencies between the components. This example includes three constraints in a single component definition and no shape constraints between components (as there is only one component).

In contrast, the couch shape has two shape components and four shape constraints. Note that components are identified by their index in the user's component list (0 and 1 in this example).

```
shapeConstraints = new List<ShapeConstraint>()
{
    ShapeConstraint.Create_RectanglesSameLength(0, 1, 0.6f),
    ShapeConstraint.Create_RectanglesParallel(0, 1),
    ShapeConstraint.Create_RectanglesAligned(0, 1, 0.3f),
    ShapeConstraint.Create_AtBackOf(1, 0),
};
```

Wrapper functions are provided in the Unity module for easy creation of custom shape definitions. The full list of component and shape constraints can be found in **SpatialUnderstandingDll.cs** within the **ShapeComponentConstraint** and the **ShapeConstraint** structures.



The blue rectangle highlights the results of the chair shape query.

Object placement solver

Object placement queries can be used to identify ideal locations in the physical room to place your objects. The solver will find the best-fit location given the object rules and constraints. In addition, object queries persist until the object is removed with **Solver_RemoveObject** or **Solver_RemoveAllObjects** calls, allowing constrained multi-object placement.

Object placement queries consist of three parts: placement type with parameters, a list of rules, and a list of constraints. To run a query, use the following API:

```
public static int Solver_PlaceObject(
    [In] string objectName,
    [In] IntPtr placementDefinition, // ObjectPlacementDefinition
    [In] int placementRuleCount,
    [In] IntPtr placementRules,      // ObjectPlacementRule
    [In] int constraintCount,
    [In] IntPtr placementConstraints, // ObjectPlacementConstraint
    [Out] IntPtr placementResult)
```

This function takes an object name, placement definition, and a list of rules and constraints. The C# wrappers provide construction helper functions to make rule and constraint construction easy. The placement definition contains the query type — that is, one of the following:

```

public enum PlacementType
{
    Place_OnFloor,
    Place_OnWall,
    Place_OnCeiling,
    Place_OnShape,
    Place_OnEdge,
    Place_OnFloorAndCeiling,
    Place_RandomInAir,
    Place_InMidAir,
    Place_UnderFurnitureEdge,
}

```

Each of the placement types has a set of parameters unique to the type. The **ObjectPlacementDefinition** structure contains a set of static helper functions for creating these definitions. For example, to find a place to put an object on the floor, you can use the following function:

```

public static ObjectPlacementDefinition Create_OnFloor(Vector3 halfDims)

```

In addition to the placement type, you can provide a set of rules and constraints. Rules cannot be violated. Possible placement locations that satisfy the type and rules are then optimized against the set of constraints to select the optimal placement location. Each of the rules and constraints can be created by the provided static creation functions. An example rule and constraint construction function is provided below.

```

public static ObjectPlacementRule Create_AwayFromPosition(
    Vector3 position, float minDistance)
public static ObjectPlacementConstraint Create_NearPoint(
    Vector3 position, float minDistance = 0.0f, float maxDistance = 0.0f)

```

The object placement query below is looking for a place to put a half meter cube on the edge of a surface, away from other previously placed objects and near the center of the room.

```

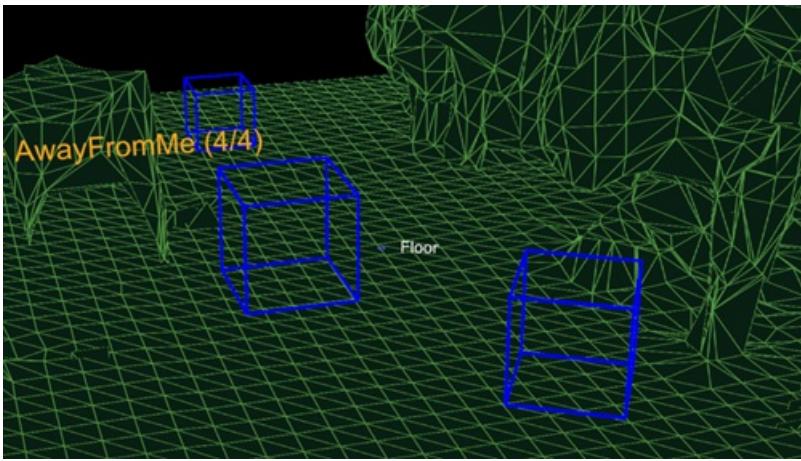
List<ObjectPlacementRule> rules =
    new List<ObjectPlacementRule>() {
        ObjectPlacementRule.Create_AwayFromOtherObjects(1.0f),
    };

List<ObjectPlacementConstraint> constraints =
    new List<ObjectPlacementConstraint> {
        ObjectPlacementConstraint.Create_NearCenter(),
    };

Solver_PlaceObject(
    "MyCustomObject",
    new ObjectPlacementDefinition.Create_OnEdge(
    new Vector3(0.25f, 0.25f, 0.25f),
    new Vector3(0.25f, 0.25f, 0.25f)),
    rules.Count,
    UnderstandingDLL.PinObject(rules.ToArray()),
    constraints.Count,
    UnderstandingDLL.PinObject(constraints.ToArray()),
    UnderstandingDLL.GetStaticObjectPlacementResultPtr());

```

If successful, an **ObjectPlacementResult** structure containing the placement position, dimensions and orientation is returned. In addition, the placement is added to the DLL's internal list of placed objects. Subsequent placement queries will take this object into account. The **LevelSolver.cs** file in the Unity sample contains more example queries.



The blue boxes show the result from three Place On Floor queries with "away from camera position" rules.

Tips:

- When solving for placement location of multiple objects required for a level or application scenario, first solve indispensable and large objects to maximize the probability that a space can be found.
- Placement order is important. If object placements cannot be found, try less constrained configurations. Having a set of fallback configurations is critical to supporting functionality across many room configurations.

Ray casting

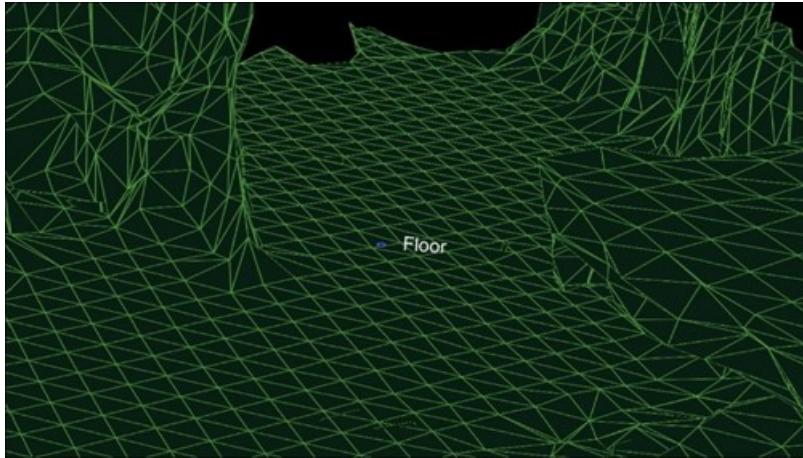
In addition to the three primary queries, a ray casting interface can be used to retrieve tagged surface types and a custom watertight playspace mesh can be copied out. After the room has been scanned and finalized, labels are internally generated for surfaces like the floor, ceiling, and walls. The **PlayspaceRaycast** function takes a ray and returns if the ray collides with a known surface and if so, information about that surface in the form of a

RaycastResult

```
struct RaycastResult
{
    enum SurfaceTypes
    {
        Invalid, // No intersection
        Other,
        Floor,
        FloorLike, // Not part of the floor topology,
                    // but close to the floor and looks like the floor
        Platform, // Horizontal platform between the ground and
                    // the ceiling
        Ceiling,
        WallExternal,
        WallLike, // Not part of the external wall surface,
                  // but vertical surface that looks like a
                  // wall structure
    };
    SurfaceTypes SurfaceType;
    float SurfaceArea; // Zero if unknown
                      // (i.e. if not part of the topology analysis)
    DirectX::XMFLOAT3 IntersectPoint;
    DirectX::XMFLOAT3 IntersectNormal;
};
```

Internally, the raycast is computed against the computed 8cm cubed voxel representation of the playspace. Each voxel contains a set of surface elements with processed topology data (also known as surfels). The surfels contained within the intersected voxel cell are compared and the best match used to look up the topology information. This topology data contains the labeling returned in the form of the **SurfaceTypes** enum, as well as the surface area of the intersected surface.

In the Unity sample, the cursor casts a ray each frame. First, against Unity's colliders; second, against the understanding module's world representation; and finally, against the UI elements. In this application, UI gets priority, then the understanding result, and finally, Unity's colliders. The **SurfaceType** is reported as text next to the cursor.



Raycast result reporting intersection with the floor.

Get the code

The open-source code is available in [MixedRealityToolkit](#). Let us know on the [HoloLens Developer Forums](#) if you use the code in a project. We can't wait to see what you do with it!

About the author



Jeff Everett is a software engineering lead who has worked on HoloLens since the early days, from incubation to experience development. Before HoloLens, he worked on the Xbox Kinect and in the games industry on a wide variety of platforms and games. Jeff is passionate about robotics, graphics, and things with flashy lights that go beep. He enjoys learning new things and working on software, hardware, and particularly in the space where the two intersect.

See also

- [Spatial mapping](#)
- [Spatial mapping design](#)
- [Room scan visualization](#)
- [MixedRealityToolkit-Unity](#)
- [Asobo Studio: Lessons from the frontline of HoloLens development](#)

Case study - 3 HoloStudio UI and interaction design learnings

11/6/2018 • 3 minutes to read • [Edit Online](#)

HoloStudio was one of the first Microsoft apps for HoloLens. Because of this, we had to create new best practices for 3D UI and interaction design. We did this through a lot of user testing, prototyping, and trial and error.

We know that not everyone has the resources at their disposal to do this type of research, so we had our Sr. Holographic Designer, Marcus Ghaly, share three things we learned during the development of HoloStudio about UI and interaction design for HoloLens apps.

Watch the video

Problem #1: People didn't want to move around their creations

We originally designed the workbench in HoloStudio as a rectangle, much like you'd find in the real world. The problem is that people have a lifetime of experience that tells them to stay still when they're sitting at a desk or working in front of a computer, so they weren't moving around the workbench and exploring their 3D creation from all sides.



We had the insight to make the workbench round, so that there was no "front" or clear place that you were supposed to stand. When we tested this, suddenly people started moving around and exploring their creations on their own.



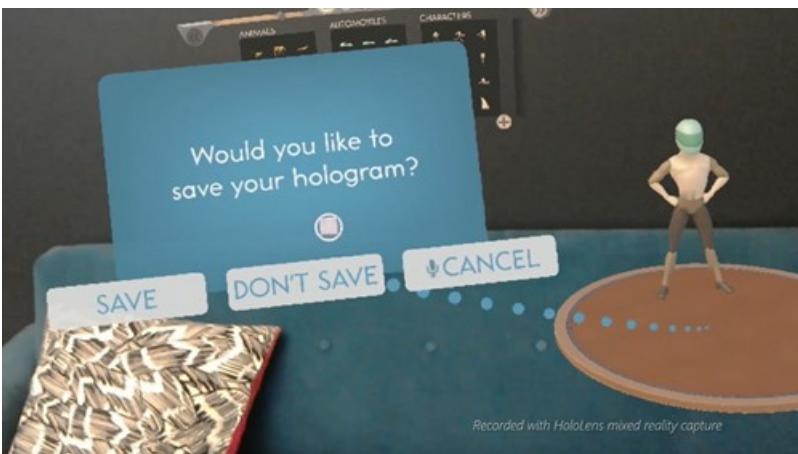
What we learned

Always be thinking about what's comfortable for the user. Taking advantage of their physical space is a cool feature of HoloLens and something you can't do with other devices.

Problem #2: Modal dialogs are sometimes out of the holographic frame

Sometimes, your user may be looking in a different direction from something that needs their attention in your app. On a PC, you can just pop up a dialog, but if you do this in someone's face in a 3D environment, it can feel like the dialog is getting in their way. You need them to read the message, but their instinct is to try to get away from it. This reaction is great if you're playing a game, but in a tool designed for work, it's less than ideal.

After trying a few different things, we finally settled on using a "thought bubble" system for our dialogs and added tendrils that users can follow to where their attention is needed in our application. We also made the tendrils pulse, which implied a sense of directionality so that users knew where to go.

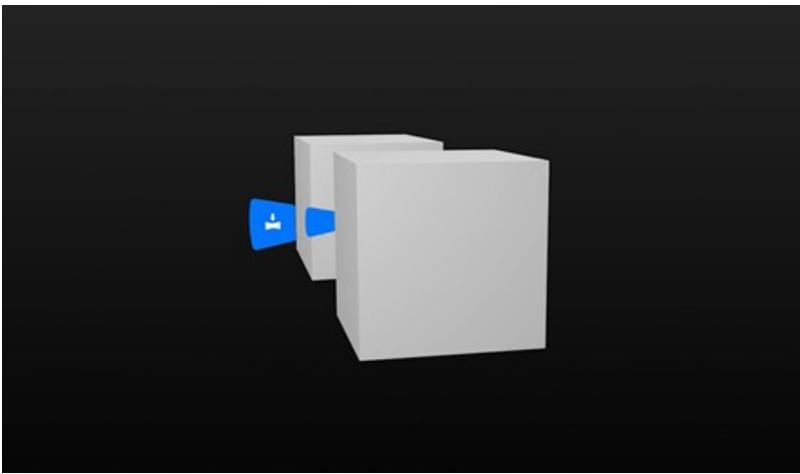


What we learned

It's much harder in 3D to alert users to things they need to pay attention to. Using attention directors such as [spatial sound](#), light rays, or thought bubbles, can lead users to where they need to be.

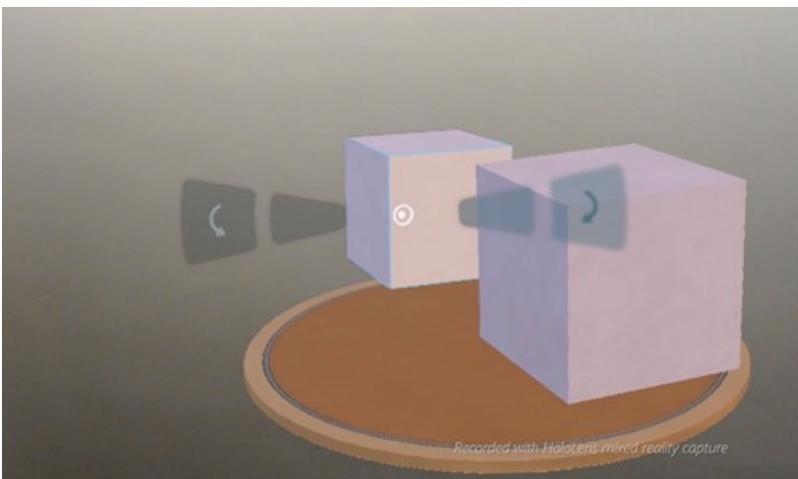
Problem #3: Sometimes UI can get blocked by other holograms

There are times when a user wants to interact with a hologram and its associated UI controls, but they are blocked from view because another hologram is in front of them. While we were developing HoloStudio, we used trial and error to come to a solution for this.



We tried moving the UI control closer to the user so it couldn't get blocked, but found that it wasn't comfortable for the user to look at a control that was near to you while simultaneously looking at a hologram that was far away. If, however, we moved the control in front of the closest hologram to the user, they felt like it was detached from the hologram it should be affecting.

We finally ended up ghosting the UI control, and put it at the same distance from the user as the hologram it's associated with, so they both feel connected. This allows the user to interact with the control even though it's been obscured.



What we learned

Users need to be able to easily access UI controls even if they've been blocked, so figure out methods to ensure that users can complete their tasks no matter where their holograms are in the real world.

About the author



Marcus Ghaly

Sr. Holographic Designer @Microsoft

See also

- [Interaction fundamentals](#)

Case study - Lessons from the Lowe's kitchen

11/6/2018 • 2 minutes to read • [Edit Online](#)

The HoloLens team wants to share some of the best practices that were derived from the Lowe's HoloLens project. Below is a video of the Lowe's HoloLens projected demonstrated at Satya's 2016 Ignite keynote.

Lowe's HoloLens Best Practices

The two videos cover best practices that were derived from the Lowe's HoloLens Pilot that has been in two Lowe's stores since April 2016. The key topics are:

- Maximize performance for a mobile device
- Create UX methods with a full holographic frame (2nd talk)
- Precision alignment (2nd talk)
- Shared holographic experiences (2nd talk)
- Interacting with customers (2nd talk)

Video 1

Maximize performance for a mobile device HoloLens is an untethered device with all the processing taking place in the device. This requires a mobile platform and requires a mindset similar to creating mobile applications. Microsoft recommends that your HoloLens application maintain 60FPS to provide a delicious experience for your users. Having low FPS can result in unstable holograms.

Some of the most important things to look at when developing on HoloLens is asset optimization/decimation, using custom shaders (available for free in the [HoloLens Toolkit](#)). Another important consideration is to measure the frame rate from the very beginning of your project. Depending on the project, the order of displaying your assets can also be a big contributor

Video 2

Create UX methods with a full holographic frame It's important to understand the placement of holograms in a physical world. With Lowe's we talk about different UX methods that help users experience holograms up close while still seeing the larger environment of holograms.

Precision alignment For the Lowe's scenario, it was paramount to the experience to have precision alignment of the holograms to the physical kitchen. We discuss techniques helps ensure an experience that convinces users that their physical environment has changed.

Shared holographic experiences Couples are the primary way that the Lowe's experience is consumed. One person can change the countertop and the other person will see the changes. We called this "shared experiences".

Interacting with customers Lowe's designers are not using a HoloLens, but they need to see what the customers are seeing. We show how to capture what the customer is seeing on a UWP application.

Case study - Looking through holes in your reality

11/6/2018 • 6 minutes to read • [Edit Online](#)

When people think about mixed reality and what they can do with Microsoft HoloLens, they usually stick to questions like "What objects can I add to my room?" or "What can I layer on top of my space?" I'd like to highlight another area you can consider—essentially a magic trick—using the same technology to look into or through real physical objects around you.

The tech

If you've fought aliens as they break through your walls in [RoboRaid](#), unlocked a wall safe in [Fragments](#), or were lucky enough to see the UNSC Infinity hangar in the [Halo 5 experience at E3 in 2015](#), then you've seen what I'm talking about. Depending on your imagination, this visual trick can be used to put temporary holes in your drywall or to hide worlds under a loose floorboard.



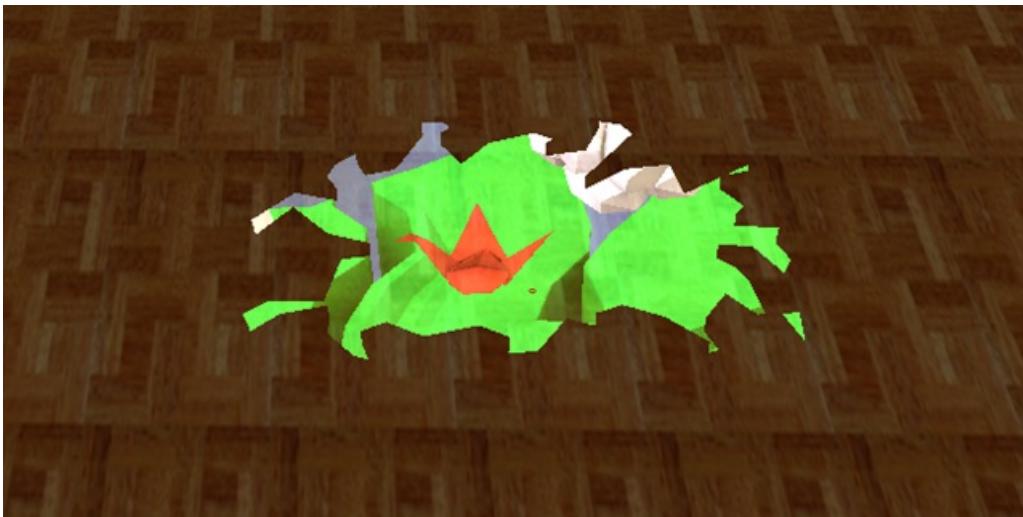
RoboRaid adds three-dimensional pipes and other structure behind your walls, visible only through holes created as the invaders break through.

Using one of these unique holograms on HoloLens, an app can provide the illusion of content behind your walls or through your floor in the same way that reality presents itself through an actual window. Move yourself left, and you can see whatever is on the right side. Get closer, and you can see a bit more of everything. The major difference is that real holes allow you through, while your floor stubbornly won't let you climb through to that magical holographic content. (I'll add a task to the backlog.)

Behind the scenes

This trick is a combination of two effects. First, holographic content is pinned to the world using "spatial anchors." Using anchors to make that content "world-locked" means that what you're looking at doesn't visually drift away from the physical objects near it, even as you move or the underlying spatial mapping system updates its 3D model of your room.

Secondly, that holographic content is visually limited to a very specific space, so you can only see through the hole in your reality. That occlusion is necessary to require looking through a logical hole, window, or doorway, which sells the trick. Without something blocking most of the view, a crack in space to a secret Jurassic dimension might just look like a poorly placed dinosaur.



This is not an actual screenshot, but an illustration of how the secret underworld from the [MR Basics 101](#) looks on HoloLens. The black enclosure doesn't show up, but you can see content through a virtual hole. (When looking through an actual device, the floor would seem to disappear even more because your eyes focus at a further distance as if it's not even there.)

World-locking holographic content

In Unity, causing holographic content to stay world-locked is as easy as adding a WorldAnchor component:

```
myObject.AddComponent<WorldAnchor>();
```

The WorldAnchor component will constantly adjust the position and rotation of its GameObject (and thus anything else under that object in the hierarchy) to keep it stable relative to nearby physical objects. When authoring your content, create it in such a way that the root pivot of your object is centered at this virtual hole. (If your object's pivot is deep in the wall, its slight tweaks in position and rotation will be much more noticeable, and the hole may not look very stable.)

Occluding everything but the virtual hole

There are a variety of ways to selectively block the view to what is hidden in your walls. The simplest one takes advantage of the fact that HoloLens uses an additive display, which means that fully black objects appear invisible. You can do this in Unity without doing any special shader or material tricks—just create a black material and assign it to an object that boxes in your content. If you don't feel like doing 3D modeling, just use a handful of default Quad objects and overlap them slightly. There are a number of drawbacks to this approach, but it is the fastest way to get something working, and getting a low-fidelity proof of concept working is great, even if you suspect you might want to refactor it later.

One major drawback to the above "black box" approach is that it doesn't photograph well. While your effect might look perfect through the display of HoloLens, any screenshots you take will show a large black object instead of what remains of your wall or floor. The reason for this is that the physical hardware and screenshots composite holograms and reality differently. Let's detour for a moment into some fake math...

Fake math alert! These numbers and formulas are meant to illustrate a point, not to be any sort of accurate metric!

What you see through HoloLens:

```
( Reality * darkening_amount ) + Holograms
```

What you see in screenshots and video:

```
( Reality * ( 1 - hologram_alpha ) ) + Holograms * hologram_alpha
```

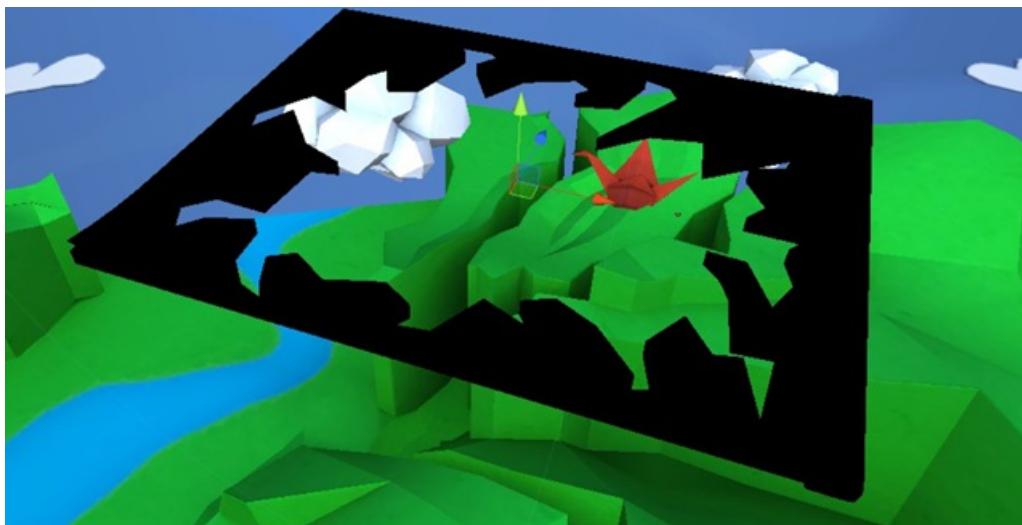
In English: What you see through HoloLens is a simple combination of darkened reality (like through sunglasses) and whatever holograms the app wants to show. But when you take a screenshot, the camera's image is blended with the app's holograms according to the per-pixel transparency value.

One way to get around this is to change the "black box" material to only write to the depth buffer, and sort with all the other opaque materials. For an example of this, check out the [WindowOcclusion.shader file in the MixedRealityToolkit on GitHub](#). The relevant lines are copied here:

```
"RenderType" = "Opaque"  
"Queue" = "Geometry"  
ColorMask 0
```

(Note the "Offset 50, 100" line is to deal with unrelated issues, so it'd probably make sense to leave that out.)

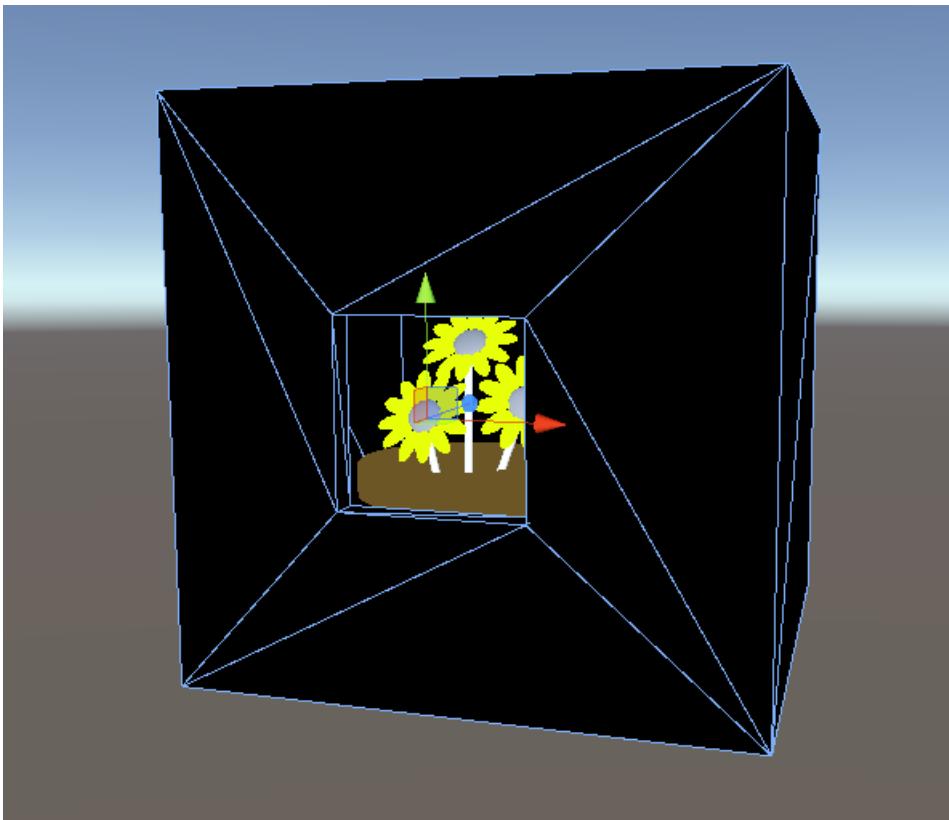
Implementing an invisible occlusion material like that will let your app draw a box that looks correct in the display and in mixed-reality screenshots. For bonus points, you can try to improve the performance of that box even further by doing clever things to draw even fewer invisible pixels, but that can really get into the weeds and usually won't be necessary.



Here is the secret underworld from [MR Basics 101](#) as Unity draws it, except for the outer parts of the occluding box. Note that the pivot for the underworld is at the center of the box, which helps keep the hole as stable as possible relative to your actual floor.

Do it yourself

Have a HoloLens and want to try out the effect for yourself? The easiest thing you can do (no coding required) is to install the free 3D Viewer app and then load the [download the.fbx file I've provided on GitHub](#) to view a flower pot model in your room. Load it on HoloLens, and you can see the illusion at work. When you're in front of the model, you can only see into the small hole—everything else is invisible. Look at the model from any other side and it disappears entirely. Use the movement, rotation, and scale controls of 3D Viewer to position the virtual hole against any vertical surface you can think of to generate some ideas!



Viewing this model in your Unity editor will show a large black box around the flowerpot. On HoloLens, the box disappears, giving way to a magic window effect.

If you want to build an app that uses this technique, check out the [MR Basics 101 tutorial](#) in the [Mixed Reality Academy](#). Chapter 7 ends with an explosion in your floor that reveals a hidden underworld (as pictured above). Who said tutorials had to be boring?

Here are some ideas of where you can take this idea next:

- Think of ways to make the content inside the virtual hole interactive. Letting your users have some impact beyond their walls can really improve the sense of wonder that this trick can provide.
- Think of ways to see through objects back to known areas. For example, how can you put a holographic hole in your coffee table and see your floor beneath it?

About the author



Eric Rehmeyer

Senior Software Engineer @Microsoft

See also

- [MR Basics 101: Complete project with device](#)
- [Coordinate systems](#)
- [Spatial anchors](#)
- [Spatial mapping](#)

Case study - My first year on the HoloLens design team

11/6/2018 • 8 minutes to read • [Edit Online](#)

My journey from a 2D flatland to the 3D world started when I joined the HoloLens design team in January, 2016. Before joining the team, I had very little experience in 3D design. It was like the Chinese proverb about a journey of a thousand miles beginning with a single step, except in my case that first step was a leap!



Taking the leap from 2D to 3D

"I felt as though I had jumped into the driver's seat without knowing how to drive the car. I was overwhelmed and scared, yet very focused."

— Hae Jin Lee

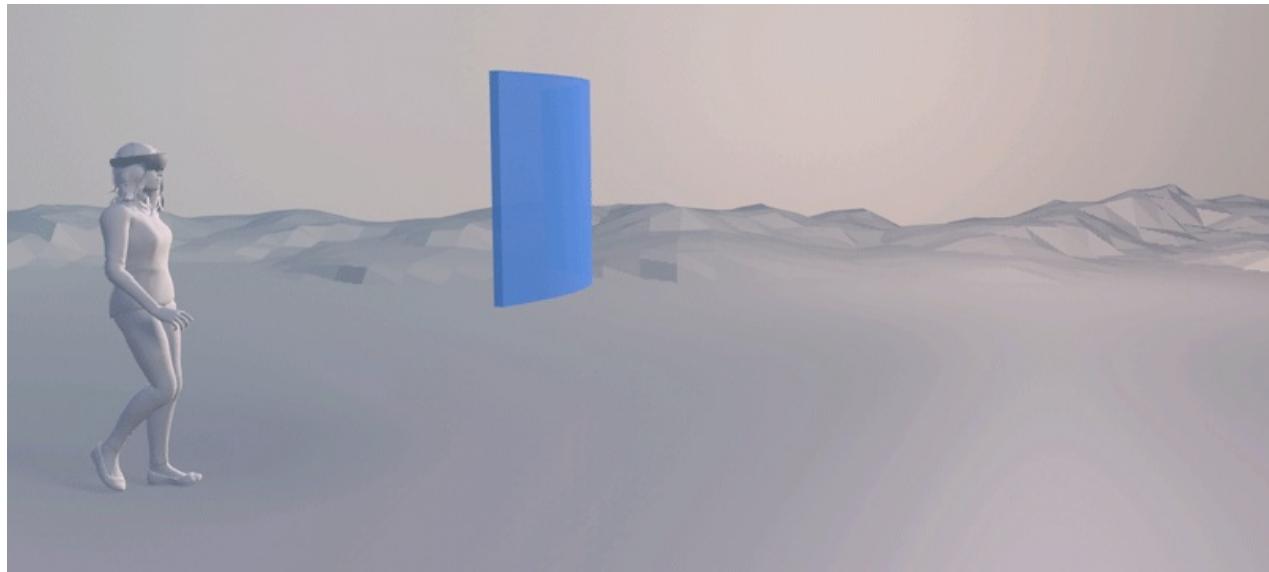
During the past year, I picked up skills and knowledge as fast as I could, but I still have a lot to learn. Here, I've written up 4 observations with a video tutorial documenting my transition from a 2D to 3D interaction designer. I hope my experience will inspire other designers to take the leap to 3D.

Good-bye frame. Hello spatial / diegetic UI

Whenever I designed posters, magazines, websites, or app screens, a defined frame (usually a rectangle) was a constant for every problem. Unless you are reading this post in a HoloLens or other VR device, you are *looking at this from the outside* through 2D screen safely guarded within a frame. Content is external to you. However, Mixed Reality headset *eliminates the frame*, so you are within the content space, looking and walking through the content from inside-out.

I understood this conceptually, but in the beginning I made the mistake of simply transferring 2D thinking into 3D space. That obviously didn't work well because 3D space has its own unique properties such as a view change (based on user's head movement) and [different requirement for user comfort](#) (based on the properties of the devices and the humans using them). For example, in a 2D UI design space, locking UI elements into the corner of a screen is a very common pattern, but this HUD (Head Up Display) style UI does not feel natural in MR/VR experiences; it hinders user's immersion into the space and causes user discomfort. It's like having an annoying

dust particle on your glasses that you are dying to get rid of. Over time, I learned that it feels more natural to position content in 3D space and add body-locked behavior that makes the content follow the user at a relative fixed distance.



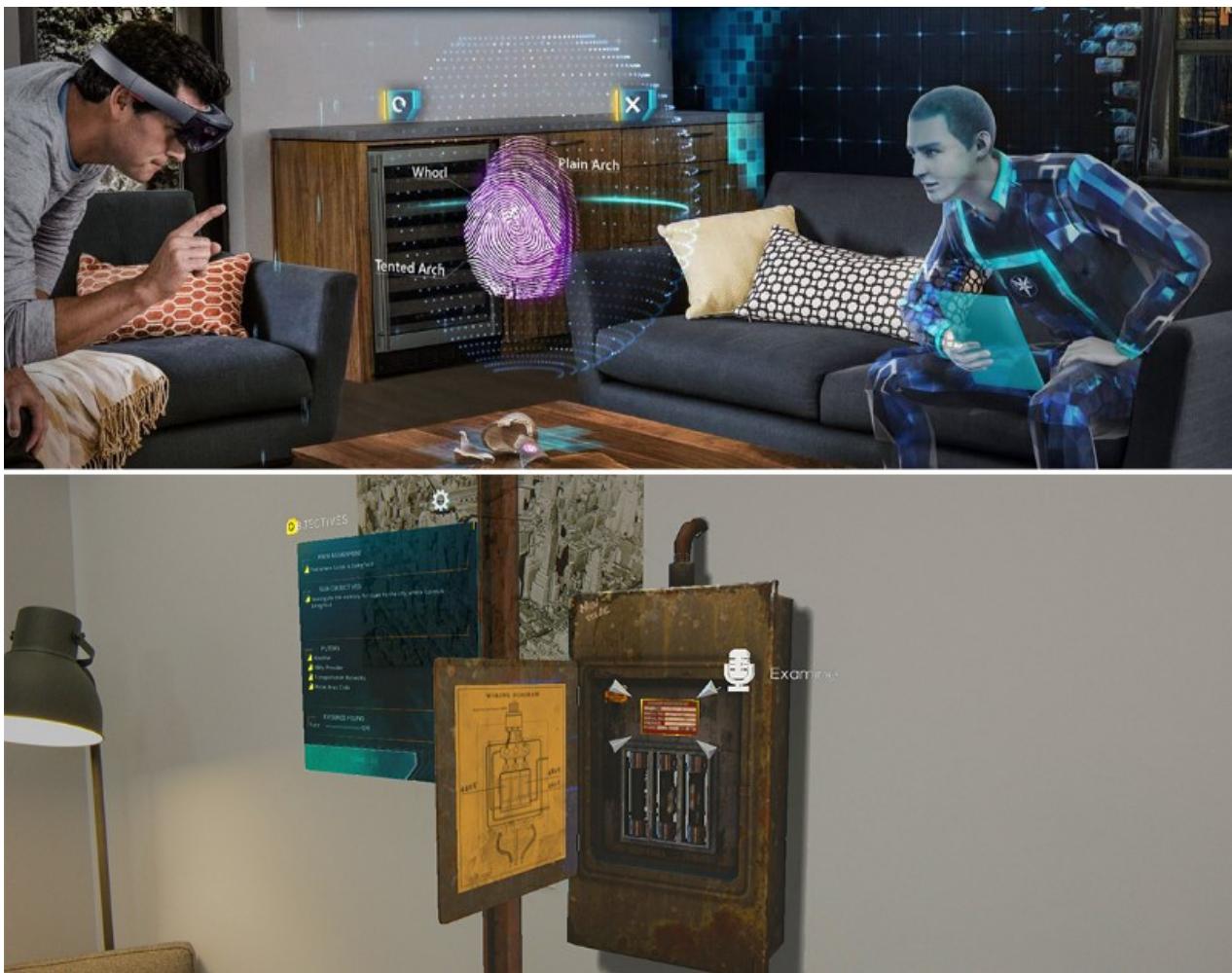
Body-locked



World-locked

Fragments: An example of great Diegetic UI

[Fragments](#), a first-person crime thriller developed by [Asobo Studio](#) for HoloLens demonstrates a great Diegetic UI. In this game, the user becomes a main character, a detective who tries to solve a mystery. The pivotal clues to solve this mystery get sprinkled in the user's physical room and are often times embedded inside a fictional object rather than existing on their own. This diegetic UI tends to be less discoverable than body-locked UI, so the Asobo team cleverly used many cues including virtual characters' gaze direction, sound, light, and guides (e.g., arrow pointing the location of the clue) to grab user's attention.



Fragments - Diegetic UI examples

Observations about diegetic UI

Spatial UI (both body-locked and world-locked) and diegetic UI have their own strengths and weaknesses. I encourage designers to try out as many MR/VR apps as possible, and to develop their own understanding and sensibility for various UI positioning methods.

The return of skeuomorphism and magical interaction

Skeuomorphism, a digital interface that mimics the shape of real world objects has been “uncool” for the last 5–7 years in the design industry. When Apple finally gave way to flat design in iOS 7, it seemed like Skeuomorphism was finally dead as an interface design methodology. But then, a new medium, MR/VR headset arrived to the market and it seems like Skeuomorphism returned again. :)

Job Simulator: An example of skeuomorphic VR design

[Job Simulator](#), a whimsical game developed by [Owlchemy Labs](#) is one of the most popular example for skeuomorphic VR design. Within this game, players are transported into future where robots replace humans and humans visit a museum to experience what it feels like to perform mundane tasks at one of four different jobs: Auto Mechanic, Gourmet Chef, Store Clerk, or Office Worker.

The benefit of Skeuomorphism is clear. Familiar environments and objects within this game help new VR users feel more comfortable and present in virtual space. It also makes them feel like they are in control by associating familiar knowledge and behaviors with objects and their corresponding physical reactions. For example, to drink a cup of coffee, people simply need to walk to the coffee machine, press a button, grab the cup handle and tilt it towards their mouth as they would do in the real world.



Job Simulator

Because MR/VR is still a developing medium, using a certain degree of skeuomorphism is necessary to demystify MR/VR technology and to introduce it to larger audiences around the world. Additionally, using skeuomorphism or realistic representation could be beneficial for specific types of applications like surgery or flight simulation. Since the goal of these apps is to develop and refine specific skills that can be directly applied in the real world, the closer the simulation is to the real world, the more transferable the knowledge is.

Remember that skeuomorphism is only one approach. The potential of the MR/VR world is far greater than that, and designers should strive to create magical hyper-natural interactions — new affordances that are uniquely possible in MR/VR world. As a start, consider adding magical powers to ordinary objects to enable users to fulfill their fundamental desires—including teleportation and omniscience.



Doraemon's magical door (left) and ruby slippers(right)

Observations about skeuomorphism in VR

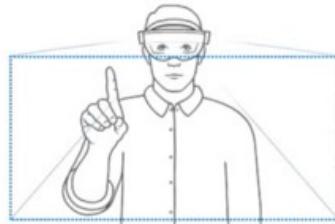
From “Anywhere door” in Doraemon, “Ruby Slippers” in The Wizard of Oz to “Maurader’s map” in Harry Potter, examples of ordinary objects with magical power abound in popular fiction. These magical objects help us visualize a connection between the real-world and the fantastic, between what is and what could be. Keep in mind that when designing the magical or surreal object one needs to strike a balance between functionality and entertainment. Beware of the temptation to create something purely magical just for novelty’s sake.

Understanding different input methods

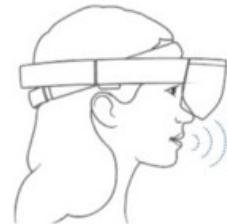
When I designed for the 2D medium, I had to focus on touch, mouse, and keyboard interactions for inputs. In the MR/VR design space, our body becomes the interface and users are able to use a broader selection of input methods: including speech, gaze, gesture, [6-dof controllers](#), and gloves that afford more intuitive and direct connection with virtual objects.



Gaze



Gesture



Voice

Available inputs in HoloLens

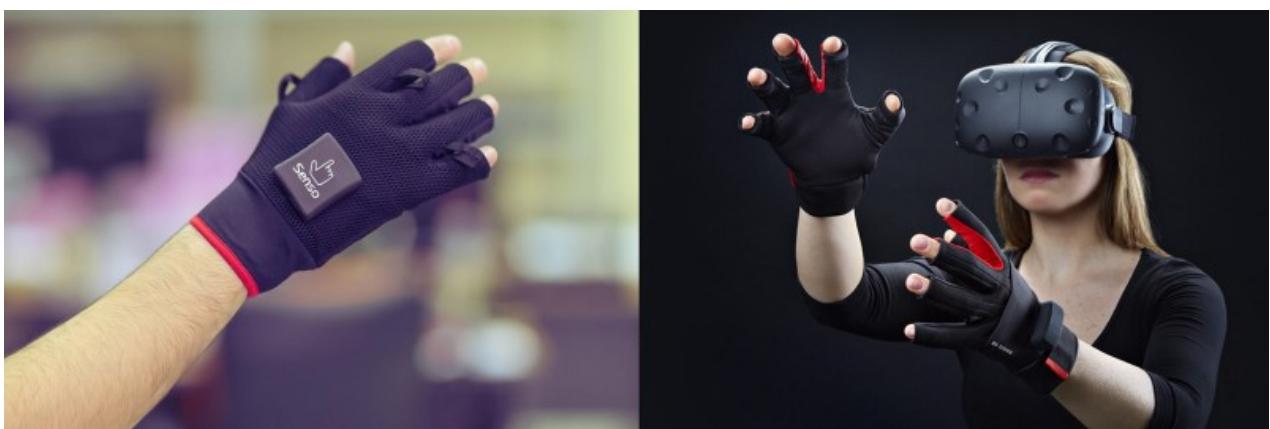
"Everything is best for something, and worst for something else."

— Bill Buxton

For example, gesture input using bare hand and camera sensors on an HMD device frees users hand from holding controllers or wearing sweaty gloves, but frequent use can cause physical fatigue (a.k.a gorilla arm). Also, users have to keep their hands within the line of sight; if the camera cannot see the hands, the hands cannot be used.

Speech input is good at traversing complex tasks because it allows users to cut through nested menus with one command (e.g., "Show me the movies made by Laika studio.") and also very economical when coupled with other modality (e.g., "Face me" command orients the hologram a user is looking at towards the user). However, speech input may not work well in noisy environment or may not appropriate in a very quiet space.

Besides gesture and speech, hand-held tracked controllers (e.g., Oculus touch, Vive, etc.) are very popular input methods because they are easy to use, accurate, leverage people's [proprioception](#), and provide passive haptic cues. However, these benefits come at the cost of not being able to be bare-hands and use full finger tracking.



Senso (Left) and Manus VR (Right)

While not as popular as controllers, gloves are gaining momentum again thanks to the MR/VR wave. Most recently, brain/mind input have started to gain traction as an interface for virtual environments by integrating EEG or EMG sensor to headset (e.g., [MindMaze VR](#)).

Observations about input methods

These are just a sample of input devices available in the market for MR/VR. They will continue to proliferate until the industry matures and agrees upon best practices. Until then, designers should remain aware of new input

devices and be well-versed in the specific input methods for their particular project. Designers need to look for creative solutions inside of limitations, while also playing to a device's strengths.

Sketch the scene and test in the headset

When I worked in 2D, I mostly sketched just the content. However, in mixed reality space that wasn't sufficient. I had to sketch out the entire scene to better imagine the relationships between the user and virtual objects. To help my spatial thinking, I started to sketch scenes in [Cinema 4D](#) and sometimes create simple assets for prototyping in [Maya](#). I had never used either program before joining the HoloLens team and I am still a newbie, but working with these 3D programs definitely helped me get comfortable with new terminology, such as [shader](#) and [IK \(inverse kinematics\)](#).

"No matter how closely I sketched out the scene in 3D, the actual experience in headset was almost never the same as the sketch. That's why it's important to test out the scene in the target headsets." — **Hae Jin Lee**

For HoloLens prototyping, I tried out all the tutorials at [Mixed Reality Academy](#) to start. Then I began to play with [HoloToolkit.Unity](#) that Microsoft provides to developers to accelerate development of holographic applications. When I got stuck with something, I posted my question to [HoloLens Question & Answer Forum](#).

After acquiring basic understanding of HoloLens prototyping, I wanted to empower other non-coders to prototype on their own. So I made a video tutorial that teaches how to develop a simple projectile using HoloLens. I briefly explain the basic concepts, so even if you have zero experience in HoloLens development, you should be able to follow along.

I made this simple tutorial for non-programmers like myself.

For VR prototyping, I took courses at [VR Dev School](#) and also took [3D Content Creation for Virtual Reality](#) at Lynda.com. VR Dev school provided me more in depth knowledge in coding and the Lynda course offered me a nice short introduction to creating assets for VR.

Take the leap

A year ago, I felt like all of this was a bit overwhelming. Now I can tell you that it was 100% worth the effort. MR/VR is still very young medium and there are so many interesting possibilities waiting to be realized. I feel inspired and fortunate to be able to play one small part in designing the future. I hope you will join me on the journey into 3D space!

About the author



Hae Jin Lee

UX Designer @Microsoft

Case study - Representing humans in mixed reality

11/6/2018 • 6 minutes to read • [Edit Online](#)

James Turrell designs with light. Stepping into his work blurs one's sense of depth and focus. Walls seem both close and infinite, brightness gives way to shadows. Unfamiliar perceptions designed by carefully balancing the light's color and diffusion. [Turrell describes these sensations](#) as '*feeling with your eyes*', a way of extending one's understanding of reality. Fantastic worlds, like the ones Turrell imagines, are powerful tools to exploit our senses, not unlike the immersive environments of mixed reality today.



How do you represent complex real-world environments in mixed reality?

Representing Turrell's work in an immersive experience makes for a compelling challenge. Lighting, scale, and spatial audio present opportunities to represent his work. While the exhibit's geometric surroundings would require relatively simple 3D modeling, they are secondary to the artist's focus: the light's impact on the senses.

Turrell's stark, surreal minimalism is the hallmark of his work, but what if we wanted to represent an exhibit with more complex materials in mixed reality?



In 2013, the artist Ai Weiwei unveiled [a tangling work of art](#) featuring 886 antique stools at the Venice Biennale. Each wooden stool came from an era where Chinese craftsmanship was highly valued, where these stools would have been passed down between generations. The stools themselves — the intricacies of the wood, the precision of the pieces, their careful placement — are critical to Ai's commentary on modern culture.

The antique stools deliver the artist's message through their authenticity. Their realistic representation is critical to the experience, creating a technical challenge: Sculpting each of the 886 stools by hand would be enormously exhaustive and expensive. How long would it take to model and position? How would you maintain the authenticity of the material? Recreating these objects from scratch becomes, in many ways, an interpretation of the artwork itself. How can you preserve the artist's intent?

Methods of capturing mixed reality assets

The alternative to creating something from scratch is capturing the real thing. Through an ever-advancing set of capture methods, we can develop authentic representations of each of the core asset types found in mixed reality (environments, objects, and people).

The broad categories range from well-established 2D video to the newest forms of volumetric video. In the case of Ai Weiwei's exhibit, scanning (often referred to by its fundamental technique, photogrammetry) could be employed during the creation of the exhibit, scanning each of the stools themselves. 360° photo and video capture is another method for virtualizing the experience utilizing a high-quality omni-directional camera positioned throughout the exhibit. With these techniques, one begins to understand the sense of scale, ideally with enough detail to see each piece's craftsmanship. All this while existing in a digital form that allows for new vistas and perspectives not possible in reality.



What kind of opportunities emerge when we cannot only create fantastic elements, but utilize the most realistic

captures of environments, objects, and people in mixed reality? Exploring the overlap between these methods help illuminate where the medium is headed.

For environments and objects, 360° imaging software is evolving to include elements of photogrammetry. Isolating depth information from scenes, advanced 360° videos help alleviate the feeling of having your head stuck in a fishbowl when looking around a virtual scene.

For people, new methods are emerging that combine and extend motion capture and scanning: Motion capture has been foundational to bringing detailed human movement to visual effects and cinematic characters, while scanning has advanced to capture detailed human visuals like faces and hands. With advancements in rendering technology, a new method called volumetric video builds off these techniques, combining visual and depth information, to create the next generation of 3D human captures.

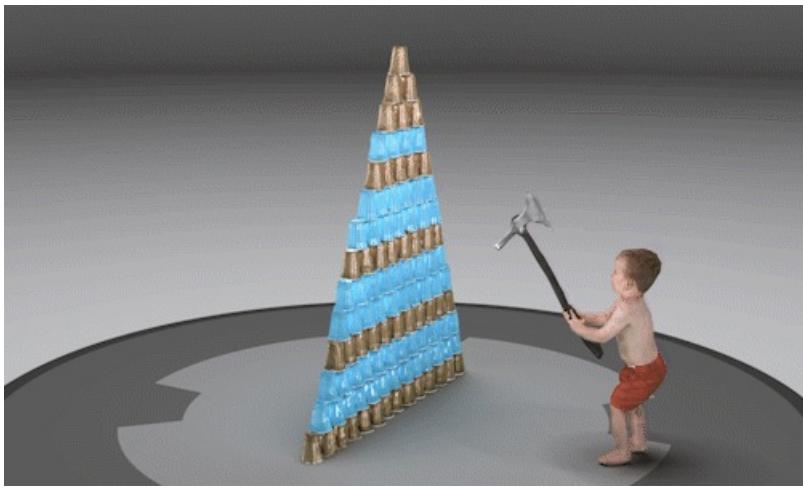
Volumetric video and the pursuit of authentic human capture

Humans are central to storytelling — in the most literal sense: a human speaking, performing, or as the story's subject. Some of the most immersive and eye-opening moments of today's early immersive experiences are social. From sharing a mixed reality experience together in your living room, to seeing your friends in unbelievable new environments. The human element makes even the most fantastic reality, a reality.



Avatars in immersive experiences enable a new kind of embodiment in storytelling. The latest apps are rethinking the concept of virtual body ownership and setting up a generational leap in eliminating the distance between people. Companies like [Mindshow](#) are developing creative tools that leverage avatars, letting users take on entirely new personas and characters. Others are exploring [methods of artistic expression](#), a potentially limitless creative opportunity to explore the nature (and necessity) of human-like attributes. Today, this absence of realism helps avoid the [uncanny valley of human likeness](#) along with a host of technical issues for everyday developers. For these reasons (and more) it is very likely that non-realistic avatars will become the default for the foreseeable future. And yet, while realism poses an enormous challenge for mixed reality, *there are key scenarios that require authentic representation of humans in 3D space*.

At Microsoft, a small team borne out of Microsoft Research has spent the past several years developing a method for capturing humans through a form of volumetric video. The process today is similar to video production: rather than applying movement to a sculpted asset it is a full, 3D recording. The performance and the image are captured in real-time — it's not the work of an artist, it's an authentic representation. And while the technology is just beginning to expand into commercial applications, the implications of volumetric video are critical to [Microsoft's vision of More Personal Computing](#).



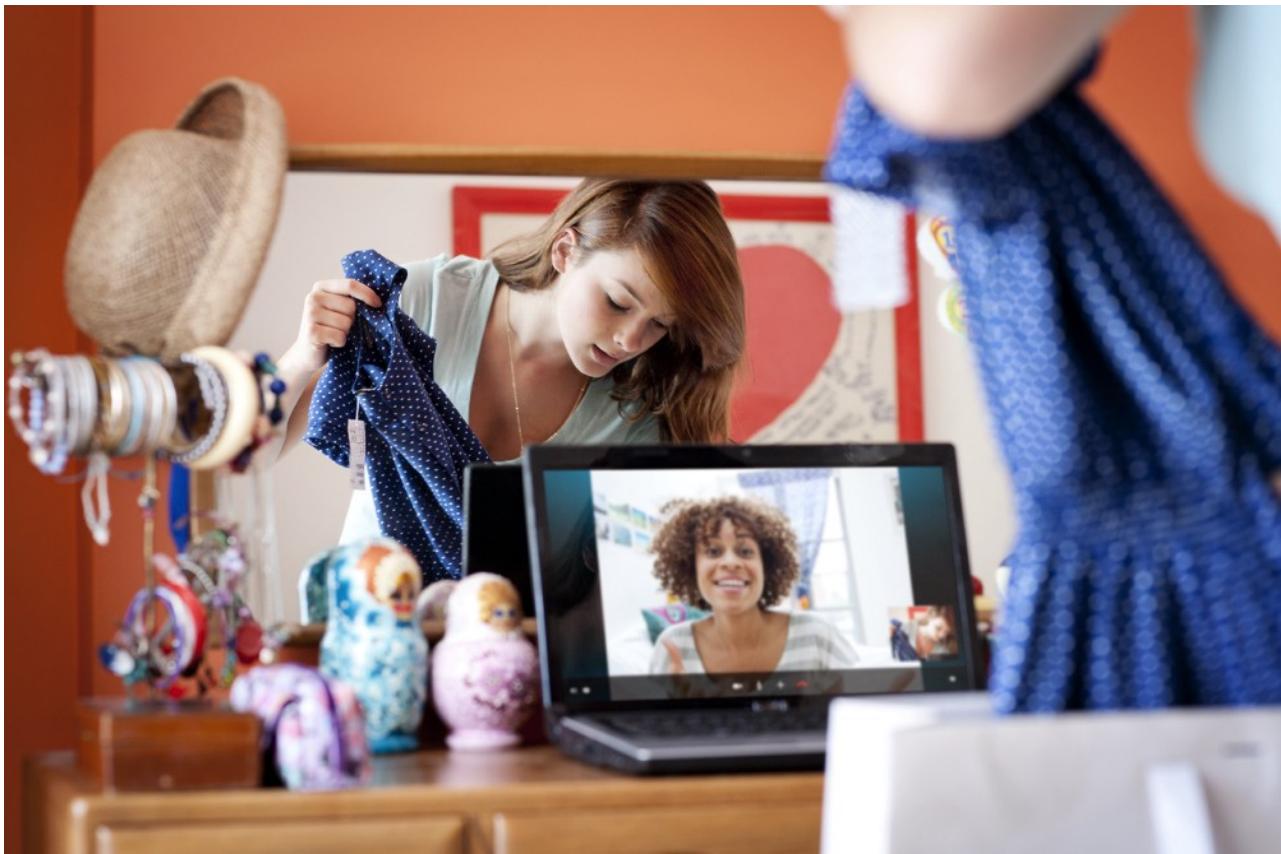
Authentic human capture unlocks new unique categories of experiences in mixed reality. Seeing someone you recognize, whether it's a celebrity, a colleague, or a loved one, creates a depth of intimacy never before possible in a digital medium. Their face, their expressions, the nuance in their movements are all part of who they are. What opportunities unlock when we can capture these human qualities in 3D space?

Today the team is pushing the bounds of volumetric video by focusing on sectors like entertainment and education: [Actiongram](#) features creative characters and [celebrities](#) to create mixed reality stories. [Destination: Mars exhibit](#), now at NASA's Kennedy Space Center, features a volumetric video of legendary astronaut Buzz Aldrin. The experience allows visitors to walk around the surface of Mars with Buzz as he introduces the pursuit of human colonization on Mars.

Humans are fundamental to mixed reality

Designing ways to make these videos seem natural poses a challenge but one in which the team sees enormous potential. And these opportunities will expand as the technology becomes more accessible and moves from recordings to real-time capture.

[Holoportation](#) is a research effort that builds upon the same fundamental technology, authentically capturing visual and depth information, and rendering the result in real-time. The team is exploring what the power of realistic human representation means for the future of conversations and shared experiences. What happens when a three-dimensional capture of someone, from anywhere in the world, can be added into your environment?



From layering a new level of immersion onto everyday apps like Skype, to radically reshaping the concept of digital meetings and business travel — volumetric video opens unique scenarios: A specialist virtually training doctors on a far-away continent or digital friends sitting on the couches and chairs in your living room. Adding authentic human representations to mixed reality experiences will radically reshape the concept of digital meetings and business travel.

Just as the abstract art of James Turrell and the critical realism of Ai Weiwei offer their own unique technical challenges, so do the methods to represent humans as creative avatars and realistic captures. One cannot be ignored in light of the other and exploring the potential of each will help us understand human interaction in this new space.

About the author



Mark Vitazko

UX Designer @Microsoft

Case study - Scaling Datascape across devices with different performance

11/6/2018 • 13 minutes to read • [Edit Online](#)

Datascape is a Windows Mixed Reality application developed internally at Microsoft where we focused on displaying weather data on top of terrain data. The application explores the unique insights users gain from discovering data in mixed reality by surrounding the user with holographic data visualization.

For Datascape we wanted to target a variety of platforms with different hardware capabilities ranging from Microsoft HoloLens to Windows Mixed Reality immersive headsets, and from lower-powered PCs to the very latest PCs with high-end GPU. The main challenge was rendering our scene in a visually appealing matter on devices with wildly different graphics capabilities while executing at a high framerate.

This case study will walk through the process and techniques used to create some of our more GPU-intensive systems, describing the problems we encountered and how we overcame them.

Transparency and overdraw

Our main rendering struggles dealt with transparency, since transparency can be expensive on a GPU.

Solid geometry can be rendered front to back while writing to the depth buffer, stopping any future pixels located behind that pixel from being discarded. This prevents hidden pixels from executing the pixel shader, speeding up the process significantly. If geometry is sorted optimally, each pixel on the screen will be drawn only once.

Transparent geometry needs to be sorted back to front and relies on blending the output of the pixel shader to the current pixel on the screen. This can result in each pixel on the screen being drawn to multiple times per frame, referred to as overdraw.

For HoloLens and mainstream PCs, the screen can only be filled a handful of times, making transparent rendering problematic.

Introduction to Datascape scene components

We had three major components to our scene; **the UI**, **the map**, and **the weather**. We knew early on that our weather effects would require all the GPU time it could get, so we purposely designed the UI and terrain in a way that would reduce any overdraw.

We reworked the UI several times to minimize the amount of overdraw it would produce. We erred on the side of more complex geometry rather than overlaying transparent art on top of each other for components like glowing buttons and map overviews.

For the map, we used a custom shader that would strip out standard Unity features such as shadows and complex lighting, replacing them with a simple single sun lighting model and a custom fog calculation. This produced a simple pixel shader and free up GPU cycles.

We managed to get both the UI and the map to render at budget where we did not need any changes to them depending on the hardware; however, the weather visualization, in particular the cloud rendering, proved to be more of a challenge!

Background on cloud data

Our cloud data was downloaded from NOAA servers (<http://nomads.ncep.noaa.gov/>) and came to us in three

distinct 2D layers, each with the top and bottom height of the cloud, as well as the density of the cloud for each cell of the grid. The data got processed into a cloud info texture where each component was stored in the red, green, and blue component of the texture for easy access on the GPU.

Geometry clouds

To make sure our lower-powered machines could render our clouds we decided to start with an approach that would use solid geometry to minimize overdraw.

We first tried producing clouds by generating a solid heightmap mesh for each layer using the radius of the cloud info texture per vertex to generate the shape. We used a geometry shader to produce the vertices both at the top and the bottom of the cloud generating solid cloud shapes. We used the density value from the texture to color the cloud with darker colors for more dense clouds.

Shader for creating the vertices:

```
v2g vert (appdata v)
{
    v2g o;
    o.height = tex2Dlod(_MainTex, float4(v.uv, 0, 0)).x;
    o.vertex = v.vertex;
    return o;
}

g2f GetOutput(v2g input, float heightDirection)
{
    g2f ret;
    float4 newBaseVert = input.vertex;
    newBaseVert.y += input.height * heightDirection * _HeighthScale;
    ret.vertex = UnityObjectToClipPos(newBaseVert);
    ret.height = input.height;
    return ret;
}

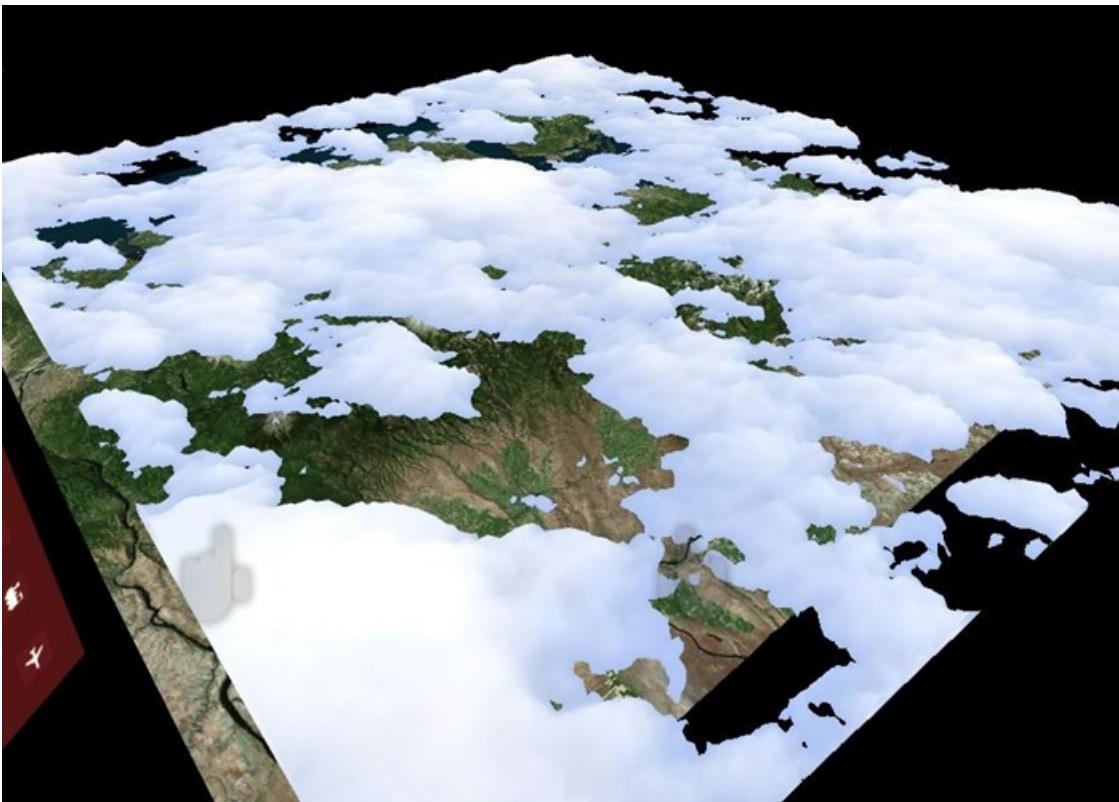
[maxvertexcount(6)]
void geo(triangle v2g p[3], inout TriangleStream<g2f> triStream)
{
    float heightTotal = p[0].height + p[1].height + p[2].height;
    if (heightTotal > 0)
    {
        triStream.Append(GetOutput(p[0], 1));
        triStream.Append(GetOutput(p[1], 1));
        triStream.Append(GetOutput(p[2], 1));

        triStream.RestartStrip();

        triStream.Append(GetOutput(p[2], -1));
        triStream.Append(GetOutput(p[1], -1));
        triStream.Append(GetOutput(p[0], -1));
    }
}
fixed4 frag (g2f i) : SV_Target
{
    clip(i.height - 0.1f);

    float3 finalColor = lerp(_LowColor, _HighColor, i.height);
    return float4(finalColor, 1);
}
```

We introduced a small noise pattern to get more detail on top of the real data. To produce round cloud edges, we clipped the pixels in the pixel shader when the interpolated radius value hit a threshold to discard near-zero values.



Since the clouds are solid geometry, they can be rendered before the terrain to hide any expensive map pixels underneath to further improve framerate. This solution ran well on all graphics cards from min-spec to high-end graphics cards, as well as on HoloLens, because of the solid geometry rendering approach.

Solid particle clouds

We now had a backup solution that produced a decent representation of our cloud data, but was a bit lackluster in the "wow" factor and did not convey the volumetric feel that we wanted for our high-end machines.

Our next step was creating the clouds by representing them with approximately 100,000 particles to produce a more organic and volumetric look.

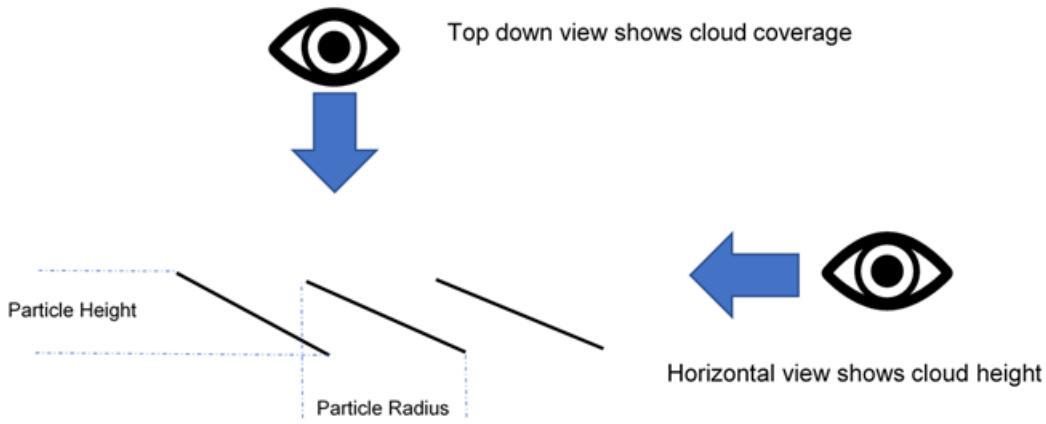
If particles stay solid and sort front-to-back, we can still benefit from depth buffer culling of the pixels behind previously rendered particles, reducing the overdraw. Also, with a particle-based solution, we can alter the amount of particles used to target different hardware. However, all pixels still need to be depth tested, which results in some additional overhead.

First, we created particle positions around the center point of the experience at startup. We distributed the particles more densely around the center and less so in the distance. We pre-sorted all particles from the center to the back so that the closest particles would render first.

A compute shader would sample the cloud info texture to position each particle at a correct height and color it based on the density.

We used *DrawProcedural* to render a quad per particle allowing the particle data to stay on the GPU at all times.

Each particle contained both a height and a radius. The height was based on the cloud data sampled from the cloud info texture, and the radius was based on the initial distribution where it would be calculated to store the horizontal distance to its closest neighbor. The quads would use this data to orient itself angled by the height so that when users look at it horizontally, the height would be shown, and when users looked at it top-down, the area between its neighbors would be covered.



Shader code showing the distribution:

```

ComputeBuffer cloudPointBuffer = new ComputeBuffer(6, quadPointsStride);
cloudPointBuffer.SetData(new[]
{
    new Vector2(-.5f, .5f),
    new Vector2(.5f, .5f),
    new Vector2(.5f, -.5f),
    new Vector2(.5f, -.5f),
    new Vector2(-.5f, -.5f),
    new Vector2(-.5f, .5f)
});

StructuredBuffer<float2> quadPoints;
StructuredBuffer<float3> particlePositions;
v2f vert(uint id : SV_VertexID, uint inst : SV_InstanceID)
{
    // Find the center of the quad, from local to world space
    float4 centerPoint = mul(unity_ObjectToWorld, float4(particlePositions[inst], 1));

    // Calculate y offset for each quad point
    float3 cameraForward = normalize(centerPoint - _WorldSpaceCameraPos);
    float y = dot(quadPoints[id].xy, cameraForward.xz);

    // Read out the particle data
    float radius = ...;
    float height = ...;

    // Set the position of the vert
    float4 finalPos = centerPoint + float4(quadPoints[id].x, y * height, quadPoints[id].y, 0) * radius;
    o.pos = mul(UNITY_MATRIX_VP, float4(finalPos.xyz, 1));
    o.uv = quadPoints[id].xy + 0.5;

    return o;
}

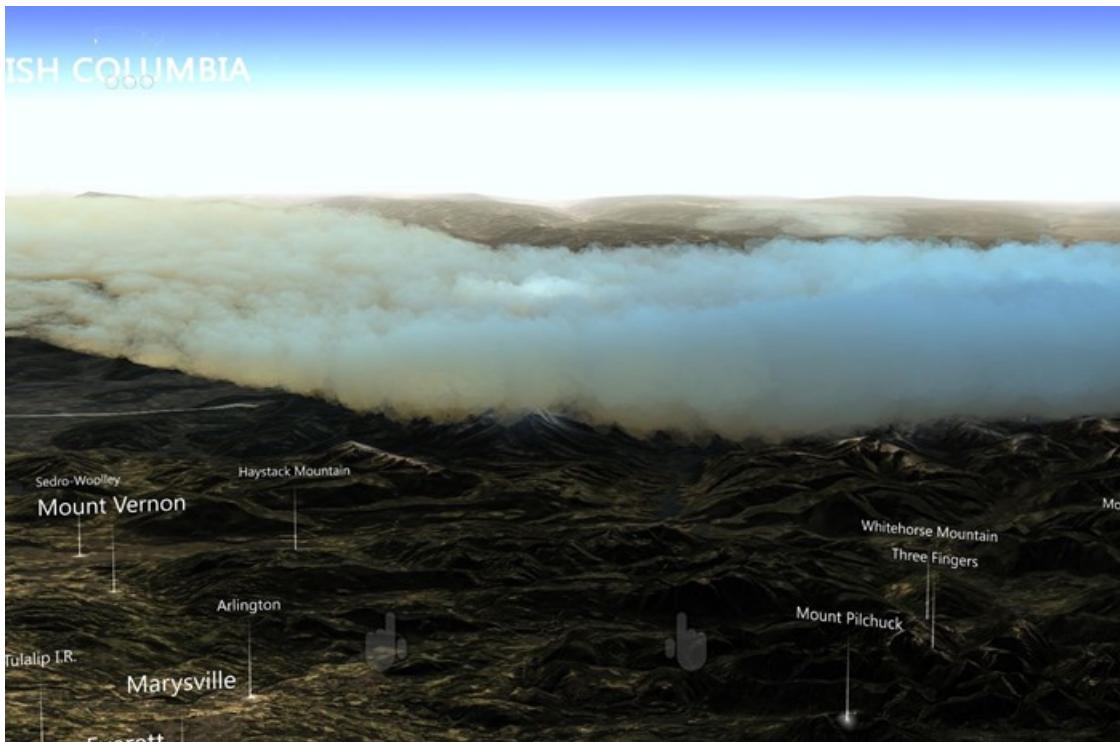
```

Since we sort the particles front-to-back and we still used a solid style shader to clip (not blend) transparent pixels, this technique handles a surprising amount of particles, avoiding costly over-draw even on the lower-powered machines.

Transparent particle clouds

The solid particles provided a good organic feel to the shape of the clouds but still needed something to sell the fluffiness of clouds. We decided to try a custom solution for the high-end graphics cards where we can introduce transparency.

To do this we simply switched the initial sorting order of the particles and changed the shader to use the textures alpha.



It looked great but proved to be too heavy for even the toughest machines since it would result in rendering each pixel on the screen hundreds of times!

Render off-screen with lower resolution

To reduce the number of pixels rendered by the clouds, we started rendering them in a quarter resolution buffer (compared to the screen) and stretching the end result back up onto the screen after all the particles had been drawn. This gave us roughly a 4x speedup, but came with a couple of caveats.

Code for rendering off-screen:

```
cloudBlendingCommand = new CommandBuffer();
Camera.main.AddCommandBuffer(whenToComposite, cloudBlendingCommand);

cloudCamera.CopyFrom(Camera.main);
cloudCamera.rect = new Rect(0, 0, 1, 1);    //Adaptive rendering can set the main camera to a smaller rect
cloudCamera.clearFlags = CameraClearFlags.Color;
cloudCamera.backgroundColor = new Color(0, 0, 0, 1);

currentCloudTexture = RenderTexture.GetTemporary(Camera.main.pixelWidth / 2, Camera.main.pixelHeight / 2, 0);
cloudCamera.targetTexture = currentCloudTexture;

// Render clouds to the offscreen buffer
cloudCamera.Render();
cloudCamera.targetTexture = null;

// Blend low-res clouds to the main target
cloudBlendingCommand.Blit(currentCloudTexture, new
RenderTargetIdentifier(BuiltinRenderTextureType.CurrentActive), blitMaterial);
```

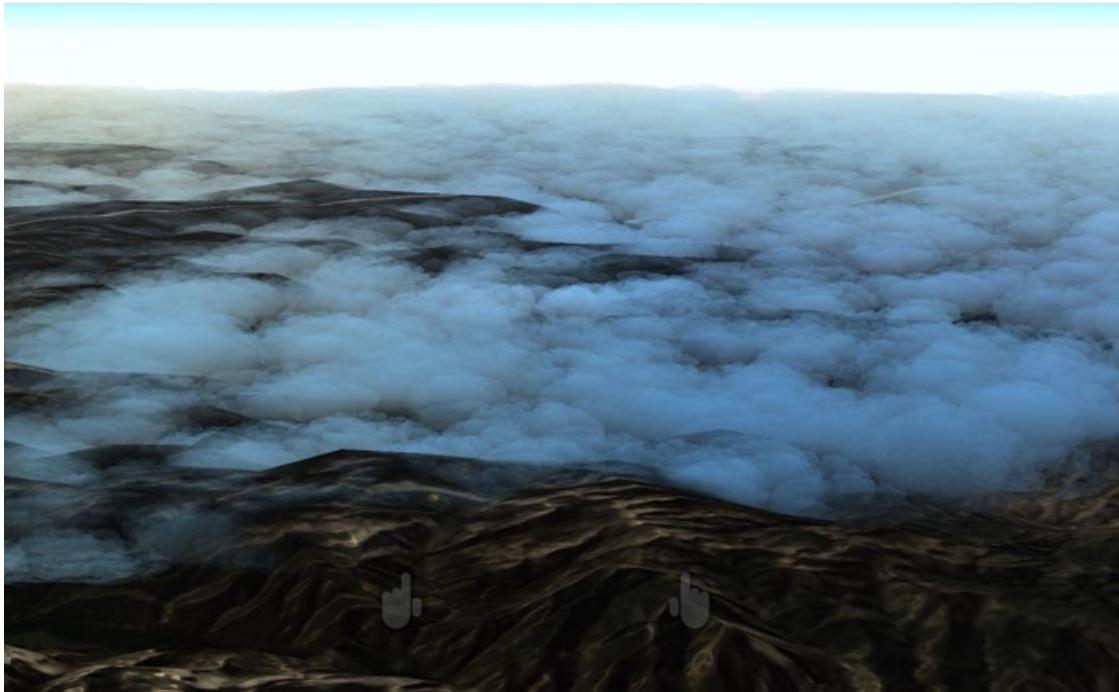
First, when rendering into an off-screen buffer, we lost all depth information from our main scene, resulting in particles behind mountains rendering on top of the mountain.

Second, stretching the buffer also introduced artifacts on the edges of our clouds where the resolution change was noticeable. The next two sections talk about how we resolved these issues.

Particle depth buffer

To make the particles co-exist with the world geometry where a mountain or object could cover particles behind it, we populated the off-screen buffer with a depth buffer containing the geometry of the main scene. To produce such depth buffer, we created a second camera, rendering only the solid geometry and depth of the scene.

We then used the new texture in the pixel shader of the clouds to occlude pixels. We used the same texture to calculate the distance to the geometry behind a cloud pixel. By using that distance and applying it to the alpha of the pixel, we now had the effect of clouds fading out as they get close to terrain, removing any hard cuts where particles and terrain meet.

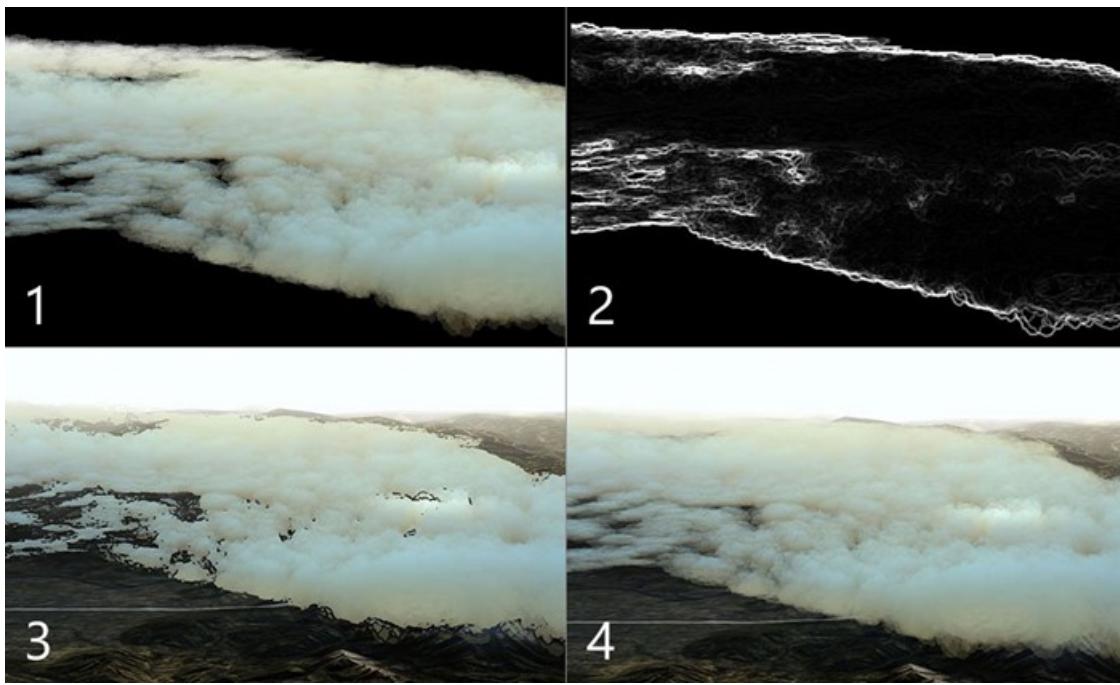


Sharpening the edges

The stretched-up clouds looked almost identical to the normal size clouds at the center of the particles or where they overlapped, but showed some artifacts at the cloud edges. Otherwise sharp edges would appear blurry and alias effects were introduced when the camera moved.

We solved this by running a simple shader on the off-screen buffer to determine where big changes in contrast occurred (1). We put the pixels with big changes into a new stencil buffer (2). We then used the stencil buffer to mask out these high contrast areas when applying the off-screen buffer back to the screen, resulting in holes in and around the clouds (3).

We then rendered all the particles again in full-screen mode, but this time used the stencil buffer to mask out everything but the edges, resulting in a minimal set of pixels touched (4). Since the command buffer was already created for the particles, we simply had to render it again to the new camera.



The end result was sharp edges with cheap center sections of the clouds.

While this was much faster than rendering all particles in full screen, there is still a cost associated with testing a pixel against the stencil buffer, so a massive amount of overdraw still came with a cost.

Culling particles

For our wind effect, we generated long triangle strips in a compute shader, creating many wisps of wind in the world. While the wind effect was not heavy on fill rate due to skinny strips generated, it produced many hundreds of thousands of vertices resulting in a heavy load for the vertex shader.

We introduced append buffers on the compute shader to feed a subset of the wind strips to be drawn. With some simple view frustum culling logic in the compute shader, we could determine if a strip was outside of camera view and prevent it from being added to the push buffer. This reduced the amount of strips significantly, freeing up some needed cycles on the GPU.

Code demonstrating an append buffer:

Compute shader:

```
AppendStructuredBuffer<int> culledParticleIdx;

if (show)
    culledParticleIdx.Append(id.x);
```

C# code:

```

protected void Awake()
{
    // Create an append buffer, setting the maximum size and the contents stride length
    culledParticlesIdxBuffer = new ComputeBuffer(ParticleCount, sizeof(int), ComputeBufferType.Append);

    // Set up Args Buffer for Draw Procedural Indirect
    argsBuffer = new ComputeBuffer(4, sizeof(int), ComputeBufferType.IndirectArguments);
    argsBuffer.SetData(new int[] { DataVertCount, 0, 0, 0 });

}

protected void Update()
{
    // Reset the append buffer, and dispatch the compute shader normally
    culledParticlesIdxBuffer.SetCounterValue(0);

    computer.Dispatch(...)

    // Copy the append buffer count into the args buffer used by the Draw Procedural Indirect call
    ComputeBuffer.CopyCount(culledParticlesIdxBuffer, argsBuffer, dstOffset: 1);
    ribbonRenderCommand.DrawProceduralIndirect(Matrix4x4.identity, renderMaterial, 0, MeshTopology.Triangles,
    dataBuffer);
}

```

We tried using the same technique on the cloud particles, where we would cull them on the compute shader and only push the visible particles to be rendered. This technique actually did not save us much on the GPU since the biggest bottleneck was the amount pixels rendered on the screen, and not the cost of calculating the vertices.

The other problem with this technique was that the append buffer populated in random order due to its parallelized nature of computing the particles, causing the sorted particles to be un-sorted, resulting in flickering cloud particles.

There are techniques to sort the push buffer, but the limited amount of performance gain we got out of culling particles would likely be offset with an additional sort, so we decided to not pursue this optimization.

Adaptive rendering

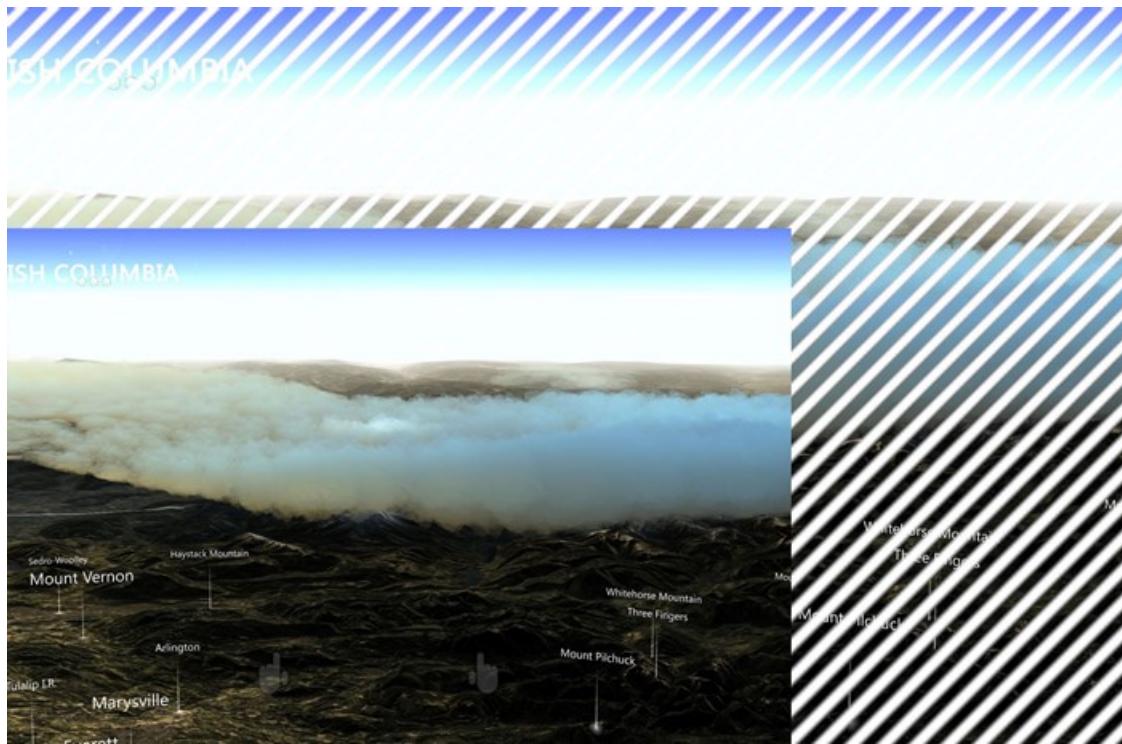
To ensure a steady framerate on an app with varying rendering conditions like a cloudy vs a clear view, we introduced adaptive rendering to our app.

The first step of adaptive rendering is to measure GPU. We did this by inserting custom code into the GPU command buffer at the beginning and the end of a rendered frame, capturing both the left and right eye screen time.

By measuring the time spent rendering and comparing it to our desired refresh-rate we got a sense of how close we were to dropping frames.

When close to dropping frames, we adapt our rendering to make it faster. One simple way of adapting is changing the viewport size of the screen, requiring less pixels to get rendered.

By using `UnityEngine.XR.XRSettings.renderViewportScale` the system shrinks the targeted viewport and automatically stretches the result back up to fit the screen. A small change in scale is barely noticeable on world geometry, and a scale factor of 0.7 requires half the amount of pixels to be rendered.



When we detect that we are about to drop frames we lower the scale by a fixed number, and increase it back when we are running fast enough again.

While we decided what cloud technique to use based on graphics capabilities of the hardware at startup, it is possible to base it on data from the GPU measurement to prevent the system from staying at low resolution for a long time, but this is something we did not have time to explore in Datascape.

Final thoughts

Targeting a variety of hardware is challenging and requires some planning.

We recommend that you start targeting lower-powered machines to get familiar with the problem space and develop a backup solution that will run on all your machines. Design your solution with fill rate in mind, since pixels will be your most precious resource. Target solid geometry over transparency.

With a backup solution, you can then start layering in more complexity for high end machines or maybe just enhance resolution of your backup solution.

Design for worst case scenarios, and maybe consider using adaptive rendering for heavy situations.

About the authors



Robert Ferrese

Software engineer @Microsoft



Dan Andersson

Software engineer @Microsoft

See also

- [Performance recommendations for immersive headset apps](#)

Case study - Spatial sound design for HoloTour

11/6/2018 • 7 minutes to read • [Edit Online](#)

To create a truly immersive 3D virtual tour for Microsoft HoloLens, the panoramic videos and holographic scenery are only part of the formula. Audio designer Jason Syltebo talks about how sound was captured and processed to make you feel like you're actually in each of the locations in HoloTour.

The tech

The beautiful imagery and holographic scenes you see in HoloTour are only one part of creating a believable mixed reality experience. While holograms can only appear visually in front of a user, the [spatial sound](#) feature of HoloLens allows audio to come from all directions, which gives the user a more complete sensory experience.

Spatial sound allows us to give audio cues to indicate a direction in which the user should turn, or to let the user know there are more holograms for them to see within their space. We can also attach a sound directly to a hologram and continually update the direction and distance the hologram is from a user to make it seem as if the sound is coming directly from that object.

With HoloTour, we wanted to take advantage of the spatial sound capabilities of HoloLens to create a 360-degree ambient environment, synchronized with the video to reveal the sonic highlights of specific locations.

Behind the scenes

We created HoloTour experiences of two different locations: Rome and Machu Picchu. To make these tours feel authentic and compelling we wanted to avoid using generic sounds and instead capture audio directly from the locations where we were filming.

Capturing the audio

In our [case study about capturing the visual content for HoloTour](#), we talked about the custom design of our camera rig. It consisted of 14 GoPro cameras contained in a 3D-printed housing, designed to the specific dimensions of the tripod. To capture audio from this rig, we added a quad-microphone array beneath the cameras, which fed into a compact 4-channel recording unit that sat at the base of the tripod. We chose microphones that not only performed well, but which had a very small footprint, so as to not occlude the view of the camera.



Custom camera and microphone rig

This setup captured sound in four directions from the precise location of our camera, giving us enough information to re-create a 3D aural panorama using spatial sound, which we could later synchronize to the 360-degree video.

One of the challenges with the camera array audio is that you are at the mercy of what is recorded at the time of capture. Even if the video capture is good, the sound capture can become problematic due to off-camera sounds such as sirens, airplanes, or high winds. To assure we had all the elements we need, we used a series of stereo and mono mobile recording units to get asynchronous, ambient elements at specific points of interest in each location. This capture is important as it gives the sound designer the ability to seek out clean and usable content that can be used in post-production to craft interest and add further directionality.

Any given capture day would generate a large number of files. It was important to develop a system to track which files correspond to a particular location or camera shot. Our recording unit was set up to auto-name files by date and take number and we would back these up at the end of the day to external drives. At the very least, verbally slating the beginning of audio recordings was important as this allows easy contextual identification of the content should filenames become a problem. It was also important for us to visually slate the camera rig capture as the video and audio were recorded as separate media and needed to be synchronized during post.

Editing the audio

Back at the studio after the capture trip, the first step in assembling a directional and immersive aural experience is to review all of the audio capture for a location, picking out the best takes and identifying any highlights that could be applied creatively during integration. The audio is then edited and cleaned up. For example, a loud car horn lasting a second or so and repeating a few times, can be replaced and stitched in with sections of quiet, ambient audio from the same capture.

Once the video edit for a location has been established the sound designer can synchronize the corresponding audio. At this point we worked with both camera rig capture and mobile capture to decide what elements, or combination thereof, will work to build an immersive audio scene. A technique we found useful was to add all the sound elements into an audio editor and build quick linear mock ups to experiment with different mix ideas. This gave us better formed ideas when it came time to build the actual HoloTour scenes.

Assembling the scene

The first step to building a 3D ambient scene is to create a bed of general background ambient looping sounds that will support other features and interactive sound elements in a scene. In doing so, we took a holistic approach towards different implementation techniques determined by the specific needs and design criteria of any particular scene. Some scenes might index towards using the synchronized camera capture, whereas others might require a more curated approach that uses more discretely placed sounds, interactive elements and music and sound effects for the more cinematic moments in HoloTour.

When indexing on the use of the camera capture audio, we placed spatial sound-enabled ambient audio emitters corresponding to the directional coordinates of the camera orientation such that the north camera view plays audio from the north microphone and likewise for the other cardinal directions. These emitters are world-locked, meaning the user can freely turn their head in relation to these emitters and the sound will change accordingly, effectively modeling the sound of standing at that location. Listen to Piazza Navona or The Pantheon for examples of scenes that use a good mix of camera captured audio.

A different approach involved playing a looping stereo ambience in conjunction with spatial sound emitters placed around the scene playing one-off sounds that are randomized in terms of volume, pitch and trigger frequency. This creates an ambience with an enhanced sense of directionality. In Aguas Calientes, for example, you can listen to how each quadrant of the panorama has specific emitters that intentionally highlight specific areas of the geography, but work together to create an overall immersive ambience.

Tips and tricks

When you're putting together audio for a scene, there are some additional methods you can use to further highlight directionality and immersion, making full use of the spatial sound capabilities of HoloLens. We've provided a list of some below—listen for them the next time you try HoloTour.

- **Look Targets:** These are sounds that trigger only when you are looking at a specific object or area of the holographic frame. For example, looking in the direction of the street-side café in Rome's Piazza Navona will subtly trigger the sounds of a busy restaurant.
- **Local Vision:** The journey though HoloTour contains certain beats where your tour guide, aided by holograms, will explore a topic in-depth. For instance, as the façade of the Pantheon dissolves to reveal the oculus, reverberating audio placed as a 3D emitter from the inside of the Pantheon encourages the user to explore the interior model.
- **Enhanced directionality:** Within many scenes, we placed sounds in various ways to add to the directionality. In the Pantheon scene, for example, the sound of the fountain was placed as a separate emitter close enough to the user so that they could get a sense of 'sonic parallax' as they walked around the play space. In Peru's Salinas de Maras scene, the individual perspective of some of the little streams were placed as separate emitters to build a more immersive ambient environment, surrounding the user with the authentic sounds of that location.
- **Spline emitter:** This special spatial sound emitter moves in 3D space relative to the visual position of the object it's attached to. An example of this was the train in Machu Picchu, where we used a spline emitter to give a distinct sense of directionality and movement.
- **Music and SFX:** Certain aspects of HoloTour that represent a more stylized or cinematic approach use music and sound effects to heighten the emotional impact. In the gladiator battle at the end of the Rome tour, special effects like whooshes or stingers were used to help strengthen the effect of labels appearing in scenes.

About the author



Jason Syltebo

Audio Designer @Microsoft

See also

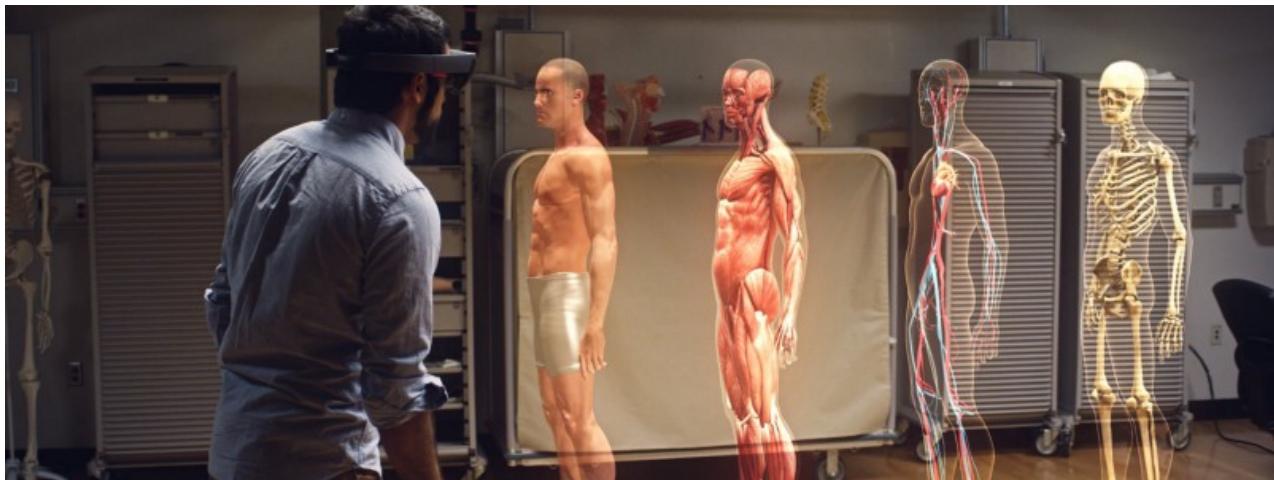
- Spatial sound
- Spatial sound design
- Spatial sound in Unity
- MR Spatial 220
- Video: Microsoft HoloLens: HoloTour

Case study - The pursuit of more personal computing

11/6/2018 • 12 minutes to read • [Edit Online](#)

Tomorrow's opportunities are uncovered by building products today. The solutions these products provide reveal what's necessary to advance the future. With mixed reality this is especially true: Meaningful insight comes from getting hands-on with real work — real devices, real customers, real problems.

At Microsoft, I'm part of the design team helping enterprise partners build experiences for their business using Windows Mixed Reality. Over the past year, our team has focused on HoloLens and understanding how Microsoft's flagship holographic device can deliver value to customers today. Working closely with designers and developers from these companies, our team focuses on uncovering solutions that would be technically unfeasible, financially impractical, or otherwise impossible without HoloLens.



HoloAnatomy from Case Western Reserve University

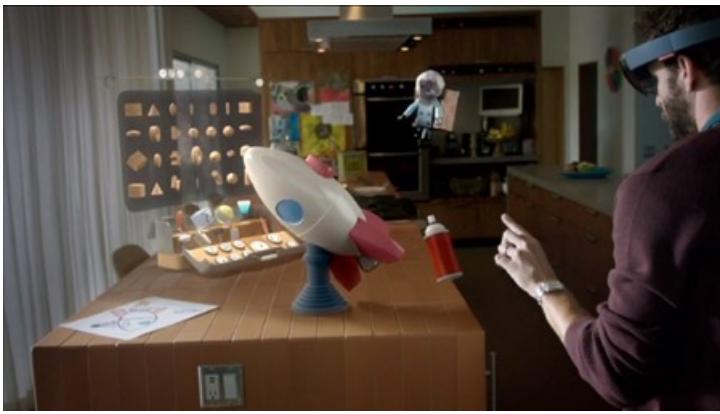
Building these solutions helps Microsoft's internal teams prepare for the next generation of computing. Learning how individuals and businesses interact with core technologies like mixed reality, voice, and AI, helps Microsoft build better devices, platforms, and tools for developers. If you are a designer or a developer exploring this space, understanding what our teams are learning from partners today is critical to preparing for tomorrow's mixed reality opportunities.

Microsoft's ambition for mixed reality

We live our lives among two worlds: the physical and the digital. Both have fundamental strengths that we leverage to augment and extend our abilities. We can talk in-person with a friend, using our physical senses to understand things like body language, or we can augment our ability to talk by video-chatting with a friend from miles away. Until now these two worlds, and their respective strengths, have been fundamentally separated.

The physical world is one of atoms and physics. We use our senses to make decisions, leveraging years of learned behavior to interact with objects and people in our environments. Despite the ease of these interactions, we are limited by our physical abilities and the laws of nature.

The digital world is one of bits and logic. Computations are made instantly while information can be distributed effortlessly. Despite the speed and flow of information, interactions are too often limited to small screens, abstract inputs, and noisy feeds.



HoloStudio

What if we could combine the advantages of the physical and digital worlds? This is the cornerstone of experiences across the spectrum of mixed reality: a medium where the physical and digital co-exist and seamlessly interact. Combining these worlds builds a new foundation for interacting more naturally with technology — an evolution in personal computing.

The spectrum of mixed reality has exploded as developers have begun exploring the opportunities of immersion and presence. Bringing users into the digital world with immersive (virtual reality) experiences and creating digital objects in the physical world with holographic (augmented reality) experiences. But what are the benefits of mapping the physical world to the digital world? What happens when we give computers eyes to see?

The fundamental camera vision technology behind holograms acts like a pair of eyes for the computer to see the environment around you: Objects in the world, people around you, changes as they happen. A digital understanding of your context in the physical world. This leads to an enormous amount of information, the implications of which we are only beginning to understand.

Culminating core technologies

Computing is too often a tangible thing. Grabbing our devices to tell them who we are and what we want. Contorting our thinking and aligning what we say to match what we believe the computer needs to hear.

The promise of mixed reality, especially in the real world with holographic experiences, is to lessen the burden of interacting with technology. Reducing cognitive load as users navigate the layers of abstraction inherent to computing today. How can we design experiences that not only take advantage of contextual understanding, but make it easier to draw insight and take action? Two contributing technologies are also attempting to solve this problem:

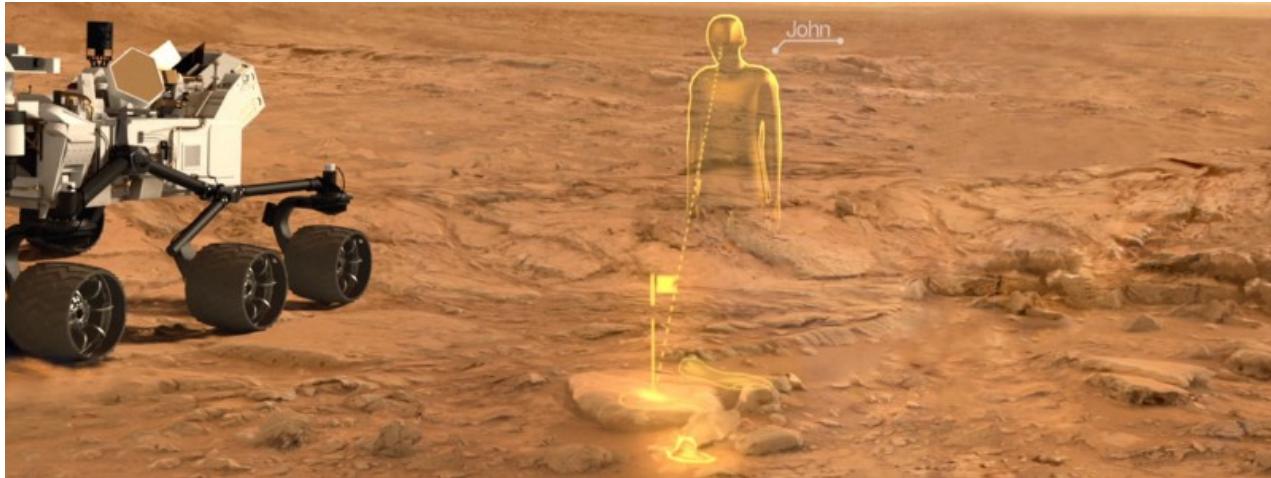
- **Voice**, in terms of speech and conversation, is enabling users to communicate with computers through more natural means — answering bots through text or issuing commands to conversational hardware.
- **AI** is powering experiences that distill insights from increasingly complex datasets. While AI is an enormous topic, recent progress has provided the groundwork for devices that rely on computer vision, more natural digital assistants, and recommending actions to users.

Mixed reality provides a medium to combine these technologies into a single user experience. Voice becomes a powerful, natural method for input when wearing a holographic headset. AI acts as a critical cipher to contextualize the enormous amounts of information connecting the physical and digital worlds. This is why Sataya Nadella refers to HoloLens as 'the ultimate computer', it's a culminating device for three core technologies. A platform to empower humans to more easily interact with the growing complexity of devices and services.

Less interface in your face

A culminating device that connects the physical world to the digital world allows us to design experiences that fit more naturally, without cumbersome abstractions. Consider the experiences you've created: When the barriers of abstraction are removed, how much of your interface is left? Which parts of your app flow change when you know the user and their context? How many menus and buttons remain?

For example, think about shared experiences in mixed reality like the OnSight tool NASA's Jet Propulsion Laboratory built for scientists. Instead of building a system to look at Martian data (abstracting the data onto screens or displays), they brought scientists inside the data, effectively putting them on the surface of Mars as they collaborated.



NASA/JPL OnSight

Instead of finding the button to draw attention to some Martian geology, scientists can point to it directly. No menus, no pens, no learning curve to using the tool effectively. By leveraging our known abilities from the physical world, more natural interactions in mixed reality can circumvent deep technical literacy in even the most advanced industry tools.

Likewise, voice and AI can extend natural interaction in experience like this. When digital assistants can 'see' into the world around us, conversations will feel less cumbersome. A bot in NASA's tool might extract context from the scientists conferring over Martian geology, acting as a ready (and conversational) source of information when scientists gesture and ask about 'this' or 'that'. A computer that knows your context is ready to jump in with information appropriate to you, through a method of interaction most appropriate to your context.

Building on a foundation

In this world of contextual experiences, catering to the mobility of the user is key. Moving fluidly between devices and services, with highly personalized contexts, users will travel about the physical world seamlessly — creating a massive platform challenge. When the physical world becomes a digital end-point, interoperability reigns supreme.

"Bringing together the digital universe and the physical world will unlock human potential... enabling every person and organization on the planet to achieve more."

— Satya Nadella

Windows Mixed Reality is an effort to create a platform for an ecosystem of devices, enabling developers to create immersive, affordable, and compatible experiences for the largest possible audience. The future will not be limited to a single manufacturer, let alone a single device. Headsets, mobile devices, PCs, accessories... all these physical things must interoperate (as well as digital things like graphs and services) through an underlying platform to successfully deliver on the promise of mixed reality.

Designing for tomorrow's experiences today



Each one of the core technologies behind this new class of experiences are enabling designers and developers to create compelling and successful experiences today. By reducing abstraction, we can interact more directly with the digital world, allowing us to design in ways that augment and amplify human abilities. Voice technology (through bots and digital assistants like Cortana) is allowing users to carry out increasingly complex conversations and scenarios, while AI technology (through tools like Microsoft Cognitive Services) is causing companies to rethink how users will interact with everything from social media to supply chain management.

These types of interactions will rely on both a new class of design tools as well fundamental support from the platform. Creating these tools and building this platform for devices and services relies on understanding how tomorrow's experiences will solve real, tangible problems. We've identified five areas of opportunity where our enterprise partners have delivered valuable solutions and where we believe continued investment will help us prepare for this new class of computing.

Areas of opportunity

The past year of developer partnerships has uncovered areas of opportunity that resonate with customers and create successful enterprise solutions. From scientists and technicians to designers and clients, five areas of opportunity have emerged where Microsoft partners are finding value with mixed reality. These areas are already providing massive insight into the future needs of platforms like Windows Mixed Reality and can help you understand how these new experiences will impact the ways we learn, collaborate, communicate, and create.

1. Creation and design

One of the chief opportunities of mixed reality is the ability to see and manipulate 3D designs in real time, in a real-world setting, at true size and scale. Design and prototyping tools have escaped the confines of screens, returning to a realm of design usually reserved for tangible, physical materials like wood and clay.

[Autodesk](#) created a mixed reality experience aimed at improving collaboration across the entire product development process. The ability for engineers and designers to survey and critique 3D models in their environment allowed them to iterate on a design in real-time. This not only enabled faster prototyping but gave way to more confident decisions in the process.

Experiences like this highlight the need for core collaboration experiences, the ability for users to see and communicate with shared objects. Autodesk's goal is to expand the product from design professionals and engineers to digital artists, students, and hobbyists. As the level of 3D expertise of users decreases, the ability to interact with objects naturally becomes crucial.

2. Assembly and manufacturing

From the increasing amount of specialization on factory floors to the rapid advancements of supply chain management, seamless access to relevant information is key. Mixed reality offers the ability to synthesize extensive data sets and provide visual displays that aid in areas like navigation and operations. These are often highly

technical, niche fields where integration with custom datasets and services are crucial to reducing complexity and providing a successful experience.

Elevator manufacturer [ThyssenKrupp](#) created an experience for elevator service technicians, allowing them to visualize and identify problems in preparation for a job. With a team spanning over 24,000 technicians, devices like HoloLens allow these technicians to have remote, hands-free access to technical and expert information.

ThyssenKrupp highlights a powerful concept here that critical, contextually-relevant information can be delivered quickly to users. As we look ahead to a new class of experiences, distilling vast amounts of possible information to content that is relevant to the user will be key.

3. Training and development

Representing objects and information in three dimensions offer new ways to explain scenarios visually and with spatial understanding. Training becomes a key area of opportunity, leveraging the ability to digitally represent enormous, complex objects (like jet engines) to create training simulations for a fraction of the cost of a physical solution.

[Japan Airlines](#) has been experimenting with concept programs to provide supplemental training for engine mechanics and flight crews. The massive jet engines (in both size and complexity) can be represented in 3D, ignoring the limitations of the physical world to move and rotate virtual objects around trainees, allowing them to see how airline components work in real-time.

Training with virtual components (and reducing the need for expensive, physical training simulators) is a key way to deliver value in enterprise scenarios today. As this scenario expands (as we've seen in areas like medicine), computer vision becomes especially important to recognize unique objects in the environment, understand the context of the user, and deliver relevant instructions.

4. Communication and understanding

Interactions between two people (whether both participants are in mixed reality devices or one is on a traditional PC or phone) can provide both a sense of immersion within a new environment or a sense of presence when communicating virtually with others. In enterprise scenarios this sense of presence is a boon to mobile teams, helping to increase understanding of projects and lessen the need for travel.

Commercial tech manufacturer [Trimble](#) developed a solution for architecture and construction industry professionals to collaborate and review work during building development. Professionals can remotely immerse themselves into a project to discuss progress or be on location and review plans as they would look (in their final form) in the environment around them.

Shared experiences are a major area of investment for Microsoft, with apps like Skype exploring new ways to represent humans in digital space. Teams are exploring volumetric video recordings, avatars, and recreations of a participant's physical space.

5. Entertainment and engagement

The nature of immersion in mixed reality can have a tremendous impact on the way customers engage with entertainment and marketing. Artists and entertainment studios have explored mixed reality as a compelling medium for storytelling, while companies are exploring the implications for brand marketing. From product demos in private showrooms to vision pieces told at trade expos, content can be delivered in more immersive and tailored ways.

Volvo created an experience for showcasing their latest car models (immersing users in different colors and configurations) while highlighting how advanced sensors work to create a better, safer driving experience. Taking customers through a guided showroom experience allows Volvo to tell the story behind cutting-edge car feature while delivering a memorable portrayal of their brand story.

Entertainment is in many ways pushing the bounds of mixed reality (especially virtual reality) and perhaps most compelling in this space is how it will interact combine with the previous area of opportunity: communication and understanding. With multiple users, each with their own variant of devices and interface methods, you can imagine a vast future of personalization in the realm of entertainment.

Start building today

It's hard to say what the far future of mixed reality will look like for consumers, but focusing on unique problems, getting hands-on with real hardware, and **experimenting today with the intersection between mixed reality, voice, and AI is key**. Microsoft is just getting started with mixed reality but learning from the successes realized by businesses today will help you create the experiences of tomorrow.

About the author



Mark Vitazko
UX Designer @Microsoft

Case study - Using spatial sound in RoboRaid

11/6/2018 • 8 minutes to read • [Edit Online](#)

Charles Sinex, audio lead on the Microsoft HoloLens Experience Team, talks about the unique challenges he encountered when creating audio for [RoboRaid](#), a mixed reality first-person shooter.

The tech

[Spatial sound](#) is one of the most exciting features of Microsoft HoloLens, providing a way for users to perceive what's going on around them when objects are out of the line of sight.

In RoboRaid, the most obvious and effective use of spatial sound is to alert the player to something happening outside of the user's peripheral vision. For example, the Breacher can enter from any of the scanned walls in the room, but if you're not facing the location where it's entering, you might miss it. To alert you to this invasion, you'll hear a distinct bit of audio coming from where the Breacher is entering, which lets you know that you need to act quickly to stop it.

Behind the scenes

The process of creating spatial sound for HoloLens apps is so new and unique and the lack of past projects to use for reference can lead to a lot of head scratching when you run into a problem. Hopefully, these examples of the audio challenges we faced while making RoboRaid will help you as you create audio for your own apps.

Be mindful of taxing the CPU

Spatial sound can be demanding on the CPU. For a busy experience like RoboRaid it was crucial to keep the instances of spatial sound to under eight at any given time. For the most part, it was as easy as setting the limit of instances of different audio events so that any instances that happen after the limit is reached are killed. For example, when drones spawn, their screams are limited to three instances at any given time. Considering only about four drones can spawn at once, three screams are plenty since there's no way your brain can keep track of that many similar-sounding audio events. This freed up resources for other spatial sound events, like enemy explosions or enemies preparing to shoot.

Rewarding a successful dodge

The dodging mechanic is one of the most important aspects of gameplay in RoboRaid, and also something that we felt was truly unique to the HoloLens experience. As such, we wanted to make successful dodges very rewarding to the player. We got the Doppler "whizz-by" to sound compelling fairly early on in the development. Initially, my plan was to use a loop and manipulate it in real-time using volume, pitch, and filter. The implementation for this was going to be very elaborate, so before committing resources to actually build this we created a cheap prototype using an asset with the Doppler effect baked in just to find out how it felt*. Our talented dev made it so that this whizz-by asset would play back exactly 0.7 seconds before the projectile will have passed by the player's ear and the results felt really amazing! Needless to say, we ditched the more complex solution and implemented the prototype.

**(If you'd like more information about creating an audio asset with the Doppler effect built in, check out an article by sound designer Charles Deenan called [100 Whooshes in 2 Minutes](#).) *



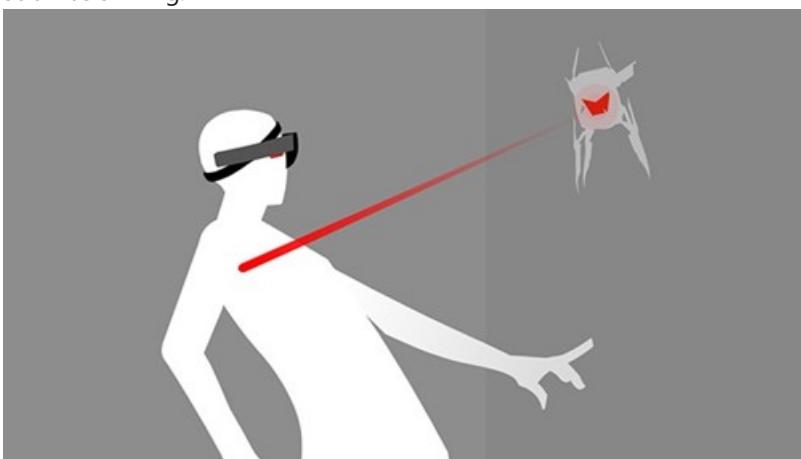
Ditching ineffective sounds

Originally, we had wanted to play an explosion sound behind the player once they've successfully dodged the enemy projectile, but we decided to ditch this for several reasons. First, it didn't feel as effective as the whizz-by SFX we used for the dodge. By the time the projectile hits a wall behind you, something else would have happened in the game that would pretty much mask that sound. Secondly, we didn't have collision on the floor, so we couldn't get the explosion to play when the projectile hit the floor instead of the walls. And finally, there was the CPU cost of spatial sound. The Elite Scorpion enemy (one that can crawl inside the wall) has a special attack that shoots about eight projectiles. Not only did that make a huge mess in the mix, it also introduced awful crackling because it was hitting the CPU too hard.

Communicating a hit

An interesting issue we ran into, which we felt was unique to the HoloLens experience, was the difficulty of effectively communicating to the players that they have been hit. What makes a mixed reality experience successful is the feeling that the story is happening to you. That means that you have to believe that YOU are fighting an alien robot invasion in your own living room.

Players obviously won't feel anything when they get hit, so we had to find a way to really convince the player that something bad had happened to them. In conventional games you might see an animation that lets you know your character has taken a hit, or the screen might flash red and your character might grunt a little. Since these types of cues don't work in a mixed reality experience, we decided to combine the visual cue with a really exaggerated sound that indicates you've taken damage. I created a big sound, and made it so prominent in the mix that it ducked everything down. Then, to make it stand out even more, we added a short warning sound as if a nuclear sub was sinking.



Getting big sound from small speakers

HoloLens speakers are small and light to suit the needs of the device, so you can't expect to hear too much low-end. Similar to developing for smart phones or handheld gaming devices, sound designers and composers have to be mindful of the frequency content of their audio. I always design sounds or write music with full frequency range because wearing headphones is an option for the users. However, to ensure compatibility with HoloLens speakers,

I run a test occasionally by putting an EQ in the master of any DAW I happen to be working in. The EQ setting consists of a high-pass filter around 600 to 700 Hz (not too steep) and low-pass filter at around 10K (very steep). That should give you a ballpark idea of how your sounds will playback on the device.

If you are relying on bass to give the sense of chord changing in your music, you may find that your music completely loses the sense of root when you apply this EQ setting. To remedy this, I added another layer to the bass that is one octave higher (with some rich harmonics) and mixed it to get the sense of root back. Sometimes using distortion to amp up the harmonics will give enough frequency content in the upper range to make our brain think that there's something underneath it. This is true for SFX like impacts, explosions, or sounds for special moments, such as a boss' super attacks. You really can't rely on the low-end to give the player a sense of impact or weight. As with music, using distortion to give a little bit of crunch definitely helped.

Making your audio cues stand out

Naturally, everyone on the team wanted bombastic music, loud guns, and crazy explosions; but they also wanted to be able to hear voiceover or any other game-critical audio cues.

On a console game with full range of frequency you have more options to divide frequencies up depending on the importance of the sound. For RoboRaid, however, I was limited in the number of ranges of frequencies I could curve out from sounds. For instance, if you use low-pass filter and curve out too much from the higher end of the spectrum, you won't have anything left on the sound because there's not much low-end.

In order to make RoboRaid sound as big as it can on the device, we had to lower the dynamic range of the whole experience and made extensive use of ducking by creating a clear hierarchy of importance for different types of sounds. I set the ducking from -2 to -6 dB depending on the importance. I personally don't like obvious ducking in games, so I spent a lot of time tuning the fade in/out timing and the amount of volume attenuation. We set up separate busses for spatial sound, non-spatial sound, VO, and dry bus without reverb for music, then created very high priority, critical, and non-critical busses. The assets were then set up to go to their appropriate busses.

I hope audio professionals out there will have as much fun and excitement working on their own apps as I did working on RoboRaid. I can't wait to see (and hear!) what the talented folks outside Microsoft will come up with for HoloLens.

Do it yourself

One trick I discovered to make certain events (such as explosions) sound "bigger"—like they're filling up the room—was to create a mono asset for the spatial sound and mix it with a 2D stereo asset, to be played back in 3D. It does take some tuning, since having too much information in the stereo content will lessen the directionality of the mono assets. However, getting the balance right will result in huge sounds that will get players to turn their heads in the right direction.

You can try this yourself using the audio assets below:

Scenario 1

1. Download [roboraid_enemy_explo_mono.wav](#) and set to playback through spatial sound and assign it to an event.
2. Download [roboraid_enemy_explo_stereo.wav](#) and set to playback in 2D stereo and assign to the same event as above. Because these assets are normalized to Unity, attenuate volume of both assets so that it doesn't clip.
3. Play both sounds together. Move your head around to feel how spatial it sounds.

Scenario 2

1. Download [roboraid_enemy_explo_summed.wav](#) and set to playback through spatial sound and assign to an event.
2. Play this asset by itself then compare it to the event from Scenario 1.
3. Try different balance of mono and stereo files.

About the author



Charles Sinex
Audio Engineer @Microsoft

See Also

- [Spatial sound](#)
- [RoboRaid for Microsoft HoloLens](#)

Case study - Using the stabilization plane to reduce holographic turbulence

11/6/2018 • 7 minutes to read • [Edit Online](#)

Working with holograms can be tricky. The fact that you can move around your space and see your holograms from all different angles provides a level of immersion that you can't get with a normal computer screen. Keeping these holograms in place and looking realistic is a technical feat accomplished by both the Microsoft HoloLens hardware and the intelligent design of holographic apps.

The tech

To make holograms appear as though they're actually sharing the space with you, they should render properly, without color separation. This is achieved, in part, by technology built-in to the HoloLens hardware which keeps holograms anchored on what we call a [stabilization plane](#).

A plane is defined by a point and a normal, but since we always want the plane to face the camera, we're really just concerned with setting the plane's point. We can tell HoloLens which point to focus its processing on to keep everything anchored and stable, but how to set this focus point is app-specific, and can make or break your app depending on the content.

In a nutshell, holograms work best when the stabilization plane is properly applied, but what that actually means depends on the type of application you're creating. Let's take a look at how some of the apps currently available for HoloLens tackle this problem.

Behind the scenes

When developing the following apps, we noticed that when we didn't use the plane, objects would sway when our head moved and we'd see color separation with quick head or hologram movements. Over the course of the development timeframe, we learned through trial and error how to best use the stabilization plane and how to design our apps around the problems that it can't fix.

Galaxy Explorer: Stationary content, 3D interactivity

[Galaxy Explorer](#) has two major elements in the scene: The main view of the celestial content and the small UI toolbar that follows your gaze. For the stabilization logic, we look at what your current gaze vector intersects with in each frame to determine if it hits anything on a specified collision layer. In this case, the layers we're interested in are the planets, so if your gaze falls on a planet, the stabilization plane is placed there. If none of the objects in the target collision layer are hit, the app uses a secondary "plan B" layer. If nothing is being gazed at, the stabilization plane is kept at the same distance as it was when gazing at the content. The UI tools are left out as a plane target as we found the jump between near and far reduced the stability of the overall scene.

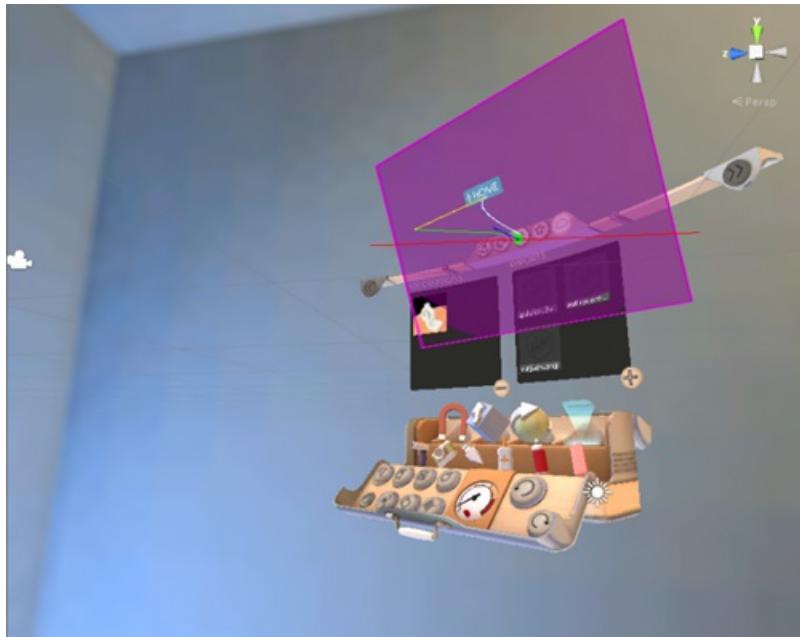
The design of Galaxy Explorer lends itself well to keeping things stable and reducing the effect of color separation. The user is encouraged to walk around and orbit the content rather than move along it from side to side, and the planets are orbiting slowly enough that the color separation isn't noticeable. Additionally, a constant 60 FPS is maintained, which goes a long way in preventing color separation from happening.

To check this out yourself, look for a file called LSRPlaneModifier.cs in the [Galaxy Explorer code on GitHub](#).

HoloStudio: Stationary content with a UI focus

In HoloStudio, you spend most of your time looking at the same model you're working on. Your gaze doesn't move a significant amount, except for when you select a new tool or want to navigate the UI, so we can keep the

plane setting logic simple. When looking at the UI, the plane is set to whatever UI element your gaze snaps to. When looking at the model, the plane is a set distance away, corresponding with the default distance between you and the model.



HoloTour and 3D Viewer: Stationary content with animation and movies

In HoloTour and 3D Viewer, you're looking at a solitary animated object or movie with 3D effects added on top of it. The stabilization in these apps is set to whatever you're currently viewing.

HoloTour also prevents you from straying too far away from your virtual world by having it move with you instead of staying in a fixed location. This ensures that you won't get far enough away from other holograms for stability issues to creep in.



RoboRaid: Dynamic content and environmental interactions

Setting the stabilization plane in RoboRaid is surprisingly simple, despite being the app that requires the most sudden movement. The plane is geared towards sticking to the walls or the surrounding objects and will float at a fixed distance in front of you when you're far enough away from them.

RoboRaid was designed with the stabilization plane in mind. The reticle, which moves the most since it's headlocked, circumvents this by using only red and blue which minimizes any color bleeding. It also contains a small bit of depth between the pieces, minimizing any color bleed that would occur by masking it with an already expected parallax effect. The robots don't move very quickly and only travel short distances in regular intervals. They tend to stay around 2 meters in front of you, where the stabilization is set by default.

Fragments and Young Conker: Dynamic content with environmental interaction

Written by Asobo Studio in C++, Fragments and Young Conker take a different approach to setting the

stabilization plane. Points of interest (POI) are defined in the code and ordered in terms of priority. POIs are in-game content such as the Conker model in Young Conker, menus, the aiming reticle, and logos. The POIs are intersected by the user's gaze and the plane is set to the center of the object with the highest priority. If no intersection occurs, the plane is set to the default distance.

Fragments and Young Conker also design around you straying too far from the holograms by pausing the app if you move outside of what's been previously scanned as your play space. As such, they keep you within the boundaries that are found to provide the most stable experience.

Do it yourself

If you have a HoloLens and would like to play around with the concepts I've discussed, you can download a test scene and try out the exercises below. It uses Unity's built-in gizmo API and it should help you visualize where your plane is being set. This code was also used to capture the screenshots in this case study.

1. Sync the latest version of [MixedRealityToolkit-Unity](#).
2. Open the [HoloToolkit-Examples/Utilities/Scenes/StabilizationPlaneSetting.unity](#) scene.
3. Build and configure the generated project.
4. Run on your device.

Exercise 1

You'll see several white dots around you at different orientations. In front of you, you'll see three dots at different depths. Air tap to change which dot the plane is set to. For this exercise, and for the other two, move around your space while gazing at the dots. Turn your head left, right, up, and down. Move closer to and farther from the dots. See how they react when the stabilization plane is set to different targets.

Exercise 2

Now, turn to your right until you see two moving dots, one oscillating on a horizontal path and one on a vertical path. Once again, air-tap to change which dot the plane is set to. Notice how color separation is lessened appears on the dot that is connected to the plane. Tap again to use the dot's velocity in the plane setting function. This parameter gives a hint to HoloLens about the object's intended motion. It's important to know when to use this, as you'll notice when velocity is used on one dot, the other moving dot will show greater color separation. Keep this in mind when designing your apps—having a cohesive flow to the motion of your objects can help prevent artifacts from appearing.

Exercise 3

Turn to your right once more until you see a new configuration of dots. In this case there are dots in the distance and one dot spiraling in and out in front of them. Air tap to change which dot the plane is set to, alternating between the dots in the back and the dot in motion. Notice how setting the plane position and the velocity to that of the spiraling dot makes artifacts appear everywhere.

Tips

- Keep your plane setting logic simple. As you've seen, you don't need complex plane setting algorithms to make an immersive experience. The stabilization plane is only one piece of the puzzle.
- When at all possible, always move the plane between targets smoothly. Instantly switching distant targets can visually disrupt the scene.
- Consider having an option in your plane setting logic to lock onto a very specific target. That way, you can have the plane locked on an object, such as a logo or title screen, if needed.

About the author



Ben Strukus
Software Engineer @Microsoft

See also

- [MR Basics 100: Getting started with Unity](#)
- [Focus point in Unity](#)
- [Hologram stability](#)

Windows Mixed Reality Events

11/6/2018 • 2 minutes to read • [Edit Online](#)

Join the Windows Mixed Reality Team at these upcoming events!

San Francisco Reactor

680 Folsom St.
San Francisco, CA 94107

IGNITE

September 24-28, 2018

Get the latest insights and skills from technology leaders and practitioners shaping the future of cloud, data, business intelligence, teamwork, and productivity. Immerse yourself with the latest tools, tech, and experiences that matter, and hear the latest updates and ideas directly from the experts.

This event has sold out, however make sure to join us for the keynote, more information is available [here](#).

AWE: Projection Mapping

October 9, 2018 6:30PM - 9:00PM

This month AWE Nite SF explores the world of projection mapping. Beer and pizza will be served during networking.

Interested in demoing, sponsoring or volunteering at this event? Fill out this form
<https://augmentedworldexpo.com/awe-nite-meetups/>

Register for this event [here](#).

VRS 2018

October 16-17, 2018

Microsoft will be sponsoring VRS 2018, the annual executive conference produced by Greenlight Insights, the global leader in virtual and augmented reality market intelligence.

Register for this event [here](#).

Microsoft HoloLens & Windows Mixed Reality Meet-Up

October 15, 2018

Let's keep on talking about HoloLens and Mixed Reality. This time it's about what it takes to make HoloLens apps, and an amazing new feature coming to HoloLens!.

Register for this event [here](#).

VRS Summit

October 18, 2018

This Innovation Junto will explore a variety of the hottest topics and themes in technology & innovation. Top 5 Junto 2018 Themes:

- Natural Language Processing & Voice Interfaces
- Blockchain
- Deep Neural Networks
- Big Data, the Cloud, and XR
- Artificial Intelligence & Computer Vision

Register for this event [here](#).

Unite LA

October 23-25, 2018

Come and check out the Microsoft booth at Unite LA – Unity's premium developer event. Get up-to-date with the latest developments in the content-creation engine used by a community of millions. And see the beautiful and amazing things other creators are doing with 2D, 3D, and VR/AR games and experiences in Unity. Thousands of creators just like you, across multiple industries including games, film, auto and AEC, come together at Unite Los Angeles. It's not to be missed!

Register for this event [here](#).

Windows Mixed Reality Hackathon with Springer Nature

November 7-9, 2018

Explore the potential to visualize Springer Nature publications in new ways, using Augmented Reality with Microsoft HoloLens

- Help to shape the external awareness of Springer Nature: from traditional publisher to becoming a tech company with key focus on Open Research and Open Access publishing
- Serve as innovation hub for colleagues and industry partners to generate new technology
- Build strong relationships with top research institutes and corporations in the Bay Area
- Provide Springer Nature with first-hand feedback from users of our content and data and help to identify pain points and room for improvement
- Allow trial & error with swift prototypes and proofs of concept

Register for this event [here](#).

Past Events

[Mixed Reality: Mixed Reality Workgroup](#) - September 24, 2018

[Mixed Reality: Augmenting Our World](#) - September 20, 2018

[AWE Nite SF – Augmented Reality & Sports](#) - September 11, 2018

[Mixed Reality & HoloLens Work Group](#) - September 10, 2018

[WebXR Hackathon Challenge](#) - June 29 - July 1, 2018

[Non-Gaming meet-up \(Medical\)](#) - July 16, 2018

[AWE meet-up](#) - July 17, 2018

[Women in Data -Soft Skills in Data Science meet-up](#) - July 18, 2018

[XRedu Hackathon](#) - July 27-29, 2018

HoloLens Research Mode tutorial at CVPR 2018

11/6/2018 • 2 minutes to read • [Edit Online](#)

1:30pm - 2:50pm. June 19th, 2018

[CVPR 2018 Conference](#)

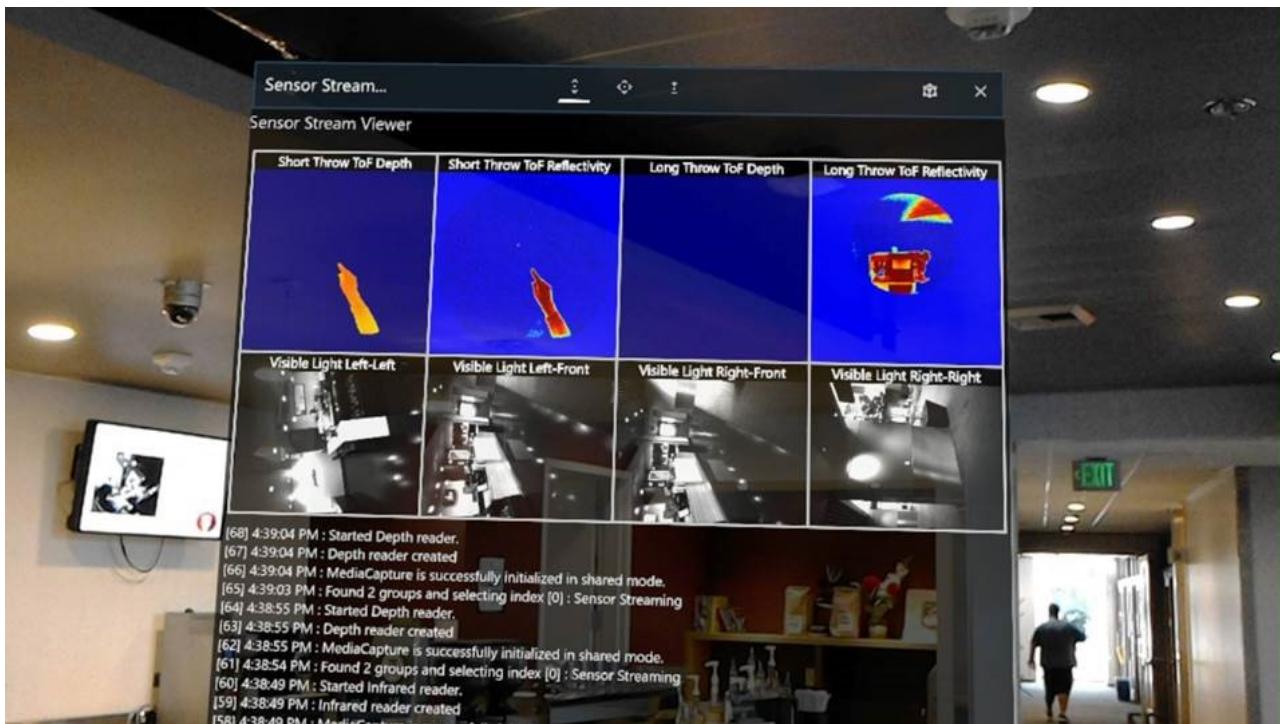
Presenters

- Marc Pollefeys
- Paweł Olszta

Overview

Microsoft HoloLens is the world's first self-contained, holographic computer, but it's also a potent computer vision research device. Application code can access audio and video streams and surface meshes, all in a world coordinate space maintained by HoloLens' highly accurate head-tracking. This short tutorial will dive into the new "Research Mode" capability of HoloLens (available with the [Windows 10 April 2018 Update](#) for HoloLens).

We will show you how to access the raw head-tracking and depth sensor data streams, and make use of the intrinsics and extrinsics of each stream. We will also be demonstrating recent advances in time of flight depth-sensing technologies.



A sample HoloLens application that displays any of the six Research Mode streams in real time.

Schedule

Note that this tutorial occurs on the same afternoon as Oral and Spotlight paper sessions, and for that reason we are keeping it short. It will be done before the papers sessions begin at 2:50 PM.

- 1:30pm Introduction to HoloLens
- 1:45pm Research Mode: getting your hands on the sensor streams

- 2:15pm Research Mode in use – demos and videos
- 2:45pm Sneak peek at recent advances in time-of-flight depth sensing

Attendees to this tutorial will leave with a good sense of how HoloLens can be used for a range of Computer Vision research tasks, and materials to quickly get them started using the device.

Mixed Reality Partner Program

11/6/2018 • 21 minutes to read • [Edit Online](#)

The Mixed Reality Partner Program is an integrated business program focused on enabling and supporting digital agencies, systems integrators, and solution providers who are committed to building mixed reality solutions. It is a performance-based program that offers incentives to partners as they help customers successfully pilot and deploy commercial mixed reality solutions.

The goal of the Mixed Reality Partner Program is to enable Microsoft partners to collaborate and build enterprise commercial mixed reality solutions that improve communications, streamline operations, drive worker productivity, and create amazing customer experiences. The ultimate business outcome of the program is a customer engagement that leads to enterprise deployment.

Mixed Reality Partner Program Overview (MRPP)

Shortly after announcing HoloLens in 2015, Microsoft invited digital and creative agencies to develop mixed reality solutions as part of the HoloLens Agency Readiness Program. In return, we provided partners with technical readiness training to help them deliver compelling mixed reality solutions across multiple industries.

With agencies around the globe now skilled in mixed reality development, including those in Australia, Ireland, France, Germany, New Zealand, the United Kingdom, the Agency Readiness Program partners have produced tangible results, such as POCs, pilots, and deployments of world-class mixed reality solutions. Microsoft partners are now leading the digital transformation of enterprise solutions for customers including Stryker Communications, PGA Tour, Paccar, Helsinki Airport, SITA, Grundfos Group, and many more. The Mixed Reality Partner Program expands on the early success of the Agency Readiness Program by inviting agencies, system integrators, and solution partners from around the globe to join us on this exciting mixed reality journey.

Partner types

This program is for system integrators and agencies that meet the [requirements](#) for our target markets.

Target industries and scenarios

The Mixed Reality Partner Program is focused primarily on six industries: manufacturing, public sector/government, education, healthcare, architecture/engineering/construction, and retail.

There are several commercial use cases that span our target industries:

- Creation & design
- Assembly/manufacturing
- Training & development
- Communications
- Entertainment
- Frontline worker scenarios (technician, support, customer service)

Program framework summary

The program is a four-phased program with incentives (benefits) that are earned as the partner achieves key milestones, such as the successful completion of business planning, technical training, or implementation of a POC, pilot, or deployment.

The four phases of the program are:

1. [Intake application](#)

2. [Onboarding](#)
3. [Readiness training](#)
4. [Customer engagement](#)

Partner benefits summary

The four categories of incentives of the program are:

1. Joint business planning
2. Technical readiness training
3. Technical assistance
4. Sales and marketing enablement

For details on how to qualify for these incentives, see [Partner incentives](#).



Joint business planning



Readiness training



Technical assistance & mentoring



Sales & marketing enablement

The four categories of program incentives

Joint business planning

- Connection to partner community (SIs and agencies)
- Insights into business directions, new features, events, speaking opportunities, etc

Technical readiness training

- Envisioning workshop & readiness program (multi-week) led by the product team
- Mentorship and quality reviews

Technical assistance

- Engineering support for pilots and deployments (up to 20 hours)
- Enablement Services direct from engineering team available for purchase

Sales and marketing enablement

- Sales and marketing readiness workshop and access to content
- Permission to use Mixed Reality Partner Program badge
- Business support via Partner Sales Executive
- Case study development support; PR support

Partners who meet the program requirements, apply, agree to the terms of the program, and are admitted into the program and successfully complete the milestones in each phase, will be connected to a Microsoft partner sales executive in their geography, trained on how to build mixed reality solutions, supported by the product marketing and field marketing organizations, and assisted directly by the Microsoft Mixed Reality engineering team in POCs, pilots, and deployments.

Partner commitments

There are three categories of commitments partners make when they join the program:

Readiness

- Complete the mixed reality technical readiness program
- Attend annual sales and marketing training webinar

Investment

- Minimum purchase of 3 HoloLens units
- MR development team consisting of at least 2 engineer and 1 designer
- MR sales/marketing lead

Business performance

- 2 POCs/yr
- 1 mixed reality pilot in the first year (SIs)
- 1 deployment within 18 months
- 1 case study within 18 months
- Abide by program and reporting requirements, which includes Quarterly Business Reviews (QBRs) and pipeline tracking

Minimum application requirements

Acceptance into the program is based on an [application](#) and interview. Partners must meet these minimum requirements to qualify for the Mixed Reality Partner Program. Meeting the minimum requirement is not a guarantee of entry.

Systems integrator and solution provider minimum requirements:

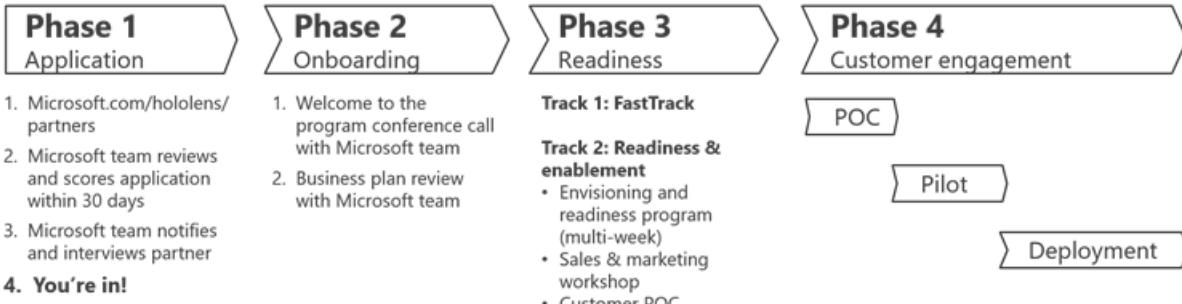
- Minimum of 25 full time employees
- Minimum of \$15M total annual revenue
- Relationships with a client who is prepared to go through a POC (the client should be aligned with one Windows Mixed Reality target industries)
- Deep expertise in custom development and enterprise integration (with referenceable customers)

Digital agency minimum requirements:

- Minimum of 5 full time employees
- Have designed a POC using HoloLens, holographic computing, or AR/VR technology (or have an immediate opportunity in the pipeline)
- Developers and designers have a strong understanding of 3D development and performance tuning (experience with Unity, DirectX or building 3D assets)
- Active in developer communities

Program framework

The MRPP program consist of four phases starting with an application:



Benefits unlocked

Phase 1

- PSE assigned
- Insights into biz directions

Phase 2

- Readiness workshops
- Connection between SI/agency
- Early access to devkits & EV kits

Phase 3

- Engineering support
- Mixed Reality Partner badge
- Partner logo on MS.com

Phase 4

- Engineering support
- Case study & PR support

The four phases of the MRPP program: intake, onboarding, training, and engagement

Phase 1: Intake application

The intake application phase is when you communicate to us your interest in becoming a Windows Mixed Reality partner. The journey starts by completing an intake application form. On the form you will be asked a variety of business and technical questions to help us evaluate your initial fit to the program. If you meet the initial requirements, the next step will be a phone interview. Following the phone interview, we will make a decision about your application. In general, you should expect the process, including notification of our decision and entry into the program, to take between 60 and 90 days.

The [intake application form](#) will take some time, but the completeness of the application is an important step in our decision making process. We value your time and effort to share your business and interest in Windows Mixed Reality.

Once we receive your application, we will review it for fit within the program. If you are not selected, you will receive an email informing you of our decision. There are three possible reasons why an application would not be selected:

1. The application was not complete or did not provide enough details to make a decision.
2. The application did not align with our current needs such as target industries, skills, or location.
3. The program is full. We will keep your application on record for another 90 days.

If the application looks promising, one of our regional partner managers (RPMs) will reach out to schedule a phone interview with you. The interview will be scheduled for one hour. We ask that you have both a business and technical leader from your company on the call. Usually one of these roles will also be your company's sponsor of the partnership, but if not, please have the internal sponsor on the call as well. During the interview, we will ask some deeper questions pertaining to the answers on your application, including:

- Business opportunities related to Windows Mixed Reality
- How you approach envisioning and design
- Team makeup and location
- 3D asset experience
- Integration and deployment experience
- What you hope to get out of the program

We will also provide details around commitments and next steps, and you will have an opportunity to ask us questions. Following the interview, we will inform you of our decision, usually within one week of the call. Should you be accepted into the program, the next step will be to onboard you into the program.

Phase 2: Onboarding

Coming out of a successful interview call there are two possible paths:

1. You've already completed a project with a client and we want to fast-track you into the program.
2. You have not completed a project with a client, or the project doesn't quite meet our expectations, and we decide you would benefit from readiness training.

Given these two possible paths, the primary purpose of onboarding is to set mutual expectations and agree to move forward in the program. There will be two onboarding calls:

- Welcome to the program to take care of administrative items and go over the structure and expectations.
- Business and technical planning discussion to align around your client opportunity and training needs.

The calls will be led by a Microsoft Regional Partner Manager (RPM). A template and agenda will be provided for each call. Agenda topics will include:

- Overview of the process, dates, and resource needs from your company
- Identify administrative needs like NDA, system onboarding, and contact information for the participants
- POC/client selection for the program
- What happens at the end of the program
- Development of a one page partner profile

Phase 3: Readiness training

The purpose of the Readiness phase is to demonstrate your company's ability to create and deliver compelling mixed reality experiences. To this end, we will ask you to complete a HoloLens app for a customer in one of our [target industries](#). The application must demonstrate a mastery of creating compelling user experiences coupled with high quality assets. Further the application must flex the unique features of HoloLens and demonstrate end value to your customer.

Please reference [Why HoloLens](#) for more information on the unique features and value of HoloLens. Your team must be composed of both developers and designers that have a strong understanding of 3D development and performance tuning.

The exit criteria for the readiness step includes:

1. Deliver to a customer a HoloLens app meets our [app quality criteria](#). There are basically two ways to complete this criteria:
 - Fast-track: show us a finished HoloLens project you completed for a customer
 - Readiness training: we will provide training and mentorship through the process of you completing a HoloLens project for a customer
2. Complete our sales and marketing readiness training
3. Complete a business plan with us that includes a view into your MR opportunity pipeline

Fast-track

If you have already completed a HoloLens project for a customer, then fast-track may be the quickest way into the program. The exit criteria defined above still applies, but going through a three-month training program is not necessary. During fast-track you must:

- Be willing to show us your project in device. We will evaluate the project against our [app quality criteria](#).
- You must take our sales and marketing training, or share with us your customer-facing MR sales and marketing plans. The material will be evaluated by our marketing team.
- If the above deliverables pass our quality bar, then we will work with you to develop a business plan that must include a view into your MR opportunity pipeline.

If you don't pass our quality bar, we will provide you details as to why. You will have 90 days to make improvements and try again.

Fast-track alternative for System Integrators

System Integrators can partner with one of our [recommended digital agencies](#) rather than building the project directly. Since the agency is already qualified, we can assume the HoloLens experience will meet our standards. You will need to continue to work with a recommended digital agency for future projects. This requirement will be removed once your company is able to demonstrate a proficiency in building HoloLens experiences based on our [app quality criteria](#).

Readiness training

Our readiness training program is designed to help your company deliver quality MR experiences on HoloLens. The program typically takes three to four months depending on the nature of your project. The project must be for a customer and meet our expectations regarding [target markets](#) and scenarios. We are constantly evolving the program to meet the needs of our partners, but at a minimum you should expect:

- A week long immersive training at a Microsoft site typically in Redmond Washington, or London UK. This training is held the first week of the program:
- Sales and marketing training either onsite or via a webinar
- Regular calls with a Microsoft team member to review progress, provide feedback, and unblock problems
- A storyboard check-point review for the POC
- A mid-term POC check-point review
- A final POC review and pass/no-pass grade
- Any variation to this format will be discuss during the [onboarding call](#)

The final POC must pass our quality bar. If you don't pass our quality bar, we will provide you details as to why. You will have 30 days to make improvements and try again. If you don't pass the second time, you must enter through the fast-track program.

Upon successful completion of the Readiness phase, you qualify for all program benefits.

Phase 4: Customer engagement

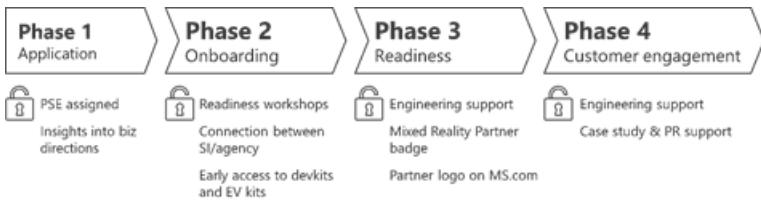
Once you graduate from the Readiness phase, your journey as a Mixed Reality Partner is just beginning. This is the time you will be helping us grow this market through POCs, pilots, and deployments of HoloLens and Windows Mixed Reality devices. During our partnership, we will set mutual expectations and evaluate them regularly. From our perspective, there are a few things we are looking for from our partners:

- **Execute account plans that drive device sales.** We will be looking at the number units sold through your efforts including POC, pilots, and large deployments. As a part of this, we will want to understand your MR opportunity pipeline. This will help us better understand market trends, product needs, and (in general) where we should be putting our outbound efforts and resources.
- **Deliver light-up scenarios.** We've only scratched the surface of how MR will benefit businesses and consumers. Compelling demos and applications show people what's possible and spawn new ideas. We value partners that are contributing to the Windows Mixed Reality Store. There are also opportunity to showcase your work at our Microsoft Technology Centers and other high-profile locations and events.
- **Contribute to the MR community.** Windows Mixed Reality is a growing and active community, and being a part of it means learning from each other. We value partners who contribute to our forums and open source projects. Blog, speaking opportunities, hackathons, and other events are all great ways to get involved.
- **Establish a deep partnership with us.** We encourage you to get to know us and stay in contact. Let us know about exciting new developments and opportunities. Contribute to our events and summits. Occasionally we may offer early access to information or hardware, so engage and give us feedback. By helping us, we can better help you.

All of the above represents elements of a business plan. You should expect your Microsoft team to review during our quarterly business updates. It is important to understand that graduation from the readiness program is not a long-term guarantee of staying in the program.

Partner incentives

The Mixed Reality Partner Program includes a rich set of partner incentives that are received (“unlocked”) by partners as they move through the program. The graphic below illustrates just some of the incentives that are unlocked in each phase.



The incentives of the program for each phase

Joint business planning

Planning between a partner and the Microsoft team begins in earnest in Phase 3 (Readiness) once the partner is on a path towards graduation. Several incentives associated with joint business planning are described in this section.

Account management

During the late stages of readiness, partners are assigned a Partner Sales Executive (PSE) as a primary sales contact. The PSE is primarily responsible for helping the partner drive sales. To achieve this, a PSE is accountable to help partners in four areas:

- Business planning—assist partners as they build their business plan, which should include strategies for account-based targeting, marketing, and selling
- Sales engagements—ensure partners have the content, evidence, and tools they need to pursue opportunities, and assist with sales engagements
- Field connections—connect partners with local Microsoft field marketers and sellers to drive joint go-to-market efforts, including events, campaigns, and joint selling.
- Engineering services—connect partners with the Microsoft engineering team to provide guidance, or to assist with technical aspects of customer engagements in POC, Pilot, or Deployment mode.

Insights into business directions and new features

Partners will receive regular updates from the Microsoft team that include confidential material covered under the program’s Non-Disclosure Agreement:

- the state of the business
- industry trends and data
- new areas of investment
- new product capabilities and features
- upcoming product announcements

Most of these insights will be communicated during Quarterly Business Updates, which will be coordinated by your regional partner manager (RPM).

Partner connections

The Microsoft team will help make connections between partners in the program to optimize their ability to move a customer from POC to deployment. One of the most important business planning topics we cover in the Onboarding phase is whether a partner will drive a customer engagement from POC to pilot to deployment with or without assistance from another partner.



Work directly with a customer or outsource development to an MRPP agency partner

While some partners will have the capabilities to drive a project end to end, we've learned through experience that projects are often most successful when the SI partners with a digital agency on the mixed reality creative development work. Likewise, we find that digital agencies are most successful when they partner with an SI to help drive deployment and backend IT integration (security, MDM, etc.).

Industry and account targeting

The Microsoft team will develop with partners strategies to build awareness, interest, and POC opportunities at specific accounts in the enterprise customer segment. Microsoft will also share recommendations on tools and tactics that help optimize account-based selling and marketing.

Customer and industry events and promotional opportunities

Due to the high levels of interest industry-wide in mixed reality technologies, the Microsoft team is in a constant state of planning for customer and industry events. Partners will have numerous opportunities to coordinate their presence at these events, which may include:

- presenting in an event keynote, breakout, or working session
- being featured in videos and keynotes
- showing a demo or having a presence on the event floor
- sharing their successes with the press

Establish a joint rhythm of the business (ROB)

During Onboarding, the Microsoft team and partner will plan a business meeting cadence to stay connected, informed, and aligned. This cadence will include scheduling Quarterly Business Updates, technical readiness training, and a sales and marketing workshop, and any other connection meetings that are needed.

The team will also identify key contacts and roles and responsibilities.

Readiness training

Partners receive deep technical training as well as sales and marketing training in Phase 3 of the Mixed Reality Partner Program. The incentives associated with readiness training are described in this section.

Envisioning + readiness

As described in the Readiness training section, partners participate in a week long immersive training program at a Microsoft location, typically in Redmond, Washington or London, UK. Over the course of a week, expert-level engineers and technical program managers work closely with partners to help them ramp on mixed reality technologies. During the envisioning and readiness training week, the Microsoft team delivers the following:

- Coaching on how to use envisioning as a process to help customers identify opportunities to use mixed reality to deliver value to their business
- The fundamentals of building mixed reality solutions
- Technical guidance on tools and techniques
- Hands-on labs to help familiarize your team with mixed reality environments
- Instruction on how to deliver a compelling POC
- How to access resources and get assistance with customer engagements

While there is T&E expense for partners, the entire readiness curriculum—the classroom experience and webinars—are free to partners.

Sales & marketing workshop

As part of readiness training, the Microsoft team provides partners sales and marketing training via a webinar (so that the partner's marketing and sales team members can attend). During the webinar, partners will hear from Microsoft marketers and sellers on several topics:

- Market opportunity
- Competitive landscape
- Positioning and messaging
- How to work with the Microsoft field marketing team
- Account-based marketing strategies that work
- Selling 101
- Generating demand with digital marketing

Partners that complete the sales and marketing workshop receive access to a marketing and sales bill of materials that contains content and tools to support partner-led sales and marketing.

Customer POC

To "graduate" from the Readiness training phase, partners must conduct a successful customer proof of concept. To help partners meet this requirement, the Microsoft team will assist with the following:

- Help the partner scope potential target accounts
- Provide a POC framework
- Provide templates to support the project management of the POC
- Participate in updates with the partner team
- Engage with customers as needed

Technical assistance

Mixed reality is a new paradigm, and the skills needed to design, develop, and deploy a mixed reality solution are new. By providing technical assistance and mentoring to our partners as they conduct POCs, pilots, and deployments, we pass along the technical know-how so that our partners can drive successful POCs, pilots, and deployments. Several incentives associated with technical assistance and mentoring are described in this section.

Engineering assistance for pilot and deployment opportunities

Partners that complete Phase 3 (Readiness) qualify for 20 hours of (free) technical assistance from the Microsoft mixed reality engineering team to help with pilot and deployment opportunities. A qualified opportunity is past the lead stage and is a customer project with a high likelihood of moving into a pilot or full-scale deployment.

An opportunity that would not qualify for engineering assistance would include a POC, such as a marketing demo with no opportunity to grow into a pilot project, or a project that is aimed at selling devices to consumers. Your Partner Sales Executive (PSE) and Regional Partner Manager (RPM) can help qualify opportunities that are eligible for engineering assistance.

Enablement Services hours for purchase

If a partner wants additional assistance directly from the Microsoft engineering team, additional hours are available for purchase via our Enablement Services program. Your Partner Sales Executive (PSE) and Regional Partner Manager (RPM) can provide details and facilitate this transaction.

Deployment assistance

Once a POC and pilot have completed (by meeting the requirements for advancement) and the customer is ready

to deploy a mixed reality solution, our team offers deployment assistance for qualified opportunities. They will provide the following services:

- Evaluate deployment plan
- Assess solution architecture and technology stack
- Identify integration plans (IT security, MDM, etc.)
- Provide implementation guidance
- Provide limited escalation support

Your Partner Sales Executive (PSE) and Regional Partner Manager (RPM) can help qualify opportunities that are eligible for deployment assistance, and help scope the hours, people, and processes the engineering team will use to provide the support.

Pre-sales escalation support

The Microsoft team is available to provide technical assistance with pre-sales customer presentations and meetings. Your Partner Sales Executive (PSE) can help coordinate this support.

Sales and marketing enablement

An important aim of the Mixed Reality Partner Program is to help our partners market and sell their solutions and services. When accepted into the program, partners get access to content, tools, and a variety of sales and marketing support services.

Partner-led sales and marketing

Microsoft provides assistance via market intelligence, content, tools, joint go-to-market execution (including PR and case study support), and a range of incentives; however, partners are expected to lead their own sales and marketing efforts.

There are four key principles of partner-led sales and marketing of mixed reality solutions:

- **Differentiated.** Partners lead with their own differentiated services/product offering, and they should have or develop advanced marketing skills and a differentiated value proposition.
- **Performance-based.** As part of business planning, partners agree to specific, measurable performance metrics, and demonstrate measurable return on marketing investment in order to receive additional investment and incentives via the program.
- **Digital first.** Partners should prioritize digital marketing and sales strategies. They should collaborate with and learn from digital marketing agencies (if needed) to plan and execute cost-effective digital marketing and social selling strategies across all their customer engagement channels.
- **Intelligent.** Partners should leverage market data insights, account data, analyst report, Microsoft customer data and guidance to tune sales and marketing strategies, and to execute account-based marketing campaigns.

Sales and marketing content

Partners will be provided access to a sales and marketing bill of materials to facilitate their customer marketing efforts. The marketing content (pitch deck, data sheet, email template, etc.) are customizable so that you can present them with your logo.

In addition, all program partners are welcome to join the MPN program, Microsoft's global partner program, and get access to an extensive library of sales and marketing guidance and tools.

Mixed Reality Partner badge

The Mixed Reality partner badge helps customers identify Microsoft partners who have achieved a high level of competency on the Windows Mixed Reality platform. Partners who attain and display this badge stand out as having been endorsed by Microsoft, differentiate themselves from competitors, and increase their credibility in the market. For an overview of the badge usage guidance, see the Mixed Reality Partner Program Badge Usage Guide.

Case study development

Partners receive assistance with building case studies after successful customer deployments. Microsoft provides templates, guidance, and (for qualified opportunities) will even provide resources to interview the customer, create case study content, and publish the case study on Microsoft.com.

PR support

Partners receive PR assistance for news-worthy events, customer wins, and large deployments. In addition, partners may be invited to participate in press briefings, speaking engagements, and other opportunities to drive press for their company.

Web presence

Partners get placement of company logo and short description on HoloLens.com/partners site, the Microsoft Partner Center, and the soon-to-be-published Mixed Reality web site. In addition, company contact info will be included in auto-locator on HoloLens web site.

Microsoft Technology Centers

Program partners are welcome to use Microsoft MTCs (with over 40 locations around the globe) to engage customers, empower employees, optimize operations, and reinvent products and business models. The MTC staff and local Microsoft account teams will help you position your offering to customers, conduct envisioning workshops with your customers, and demo your solutions on Microsoft technologies. Our goal is to move your customer closer to deployment.

See also

- [App quality criteria](#)
- [Development resources](#)
- [Design resources](#)
- [Mixed Reality Academy](#)
- [Microsoft Technology Centers](#)

App quality criteria

11/6/2018 • 18 minutes to read • [Edit Online](#)

This document describes the top factors impacting the quality of mixed reality apps. For each factor the following information is provided

- Overview – a brief description of the quality factor and why it is important.
- Device impact - which type of Window Mixed Reality device is impacted.
- Quality criteria – how to evaluate the quality factor.
- How to measure – methods to measure (or experience) the issue.
- Recommendations – summary of approaches to provide a better user experience.
- Resources – relevant developer and design resources that are useful to create better app experiences.

Frame rate

Frame rate is the first pillar of hologram stability and user comfort. Frame rate below the recommended targets can cause holograms to appear jittery, negatively impacting the believability of the experience and potentially causing eye fatigue. The target frame rate for your experience on Windows Mixed Reality immersive headsets will be either 60Hz or 90Hz depending on which Windows Mixed Reality Compatible PCs you wish to support. For HoloLens the target frame rate is 60Hz.

Device impact

HOLOLENS	IMMERSIVE HEADSETS
✓ <input checked="" type="checkbox"/>	✓ <input type="checkbox"/>

Quality criteria

BEST	MEETS	FAIL
The app consistently meets frames per second (FPS) goal for target device: 60fps on HoloLens; 90fps on Ultra PCs; and 60fps on mainstream PCs.	The app has intermittent frame drops not impeding the core experience; or FPS is consistently lower than desired goal but doesn't impede the app experience.	The app is experiencing a drop in frame rate on average every ten seconds or less.

How to measure

- A real-time frame rate graph is provided through by the [Windows Device Portal](#) under "System Performance".
- For development debugging, add a frame rate diagnostic counter into the app. See Resources for a sample counter.
- Frame rate drops can be experienced in device while the app is running by moving your head from side to side. If the hologram shows unexpected jittery movement, then low frame rate or the stability plane is likely the cause.

Recommendations

- Add a frame rate counter at the beginning of the development work.
- Changes that incur a drop in frame rate should be evaluated and appropriately resolved as a performance bug.

Resources

Documentation

- Performance recommendations for HoloLens apps
- Performance recommendations for Windows MR headset apps
- Hologram stability and framerate
- Asset performance budget
- Performance recommendations for Unity

Tools and tutorials

- [MRToolkit, FPS counter display](#)
- [MRToolkit, Shaders](#)

External references

- [Unity, Optimizing mobile applications](#)

Hologram stability

Stable holograms will increase the usability and believability of your app, and create a more comfortable viewing experience for the user. The quality of hologram stability is a result of good app development and the device's ability to understand (track) its environment. While frame rate is the first pillar of stability, other factors can impact stability including:

- Use of the stabilization plane
- Distance to spatial anchors
- Tracking

Device impact

HOLOLENS	IMMERSIVE HEADSETS
✓ <input type="checkbox"/>	

Quality criteria

BEST	MEETS	FAIL
Holograms consistently appear stable.	Secondary content exhibits unexpected movement; or unexpected movement does not impede overall app experience.	Primary content in frame exhibits unexpected movement.

How to measure

While wearing the device and viewing the experience:

- Move your head from side to side, if the holograms show unexpected movement then low frame rate or improper alignment of the stability plane to the focal plane is the likely cause.
- Move around the holograms and environment, look for behaviors such as swim and jumpiness. This type of motion is likely caused by the device not tracking the environment, or the distance to the spatial anchor.
- If large or multiple holograms are in the frame, observe hologram behavior at various depths while moving your head position from side to side, if shakiness appears this is likely caused by the stabilization plane.

Recommendations

- Add an frame rate counter at the beginning of the development work.
- Use the stabilization plane.
- Always render anchored holograms within 3 meters of their anchor.
- Make sure your environment is setup for proper tracking.

- Design your experience to avoid holograms at various focal depth levels within the frame.

Resources

Documentation

- [Hologram stability and framerate](#)
- [Case study, Using the stabilization plane](#)
- [Performance recommendation for HoloLens apps](#)
- [Spatial anchors](#)
- [Handling tracking errors](#)
- [Stationary frame of reference](#)

Tools and tutorials

- [MR Companion Kit, Kinect IPD](#)

Holograms position on real surfaces

Misalignments of holograms with physical objects (if intended to be placed in relation to one another) is a clear indication of the non-union of holograms and real-world. Accuracy of the placement should be relative to the needs of the scenario; for example, general surface placement can use the spatial map, but more accurate placement will require some use of markers and calibration.

Device impact

HOLOLENS	IMMERSIVE HEADSETS
✓ <input checked="" type="checkbox"/>	

Quality criteria

BEST	MEETS	FAIL
Holograms align to the surface typically in the centimeters to inches range. If more accuracy is required, the app should provide an efficient means for collaboration within the desired app spec.	NA	The holograms appear unaligned with the physical target object by either breaking the surface plane or appearing to float away from the surface. If accuracy is required, Holograms should meet the proximity spec of the scenario.

How to measure

- Holograms that are placed on spatial map should not appear to dramatically float above or below the surface.
- Holograms that require accurate placement should have some form of marker and calibration system that is accurate to the scenario's requirement.

Recommendations

- Spatial map is useful for placing objects on surfaces when precision isn't required.
- For the best precision, use markers or posters to set the holograms and an Xbox controller (or some manual alignment mechanism) for final calibration.
- Consider breaking extra-large holograms into logical parts and aligning each part to the surface.
- Improperly set interpupillary distance (IPD) can also effect hologram alignment. Always configure HoloLens to the user's IPD.

Resources

Documentation

- [Spatial mapping placement](#)

- Room scanning process
- Spatial anchors best practices
- Handling tracking errors
- Spatial mapping in Unity
- Vuforia development overview

Tools and tutorials

- [MR Spatial 230: Spatial mapping](#)
- [MR Toolkit, Spatial Mapping Libraries](#)
- [MR Companion Kit, Poster Calibration Sample](#)
- [MR Companion Kit, Kinect IPD](#)

External references

- [Lowes Case Study, Precision alignment](#)

Viewing zone of comfort

App developers control where users' eyes converge by placing content and holograms at various depths. Users wearing HoloLens will always accommodate to 2.0m to maintain a clear image because HoloLens displays are fixed at an optical distance approximately 2.0m away from the user. Improper content depth can lead to visual discomfort or fatigue.

Device impact

HOLOLENS	IMMERSIVE HEADSETS
✓ <input checked="" type="checkbox"/>	

Quality criteria

Best	<ul style="list-style-type: none"> ● Place content at 2m. ● When holograms cannot be placed at 2m and conflicts between convergence and accommodation cannot be avoided, the optimal zone for hologram placement is between 1.25m and 5m. ● In every case, designers should structure content to encourage users to interact 1+ m away (e.g. adjust content size and default placement parameters). ● Unless specifically not required by the scenario, a clipping plane should be implemented with fadeout starting at 1m. ● In cases where closer observation of a motionless hologram is required, the content should not be closer than 50cm.
Meets	Content is within the viewing and motion guidance, but improper use or no use of the clipping plane.
Fail	Content is presented too close (typically <1.25m, or <50cm for stationary holograms requiring closer observation.)

How to measure

- Content should typically be 2m away, but no closer than 1.25 or further than 5m.
- With few exceptions, the HoloLens clipping render distance should be set to .85CM with fadeout of content starting at 1m. Approach the content and note the clipping plane effect.

- Stationary content should not be closer than 50cm away.

Recommendations

- Design content for the optimal viewing distance of 2m.
- Set the clipping render distance to 85cm with fadeout of content starting at 1m.
- For stationary holograms that need closer viewing, the clipping plane should be no closer than 30cm and fadeout should start at least 10cm away from the clipping plane.

Resources

- [Render distance](#)
- [Focus point in Unity](#)
- [Experimenting with scale](#)
- [Text, Recommended font size](#)

Depth switching

Regardless of viewing zone of comfort issues, demands for the user to switch frequently or quickly between near and far focal objects (including holograms and real-world content) can lead to oculomotor fatigue, and general discomfort.

Device impact

HOLOLENS	IMMERSIVE HEADSETS
✓ <input type="checkbox"/>	✓ <input type="checkbox"/>

Quality criteria

BEST	MEETS	FAIL
Limited or natural depth switching that doesn't cause the user to unnaturally refocus.	Abrupt depth switch this is core and designed into the app experience, or abrupt depth switch that is caused by unexpected real-world content.	Consistent depth switch, or abrupt depth switching that isn't necessary or core to the app experience.

How to measure

- If the app requires the user to consistently and/or abruptly change depth focus, there is depth switching problem.

Recommendations

- Keep primary content at a consistent focal plane and make sure the stabilization plane matches the focal plane. This will alleviate oculomotor fatigue and unexpected hologram movement.

Resources

- [Render distance](#)
- [Focus point in Unity](#)

Use of spatial sound

In Windows Mixed Reality, the audio engine provides the aural component of the mixed reality experience by simulating 3D sound using direction, distance, and environmental simulations. Using spatial sound in an application allows developers to convincingly place sounds in a 3 dimensional space (sphere) all around the user. Those sounds will then seem as if they were coming from real physical objects or the mixed reality holograms in the user's surroundings. Spatial sound is a powerful tool for immersion, accessibility, and UX design in mixed

reality applications.

Device impact

HOLOLENS	IMMERSIVE HEADSETS
✓ <input type="checkbox"/>	✓ <input type="checkbox"/>

Quality criteria

BEST	MEETS	FAIL
Sound is logically spatialized, and the UX appropriately uses sound to assist with object discovery and user feedback. Sound is natural and relevant to objects and normalized across the scenario.	Spatial audio is used appropriately for believability but missing as means to help with user feedback and discoverability.	Sound is not spatialized as expected, and/or lack of sound to assist user within the UX. Or spatial audio was not considered or used in the design of the scenario.

How to measure

- In general, relevant sounds should emit from target holograms (eg, bark sound coming from holographic dog.)
- Sound cues should be used throughout the UX to assist the user with feedback or awareness of actions outside the holographic frame.

Recommendations

- Use spatial audio to assist with object discovery and user interfaces.
- Real sounds work better than synthesize or unnatural sound.
- Most sounds should be spatialized.
- Avoid invisible emitters.
- Avoid spatial masking.
- Normalize all sounds.

Resources

Documentation

- [Spatial sound](#)
- [Spatial sound design](#)
- [Spatial sound in Unity](#)
- [Case study, Spatial sound for HoloTour](#)
- [Case study, Using spatial sound in RoboRaid](#)

Tools and tutorials

- [MR Spatial 220: Spatial sound](#)
- [MRToolkit, Spatial Audio](#)

Focus on holographic frame (FOV) boundaries

Well-designed user experiences can create and maintain useful context of the virtual environment that extends around the users. Mitigating the effect of the FOV boundaries involves a thoughtful design of content scale and context, use of spatial audio, guidance systems, and the user's position. If done right, the user will feel less impaired by the FOV boundaries while having a comfortable app experience.

Device impact

HOLOLENS	IMMERSIVE HEADSETS
----------	--------------------

<input checked="" type="checkbox"/>	
-------------------------------------	--

Quality criteria

BEST	MEETS	FAIL
User never loses context and viewing is comfortable. Context assistance is provided for large objects. Discoverability and viewing guidance is provided for objects outside the frame. In general, motion design and scale of the holograms are appropriate for a comfortable viewing experience.	User never loses context, but extra neck motion may be required in limited situations. In limited situations scale causes holograms to break either the vertical or horizontal frame causing some neck motion to view holograms.	User likely to lose context and/or consistent neck motion is required to view holograms. No context guidance for large holographic objects, moving objects easy to lose outside the frame with no discoverability guidance, or tall holograms requires regular neck motion to view.

How to measure

- Context for a (large) hologram is lost or not understood due to being clipped at the boundaries.
- Location of holograms are hard to find due to the lack of attention directors or content that rapidly moves in and out of the holographic frame.
- Scenario requires regular and repetitive up and down head motion to fully see a hologram resulting in neck fatigue.

Recommendations

- Start the experience with small objects that fit the FOV, then transition with visual cues to larger versions.
- Use spatial audio and attention directors to help the user find content that is outside the FOV.
- As much as possible, avoid holograms that vertically clip the FOV.
- Provide the user with in-app guidance for best viewing location.

Resources

Documentation

- [Holographic frame](#)
- [Case Study, HoloStudio UI and interaction design learnings](#)
- [Scale of objects and environments](#)
- [Cursors, Visual cues](#)

External references

- [Much ado about the FOV](#)

Content reacts to user position

Holograms should react to the user position in roughly the same ways that "real" objects do. A notable design consideration is UI elements that can't necessarily assume a user's position is stationary and adapt to the user's motion. Designing an app that correctly adapts to user position will create a more believable experience and make it easier to use.

Device impact

HOLOLENS	IMMERSIVE HEADSETS
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

Quality criteria

Best	Content and UI adapt to user positions allowing user to naturally interact with content within the scope of expected user movement.
Meets	UI adapts to the user position, but may impede the view of key content requiring the user to adjust their position.
Fail	<ol style="list-style-type: none"> 1. UI elements are lost or locked during movement causing user to unnaturally return to (or find) controls. 2. UI elements limit the view of primary content. 3. UI movement is not optimized for viewing distance and momentum particularly with tag-along UI elements.

How to measure

- All measurements should be done within a reasonable scope of the scenario. While user movement will vary, don't try to trick the app with extreme user movement.
- For UI elements, relevant controls should be available regardless of user movement. For example, if the user is viewing and walking around a 3D map with zoom, the zoom control should be readily available to the user regardless of location.

Recommendations

- The user is the camera and they control the movement. Let them drive.
- Consider billboarding for text and menuing systems that would otherwise be world-locked or obscured if a user were to move around.
- Use tag-along for content that needs to follow the user while still allowing the user to see what is in front of them.

Resources

Documentation

- [Interaction design](#)
- [Color, light, and material](#)
- [Billboarding and tag-along](#)
- [Interaction fundamentals](#)
- [Self-motion and user locomotion](#)

Tools and tutorials

- [MR Input 210: Gaze](#)

Input interaction clarity

Input interaction clarity is critical to an app's usability and includes input consistency, approachability, discoverability of interaction methods. User should be able to use platform-wide common interactions without relearning. If the app has custom input, it should be clearly communicated and demonstrated.

Device impact

HOLOLENS	IMMERSIVE HEADSETS
✓ <input type="checkbox"/>	✓ <input type="checkbox"/>

Quality criteria

BEST	MEETS	FAIL
Input interaction methods are consistent with Windows Mixed Reality provided guidance . Any custom input should not be redundant with standard input (rather use standard interaction) and must be clearly communicated and demonstrated to the user.	Similar to best, but custom inputs are redundant with standard input methods. User can still achieve the goal and progress through the app experience.	Difficult to understand input method or button mapping. Input is heavily customized, does not support standard input, no instructions, or likely to cause fatigue and comfort issues.

How to measure

- The app uses consistent [standard input methods](#).
- If the app has custom input, it is clearly communicated through:
 - First-run experience
 - Introductory screens
 - Tooltips
 - Hand coach
 - Help section
 - Voice over

Recommendations

- Use standard input methods whenever possible.
- Provide demonstrations, tutorials, and tooltips for non-standard input methods.
- Use a consistent interaction model throughout the app.

Resources

Documentation

- [Windows MR interaction fundamentals](#)
- [Interactable objects](#)
- [Gaze targeting](#)
- [Cursors](#)
- [Comfort and gaze](#)
- [Gestures](#)
- [Voice input](#)
- [Voice design](#)
- [Motion controllers](#)
- [Input porting guide for Unity](#)
- [Keyboard input in Unity](#)
- [Gaze in Unity](#)
- [Gestures and motion controllers in Unity](#)
- [Voice input in Unity](#)
- [Keyboard, mouse, and controller input in DirectX](#)
- [Gaze, gesture, and motion controllers in DirectX](#)
- [Voice input in DirectX](#)

Tools and tutorials

- [Case study: The pursuit of more personal computing](#)
- [Cast study: HoloStudio UI and interaction design learnings](#)
- [Sample app: Periodic table of the elements](#)
- [Sample app: Lunar module](#)

- [MR Input 210: Gaze](#)
- [MR Input 211: Gestures](#)
- [MR Input 212: Voice](#)

Interactable objects

A button has long been a metaphor used for triggering an event in the 2D abstract world. In the three-dimensional mixed reality world, we don't have to be confined to this world of abstraction anymore. Anything can be an Interactable object that triggers an event. An interactable object can be represented as anything from a coffee cup on the table to a balloon floating in the air. Regardless of the form, interactable objects should be clearly recognizable by the user through visual and audio cues.

Device impact

HOLOLENS	IMMERSIVE HEADSETS
✓ <input type="checkbox"/>	✓ <input type="checkbox"/>

Quality criteria

BEST	MEETS	FAIL
Regardless of form, interactable objects are recognizable through visual and audio cues across three states: idle, targeted, and selected. "See it, say it" is clear and consistently used throughout the experience. Objects are scaled and distributed to allow for error free targeting.	User can recognize object as interactable through audio or visual feedback, and can target and activate the object.	Given no visual or audio cues, user cannot recognize an interactable object. Interactions are error prone due to object scale or distance between objects.

How to measure

- Interactable objects are recognizable as 'interactable'; including buttons, menus, and app specific content. As a rule of thumb there should be a visual and audio cue when targeting interactable objects.

Recommendations

- Use visual and audio feedback for interactions.
- Visual feedback should be differentiated for each input state (idle, targeted, selected)
- Interactable objects should be scaled and placed for error free targeting.
- Grouped interactable objects (such as a menu bar or list) should have proper spacing for targeting.
- Buttons and menus that support voice command should provide text labels for the command keyword ("See it, say it")

Resources

Documentation

- [Interactable object](#)
- [Text in Unity](#)
- [App bar and bounding box](#)
- [Voice design](#)

Tools and tutorials

- [Mixed Reality Toolkit - UX](#)

Room scanning

Apps that require spatial mapping data rely on the device to automatically collect this data over time and across sessions as the user explores their environment with the device active. The completeness and quality of this data depends on a number of factors including the amount of exploration the user has done, how much time has passed since the exploration and whether objects such as furniture and doors have moved since the device scanned the area. Many apps will analyze the spatial mapping data at the start of the experience to judge whether the user should perform additional steps to improve the completeness and quality of the spatial map. If the user is required to scan the environment, clear guidance should be provided during the scanning experience.

Device impact

HOLOLENS	IMMERSIVE HEADSETS
✓ <input type="checkbox"/>	

Quality criteria

BEST	MEETS	FAIL
Visualization of the spatial mesh tell users scanning is in progress. User clearly knows what to do and when the scan starts and stops.	Visualization of the spatial mesh is provided, but the user may not clearly know what to do and no progress information is provided.	No visualization of mesh. No guidance information provided to the user regarding where to look, or when the scan starts/stops.

How to measure

- During a required room scan, visual and audio guidance is provided indicating where to look, and when to start and stop scanning.

Recommendations

- Indicate how much of the total volume in the users vicinity needs to be part of the experience.
- Communicate when the scan starts and stops such as a progress indicator.
- Use a visualization of the mesh during the scan.
- Provide visual and audio cues to encourage the user to look and move around the room.
- Inform the user where to go to improve the data. In many cases, it may be best to tell the user what they need to do (e.g. look at the ceiling, look behind furniture), in order to get the necessary scan quality.

Resources

Documentation

- [Room scan visualization](#)
- [Case study: Expanding the spatial mapping capabilities of HoloLens](#)
- [Case study: Spatial sound design for HoloTour](#)
- [Case study: Creating an immersive experience in Fragments](#)

Tools and tutorials

- [Mixed Reality Toolkit - UX](#)

Directional indicators

In a mixed reality app, content may be outside the field of view or occluded by real-world objects. A well designed app will make it easier for the user to find non-visible content. Directional indicators alert a user to important content and provide guidance to the content relative to the user's position. Guidance to non-visible content can take the form of sound emitters, directional arrows, or direct visual cues.

Device impact

HOLOLENS	IMMERSIVE HEADSETS
----------	--------------------

<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
-------------------------------------	-------------------------------------

Quality criteria

BEST	MEETS	FAIL
Visual and audio cues directly guide the user to relevant content outside the field of view.	An arrow or some indicator that points the user in the general direction of the content.	Relevant content is outside of the field of view, and poor or no location guidance is provided to the user.

How to measure

- Relevant content outside of the user field of view is discoverable through visual and/or audio cues.

Recommendations

- When relevant content is outside the user's field of view, use directional indicators and audio cues to guide the user to the content. In many cases, a direct visual guide is preferred over directional arrows.
- Directional indicators should not be built into the cursor.

Resources

- [Holographic frame](#)

Data loading

A progress control provides feedback to the user that a long-running operation is underway. It can mean that the user cannot interact with the app when the progress indicator is visible and can also indicate how long the wait time might be.

Device impact

HOLOLENS	IMMERSIVE HEADSETS
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

Quality criteria

BEST	MEETS	FAIL
Animated visual indicator, in the form of a progress bar or ring, showing progress during any data loading or processing. The visual indicator provides guidance on how long the wait could be.	User is informed that data loading is in progress, but there is no indication of how long the wait could be.	No data loading or process indicators for task taking longer than 5 seconds.

How to measure

- During data loading verify there is no blank state for more than 5 seconds.

Recommendations

- Provide a data loading animator showing progress in any situation when the user may perceive this app to have stalled or crashed. A reasonable rule of thumb is any 'loading' activity that could take more than 5 seconds.

Resources

- [Displaying progress](#)

SpectatorView for HoloLens

11/6/2018 • 18 minutes to read • [Edit Online](#)



When wearing a HoloLens, we often forget that a person who does not have it on is unable to experience the wonders that we can. SpectatorView allows others to see on a 2D screen what a HoloLens user sees in their world. SpectatorView (Preview) is fast and affordable approach to recording holograms in HD, while SpectatorView Pro is intended for professional quality recording of holograms.

The following table shows both options and their capabilities. Choose the option that best fits your video recording needs:

	SPECTATORVIEW (PREVIEW)	SPECTATORVIEW PRO
HD quality	Full HD	Professional quality filming (as determined by DSLR)
Easy camera movement	✓	
Third-person view	✓	✓
Can be streamed to screens	✓	✓
Portable	✓	
Wireless	✓	
Additional required hardware	iPhone (or iPad)	HoloLens + Rig + Tripod + DSLR + PC + Unity
Hardware investment	Low	High

	SPECTATORVIEW (PREVIEW)	SPECTATORVIEW PRO
Cross-platform	iOS	
Viewer can interact	✓	
Networking required (UNET scripting)	✓	✓
Runtime setup duration	Instant	Slow

SpectatorView (Preview)



Use cases

- Filming holograms in HD: Using SpectatorView (Preview), you can record a mixed reality experience using an iPhone. Record in full HD and apply anti-aliasing to holograms and even shadows. It is a cost-effective and quick way to capture video of holograms.
- Live demos: Stream live mixed reality experiences to an Apple TV directly from your iPhone or iPad, lag-free!
- Share the experience with guests: Let non-HoloLens users experience holograms directly from their phones or tablets.

Current features

- Network auto-discovery for adding phones to the session.
- Automatic session handling, so users are added to the correct session.
- Spatial synchronization of Holograms, so everyone sees holograms in the exact same place.
- iOS support (ARKit-enabled devices).
- Multiple iOS guests.
- Recording of video + holograms + ambient sound + hologram sound.
- Share sheet so you can save video, email it, or share with other supporting apps.

NOTE

The SpectatorView (Preview) code cannot be used with the SpectatorView Pro version code. We recommend to implement it in new projects where video recording of holograms is required.

Licenses

- OpenCV - (3-clause BSD License) <https://opencv.org/license.html>
- Unity ARKit - (MIT License) https://bitbucket.org/Unity-Technologies/unity-arkit-plugin/src/3691df77caca2095d632c5e72ff4ffa68ced111f/LICENSES/MIT_LICENSE?at=default&fileviewer=file-view-default

How to set up SpectatorView (Preview)

Requirements

- SpectatorView plugin and required OpenCV binaries, which can be found at <https://github.com/Microsoft/MixedRealityToolkit/tree/master/SpectatorViewPlugin>. Details on how to build the SpectatorView Native Plugin can be found below. From the generated binaries you will need:
 - opencv_aruco341.dll
 - opencv_calib3d341.dll
 - opencv_core341.dll
 - opencv_features2d341.dll
 - opencv_flann341.dll
 - opencv_imgproc341.dll
 - zlib1.dll
 - SpectatorViewPlugin.dll
- SpectatorView uses Unity Networking (UNET) for its network discovery and spatial syncing. This means all interactivity during the application needs to be synced between the devices.
- Unity 2017.2.1p2 or later
- Hardware
 - A HoloLens
 - Windows PC running Windows 10
 - ARKit compatible device (iPhone 6s onwards / iPad Pro 2016 onwards / iPad 2017 onwards) - running iOS 11 or above
 - Mac with xcode 9.2 onwards
- Apple developer account, free or paid (<https://developer.apple.com/>)
- Microsoft Visual Studio 2017
- **Optional:** - UnityARKitPlugin. The required components of this plugin are already included in the MixedRealityToolkit-Unity project. The entire ARKit plugin can be downloaded from the asset store here: <https://assetstore.unity.com/packages/essentials/tutorial-projects/unity-arkit-plugin-92515>

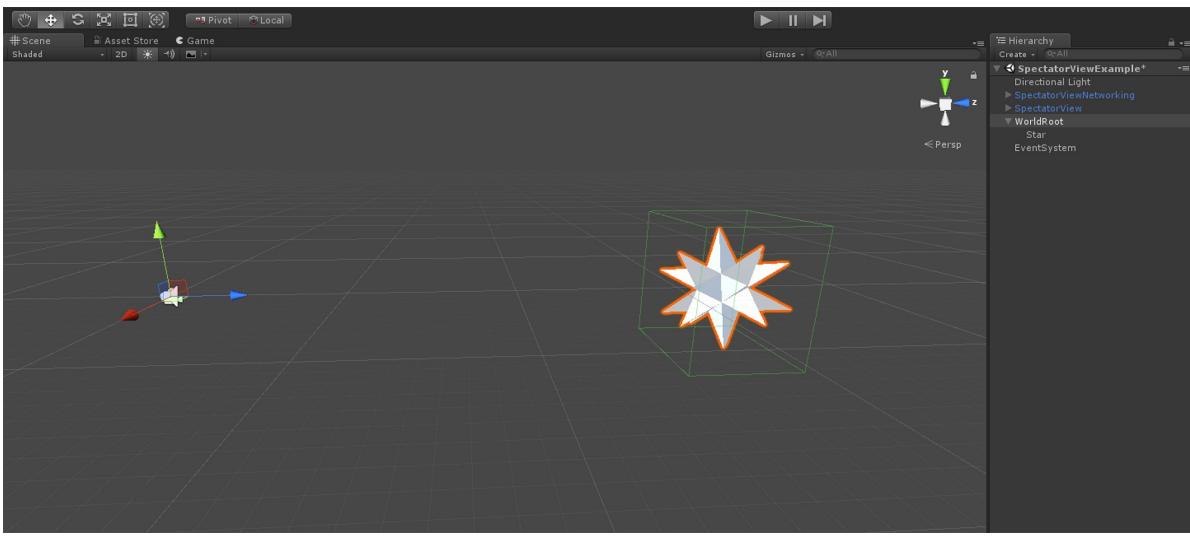
Building the SpectatorView native plugin

To generate the required files, follow these steps:

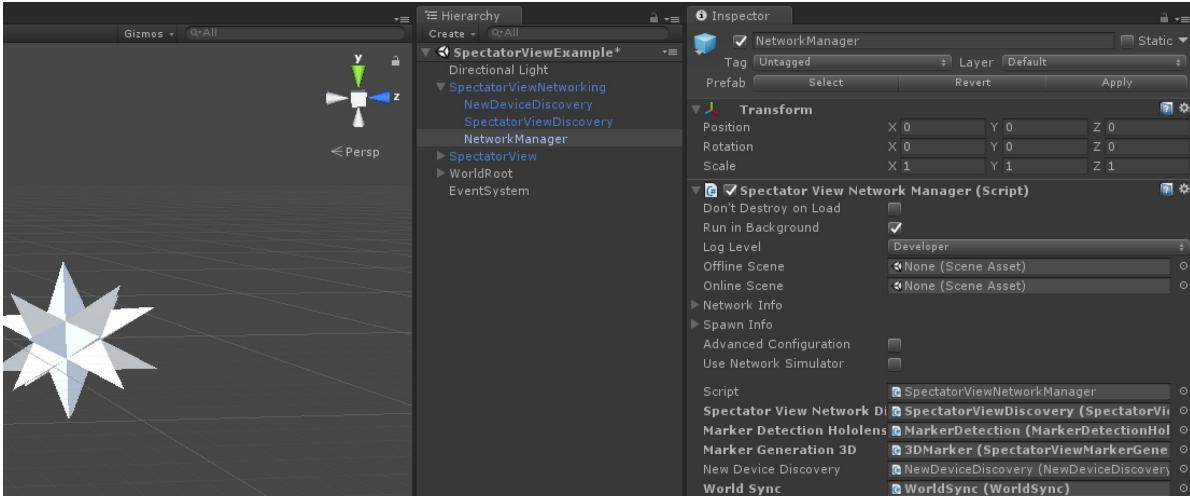
<https://github.com/Microsoft/MixedRealityToolkit/blob/master/SpectatorViewPlugin/README.md>

Project setup

1. Prepare your scene, ensuring all visable gameobjects within your scene are contained under a world root gameobject.



2. Add the SpectatorView prefab ([Assets/HoloToolkit-Preview/SpectatorView/Prefabs/SpectatorView.prefab](#)) into your scene.
3. Add the SpectatorViewNetworking prefab ([Assets/HoloToolkit-Preview/SpectatorView/Prefabs/SpectatorViewNetworking.prefab](#)) into your scene.
4. Select the SpectatorViewNetworking gameobject and on the SpectatorViewNetworkingManager component, there's a few things you can link up. If left untouched this component will search for necessary scripts at runtime.
 - Marker Detection HoloLens -> SpectatorView/Hololens/MarkerDetection
 - Marker Generation 3D -> SpectatorView/IPhone/SyncMarker/3DMarker



Networking your app

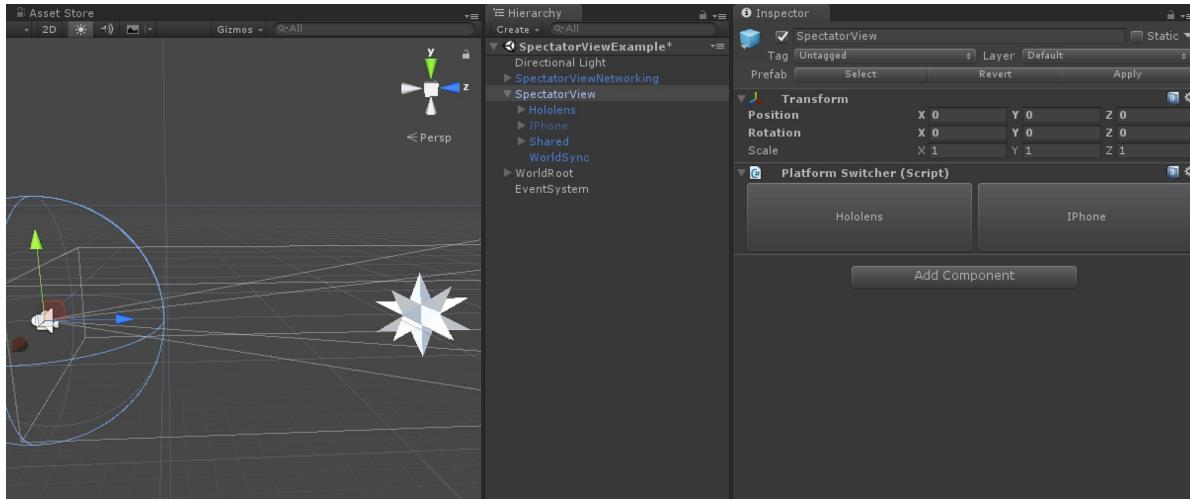
- SpectatorView uses UNET for its networking and manages all host-client connections for you.
- Any app specific data has to be synced and implemented by you, using e.g. SyncVars, NetworkTransform, NetworkBehavior.
- For more information and tutorials on Unity Networking please visit <https://unity3d.com/learn/tutorials/s/multiplayer-networking>

Building for each platform (HoloLens or iOS)

- When building for iOS, ensure you remove the GLTF component of MRTK as this is not yet compatible with this platform.
- At the top level of the SpectatorView prefab there is a component called 'Platform Switcher'. Select the platform you want to build for.
 - If selecting 'Hololens' you should see all gameobjects beneath the iPhone gameobject in the

SpectatorView prefab become inactive and all the gameobjects under 'Hololens' become active.

- This can take a little while as depending on the platform you choose the HoloToolkit is being added or removed from the project.

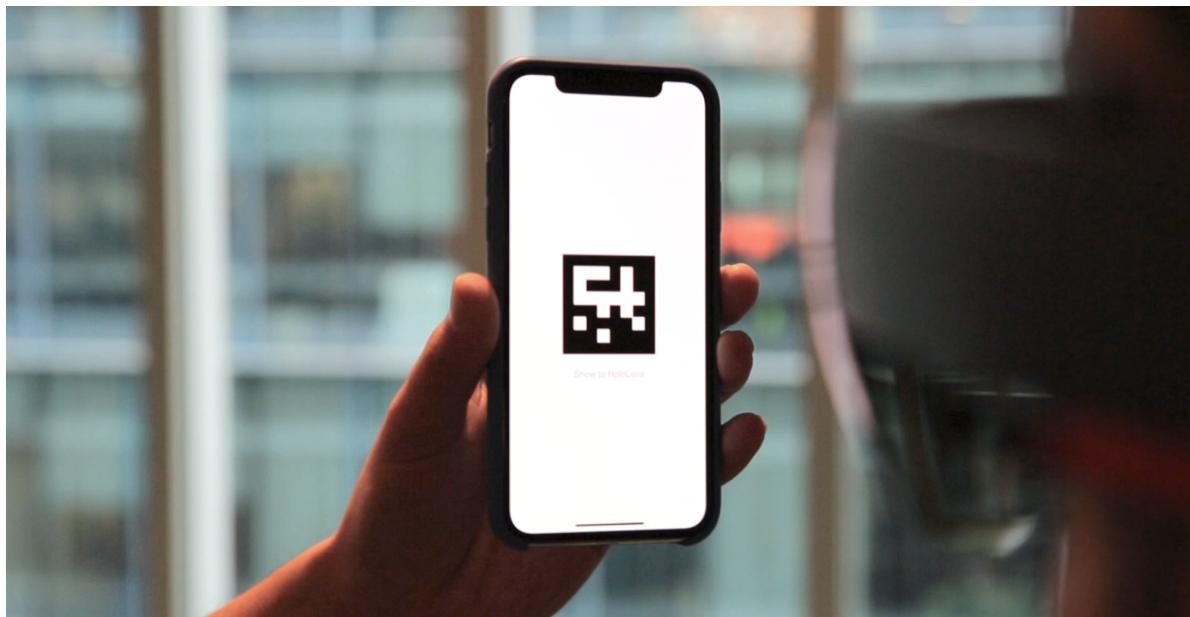


- Ensure you build all versions of the application using the same Unity editor instance (do not close Unity between builds) due to an unresolved issue with Unity.

Running your app

Once you have built and deployed a version of your application on iPhone and on HoloLens, you should be able to connect them.

1. Ensure that both devices are on the same Wi-Fi network.
2. Start the application on the both devices, in no specific order. The process of starting the application on the iPhone should trigger the HoloLens camera to turn on and begin taking pictures.
3. As soon as iPhone app starts, it will look for surfaces like floors or tables. When surfaces are found, you should see a marker similar to the one below. Show this marker to the HoloLens.



4. Once the marker has been detected by the HoloLens it should disappear and both devices should be connected and spatially synced.

Recording video

1. To capture and save a video from the iPhone, tap and hold the screen for 1 second. This will open the recording menu.

2. Tap the red record button, this will start a countdown before beginning to record the screen.
3. To finish recording tap and hold the screen for another 1 second, then tap the stop button.
4. Once the recorded video loads, a Preview button (blue button) will appear, tap to watch the recorded video.
5. Open the Share sheet and select Save to camera roll.

Example scene

An example scene can be found in [HoloToolkit-Examples\SpectatorView\Scenes\SpectatorViewExample.unity](#)

Troubleshooting

The Unity Editor won't connect to a session hosted by a HoloLens device

- This could be the Windows Firewall.
- Go to Windows Firewall options.
- Allow an app or feature through Windows Firewall.
- For all instances of Unity Editor in the list tick, Domain, Private and Public.
- Then go back to Windows Firewall options.
- Click Advanced settings.
- Click Inbound Rules.
- All instances of Unity Editor should have a green tick.
- For any instances of Unity Editor that don't have a green tick:
 - Double click it.
 - In the action dialog select Allow the connection.
 - Click OK.
 - Restart the Unity Editor.

At runtime the iPhone screen says "Locating Floor..." but does not display an AR marker.

- The iPhone is looking for a horizontal surface so try pointing the iPhone camera towards the floor or a table. This will help ARKit find surfaces necessary to spatially sync with HoloLens.

HoloLens camera does not turn on automatically when iPhone tries to join.

- Make sure both HoloLens and iPhone are running on the same Wi-Fi network.

When launching an SpectatorView application on a mobile device, other hololens running other SpectatorView apps turn on their camera

- Goto the NewDeviceDiscovery component and change the both the Broadcast Key and Broadcast port to two unique values.
- Go to SpectatorViewDiscovery and change the Broadcast Key and Broadcast port to another set of unique numbers.

The HoloLens won't connect with the mac Unity Editor.

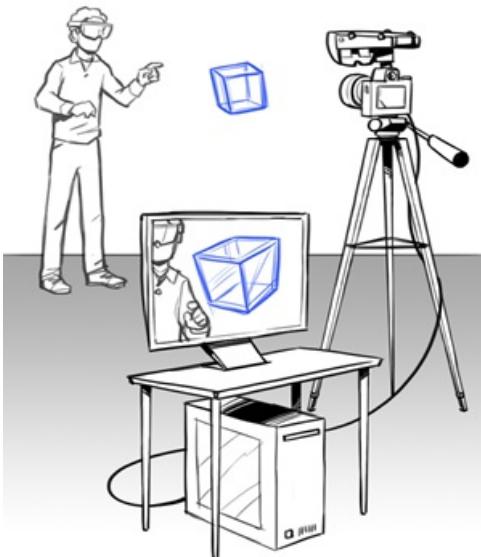
- This is a known issue which could be linked to the Unity version. Building to the iPhone should still work and connect as normal.

The HoloLens camera turns on but is not able to scan the marker.

- Ensure that you build all versions of the application using the same Unity Editor instance (do not close Unity between builds). This is due to an unknown issue with Unity.

SpectatorView Pro

SpectatorView Pro setup



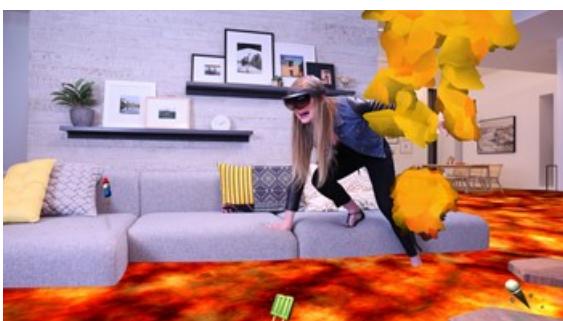
Using SpectatorView Pro involves these four components:

1. An app built specifically to enable spectator view, which is based on [shared experiences in mixed reality](#).
2. A user wearing HoloLens using the app.
3. A spectator view camera rig providing a third-person perspective video.
4. A desktop PC running the shared experience app and compositing the holograms into a spectator view video.



Use cases

SpectatorView Pro photo capture scenario example



SpectatorView Pro video capture scenario example



There are three key scenarios that work well with this technology:

1. **Photo capture** Using this technology, you can capture high resolution images of your holograms. These images can be used to showcase content at marketing events, send to your potential clients, or even submit your application to the Windows Store. You get to decide which photo camera you would like to use to capture these images, as such you may prefer a quality DSLR camera.
2. **Live demonstrations** Spectator view is a preferred approach for live demonstrations as the camera position remains steady or controlled. Because you can use a high-quality video camera, you can also produce high-quality images meant for a big screen. This is also appropriate for streaming live demos on a screen, possibly to eager participants waiting in line for their turn.
3. **Video capture** Videos are the best story telling mechanism for sharing a holographic app experience with many people. Spectator view lets you choose the camera, lens, and framing that best suits how you want to showcase your app. It puts you in control of the video quality based on the video hardware you have available.

Comparing video capture techniques

[Mixed reality capture](#) (MRC) provides a video composite of what the HoloLens user is seeing from a first person point-of-view. Spectator view produces a video from a third-person perspective, allowing the video observer to see the environment with holograms and the user wearing a HoloLens device. Because you have a choice of camera, spectator views can also produce higher resolution and better quality images than the built-in HoloLens camera used for MRC images. For this reason, spectator view is better suited for app images in the Windows Store, marketing videos, or for projecting a live view for an audience.

SpectatorView Pro professional camera used in Microsoft keynote presentations



Spectator view has been an essential piece of how Microsoft HoloLens has presented experiences to audiences since the very beginning when the product was announced in January 2015. The professional setup used had high demands and an expensive price tag to go with it. For example, the camera uses a genlock signal to ensure precise timing that coordinates with the HoloLens tracking system. In this setup, moving the spectator view camera was possible while keeping holograms stable to match the experience of someone who is seeing the experience directly in HoloLens.

The open-source version of spectator view trades off the ability to move the camera rig in order to dramatically lower the cost of the overall setup. This project uses an external camera rigidly mounted to a HoloLens to take

high-definition pictures and video of your holographic Unity project. **During live demonstrations, the camera should remain in a fixed position.** Movement of the camera can lead to hologram jitter or drift. This is because the timing of the video frame and the rendering of holograms on the PC may not be precisely synchronized. Therefore, keeping the camera steady or limiting movement will produce a result close to what the person wearing a HoloLens can see.

To make your app ready for spectator view, you'll need to build a [shared experience](#) app and ensure the app can run on both HoloLens as well as desktop in the Unity editor. The desktop version of the app will have additional components built in that composite the video feed with the rendered holograms.

How to set up SpectatorView Pro

Requirements

Hardware shopping list



Below is a recommended list of hardware, but you can experiment with other compatible units too.

| Hardware component | Recommendation || --- | --- || A PC configuration that works for holographic development with the HoloLens emulator. || | Camera with HDMI out or photo capture SDK. | For photo and video capture, we have tested the [Canon EOS 5D Mark III](#) camera. For live demonstrations, we have tested the [Blackmagic Design Production Camera 4K](#). Note, any camera with HDMI out (e.g. GoPro) should work. Many of our videos use the [Canon EF 14mm f/2.8L II USM Ultra-Wide Angle Fixed Lens](#), but you should choose a camera lens that meets your needs. || | Capture card for your PC to get color frames from your camera to calibrate your rig and preview your composite scene. | We have tested the [Blackmagic Design Intensity Pro 4K capture card](#). || Cables | [HDMI to Mini HDMI](#) for attaching your camera to your capture card. Ensure you purchase an HDMI form factor that fits your camera. (GoPro, for instance, outputs over [Micro HDMI](#).)

[HDMI cable](#) for viewing the composite feed on a preview monitor or television. || | Machined aluminum bracket to connect your HoloLens to the camera. More details can be found in the OSS project README. || | 3D printed adapter to connect the HoloLens mount to the camera hotshoe. More details can be found in the OSS project README. || | Hotshoe fastener to mount the hotshoe adapter. | [Hotshoe Fastener](#) || Assorted nuts, bolts, and tools. | [1/4-20" Nuts](#)

[1/4-20" x 3/4" Bolts](#)

[7/16 Nut Driver](#)

[T15 Torx](#)

[T7 Torx](#) |

Software components

1. Software downloaded from the [GitHub project for spectator view](#).
2. [Blackmagic Capture Card SDK](#).
Search for Desktop Video Developer SDK in "Latest Downloads".
3. [Blackmagic Desktop Video Runtime](#).
Search for Desktop Video Software Update in "Latest Downloads".
Ensure the version number matches the SDK version.
4. [OpenCV 3.1](#) For calibration or video capture without the Blackmagic capture card.

5. [Canon SDK](#) (Optional).

If you are using a Canon camera and have access to the Canon SDK, you can tether your camera to your PC to take higher resolution images.

6. Unity for your holographic app development.

Supported version can be found in the OSS project.

7. Visual Studio 2015 with latest updates.

Building your own SpectatorView Pro camera

NOTICE & DISCLAIMER: When making modifications to your HoloLens hardware (including, but not limited to, setting up your HoloLens for "spectator view") basic safety precautions should always be observed. Read all instructions and manuals before making any modifications. It is your responsibility to follow all instructions and use tools as directed. You may have purchased or licensed your HoloLens with a limited warranty or no warranty. Please read your applicable [HoloLens License Agreement or Terms of Use and Sale](#) to understand your warranty options.

Rig Assembly

Assembled SpectatorView Pro rig with HoloLens and DSLR camera



- Use a T7 screwdriver to remove the headband from the HoloLens. Once the screws are loose, poke them out with a paperclip from the other side.
- Remove the screw cap on the inside front of the HoloLens visor with a small flat head screwdriver.
- Use a T15 screwdriver to remove the small torx bolts from the HoloLens bracket to remove the U and Hook-shaped attachments.
- Place the HoloLens on the bracket, lining up the exposed hole on the inside of the visor with the extrusion on the front of the bracket. The HoloLens' arms should be kept in place by the pins on the bottom of the bracket.
- Reattach the U and Hook-shaped attachments to secure the HoloLens to the bracket.
- Attach the hotshoe fastener to the hotshoe of your camera.
- Attach the mount adapter to the hotshoe fastener.
- Rotate the adapter so the narrow side is facing forward and parallel to the camera's lens.
- Secure the adapter in place with a 1/4" nut using the 7/16 nut driver.
- Position the bracket against the adapter so the front of the HoloLens' visor is as close as possible to the front of the camera's lens.
- Attach the bracket with 4 1/4" nuts and bolts using the 7/16 nut driver.

PC Setup

- Install the software from the software components section.
- Add the capture card to an open PCIe slot on your motherboard.
- Plug an HDMI cable from your camera to the outer HDMI slot (HDMI-In) on the capture card.

- Plug an HDMI cable from the center HDMI slot (HDMI-Out) on the capture card to an optional preview monitor.

Camera Setup

- Change your camera to Video Mode so it outputs at the full 1920x1080 resolution rather than a cropped 3:4 photo resolution.
- Find your camera's HDMI settings and enable **Mirroring** or **Dual Monitor**.
- Set the output resolution to 1080P.
- Turn off **Live View On Screen Display** so any screen overlays do not appear in the composite feed.
- Turn on your camera's **Live View**.
- If using the **Canon SDK** and would like to use a flash unit, disable **Silent LV Shoot**.
- Plug an HDMI cable from the camera to the outer HDMI slot (HDMI-In) on the capture card.

Calibration

After setting up your spectator view rig, you must calibrate in order to get the position and rotation offset of your camera to your HoloLens.

- Open the Calibration Visual Studio solution under Calibration\Calibration.sln.
- In this solution, you will find the file dependencies.props which creates macros for the inc locations of the 3rd party sources.
- Update this file with the location you installed OpenCV 3.1, the Blackmagic SDK, and the Canon SDK (if applicable)

Dependency locations snapshot in Visual Studio

```

dependencies.props
1 <?xml version="1.0" encoding="utf-8"?>
2 <Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
3   <ImportGroup Label="PropertySheets" />
4   <PropertyGroup Label="UserMacros" />
5     <OpenCV_vc14>C:\OpenCV\build\x64\vc14</OpenCV_vc14>
6     <DeckLink_inc>C:\BlackMagic\Blackmagic DeckLink SDK 10.6.6\Win\include</DeckLink_inc>
7     <Canon_SDK>C:\CanonSDK\Windows\EDSDK</Canon_SDK>
8   </PropertyGroup>
9   <PropertyGroup />
10  <ItemDefinitionGroup />
11  <ItemGroup />
12 </Project>

```

- Print out the calibration pattern Calibration\CalibrationPatterns\2_66_grid_FULL.png on a flat, rigid surface.
- Plug your HoloLens into your PC over USB.
- Update the preprocessor definitions **HOLOLENS_USER** and **HOLOLENS_PWD** in stdafx.h with your HoloLens' device portal credentials.
- Attach your camera to your capture card over HDMI and turn it on.
- Run the Calibration solution.
- Move the checkerboard pattern around the view like this:

Calibrating the SpectatorView Pro rig

Calibrating the SpectatorView Pro rig

- A picture will automatically be taken when a checkerboard is in view. Look for the white light on the HoloLens' visor before advancing to the next pose.
- When finished, press **Enter** with the Calibration app in focus to create a **CalibrationData.txt** file.
- This file will be saved to **Documents\CalibrationFiles\CalibrationData.txt**
- Inspect this file to see if your calibration data is accurate:
 - **DSLR RMS** should be close to 0.
 - **HoloLens RMS** should be close to 0.
 - **Stereo RMS** may be 20-50, this is acceptable since the field of view between the two cameras may be

different.

- **Translation** is the distance from the HoloLens' camera to the attached camera's lens. This is in meters.
- **Rotation** should be close to identity.
- **DSLR_fov** y value should be close to the vertical field of view expected from your lens' focal length and any camera body crop factor.
- If any of the above values do not appear to make sense, recalibrate.
- Copy this file to the **Assets** directory in your Unity project.

Compositor

The compositor is a Unity extension that runs as a window in the Unity editor. To enable this, the Compositor Visual Studio solution first needs to be built.

- Open the Compositor Visual Studio solution under Compositor\Compositor.sln
- Update dependencies.props with the same criteria from the Calibration solution above. If you followed the calibration steps, this file will already have been updated.
- Build the entire solution as Release and the architecture that matches your Unity version's architecture. If in doubt, build both x86 and x64.
- If you built the solution for x64, also build the SpatialPerceptionHelper project as x86 since it will run on the HoloLens.
- Close Unity if it is running your application. Unity needs to be relaunched if DLLs are changed at runtime.
- Run Compositor\CopyDLL.cmd to copy the DLLs built from this solution to your Unity project. This script will copy the DLLs to the included sample project. Once you have your own project set up, you can run CopyDLL with a command line argument pointing to your project's Assets directory to copy there as well.
- Launch the sample Unity app.

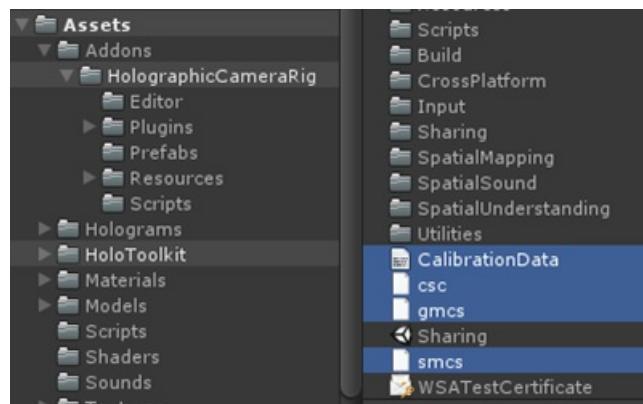
Unity app

The compositor runs as a window in the Unity Editor. The included sample project has everything set up to work with spectator view once the compositor DLLs are copied.

Spectator view requires the application to be run as a [shared experience](#). This means that any application state changes that happen on the HoloLens need to be networked to update the app running in Unity too.

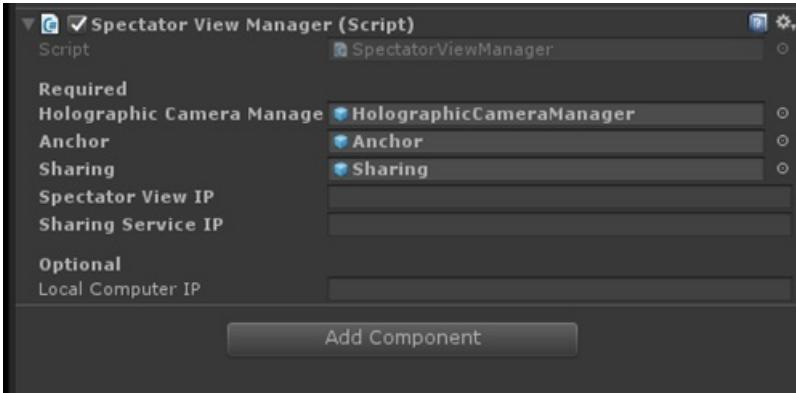
If starting from a new Unity project, you will need to do some setup first:

- Copy **Assets/Addons/HolographicCameraRig** from the sample project to your project.
- Add the latest MixedRealityToolkit to your project, including **Sharing**, **csc.rsp**, **gmcs.rsp**, and **smcs.rsp**.
- Add your **CalibrationData.txt** file to your Assets directory.

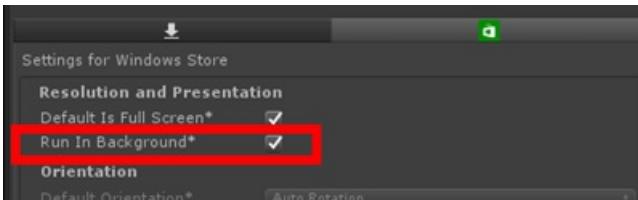


- Add the **HolographicCameraRig\Prefabs\SpectatorViewManager** prefab to your scene and fill in the fields:
 - **HolographicCameraManager** should be populated with the HolographicCameraManager prefab from the HolographicCameraRig prefab directory.

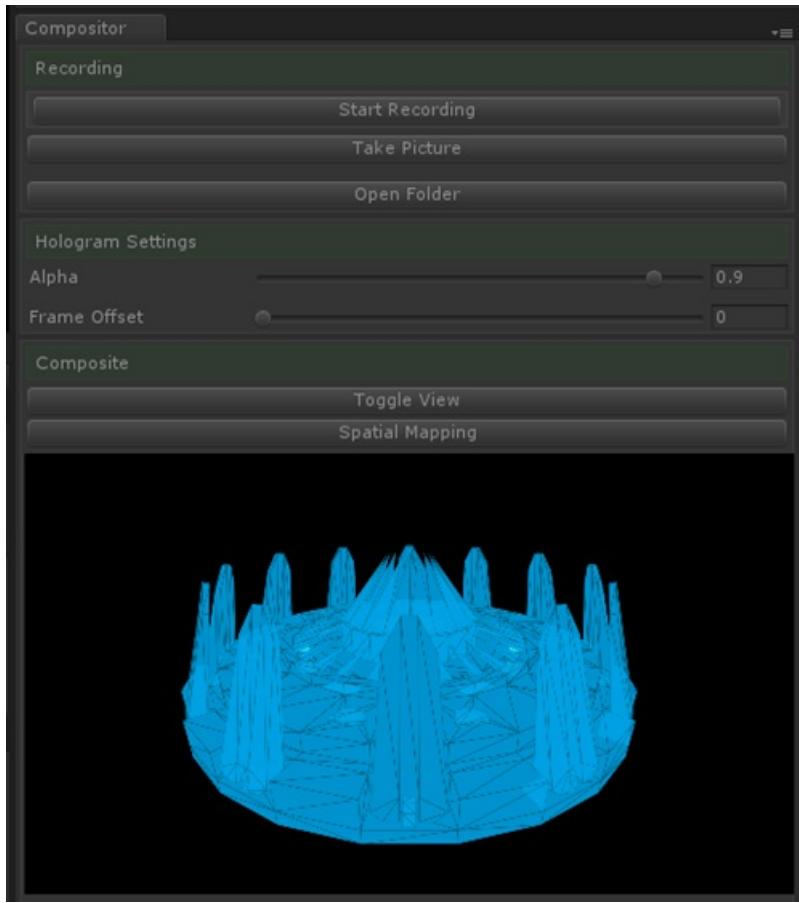
- **Anchor** should be populated with the Anchor prefab from the HolographicCameraRig prefab directory.
- **Sharing** should be populated with the Sharing prefab from the MixedRealityToolkit.
- Note: If any of these prefabs already exist in your project hierarchy, the existing prefabs will be used instead of these ones.
- **Spectator View IP** should be the IP of your HoloLens attached to your spectator view rig.
- **Sharing Service IP** should be the IP of the PC running the MixedRealityToolkit SharingService.
- Optional: If you have multiple spectator view rigs attached to multiple PC's, **Local Computer IP** should be set with the PC the spectator view rig will communicate with.



- Start the MixedRealityToolkit **Sharing Service**
- Build and deploy the app as a D3D UWP to the HoloLens attached to the spectator view rig.
- Deploy the app to any other HoloLens devices in the experience.
- Check the **Run In Background** checkbox under **edit/project settings/player**.



- Launch the compositor window under **Spectator View/Compositor**



- This window allows you to:
 - Start recording video
 - Take a picture
 - Change hologram opacity
 - Change the frame offset (which adjusts the color timestamp to account for capture card latency)
 - Open the directory the captures are saved to
 - Request spatial mapping data from the spectator view camera (if a SpatialMappingManager exists in your project)
 - Visualize the scene's composite view as well as color, holograms, and alpha channel individually.
 - Turn your camera on.
 - Press play in Unity.
 - When the camera is moved, holograms in Unity should be where they are in the real world relative to your camera color feed.

See also

- [Mixed reality capture](#)
- [Mixed reality capture for developers](#)
- [Shared experiences in mixed reality](#)
- [MR Sharing 240](#)
- [SpectatorView \(Preview\) code on GitHub](#)
- [SpectatorView Pro code on GitHub](#)

How it works - Mixed Reality Capture Studios

11/6/2018 • 2 minutes to read • [Edit Online](#)

Microsoft Mixed Reality Capture Studios enable content creators to create 360-degree holograms from real life subjects that can be used in applications across augmented reality, virtual reality, and 2D screens. To learn more about the potential of using a Mixed Reality Capture Studio to bring life to mixed reality, [visit the official website](#).

To learn more about the technology behind the studio, and how it's brought to life on a range of devices, watch the video below and read the "High-Quality Streamable Free-Viewpoint Video" technical whitepaper, originally presented at Siggraph 2015.

Whitepaper:



[Download "High-Quality Streamable Free-Viewpoint Video" whitepaper](#)

Microsoft Holographic Remoting Software License Terms

11/6/2018 • 7 minutes to read • [Edit Online](#)

MICROSOFT SOFTWARE LICENSE TERMS

MICROSOFT HOLOGRAPHIC REMOTING

These license terms are an agreement between Microsoft Corporation (or based on where you live, one of its affiliates) and you. Please read them. They apply to the software named above, which includes the media on which you received it, if any. The terms also apply to any Microsoft

- * updates,
- * supplements,
- * Internet-based services, and
- * support services

for this software, unless other terms accompany those items. If so, those terms apply.

BY USING THE SOFTWARE, YOU ACCEPT THESE TERMS. IF YOU DO NOT ACCEPT THEM, DO NOT USE THE SOFTWARE.

If you comply with these license terms, you have the rights below.

1. INSTALLATION AND USE RIGHTS.

- a. Installation and Use. You may install and use any number of copies of the software on your devices.
- b. Third Party Programs. The software may include third party programs that Microsoft, not the third party, licenses to you under this agreement. Notices, if any, for the third party program are included for your information only.

2. ADDITIONAL LICENSING REQUIREMENTS AND/OR USE RIGHTS.

- a. Distributable Code. The software contains code that you are permitted to distribute in programs you develop if you comply with the terms below.
 - i. Right to Use and Distribute. The code and text files listed below are "Distributable Code."
 - * Software. You may copy and distribute the object code form of the software.
 - * Sample Code. You may modify, copy, and distribute the source and object code form of code marked as "sample."
 - * Third Party Distribution. You may permit distributors of your programs to copy and distribute the Distributable Code as part of those programs.
 - ii. Distribution Requirements. For any Distributable Code you distribute, you must
 - * add significant primary functionality to it in your programs;
 - * for any Distributable Code having a filename extension of .lib, distribute only the results of running such Distributable Code through a linker with your program;
 - * distribute Distributable Code included in a setup program only as part of that setup program without modification;
 - * require distributors and external end users to agree to terms that protect it at least as much as this agreement;
 - * display your valid copyright notice on your programs; and
 - * indemnify, defend, and hold harmless Microsoft from any claims, including attorneys' fees, related to the distribution or use of your programs.
 - iii. Distribution Restrictions. You may not

- * alter any copyright, trademark or patent notice in the Distributable Code;
- * use Microsoft's trademarks in your programs' names or in a way that suggests your programs come from or are endorsed by Microsoft;
- * distribute Distributable Code to run on a platform other than the Windows platform;
- * include Distributable Code in malicious, deceptive or unlawful programs; or
- * modify or distribute the source code of any Distributable Code so that any part of it becomes subject to an Excluded License. An Excluded License is one that requires, as a condition of use, modification or distribution, that
 - * the code be disclosed or distributed in source code form; or
 - * others have the right to modify it.

3. SCOPE OF LICENSE. The software is licensed, not sold. Unless applicable law gives you more rights, Microsoft reserves all other rights not expressly granted under this agreement, whether by implication, estoppel or otherwise. You may use the software only as expressly permitted in this agreement. In doing so, you must comply with any technical limitations in the software that only allow you to use it in certain ways. You may not

- * disclose the results of any benchmark tests of the software to any third party without Microsoft's prior written approval;
- * work around any technical limitations in the software;
- * reverse engineer, decompile or disassemble the software, except and only to the extent that applicable law expressly permits, despite this limitation;
- * make more copies of the software than specified in this agreement or allowed by applicable law, despite this limitation;
- * publish the software for others to copy;
- * rent, lease or lend the software;
- * transfer the software or this agreement to any third party; or
- * use the software for commercial software hosting services.

4. BACKUP COPY. You may make one backup copy of the software. You may use it only to reinstall the software.

5. DOCUMENTATION. Any person that has valid access to your computer or internal network may copy and use the documentation for your internal, reference purposes.

6. EXPORT RESTRICTIONS. The software is subject to United States export laws and regulations. You must comply with all domestic and international export laws and regulations that apply to the software. These laws include restrictions on destinations, end users and end use. For additional information, see www.microsoft.com/exporting.

7. SUPPORT SERVICES. Because this software is "as is," we may not provide support services for it.

8. ENTIRE AGREEMENT. This agreement, and the terms for supplements, updates, Internet-based services and support services that you use, are the entire agreement for the software and support services.

9. APPLICABLE LAW.

- a. United States. If you acquired the software in the United States, Washington state law governs the interpretation of this agreement and applies to claims for breach of it, regardless of conflict of laws principles. The laws of the state where you live govern all other claims, including claims under state consumer protection laws, unfair competition laws, and in tort.
- b. Outside the United States. If you acquired the software in any other country, the laws of that country apply.

10. LEGAL EFFECT. This agreement describes certain legal rights. You may have other rights under the laws of your country. You may also have rights with respect to the party from whom you acquired the software. This agreement does not change your rights under the laws of your country if the laws of your country do not permit it to do so.

11. DISCLAIMER OF WARRANTY. THE SOFTWARE IS LICENSED "AS-IS." YOU BEAR THE RISK OF USING IT. MICROSOFT GIVES NO EXPRESS WARRANTIES, GUARANTEES OR CONDITIONS. YOU MAY HAVE ADDITIONAL CONSUMER RIGHTS OR STATUTORY GUARANTEES UNDER YOUR LOCAL LAWS WHICH THIS AGREEMENT CANNOT CHANGE. TO THE EXTENT PERMITTED UNDER YOUR LOCAL LAWS, MICROSOFT EXCLUDES THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT.

FOR AUSTRALIA - YOU HAVE STATUTORY GUARANTEES UNDER THE AUSTRALIAN CONSUMER LAW AND NOTHING IN THESE TERMS IS INTENDED TO AFFECT THOSE RIGHTS.

12. LIMITATION ON AND EXCLUSION OF REMEDIES AND DAMAGES. YOU CAN RECOVER FROM MICROSOFT AND ITS SUPPLIERS ONLY DIRECT DAMAGES UP TO U.S. \$5.00. YOU CANNOT RECOVER ANY OTHER DAMAGES, INCLUDING CONSEQUENTIAL, LOST PROFITS, SPECIAL, INDIRECT OR INCIDENTAL DAMAGES.

This limitation applies to

- * anything related to the software, services, content (including code) on third party Internet sites, or third party programs; and
- * claims for breach of contract, breach of warranty, guarantee or condition, strict liability, negligence, or other tort to the extent permitted by applicable law.

It also applies even if Microsoft knew or should have known about the possibility of the damages. The above limitation or exclusion may not apply to you because your country may not allow the exclusion or limitation of incidental, consequential or other damages.

Please note: As this software is distributed in Quebec, Canada, these license terms are provided below in French.

Remarque : Ce logiciel étant distribué au Québec, Canada, les termes de cette licence sont fournis ci-dessous en français.

EXCLUSIONS DE GARANTIE. Le logiciel est concédé sous licence « en l'état ». Vous assumez tous les risques liés à son utilisation. Microsoft n'accorde aucune garantie ou condition expresse. Vous pouvez bénéficier de droits des consommateurs supplémentaires ou de garanties statutaires dans le cadre du droit local, que ce contrat ne peut modifier. Lorsque cela est autorisé par le droit local, Microsoft exclut les garanties implicites de qualité, d'adéquation à un usage particulier et d'absence de contrefaçon.

POUR L'AUSTRALIE - La loi australienne sur la consommation (Australian Consumer Law) vous accorde des garanties statutaires qu'aucun élément du présent accord ne peut affecter.

LIMITATION ET EXCLUSION DE RECOURS ET DE DOMMAGES. Vous pouvez obtenir de Microsoft et de ses fournisseurs une indemnisation en cas de dommages directs limitée uniquement à hauteur de 5,00 \$ US. Vous ne pouvez prétendre à aucune indemnisation pour les autres dommages, y compris les dommages spéciaux, indirects ou accessoires et pertes de bénéfices.

Cette limitation concerne :

- * toute affaire liée au logiciel, aux services ou au contenu (y compris le code) figurant sur des sites Internet tiers ou dans des programmes tiers et
- * les réclamations au titre de violation de contrat ou de garantie, ou au titre de responsabilité stricte, de négligence ou d'une autre faute dans la limite autorisée par la loi en vigueur.

Elle s'applique également même si Microsoft connaissait l'éventualité d'un tel dommage. La limitation ou exclusion ci-dessus peut également ne pas vous être applicable, car votre pays n'autorise pas l'exclusion ou la limitation de responsabilité pour les dommages indirects, accessoires ou de quelque nature que ce soit.

Release notes - October 2018

11/13/2018 • 5 minutes to read • [Edit Online](#)

The [Windows 10 October 2018 Update](#) (also known as RS5) includes new features for both HoloLens and Windows Mixed Reality immersive headsets connected to PCs.

To update to the latest release on HoloLens or PC (for Windows Mixed Reality immersive (VR) headsets), open the **Settings** app, go to **Update & Security**, then select the **Check for updates** button. On a Windows 10 PC, you can also manually install the Windows 10 October 2018 Update using the [Windows media creation tool](#).

Latest release for Desktop: Windows 10 October 2018 Update (**10.0.17763.107**)

Latest release for HoloLens: Windows 10 October 2018 Update (**10.0.17763.134**)

New features for Windows Mixed Reality immersive headsets

The Windows 10 October 2018 Update includes many improvements for using Windows Mixed Reality immersive (VR) headsets with your desktop PC.

For everyone

- **Mixed Reality Flashlight** - Open a portal into the real world to find your keyboard, see someone nearby, or take a look at your surroundings without removing your headset! You can turn on Mixed Reality Flashlight from the Start menu, by pressing Windows + Grab on your motion controller, or by saying "Flashlight on/off." Point your controller in the direction of what you want to see, like using a flashlight in the dark.



- **New apps and ways to launch content in the mixed reality home**

- If you're using [Windows Mixed Reality for SteamVR](#), your SteamVR titles now show up in the Start menu and app launchers for each can be placed in the mixed reality home.



- New 360 Videos app for discovering a regularly-curated selection of 360-degree videos.
- New WebVR Showcase app for discovering a regularly-curated selection of WebVR experiences.
- First-time Windows Mixed Reality customers will enter the Cliff House and find it pre-populated with 3D app launchers for some of our favorite immersive apps and games from the Microsoft Store.
- Microsoft Edge windows now include a *Share* button.
- **Quick actions menu** - From within an immersive mixed reality app, you can press the Windows button to access a new quick actions menu, with easy access to *SteamVR menu*, *photo/video capture*, *flashlight*, and *home*.
- **Support for backpack PCs** - Windows Mixed Reality immersive (VR) headsets run on backpack PCs without requiring a display emulator once setup has been completed.
- **New audio features** - You can now mirror the audio from a Windows Mixed Reality experience to both the audio jack (or headphones) in your headset *and* an audio device connected to your PC (like external speakers). We've also added a visual indicator for volume level in your headset's display.

● **Other improvements**

- Mixed Reality Portal updates are now delivered through the Microsoft Store, enabling quicker updates between major Windows releases. Note that this only applies to the desktop app and the Windows Mixed Reality headset experience still updates with the OS.
- When headsets go to sleep, Windows Mixed Reality apps are suspended instead of terminated (until Mixed Reality Portal is closed).

For developers

- **QR code tracking** - Enable QR code tracking in your mixed reality app, allowing Windows Mixed Reality immersive (VR) headsets to scan for QR codes and report them back to interested apps.
- **Hardware DRM support for immersive apps** - Developers can now request hardware-protected backbuffer textures if supported by the display hardware, allowing applications to use hardware-protected content from sources like PlayReady.
- **Integrate mixed reality capture UI into immersive apps** - Developers can integrate mixed reality capture into their apps using the built-in Windows [camera capture UI](#) with just a few lines of code.

New features for HoloLens

The Windows 10 October 2018 Update is publicly available for all HoloLens customers, and includes a number of improvements, such as:

For everyone

- **Quick actions menu** - From within an immersive mixed reality app, you can press the Windows button to access a new quick actions menu, with easy access to *Start recording video*, *Take pictures*, *Mixed Reality Home*, *Change volume*, and *Connect*.
- **Start/stop video capture from the Start or quick actions menu** - If you start video capture from the Start menu or quick actions menu, you'll be able to stop recording from the same place. (Don't forget, you can always do this with voice commands too.)
- **Project to a Miracast-enabled device** - Project your HoloLens content to a nearby Surface device or TV/monitor if using a Miracast-enabled display or adapter.
- **New notifications** - View and respond to notifications on HoloLens, just like you do on a PC.
- **Useful overlays in immersive mixed reality apps** - You'll now see overlays such as the keyboard, dialogs, file picker, etc. when using immersive mixed reality apps.
- **Visual indicator for volume change** - When you use the volume up/down buttons on your HoloLens you'll see a visual indicator of the volume level in the headset.
- **New visuals for device boot** - A loading indicator was added during the boot process to provide visual feedback that the system is loading.
- **Nearby Sharing** - The Windows Nearby Sharing experience allows you to share a capture with a nearby Windows device.
- **Share from Microsoft Edge** - Microsoft Edge now includes a *Share* button.

For developers

- **Integrate mixed reality capture UI into immersive apps** - Developers can integrate mixed reality capture into their apps using the built-in Windows [camera capture UI](#) with just a few lines of code.

For commercial customers

- **Enable post-setup provisioning** - You can now apply a runtime provisioning package at any time using Settings.
- **Assigned access with Azure AD groups** - You can now use Azure AD groups for configuration of Windows assigned access to set up single or multi-app kiosk configuration.
- **PIN sign-in on profile switch from sign-in screen** - PIN sign-in is now available for "Other User" at the sign-in screen.
- **Sign in with Web Credential Provider using password** - You can now select the Globe icon to launch web sign-in with your password.
- **Read device hardware info through MDM** - IT administrators can see and track HoloLens by device serial number in their MDM console.
- **Set HoloLens device name through MDM (rename)** - IT administrators can see and rename HoloLens devices in their MDM console.

For international customers

You can now use HoloLens with localized user interface for Simplified Chinese or Japanese, including localized Pinyin keyboard, dictation, text-to-speech (TTS), and voice commands.

Provide feedback and report issues

Please use the [Feedback Hub app on your HoloLens or Windows 10 PC](#) to provide feedback and report issues. Using Feedback Hub ensures that all necessary diagnostics information is included to help our engineers quickly debug and resolve the problem.

NOTE

Be sure to accept the prompt that asks whether you'd like Feedback Hub to access your Documents folder (select **Yes** when prompted).

Prior release notes

- [Release notes - April 2018](#)
- [Release notes - October 2017](#)
- [Release notes - August 2016](#)
- [Release notes - May 2016](#)
- [Release notes - March 2016](#)

See also

- [Immersive headset support \(external link\)](#)
- [HoloLens support \(external link\)](#)
- [Install the tools](#)
- [Give us feedback](#)

Release notes - April 2018

11/6/2018 • 18 minutes to read • [Edit Online](#)

The [Windows 10 April 2018 Update](#) (also known as RS4) includes new features for both HoloLens and Windows Mixed Reality immersive headsets connected to PCs.

To update to the latest release for either device type, open the **Settings** app, go to **Update & Security**, then select the **Check for updates** button. On a Windows 10 PC, you can also manually install the Windows 10 April 2018 Update using the [Windows media creation tool](#).

Latest release for Desktop: Windows 10 April 2018 Update ([10.0.17134.1](#))

Latest release for HoloLens: Windows 10 April 2018 Update ([10.0.17134.80](#))

A message from Alex Kipman and overview of new mixed reality features in the Windows 10 April 2018 Update

New features for Windows Mixed Reality immersive headsets

The Windows 10 April 2018 Update includes many improvements for using Windows Mixed Reality immersive (VR) headsets with your desktop PC, such as:

- **New environments for the mixed reality home** - You can now choose between the Cliff House and the new Skyloft environment by selecting **Places** on the Start menu. We've also added [an experimental feature](#) that will let you use custom environments you've created.
- **Quick access to mixed reality capture** - You can now take mixed reality photos using a motion controller. Hold the Windows button and then tap the trigger. This works across environments and apps, but will not capture content protected with DRM.
- **New options for launching and resizing content** - Apps are now automatically placed in front of you when you launch them from the Start menu. You can also now resize 2D apps by dragging the edges and corners of the window.
- **Easily jump to content with "teleport" voice command** - You can now quickly teleport to be in front of content in the Windows Mixed Reality home by gazing at content and saying "teleport."
- **Animated 3D app launchers and decorative 3D objects for the mixed reality home** - You can now add animation to 3D app launchers and allow users to place decorative 3D models from a webpage or 2D app into the Windows Mixed Reality home.
- **Improvements to Windows Mixed Reality for SteamVR** - Windows Mixed Reality for SteamVR is out of "early access" with new upgrades, including: haptic feedback when using motion controllers, improved performance and reliability, and improvements to the appearance of motion controllers in SteamVR.
- **Other improvements** - Automatic performance settings have been updated to provide a more optimized experience (you can [manually override](#) this setting). Setup now provides more detailed information about common compatibility issues with USB 3.0 controllers and graphics cards.

New features for HoloLens

The Windows 10 April 2018 Update has arrived for all HoloLens customers! This update is packed with improvements that have been introduced since the last major release of HoloLens software in [August 2016](#).

For everyone

FEATURE	DETAILS	INSTRUCTIONS
Auto-placement of 2D and 3D content on launch	<p>A 2D app launcher or 2D UWP app auto-places in the world at an optimal size and distance when launched instead of requiring the user to place it. If an immersive app uses a 2D app launcher instead of a 3D app launcher, the immersive app will auto-launch from the 2D app launcher same as in RS1.</p> <p>A 3D app launcher from the Start menu also auto-places in the world. Instead of auto-launching the app, users can then click on the launcher to launch the immersive app. 3D content opened from the Holograms app and from Edge also auto-places in the world.</p>	<p>When opening an app from the Start menu, you will no longer be asked to place it in the world.</p> <p>If the 2D app/3D app launcher placement is not optimal, you can easily move them using new fluid app manipulations described below. You can also re-position the 2D app launcher/3D content by saying "Move this" and then using gaze to re-position the content.</p>
Fluid app manipulation	<p>Move, resize, and rotate 2D and 3D content without having to enter "Adjust" mode.</p>	<p>To move a 2D UWP app or 2D app launcher, simply gaze at its app bar and then use the tap + hold + drag gesture. You can move 3D content by gazing anywhere on the object and then using tap + hold + drag.</p> <p>To resize 2D content, gaze at its corner. The gaze cursor will turn into a resize cursor, and then you can tap + hold + drag to resize. You can also make 2D content taller or wider by looking at its edges and dragging.</p> <p>To resize 3D content, lift up both your hands into gesture frame, fingers up in the ready position. You'll see the cursor turn into a state with 2 little hands. Do the tap and hold gesture with both your hands. Moving your hands closer or farther apart you will change the size of the object. Moving your hands forward and backward relative to each other will rotate the object. You can also resize/rotate 2D content this way.</p>
2D app horizontal resize with reflow	<p>Make a 2D UWP app wider in aspect ratio to see more app content. For example, making the Mail app wide enough to show the Preview Pane.</p>	<p>Simply gaze at the left or right edge of the 2D UWP app to see the resize cursor, then use the tap + hold + drag gesture to resize.</p>
Expanded voice command support	<p>You can do more simply using your voice.</p>	<p>Try these voice commands:</p> <ul style="list-style-type: none"> ● "Go to Start" - Brings up the Start menu or exits an immersive app. ● "Move this" - Allows you to move an object.

Updated Holograms and Photos apps	Updated Holograms app with new holograms. Updated Photos app.	You will notice an updated look to the Holograms and Photos apps. The Holograms app includes several new Holograms and a label maker for easier creation of text.
Improved mixed reality capture	Hardware shortcut start and end MRC video.	Hold Volume Up + Down for 3 seconds to start recording MRC video. Tap both again or use the bloom gesture to end.
Consolidated spaces	Simplify space management for holograms into a single space.	HoloLens finds your space automatically, and no longer requires you to manage or select spaces. If you have problems with holograms around you, you can go to Settings > System > Holograms > Remove nearby holograms . If needed, you can also select Remove all holograms .
Improved audio immersion	You can now hear HoloLens better in noisy environments, and experience more lifelike sound from applications as their sound will be obscured by real walls detected by the device.	You don't have to do anything to enjoy the improved spatial sound.
File Explorer	Move and delete files from within HoloLens.	<p>You can use the File Explorer app to move and delete files from within HoloLens.</p> <p>Tip: If you don't see any files, the "Recent" filter may be active (clock icon is highlighted in left pane). To fix, select the This Device document icon in the left pane (beneath the clock icon), or open the menu and select This Device.</p>
MTP (Media Transfer Protocol) support	Enables your desktop PC to access your libraries (photos, videos, documents) on HoloLens for easy transfer.	<p>Similar to other mobile devices, connect your HoloLens to your PC to bring up File Explorer to access your HoloLens libraries (photos, videos, documents) for easy transfer.</p> <p>Tips:</p> <ul style="list-style-type: none"> If you don't see any files, please ensure you sign in to your HoloLens to enable access to your data. From File Explorer on your PC, you can select Device properties to see Windows Holographic OS version number (firmware version) and device serial number. <p>Known issue: Renaming HoloLens via File Explorer on your PC is not enabled.</p>

Captive portal network support during setup	You can now set up your HoloLens on a guest network at hotels, conference centers, retail shops, or businesses that use captive portal.	During setup, select the network, check connect automatically if desired, and enter the network information as prompted.
Photo and video sync through OneDrive app	Your photos and videos from HoloLens will now sync via the OneDrive app from the Microsoft Store instead of directly through the Photos app.	To set this up, download and launch the OneDrive app from the Store. On first run you should be prompted to automatically upload your photos to OneDrive, or you can find the option in the app settings.

For developers

FEATURE	DETAILS	INSTRUCTIONS
Spatial mapping improvements	Quality, simplification, and performance improvements.	Spatial mapping mesh will appear cleaner – fewer triangles are required to represent the same level of detail. You may notice changes in triangle density in the scene.
Automatic selection of focus point based on depth buffer	Submitting a depth buffer to Windows allows HoloLens to select a focus point automatically to optimize hologram stability.	In Unity, go to Edit > Project Settings > Player > Universal Windows Platform tab > XR Settings , expand the Windows Mixed Reality SDK item, and ensure Enable Depth Buffer Sharing is checked. This will be automatically checked for new projects. For DirectX apps, ensure you call the CommitDirect3D11DepthBuffer method on HolographicRenderingParameters each frame to supply the depth buffer to Windows.
Holographic reprojection modes	You can now disable positional reprojection on HoloLens to improve the hologram stability of rigidly body-locked content such as 360-degree video.	In Unity, set HolographicSettings.ReprojectionMode to HolographicReprojectionMode.OrientationOnly when all content in view is rigidly body-locked. For DirectX apps, set HolographicCameraRenderingParameters.ReprojectionMode to HolographicReprojectionMode.OrientationOnly when all content in view is rigidly body-locked.

App tailoring APIs	<p>Windows APIs know more about where your app is running, such as whether the device's display is transparent (HoloLens) or opaque (immersive headset) and whether a UWP app's 2D view is showing up in the holographic shell.</p>	<p>Unity had previously manually exposed HolographicSettings.IsDisplayOpaque in a way that worked even before this build.</p> <p>For DirectX apps, you can now access existing APIs like HolographicDisplay.GetDefault().IsOpaque and HolographicApplicationPreview.IsCurrentViewPresentedOnHolographicDisplay on HoloLens as well.</p>
Research mode	<p>Allows developers to access key HoloLens sensors when building academic and industrial applications to test new ideas in the fields of computer vision and robotics, including:</p> <ul style="list-style-type: none"> • The four environment tracking cameras • Two versions of the depth mapping camera data • Two versions of an IR-reflectivity stream 	<p>Research mode documentation Research mode sample apps</p>

For commercial customers

FEATURE	DETAILS	INSTRUCTIONS
Use multiple Azure Active Directory user accounts on a single device	Share a HoloLens with multiple Azure AD users, each with their own user settings and user data on device.	IT Pro Center: Share HoloLens with multiple people
Change Wi-Fi network on sign-in	Change Wi-Fi network before sign-in to enable another user to sign in with his or her Azure AD user account for the first time, allowing users to share devices at various locations and job sites.	On the sign-in screen, you can use the network icon below the password field to connect to a network. This is helpful when this is your first time signing into a device.
Unified enrollment	It's now easy for a HoloLens user who set up the device with a personal Microsoft account to add a work account (Azure AD) and join the device to their MDM server.	Simply sign in with an Azure AD account, and enrollment happens automatically.
Mail Sync without MDM enrollment	Support for Exchange Active Sync (EAS) mail sync without requiring MDM enrollment.	You can now sync email without enrolling in MDM. You can set up the device with a Microsoft Account, download and install the Mail app, and add a work email account directly.

For IT pros

FEATURE	DETAILS	INSTRUCTIONS

New "Windows Holographic for Business" OS name	Clear edition naming to reduce confusion on edition upgrade license application when Commercial Suite features are enabled on HoloLens.	You can see which edition of Windows Holographic is on your device in Settings > System > About . "Windows Holographic for Business" will appear if an edition update has been applied to enable Commercial Suite features. Learn how to unlock Windows Holographic for Business features .
Windows Configuration Designer (WCD)	Create and edit provisioning packages to configure HoloLens via updated WCD app. Simple HoloLens wizard for edition update, configurable OOBE, region/time zone, bulk Azure AD token, network, and developer CSP. Advanced editor filtered to HoloLens supported options, including Assigned Access and Account Management CSPs.	IT Pro Center: Configure HoloLens using a provisioning package
Configurable setup (OOBE)	Hide calibration, gesture/gaze training, and Wi-Fi configuration screens during setup.	IT Pro Center: Configure HoloLens using a provisioning package
Bulk Azure AD token support	Pre-register device to Azure AD directory tenant for quicker user setup flow.	IT Pro Center: Configure HoloLens using a provisioning package
DeveloperSetup CSP	Deploy profile to set up HoloLens in Developer mode. Useful for both development and demo devices.	IT Pro Center: Configure HoloLens using a provisioning package
AccountManagement CSP	Share a HoloLens device and remove user data after sign-out or inactivity/storage thresholds for temporary usage. Supports Azure AD accounts.	IT Pro Center: Configure HoloLens using a provisioning package
Assigned access	Windows assigned access for first-line workers or demos. Single or multi-app lockdown. No need to developer unlock.	IT Pro Center: Set up HoloLens in kiosk mode
Guest access for kiosk devices	Windows assigned access with password-less guest account for demos. Single or multi-app lockdown. No need to developer unlock.	IT Pro Center: Set up HoloLens in kiosk mode
Set up (OOBE) diagnostics	Get diagnostic logs from HoloLens so you can troubleshoot Azure AD sign-in failures (before Feedback Hub is available to the user whose sign-in failed).	When setup or sign-in fails, choose the new Collect info option to get diagnostic logs for troubleshooting.
Local account indefinite password expiry	Remove disruption of device reset when local account password expires.	When provisioning a local account, you no longer need to change the password every 42 days in Settings , as the account password no longer expires.

MDM sync status and details	Standard Windows functionality to understand MDM sync status and details from within HoloLens.	You can check the MDM sync status for a device in Settings > Accounts > Access Work or School > Info . In the Device sync status section , you can start a sync, see areas managed by MDM, and create and export an advanced diagnostics report.
-----------------------------	--	---

Known issues

We've worked hard to deliver a great Windows Mixed Reality experience, but we're still tracking some known issues. If you find others, please [give us feedback](#).

HoloLens

After update

You may notice the following issues after updating from RS1 to RS4 on your HoloLens:

- **Pins reset** - The 3x3 apps pinned to your Start menu will be reset to the defaults after update.
- **Apps and placed holograms reset** - Apps placed in your world will be removed after the update and will need to be re-placed throughout your space.
- **Feedback Hub may not launch immediately** - Immediately after update, it will take a few minutes before you are able to launch some inbox apps such as Feedback Hub, while they update themselves.
- **Corporate Wi-Fi certificates need to be re-synced** - We're investigating an issue that requires the HoloLens to be connected to a different network in order for corporate certificates to be re-synced to the device before it is able to reconnect to corporate networks using certificates.
- **H.265 HEVC Video Playback does not work** - Applications that attempt to play back H.265 videos will receive an error message. The workaround is to [access the Windows Device Portal](#), select **Apps** on the left navigation bar, and **remove** the HEVC application. Then, install the latest [HEVC Video Extension](#) from the Microsoft Store. We are investigating the issue.

For developers: updating HoloLens apps for devices running Windows 10 April 2018 Update

Along with a great list of [new features](#), the Windows 10 April 2018 Update (RS4) for HoloLens enforces some code behaviors that previous versions did not:

- **Permission requests to use sensitive resources (camera, microphone, etc.)** - RS4 on HoloLens will enforce permission requests for apps intending to access sensitive resources, such as the camera or microphone. RS1 on HoloLens did not force these prompts, so, if your app assumes immediate access to these resources, it may crash in RS4 (even if the user grants permission to the requested resource). Please read the relevant [UWP app capability declarations article](#) for more information.
- **Calls to apps outside your own** - RS4 on HoloLens will enforce proper use of the [Windows.System.Launcher class](#) to launch another app from your own. For example, we've seen issues with apps calling [Windows.System.Launcher.LaunchUriForResultsAsync](#) from a non-ASTA (UI) thread. This would succeed in RS1 on HoloLens, but RS4 requires the call to be executed on the UI thread.

Windows Mixed Reality on Desktop

Visual quality

- If you notice after the Windows 10 April 2018 Update that graphics are more blurry than before, or that the field of view looks smaller on your headset, the automatic performance setting may have been changed in order to maintain a sufficient framerate on a less powerful graphics card (such as Nvidia 1050). You can manually override this (if you choose) by navigating to **Settings > Mixed reality > Headset display > Experience options > Change** and changing "Automatic" to "90 Hz." You can also try changing **Visual quality** (on the same Settings page) to "High."

Windows Mixed Reality setup

- When setting up Windows with a headset connected, your PC monitor may go blank. Unplug your headset to enable output to your PC monitor to complete Windows setup.
- If you do not have headphones connected, you may miss additional tips when you first visit the Windows Mixed Reality home.
- Other Bluetooth devices can cause interference with motion controllers. If the motion controllers have connection/pairing/tracking issues, make sure the Bluetooth radio (if an external dongle) is plugged in to an unobstructed location and not immediately next to another Bluetooth dongle. Also try powering down other Bluetooth peripherals during Windows Mixed Reality sessions to see if it helps.

Games and apps from the Microsoft Store

- Some graphically intensive games, like Forza Motorsport 7, may cause performance issues on less capable PCs when played inside Windows Mixed Reality.

Audio

- If you have Cortana enabled on your host PC prior to using your Windows Mixed Reality headset, you may lose spatial sound simulation applied to the apps you place around the Windows Mixed Reality home.
 - The work around is to enable "Windows Sonic for Headphones" on all the audio devices attached to your PC, even your headset-connected audio device:
 1. Left-click the speaker icon on the desktop taskbar and select from list of audio devices.
 2. Right-click the speaker icon on the desktop taskbar and select "Windows Sonic for Headphones" in the "Speaker setup" menu.
 3. Repeat these steps for all of your audio devices (endpoints).
 - Another option is turning off "Let Cortana respond to Hey Cortana" in **Settings > Cortana** on your desktop prior to launching Windows Mixed Reality.
- When another multimedia USB device (such as a web cam) shares the same USB hub (either external or inside your PC) with the Windows Mixed Reality headset, in rare cases the headset's audio jack/headphones may either have a buzzing sound or no audio at all. You can fix this by plugging your headset into a USB port that does not share the same hub as the other device, or disconnect/disable your other USB multimedia device.
- In very rare cases, the host PC's USB hub cannot provide enough power to the Windows Mixed Reality headset and you may notice a burst of noise from the headphones connected to the headset.

Holograms

- If you've placed a large number of holograms in your Windows Mixed Reality home, some may disappear and reappear as you look around. To avoid this, remove some of the holograms in that area of the Windows Mixed Reality home.

Motion controllers

- If input is not being routed to the headset, the motion controller will briefly disappear when being held next to the room boundary. Pressing Win+Y to ensure there's a blue banner across the Desktop monitor will resolve this.
- Occasionally, when you click on a web page in Microsoft Edge, the content will zoom instead of click.

Desktop app in the Windows Mixed Reality home

- Snipping Tool does not work in Desktop app.
- Desktop app does not persist setting on re-launch.
- If you're using Mixed Reality Portal preview on your desktop, when opening the Desktop app in the Windows Mixed Reality home, you may notice the infinite mirror effect.
- Running the Desktop app may cause performance issues on non-Ultra Windows Mixed Reality PCs; it is not recommended.
- Desktop app may auto-launch because an invisible window on Desktop has focus.
- Desktop User Account Control prompt will make headset display black until the prompt is completed.

Windows Mixed Reality for SteamVR

- You may need to launch Mixed Reality Portal after updating to ensure the necessary software updates for the Windows 10 April 2018 Update have completed before launching SteamVR.
- You must be on a recent version of Windows Mixed Reality for SteamVR to remain compatible with the Windows 10 April 2018 Update. Make sure automatic updates are turned on for Windows Mixed Reality for SteamVR, which is located in the "Software" section of your library in Steam.

Other issues

IMPORTANT

An early version of the Windows 10 April 2018 Update pushed to Insiders (version 17134.5) was missing a piece of software necessary to run Windows Mixed Reality. We recommend avoiding this version if using Windows Mixed Reality.

We've identified a performance regression when using Surface Book 2 on the initial release of this update (10.0.17134.1) that we are working to fix in an upcoming update patch. We suggest waiting until this has been fixed before updating manually or waiting for the update to roll out normally.

Provide feedback and report issues

Please use the [Feedback Hub app on your HoloLens or Windows 10 PC](#) to provide feedback and report issues. Using Feedback Hub ensures that all necessary diagnostics information is included to help our engineers quickly debug and resolve the problem.

NOTE

Be sure to accept the prompt that asks whether you'd like Feedback Hub to access your Documents folder (select **Yes** when prompted).

Prior release notes

- [Release notes - October 2017](#)
- [Release notes - August 2016](#)
- [Release notes - May 2016](#)
- [Release notes - March 2016](#)

See also

- [Immersive headset support \(external link\)](#)
- [HoloLens support \(external link\)](#)
- [Install the tools](#)
- [Give us feedback](#)

Release notes - October 2017

11/6/2018 • 7 minutes to read • [Edit Online](#)

Welcome to Windows Mixed Reality! The release of the [Windows 10 Fall Creators Update](#) introduces support for new [Windows Mixed Reality immersive headsets](#) and [motion controllers](#), enabling you to explore new worlds, play VR games, and experience immersive entertainment when connected to a [Windows Mixed Reality capable PC](#).

The release of Windows Mixed Reality headsets and motion controllers is the culmination of a massive team effort and a major step forward for the [Windows Mixed Reality platform](#), which includes [Microsoft HoloLens](#). While HoloLens isn't receiving an update with the release of the Windows 10 Fall Creators Update, know that work on HoloLens hasn't stopped; we'll have a lot of learnings and insight to apply from our recent work across Windows Mixed Reality as a whole. In fact, Windows Mixed Reality immersive headsets and motion controllers represent a great entry point to development for HoloLens too, as the same APIs, tools, and concepts apply to both.

To update to the latest release for each device, open the **Settings** app, go to **Update & Security**, then select the **Check for updates** button. On a Windows 10 PC, you can also manually install the Windows 10 Fall Creators Update using the [Windows media creation tool](#).

Latest release for Desktop: Windows 10 Desktop October 2017 ([10.0.16299.15](#), Windows 10 Fall Creators Update)

Latest release for HoloLens: [Windows 10 Holographic August 2016](#) ([10.0.14393.0](#), Windows 10 Anniversary Update)

Introducing Windows Mixed Reality

The Windows 10 Fall Creators Update officially introduces support for Windows Mixed Reality headsets and motion controllers, as well as making Windows 10 the world's first spatial operating system. Here are the highlights:

- **Variety of headsets** - Windows Mixed Reality is enabling partners to offer a variety of headsets which start at \$399 USD bundled with motion controllers.
- **Motion controllers** - Windows Mixed Reality motion controllers wirelessly pair with your PC via Bluetooth and feature six degrees-of-freedom tracking, plenty of input methods, and IMUs.
- **Easy setup and portability** - Set up and get started in under 10 minutes. Immersive headsets use inside-out tracking to track your movement, and your motion controllers, with six degrees-of-freedom. No external cameras or lighthouse markers required!
- **Support for a wider range of PCs** - Windows Mixed Reality will allow more people to experience desktop VR than ever before, with support for select integrated graphics cards and PCs starting at \$499 USD.
- **Windows Mixed Reality home** - The world's first spatial operating system provides a familiar home environment for multi-tasking with 2D apps, launching VR games and apps, and placing decorative holograms.
- **Amazing VR games and apps in the Microsoft Store** - From immersive entertainment like Hulu VR and 360 video to epic games like SUPERHOT VR and Arizona Sunshine, the Microsoft Store has a range of content to experience in Windows Mixed Reality.
- **SteamVR Early Access** - The Windows 10 Fall Creators Update enables support for SteamVR titles to be played with Windows Mixed Reality headsets and controllers, making the largest catalog of VR titles available to Windows Mixed Reality users.

Known issues

We've worked hard to deliver a great Windows Mixed Reality experience, but we're still tracking some known issues. If you find others, please [Give us feedback](#).

Desktop app in the Windows Mixed Reality home

- Snipping Tool does not work in Desktop app.
- Desktop app does not persist setting on re-launch.
- If you're using Mixed Reality Portal preview on your desktop, when opening the Desktop app in the Windows Mixed Reality home, you may notice the infinite mirror effect.
- Running the Desktop app may cause performance issues on non-Ultra Windows Mixed Reality PCs; it is not recommended.
- Desktop app may auto-launch because an invisible window on Desktop has focus.
- Desktop User Account Control prompt will make headset display black until the prompt is completed.

Windows Mixed Reality setup

- When setting up Windows with a headset connected, your PC monitor may go blank. Unplug your headset to enable output to your PC monitor to complete Windows setup.
- When creating a boundary, tracing may fail. If so, try again, as the system will learn more about your space over time.
- If you turn Cortana on or off during Windows Mixed Reality setup, this change will be applied to your desktop Cortana settings.
- If you do not have headphones connected, you may miss additional tips when you first visit the Windows Mixed Reality home.
- Bluetooth headphones can cause interference with motion controllers. We recommend unpairing or powering down Bluetooth controllers during Windows Mixed Reality sessions.

Games and apps from Windows Store

- Some graphically intensive games, like Forza Motorsports 6, may cause performance issues on less capable PCs when played inside Windows Mixed Reality.

Audio

- As noted above, Bluetooth Audio peripherals do not work well with Windows Mixed Reality voice and spatial sound experiences. They can also negatively affect your motion controller experience. WE do not recommend using Bluetooth Audio headsets with Windows Mixed Reality.
- You can't use the audio device connected to (or part of) the headset for audio playback when the device is not being worn. If you only have one audio headset, you may want to connect the audio headset to the host PC instead of the headset. If so, then you must turn off "switch to headset audio" in **Settings > Mixed Reality > Audio and speech**.
- Some applications, including many of those launched through SteamVR, can lose audio or hang when the audio device changes as you start or stop the Mixed Reality Portal. Restart the app after you have opened the Mixed Reality Portal app to correct this.
- If you have Cortana enabled on your host PC prior to using your Windows Mixed Reality headset, you may lose the spatial sound simulation applied to the apps you place around the Windows Mixed Reality home. The work around is to enable "Windows Sonic for Headphones" on all the audio devices attached to your PC, even your headset-connected audio device:
 1. Left-click the speaker icon on the desktop taskbar and select from list of audio devices.
 2. Right-click the speaker icon on the desktop taskbar and select "Windows Sonic for Headphones" in the "Speaker setup" menu.
 3. Repeat these steps for all of your audio devices (endpoints).

NOTE

- Because the headphones/speakers connected to your headset won't appear unless you're wearing it, you have to do this from within the Desktop app window in the Windows Mixed Reality home to apply this setting to the audio device connected to your headset (or integrated into your headset).
- Another option is to turn off "Let Cortana respond to Hey Cortana" in **Settings > Cortana** on your desktop prior to launching Windows Mixed Reality.

- When another multimedia USB device (such as a web cam) shares the same USB hub (either external or inside your PC) with the Windows Mixed Reality headset, in rare cases the headset's audio jack/headphones may either have a buzzing sound or no audio at all. You can fix this by plugging your headset into a USB port that does not share the same hub as the other device, or disconnect/disable your other USB multimedia device.
- In very rare cases, the host PC's USB hub cannot provide enough power to the Windows Mixed Reality headset and you may notice a burst of noise from the headphones connected to the headset.

Speech

- Cortana may fail to play her audio cues for listening/thinking and audio responses to commands.
- Cortana in China and Japan markets do not correctly show text below the Cortana circle during use.
- Cortana can be slow the first time she is invoked in a Mixed Reality Portal session. You can work around this by making sure "Let Cortana respond to Hey Cortana" under **Settings > Cortana > Talk to Cortana** is enabled.
- Cortana may run slower on PCs that are not Windows Mixed Reality Ultra PCs.
- When your system keyboard is set to a language different from the UI language in Windows Mixed Reality, using dictation from the keyboard in Windows Mixed Reality will result in an error dialog about dictation not working due to not having Wi-Fi connection. To fix the issue simply make sure the system keyboard language matches the Windows Mixed Reality UI language.
- Spain is not correctly being recognized as a market where speech is enabled for Windows Mixed Reality.

Holograms

- If you've placed a large number of holograms in your Windows Mixed Reality home, some may disappear and reappear as you look around. To avoid this, remove some of the holograms in that area of the Windows Mixed Reality home.

Motion controllers

- Occasionally, if you click on a webpage in Edge, the content will zoom instead of click.
- Sometimes when you click on a link in Edge, the selection won't work.

Prior release notes

- [Release notes - August 2016](#)
- [Release notes - May 2016](#)
- [Release notes - March 2016](#)

See also

- [Immersive headset support \(external link\)](#)
- [HoloLens known issues](#)
- [Install the tools](#)
- [Give us feedback](#)

Release notes - August 2016

11/6/2018 • 3 minutes to read • [Edit Online](#)

The HoloLens team is listening to feedback from developers in the Windows Insider Program to prioritize our work. Please continue to [give us feedback](#) through the Feedback Hub, the [developer forums](#) and [Twitter via @HoloLens](#). As Windows 10 embraces the Anniversary Update, the HoloLens team is happy to deliver further improvements to the holographic experience. In this update, we focused on major fixes, improvements, and introducing features requested by businesses and available in the Microsoft HoloLens Commercial Suite.

Latest release: Windows Holographic August 2016 Update (**10.0.14393.0**, Windows 10 Anniversary Release)

To [update to the current release](#), open the *Settings* app, go to *Update & Security*, then select the *Check for updates* button.

New features

Attach To Process Debugging HoloLens now supports attach-to-process debugging. You can use Visual Studio 2015 Update 3 to connect to a running app on a HoloLens and [start debugging it](#). This works without the need to deploy from a Visual Studio project.

Updated HoloLens Emulator We've also released an updated version of the HoloLens Emulator.

Gamepad Support You can now pair and use Bluetooth gamepads with HoloLens! The newly released Xbox Wireless Controller S features Bluetooth capabilities and can be used to play your favorite gamepad-enabled games and apps. A [controller update](#) must be applied before you can connect the Xbox Wireless Controller S with HoloLens. The Xbox Wireless Controller S is supported by [XInput](#) and [Windows.Gaming.Input](#) APIs. Additional models of Bluetooth controllers may be accessed through the [Windows.Gaming.Input](#) API.

Improvements and fixes

We are in sync with the rest of the Windows 10 Anniversary update, so in addition to the Hololens specific fixes, you are also receiving all the goodness from the Windows update to increase platform reliability and performance. Your feedback is highly valued and prioritized for fixes in the release.

We've improved the following experiences:

- log in experiences.
- workplace join.
- power efficiency for device power state transitions.
- stability with Mixed Reality Captures.
- reliability for Bluetooth connectivity
- hologram persistence in multi app scenario.

We've fixed the following issues:

- the Visual Studio profilers and graphics debugger fail to connect.
- photos & documents do not show up in the file explorer in the device portal.
- the App Bar can flash when the cursor is placed above it while in Adjust mode.
- When in Adjust mode, the eye gaze dot cursor will change to the 4-arrow cursor sometime more slowly.
- "Hey Cortana play music" does not launch Groove.

- after the previous update, saying "Go Home" does not display the pins panel correctly.

Introducing Microsoft HoloLens Commercial Suite

The Microsoft HoloLens Commercial Suite is ready for enterprise deployment. We've added several highly requested [commercial features](#) from our early business partners.

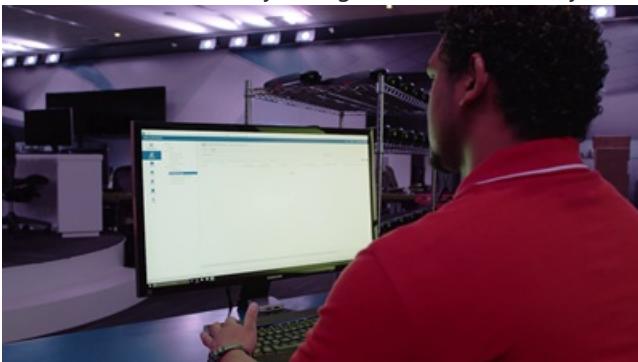
Please contact your local Microsoft account manager to purchase the Microsoft HoloLens Commercial Suite.

Key Commercial Features

- **Kiosk mode.** With HoloLens kiosk mode, you can limit which apps to run to enable demo or showcase experiences.



- **Mobile Device Management (MDM) for HoloLens.** Your IT department can manage multiple HoloLens devices simultaneously using solutions like Microsoft Intune. You will be able to manage settings, select apps to install and set security configurations tailored to your organization's need.



- **Windows Update for Business.** Controlled operating system updates to devices and support for long term servicing branch.
- **Data security.** BitLocker data encryption is enabled on HoloLens to provide the same level of security protection as any other Windows device.
- **Work access.** Anyone in your organization can remotely connect to the corporate network through virtual private network on a HoloLens. HoloLens can also access Wi-Fi networks that require credentials.
- **Microsoft Store for Business.** Your IT department can also set up an enterprise private store, containing only your company's apps for your specific HoloLens usage. Securely distribute your enterprise software to selected group of enterprise users.

Development Edition vs. Commercial Suite

FEATURES	DEVELOPMENT EDITION	COMMERCIAL SUITE
Device Encryption (Bitlocker)		✓ <input type="checkbox"/>
Virtual Private Network (VPN)		✓ <input type="checkbox"/>
Kiosk mode		✓ <input type="checkbox"/>

MANAGEMENT AND DEPLOYMENT		
Mobile Device Management (MDM)		✓□
Ability to block un-enrollment		✓□
Cert Based Corporate Wi-Fi Access		✓□
Microsoft Store (Consumer)	Consumer	Filtering via MDM
Business Store Portal		✓□
SECURITY AND IDENTITY		
Login with Azure Active Directory (AAD)	✓□	✓□
Login with Microsoft Account (MSA)	✓□	✓□
Next Generation Credentials with PIN unlock	✓□	✓□
Secure boot	✓□	✓□
SERVICING AND SUPPORT		
Automatic system updates as they arrive	✓□	✓□
Windows Update for Business		✓□
Long term servicing branch		✓□

Prior release notes

- [Release notes - May 2016](#)
- [Release notes - March 2016](#)

See also

- [HoloLens known issues](#)
- [Commercial features](#)
- [Install the tools](#)
- [Using the HoloLens emulator](#)

Release notes - May 2016

11/6/2018 • 6 minutes to read • [Edit Online](#)

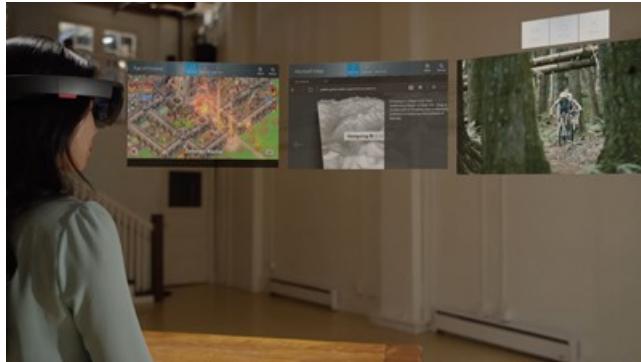
The HoloLens team is committed to provide you with an update on our latest feature development and major fixes through the Windows Insider Program. Thanks to all your suggestions, we take your feedback to heart. Please continue to [give us feedback](#) through the Feedback Hub, the [developer forums](#) and [Twitter via @HoloLens](#).

Release version: Windows Holographic May 2016 Update (**10.0.14342.1016**)

To update to the current release, open the *Settings* app, go to *Update & Security*, then select the *Check for updates* button.

New features

- You can now **run up to three apps in 2D view simultaneously**. This enables endless use cases for multi-tasking in HoloLens. Have the new Feedback Hub with the list of Quests open while exploring the new features on this flight.



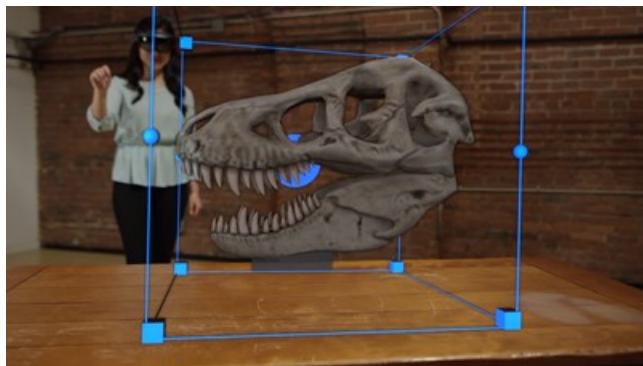
Run up to three apps in 2D view simultaneously

- We've added new **voice commands**:
 - Try looking at a hologram and rotate it by saying "face me"
 - Change its size by saying "bigger" or "smaller"
 - Move an app by saying "Hey Cortana, move *app name* here."
- We've made **developing on HoloLens easier**. You can now browse, upload, and download files through the [Windows Device Portal](#). You can access the Documents folder, Pictures folder, and the local storage for any app you side-loaded or deployed through Visual Studio.
- The **emulator now supports log-in with a Microsoft Account** just like you would on a real HoloLens. You can enable this from the Additional Tools menu ">>".
- **2D Apps now hide the app bar and cursor when watching video full screen** to avoid distraction. With this, your video watching experience will be even more enjoyable on HoloLens.
- You can also **pin photos without the app bar** in your world .



The app bar can be hidden for apps with a 2D view, like Photos

- **File picker** works just like you expect it to on HoloLens.
- Updated **Edge browser** to map unified user experience with desktop and phone. We enabled multiple instances of your browser, custom HoloLens new tab page, tab peek, and open in new windows, along with power & performance improvements.
- **Groove Music** app is now on HoloLens. Visit the store to download it and try playing in the background.
- You can easily customize how apps are arranged in your world. Try **rotating your holograms** in adjust mode by simply click and drag on circle in the middle vertical wireframes. You might notice holograms have **tighter fitted bounding boxes** to ensure maximized interaction. You can also resize all flat apps vertically (except Photos and Hologram apps).



Rotate holograms after you place them in your world

- We've made a lot of **input improvement** in this flight. You can connect a regular Bluetooth mouse to HoloLens. The clicker has been fine tuned to enable resizing & moving holograms with a clicker. Keyboard is also running better than ever.
- Now you can take **mixed reality pictures** by simply pressing down the volume up + volume down simultaneously. You can also share your mixed reality captured photos & videos to Facebook, Twitter and YouTube.
- The maximum recording length of **mixed reality videos** has been increased to five minutes.
- **Photos app** now streams videos from OneDrive instead of having to download the entire video before playback.
- We've improved how your **holograms will be right where you left them**. You will also be presented the option to re-connect to Wi-Fi and try again if HoloLens cannot detect where they are.
- The keyboard has an **improved layout for entering email address** with keys that allow you to enter the most popular email domains with a single click.
- Faster **app registration** and **auto detection of time zone** during OOB, giving you the best first user experience.

- **Storage sense** allows you to view remaining and used disk space by the system and apps in the settings app.
- We have converged Feedback App and Inside Hub into a single app **Feedback Hub** that will be the go-to tool for **giving us feedback**, which features you love, which features you could do without, or when something could be better. When you join the Insider Program, you can keep up on **get latest Insider news, rate builds** and go on **feedback quests** from Feedback Hub.
- We've also [published an updated HoloLens Emulator build](#).
- Your mixed reality videos now look better due to automatic **video stabilization**.

Major fixes

We fixed issues with hologram spaces where the spaces would be slow or incorrectly detected. We fixed a shutdown issue that could continue to try launching the shell during shutdown.

We fixed an issue

- that blocks the ability to resume a XAML application.
- where a crash would leave a black screen and some jagged lines.
- scrolling would sometimes stick in the wrong direction when using some apps.
- the power LEDs could indicate an off state when the device was still on.
- WiFi could get turned off after resuming from standby.
- the Xbox identity provider offers gamertag setup and then gets stuck in a loop.
- the Shell occasionally crashes when selecting a downloaded file in the OneDrive File Picker.
- pressing and holding on a link sometimes both opens a new, broken tab and opens a context menu.
- where windows device portal didn't allow IPD adjustments from 50 to 80

We fixed issues with Photos where

- an image would occasionally display rotated due to ignoring the EXIF orientation property.
- it could crash during start-up on pinned Photos.
- videos would restart after pausing instead of continuing from where last paused.
- replay of a shared video could be prevented if it was shared while playing.
- Mixed Reality Capture recordings would begin with 0.5-1 second of audio only feed.
- the Sync button disappears during initial OneDrive sync.

We fixed issues with settings where

- a refresh was needed when the environment changes.
- 'Enter' on keyboard would not behave like clicking Next in some dialogs.
- it was hard to know when the clicker failed pairing.
- it could become unresponsive with WiFi disconnect and connect.

We fixed issues with Cortana where

- it could get stuck displaying the Listening UI.
- asking "Hey Cortana, what can I say" from an exclusive mode app would get stuck if you answered maybe rather yes/no to the request to exit the app.
- the Cortana listening UI doesn't resume correctly if you ask Cortana to go to sleep and then resume.
- the queries "What network am I connected to?" and the "Am I connected?" could fail when the first network profile comes back with no connectivity.
- the UI froze on "Listening" but upon exiting an app would immediately began doing speech recognition again.

- where signing out of the Cortana app wouldn't let you sign back into it until a reboot.
- it would not launch when Mixed Reality Capture UI was active.

We fixed issues with Visual Studio where

- background task debugging did not work.
- frame analysis in the graphics debugger did not work.
- the HoloLens Emulator did not appear in the drop-down list for Visual Studio unless your project's TargetPlatformVersion was set to 10240.

Changes from previous release

- The Cortana command **Hey Cortana, reboot the device** does not work. User can say **Hey Cortana, restart** or **Hey Cortana, restart the device**.
- Kiosk mode has been removed from the product.

Prior release notes

- [Release notes - March 2016](#)

See also

- [HoloLens known issues](#)
- [Install the tools](#)
- [Shell](#)
- [Updating 2D UWP apps for mixed reality](#)
- [Hardware accessories](#)
- [Mixed reality capture](#)
- [Voice input](#)
- [Submitting an app to the Windows Store](#)
- [Using the HoloLens emulator](#)

Release notes - March 2016

11/6/2018 • 2 minutes to read • [Edit Online](#)

Welcome to Windows Holographic, available for the first time on Microsoft HoloLens. We want to [hear your feedback](#). Use the Feedback Hub, the [developer forums](#) and [Twitter](#) via [@HoloLens](#).

Release version: Windows Holographic March 2016 Release (**10.0.11082.1033**)

What's in Windows Holographic

Try the inbox apps

- **Microsoft Edge.** Modern web browser for Windows 10.
- **Settings.** Check system information, [connect to Wi-Fi](#), and [connect to Bluetooth devices](#).
- **Holograms.** Place holograms in your world, walk around them, then see and hear them from any angle.
- **Calibration.** Repeat the [calibration](#) done during the out-of-box-experience. Do this whenever switching users.
- **Learn Gestures.** Repeat the gesture tutorial done during the out-of-box experience.
- **Photos.** See the mixed reality capture photos and videos you take with the device, then place them in world along with holograms.

To unlock the device for development, open the *Settings* app, go to *Update & Security*, switch to the *For developers* page, and set the developer mode toggle to on. This is the same place where you can enable the [Device Portal](#).

See also

- [HoloLens known issues](#)
- [Install the tools](#)
- [Navigating the Windows Mixed Reality home](#)
- [Hardware accessories](#)
- [Mixed reality capture](#)
- [Using the HoloLens emulator](#)

Accounts on HoloLens

11/6/2018 • 3 minutes to read • [Edit Online](#)

During initial HoloLens setup, users are required to sign in with the account they want to use on the device. This account can be either a consumer Microsoft account or an enterprise account that has been configured in Azure Active Directory (AAD) or Active Directory Federation Services (ADFS).

Signing into this account during setup creates a user profile on the device which the user can use to sign-in, and against which all apps will store their data. This same account also provides Single Sign On for apps such as Edge or Skype via the Windows Account Manager APIs.

Additionally, when signing into an enterprise or organizational account on the device, it may also apply Mobile Device Management (MDM) policy, if configured by your IT Admin.

Whenever the device restarts or resumes from standby, the credentials for this account are used to sign-in again. If the option enforcing an explicit sign-in is enabled in Settings, the user will be required to type in their credentials again. Anytime the device restarts after receiving and applying an OS update, an explicit sign-in is required.

Multi-user support

NOTE

Multi-user support requires the Commercial Suite, as this is a [Windows Holographic for Business](#) feature.

Starting with the [Windows 10 April 2018 Update](#), HoloLens supports multiple users from within the same AAD tenant. To use this you must set up the device initially with an account that belongs to your organization.

Subsequently, other users from the same tenant will be able to sign into the device from the sign-in screen or by tapping the user tile on the Start panel to sign out the existing user.

Apps installed on the device will be available to all other users, but each will have their own app data and preferences. Removing an app will also remove it for all other users though.

You can remove device users from the device to reclaim space by going to Settings > Accounts > Other people. This will also remove all of the other users' app data from the device.

Linked accounts

Within a single device account, users can link additional web account credentials for the purpose of the easier access within apps (such as the Store) or to combine access to personal and work resources, similar to the Desktop version of Windows. Signing into an additional account in this way does not separate the user data created on the device, such as images or downloads. Once an account has been connected to a device, apps can make use of it with your permission to reduce having to sign into each app individually.

Using single sign-on within an app

As an app developer, you can take advantage of having a connected identity on HoloLens with the [Windows Account Manager APIs](#), just as you would on other Windows devices. Some code samples for these APIs are available [here](#).

Any account interrupts that may occur such as requesting user consent for account information, two-factor authentication etc. must be handled when the app requests an authentication token.

If your app requires a specific account type that hasn't been linked previously, your app can ask the system to prompt the user to add one. This will trigger the account settings pane to be launched as a modal child of your app. For 2D apps, this window will render directly over the center of your app and for Unity apps, this will briefly take the user out of your holographic app so that this child window can be rendered. Customizing the commands and actions on this pane is described [here](#).

Enterprise and other authentication

If your app makes use of other types of authentication, such as NTLM, Basic, or Kerberos, you can use [Windows Credential UI](#) to collect, process, and store the user's credentials. The user experience for collecting these credentials is very similar to other cloud driven account interrupts and will appear as a child app on top of your 2D app or briefly suspend a Unity app to show the UI.

Deprecated APIs

One difference for developing on HoloLens from Desktop is that [OnlinelDAuthenticator](#) API is not fully supported. Although it will return a token if the primary account is in good-standing, interrupts such as those described above will not display any UI for the user, and will fail to correctly authenticate the account.

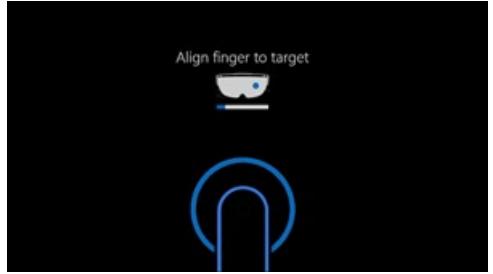
Calibration

11/6/2018 • 2 minutes to read • [Edit Online](#)

Calibrating your IPD (interpupillary distance) can improve the quality of your visuals. Both HoloLens and Windows Mixed Reality immersive headsets offer ways to customize IPD.

HoloLens

During setup



On HoloLens, you'll be prompted to calibrate your visuals during setup. This allows the device to adjust hologram display according to the user's [interpupillary distance](#) (IPD). With an incorrect IPD, holograms may appear unstable or at an incorrect distance.

After Cortana introduces herself, the first setup step is calibration. It's recommended that you complete the calibration step during this setup phase, but it can be skipped by waiting until Cortana prompts you to say "Skip" to move on.

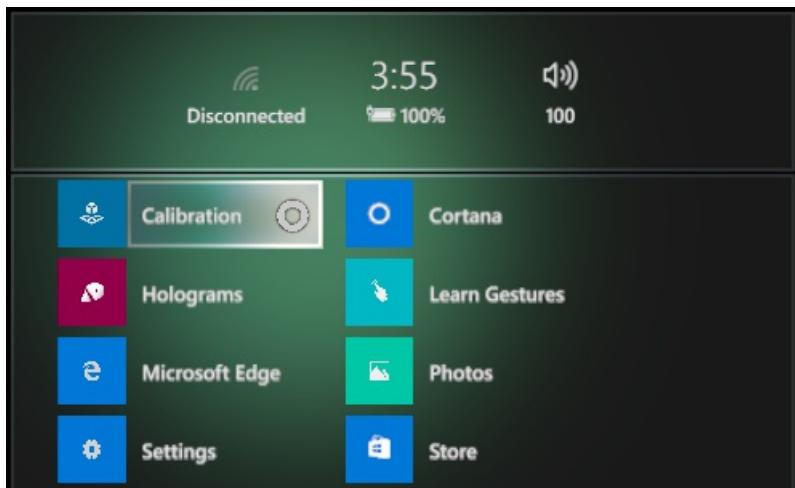
Users are asked to align their finger with a series of six targets per eye. Through this process, HoloLens sets the correct IPD for the user. If the calibration needs to be updated or adjusted for a new user, it can be run outside of setup using the Calibration app.

Calibration app

Calibration can be performed any time through the Calibration app. The Calibration app is installed by default and may be accessed from the Start menu, or through the Settings app. Calibration is recommended if you'd like to improve the quality of your visuals or calibrate visuals for a new user.

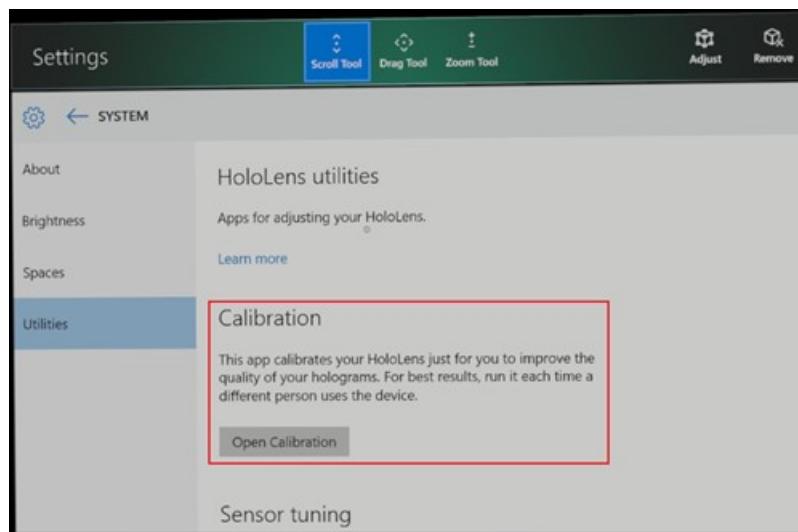
Launching the app from Start

1. Use [bloom](#) to get to [Start Menu](#).
2. Select + to view all apps.
3. Launch **Calibration**.



Launching the app from Settings

1. Use [bloom](#) to get to [Start Menu](#).
2. Select + to view all apps if **Settings** isn't pinned to Start.
3. Launch **Settings**.
4. Navigate to **System > Utilities** and select **Open Calibration**.



Immersive headsets

To change IPD within your headset, open the Settings app and navigate to **Mixed reality > Headset display** and move the slider control. You'll see the changes in real time in your headset. If you know your IPD, maybe from a visit to the optometrist, you can enter it directly as well.

You can also adjust this setting by going to **Settings > Mixed reality > Headset display** on your PC.

If your headset does not support IPD customization, this setting will be disabled.

See also

- Environment considerations for HoloLens

Commercial features

11/6/2018 • 2 minutes to read • [Edit Online](#)

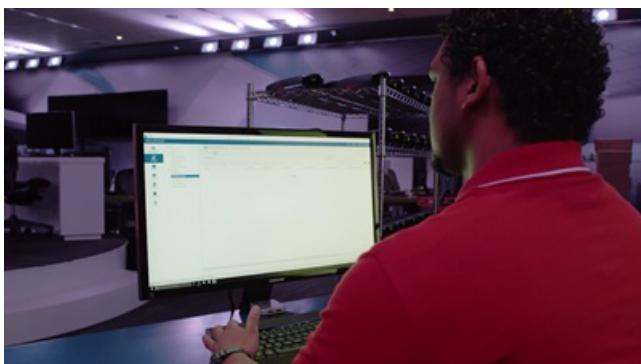
The Microsoft HoloLens Commercial Suite includes features that make it easier for businesses to manage HoloLens devices. Commercial features are included in the Windows operating system, but they are enabled by a license. In nearly all cases, the license is enabled by Microsoft Device Management when HoloLens enrolls in an organization. Contact your local Microsoft account manager to purchase the Microsoft HoloLens Commercial Suite.

Key commercial features

- **Kiosk mode.** With HoloLens kiosk mode, you can limit which apps to run to enable demo or showcase experiences.



- **Mobile Device Management (MDM) for HoloLens.** Your IT department can manage multiple HoloLens devices simultaneously using solutions like Microsoft Intune. You will be able to manage settings, select apps to install and set security configurations tailored to your organization's need.



- **Windows Update for Business.** Controlled operating system updates to devices and support for long term servicing branch.
- **Data security.** BitLocker data encryption is enabled on HoloLens to provide the same level of security protection as any other Windows device.
- **Work access.** Anyone in your organization can remotely connect to the corporate network through virtual private network on a HoloLens. HoloLens can also access Wi-Fi networks that require credentials.
- **Microsoft Store for Business.** Your IT department can also set up an enterprise private store, containing

only your company's apps for your specific HoloLens usage. Securely distribute your enterprise software to selected group of enterprise users.

Development Edition vs. Commercial Suite

FEATURES	DEVELOPMENT EDITION	COMMERCIAL SUITE
Device Encryption (Bitlocker)		✓□
Virtual Private Network (VPN)		✓□
Kiosk mode		✓□
MANAGEMENT AND DEPLOYMENT		
Mobile Device Management (MDM)		✓□
Ability to block un-enrollment		✓□
Cert Based Corporate Wi-Fi Access		✓□
Microsoft Store (Consumer)	Consumer	Filtering via MDM
Business Store Portal		✓□
SECURITY AND IDENTITY		
Login with Azure Active Directory (AAD)	✓□	✓□
Login with Microsoft Account (MSA)	✓□	✓□
Next Generation Credentials with PIN unlock	✓□	✓□
Secure boot	✓□	✓□
SERVICING AND SUPPORT		
Automatic system updates as they arrive	✓□	✓□
Windows Update for Business		✓□
Long term servicing branch		✓□

Enabling commercial features

Commercial features like Microsoft Store for Business, kiosk mode, and enterprise Wi-Fi access are setup by an organization's IT admin. The [Windows IT Center for HoloLens](#) provides step by step instructions for device enrollment and installing apps from Microsoft Store for Business.

See also

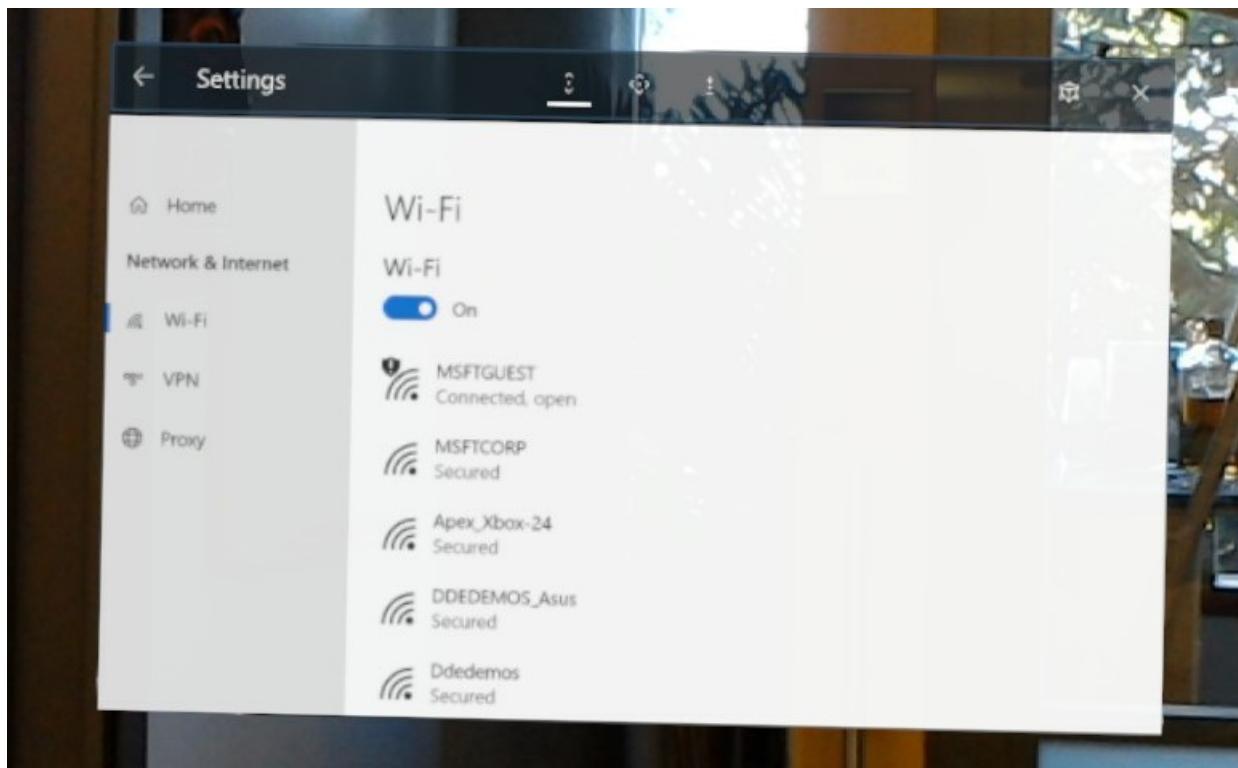
- [IT Pro Guide for HoloLens](#)

- Kiosk mode
- Supported CSPs in Windows Holographic for enterprise
- Microsoft Store For Business and line of business applications
- Working with line-of-business apps

Connecting to Wi-Fi on HoloLens

11/6/2018 • 2 minutes to read • [Edit Online](#)

HoloLens contains a 802.11ac-capable, 2x2 Wi-Fi radio. Connecting HoloLens to a Wi-Fi network is similar to connecting a Windows 10 Desktop or Mobile device to a Wi-Fi network.



Connecting to a Wi-Fi network on HoloLens

1. [Bloom](#) to the **Start** menu.
2. Select the **Settings** app from Start or from the **All Apps** list on the right of the Start menu.
3. The **Settings** app will be auto-placed in front of you.
4. Select **Network & Internet**.
5. Make sure Wi-Fi is turned on.
6. Select a Wi-Fi network from the list.
7. Type in the Wi-Fi network password (if needed).

Disabling Wi-Fi on HoloLens

Using the Settings app on HoloLens

1. [Bloom](#) to the **Start** menu.
2. Select the **Settings** app from Start or from the **All Apps** list on the right of the Start menu.
3. The **Settings** app will be auto-placed in front of you.
4. Select **Network & Internet**.
5. Select the Wi-Fi slider switch to move it to the "Off" position. This will turn off the RF components of the Wi-Fi radio and disable all Wi-Fi functionality on HoloLens.

WARNING

HoloLens will not be able to automatically load your **spaces** when the Wi-Fi radio is disabled.

6. Move the slider switch to the "On" position to turn on the Wi-Fi radio and restore Wi-Fi functionality on Microsoft HoloLens. The selected Wi-Fi radio state ("On" or "Off") will persist across reboots.

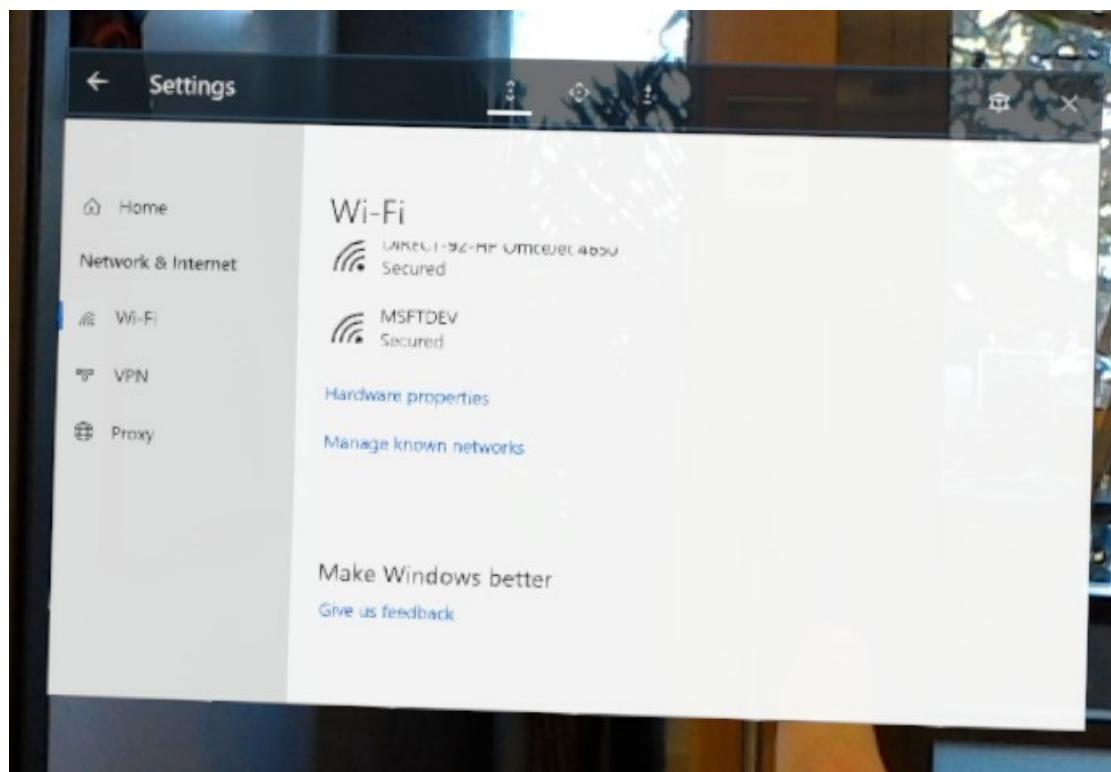
How to confirm you are connected to a Wi-Fi network

1. **Bloom** to bring up the **Start** menu.
2. Look at the top left of the Start menu for Wi-Fi status. The state of Wi-Fi and the SSID of the connected network will be shown.

Identifying the IP Address of your HoloLens on the Wi-Fi network

Using the **Settings** app

1. **Bloom** to the **Start** menu.
2. Select the **Settings** app from Start or from the **All Apps** list on the right of the Start menu.
3. The **Settings** app will be auto-placed in front of you.
4. Select **Network & Internet**.
5. Scroll down to beneath the list of available Wi-Fi networks and select **Hardware properties**.



The IP address will be shown next to **IPv4 address**.

Using Cortana

Say "*Hey Cortana, What's my IP address?*" and Cortana will display and read out your IP address.

Using Windows Device Portal

1. Open the **device portal** in a web browser on your PC.
2. Navigate to the **Networking** section.

Your IP address and other network information will be displayed there. This method allows for easy copy and paste of the IP address on your development PC.

Environment considerations for HoloLens

11/6/2018 • 5 minutes to read • [Edit Online](#)

HoloLens blends the holographic with the "real" world, placing holograms in your surroundings. A holographic app window "hangs" on the wall, a holographic ballerina spins on the tabletop, bunny ears sit on top of your unwitting friend's head. When you're using an immersive game or app, the holographic world will spread to fill your surroundings—but you'll still be able to see and move around the space.

The holograms you place will stay where you've put them, even if you turn off your device.

Here are some things to keep in mind for the best experience with the holographic world. Note, HoloLens is designed to be used indoors in a safe space with no tripping hazards. Don't use it while driving or performing other activities that require your full attention.

Spaces

HoloLens learns about your space so it can remember where you've placed holograms. HoloLens can remember multiple spaces, and is designed to load the correct space when you turn it on.

Setting up

HoloLens can map and remember your spaces best if you choose the right environment.

- Use a room with adequate light and plenty of space. Avoid dark spaces and rooms with a lot of dark, shiny, or translucent surfaces (such as mirrors or thin curtains).
- If you want HoloLens to learn a space, face it directly, from about one meter away.
- Avoid spaces with a lot of moving objects. If an item has moved and you want HoloLens to learn its new position (for example, you moved your coffee table to a new spot), gaze and move around it.

Spatial mapping

When you enter a new space (or load an existing one), you'll see a mesh graphic spreading over the space. This means your device is [mapping your surroundings](#). If you're having trouble placing holograms, try walking around the space so HoloLens can map it more fully. If your HoloLens can't map your space or is out of calibration, you may enter Limited mode. In Limited mode, you won't be able to place holograms in your surroundings.

Managing your spaces

HoloLens remembers the spaces you've used it in by associating them with what Wi-Fi networks are available in that location. A space will get its name from the Wi-Fi network that HoloLens is connected to in that location, but being connected to a Wi-Fi network or not, does not affect the way HoloLens remembers the space you are in.

When you are in a new location and HoloLens does not recognize the Wi-Fi networks around you, it will create a new space, which will have a default name since you are not connected yet, so once you connect to the Wi-Fi network, the space name will change to be the same as the Wi-Fi network. Note that this is not the case where you first set up your HoloLens and connect to Wi-Fi, by the time you are done with the initial set up, a space will already be created with the name of the Wi-Fi network you connected to.

When you go to a location that HoloLens recognizes, the space for that location will be automatically loaded and any holograms that were already placed in that space will appear as well.

You can see and manage your spaces by going to Settings > System > Spaces. There, you can remove a space you no longer need or load the space you want. There are important points to have in mind when using HoloLens and managing spaces:

- HoloLens assigns a numerical value to each space that represents how confident it is that the space corresponds to a given physical location based on the Wi-Fi networks that are available and their signal strengths, this process is called Wi-Fi fingerprinting, and is constantly running as the user walks around the space, so the more the user covers, the higher the confidence value will be, and the higher the chances are that the correct space will be loaded the next time HoloLens is turned on.
- When HoloLens is turned on in a location where there are no Wi-Fi networks available, their signal strength are just too low, or Wi-Fi is disabled in Settings > Network & Internet, HoloLens will be in Limited Mode and will remain there, a new space will not be created, but you can load an existing space from Settings > System > Spaces. The user can get out of this state by turning Wi-Fi on if it was disabled or moving to an area where there are Wi-Fi networks with good signal, and then restart.
- When HoloLens can't tell which space you are in because there are two or more spaces that have similar Wi-Fi fingerprinting (the confidence value for those spaces is similar), it will show a dialog so you can select which space you want to load (top 2 choices, ranked by confidence in decreasing order) and an option to go to Settings to select another space.

Hologram quality

Holograms can be placed throughout your environment—high, low, and all around you—but you'll see them through a [holographic frame](#) that sits in front of your eyes. To get the best view, make sure to adjust your device so you can see the entire frame. And don't hesitate to walk around your environment and explore!

For your [holograms](#) to look crisp, clear, and stable, your HoloLens needs to be calibrated just for you. When you first set up your HoloLens, you'll be guided through this process. Later on, if holograms don't look right or you're seeing a lot of errors, you can make adjustments.

Calibration

If your holograms look jittery or shaky, or if you're having trouble placing holograms, the first thing to try is the [Calibration app](#). This app can also help if you're experiencing any discomfort while using your HoloLens.

To get to the Calibration app, go to Settings > System > Utilities. Select Open Calibration and follow the instructions.

If you run the Calibration app and are still having problems with hologram quality, or you're seeing a frequent "tracking lost" message, try the Sensor Tuning app. Go to Settings > System > Utilities, select Open Sensor Tuning, and follow the instructions.

If someone else is going to be using your HoloLens, they should run the Calibration app first so the device is set up properly for them.

See also

- [Spatial mapping design](#)
- [Holograms](#)
- [Calibration](#)

Get apps for HoloLens

11/6/2018 • 2 minutes to read • [Edit Online](#)

As a Windows 10 device, HoloLens supports many existing UWP applications from the app store, as well as new apps built specifically for HoloLens. On top of these, you may even want to [develop](#) and install your own apps or those of your friends!

Installing Apps

There are three ways to install new apps on your HoloLens. The primary method will be to install new applications from the Windows Store. However, you can also install your own applications using either the Device Portal or by deploying them from Visual Studio.

From the Microsoft Store

1. Perform a [bloom](#) gesture to open the [Start Menu](#).
2. Select the Store app and then tap to place this tile into your world.
3. Once the Store app opens, use the search bar to look for any desired application.
4. Select **Get** or **Install** on the application's page (a purchase may be required).

Installing an application package with the Device Portal

1. Establish a connection from [Device Portal](#) to the target HoloLens.
2. Navigate to the **Apps** page in the left navigation.
3. Under **App Package** Browse to the .appx file associated with your application.

IMPORTANT

Make sure to reference any associated dependency and certificate files.

4. Click **Go**.

+ Install app

App package

Dependency

Deploy

Using Windows Device Portal to install an app on HoloLens

Deploying from Microsoft Visual Studio 2015

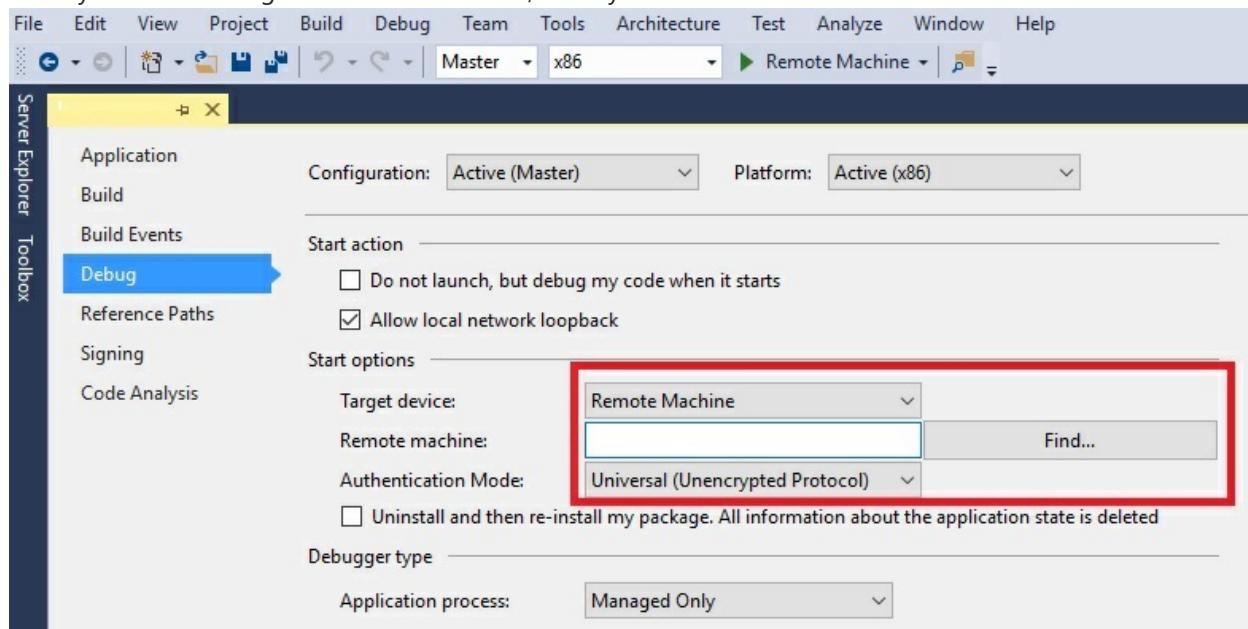
1. Open your app's Visual Studio solution (.sln file).
2. Open the project's **Properties**.
3. Select the following build configuration: Master/x86/Remote Machine.

4. When you select Remote Machine:

- Make sure the address points to the HoloLens' WiFi IP address.
- Set authentication to Universal (Unencrypted Protocol).

5. Build your solution.

6. Click the **Remote Machine** button to deploy the app from your development PC to your HoloLens. If you already have an existing build on the HoloLens, select yes to re-install this newer version.



7. The application will install and auto launch on your HoloLens.

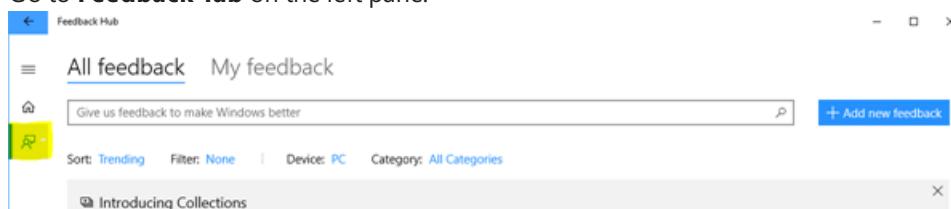
Give us feedback

11/6/2018 • 2 minutes to read • [Edit Online](#)

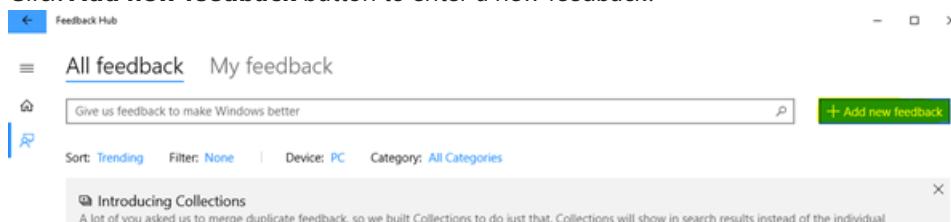
Use the Feedback Hub to tell us which features you love, which features you could do without, or when something could be better.

Feedback for Windows Mixed Reality immersive headset on PC

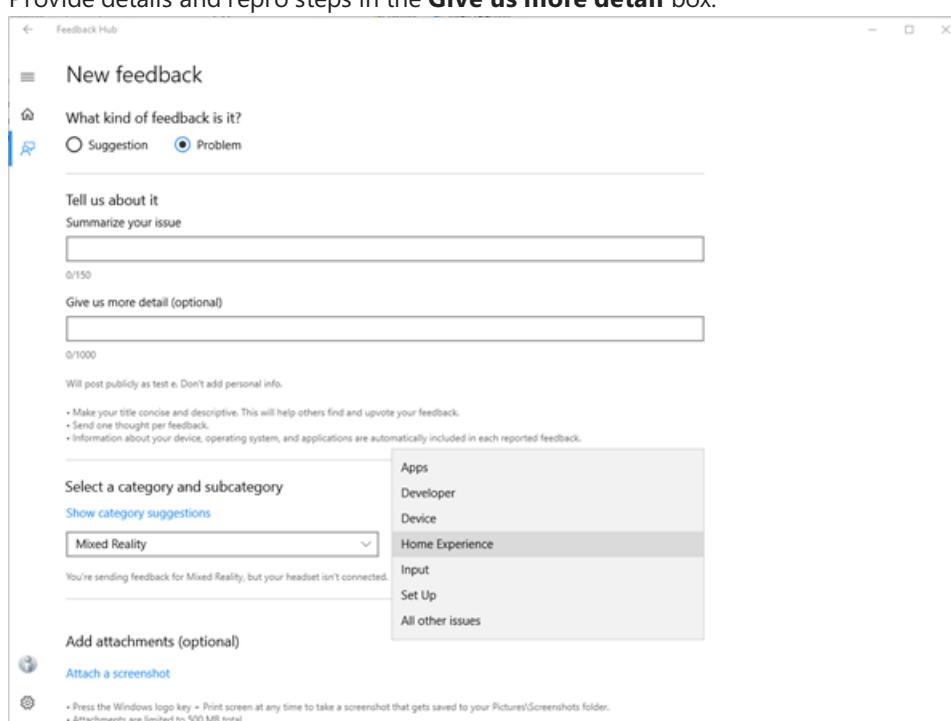
- Ensure you have the immersive headset connected to your PC.
- Launch **Feedback Hub** on desktop with the HMD connected.
- Go to **Feedback Tab** on the left pane.



- Click **Add new feedback** button to enter a new feedback.



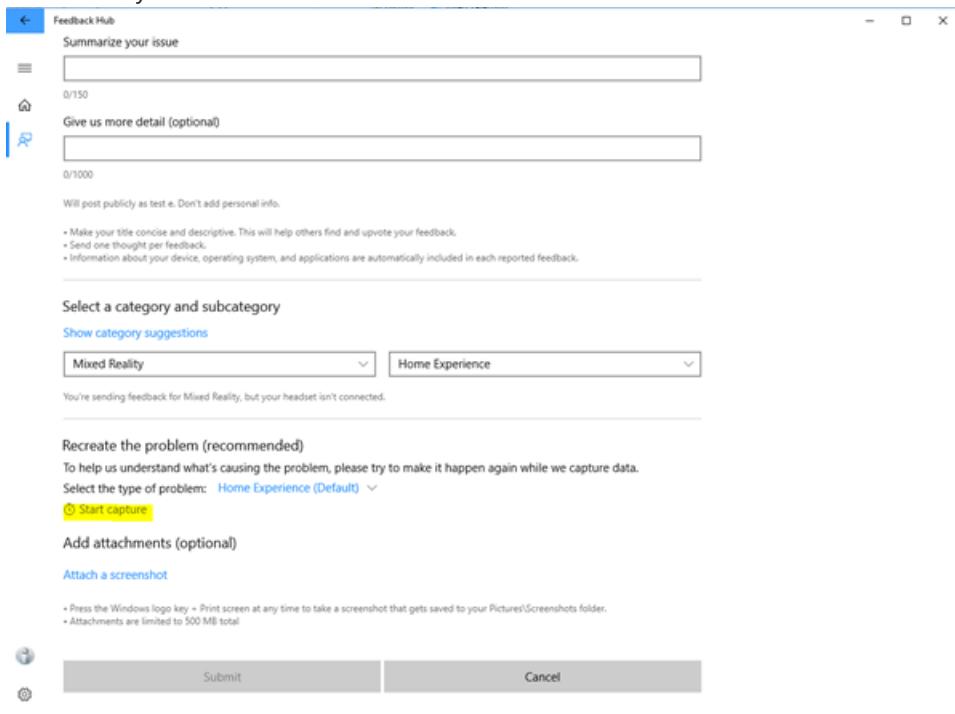
- Select **Problem** in **What kind of feedback is this?** to make feedback actionable.
- Provide meaningful feedback title in **Summarize your issue** box.
- Provide details and repro steps in the **Give us more detail** box.



- Select **Mixed Reality** top category and then pick an applicable sub category:

SUBCATEGORY	DESCRIPTION
Apps	Issues with a specific application.
Developer	Issues in authoring / running an app for Mixed Reality.
Device	Issues with the HMD itself.
Home experience	Issues with your VR environment: interactions with the your mixed reality home.
Input	Issues with input methods: motion controllers, speech, gamepad, or mouse and keyboard.
Set up	Anything that is preventing you from setting up the device.
All other issues	Anything else.

- To help us identify and fix the bug faster, capturing traces and video is extremely helpful. To start collecting traces, click on **Start capture**. This will begin collecting traces and a video capture of your mixed reality scenario.



- Leave the Feedback app and run through the broken scenario. Do not close the Feedback Hub app at this point.
- After you are done with your scenario, go back to the feedback app and click **Stop Capture**. Once you do that, you should see that a file containing the traces has been added.
- Click **Submit**.

Will post publicly as test e. Don't add personal info.

- Make your title concise and descriptive. This will help others find and upvote your feedback.
- Send one thought per feedback.
- Information about your device, operating system, and applications are automatically included in each reported feedback.

Select a category and subcategory

Show category suggestions

Mixed Reality Home Experience

You're sending feedback for Mixed Reality, but your headset isn't connected.

Recreate the problem (recommended)

To help us understand what's causing the problem, please try to make it happen again while we capture data.

Select the type of problem: Home Experience (Default)

Start capture
 Capture file [Remove](#)

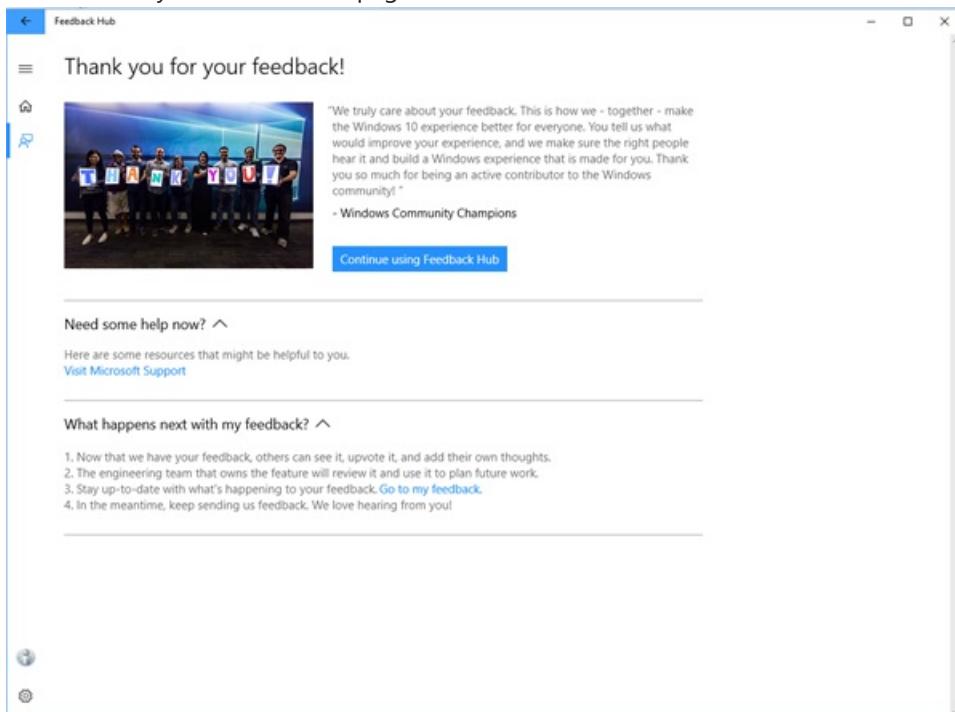
Add attachments (optional)

Attach a screenshot

• Press the Windows logo key + Print screen at any time to take a screenshot that gets saved to your Pictures\Screenshots folder.
 • Attachments are limited to 500 MB total

[Submit](#) [Cancel](#)

- This will lead you to Thank You page.



- At this point, your feedback has been successfully submitted.
- After you submit feedback, to easily direct other people (e.g. co-workers, Microsoft staff, [forum](#) readers etc) to the issue go to Feedback > My Feedback, click on the issue, and use the **Share** icon to get a shortened URL you can give to others to upvote, or escalate.

NOTE

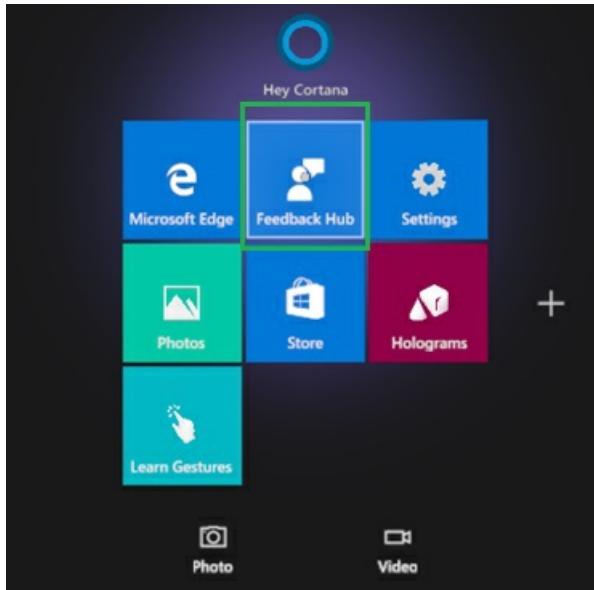
Before filing a bug, please ensure you meet the following constraints so that the logs are successfully uploaded with the feedback:

- Have a minimum of 3GB free disk space available on the main drive of the device.
- Ensure that a non-metered network is available in order to upload cabs.

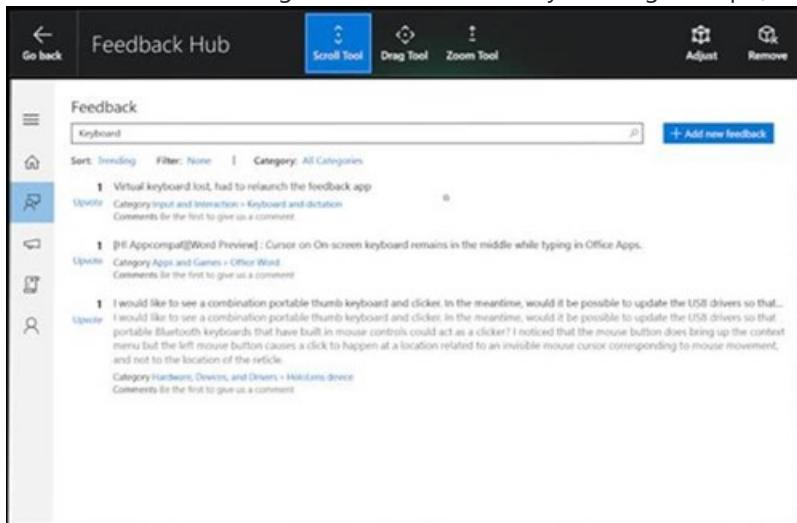
Feedback for HoloLens

1. Use the **bloom** gesture to bring up the Start menu.

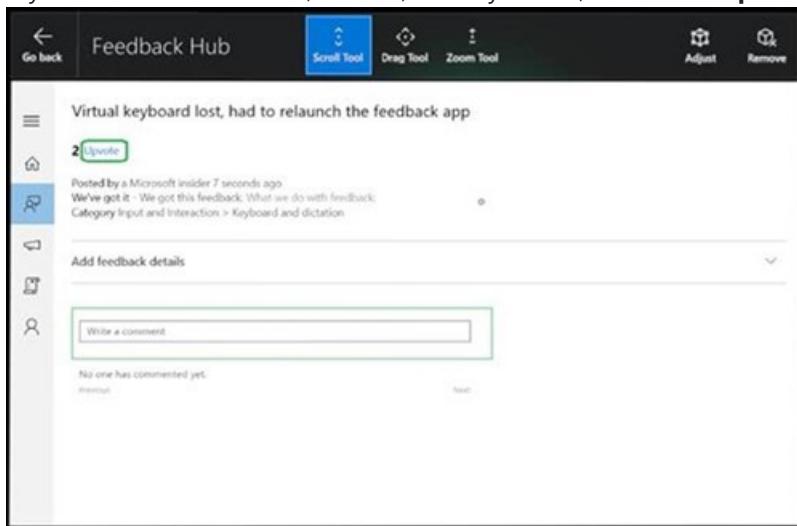
2. Select **Feedback Hub** from Start.



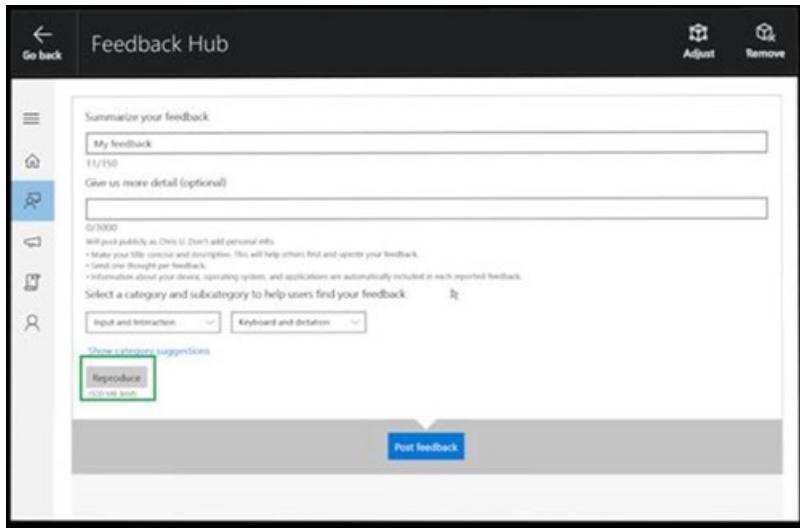
3. Place the app in your environment and then select the app to launch it.
4. See if someone else has given similar feedback by entering the topic, such as "Keyboard".



5. If you find similar feedback, select it, add any details, then select **Upvote**.



6. If you don't find any similar feedback, select **Add new feedback**, and choose a topic from **Select a category** and then **Select a subcategory**.



7. Enter your feedback.
8. If you have a repro, you can click **Reproduce** and repro your feedback, once you are done, come back to Feedback Hub and click **I'm done**. This will collect Mixed Reality Capture of your repro and relevant diagnostic logs will be collected.
9. You're done.

HoloLens known issues

11/6/2018 • 2 minutes to read • [Edit Online](#)

This is the current list of known issues for HoloLens affecting developers. Check here first if you are seeing an odd behavior. This list will be kept updated as new issues are discovered or reported, or as issues are addressed in future HoloLens software updates.

Connecting to WiFi

During OOBE & Settings, there is a credential timeout of 2 mins. The username/password needs to be entered within 2 mins otherwise the username field will be automatically cleared.

We recommend using a Bluetooth keyboard for entering long passwords.

Device Update

- 30 seconds after a new update, the shell may disappear one time. Please perform the **bloom** gesture to resume your session.

Visual Studio

- See [Install the tools](#) for the most up-to-date version of Visual Studio recommended for HoloLens development.
- When deploying an app from Visual Studio to your HoloLens, you may see the error: **The requested operation cannot be performed on a file with a user-mapped section open. (Exception from HRESULT: 0x800704C8)**. If this happens, try again and your deployment will generally succeed.

Emulator

- Not all apps in the Microsoft Store are compatible with the emulator. For example, Young Conker and Fragments are not playable on the emulator.
- You cannot use the PC webcam in the Emulator.
- The Live Preview feature of the Windows Device Portal does not work with the emulator. You can still capture Mixed Reality videos and images.

Unity

- See [Install the tools](#) for the most up-to-date version of Unity recommended for HoloLens development.
- Known issues with the Unity HoloLens Technical Preview are documented in the [HoloLens Unity forums](#).

Windows Device Portal

- The Live Preview feature in Mixed Reality capture may exhibit several seconds of latency.
- On the Virtual Input page, the Gesture and Scroll controls under the Virtual Gestures section are not functional. Using them will have no effect. The virtual keyboard on the same page works correctly.
- After enabling Developer Mode in Settings, it may take a few seconds before the switch to turn on the Device Portal is enabled.

API

- If the application sets the [focus point](#) behind the user or the normal to camera.forward, holograms will not appear in Mixed Reality Capture photos or videos. Until this bug is fixed in Windows, if applications actively set the [focus point](#) they should ensure the plane normal is set opposite camera-forward (e.g. normal = -camera.forward).

Xbox Wireless Controller

- Xbox Wireless Controller S must be updated before it can be used with HoloLens. Ensure you are [up to date](#) before attempting to pair your controller with a HoloLens.
- If you reboot your HoloLens while the Xbox Wireless Controller is connected, the controller will not automatically reconnect to HoloLens. The Guide button light will flash slowly until the controller powers off after 3 minutes. To reconnect your controller immediately, power off the controller by holding the Guide button until the light turns off. When you power your controller on again, it will reconnect to HoloLens.
- If your HoloLens enters standby while the Xbox Wireless Controller is connected, any input on the controller will wake the HoloLens. You can prevent this by powering off your controller when you are done using it.

HoloLens Troubleshooting

11/6/2018 • 3 minutes to read • [Edit Online](#)

My HoloLens is unresponsive or won't boot

If your HoloLens won't boot:

- If the LEDs by the power button don't light up, or only 1 LED briefly blinks, you may need to charge your HoloLens.
- If the LEDs light up when you press the power button but you can't see anything on the displays, hold the power button until all 5 of the LEDs by the power button turn off.

If your HoloLens becomes frozen or unresponsive:

- Turn off your HoloLens by pressing the power button until all 5 of the LEDs by the power button turn themselves off, or for 10 seconds if the LEDs are unresponsive. Press the power button again to boot.

If these steps don't work:

- You can try [recovering your device](#).

Holograms don't look good or are moving around.

If your holograms are unstable, jumpy, or don't look right, try one of these fixes:

- Clean your device visor and make sure nothing is obstructing the sensors.
- Make sure there's enough light in your room.
- Try walking around and looking at your surroundings so HoloLens can scan them more completely.
- Try running the Calibration app. It calibrates your HoloLens to work best for your eyes. Go to **Settings > System > Utilities**. Under Calibration, select **Open Calibration**.
- If you're still having trouble after running the Calibration app, use the Sensor Tuning app to tune your device sensors. Go to **Settings > System > Utilities**. Under Sensor tuning, select **Open Sensor Tuning**.

HoloLens doesn't respond to my gestures.

To make sure HoloLens can see your gestures, keep your hand in the gesture frame, which extends a couple of feet on either side of you. When HoloLens can see your hand, the cursor will change from a dot to a ring. Learn more about using [gestures](#).

If your environment is too dark, HoloLens might not see your hand, so make sure there's enough light.

If your visor has fingerprints or smudge, please use the microfiber cleaning cloth that came with the HoloLens to clean your visor gently.

HoloLens doesn't respond to my voice commands.

If Cortana isn't responding to your voice commands, make sure Cortana is turned on. On the All apps list, select Cortana > Menu > Notebook > Settings to make changes. To learn more about what you can say, see Use your voice to control HoloLens.

I can't place holograms or see holograms I previously placed.

If HoloLens can't map or load your space, it will enter Limited mode and you won't be able to place holograms or see holograms you've placed. Here are some things to try:

- Make sure there's enough light in your environment so HoloLens can see and map the space.
- Make sure you're connected to a Wi-Fi network. If you're not connected to Wi-Fi, HoloLens can't identify and load a known space.
- If you need to create a new space, connect to Wi-Fi, then restart your HoloLens.
- To see if the correct space is active, or to manually load a space, go to **Settings > System > Spaces**.
- If the correct space is loaded and you're still having problems, the space may be corrupt. To fix this, select the space, then select Remove. Once the space is removed, HoloLens will start mapping your surroundings and create a new space.

My HoloLens frequently enters Limited mode or shows a "Tracking lost" message.

If your device often shows a "limited mode" or "tracking lost" message, try the suggestions from [My Holograms don't look good or are moving around](#).

My HoloLens can't tell what space I'm in.

If your HoloLens can't automatically identify and load the space you're in, make sure you're connected to Wi-Fi, there's plenty of light in the room, and there haven't been any major changes to the surroundings. You can also load a space manually or manage your spaces by going to **Settings > System > Spaces**.

I'm getting a "low disk space" error.

You'll need to free up some storage space by doing one or more of the following:

- Delete some unused spaces. Go to **Settings > System > Spaces**, select a space you no longer need, and then select **Remove**.
- Remove some of the holograms you've placed.
- Delete some pictures and videos in the Photos app.
- Uninstall some apps from your HoloLens. In the All apps list, tap and hold the app you want to uninstall, and then select **Uninstall**.

My HoloLens can't create a new space.

The most likely problem is that you're running low on storage space. Try one of the [tips above](#) to free up some disk space.

Mixed reality capture

11/6/2018 • 3 minutes to read • [Edit Online](#)

HoloLens gives users the experience of mixing the real world with the digital world. Mixed reality capture (MRC) let you capture that experience as either a photograph or a video. This lets you share the experience with others by allowing them to see the holograms as you see them. Such videos and photos are from a first-person point of view. For a third-person point of view, use [spectator view](#).

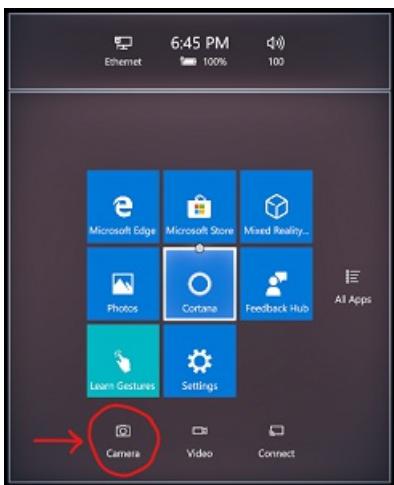
Use cases for mixed reality capture go beyond sharing videos amongst a social circle. Videos can be used to instruct others on how to use an app. Developers can use videos or stills to improve repro steps and debug app experiences.

Live streaming from HoloLens

The [Windows 10 October 2018 Update](#) adds Miracast support to HoloLens. Select the **Connect** button at the bottom of the Start menu to bring up a picker for Miracast-enabled devices and adapters. Select the device to which you want to begin streaming. When done, select the **Disconnect** button at the bottom of the Start menu. **Connect** and **Disconnect** are also available on the quick actions menu.

The [Windows Device Portal](#) exposes live streaming options for devices that are in Developer mode.

Taking mixed reality captures



Click the camera icon at the bottom of the Start menu

There are multiple ways to initiate a mixed reality capture:

- Cortana can be used at all times regardless of the app currently running. Just say, "Hey Cortana, take a picture" or "Hey Cortana, start recording." To stop a video, say "Hey Cortana, stop recording."
- On the Start menu, select either **Camera** or **Video**. Use [air-tap](#) to open the built-in MRC camera UI.
- On the quick actions menu, select either **Camera** or **Video** to open the built-in MRC camera UI.
- Apps are able to expose their own UI for mixed reality capture using custom or, as of the [Windows 10 October 2018 Update](#), built-in MRC camera UI.
- Unique to HoloLens:
 - [Windows Device Portal](#) has a mixed reality capture page that can be used to take photos, videos, live stream, and view captures.
 - Press both the **volume up** and **volume down** buttons simultaneously to take a picture, regardless of the app currently running.

- Hold the **volume up** and **volume down** buttons for three seconds to start recording a video. To stop a video, tap both **volume up** and **volume down** buttons simultaneously.
- Unique to immersive headsets:
 - Using a motion controller, hold the **Windows** button and then tap the **trigger** to take a picture.
 - Using a motion controller, hold the **Windows** button and then tap the **menu** button to start recording video. Hold the **Windows** button and then tap the **trigger** to stop recording video.

NOTE

The [Windows 10 October 2018 Update](#) changes how bloom and Windows button behave. Before the update, the bloom gesture or Windows button would stop recording. After the update, the bloom gesture or the Windows button opens the Start menu (or the quick actions menu if you are in an app). In the menu, select **Stop video** to stop recording.

Limitations of mixed reality capture

On HoloLens, the system will throttle the render rate to 30Hz. This creates some headroom for MRC to run so the app doesn't need to keep a constant budget reserve, and also matches the MRC video record framerate of 30fps.

Videos have a maximum length of five minutes.

The built-in MRC camera UI only supports a single MRC operation at a time (taking a picture is mutually exclusive from recording a video).

File formats

Mixed reality captures from Cortana voice commands and Start Menu tools create files in the following formats:

TYPE	FORMAT	EXTENSION	RESOLUTION	AUDIO
Photo	JPEG	.jpg	1408x792px (HoloLens) 1920x1080px (Immersive headsets)	N/A
Video	MPEG-4	.mp4	1408x792px (HoloLens) 1632x918px (Immersive headsets)	48kHz Stereo

Viewing mixed reality captures

Mixed reality capture photos and videos are saved to the device's "Camera roll" folder. These can be accessed via the [Photos app](#) or File Explorer.

On a PC connected to HoloLens, you can also use [Windows Device Portal](#) or your PC's File Explorer ([via MTP](#)).

If you install the [OneDrive app](#), you can turn on **Camera upload**, and your MRC photos and videos will sync to OneDrive and your other devices using OneDrive.

NOTE

As of the Windows 10 April 2018 Update, the Photos app will no longer upload your photos and videos to OneDrive.

See also

- Spectator view
- Locatable camera
- Mixed reality capture for developers
- See your photos
- Using the Windows Device Portal

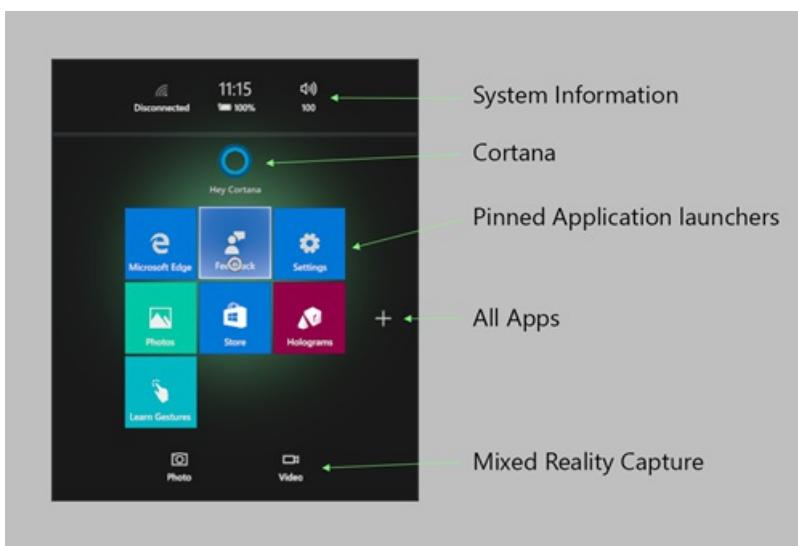
Navigating the Windows Mixed Reality home

11/6/2018 • 6 minutes to read • [Edit Online](#)

Just like the Windows PC experience starts with the desktop, Windows Mixed Reality starts with the home. The Windows Mixed Reality home leverages our innate ability to understand and navigate 3D places. With HoloLens, your home is your physical space. With immersive headsets, your home is a virtual place.

Your home is also where you'll use the Start menu to open and place apps and content. You can fill your home with mixed reality content and multitask by using multiple apps at the same time. The things you place in your home stay there, even if you restart your device.

Start Menu



The Start menu consists of:

- System information (network status, battery percentage, current time, and volume)
- Cortana (on immersive headsets, a Start tile; on HoloLens, at the top of Start)
- Pinned apps
- The All apps button (plus sign)
- Photo and video buttons for [mixed reality capture](#)

Switch between the pinned apps and All apps views by selecting the plus or minus buttons. To open the Start menu on HoloLens, use the bloom gesture. On an immersive headset, press the Windows button on your controller.

Launching apps

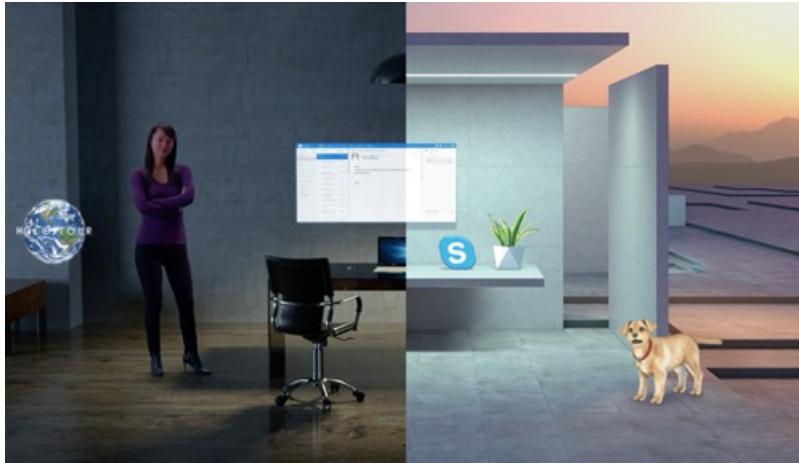
To launch an app, select it on Start. The Start menu will disappear, and the app will open in placement mode, as either a 2D window or a [3D model](#).

To run the app, you'll need to then place it in your home:

1. Use your [gaze](#) or controller to position the app where you want it. It will automatically adjust (in size and position) to conform to the space where you place it.
2. Place the app using air-tap (HoloLens) or the Select button (immersive headsets). To cancel and bring back the Start menu, use the bloom gesture or the Windows button.

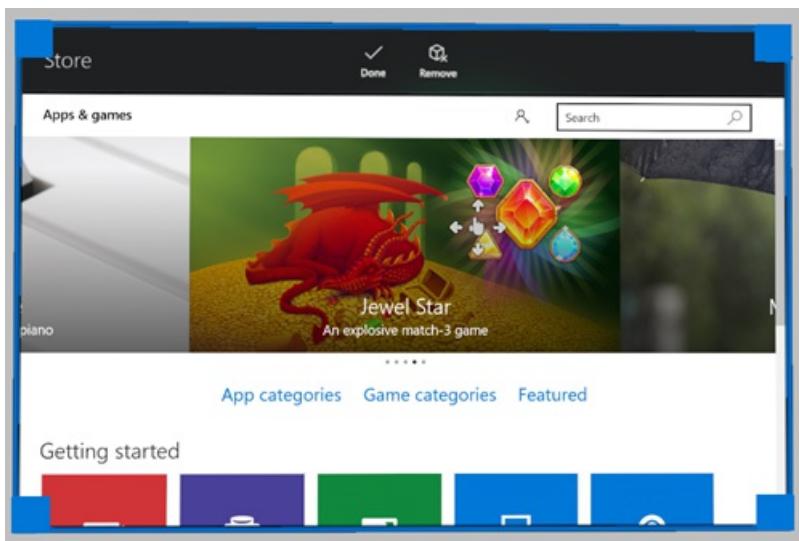
[2D apps](#), created for desktop, mobile, or Xbox can be modified to run as mixed reality immersive apps using the [HolographicSpace API](#). An immersive app takes the user out of the home and into an immersive experience. Users can return home with the bloom gesture (HoloLens) or by pressing the Windows button on their controller (immersive headsets).

Apps can also be launched via an app-to-app API or via Cortana.



Moving and adjusting apps

Select **Adjust** on the app bar to reveal controls that move, scale, and rotate mixed reality content. When you're finished, select **Done**.



Different apps may have additional options on the app bar. For example, Microsoft Edge has *Scroll*, *Drag*, and *Zoom* choices.



The **Back** button navigates back to previously viewed screens in the app. It will stop when you reach the beginning of the experiences that have been shown in the app, and will not navigate to other apps.

Getting around your home

With **HoloLens**, you move through physical space to move around your home.

With **immersive headsets**, you can similarly get up and walk around in your playspace to move within a similar area in the virtual world. To move across longer distances, you can use the thumbstick on your controller to virtually "walk," or you can use *teleportation* to immediately jump longer distances.



To teleport:

1. Bring up the teleportation reticle.
 - Using [motion controllers](#): press the thumbstick forward and hold it in that position.
 - Using an Xbox controller: press the left thumbstick forward and hold it in that position.
 - Using a mouse: hold down the right-click mouse button (and use the scroll wheel to rotate the direction you want to face when you teleport).
2. Place the reticle where you want to teleport.
 - Using [motion controllers](#): tilt the controller (on which you're holding the thumbstick forward) to move the reticle.
 - Using an Xbox controller: use your [gaze](#) to move the reticle.
 - Using a mouse: move your mouse to move the reticle.
3. Release the button to teleport where the reticle was placed.

To virtually "walk:"

- Using [motion controllers](#): click down on the thumbstick and hold, then move the thumbstick in the direction you want to "walk."
- Using an Xbox controller: click down on the left thumbstick and hold, then move the thumbstick in the direction you want to "walk."

Immersive headset input support

[Windows Mixed Reality immersive headsets](#) support multiple input types for navigating the Windows Mixed Reality home. HoloLens does not support accessory inputs for navigation, because you physically walk around and see your environment. However, HoloLens does [support inputs](#) for interacting with apps.

Motion controllers

The best Windows Mixed Reality experience will be with Windows Mixed Reality [motion controllers](#) that support 6 degrees-of-freedom tracking using just the sensors in your headset - no external cameras or markers required!

Navigation commands coming soon.

Gamepad

- **Left thumbstick:**

- Press and hold the left thumbstick forward to bring up the [teleportation](#) reticle.
- Tap the thumbstick left, right, or back to move left, right, or back in small increments.
- Click down on the left thumbstick and hold, then move the thumbstick in the direction you want to [virtually "walk."](#)
- Tap the **right thumbstick** left or right to rotate the direction you're facing by 45 degrees.
- Pressing the **A** button performs a select and acts like the [air tap](#) gesture.
- Pressing the **Guide** button brings up the [Start menu](#) and acts like the [bloom](#) gesture.
- Pressing the **left and right triggers** lets you zoom in and out of a 2D desktop app you're interacting with in the home.

Keyboard and mouse

Note: Use **Windows Key + Y** to switch the mouse between controlling your PC's desktop and the Windows Mixed Reality home.

Within the Windows Mixed Reality home:

- Pressing the **left-click** mouse button performs a select and acts like the [air tap](#) gesture.
- Holding the **right-click** mouse button brings up the [teleportation](#) reticle.
- Pressing the **Windows** key on the keyboard brings up the [Start Menu](#) and acts like the [bloom](#) gesture.
- When [gazing](#) at a 2D desktop app, you can **left-click** to select, **right-click** to bring up context menus, and use the **scroll wheel** to scroll (just like on your PC's desktop).

Cortana

[Cortana](#) is your personal assistant in Windows Mixed Reality, just like on PC and phone. HoloLens has a built-in microphone, but immersive headsets may require additional hardware. Use Cortana to open apps, restart your device, look things up online, and more. Developers may also choose to [integrate Cortana](#) into their experiences.

You can also use voice commands to get around your home. For example, point at a button (using [gaze](#) or a controller, depending on the device) and say "Select." Other voice commands include "Go home," "Bigger," "Smaller," "Close," and "Face me."

Store, Settings, and system apps

Windows Mixed Reality has a number of built-in apps, such as:

- **Microsoft Store** to get apps and games
- **Feedback Hub** to submit feedback about the system and system apps
- **Settings** to configure system settings ([including networking](#) and system updates)
- **Microsoft Edge** to browse websites
- **Photos** to view and share photos and videos
- **Calibration** (HoloLens only) for adjusting the HoloLens experience to the current user
- **Learn Gestures** (HoloLens) or **Learn Mixed Reality** (immersive headsets) to learn about using your device
- **3D Viewer** to decorate your world with mixed reality content
- **Mixed Reality Portal** (desktop) for setting up and managing your immersive headset and streaming a live preview of your view in the headset for others to see.
- **Movies and TV** for viewing 360 videos and the latest movies and tv shows
- **Cortana** for all of your virtual assistant needs
- **Desktop** (immersive headsets) for viewing your desktop monitor while in an immersive headset
- **File Explorer** Access files and folders located on your device

See also

- [App views](#)
- [Motion controllers](#)
- [Hardware accessories](#)
- [Environment considerations for HoloLens](#)
- [Implementing 3D app launchers](#)
- [Creating 3D models for use in the Windows Mixed Reality home](#)

Reset or recover your HoloLens

11/6/2018 • 4 minutes to read • [Edit Online](#)

If you're experiencing problems with your HoloLens you may want to try a restart, reset, or even re-flash with device recovery. This document will guide you through the recommended steps in succession.

Perform a device reboot

If your HoloLens is experiencing issues or is unresponsive, first try rebooting it via one of the below methods.

Perform a safe reboot via Cortana

The safest way to reboot the HoloLens is via Cortana. This is generally a great first-step when experiencing an issue with HoloLens:

1. Put on your device
2. Make sure it's powered on, a user is logged in, and the device is not waiting for a password to unlock it.
3. Say "Hey Cortana, reboot" or "Hey Cortana, restart."
4. When she acknowledges she will ask you for confirmation. Wait a second for a sound to play after she has finished her question, indicating she is listening to you and then say "Yes."
5. The device will now reboot/restart.

Perform a safe reboot via Windows Device Portal

If the above doesn't work, you can try to reboot the device via [Windows Device Portal](#). In the upper right corner, there is an option to restart/shutdown the device.

Perform a safe reboot via the power button

If you still can't reboot your device, you can try to issue a reboot via the power button:

1. Press and hold the power button for 5 seconds
 - a. After 1 second, you will see all 5 LEDs illuminate, then slowly turn off from right to left
 - b. After 5 seconds, all LEDs will be off, indicating the shutdown command was issued successfully
 - c. Note, it's important to stop pressing the button immediately after all the LEDs have turned off
2. Wait 1 minute for the shutdown to cleanly succeed. Note that the shutdown may still be in progress even if the displays are off
3. Power on the device again by pressing and holding the power button for 1 second

Perform an unsafe forced reboot

If none of the above methods are able to successfully reboot your device, you can force a reboot. Note that this method is equivalent to pulling the battery from the HoloLens, and as such, is a dangerous operation which may leave your device in a corrupt state.

WARNING

This is a potentially harmful method and should only be used in the event none of the above methods work.

1. Press and hold the power button for at least 10 seconds
 - It's okay to hold the button for longer than 10 seconds
 - It's safe to ignore any LED activity
2. Release the button and wait 2-3 seconds

3. Power on the device again by pressing and holding the power button for 1 second

Reset the device to a factory clean state

If your HoloLens is still experiencing issues after rebooting, you can try resetting it to a factory clean state. If you reset your device, all your personal data, apps, and settings will be erased. Resetting will only install the latest installed version of Windows Holographic and you will have to redo all the initialization steps (calibrate, connect to WiFi, create a user account, download apps, etc...).

1. Launch the **Settings** app -> **Update** -> **Reset**
2. Select the **Reset device** option and read the confirmation dialog
3. If you agree to reset your device, the device will reboot and display a set of spinning gears with a progress bar
4. Wait about 30 minutes for this process to complete
5. The reset will complete and the device will reboot into the out of the box experience

Perform a full device recovery

If, after performing the above options, your device is **still** frozen, unresponsive, or experiencing update or software problems you can recover it using the Windows Device Recovery Tool. Recovering your device is similar to resetting it in the sense that it will erase all user content on the device, including apps, games, photos, user accounts, and more. If possible, backup any information you want to keep.

To fully recover your HoloLens:

1. Disconnect all phones and Windows devices from your PC
2. Install and launch the [Windows Device Recovery Tool](#) (WDRT) on your PC
3. Connect your HoloLens to your PC using the micro-USB cable it came with
 - Note that not all USB cables are created equal. Even if you've been using another cable successfully, this flow will expose new states where the cable may not perform as well. The one your HoloLens came with is the best and most well tested option
4. If the tool automatically detects your device it will display a HoloLens tile. Click it and follow the instructions to complete the process

NOTE

WDRT may recover your device to an older version of Windows Holographic; you may need to install updates after flashing

If the tool is unable to detect your device automatically, try the following:

1. Reboot your PC and try again (this fixes most issues)
2. Click the **My device was not detected** button, choose **Microsoft HoloLens**, and follow the rest of the instructions on the screen

Saving and finding your files

11/6/2018 • 2 minutes to read • [Edit Online](#)

Files can be saved and managed in a similar manner to other Windows 10 Desktop and Mobile devices:

- Using the File Explorer app to access local folders
- Within an app's own storage
- In a special known folder (such as the video or music library)
- Using a storage service that includes an app and file picker (such as OneDrive)
- Using a desktop PC connected to your HoloLens via USB, via MTP (Media Transfer Protocol) support

File Explorer

You can use the File Explorer app to move and delete files from within HoloLens.

NOTE

If you don't see any files in File Explorer, the "Recent" filter may be active (clock icon is highlighted in left pane). To fix this, select the **This Device** document icon in the left pane (beneath the clock icon), or open the menu and select **This Device**.

Files within an app

If an application saves files on your device, you can use that application to access them.

Where are my photos/videos?

Mixed reality capture photos and videos are saved to the device's Camera Roll folder. These can be accessed via the [Photos app](#). You can use the Photos app to sync your photos and videos to OneDrive. You can also access your photos and videos via the Mixed Reality Capture page of the [Windows Device Portal](#).

Requesting files from another app

An application can request to save a file or open a file from another app via [file pickers](#).

Known folders

HoloLens supports a number of [known folders](#) that apps can request permission to access.

Files in a service

To save a file to (or access files from) a service, the app associated with the service has to be installed. In order to save files to and access files from OneDrive, you will need to install the [OneDrive app](#).

MTP (Media Transfer Protocol)

Similar to other mobile devices, connect HoloLens to your desktop PC and open File Explorer on the PC to access your HoloLens libraries (photos, videos, documents) for easy transfer.

Clarifications

- HoloLens does not support connecting to external hard drives or SD cards.

- As of the [Windows 10 April 2018 Update \(RS4\) for HoloLens](#), HoloLens includes File Explorer for saving and managing files on-device. The addition of File Explorer also gives you the ability to choose your file picker (for example, saving a file to your device or to OneDrive).

See your photos on HoloLens

11/6/2018 • 2 minutes to read • [Edit Online](#)

When you take [mixed reality photos and videos](#) on HoloLens, they are saved to the "Camera roll" folder. The built-in Photos app can be used to view, manage, and share photos and videos. You can also install the [OneDrive app](#) from the Microsoft Store to sync photos to other devices.

Photos app

The Photos app is one of the default apps on the Start menu, and comes built-in with HoloLens. You can learn more about using the Photos app to view content and place it in your physical environment [on the HoloLens support site](#).

OneDrive app

With [OneDrive](#) you can access, manage, and share your photos and videos with any device and with any user. To access the photos and videos captured on HoloLens, first download the [OneDrive app](#) from the Microsoft Store on your HoloLens. Then open the OneDrive app and select **Settings > Camera upload**, and turn on **Camera upload**.

USB

If your HoloLens is running the [Windows 10 April 2018 update](#) or later, you can connect your HoloLens to a Windows 10 PC over USB to browse photos and videos on the device using MTP (media transfer protocol). You'll need to make sure the device is unlocked to browse files if you have a PIN or password set up on your device.

Windows Device Portal

If you have enabled the [Windows Device Portal](#), you can use it to browse, retrieve, and manage the photos and videos stored on your device.

See also

- [Photos on HoloLens support article \(external link\)](#)
- [Mixed reality capture for developers](#)
- [Mixed reality capture](#)
- [Windows Device Portal](#)

Sensor tuning

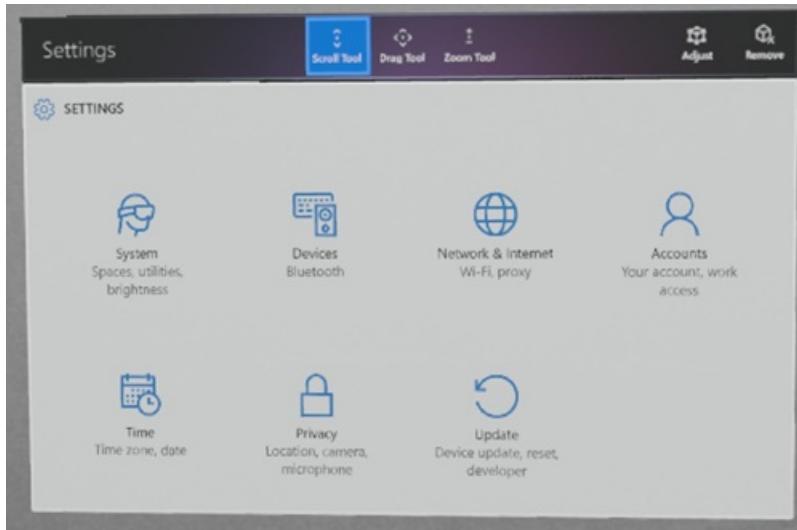
11/6/2018 • 2 minutes to read • [Edit Online](#)

WARNING

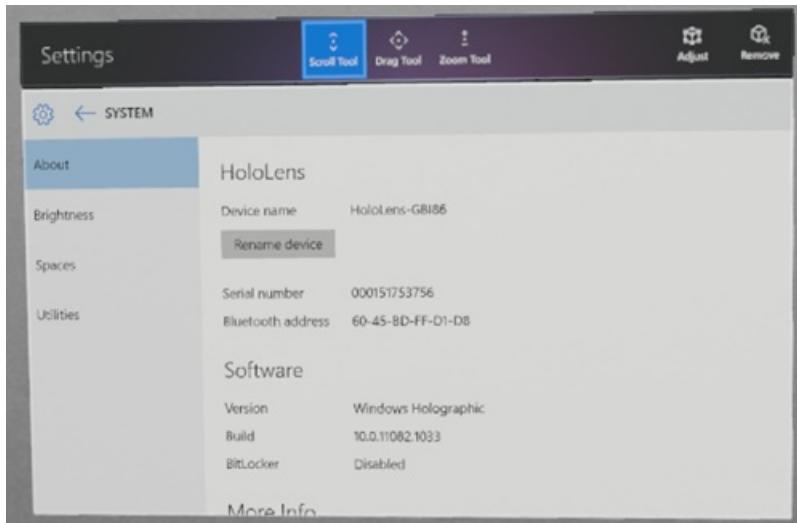
This app is no longer available on HoloLens as of the Windows 10 April 2018 Update. The sensors now recalibrate themselves automatically.

The sensor tuning utility allows HoloLens to update its sensor calibration information. Running this application can improve hologram quality.

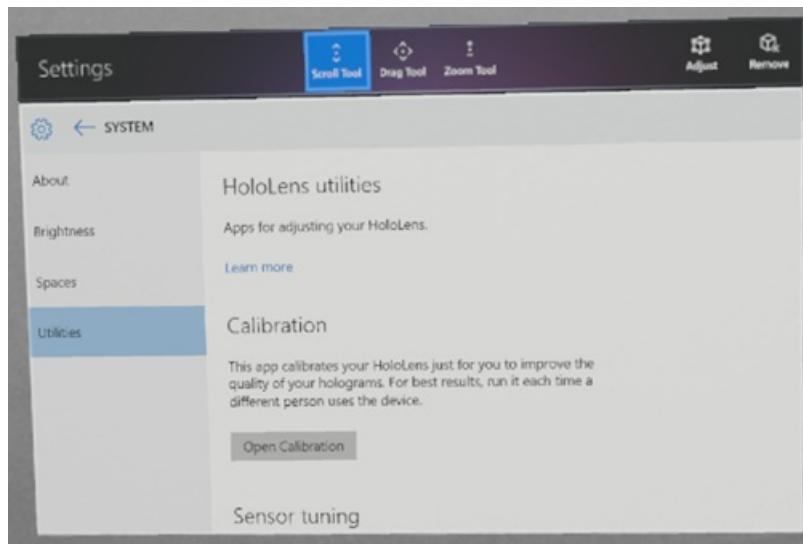
1. Open Settings App



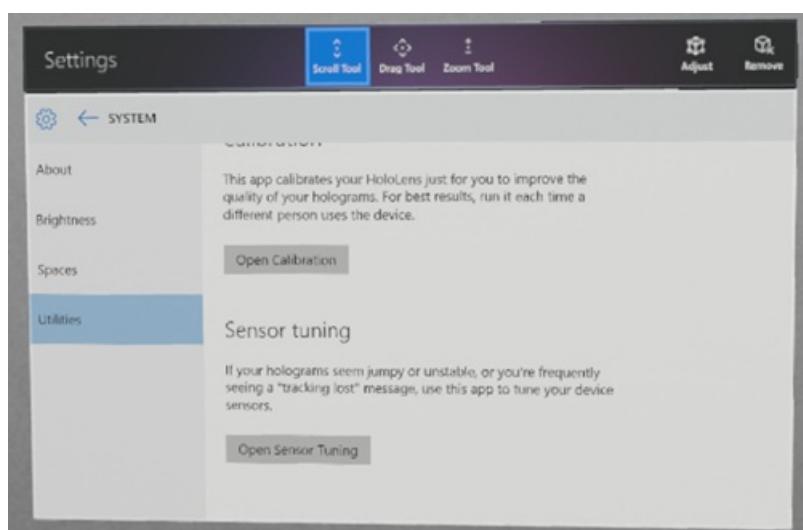
2. Select System



3. Select Utilities



4. Scroll down to Sensor Tuning



Updating HoloLens

11/6/2018 • 2 minutes to read • [Edit Online](#)

Your HoloLens will automatically download and install system updates whenever it is plugged-in and connected to Wi-Fi, even when it is in standby. However, you can also trigger a system update manually if there is one available.

Manual update

To perform a manual update:

- Open the **Settings** app.
- Navigate to **Update & Security > Windows Update**.
- Select the **Check for updates** button.

If an update is available, it will start downloading the new version. Once the download is complete, select the **Restart Now** button to trigger the installation. If your device is below 40% and not plugged in, restarting will not start installing the update.

While your HoloLens is installing the update, it will display spinning gears and a progress indicator. Do not turn off your HoloLens during this time. It will reboot automatically once it has completed the installation.

You can verify the system version number by opening the **Settings** app and selecting **System** and then **About**. HoloLens currently supports applying only one update at a time, so if your HoloLens is more than one version behind the latest you may need to run through the update process multiple times to get it fully up to date.

Windows Insider Program on HoloLens

By joining the Windows Insider Program from your HoloLens, you'll get access to preview builds of HoloLens software updates before they're available to the general public.

For information on how to participate in the Windows Insider Program on HoloLens, please refer to the [Insider preview for Microsoft HoloLens](#) article in our IT Pro docs.

Contributing to Windows Mixed Reality developer documentation

11/7/2018 • 8 minutes to read • [Edit Online](#)

Welcome to the [public repo for Windows Mixed Reality developer documentation](#)! Any articles you create or edit in this repo **will be visible to the public**.

Windows Mixed Reality docs are now on the docs.microsoft.com platform, which uses GitHub-flavored Markdown (with Markdig features). Essentially, the content you edit in this repo gets turned into formatted and stylized pages that show up at <https://docs.microsoft.com/windows/mixed-reality>.

This page covers the basic steps and guidelines for contributing, as well as links to Markdown basics. Thank you for your contribution!

Before you start

If you don't already have one, you'll need to [create a GitHub account](#).

NOTE

If you're a Microsoft employee, link your GitHub account to your Microsoft alias on the [Microsoft Open Source portal](#). Join the "**Microsoft**" and "**MicrosoftDocs**" organizations).

When setting up your GitHub account, we also recommend these security precautions:

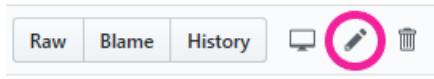
- Create a [strong password for your GitHub account](#).
- Enable [two-factor authentication](#).
- Save your [recovery codes](#) in a safe place.
- Update your [public profile settings](#).
 - Set your name, and consider setting your *Public email* to *Don't show my email address*.
 - We recommend you upload a profile picture, as a thumbnail will be shown on docs pages to which you contribute.
- If you plan to use a command line workflow, consider setting up [Git Credential Manager for Windows](#) so that you don't have to enter your password each time you make a contribution.

Taking these steps is important, as the publishing system is tied to GitHub and you'll be listed as either author or contributor to each article using your GitHub alias.

Editing an existing article

Use the following workflow to make updates to *an existing article* via GitHub in a web browser:

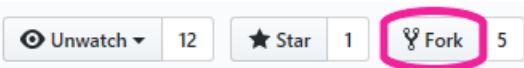
1. Navigate to the article you wish to edit in the "mixed-reality-docs" folder.
2. Select the edit button (pencil icon) in the top right. This will automatically fork a disposable branch off the 'master' branch.



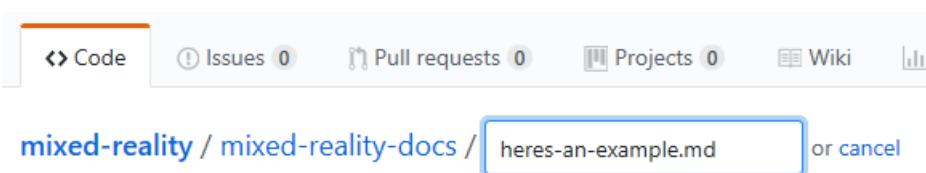
3. Edit the content of the article (see "[Markdown basics](#)" below for guidance).
4. Update metadata as relevant at the top of each article:
 - title: This is the page title that appears in the browser tab when the article is being viewed. As this is used for SEO and indexing, you shouldn't change the title unless necessary (though this is less critical before documentation goes public).
 - description: Write a brief description of the article's content. This aids in SEO and discovery.
 - author: If you are the primary owner of the page, add your GitHub alias here.
 - ms.author: If you are the primary owner of the page, add your Microsoft alias here (you don't need @microsoft.com, just the alias).
 - ms.date: Update the date if you're adding major content to the page, but not for fixes like clarification, formatting, grammar, or spelling.
 - keywords: Keywords aid in SEO (search engine optimization). Add keywords, separated by a comma and a space, that are specific to your article (but no punctuation after the last keyword in your list); you don't need to add global keywords that apply to all articles, as those are managed elsewhere.
5. When you've completed your article edits, scroll down and click the **Propose file change** button.
6. On the next page, click **Create pull request** to merge your automatically-created branch into 'master.'
7. Repeat the steps above for the next article you want to edit.

Creating a new article

Use the following workflow to *create new articles* in the documentation repo via GitHub in a web browser:

1. Create a fork off the MicrosoftDocs/mixed-reality 'master' branch (using the **Fork** button in the top right).
- 
2. In the "mixed-reality-docs" folder, click the **Create new file** button in the top right.

3. Create a page name for the article (use hyphens instead of spaces and don't use punctuation or apostrophes) and append ".md"



IMPORTANT

Make sure you create the new article from within the "mixed-reality-docs" folder. You can confirm this by checking for "/mixed-reality-docs/" in the new file name line.

4. At the top of your new page, add the following metadata block:

```
---  
title:  
description:  
author:  
ms.author:  
ms.date:  
ms.topic: article  
keywords:  
---
```

5. Fill in the relevant metadata fields per the instructions in the [section above](#).
6. Write article content using [Markdown basics](#).
7. Add a `## See also` section at the bottom of the article with links to other relevant articles.
8. When finished, click **Commit new file**.
9. Click **New pull request** and merge your fork's 'master' branch into MicrosoftDocs/mixed-reality 'master' (make sure the arrow is pointing the correct way).



Markdown basics

The following resources will help you learn how to edit documentation using the Markdown language:

- [Markdown basics](#)
- [Markdown-at-a-glance reference poster](#)
- [Additional resources for writing Markdown for docs.microsoft.com](#)
- [Unique Markdig differences](#) and [Markdig multi-column support](#)

Adding tables

Because of the way docs.microsoft.com styles tables, they won't have borders or custom styles, even if you try inline CSS. It will appear to work for a short period of time, but eventually the platform will strip the styling out of the table. So plan ahead and keep your tables simple. [Here's a site that makes Markdown tables easy](#).

The [Docs Markdown Extension for Visual Studio Code](#) also makes table generation easy if you're using [Visual Studio Code \(see below\)](#) to edit the documentation.

Adding images

You'll need to upload your images to the "mixed-reality-docs/images" folder in the repo, and then reference them appropriately in the article. Images will automatically show up at full-size, which means if your image is large, it'll fill the entire width of the article. Thus, we recommend pre-sizing your images before uploading them. The recommended width is between 600 and 700 pixels, though you should size up or down if it's a dense screenshot or a fraction of a screenshot, respectively.

If you're a Microsoft employee, you can find solid guidance on formatting images [here](#).

IMPORTANT

You can only upload images to your forked repo before merging. So, if you plan on adding images to an article, you'll need to use [Visual Studio Code](#) to add the images to your fork's "images" folder first or make sure you've done the following in a web browser:

1. Forked the MicrosoftDocs/mixed-reality repo.
2. Edited the article in your fork.
3. Uploaded the images you're referencing in your article to the "mixed-reality-docs/images" folder in your fork.
4. Created a **pull request** to merge your fork into the MicrosoftDocs/mixed-reality 'master' branch.

To learn how to set up your own forked repo, follow the instructions for [creating a new article](#).

Previewing your work

While editing in GitHub via a web browser, you can click the **Preview** tab near the top of the page to preview your work before committing.

NOTE

Previewing your changes on review.docs.microsoft.com is only available to Microsoft employees

Microsoft employees: once your contributions have been merged into the 'master' branch, you can see what the documentation will look like before it goes public at <https://review.docs.microsoft.com/en-us/windows/mixed-reality/install-the-tools?branch=master> (find your article using the table of contents in the left column).

Editing in the browser vs. editing with a desktop client

Editing in the browser is the easiest way to make quick changes, however, there are a few disadvantages:

- You don't get spell-check.
- You don't get any smart-linking to other articles (you have to manually type the article's filename).
- It can be a hassle to upload and reference images.

If you'd rather not deal with these issues, you may prefer to use a desktop client like [Visual Studio Code](#) with a couple [helpful extensions](#) to contribute to documentation.

Using Visual Studio Code

For the reasons listed [above](#), you may prefer using a desktop client to edit documentation instead of a web browser. We recommend using [Visual Studio Code](#).

Setup

Follow these steps to configure Visual Studio Code to work with this repo:

1. In a web browser:
 - a. Install [Git for your PC](#).
 - b. Install [Visual Studio Code](#).
 - c. [Fork MicrosoftDocs/mixed-reality](#) if you haven't already.
 - d. In your fork, click **Clone or download** and copy the URL.
2. Create a local clone of your fork in Visual Studio Code:
 - a. From the **View** menu, select **Command Palette**.
 - b. Type "Git:Clone."

- c. Paste the URL you just copied.
- d. Choose where to save the clone on your PC.
- e. Click **Open repo** in the pop-up.

Editing documentation

Use the following workflow to make changes to the documentation with Visual Studio Code:

NOTE

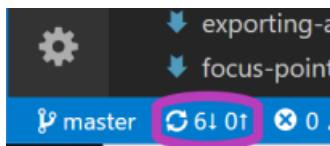
All the guidance for [editing](#) and [creating](#) articles, and the [basics of editing Markdown](#), from above applies when using Visual Studio Code as well.

1. Make sure your cloned fork is up-to-date with the official repo.

- a. In a web browser, create a pull request to sync recent changes from other contributors in MicrosoftDocs/mixed-reality 'master' to your fork (make sure the arrow is pointing the right way).

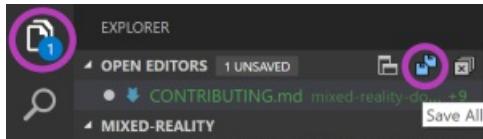


- b. In Visual Studio Code, click the sync button to sync your freshly updated fork to the local clone.

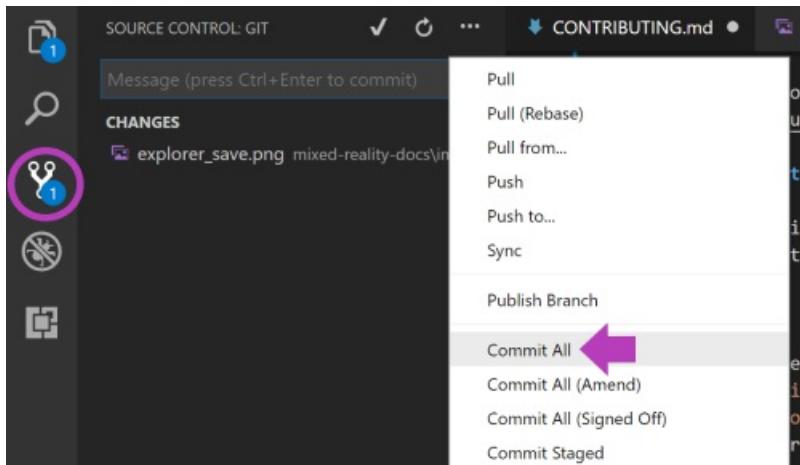


2. Create or edit articles in your cloned repo using Visual Studio Code.

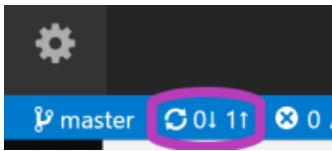
- a. Edit one or more articles (add images to "images" folder if necessary).
- b. **Save** changes in **Explorer**.



- c. **Commit all** changes in **Source Control** (write commit message when prompted).



- d. Click the **sync** button to sync your changes back to origin (your fork on GitHub).



3. In a web browser, create a pull request to sync new changes in your fork back to MicrosoftDocs/mixed-reality 'master' (make sure the arrow is pointing the correct way).

The screenshot shows a GitHub pull request interface. At the top, there are dropdown menus for 'base fork: MicrosoftDocs/mixed-reality', 'base: master', 'head fork: mattztest/mixed-reality', and 'compare: master'. Below these, a green checkmark icon indicates that the branches are 'Able to merge'. The text 'These branches can be automatically merged.' is displayed in green.

Useful extensions

The following Visual Studio Code extensions are very useful when editing documentation:

- [Docs Markdown Extension for Visual Studio Code](#) - Use **Alt+M** to bring up a menu of docs authoring options like:
 - Search and reference images you've uploaded.
 - Add formatting like lists, tables, and docs-specific call-outs like `>[!NOTE]`.
 - Search and reference internal links and bookmarks (links to specific sections within a page).
 - Formatting errors are highlighted (hover your mouse over the error to learn more).
- [Code Spell Checker](#) - misspelled words will be underlined; right-click on a misspelled word to change it or save it to the dictionary.