Bose AR SDK for Unity

Version 0.13.0

Contents

BoseWearable-Unity

The Bose AR SDK for Unity contains everything needed to build a Unity application for supported platforms that will search for, connect to, and pull sensor and gesture data from any Bose AR device.

The provided package includes:

- The Bose AR SDK for Unity and necessary native binaries for supported platforms.
- Prefabs and Components that enable drag-and-drop development to hit the ground running.
- Example content that can be built to device to view demos showcasing best practices and practical usage of the API.

Supported Platforms

The Bose AR SDK for Unity currently supports the following platforms:

- Android (5.0+)
- iOS (11.4+)

Getting Started

Basic Guides

- Build, test and study the included demo content.

Advanced Guides

- Discover and connect to Bose AR devices.
- Perform on-demand device configuration with Requirements.
- Use different Providers to speed up your development.
- Learn about the managing the sensors and their data.
- Display the connected device model at runtime.
- Troubleshooting common issues with the Unity SDK.

Requirements

Specific requirements for various platforms are listed below. When possible, we have included pre- and post-processors for the build stage to automatically enforce these settings and avoid troubleshooting issues that might arise from improper configuration.

Unity

- **Unity:** 2017.4 or greater

Platform: Android

- **Android Studio:** 3.2 or greater

Your Unity project must have the following settings enabled:

- **Minimum API Level:** Android 5.0 'Lollipop' (API Level 21)
- **Unsafe Code:** Enabled

Platform: iOS

- **Xcode:** 10.2 or greater

Your Unity project must have the following settings enabled:

- **Required Architecture:** ARM64
- **Supported iOS Versions:** iOS 11.4+
- **Background Behavior:** Custom
- **Background Modes:** Uses BLE Accessories
- **Unsafe Code:** Enabled

Your Xcode Project must have the following property in your `Info.plist`:

```
<key>NSBluetoothPeripheralUsageDescription</key>
<string>
    This app uses Bluetooth to communicate with Bose AR devices.
</string>
```

Firmware

Please be sure your Bose AR device is running the latest available firmware.

Release Notes

For further information on new features and fixes in the releases of the Bose AR SDK for Unity, see the release notes.

Unity Demos

Testing the Demos

This SDK can build an example app to demonstrate various features and functionality provided. We encourage you to test it out on device to get a quick understanding of the basic functionality and connection flow documented below.

To build this demo:

1. Ensure your editor is set to a supported platform.
2. Select **Tools** > **Bose Wearable** > **Build Wearable Demo**

This will prompt you to provide a location to save an executable or a project to build and deploy the Unity SDK demo app.

Included Content

There are multiple demos included with the Unity project to demonstrate various use-cases of accessing and using sensor and gesture data in an application.

Those are covered in more detail here:

- Basic Demo
- Advanced Demo
- Gesture Demo

Interacting with a Bose AR device from a Unity application at a high level involves searching for nearby devices, connecting to one, and engaging sensors for their data. Only once the device is connected is it possible to capture data from its sensors for in-application use.

NOTE: It is not necessary to pair the Bose AR device with the phone or tablet prior to using the Bose AR SDK for Unity if only sensor data and/or gesture data is desired (utilizes Bluetooth LE); pairing is only needed if audio should be streamed from the app to the Bose AR device (utilizes Bluetooth classic, must be paired in phone settings external to app).

Removing this Content

The following folders can be deleted without removing the core SDK functionality:

- `Bose/Wearable/Connection`

- `Bose/Wearable/Examples`
    - Content in this folder depends on some UI assets present in the `Connection` folder.
- `Bose/Wearable/ModelLoader`

Basic Demo

Summary

This demo provides a demonstration of automatically loading a model of your connected Bose AR device, rotating it based on device orientation, and a rudimentary calibration method.

The `RotationMatcher` component (on the "BasicDemoController" GameObject) is used to apply the rotation of the Bose AR device to a Unity GameObject. To demonstrate this, the component will be applied to a 3D model that looks like the connected Bose AR device. This enables the representation of the device to match their physical orientation.

The "WearableModelLoader" prefab (a child of the "BasicDemoController" GameObject) is a drop-in prefab setup to load an accurate 3D representation of Bose AR products upon load, with safe fallbacks to ensure a visual representation of a device is always loaded.

In this demo, we use the rotation source `RotationSensorSource.SixDof` -- providing six degrees of freedom. For more information, please see Configuring the Rotation Sensor Source.

Functionality

Raw Data

Tapping on the "Raw" button sets the rotation of the 3D model to be directly aligned to the sensor data provided by the device.

Calibrate

Tapping on the "Calibrate" button caches the current orientation of the Bose AR device, treating that value as the new base "forward" rotation.

Advanced Demo

Summary

This demo provides a more advanced use-case of using the Bose AR SDK for Unity by demonstrating a simple gameplay mechanic where you swivel your head in several directions to collect objects.

In this demo, we use the rotation source `RotationSensorSource.NineDof` -- providing nine degrees of freedom. For more information, please see Configuring the Rotation Sensor Source.

Functionality

A rectangular widget and a cone are positioned at the center of the screen, pointing towards the view direction of the Bose AR device. The radius of the cone represents the orientation uncertainty of the hardware. A wireframe polyhedron surrounds the cone with visible vertices and edges, representing virtual space around the user’s head.

At the beginning of play, we run a simple calibration by asking players to look straight ahead and keep their head still. After the Bose AR device reports minimal movement, the capsule and cone will match orientation with the player's head rotation.

Periodically, a glowing target appears at one of the vertices, and sound begins playing from the target. The player must orient their Bose AR device to face that direction; upon reaching it (within a threshold), the target will grow in scale. Sustaining that orientation for a number of seconds will cause it to be collected and a new point to be spawned.

Gesture Demo

Summary

This demo provides a demonstration of automatically detecting and reacting to the various gestures provided by the hardware.

The `GestureDetector` component (on the "GestureDisplay" GameObjects) is updated at runtime to automatically trigger an event when a given gesture is detected. This is exhibited by a small animation and sound effect being played whenever a gesture is detected.

Functionality

To trigger an animation and sound effect, simply perform one of the gestures detected by your Bose AR device.

Detected gestures include:

1. **DoubleTap**: Double-tap the right side of your device with your finger.
2. **HeadNod**: Nod your head in a "yes" motion.
3. **HeadShake**: Nod your head in a "no" motion.

Device Discovery and Connection

There are two ways of connecting to a Bose AR device for sensor data:

- Using the included WearableConnectUIPanel Prefab
- Writing your own connection logic with the WearableControl API

Using the Prefabs

The `WearableConnectUIPanel` and `WearableDeviceDisplayButton` are two classes with their own prefabs provided. They provide a wrapper around WearableControl, add additional events for UX feedback for a user, and serve as a good example of how to best search for, display, and maintain a connection to a device.

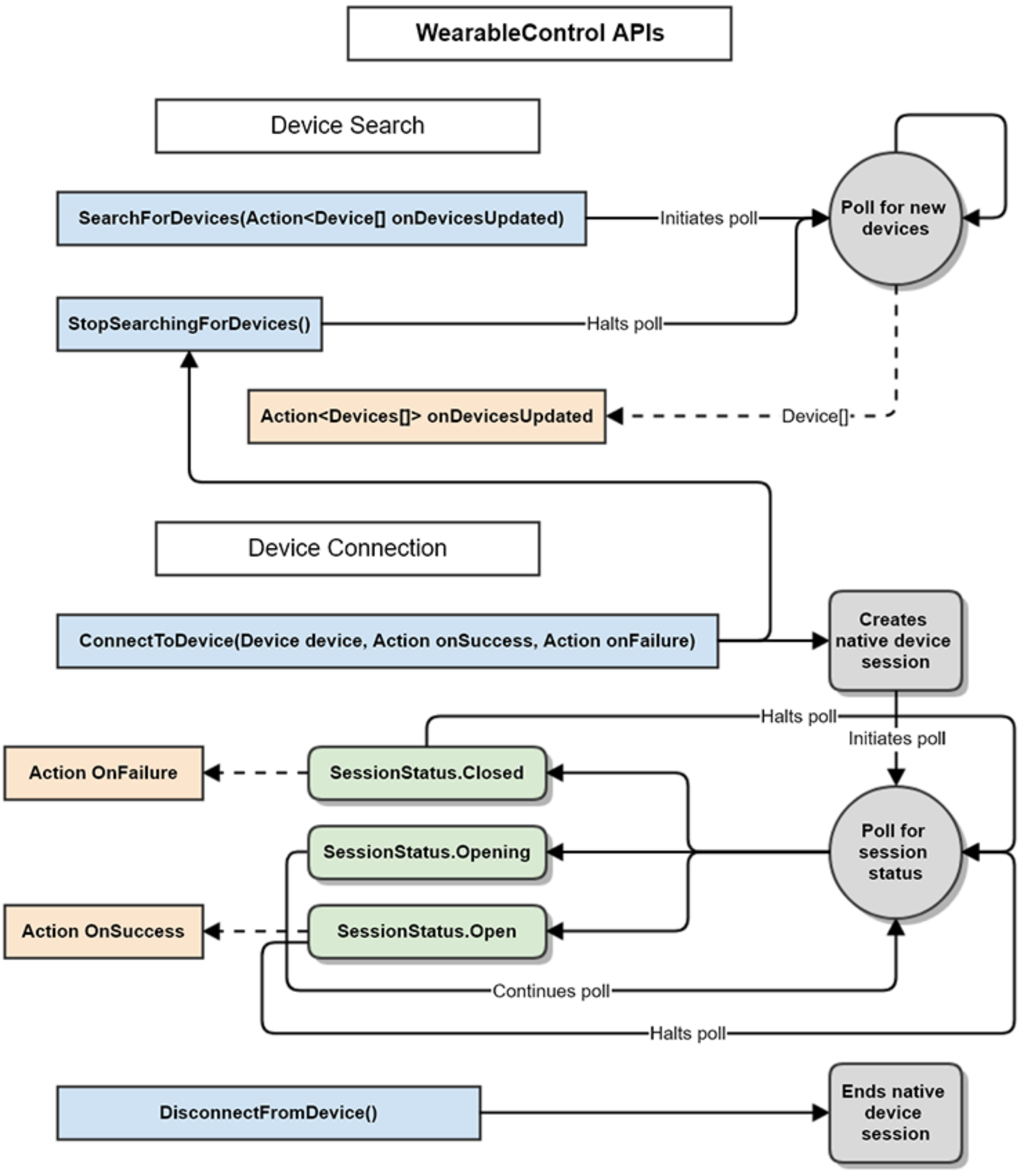
To quickly add the ability to connect to a Bose AR device, drop the `Bose/Wearable/Connection/Prefabs/WearableConnectUIPanel` Prefab into the scene you would like to the device connection screen to first appear.

If this is a brand new project, be sure to also add an `EventSystem` with the `Standard Input Module` into your scene as well.

NOTE: There is an atlas in the `Connection/Atlas` folder that has all of the art assets used for the prefabs. If customization or use of just the scripts is desired, you may want to modify or delete this. See here for more information on Unity Sprite Atlases and Sprite Packing.

Starting from scratch

WearableControl has several low-level APIs for device search and connection that can be used to directly implement a player flow or UI.



Searching for Devices

When searching for devices, the plugin periodically polls for new devices and dispatches them to a user-passed callback. This continues until either an attempt to connect to a device takes place or WearableControl is explicitly told to stop searching.

Connecting to Devices

Anytime a device is connected to, a session is created; this persists for the entire time it remains connected.

This session is created in native code when calling `ConnectToDevice` and then the status is continually polled until its value is either `Opened` or `Closed`. Once it results in either value, the user-supplied callback for `onSuccess` or `onFailure` is invoked.

This session can end either automatically when the device has become disconnected or explicitly by a user when DisconnectFromDevice is called. There are global callbacks on WearableControl for when a device connection or disconnection occurs via `DeviceConnected` and `DeviceDisconnected` that you should hookup to your gameplay or UI code to handle pausing and resuming actions when this occurs.

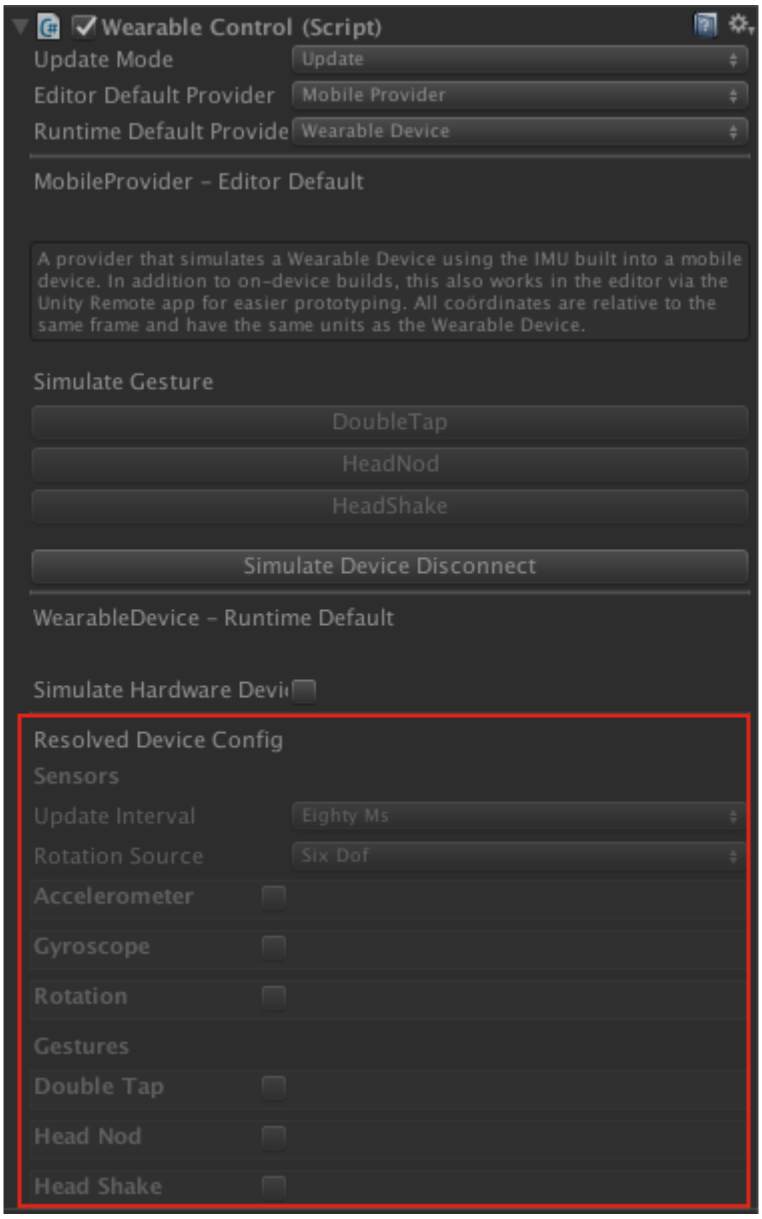
Device Configuration with WearableRequirement

Manually configuring sensors, gestures, and the global `SensorUpdateInterval` are possible via public methods and properties on `WearableControl`, but when many different systems or components demand different requirements from a device it is possible for these to conflict and overwrite each other.

The `WearableRequirement` component overcomes this challenge by allowing a user to define a desired device configuration -- either manually through the inspector or programmatically at runtime.

Each `WearableRequirement` instance is resolved with all other active `WearableRequirement`s to create an aggregate device configuration that prioritizes requests for enabling sensors or gestures over disabling them and a faster `SensorUpdateInterval` over a slower one. This allows for various systems or components to have different `WearableRequirement` configurations, but the resolved device configuration will support them all.

Under the hood, programmatic changes made via `WearableControl` itself acts as a `WearableRequirement` and factors into the same resolution process. The resolved device config representing the current device state can be seen on the inspector below.



Wearable Control (Script)

Update Mode: Update

Editor Default Provider: Mobile Provider

Runtime Default Provide: Wearable Device

MobileProvider - Editor Default

A provider that simulates a Wearable Device using the IMU built into a mobile device. In addition to on-device builds, this also works in the editor via the Unity Remote app for easier prototyping. All coördinates are relative to the same frame and have the same units as the Wearable Device.

Simulate Gesture

DoubleTap

HeadNod

HeadShake

Simulate Device Disconnect

WearableDevice - Runtime Default

Simulate Hardware Devi: ☐

Resolved Device Config

Sensors

Update Interval: Eighty Ms

Rotation Source: Six Dof

Accelerometer: ☐

Gyroscope: ☐

Rotation: ☐

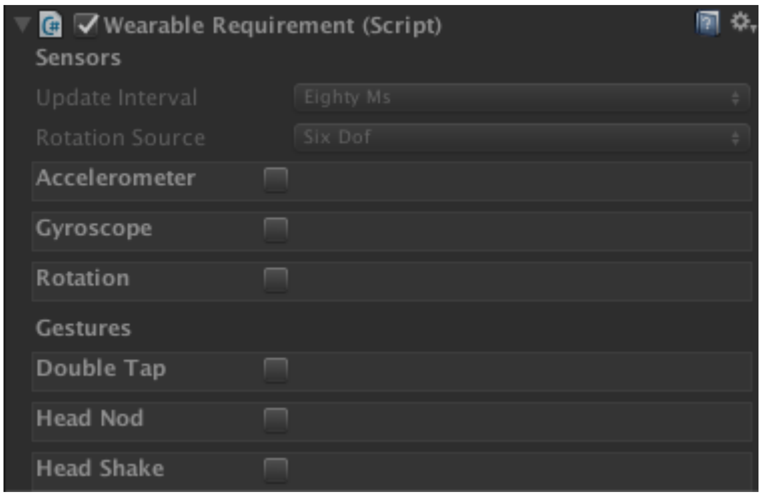
Gestures

Double Tap: ☐

Head Nod: ☐

Head Shake: ☐

Configuring a `WearableRequirement` can be done either programmatically or through the inspector (edit or runtime).



Wearable Requirement (Script)

Sensors

Update Interval: Eighty Ms

Rotation Source: Six Dof

Accelerometer: ☐

Gyroscope: ☐

Rotation: ☐

Gestures

Double Tap: ☐

Head Nod: ☐

Head Shake: ☐

Configuration Resolution

`WearableRequirement` makes a configuration request when enabled or altered and removes that request when disabled or destroyed. When any of these changes occur, it will trigger the device configuration to be resolved and alter the device's current configuration if necessary.

Configuration resolution is completed at the end of that Unity render frame (in `LateUpdate`) and prevents further device updates for a short period of time. If another configuration change is requested during this lockout period, it will set a flag to repeat the resolution process when the lockout period is complete.

Unity Object Lifecycle Support

One of the benefits of utilizing `WearableRequirement` to configure device state is that proper object lifecycle management (via disabling or destroying a `WearableRequirement` when not in use) will automatically throttle the device's capabilities and preserve battery power.

Usage Example

To better illustrate the benefit of this system, consider the following example:

Your game uses the Gyroscope at a 80ms update interval for a particular minigame. For the entire game, including the minigame, you are using Rotation sensor running a 40ms update interval to orient the player.

If you only used the WearableControl API, you would have to manage switching these sensors and updates yourself when entering and exiting the minigame. However, by simply attaching (or programmatically creating) `WearableRequirement`s for both of these two device states, the underlying system will end up delivering both the Gyroscope and Rotation sensor data at 40ms for the duration of the minigame. Upon leaving the minigame (and those objects being destroyed or disabled) the system would automatically disable the Gyroscope for you, ensuring that the least amount of data is transmitted over BLE; effectively optimizing battery usage for your users.

Managing sensors and their data.

Sensor Data

Overview

There are three sensors that can be utilized on a Bose AR device:

- **Accelerometer:** Emits a `Vector3` representing acceleration in m/s^2.
- **Gyroscope:** Emits a `Vector3` representing angular velocity in rad/s.
- **Rotation:** Emits a `Quaternion` representing a quaternion indicating the device's orientation in space.

Activating Sensors

Once a device has been connected, all of its sensors are off by default.

You may activate a sensor by:

- Establishing a WearableRequirement for that sensor (recommended).
- Call the `Start` method on the relevant sensor in `WearableControl`.

If a sensor is turned on, its data will be included with the `SensorFrame`s available on `WearableControl`; if not, the last emitted value (or default value if no data has been received) will be set in any `SensorFrame`s emitted by the device via `WearableControl`.

Configuring the Rotation Sensor Source

The IMU hardware provides two modes for determining rotation -- providing either six or nine degrees of freedom -- each of which has unique advantages and drawbacks.

- **6DOF**, the default mode, uses only the gyroscope and accelerometer to determine the device's orientation relative to the position it was in when powered on.
- **9DOF** uses the magnetometer in addition to the gyroscope and accelerometer to determine orientation. When calibrated, it provides orientation relative to magnetic north. See the Bose AR documentation for more information on how to calibrate.

You may configure the Rotation Source of the Rotation sensor by:

- Establishing a WearableRequirement for that Rotation and set the Rotation Source (recommended).

- Call `SetRotationSource()` method of `WearableControl` to switching between these modes at runtime.

Accessing Sensor Data

After any sensors have been successfully activated, there are several ways to access the data.

- `WearableControl.Instance.CurrentSensorFrames` provides an array of all `SensorFrame` s received in the last poll.
- `WearableControl.Instance.LastSensorFrame` provides the last sensor frame received.
- `WearableControl.Instance.SensorsUpdated` provides an event that triggers for every individual sensor received. (**NOTE:** This event will likely be called several times per Unity frame.)

Gesture Data

Overview

There are three gestures that can be automatically detected on a Bose AR device:

- **Double-Tap:** Double-tap the right side of your device.
- **Head Nod:** A head nod resembling the "yes" body language.
- **Head Shake:** A head shake resembling the "no" body language.

Enabling Gestures

Similar to sensors, you must enable the use of individual gestures in order to receive events related to their detection.

You may enable a gesture by:

- Establishing a WearableRequirement for that gesture (recommended).
- Call the `Enable` method on an explicit `WearableGesture` defined in WearableControl (such as `WearableControl.Instance.DoubleTapGesture`).
- Call the `Enable` method on a `WearableGesture` returned by `WearableControl.Instance.GetWearableGestureById(GestureId gestureId)` .

Accessing Gesture Data

After any gestures have been successfully enabled, there are several ways to access the data.

- Subscribe to a specific gesture event, e.g. `WearableControl.Instance.DoubleTapDetected`
- Subscribe to `WearableControl.Instance.GestureDetected` and test against the received GestureId.

- Read the incoming `SensorFrame`s. (See Accessing Sensor Data.)

You may then subscribe to a specific gesture event or the generic to receive events for a detected gesture.

Data Aggregation

All sensor data is aggregated into a single `SensorFrame` struct; it will contain the last received (or default values if no data has been received) for each sensor, a timestamp indicating the number of absolute seconds since the device was powered on, and a deltaTime that indicates the amount of time between sensor samples on the device.

The grouping of all of the sensor data onto a single struct helps enable systems to use data from multiple sensors that were captured at a specific point in time. The latest `SensorFrame` is always available at `WearableControl.Instance.LatestSensorFrame` while all frames from the last poll can be accessed at `WearableControl.Instance.CurrentSensorFrames`.

Sensor Notes

Consider the following when working with the sensors:

Accelerometer

- Measures acceleration relative to the glasses in m/s^2.
- In addition to acceleration due to motion, effect of gravity is always present as a 1g (9.806m/s^2) upwards force.
- Inherently less accurate than gyroscope: susceptible to noise and vibration.
- Acceleration is generally only helpful when detecting shaking movements or gravity, and isn't useful for determining position or velocity.

Gyroscope

- Measures angular velocity relative to the glasses in rad/s.
  - Angular velocity is a vector quantity pointing in the direction of the axis of rotation with a magnitude equal to the angle in radians swept through in one second.
- The gyro is very accurate in the short term, but can drift over time.
- The measurements from the gyroscope are angular velocities, not Euler angle velocities. They cannot be simply integrated to find the orientation.
  - It's often more useful to look at a single axis, such as pitch or yaw, when working with the gyroscope
  - If you need relative orientation, use Rotation and compare against a reference. This is more stable than trying to derive orientation from gyro data alone.

Rotation

- The rotation sensor is a virtual sensor that combines data from the gyroscope, accelerometer, and (optionally) magnetometer to obtain the device's orientation in space.
- When magnetic heading is not needed, prefer the 6DOF mode; it offers reduced latency, drift, noise, and settling time.
- Can work with the measured orientation directly, or compare to a reference rotation obtained by calibrating against a known orientation.
    - See the `RotationMatcher` component for an example of this
- Measurement uncertainty (in degrees) is provided by the sensor. It’s helpful to think about this as a cone containing the user’s actual orientation.
    - After enabling the sensor, the measurement uncertainty will start high and improve as the device stabilizes.
    - Quickly rotating the device may decrease the accuracy, but it will eventually settle within a short time period.
    - When using rotation to point to a target, make sure to take the uncertainty into account, even if it is not visible to the user. This can help users still interact even when the certainty of received data is low.
- The filtering performed by IMUs makes inferring rotational velocity from the orientation unreliable. If you need to calculate the speed and direction at which the Bose AR device is rotating, use the data from the gyro, transformed by the inverse of the orientation.
- Be aware that no IMU is perfect: the longer the device is running, the further the measured orientation will drift from its true value. If you are planning on running the device for long periods of time, consider re-calibrating periodically.

Working with Data

There are two ways to work with data from the sensor: using measurements directly (open-loop or feedforward) or making calculations that rely on previous data or results (closed-loop or feedback).

Generally, rotation is processed using feedforward systems, and acceleration/angular velocity using feedback systems.

Open-Loop / Feedforward Systems

In an open-loop system, measurement data from the sensors is used directly or transformed in some way, but does not rely on previous data or calculations. Each calculation is independent of those before and after it.

Usage examples:

- Using the accelerometer to estimate the pitch and roll of the glasses
- Detecting quick movements using the accelerometer and/or gyroscope
- Controlling the camera using the glasses' orientation
- Using the rotation sensor to point toward a target
- Rotating an object to match the glasses' orientation

Best Practices

- When using measurements directly in an open-loop/feedforward system, use `WearableControl.Instance.LastSensorFrame`. This ensures that data is always available for every Unity frame, and simplifies calculations.

Closed-Loop / Feedback Systems

In a closed-loop system, measurement data is combined with the results of previous calculations or data. Calculations depend on state, and often involve time in some capacity. Feedback systems generally show up when data is being integrated, differentiated, or filtered.

Usage examples:

- Moving an object on-screen using the gyroscope
- Calculating the average acceleration or angular velocity using a smoothing filter
- Calculating position or velocity by integrating acceleration or angular velocity

Best Practices

- When making calculations that depend on state, use the list of measurements in `WearableControl.Instance.CurrentSensorFrames`. This ensures the most accurate calculations, and prevents duplicate data if no new measurements were taken this frame.
- When working with feedback systems, remember that you are dealing with rates and velocities, not changes in value: multiply rates by the delta-time field in the sensor frame to find the change since last sensor frame

Data Providers

The Bose AR SDK for Unity supports multiple data sources to simplify prototyping and pave the way for future development. These distinct data sources are called providers, and can be swapped out both in the Unity editor and using scripts at runtime. Providers provide data seamlessly to client applications, allowing user code to interact with data independent of the underlying hardware.

One of the main goals of the provider interface was simplifying and speeding up development: for example, `WearableControl` can be set up to take data from the Wearable hardware when running on a device, and from a simulated source when running in-editor.

Wearable Device Provider

The Wearable Device Provider allows direct access to the hardware using the underlying Wearable SDK.

This provider is *only* available at runtime when built on a platform supported by the SDK.

USB Provider

The USB Provider provides all the same functionality in the Unity Editor that the Wearable Device Provider enables in a runtime environment. To use it, connect your Bose AR device to your Windows or macOS computer via the included USB cable and set the `WearableControl`'s **Editor Default Provider** to **USB Provider**.

This provider is *only* available in the Unity Editor and is not available on runtime platforms.

(**NOTE**: USB Audio is only supported on Bose Frames.)

Wearable Proxy Provider

The WearableProxy allows in-editor testing by streaming data over the network using a supplemental mobile app connected to a Wearable Device.

See Wearable Proxy for more information.

Mobile Provider

The Mobile Provider facilitates simulating device data using a mobile phone. It can be used two ways:

1. **In-Editor:** Using the free Unity Remote 5 app (iOS App Store, Google Play) a connected device will forward their motion data to Unity to mimic a WearableDevice.

2. **On Device:** When built to device, the Mobile Provider presents data from the mobile device's internal IMU in the same format as a Wearable Device.

This provider allows for iteration and tweaking of motion data without having to build the Unity project or even have a Wearable Device present.

Debug Provider

The Debug Provider represents a fully device-free provider implementation. Sensor configuration is respected and all calls into `WearableControl` are logged to the console.

The Debug Provider makes it easy to develop connection flows and basic data handling: connection and disconnection, as well as simple movement and gestures from a "virtual device" can be controlled and simluated with buttons on the `WearableControl` inspector.

Values reported by the provider are representative of what a real device would be reporting if it was physically rotating at a constant speed.

Provider Compatibility

Provider	Device Required?	Features (Editor)	Features (Runtime)
Wearable Device	Y	N/A	Data, Audio
USB	Y	Data, Audio[1]	N/A
Proxy	Y	Data	Data
Mobile	N	Data	Data, Audio
Debug	N	Data	Data, Audio

[1] When supported by the connected Bose AR device.

Wearable Proxy

The WearableProxy is a data provider that allows users to receive motion using a mobile proxy app, rather than connecting to a device directly. This facilitates seamless in-editor testing without having to repeatedly build to a mobile device. All functions of the SDK, including device searching and connecting, sensor configuration, and data acquisition are passed on through the proxy completely transparently, allowing for full in-editor testing functionality.

Building the Wearable Proxy Server

In order to use the proxy, a companion server app must be built and loaded onto a supported mobile device.

To build the Proxy Server:

1. Ensure your editor is set to a supported platform.
2. Select **Tools** > **Bose Wearable** > **Build Proxy Server**

Using the Wearable Proxy

- Find the "Wearable Control" object in the `Root` scene and change the **Editor Default Provider** field to "Wearable Proxy".
- Launch and start the proxy server app on the mobile device and note the IP and port number displayed, then enter them in the inspector.
- Press play in the editor and the SDK will automatically connect to the server and transparently pass on commands.
- Keep in mind that the Bose AR device should stay in proximity to your mobile device running the Proxy; not the client computer running the demo application.

Using the Wearable Proxy with your own apps is similar: ensure a GameObject with the `WearableControl` component exists in your scene and set the **Editor Default Provider** to the proxy, then enter the server information.

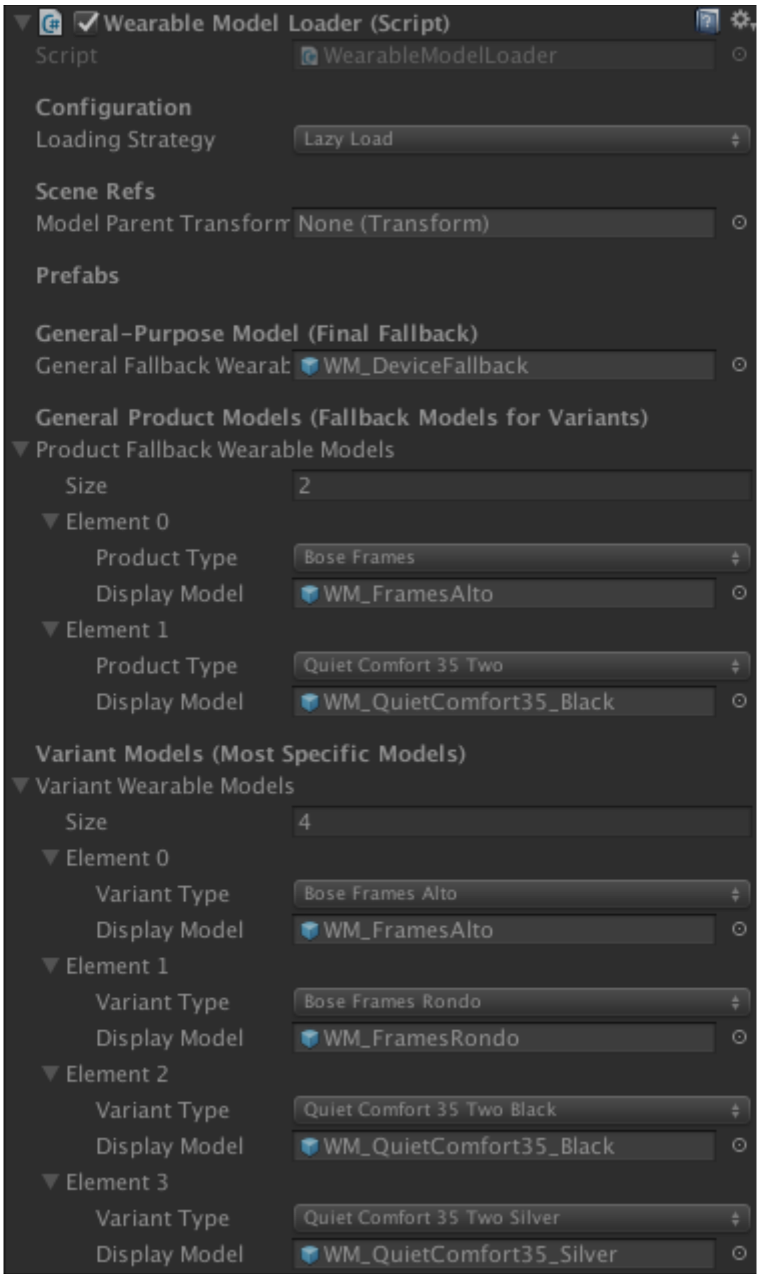
Known Issues

- The server application must be started *before* running your application in-editor, or the SDK will not be able to connect.
- The server (mobile device) and client (computer running the editor) must be on the same network.

- After halting play in the editor with a device still connected, subsequent plays will not register the device in the connection panel. To fix this, stop and start the server app between plays in the editor.

Display the connected device at runtime.

You can load a 3D model of the Bose AR device a user has connected to your app by using the "WearableModelLoader" Prefab. To ensure you always have runtime models for every supported Bose AR device, simply keep this SDK up-to-date.



Wearable Model Loader (Script)
Script: WearableModelLoader

Configuration
Loading Strategy: Lazy Load

Scene Refs
Model Parent Transform: None (Transform)

Prefabs

General-Purpose Model (Final Fallback)
General Fallback Wearab: WM_DeviceFallback

General Product Models (Fallback Models for Variants)
Product Fallback Wearable Models
Size: 2
Element 0
Product Type: Bose Frames
Display Model: WM_FramesAlto
Element 1
Product Type: Quiet Comfort 35 Two
Display Model: WM_QuietComfort35_Black

Variant Models (Most Specific Models)
Variant Wearable Models
Size: 4
Element 0
Variant Type: Bose Frames Alto
Display Model: WM_FramesAlto
Element 1
Variant Type: Bose Frames Rondo
Display Model: WM_FramesRondo
Element 2
Variant Type: Quiet Comfort 35 Two Black
Display Model: WM_QuietComfort35_Black
Element 3
Variant Type: Quiet Comfort 35 Two Silver
Display Model: WM_QuietComfort35_Silver

Loading Strategy

This field on the WearableModelLoader script allows you to specify how the Wearable device prefabs assigned to it are instantiated.

Lazy Load

Lazy Load will only instantiate a prefab when it has been requested and is not yet instantiated. This can help where many prefabs have been assigned or it is undesirable to instantiate them all at once. Once instantiated, it will be cached internally so that it will only be instantiated once.

All At Once

All At Once will instantiate one of every assigned prefab on the WearableModelLoader script upon Awake, cache them internally, and then disable them. This helps if you know you will have many different types of devices that will connect and are able to take on a more weighty CPU cost for instantiating them all at once that will not be experienced from then on.

Scene References

These are references that WearableModelLoader can have to objects in the scene.

Model Parent Transform

This field allows you to assign the transform the Wearable device prefabs will be parented to. If left null, this will be the WearableModelLoader's transform. This transform should be reserved for the model loader as it will attempt to disable any children of it and only show the actively connected device.

Levels of Specificity

The Wearable Model Loader provides a series of fallbacks in case the model of a connected device is not present at runtime. The following list details the fallback categories provided, from least to most specific.

General-Purpose Fallback

If a specific variant or product fallback cannot be found: a mock Bose AR model will be shown. This field must be assigned and cannot be null.

Product Fallback

The Product Fallback is meant to represent an entire product line that is used when a specific variant display model cannot be found. This array allows you to assign one prefab per Product Type.

Variant Models

A Variant Model is a specific version of a Bose AR product line. Variants may be differentiated by features or aesthetics. This array allows you to assign one prefab per Variant Type.

Troubleshooting

This page covers some common questions and/or issues we have run into. When troubleshooting an issue with the SDK, please ensure that the device you are using is running the latest available firmware.

Why do I see errors for the unsafe keyword not being allowed?

The `unsafe` keyword is used in the bridging code from C# to a native assembly. In Unity 2018, this has been introduced as a toggleable option in **Player Settings** that defaults to **False**. This causes compilation errors over the use of this keyword.

We have provided a preprocessor directive file (`mcs.rsp`) to enable the use of this keyword without altering the default Player Settings value for unsafe code. If you do not import this file or add the contents of it to an existing `mcs.rsp` file, you will need to enable the **Allow 'unsafe' Code** option in Player Settings, under "Configuration".

The connection dialog pops up, but I can't click on any of the devices that show up.

If you quickly dropped the `WearableConnectUIPanel` prefab into a scene to get started, be sure that you also added an `EventSystem` with the `StandardInputModule` on it. Without this, the prefab will not receive any input events from the device.

If you are dropping this prefab into an app that already had UI implemented, ensure that your UI is not blocking inputs for the `WearableConnectUIPanel` .

Why do the textures on the connection/demo scenes look corrupted?

There is a known issue (Case: 1085023) present in Unity 2018.1 and 2018.2 when importing atlases from 2017.x projects where certain properties are not deserialized. This has been resolved in 2018.3. If your project is locked on 2018.1 or 2018.2, you may resolve the issue with the following steps:

1. Go to your Project Window, and type in: **t:SpriteAtlas**
2. Select both **ConnectionAtlas** and **SharedAtlas**
3. In the Inspector, uncheck “Allow Rotation” and “Tight Packing”
4. File > Save Project

Release Notes

This includes release notes for all versions of the Bose Wearable SDK for Unity from v0.11.0 onward.

v0.13.0

Release Date

4/08/2019

Additions and Improvements

- Users can now access the firmware version of their device via `Device.firmwareVersion` after the device has connected. (NOTE: On the Debug, Mobile and USB providers this will always be returned as `0.0.0` and on the Device/Proxy providers the connected device's version will be returned.)

Known Issues

- Unity Cloud Builds for iOS containing this SDK will fail. As of 4/08/2019, Unity Cloud Build is still using Xcode 10.1 which is incompatible with the updated iOS libraries. This should be rectified soon by Unity.

Firmware Supported

- Frames >= 2.3.1
- QC35II >= 4.3.7

To build a demo to test the major features and functionality, simply switch your platform to a supported platform and select **Tools** > **Bose Wearable** > **Build Wearable Demo**

v0.12.3

Release Date

4/02/2019

Additions and Improvements

- Updated iOS SDK: 3.0.16
- Android: Properly handle missing location permissions in Unity 2018.3

Known Issues

- Unity Cloud Builds for iOS containing this SDK will fail. As of 4/02/2019, Unity Cloud Build is still using Xcode 10.1 which is incompatible with the updated iOS libraries. This should be rectified soon by Unity.

Firmware Supported

- Frames >= 2.3.1
- QC35II >= 4.3.7

To build a demo to test the major features and functionality, simply switch your platform to a supported platform and select **Tools** > **Bose Wearable** > **Build Wearable Demo**

v0.12.2

Release Date

3/29/2019

Additions and Improvements

- Added support for Xcode 10.2 and deprecated support for all previous versions.
- Updated iOS SDK: 3.0.15

Known Issues

- Unity Cloud Builds for iOS containing this SDK will fail. As of 3/29/2019, Unity Cloud Build is still using Xcode 10.1 which is incompatible with the updated iOS libraries. This should be rectified soon by Unity.

Firmware Supported

- Frames >= 2.3.1
- QC35II >= 4.3.7

To build a demo to test the major features and functionality, simply switch your platform to a supported platform and select **Tools** > **Bose Wearable** > **Build Wearable Demo**

v0.12.1

Release Date

3/21/2019

Additions and Improvements

- Added Android Preprocessor to enforce minimum SDK version.
- Rotation Source no longer requires a Rotation sensor to be enabled to be editable in a WearableRequirement.
- Added SDK Version and "About" Menu Item.
- Added Documentation PDF.
- Added Historical Release Notes to Documentation.
- Updated iOS SDK: 3.0.14
- Updated Android SDK: 3.0.11

Fixes

- Disabling a Gesture now sends the proper configuration change to device on iOS and Android.
- Fixed broken links in Documentation.

Firmware Supported

- Frames >= 2.3.1
- QC35II >= 4.3.7

To build a demo to test the major features and functionality, simply switch your platform to a supported platform and select **Tools** > **Bose Wearable** > **Build Wearable Demo**

v0.12.0

Release Date

3/6/2019

Additions and Improvements

- Added Android Support

Known Issues

- Before building on Android, please set your Minimum API Level to 21 in **Edit** > **Project Settings** > **Player**. This will be automatically set in an upcoming release.

Firmware Supported

- Frames >= 2.3.1

- QC35II >= 4.3.7

To build a demo to test the major features and functionality, simply switch your platform to a supported platform and select **Tools** > **Bose Wearable** > **Build Wearable Demo**

v0.11.1

Release Date

3/5/2019

Fixes

- Updated iOS Bridge (iOS SDK 3.0.12)

Firmware Supported

- Frames >= 2.3.1
- QC35II >= 4.3.7

To build a demo to test the major features and functionality, simply switch your platform to a supported platform and select **Tools** > **Bose Wearable** > **Build Wearable Demo**

v0.11.0

Release Date

2/27/2019

Additions and Improvements

- Added Gesture Support for all Platforms and Providers: DoubleTap, HeadNod and HeadShake
- Added support for choosing between 6DOF and 9DOF rotation sensors.
- Added WearableRequirement, a simple way to distribute Bose AR requirements across your application and have multiple requirements be automatically resolved to a single configuration and sent to the device.
- Added the WearableModelLoader to automatically load a 3D representation of the currently connected Bose AR device.
- Added USB Provider — providing the ability to use and control the device in the Unity Editor when connected over USB.
- Added a single-click Build for both Demo and Proxy, which will intelligently maintain your build settings.
- Added a new demo scene to demonstrate/test all implemented Gestures.

- Improved Audio/Particles in the Advanced Demo.
- Added component menus for WearableControl, WearableRequirement, RotationMatcher, and GestureDetector
- Added links to various resources in **Tools** > **Bose Wearable** > **Help**

Fixes

- Multiple fixes for the Connection Panel relating to usage and layout capabilities.
- Fixed compatibility issues for Unity 2018.3.4+
- Several bug fixes and improvements.

Firmware Supported

- Frames >= 2.3.1
- QC35II >= 4.3.7

To build a demo to test the major features and functionality, simply switch your platform to a supported platform and select **Tools** > **Bose Wearable** > **Build Wearable Demo**