Bose AR SDK for Unity

Version 0.15.0

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 (5.1+)
- iOS (11.4+)

Getting Started

Basic Guides

- Build, test and study the included demo content.
- Safely upgrade to the latest version of the SDK.

Advanced Guides

- Discover and connect to Bose AR devices.
- Ensure users can take full advantage of your experience with application intent validation.
- 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.
- Debug at runtime in the editor and on device.
- 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 5.1 'Lollipop' (API Level 22)
- **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.

Support

Our SDK is tested and verified against the same versions supported by the active Unity LTS releases, along with releases of the current version of Unity. (2017.4, 2018.4, 2019.1)

For additional SDK support, please visit the forums.

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.

On-Device

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.

In-Editor

You may also test the demos within the Editor, provided you use the Debug, USB, or Proxy Provider.

1. In order to do this, you must add the following scenes to your Build Settings:
2. Bose/Wearable/Examples/Shared/Scenes/Root.unity
3. Bose/Wearable/Examples/Shared/Scenes/MainMenu.unity
4. Bose/Wearable/Examples/ContentDemos/Basic/Scenes/BasicDemo.unity
5. Bose/Wearable/Examples/ContentDemos/Advanced/Scenes/AdvancedDemo.unity
6. Bose/Wearable/Examples/ContentDemos/Gesture/Scenes/GestureDemo.unity
7. Bose/Wearable/Examples/ContentDemos/Debug/Scenes/DebugDemo.unity
8. Open the `Root.unity` scene and edit the **Wearable Control** object to utilize one of the aforementioned providers as the **Editor Default Provider**
9. Press ▶ in the editor to begin.

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**
- **Debug 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 6-DOF rotation sensor. For more information, please see the section on the rotation sensor.

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 9-DOF rotation sensor. For more information, please see the section on the rotation sensor.

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 supported by the connected hardware. Icons for gestures supported by your device will be presented on screen.

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 both device-specific and device-agnostic gestures. To learn more about the differences between the two, please read about Gesture Data.

To trigger a gesture, perform one of following device-specific actions supported by your Bose AR device.

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.
4. **TouchAndHold:**: Touch and hold your finger on the right side of the device.

Debug Demo

Summary

This demo provides a demonstration of the included Debug Panel, which provides a high-level access to all major features and data available in the SDK.

Functionality

Interact with the tab to open the Debug Panel.

The Debug Panel enables you to inspect and override the current config and data that is being streamed from the device. To see Euler values for rotation, simply tap on the panel to toggle their view.

NOTE: You may move the tab somewhere else on the screen if it gets in your way by clicking and holding on it. After a short while, areas where you can relocate the tab will start glowing.

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` and serve as a good example of how to use its APIs for displaying a robust and engaging connection UI.

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 the device search and connection process to enable custom application flow or UI implementation. The process is state-driven based on the `ConnectionStatus` enum, which pinpoints the steps a Bose AR device must take to become connected and what information or options should be presented to a user. Updates to the ConnectionStatus are available via the `ConnectionStatusChanged` event on `WearableControl`.

```
public enum ConnectionStatus
{
    // No device is connected, and we are not searching for devices.
    Disconnected,

    // Searching for available devices.
    Searching,

    // Attempting to connect to a device.
    Connecting,
    // Waiting for secure pairing to complete (which may or
    // may not involve user intervention).
    SecurePairingRequired,

    // A firmware update is available; waiting for user to resolve.
    FirmwareUpdateAvailable,

    // A firmware update is required to connect; waiting for user to resolve.
    FirmwareUpdateRequired,

    // The device is successfully connected.
    Connected,

    // The device failed to connect.
    Failed,

    // The device connection was cancelled before it could complete.
    Cancelled
}
```

Searching for Devices

The connection process begins by searching for devices via `WearableControl.Instance.SearchForDevices` which will emit the `ConnectionStatus.Searching` state, periodically poll for new devices, and dispatch any found devices 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. Stopping a device search at any point will emit the `ConnectionStatus.Disconnected` state.

Connecting to Devices

Once a device has been selected for connection (either by a user selection or programmatically), it should be passed to `WearableControl.Instance.ConnectToDevice` which emits the `ConnectionStatus.Connecting` state and begins a series of potential checks to continue the process. These checks include:

- Secure Pairing
- App Intents
- Firmware

Secure Pairing

Some devices may require Secure Pairing (a method of pairing a device over Bluetooth authenticated by the user). if not required by the device's firmware, this check will be skipped. If Secure Pairing is required, it can result in the following states/flows:

- If Secure Pairing is required and a firmware update is needed to be able to perform it, a `ConnectionStatus.FirmwareUpgradeRequired` state is emitted. If the user opts to update, the connection process is halted and a `ConnectionStatus.Cancelled` state is emitted. Otherwise if they choose to continue without updating, a `ConnectionStatus.Failed` state is emitted and the connection is cancelled.
- If Secure Pairing is required and the current firmware is Bose AR enabled and the Bose AR device has not been securely paired to your mobile device before, a `ConnectionStatus.SecurePairingRequired` state is emitted. On iOS, the user will recieve a native prompt with instructions for secure pairing their device; if they fail to or select Cancel a `ConnectionStatus.Failed` state will be emitted, otherwise if successful the firmware check will commence. On Android, this process will continue without needing user intervention, such as a native prompt.
- If Secure Pairing is required and the current firmware is Bose AR enabled and the device has been securely paired to before, Secure Pairing will occur without needing user input and the connection process will continue to AppIntents and Firmware Checking.

App Intents and Firmware

AppIntents is a system for specifying a required device configuration including which sensors/gestures/update intervals must be available for any device to be compatible. In addition, if certain parts of a configuration are not present on a device's firmware but would be by updating to the latest version, this will direct the user that an update is required (blocking progress in the device connection until a user cancels or consents to updating). If the device's firmware is sufficient for the specified AppIntentProfile (or none is specified) and a firmware update is available, it will notify them that an update is available, but users may choose to continue without updating.

When users opt into updating their firmware on iOS or Android, they will leave the Unity application and be redirected to the appropriate AppStore location or local application on their phone (causing the Unity App to lose focus). This will halt the current connection flow, emitting a `ConnectionStatus.Cancelled`. Whether or not the user has updated their firmware or not, when they return to the Unity app they will be returned to the Device Searching screen. In the editor, the user will be immediately returned to the device search screen.

The following potential states/flows that can occur as a result of AppIntents and Firmware Checking:

- If there is no AppIntentProfile or it does not require anything, this step will be skipped.

- If the AppIntents specifies device capabilities that the firmware does not currenty support, but a firmware update would support them, a `ConnectionStatus.FirmwareUpgradeRequired` state is emitted. If the user opts to update, the connection process is halted and a `ConnectionStatus.Cancelled` state is emitted. Otherwise if they choose to continue without updating, a `ConnectionStatus.Failed` state is emitted.
- If the AppIntents specifies device capabilities that are supported by the current firmware and a firmware update is available, the connection process will continue to the next step below.
- Whether or not a firmware update is available (Conveyed by the `ConnectionStatus.FirmwareUpgradeAvailable` state). This check can result in several states.
  - If no update is available, this step is skipped and the connection process continues.
  - If an update is available and a user opts to update, the connection process is halted and a `ConnectionStatus.Cancelled` state is emitted.
  - If an update is available and a user opts to continue without it, the connection process continues.

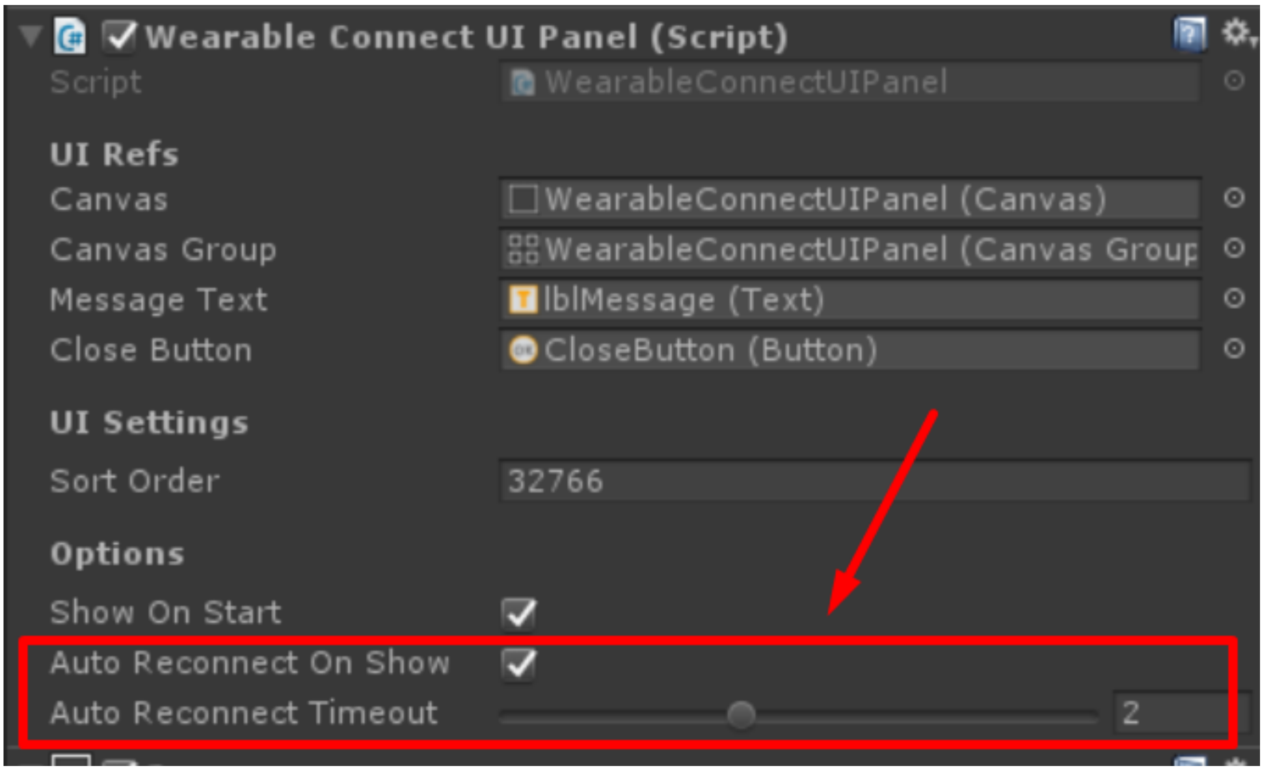
Connected to a Device

If a device passes successfully both the Secure Pairing, AppIntent, and Firmware checking phases, a `ConnectionStatus.Succeeded` state is emitted and the device will have connected successfully. Anytime a device is connected to, a session is created; this persists for the entire time it remains connected. This session can end either automatically when the device has become disconnected or explicitly by a user when `WearableControl.Instance.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. A `ConnectionStatus.Succeeded` or `ConnectionStatus.Disconnected` state can also also be listened for via the `ConnectionStatusChanged` event on `WearableControl`.

Auto-Reconnect

Auto-Reconnect is a feature that attempts to reconnect a user to a previously connected device when the connection UI is launched. When a device successfully connects, its unique identifier is saved for that application. Once this unique identifier is saved and if Auto-Reconnect is on, when the connection UI is shown a reconnect attempt will be made. This will result in a `ConnectionStatus.AutoReconnect` state being emitted rather than `ConnectionStatus.Searching`. The device connection screen will be shown rather than the device search screen. During this time, if the last connected to device is found an automatic connection attempt will be made to it without requiring user intervention. If it is not found during the timeout period, a `ConnectionStatus.Searching` state will be emitted and the user will be returned to the device search screen.

This feature is turned on by default, but is configurable by a developer on the `WearableConnectUIPanel` prefab script. The options include:
`Auto Reconnect on Show` : When set to true, an auto reconnect attempt will be made when the connection UI is shown and a previously-connected device identifier has been saved. If false, the user will be directed immediately to the device search screen.
`Auto Reconnect Timeout` : This controls the amount of time in seconds that a auto-reconnect attempt will be made before returning to the device search screen.



Wearable Connect UI Panel (Script)

Script	WearableConnectUIPanel
UI Refs	
Canvas	WearableConnectUIPanel (Canvas)
Canvas Group	WearableConnectUIPanel (Canvas Group
Message Text	lblMessage (Text)
Close Button	CloseButton (Button)
UI Settings	
Sort Order	32766
Options	
Show On Start	✓
Auto Reconnect On Show	✓
Auto Reconnect Timeout	2

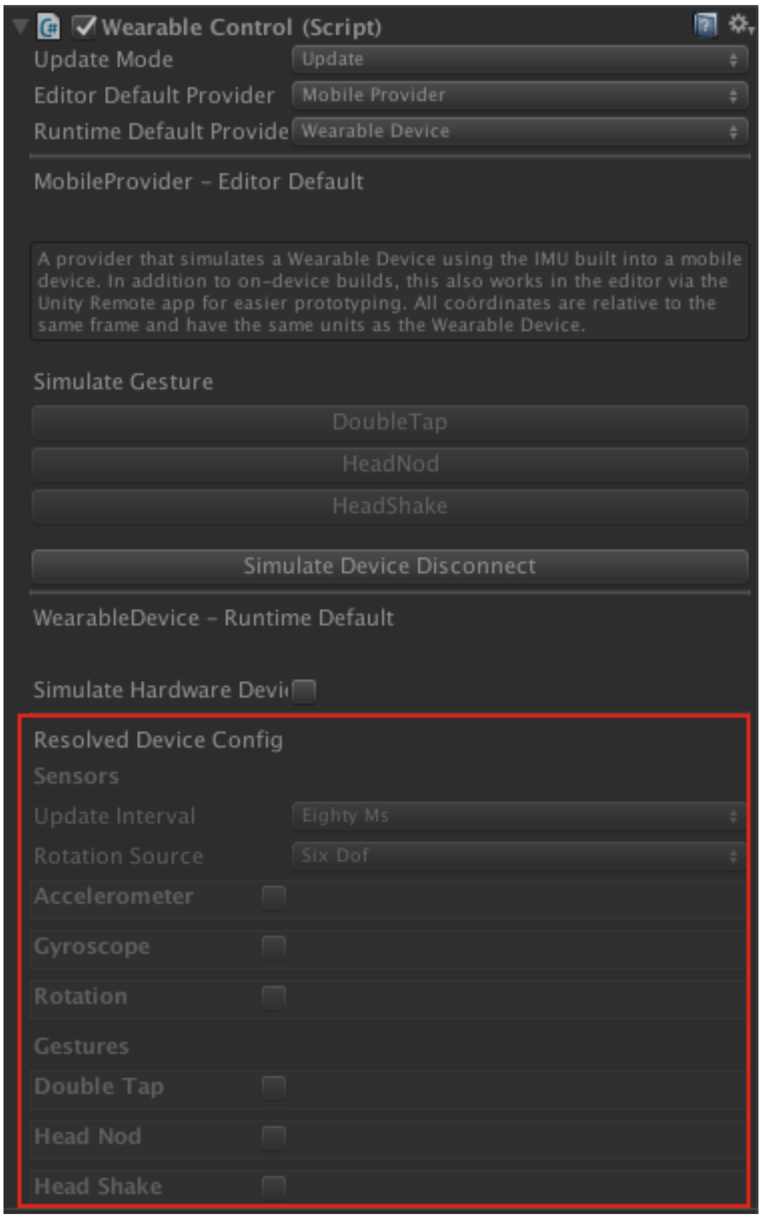
Device Configuration with WearableRequirement

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: ☐

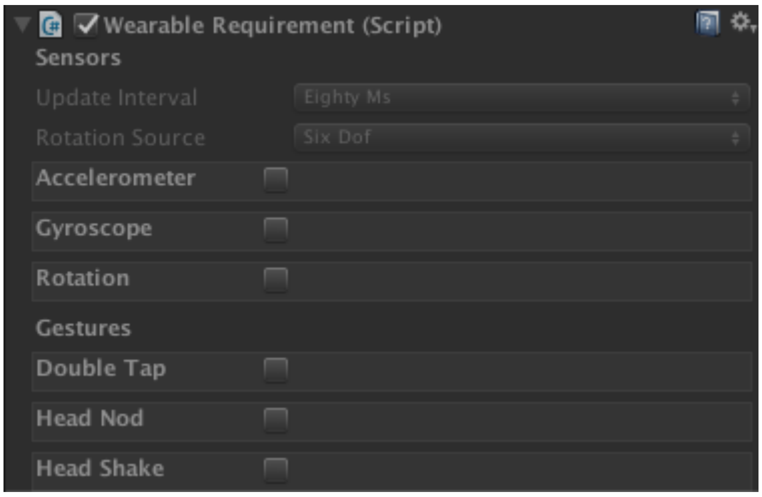
Gestures

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 four 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.
- **RotationSixDof:** Emits a `Quaternion` indicating the device's orientation in space.
- **RotationNineDof:** Similar to **RotationSixDof**, but incorporates the onboard magnetometer.

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`.

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

Bose AR devices can detect gestures as a means of input, so users may convey meanings like *affirmative* or *negative* to your application. Different devices may use different gestures for the same meaning.

These meanings are called **device-agnostic gestures**. For each device-agnostic gesture, a particular Bose AR device will have a **device-specific gesture**, like *double-tap* or *head nod*, that maps to it. An application may use device-specific gestures, but their use is discouraged, as they may not be universally available across all devices. To put it another way, every device is guaranteed to support all device-agnostic gestures, but not every device is guaranteed to support all device-specific gestures.

This table summarizes the mappings from device-specific gestures to device-agnostic gestures, by device:

Device-agnostic gesture	Frames	QuietComfort 35 II	Noise Cancelling Headphones 700
Input	Double-Tap	Double-Tap	Touch and Hold
Affirmative	Head Nod	Head Nod	Head Nod
Negative	Head Shake	Head Shake	Head Shake

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.InputGesture`).
- 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.AffirmativeGestureDetected`
- 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.
- 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.
- **6-DOF** uses only the gyroscope and accelerometer to determine the device's orientation relative to the position it was in when powered on.
- **9-DOF** 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.
- Each rotation sensor source is available as an individual sensor.
- When magnetic heading is not needed, prefer the 6-DOF option; 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.

Given that the USB Provider is not transmitting data over-the-air, data transmission is likely to be more reliable than users will experience over Bluetooth. Always test your experience on device before releasing your application to users.

Currently, the following Bose AR devices support the USB Provider:

- Bose Frames
- Bose QuietComfort 35 II

USB Audio Support

If your Bose AR device supports USB Audio, the device should enumerate as an audio output device on your operating system shortly after connecting to the device at runtime.

Currently, the following Bose AR devices support USB Audio:

- 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.

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.

Additionally, two "Simulation Modes" are available to augment the Debug Provider and enable you to test and iterate on the API based on realtime data.

- **Constant Rate** - Set a constant spin rate of the virtual device. (Note: Values reported by the provider are representative of what a real device would be reporting if it was physically rotating at a constant speed.)
- **Mobile Device** - Use a a mobile device's IMU to simulate movement in place of your own device, this is possible:
    - **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.
    - **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.

Provider Compatibility

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

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

Application Intent Validation

In order to help developers ensure that users' devices support the functionality required for their applications, the SDK provides a feature called Intent Validation. Your application's intent — the set of sensors, update intervals, and gestures it may use over the course of operation — can be sent to the Bose AR SDK and validated.

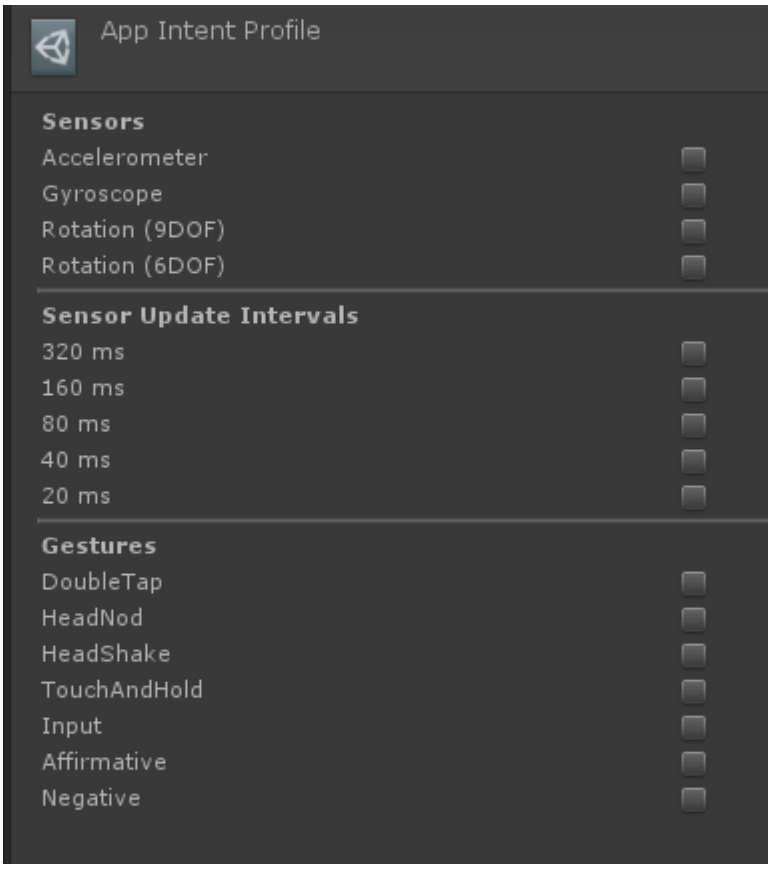
The SDK will process the App Intent Profile and determine if the connected device's hardware and firmware can support the requested configuration, and return this result back to Unity. In the event that a connected Bose AR device cannot fulfill the specified intents, users can be instructed to update their firmware or connect a device with the supported functionality.

App Intent Profiles

An App Intent Profile is used to communicate an app's desired sensors, sensor intervals, and gestures to the underlying SDK for evaluation. At its most fundamental, an App Intent Profile is a set of `SensorId`s, `SensorUpdateInterval`s, and `GestureId`s.

To create an App Intent Profile, navigate to `Assets -> Create -> Bose Wearable -> App Intent Profile`. This will create an intent profile asset in your project, which can be edited in the Unity inspector.

While we primarily expect developers to create profiles using the inspector, functionality also exists to create and modify them at runtime; see `AppIntentProfile.cs` for the relevant methods.


App Intent Profile

Sensors

Accelerometer	☐
Gyroscope	☐
Rotation (9DOF)	☐
Rotation (6DOF)	☐

Sensor Update Intervals

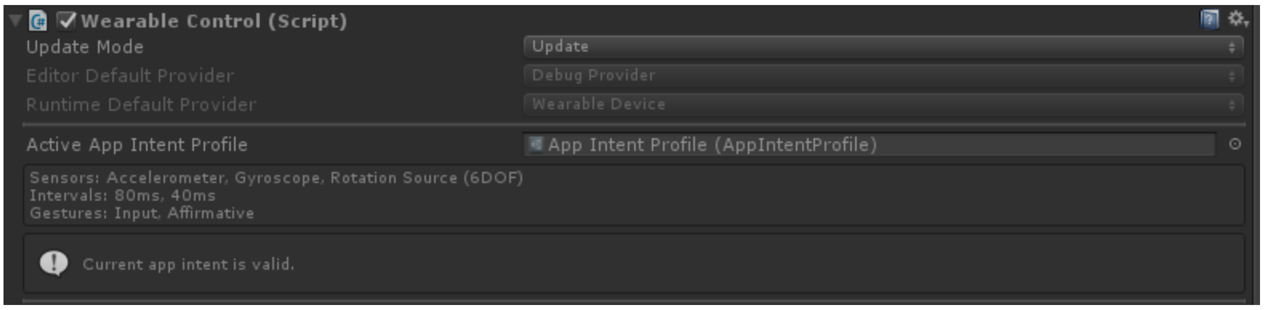
320 ms	☐
160 ms	☐
80 ms	☐
40 ms	☐
20 ms	☐

Gestures

DoubleTap	☐
HeadNod	☐
HeadShake	☐
TouchAndHold	☐
Input	☐
Affirmative	☐
Negative	☐

Validating An Intent Profile

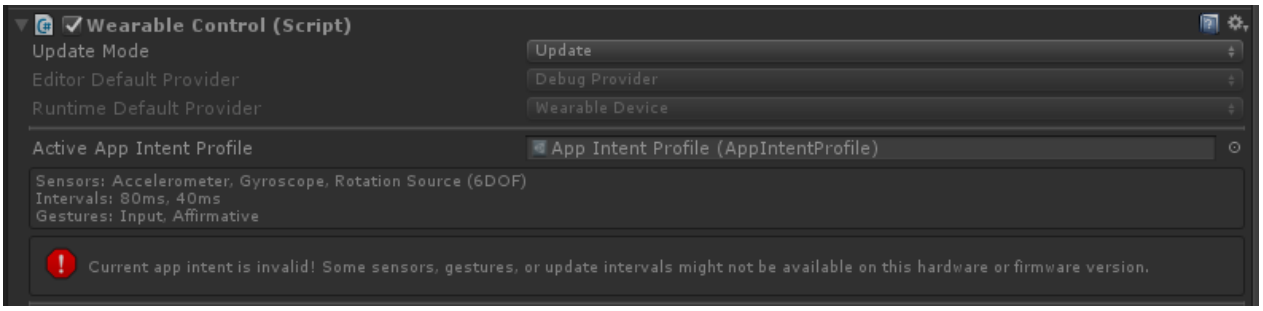
Once an intent profile has been created, it must be validated. By assigning an active intent profile in Wearable Control's inspector panel, the plugin will automatically be validated when a new device connects, and the result shown. User-subscribable events are also provided on `WearableControl` that are invoked whenever validation succeeds or fails; for example, your application might pop up a warning telling users that their device might not support all of its required features.


Wearable Control (Script)

Update Mode	Update
Editor Default Provider	Debug Provider
Runtime Default Provider	Wearable Device
Active App Intent Profile	App Intent Profile (AppIntentProfile)

```
Sensors: Accelerometer, Gyroscope, Rotation Source (6DOF)
Intervals: 80ms, 40ms
Gestures: Input, Affirmative
```

Current app intent is valid.


Wearable Control (Script)

Update Mode: Update

Editor Default Provider: Debug Provider

Runtime Default Provider: Wearable Device

Active App Intent Profile: App Intent Profile (AppIntentProfile)

Sensors: Accelerometer, Gyroscope, Rotation Source (6DOF)
Intervals: 80ms, 40ms
Gestures: Input, Affirmative

Current app intent is invalid! Some sensors, gestures, or update intervals might not be available on this hardware or firmware version.

If the profile is switched or modified at runtime, an option will become available to re-validate it in the inspector.

As before, methods are available to set, clear, and re-validate intent profiles at runtime. See `WearableControl.cs` for the relevant methods.

Usage Notes

When to validate intents

Validating intents can be costly, and incurs some delay. Therefore, it is advantageous to perform it as few times as possible. It is not necessary to send a new profile when switching a sensor/gesture on or off, or changing intervals; there is no penalty or commitment associated with adding a value to an intent profile, so it's best to include any possible sensors/intervals/gestures that might get used. This way, `WearableControl` can validate any sensors/intervals/gestures upfront on device connection without needing to re-validate at a later time.

Violating intents

As mentioned above, sending an intent for validation is not a contract; at any time, sensors/gestures can be enabled or disabled, or intervals changed, regardless of the active intent profile.

If a configuration is provided that is at odds with the active intent profile (for example, the intent profile states that you intend to use the accelerometer and gyroscope, and you enable the rotation sensor), a warning will be generated, but the configuration attempt will otherwise proceed as normal. That being said, it is in your best interests to provide accurate application intent profiles so they can preemptively catch problems before they occur with your users.

Provider-Specific Implementations

Device Provider (iOS and Android)

These providers use the underlying native SDK's validation features.

NOTE: This provider is the *only* provider where intent validation will inform a user whether or not a firmware update will enable a device to succesfully connect to your application.

USB and Debug Provider

These providers emulate the native SDK's intent validation function by comparing against the available sensors and gestures of the connected device.

The configurability of the Debug Provider offers an easy way to test your intent validation logic: simply remove a sensor or gesture from the virtual device, which should cause any intent profile that depends on it to fail.

Mobile and Proxy Provider

Currently, intent validation will always succeed regardless of the app intent profile or device configuration on these providers.

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.

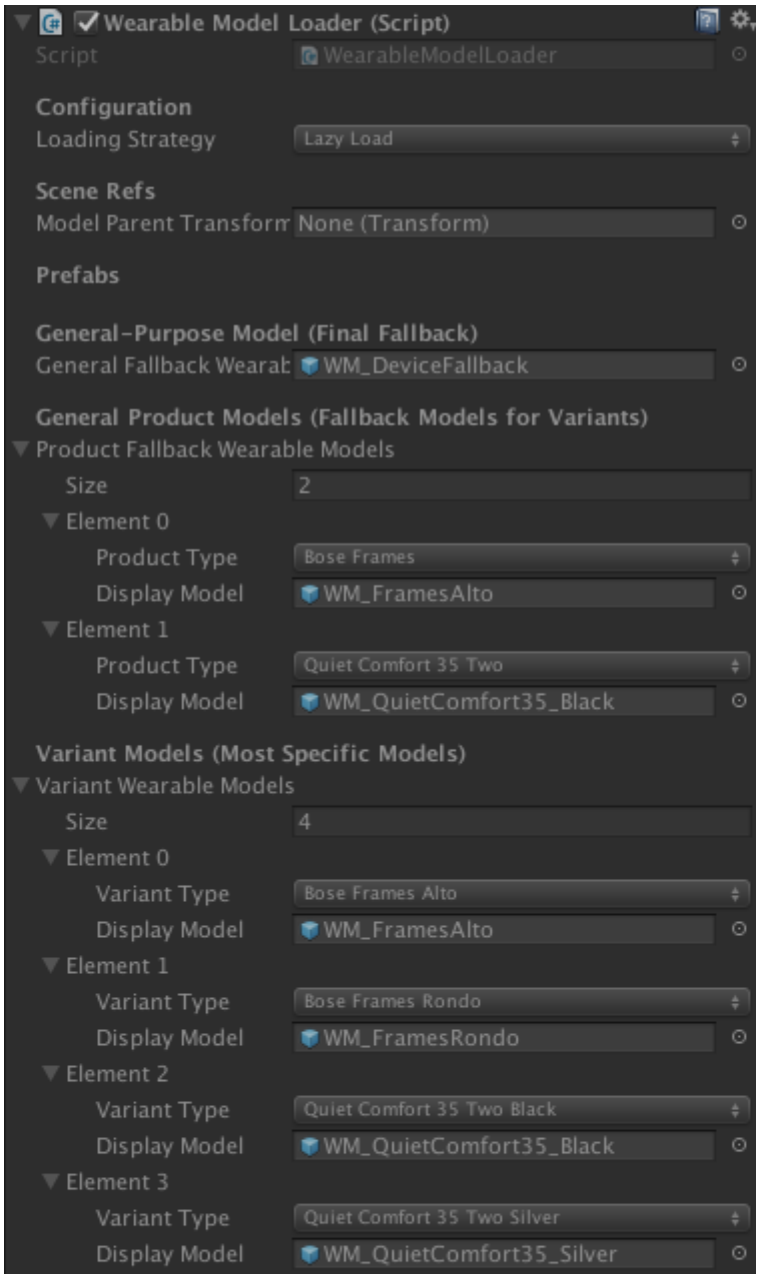
Known Issues

- 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.

Debugging the SDK at runtime.

In an effort to make the current state, data, and performance of the SDK easily accessible, the "DebugUIPanel" prefab has been provided as a standalone tool.

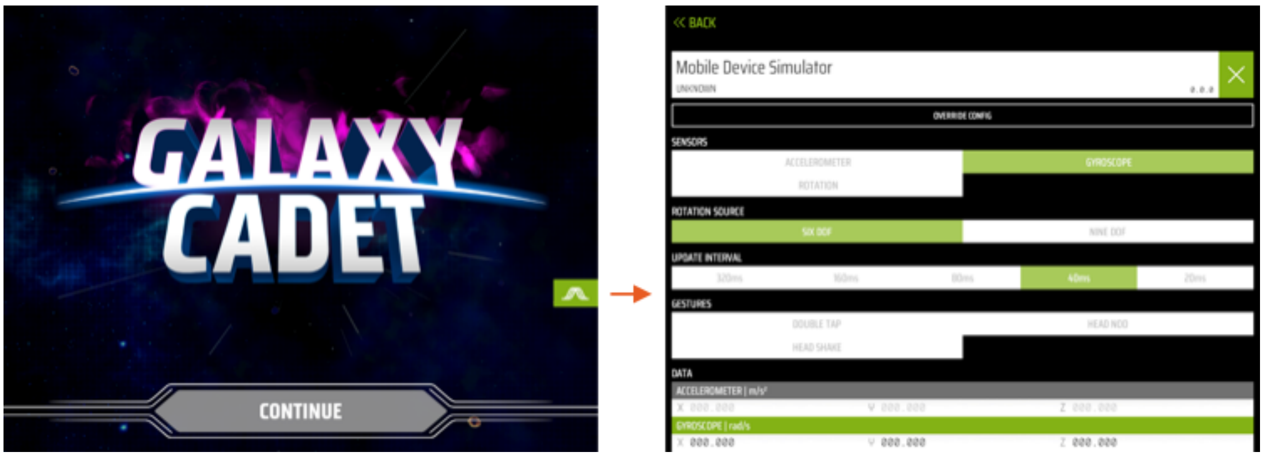
This tool functions on all supported platforms with all providers; both in-editor and on-device.

Setting up the Prefab

To add the tool to your project, simply drop the `Bose/Wearable/Connection/Prefabs/DebugUIPanel` prefab into a scene.

NOTE: If this is a brand new project, be sure to add an `EventSystem` with the `Standard Input Module` into your scene too -- otherwise pointer events will not be detected.

When the DebugUIPanel is closed, a small button will be docked on the edge of screen. To open it, click/tap on the button.

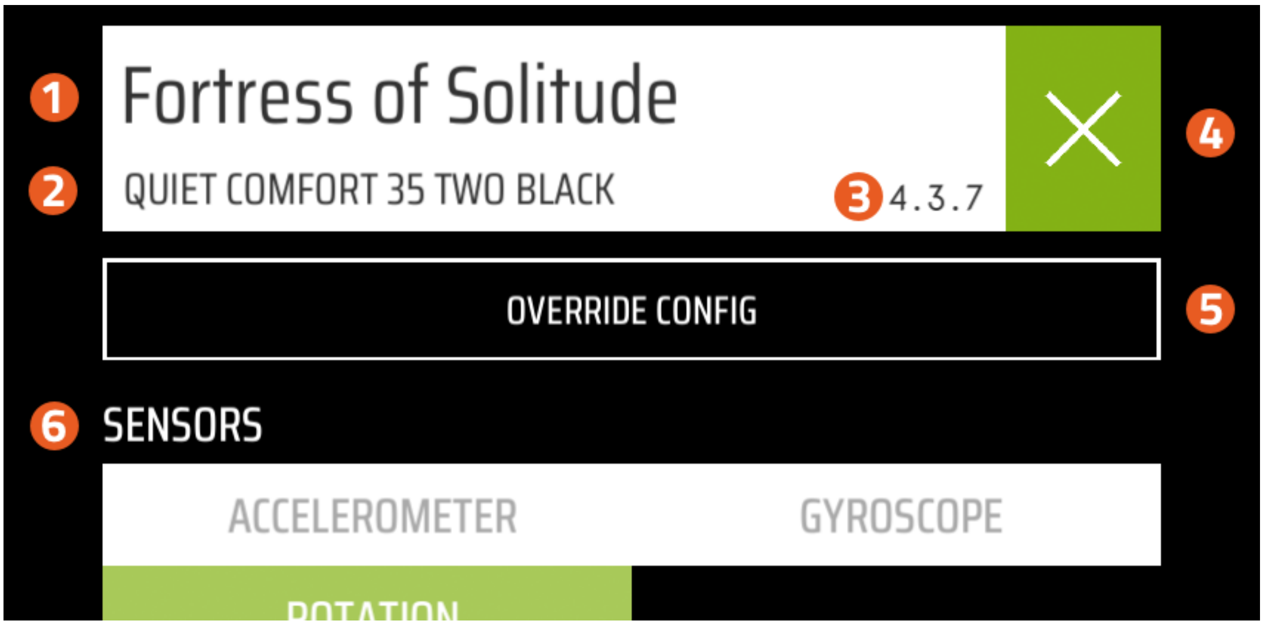


If the button to open the DebugUIPanel is in the way at runtime, you may move it to a more convienent location. Press and hold on the button and drag it anywhere within the animated glowing area that appears.

Using the Prefab

The panel splits information up into three different sections:
Device: *The current state of a connected device as configured by runtime applications.*
Data: The data being reported by the active provider via the SDK.
* **Timing:** A collection of advanced metrics to determine the overall fitness and performance of the SDK.

Debug: Device



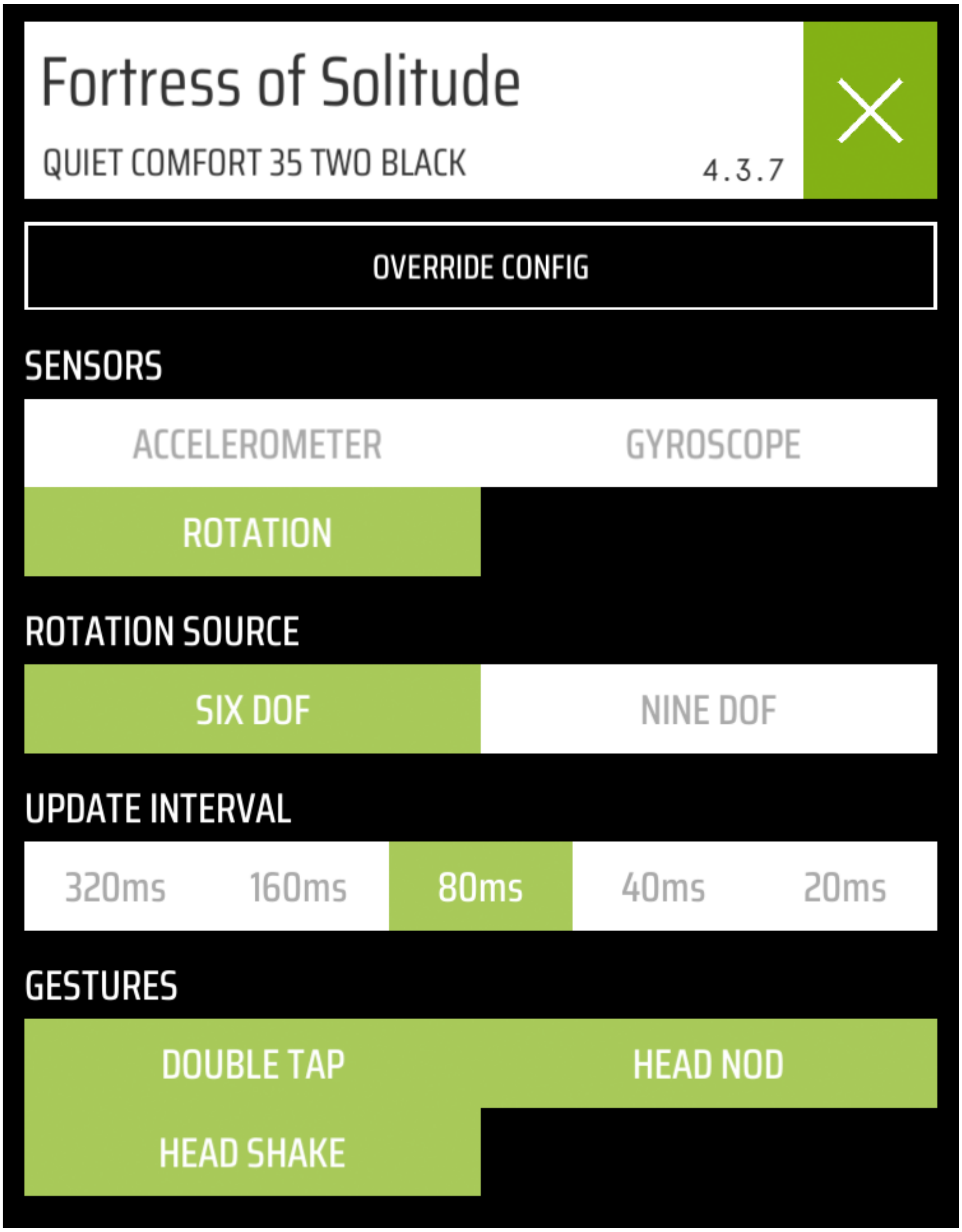
When a device is connected, you will be presented with:

1) The current name of the device.
2) The make and model of the Bose AR device.
3) The current firmware version of the device.

4) The ability to disconnect or search for and connect new devices, depending on the flow developed in your application.
5) The ability to override the device configuration.
6) The current configuration of the device.

Viewing and overriding the configuration.

When opened for the first time, the debug panel will show a series of disabled controls that represent the current device configuration. (To learn more about configuring your device at runtime, please see Device Configuration.)



Fortress of Solitude

QUIET COMFORT 35 TWO BLACK 4.3.7

OVERRIDE CONFIG

SENSORS

ACCELEROMETER GYROSCOPE [✓ ROTATION]

ROTATION SOURCE

[✓ SIX DOF] NINE DOF

UPDATE INTERVAL

320ms 160ms [✓ 80ms] 40ms 20ms

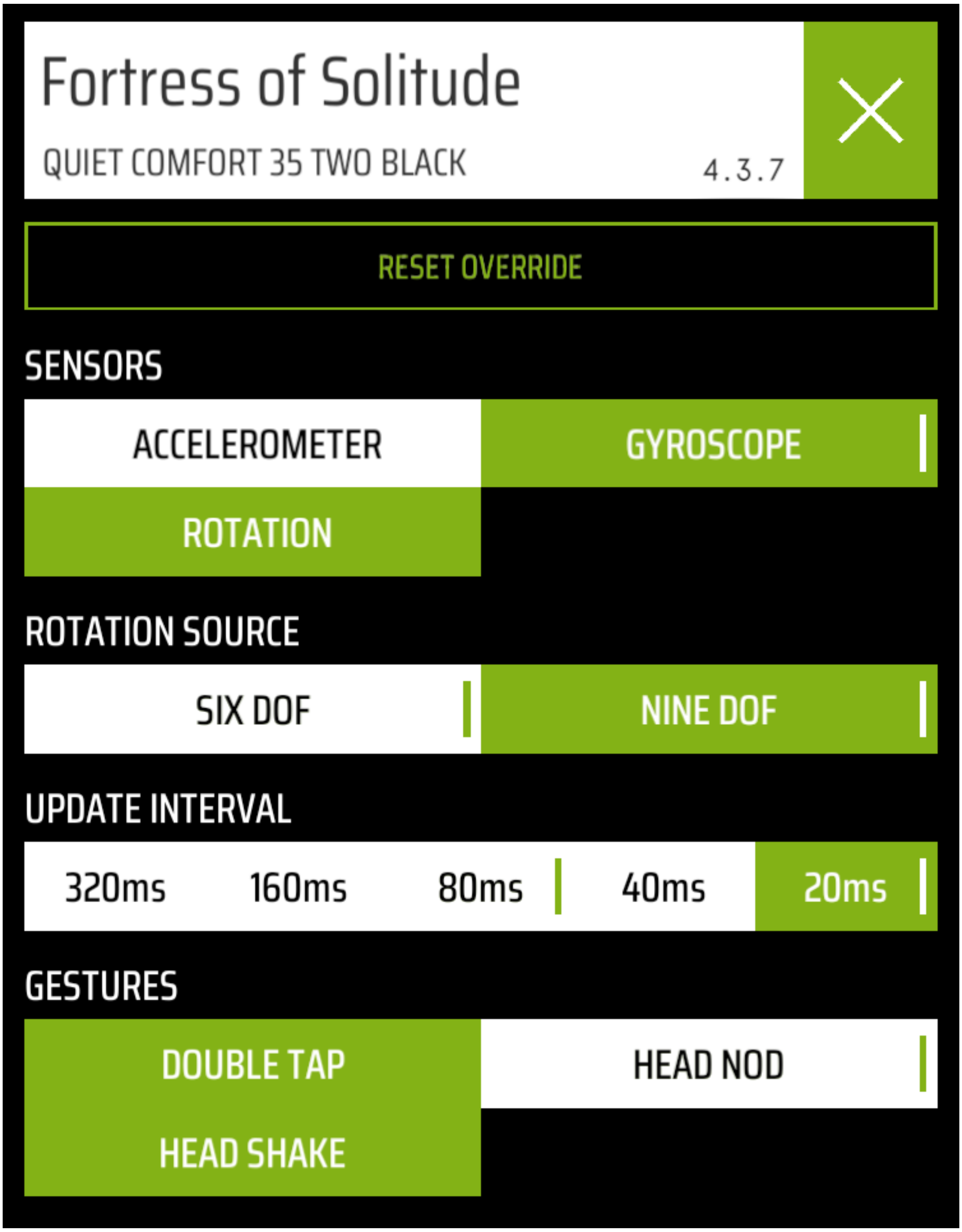
GESTURES

[✓ DOUBLE TAP] [✓ HEAD NOD] [✓ HEAD SHAKE]

In order to manipulate the device config freely, you must initiate an override of the config by selecting the `OVERRIDE CONFIG` button.

Once the device config has been overridden: the controls will become enabled. Any new configuration defined by the override will be denoted by a rectangular 'pip' and take immediate priority over any underlying configuration from your application. This includes:

- Turning sensors or gestures on or off.
- Setting a new Update Interval.



Fortress of Solitude

QUIET COMFORT 35 TWO BLACK 4.3.7

RESET OVERRIDE

SENSORS

ACCELEROMETER | GYROSCOPE | ROTATION

ROTATION SOURCE

SIX DOF | NINE DOF

UPDATE INTERVAL

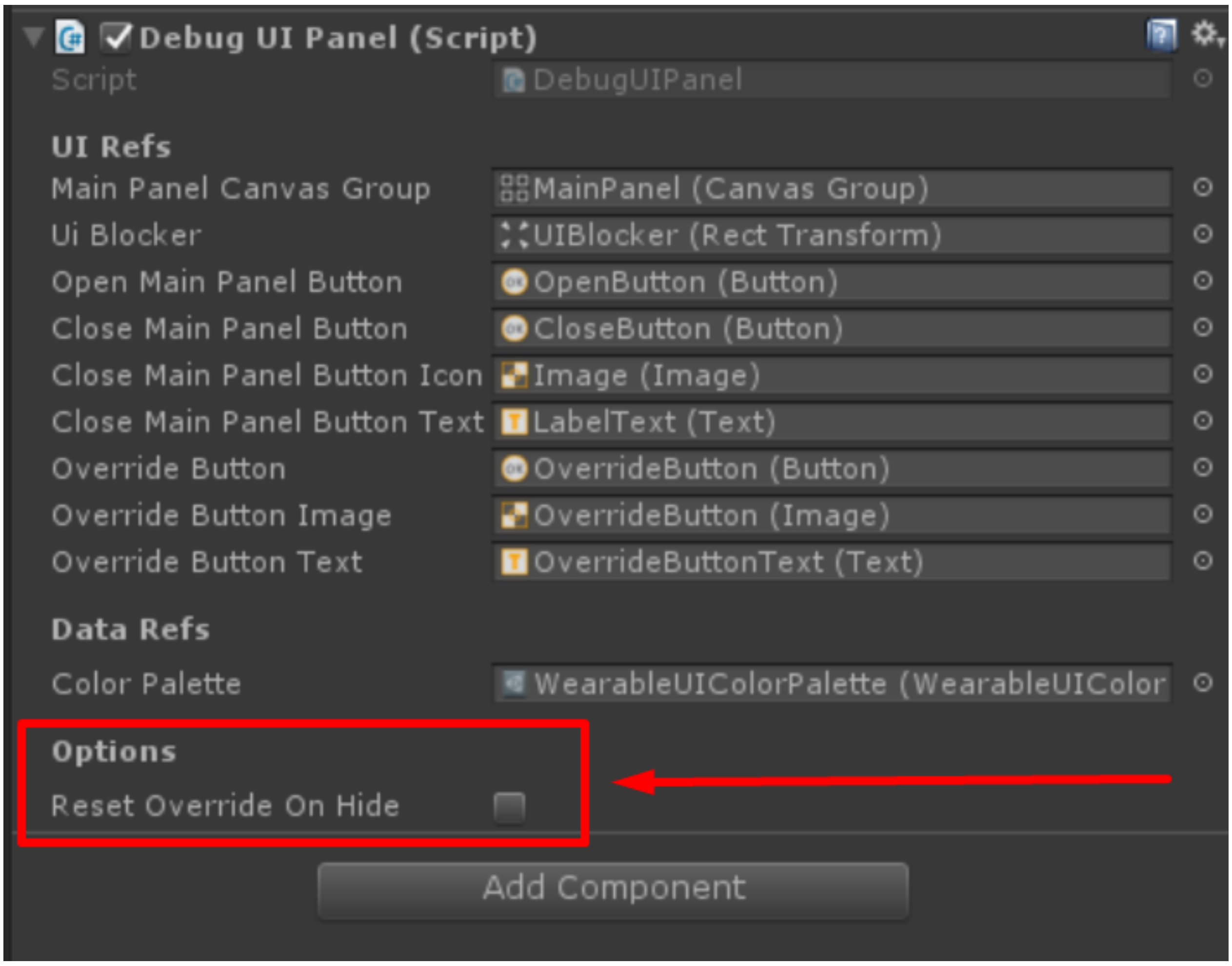
320ms | 160ms | 80ms | 40ms | 20ms

GESTURES

DOUBLE TAP | HEAD NOD | HEAD SHAKE

The override will persist until a user clicks the `RESET OVERRIDE` button on the debug panel.

If you would like to ensure that an override never persists when returning to your application, you can check `Reset Override on Hide` on the "DebugUIPanel" Prefab to automatically clear the configuration when the debug panel is closed.

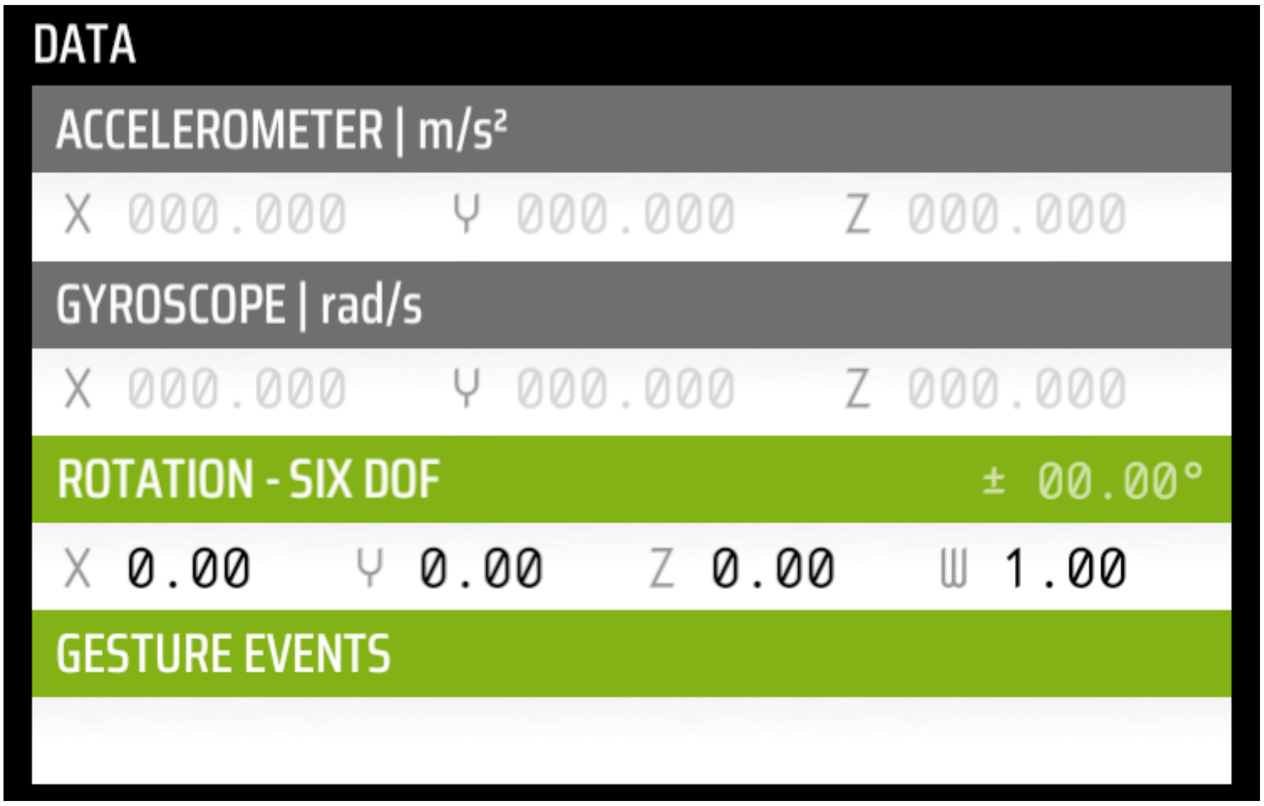

Debug UI Panel (Script)

Script	DebugUIPanel
UI Refs	
Main Panel Canvas Group	MainPanel (Canvas Group)
Ui Blocker	UIBlocker (Rect Transform)
Open Main Panel Button	OpenButton (Button)
Close Main Panel Button	CloseButton (Button)
Close Main Panel Button Icon	Image (Image)
Close Main Panel Button Text	LabelText (Text)
Override Button	OverrideButton (Button)
Override Button Image	OverrideButton (Image)
Override Button Text	OverrideButtonText (Text)
Data Refs	
Color Palette	WearableUIColorPalette (WearableUIColor
Options	
Reset Override On Hide	☐

Debug: Data

The Data section allows a user to view the last `SensorFrame` data received from the available sensors and gestures.

- Sensors or Gestures that are turned off or unavailable will be colored gray and faded.
- Rotation data may be viewed as a Quaternion (x,y,z,w) or Euler (x,y,z); tap/click on the sensor data to toggle between the modes.
- The 9-DOF rotation sensor will display its measurement uncertainty.
- Gesture Events will temporarily display the last detected gesture as reported by the active provider.


DATA

ACCELEROMETER | m/s²

X 000.000 Y 000.000 Z 000.000

GYROSCOPE | rad/s

X 000.000 Y 000.000 Z 000.000

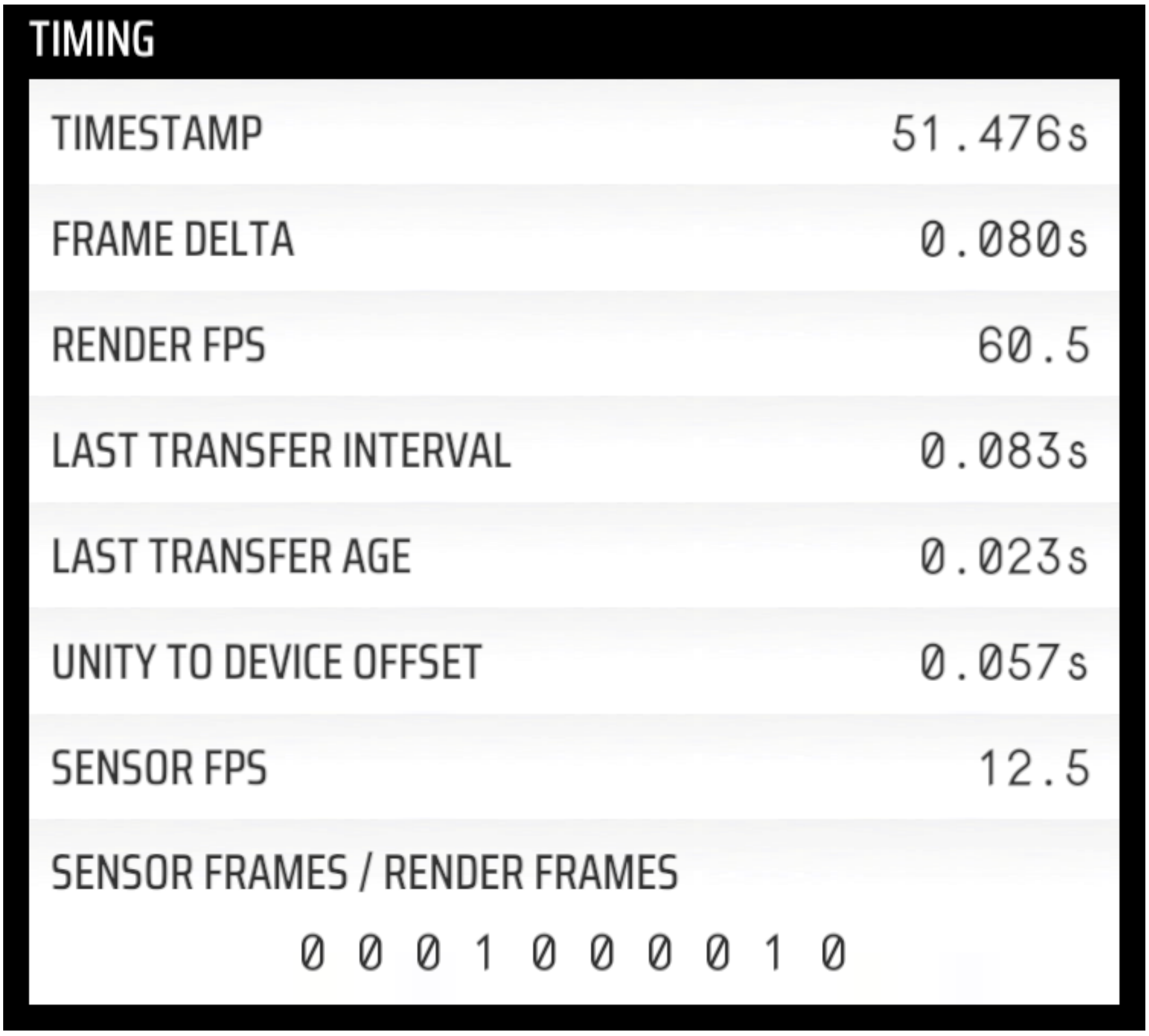
ROTATION - SIX DOF ± 00.00°

X 0.00 Y 0.00 Z 0.00 W 1.00

GESTURE EVENTS

Debug: Timing

The timing section provides a collection of advanced metrics about the data coming from a device and the current runtime session. This section is primarily used to validate the performance of our SDK within your application, and can assist in debugging any performance issues that may arise.



TIMING

TIMESTAMP	51.476s
FRAME DELTA	0.080s
RENDER FPS	60.5
LAST TRANSFER INTERVAL	0.083s
LAST TRANSFER AGE	0.023s
UNITY TO DEVICE OFFSET	0.057s
SENSOR FPS	12.5

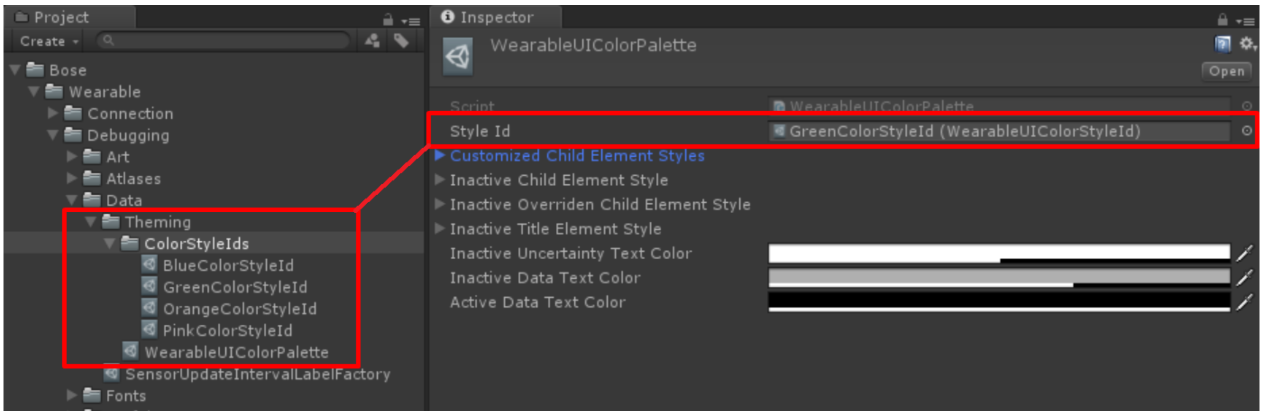
SENSOR FRAMES / RENDER FRAMES

0 0 0 1 0 0 0 0 1 0

- **Timestamp:** The last `SensorFrame` timestamp received by the device.
- **Frame Delta:** The age in seconds of the most recently-received `SensorFrame` data.
- **Render Frames Per Second:** The number of Unity render frames per second.
- **Last Transfer Interval:** The time between the most recent `SensorFrame` transfer and the prior transfer, in seconds, or null if no data has been received.
- **Last Transfer Age:** The age in seconds of the most recently-received `SensorFrame`.
- **Unity to Device Offset:** The difference between the the last `SensorFrame` timestamp received by the device and `Time.unscaledTime` when it was received, or null if no data has been received.
- **Sensor Frames Per Second:** This display allows users to get a broad overview of the device's transmission pattern in order to understand how data is sent over time. It displays a ten render frame history organized in a ring-buffer that is aged out over time.

Customization

The color theme of the `DebugUIPanel` prefab can be customized by swapping out the `UIColorStyleId` on the `WearableColorUIPalette` asset. This can be done in the Unity Editor via the inspector.



```
Bose
  Wearable
    Connection
    Debugging
      Art
      Atlases
      Data
        Theming
          ColorStyleIds
            BlueColorStyleId
            GreenColorStyleId
            OrangeColorStyleId
            PinkColorStyleId
          WearableUIColorPalette
        SensorUpdateIntervalLabelFactory
      Fonts
```

WearableUIColorPalette

Script	WearableUIColorPalette
Style Id	GreenColorStyleId (WearableUIColorStyleId)
Customized Child Element Styles	
Inactive Child Element Style	
Inactive Overriden Child Element Style	
Inactive Title Element Style	
Inactive Uncertainty Text Color	
Inactive Data Text Color	
Active Data Text Color	

Upgrading to a new version of the SDK

Prior to upgrading, it is **always encouraged** that you make a backup of your work.

To ensure a clean install of the SDK, please perform the following steps:

1. Open a new scene in Unity. (File > New Scene)
2. Delete all *Bose AR SDK for Unity* files in your Unity Project, including the:
    - `Bose/Wearable` directory.
    - `Plugins/iOS/BoseWearable` directory.
    - `Plugins/MacOS/BoseWearableUSBBridge.bundle` directory.
    - `Plugins/Windows/BoseWearableUSBBridge.dll` file.
    - `Plugins/Android/blecore.aar` file.
    - `Plugins/Android/bosewearable.aar` file.
    - `Plugins/Android/BoseWearableBridge.aar` file.
    - `Plugins/Android/localbroadcastmanager-1.0.0.aar` file.
3. Import the new SDK.

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/debug panel pops up, but I can't click on any of the devices that show up.

If you quickly dropped the `WearableConnectUIPanel` or `DebugUIPanel` 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 panel.

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.

v0.15.0

Release Date

6/24/2019

When upgrading, please be sure to follow our Upgrade Guide.

API Changes

- Several areas in the API have been marked deprecated. Please update your usage of the API with the recommended changes from these deprecation messages, as these elements will be removed in a following update.
- WearableConnectUIPanel's Show() method has had all parameters removed. Please subscribe to WearableConnectUIPanel's `Closed` event to receive a notification when the panel is closed.
- Removed RotationSource. To switch between the two, please use the 6-DOF or 9-DOF rotation sensors.

Additions and Improvements

- Support for Unity 2018.4 and 2019.1 has been verified. Going forward, we will now mirror Unity's LTS Support, supporting 2017.4, 2018.4, and all 2019.x releases.
- App Intents are now validated during the connection process for iOS and Android devices, as well as the USB and Debug providers. On supported platforms (iOS and Android), the user will be informed that a firmware update is required to satisfy the requested intents. On the Debug and USB providers, App Intent check failures will result in a failed connection attempt.
- Added Device-Agnostic Gestures: Input, Affirmative, Negative
- Added Device-Specific Gesture: TouchAndHold
- Added a new Debug Panel to provide high-level access to the SDK and an invaluable tool during development.
- Added Product Variant: QC35II - Rose Gold
- Added Product Support: Noise Cancelling Headphones 700.
- Sensor and Gesture Availability is now exposed from underlying Native SDKs.

- Sensor and Gesture Events are now exposed from underlying Native SDKs.
- Sensor Service Suspension Events are implemented for iOS, Android, USB, and Debug providers. These events are invoked when the device will not report any sensor/gesture data.
- Several improvements to the Connection Flow / Connection Panel:
  - The connection flow is entirely driven by a new `ConnectionStatus` enum, giving much more granular control and notification on the state that the Bose AR device is in when connecting.
  - Updated Connection Panel to handle new Secure Connection, Firmware Upgrade, and App Intent user flows.
  - Added a `Close` event on the Connection Panel to inform of a successful connection or user cancellation.
  - Added the ability to close the connection panel at supported points in the connection process.
  - Added configuration options to control when the Connection Panel shows and hides.
  - Added auto-reconnect functionality for the previously connected device. (Default: on)
  - Removed serial number filtering on USB Provider.
- Several updates have been made to the Demo App:
  - Gesture Demo has been updated to show all available gestures from the connected device, including new agnostic gestures.
  - A Debug Demo has been added to showcase the new Debug Panel.
  - Demos now provide both a list of features and instructions on how to use them in a new "Info" panel.
  - The main menu now lists the SDK and Unity versions. Tap to toggle between the two.
  - Demos handle device disconnection more gracefully.
- Documentation added or updated for all major new features.
- Support multiple gesture detections per-frame.
- The functionality of the Mobile Provider has been folded into the Debug Provider. See "Simulated Movement" on the Debug Provider inspector for more information.
- Obsolescence warnings have been added to items that will be removed in the next public release.

Fixes

- Gesture availability now returns proper values for iOS and Android.
- Devices now update their information (such as Signal Strength) during the search process in the Connection Panel.
- Several small fixes to the demos have been made.
- iOS no longer retries indefinitely when a configuration fails due to a sensor suspension.

Known Issues

- The USB Provider is not currently supported on NC700s.
- The Proxy Provider will not discover nor establish a connection to a device, and should not be considered as a viable provider.

Firmware Supported

- Frames >= 2.3.1
- QC35II >= 4.3.7
- NC700 >= 1.0.10

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**

v0.14.2

Release Date

6/20/2019

INTERNAL RELEASE

v0.14.1

Release Date

6/7/2019

INTERNAL RELEASE

v0.14.0

Release Date

5/23/2019

INTERNAL RELEASE

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**