Bose AR SDK for Unity

Version 4.1.1

Contents

- v4.0.0
- v0.16.0
- v0.15.0
- v0.13.0
- v0.12.3
- v0.12.2
- v0.12.1
- v0.12.0
- v0.11.1
- v0.11.0

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 (12.0+)

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.
- Handle Sensor Service Events for the best user experience.
- Display the connected device model at runtime.
- Debug at runtime in the editor and on device.
- Locale options allow for localizing content to different languages.
- Configure Editor Preferences for per-developer settings.

- 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:** 2018.4.0f1 or greater

Platform: Android

- **Android Studio:** 3.2 or greater

Your Unity project must have the following settings enabled:

- **Minimum API Level[1]:** Android 8.0 'Oreo' (API Level 26)

Your `AndroidManifest.xml` must include the following permissions:

- `BLUETOOTH` and `BLUETOOTH_ADMIN` are required to discover and communicate with Bose AR devices.
- `ACCESS_FINE_LOCATION` is required due to searching for BLE devices on Android 10.

[1]: For developers creating experiences with wider minimum requirements than our SDK supports, we provide compilation support for Android SDK 21 and greater. However, you must restrict your users from attempting to access an experience that utilizes the *Bose AR SDK for Unity* running Android under our minimum supported version.

Platform: iOS

- **Xcode:** 11.0 or greater

Your Unity project must have the following settings enabled:

- **Required Architecture:** ARM64
- **Supported iOS Versions[1]:** iOS 12.0+
- **Background Behavior:** Custom
- **Background Modes:** Uses BLE Accessories

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

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

(**NOTE:** Required for targeting < iOS 13.)

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

(**NOTE:** Required for targeting iOS 13+.)

[1]: For developers creating experiences with wider minimum requirements than our SDK supports, we provide compilation support for iOS 11 and greater. However, you must restrict your users from attempting to access an experience that utilizes the *Bose AR SDK for Unity* running iOS under our minimum supported version.

Firmware

Device	Minimum Supported	Recommended
Frames	2.1.2	4.0.3
QuietComfort 35 II	4.3.6	4.5.2
Noise Cancelling Headphones 700	0.9.2	1.2.11

NOTE: Minimum supported firmware ensures Bose AR functionality, but does not guarantee all features available in the SDK are supported. 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 (2018.4, 2019.1, 2019.2).

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 or USB Provider.

1. In order to do this, you must add the following scenes to your Build Settings:
    - Bose/Wearable/ExampleApp/Scenes/Root.unity
    - Bose/Wearable/ExampleApp/Scenes/MainMenu.unity
    - Bose/Wearable/ExampleApp/Demos/Basic/Scenes/BasicDemo.unity
    - Bose/Wearable/ExampleApp/Demos/Advanced/Scenes/AdvancedDemo.unity
    - Bose/Wearable/ExampleApp/Demos/Gesture/Scenes/GestureDemo.unity
    - Bose/Wearable/ExampleApp/Demos/Debug/Scenes/DebugDemo.unity
2. Open the `Root.unity` scene and edit the **Wearable Control** object to utilize one of the aforementioned providers as the **Editor Default Provider**
3. 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/Modules/Connection`
- `Bose/Wearable/Modules/Debugging`
- `Bose/Wearable/Modules/ModelLoader`
  - This depends on the content in the `Connection` module.
- `Bose/Wearable/ExampleApp`
  - Content in this folder depends on some UI assets present in the `Connection` and `Debugging` modules.

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 "RotationMatcher" 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 "RotationMatcher" 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

Center

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

Reset

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

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/Modules/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`.

```
// Represents the current status of a connection task.
[Serializable]
public enum ConnectionStatus
{
    // No device is connected, and we are not searching for devices.
    Disconnected,

    // Trying to connect to the last successfully connected device. If that devi
    // search a connection attempt will automatically begin, otherwise it will t
    AutoReconnect,
```

```
    // A permission is needed in order for the SDK to properly function which re
    PermissionRequired,

    // A service is needed in order for the SDK to properly function which requi
    ServiceRequired,

    // All Bose AR SDK requirements for permissions and services have been met.
    RequirementsMet,

    // 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
    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
}
```

For a thorough understanding of the flow between these various states, we encourage you to review the Bose AR Connection Flow.

Searching for Devices

The connection process is a two-step process initiated by searching for devices via `WearableControl.Instance.SearchForDevices`.

The process will first verify access to operating-system (OS) permissions and services that the *Bose AR for Unity SDK* requires in order to properly function.

If any permission or service requires user action to grant or enable, the state `ConnectionStatus.PermissionRequest` or `ConnectionStatus.ServiceRequest` will be emitted and a UI screen notifying the user that a resource is required will be displayed.

The user can then choose to grant or enable the permission or service. If they do not, the state `ConnectionStatus.Failed` is emitted and the user is warned that the resource is needed for the Bose AR experience to function. The user can then continue without Bose AR or restart the OS requirements check by pressing **Connect**.

Once the user has granted or enabled all necessary services and permissions, the state `ConnectionStatus.RequirementsMet` will be emitted and the device search will begin.

Required Permissions by Platform

Platform	Permission
iOS	Bluetooth
Android	Bluetooth
	Location

Details on the permissions required for both iOS and Android are documented in the Platform Requirements.

Required Services by Platform

Platform	Service
iOS	Bluetooth
Android	Bluetooth
	Location Services

Once all OS requirements for permissions and services have been obtained and verified by the SDK, the `ConnectionStatus.Searching` state will be emitted, 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 receive 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 currently 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

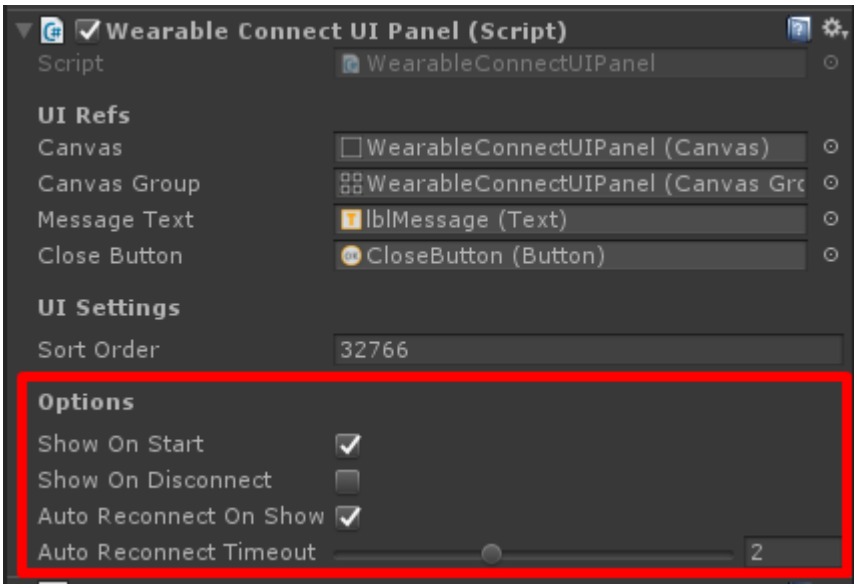
If a device passes successfully both the Secure Pairing, AppIntent, and Firmware checking phases, a `ConnectionStatus.Succeeded` state is emitted and the device will have connected successfully. Anytime a device is connected to, a session is created; this persists for the entire time it remains connected. 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.



In addition to the auto-reconnect feature on the `WearableConnectUIPanel`, an additional option for reconnection is available via the `WearableControl.Instance.ReconnectToLastSuccessfulDevice` method. This method may offer a faster connection experience as it skips several steps in the normal connection process.

These skipped steps include:
Searching for Devices
Firmware Checking
* If a firmware update is available, but not required: it will be automatically skipped.
* If a firmware update is required: it will result in a failed device connection.

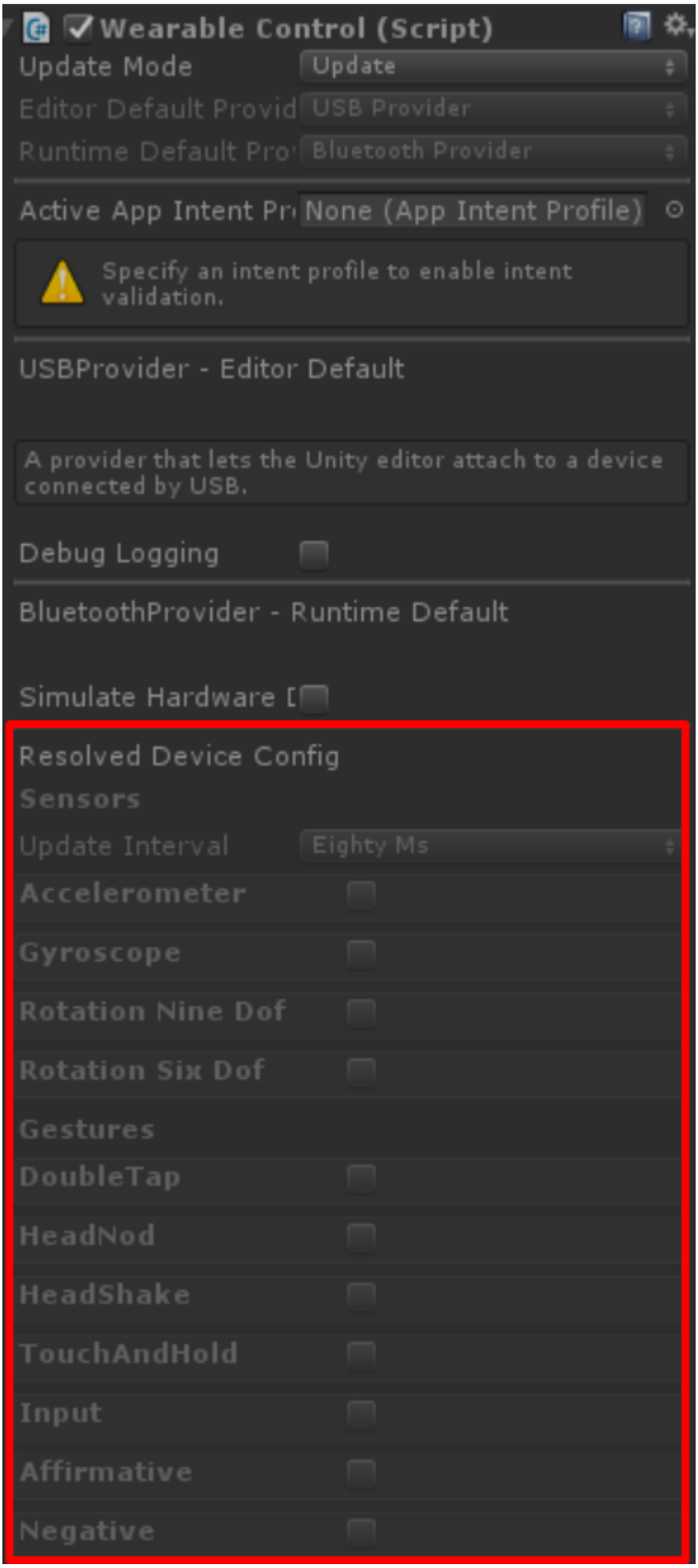
When reconnecting, the attempt will continue indefinitely until the specified device is found or explicitly cancelled by a user or developer. If a device was not previously connected to, the re-connection attempt will automatically fail.

It is advisable to only use this method of reconnection for a device previously connected in the same application session through the normal connection process. This ensures that the reconnected device will satisfy the App Intent Profile and will not automatically fail.

Device Configuration with WearableRequirement

The `WearableRequirement` component overcomes the challenge of coordinating the configuration of sensors, gestures, and update intervals across multiple components 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. The resolved device config representing the current device state can be seen on the inspector below.


Wearable Control (Script)
Update Mode
Update
Editor Default Provid
USB Provider
Runtime Default Pro
Bluetooth Provider
Active App Intent Pr
None (App Intent Profile)
Specify an intent profile to enable intent validation.
USBProvider - Editor Default
A provider that lets the Unity editor attach to a device connected by USB.
Debug Logging
BluetoothProvider - Runtime Default
Simulate Hardware
Resolved Device Config
Sensors
Update Interval
Eighty Ms
Accelerometer
Gyroscope
Rotation Nine Dof
Rotation Six Dof
Gestures
DoubleTap
HeadNod
HeadShake
TouchAndHold
Input
Affirmative
Negative

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

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.

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 and data throughput 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.

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. Note that only **device-agnostic gestures** may be directly enabled and subscribed to with the API.

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.

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.
- `WearableControl.Instance.CurrentGestureData` provides an array of all the `GestureData` received in the last poll. This field also provides access to per-gesture device timestamps.

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

Sensor Service Events

Due to bandwidth constraints associated with Bluetooth, there are certain situations when the product is not able to transmit AR data to your app. In those cases, we have to temporarily suspend AR.

For more information about these suspensions, please refer to the Service Suspensions document in the UX Design Guidelines documentation. While this document will repeat some of the information contained in the UX Design Guidelines, we will focus on the implementation and best practices for the related features available in the *Bose AR SDK for Unity*.

Introduction

Without proper notification, the result of a Sensor Service Suspension can be jarring for your users, as a connected Bose AR device will:

- Remain powered on.
- Maintain both Bluetooth Classic and BLE connection to your device.
- Stop transmitting all sensor and gesture data.

Given that the SDK can only detect this state upon connection, it is up to you to ensure that your application responds appropriately to two situations:

- A user connects to your application in a state where the sensor service is already suspended.
- The sensor service becomes suspended at runtime due to actions a user may take with their device external to your application.

Best Practices

We recommend a multi-pronged approach to handling this situation for your users:

1. Provide a UI that explains the situation to your user with instructions to resolve it.
2. Ensure your application reacts to the Sensor Service events as it is suspended and resumed.

Using the prefab

We offer a standalone UI prefab named `SensorServiceSuspensionUI` that informs the user of the proper action to take whenever the sensor service is suspended. It hides itself on `Awake`, shows itself only if `WearableControl.Instance.SensorServiceSuspended` is invoked, and

hides itself either when the connected device disconnects or when `WearableControl.Instance.SensorServiceResumed` is invoked.

- To quickly add the ability to detect and react to a Bose AR device's Sensor Service events, drop the `Bose/Wearable/Modules/SensorServiceSuspension/Prefabs/SensorServiceSuspensionUI` Prefab into the scene you would first like this UI to be available in.

Example: Detecting a suspension at connection.

The following example would provide the ability to detect a suspension upon connection and provide the ability to avoid launching your experience if a device was found to be in a suspended state.

```
using Bose.Wearable;
using UnityEngine;

public class ExampleStartGame : MonoBehaviour
{
    private WearableControl _wearableControl;
    private bool _shouldStartExperience;

    private void Awake()
    {
        _wearableControl = WearableControl.Instance;
        _wearableControl.DeviceConnected += OnDeviceConnected;
        _wearableControl.SensorServiceResumed += OnSensorServiceResumed;
    }

    private void OnDestroy()
    {
        _wearableControl.DeviceConnected -= OnDeviceConnected;
        _wearableControl.SensorServiceResumed -= OnSensorServiceResumed;
    }

    private void OnDeviceConnected(Device device)
    {
        if (device.deviceStatus.ServiceSuspended)
        {
            // Hold off on proceeding until the device is connected AND the sens
            // service is active
            _shouldStartExperience = false;
        }
        else
        {
            // Continue with the Bose AR experience
            _shouldStartExperience = true;
        }
    }
```

```
    private void OnSensorServiceResumed()
    {
        // Proceed with the experience as we can now receive sensor and gesture
        // configure devices, or any other device action.
        if (!_shouldStartExperience)
        {
            // Continue with the Bose AR experience
            _shouldStartExperience = true;
        }
    }
}
```

Example: Reacting to a suspension at runtime.

The following code example listens for the suspend/resume events, and would give you the ability to pause/resume your application in response.

```
using Bose.Wearable;
using UnityEngine;

public class ExampleGameplaySystem : MonoBehaviour
{
    private WearableControl _wearableControl;

    private void Awake()
    {
        _wearableControl = WearableControl.Instance;
        _wearableControl.SensorServiceSuspended += OnSensorServiceSuspended;
        _wearableControl.SensorServiceResumed += OnSensorServiceResumed;
    }

    private void OnDestroy()
    {
        _wearableControl.SensorServiceSuspended -= OnSensorServiceSuspended;
        _wearableControl.SensorServiceResumed -= OnSensorServiceResumed;
    }

    private void OnSensorServiceSuspended(SensorServiceSuspendedReason reason)
    {
        // Pause the Bose AR experience until the user can correct the situation
    }

    private void OnSensorServiceResumed()
    {
        // Resume the experience as we can now receive sensor and gesture data,
        // configure devices, or any other device action.
    }
}
```

Sensor Service Suspension Durations

It is worth noting that sensor service suspensions have a variable duration depending on the functionality invoked by the user.

- A suspension due to activation of the **Virtual Personal Assistant (VPA)** is often short and represents the duration in which the microphone is activated to receive the user query for the active VPA.
- A suspension due to **multi-point connectivity** or **Music Share** has an indefinite duration -- this suspension will remain until the user remedies the situation by disconnecting the other device they are currently interacting with through multi-point or music share.

Sensor Service Suspension Matrix

The following functions will cause service suspensions on the following devices:

	Virtual Personal Assistant (VPA)	**Multi-point**	**Music Share**
Frames	Yes	N/A	N/A
QuietComfort 35 II	Yes	Yes	Yes
Noise Cancelling Headphones 700	Yes	No	N/A

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.

Bluetooth Provider

The Bluetooth Provider allows direct access to the hardware using an underlying native Bose AR 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 Bluetooth 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, only the following Bose AR devices support the USB Provider:

- Bose Frames
- Bose QuietComfort 35 II

NOTE: The Bose USB Updater (as well as our USB Provider) maintains an exclusive lock on the device during operation. If you are having trouble connecting to the device with the USB Provider, please disconnect your device, close the USB Updater and the Unity Editor, reconnect your device, and relaunch the Unity Editor.

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

To hear audio through your supported device:

1. Connect to your device via the USB Provider
2. Go to your system audio settings and select your Bose AR device as the default audio output.

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 simulated 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)
Bluetooth	Y	N/A	Data, Audio
USB	Y[1]	Data, Audio[1]	N/A

Provider	Device Required?	Features (Editor)	Features (Runtime)
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.

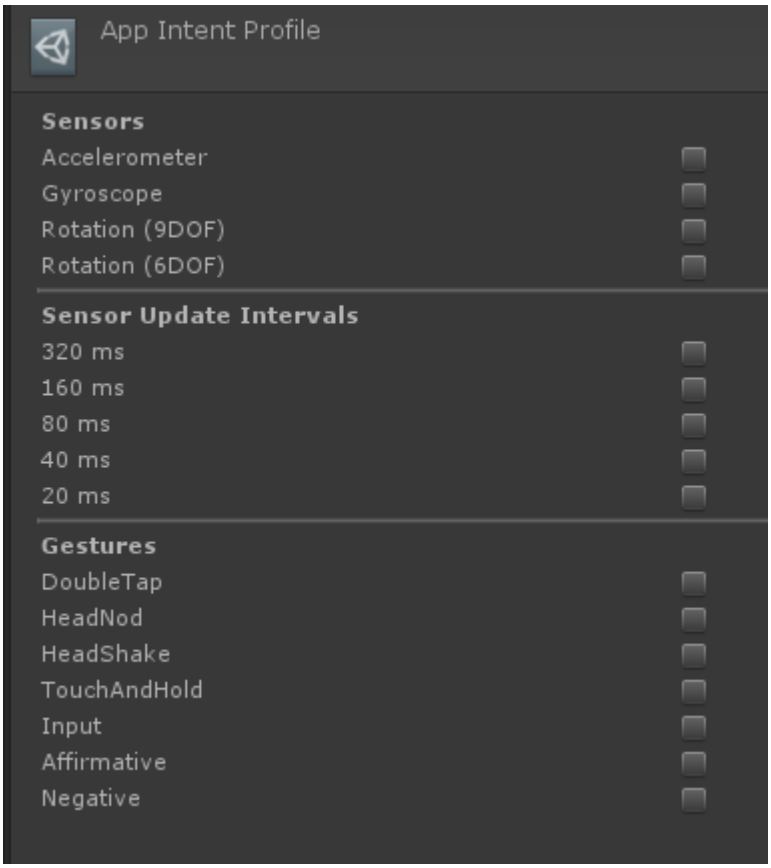
The SDK will process the App Intent Profile and determine if the connected device's hardware and firmware can support the requested configuration, and return this result back to Unity. In the event that a connected Bose AR device cannot fulfill the specified intents, users can be instructed to update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

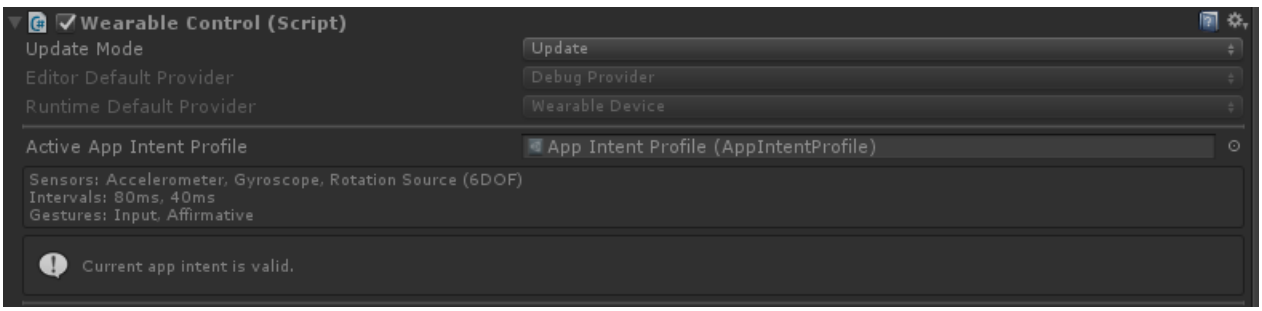
To create an App Intent Profile, navigate to `Assets -> Create -> Bose Wearable -> App Intent Profile`. This will create an intent profile asset in your project, which can be edited in the Unity inspector.

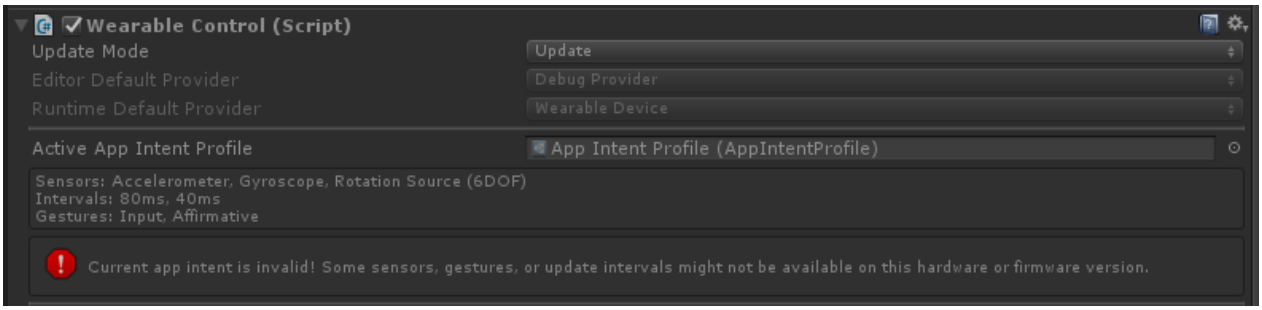
While we primarily expect developers to create profiles using the inspector, functionality also exists to create and modify them at runtime; see `AppIntentProfile.cs` for the relevant methods.



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.





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

Bluetooth 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 successfully connect to your application.

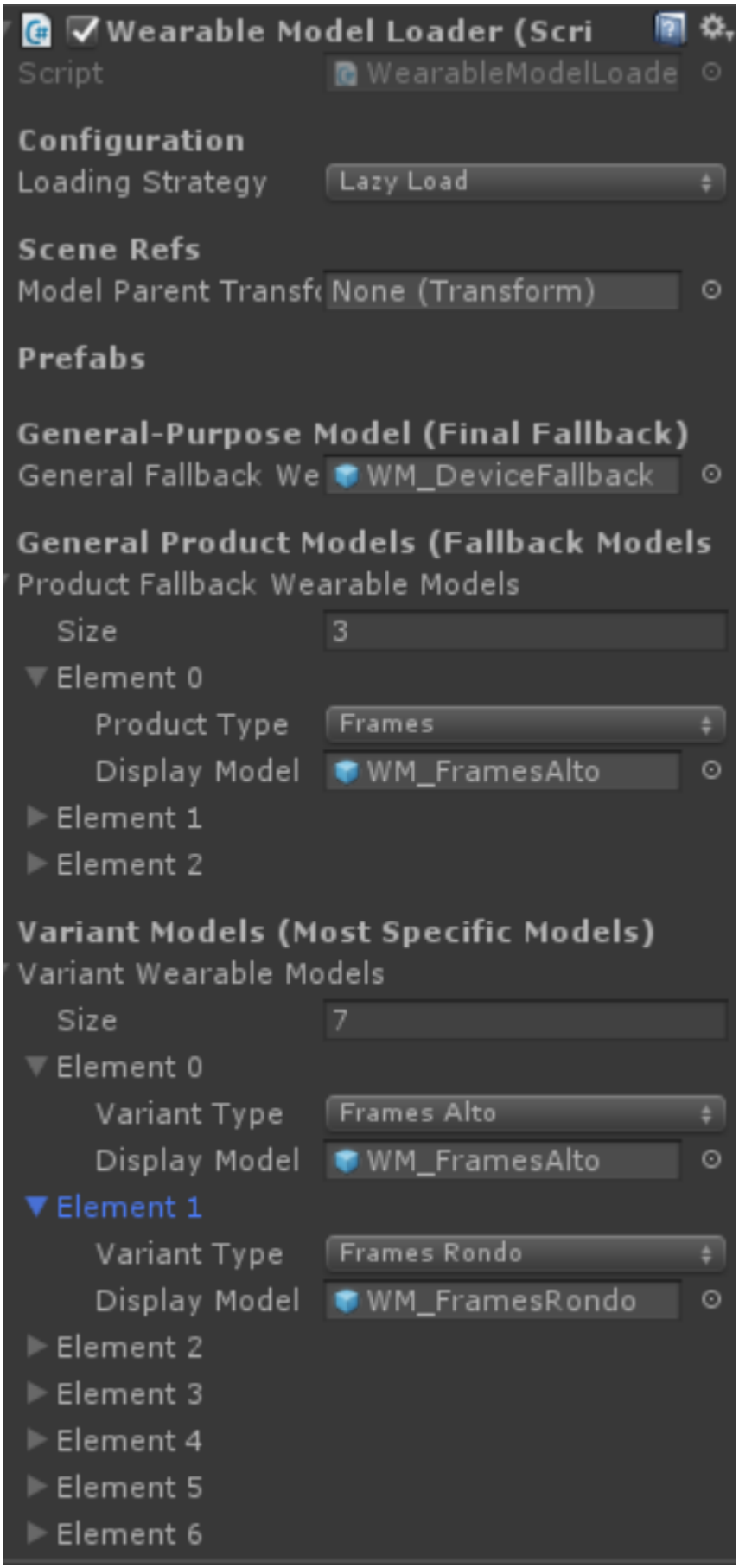
USB and Debug Provider

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.

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 (Scri
Script
WearableModelLoade
Configuration
Loading Strategy
Lazy Load
Scene Refs
Model Parent Transfo
None (Transform)
Prefabs
General-Purpose Model (Final Fallback)
General Fallback We
WM_DeviceFallback
General Product Models (Fallback Models
Product Fallback Wearable Models
Size
3
Element 0
Product Type
Frames
Display Model
WM_FramesAlto
Element 1
Element 2
Variant Models (Most Specific Models)
Variant Wearable Models
Size
7
Element 0
Variant Type
Frames Alto
Display Model
WM_FramesAlto
Element 1
Variant Type
Frames Rondo
Display Model
WM_FramesRondo
Element 2
Element 3
Element 4
Element 5
Element 6

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.

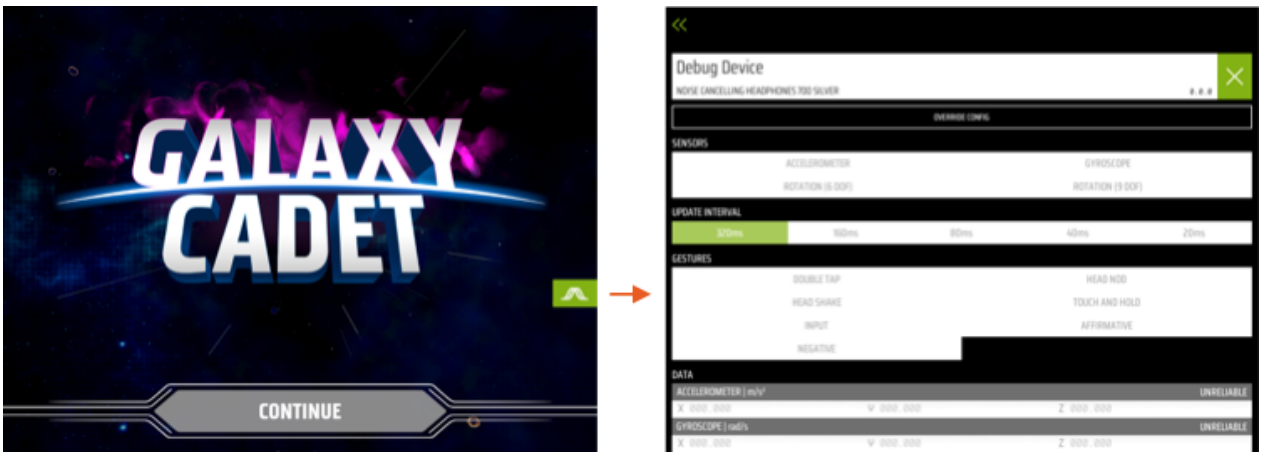
This tool functions on all supported platforms with all providers; both in-editor and on-device.

Setting up the Prefab

To add the tool to your project, simply drop the `Bose/Wearable/Modules/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.

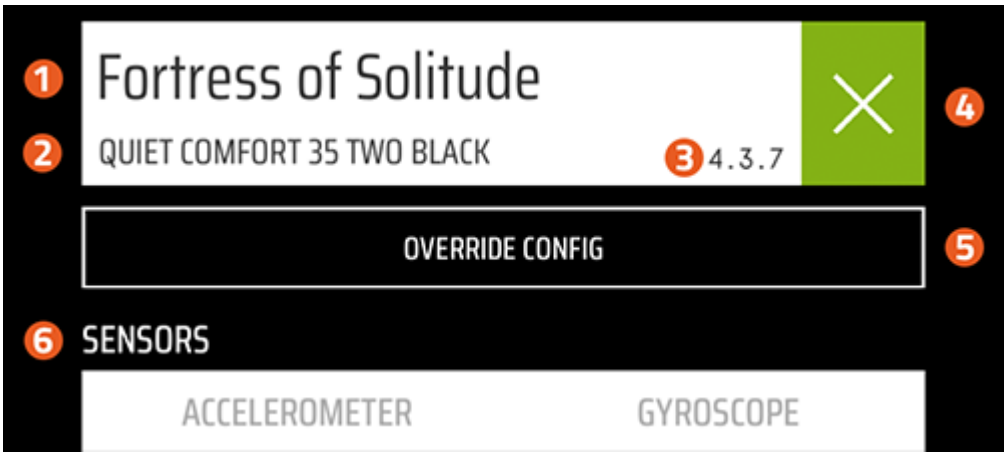


If the button to open the DebugUIPanel is in the way at runtime, you may move it to a more convenient location. Press and hold on the button and drag it anywhere within the animated glowing area that appears.

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

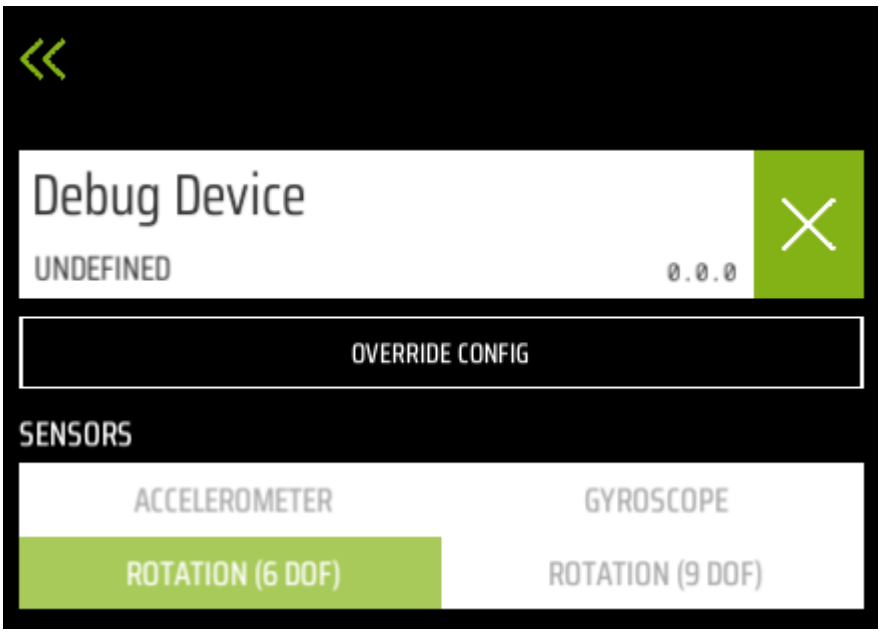


When a device is connected, you will be presented with:

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.

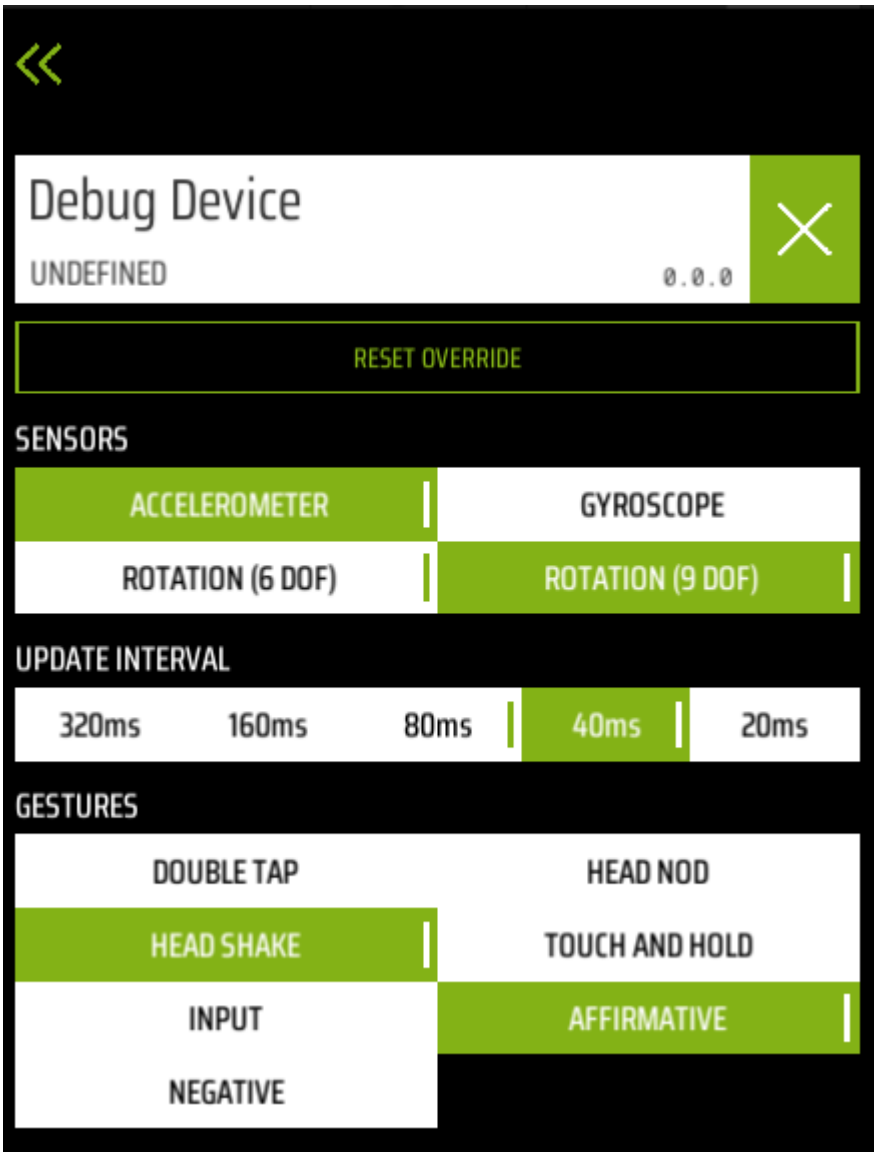
When opened for the first time, the debug panel will show a series of disabled controls that represent the current device configuration. (To learn more about configuring your device at runtime, please see Device Configuration.)



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

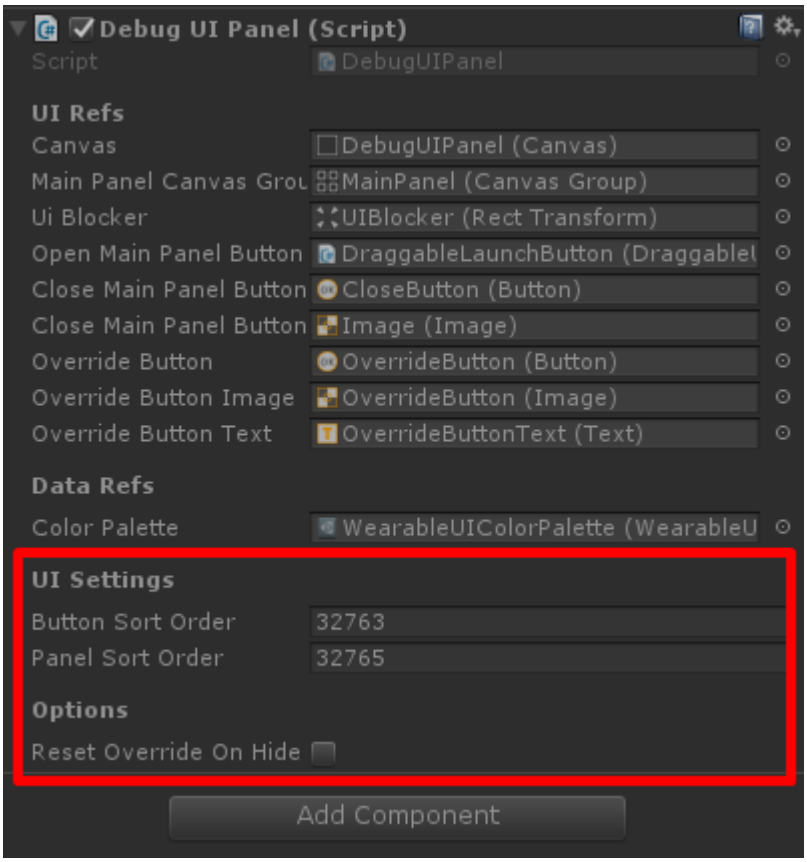
Once the device config has been overridden: the controls will become enabled. Any new configuration defined by the override will be denoted by a rectangular 'pip' and take immediate priority over any underlying configuration from your application. This includes:

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



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

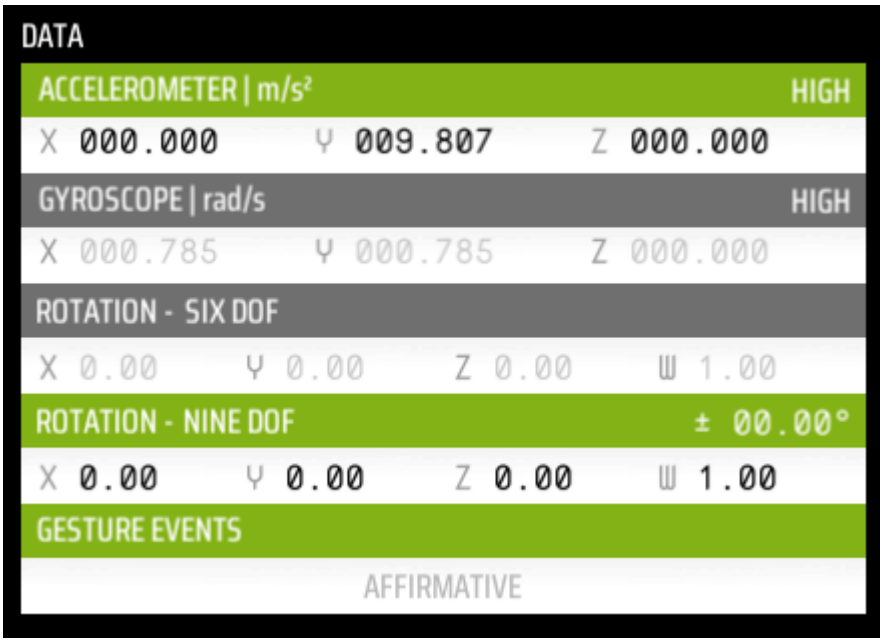
If you would like to ensure that an override never persists when returning to your application, you can check `Reset Override on Hide` on the "DebugUIPanel" Prefab to automatically clear the configuration when the debug panel is closed.



Debug: Data

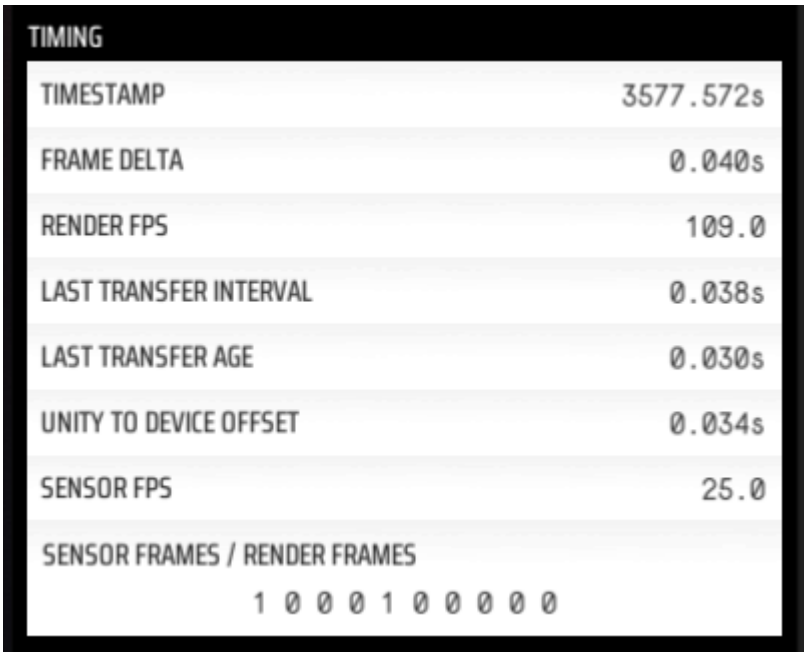
The Data section allows a user to view the last `SensorFrame` data received from the available sensors and gestures.

- Sensors or Gestures that are turned off or unavailable will be colored gray and faded.
- The Accelerometer and Gyroscope will each display their respective accuracies.
- The 9-DOF rotation sensor will display its measurement uncertainty. (**NOTE:** The 6-DOF rotation sensor will always report an uncertainty value of `± 0°`)
- 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.
- Gesture Events will temporarily display the last detected gesture as reported by the active provider.



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.

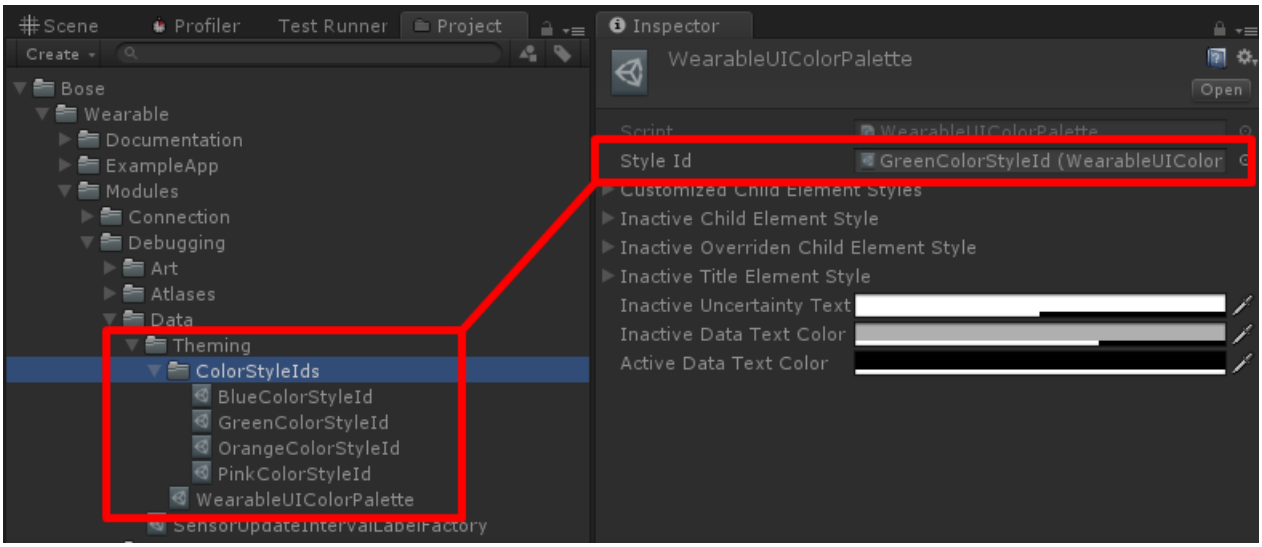


- **Timestamp:** The last `SensorFrame` timestamp received by the device.
- **Frame Delta:** The age in seconds of the most recently-received `SensorFrame` data.
- **Render Frames Per Second:** The number of Unity render frames per second.
- **Last Transfer Interval:** The time between the most recent `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.



Localization

Overview

The *Bose AR SDK for Unity* has the built-in capability to localize any of the user-facing content provided in the SDK.

Dealing with Localization in Code

The `LocaleCache` is a static class that provides an API to view which languages are supported, set a preferred language, and retrieve locale values for a given key. For easy access to localization keys in code, `LocaleConstants` is a static class containing all of them.

Example: Setting a Preferred Language

For any given region, a developer may want to set the displayed language for the *Bose AR SDK for Unity* to be in one of that region's languages. This can be done by calling `LocaleCache.SetPreferredSystemLanguage` with the desired language. It is important to note that the preferred language will only be displayed if supported by the SDK which can be checked by calling `LocaleCache.IsSupportedSystemLanguage` or using one of the languages returned by `LocaleCache.GetSupportedLanguages()`.

```
// If English is a supported language for localization, set that as our preferre
if (LocaleCache.IsSupportedSystemLanguage(SystemLanguage.English))
{
    LocaleCache.SetPreferredSystemLanguage(SystemLanguage.English);
}
```

Example: Retrieving a collection of supported languages

```
// Retrieves a list of supported languages for localization and prints them to t
var supportedLanguages = LocaleCache.GetSupportedLanguages();
foreach (var supportedLanguage in supportedLanguages)
{
    Debug.LogFormat("Bose AR SDK for Unity supports localization in {0}", suppor
}
```

Example: Retrieving Locale Values using Keys

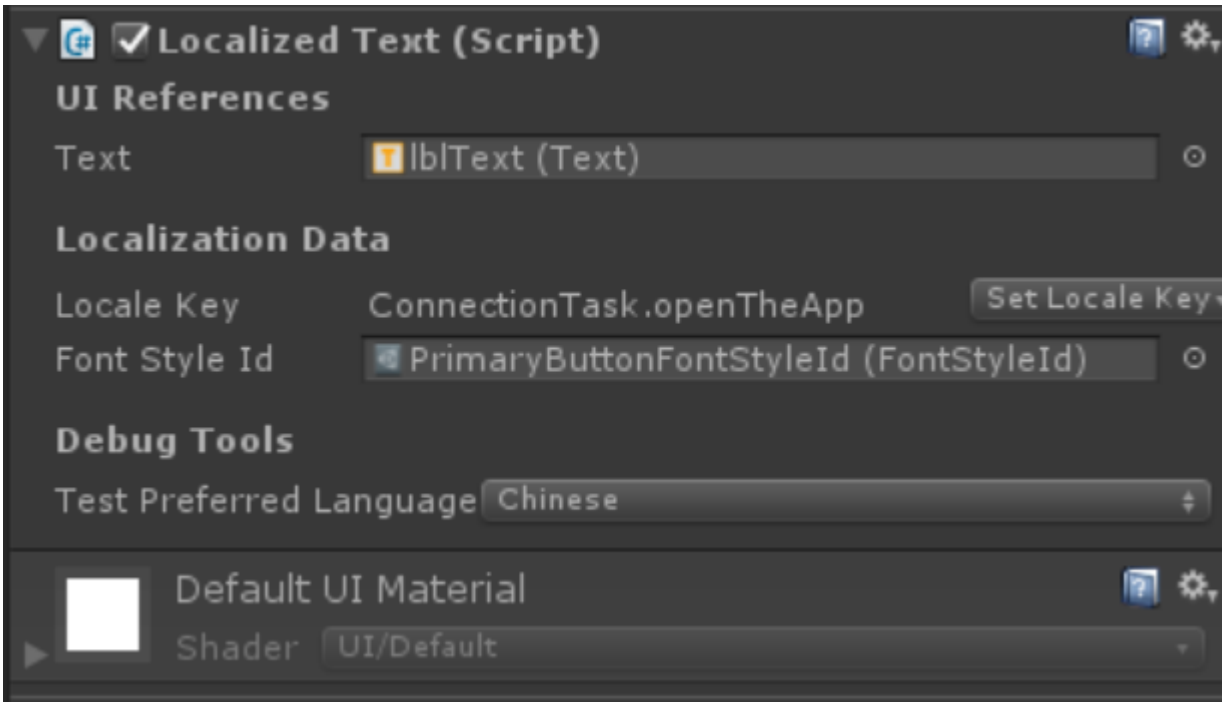
```
// Retrieves a locale value for the current display language if available; true
// and value initialized with the locale, otherwise false will be returned.
string value;
if (LocaleCache.TryGetLocaleValue(LocaleConstants.BOSE_AR_UNITY_SDK_CONNECTION_S
{
    Debug.Log("Succeeded in getting locale value.");
}
else
{
    Debug.Log("Failed to get locale value.");
}
```

```
// Retrieves a locale value for specifically the Finish language if available; t
// if found and value initialized with the locale, otherwise false will be retur
string value = null;
if (LocaleCache.TryGetLocaleValue(SystemLanguage.Finnish, LocaleConstants.BOSE_A
{
    Debug.Log("Succeeded in getting locale value.");
}
else
{
    Debug.Log("Failed to get locale value.");
}
```

Setting up Localization

The `LocalizedText` component allows for configuring a given `UnityEngine.Text` component with the proper localization value and `Font` based on a selected localization key. This localization key can be set either in the inspector of the `LocalizedText` component or programmatically via its `SetLocaleKey` method.

The language displayed can be changed via the `Test Preferred Language` dropdown, which will change the displayed language for all `LocalizedText` components.



Displayed Language

The language displayed on the `WearableConnectUIPanel` prefab is dynamic and can be one of many values depending on several factors.

- If a preferred language is set and supported, the localized text displayed will be in that language.
- If a preferred language is set and not supported, the fallback language of English will be used and the localized text displayed will be in the fallback language.
- If a preferred language is not set and the current device OS language is supported, the localized text displayed will be in the current device OS language.
- If a preferred language is not set and the device OS language is not supported, the fallback language of English will be displayed and the localized text displayed will be in the fallback language.

NOTE: The *Bose AR SDK for Unity* uses Unity's `Application.systemLanguage` to determine the device language. For more information, please refer to the Unity documentation.

Example: Setting a Locale Key on `LocalizedText`

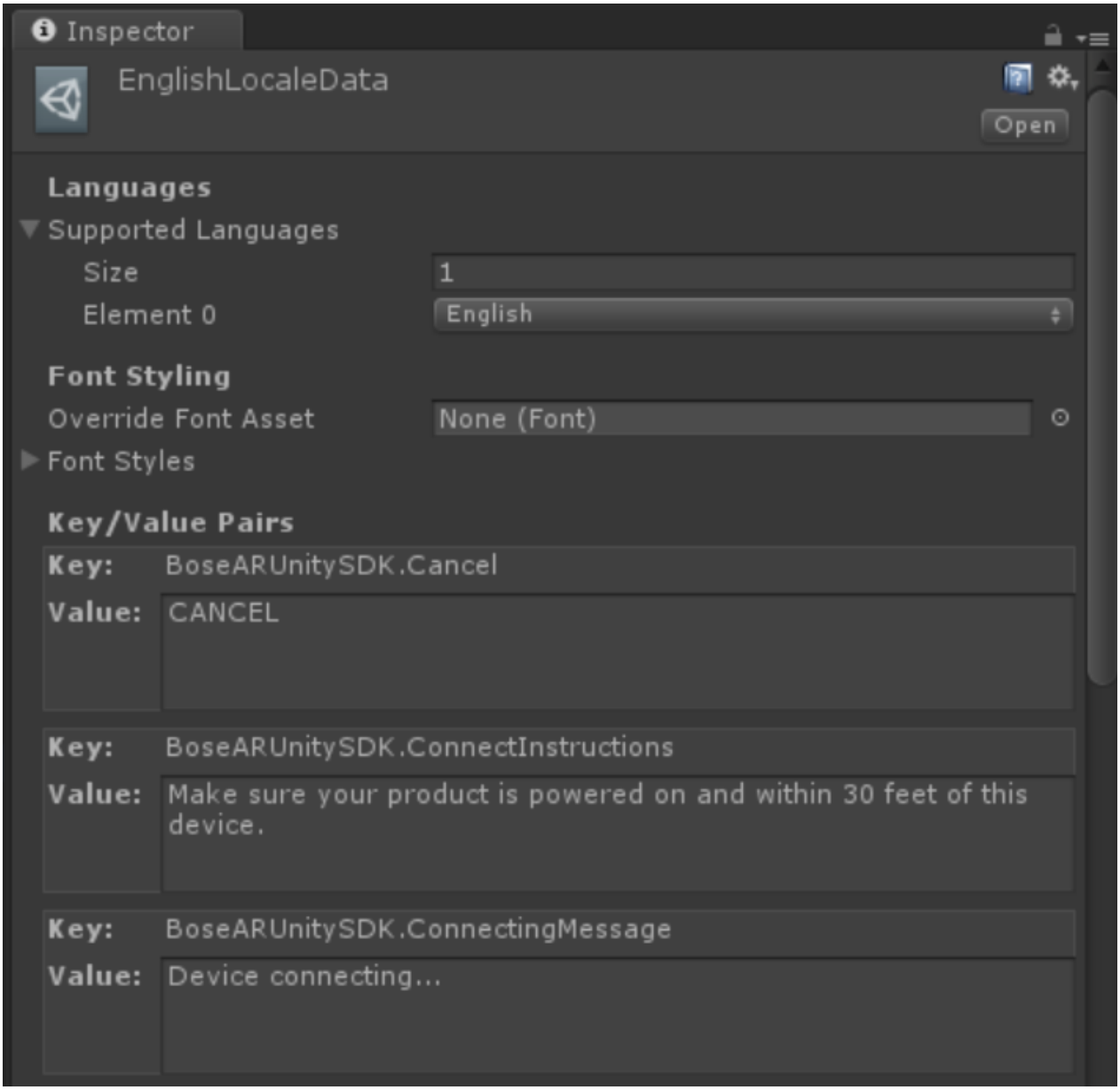
```
using Bose.Wearable;
using UnityEngine;

public class ExampleUIMonoBehaviour : MonoBehaviour
{
    [SerializeField]
    private LocalizedText _localizedText;
```

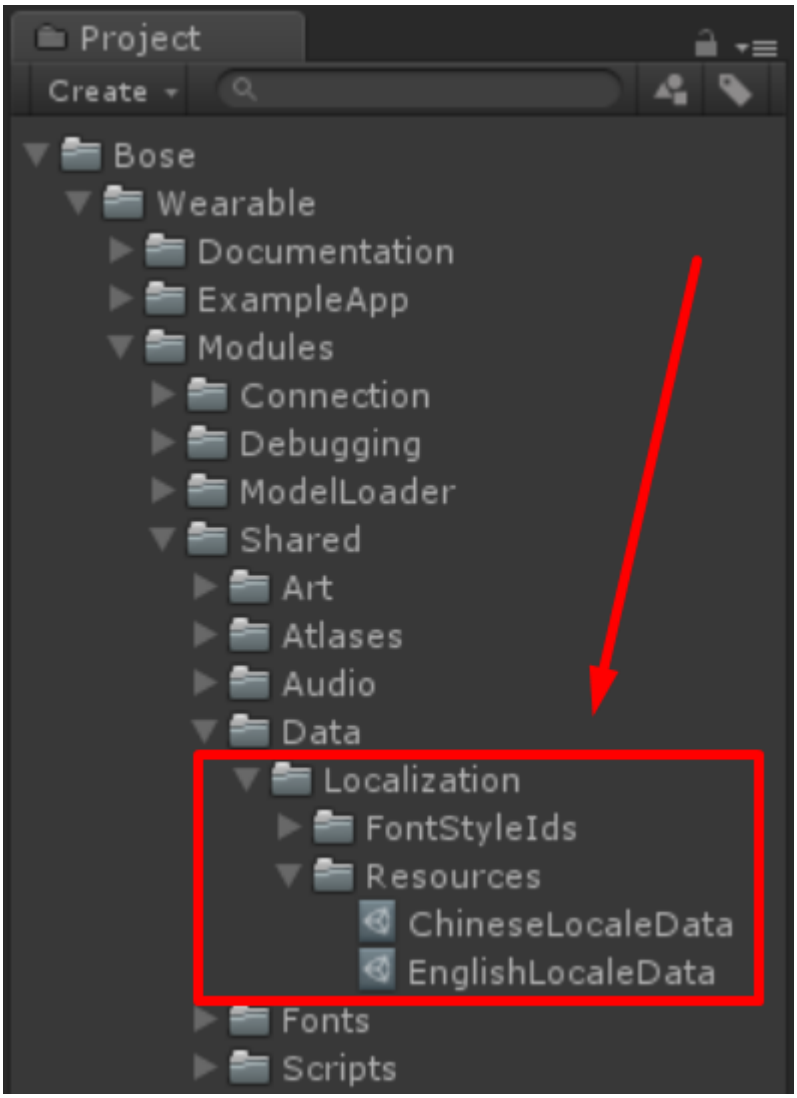
```
    private void Awake()
    {
        _localizedText.SetLocaleKey(LocaleConstants.BOSE_AR_UNITY_SDK_CONNECTION
    }
}
```

Localization Data

The `LocaleData` ScriptableObject is an asset that contains the localization keys and values for a given locale.



If need arises to view these for a given language(s), that can be done by selecting the appropriate instance in the project folder at the path `Assets\Bose\Wearable\Modules\Shared\Data\Localization\Resources`.


Project
Create
Bose
Wearable
Documentation
ExampleApp
Modules
Connection
Debugging
ModelLoader
Shared
Art
Atlases
Audio
Data
Localization
FontStyleIds
Resources
ChineseLocaleData
EnglishLocaleData
Fonts
Scripts

Editor Preferences

These settings are set on a per-user basis in the UnityEditor and persisted using EditorPrefs. They can be accessed by navigating to "Bose Wearable" section within Unity's Editor Preferences (`Edit->Preferences` for Windows and `Unity->Preferences` for macOS).

Demo Build Options

Auto-Show Demo Location on Build

This setting can be toggled on and off; when enabled, successfully building the demo will automatically open a folder to that location.

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:

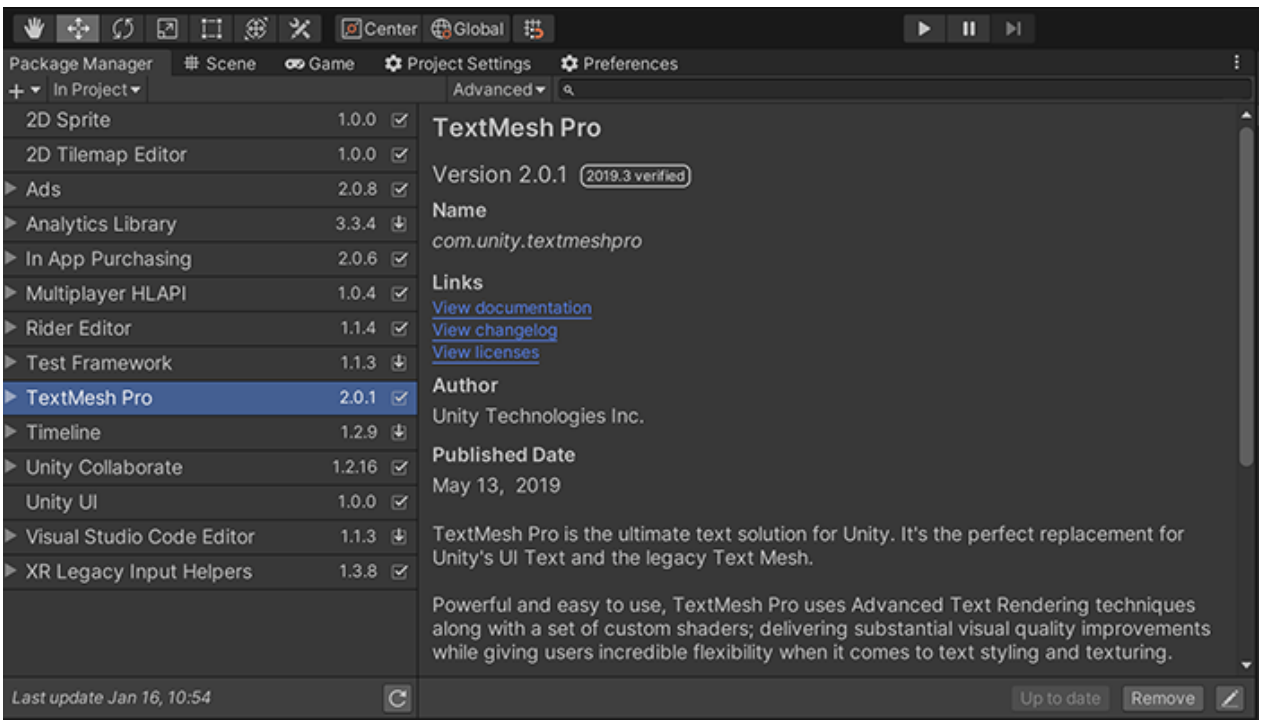
1. Open a new scene in Unity. (File > New Scene)
2. Delete all *Bose AR SDK for Unity* files in your Unity Project, including the:
    - `Bose/Tools` directory.
    - `Bose/Wearable` directory.
3. Import the new SDK.

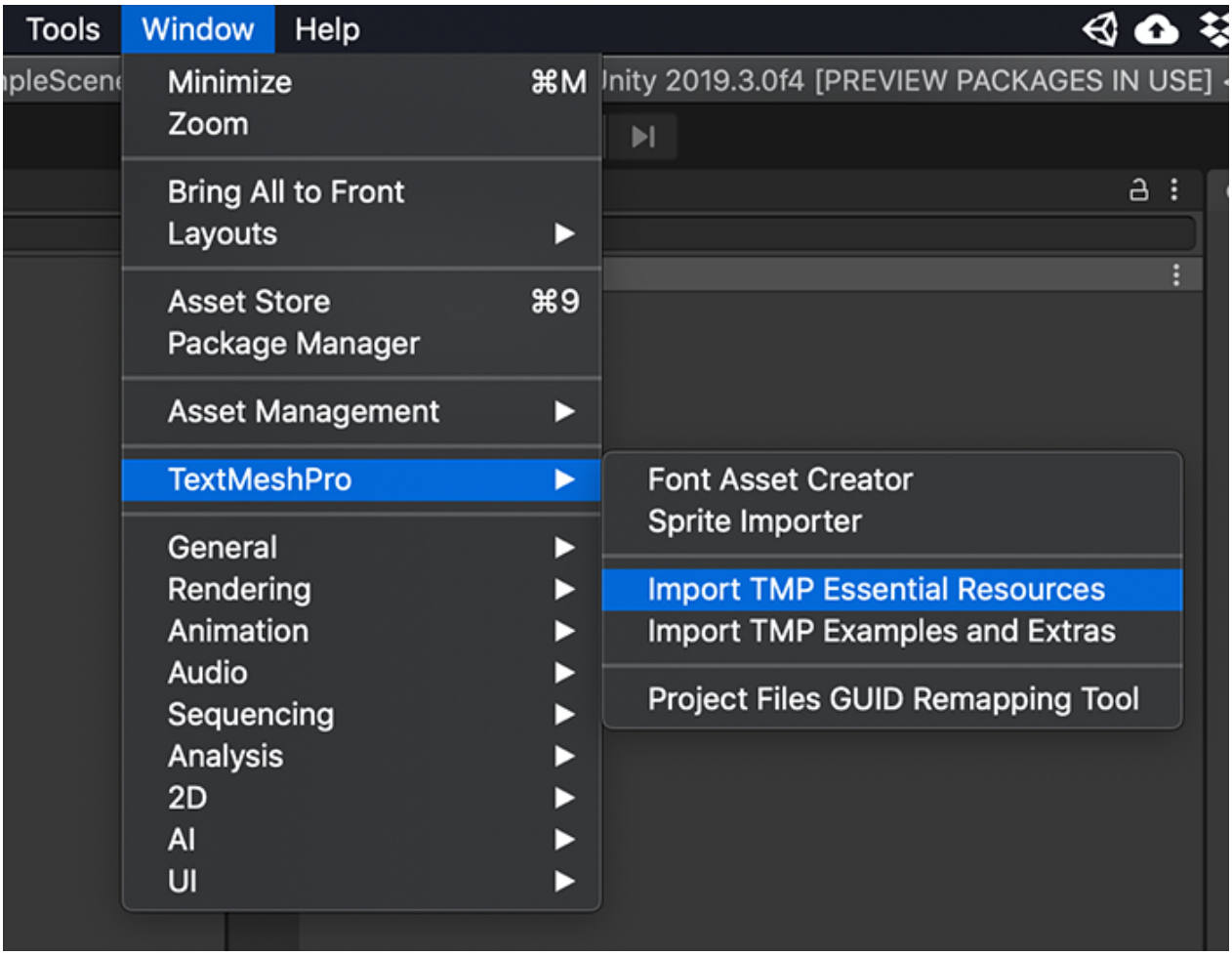
Previous Version Details

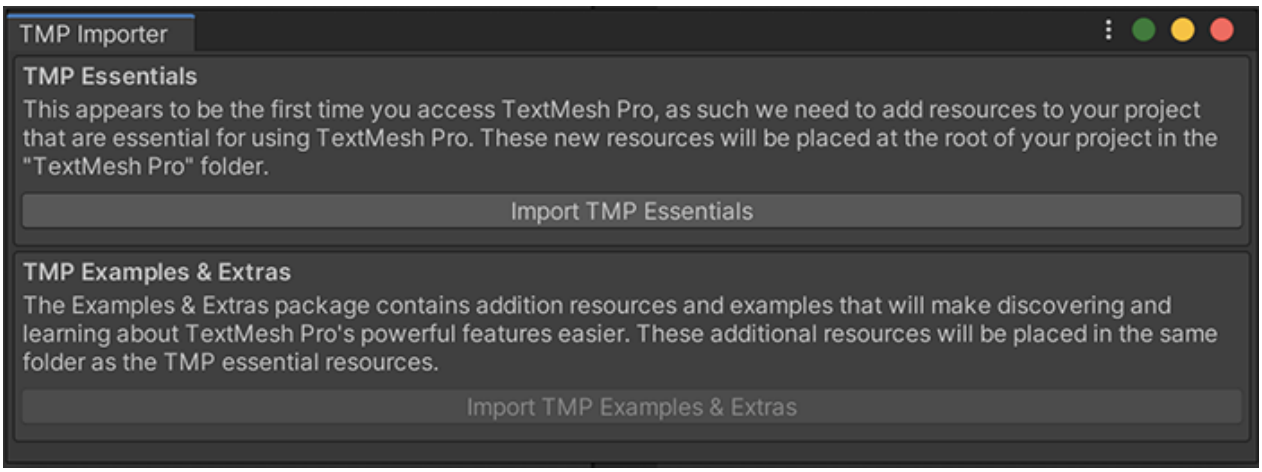
Version < 4.1.0

If you are upgrading from a version prior to `4.1.0`, you will need to import the TextMeshPro package in your project. If it is not present it can be added via the native Unity Package Manager window at **Window/PackageManager**.

When adding TextMeshPro to your project for the first time, you must also import the **TextMeshPro Essentials** package. This can be done by going to the menu item **Window/TextMeshPro/Import TMP Essential Resources** and importing the UnityPackage that it opens.







Version < 4.0.3

If you are upgrading from a version prior to `4.0.3`, you will also want to delete the following files prior to importing the new SDK.

- `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.

As of `4.0.3` and later, all native libraries were moved into the `Bose/Wearable/Plugins` directory.

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.

I can see my device listed, but connection over USB always fails.

This is commonly caused when another application (such as the Bose USB Updater) is maintaining an exclusive lock on the USB device. Close both the Updater and the Unity Editor, unplug and plug in your Bose AR device, and relaunch the Unity Editor.

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.

v4.1.1

Release Date

2/24/2020

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

Fixes

- Fixed some unintentional data corruption due to a version mismatch when upgraded to 2018.4.

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**

v4.1.0

Release Date

1/27/2019

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

SDK/API Changes

- Deprecated support for Unity 2017.4. Minimum-supported Unity version is now 2018.4.0f1.
- The `WearableConnectUIPanel` and `SensorServiceUIPanel` now require the TextMeshPro (minimum v1.3) package.
- Deprecated methods `GetPermissionRequiredText` and `GetServiceRequiredText` on `WearableControl` in favor of new localization feature.

Additions and Improvements

- SDK: New functionality has been added in the form of localization provided for all player-facing text on the `WearableConnectUIPanel` and `SensorServiceUIPanel` for any

supported languages. This is currently limited to English. A developer using the *Bose AR SDK for Unity* can also leverage localization for custom UI in the form of new APIs provided primarily by the `LocaleCache` class for retrieving locale values and `LocaleConstants` which contains fields for all locale keys as well as a new `LocalizedText` component for configuring locale in the Unity Editor.

- SDK: Reskinned WearableConnectUIPanel and SensorServiceUIPanel to be more consistent with iOS/Android SDK designs.
- SDK: Confirmed support with 2019.3.0f3. (Additional testing will be run when 2019.3 is publicly released.)
- SDK: Add new Sensor Service Suspension Reason for OTA Firmware Updates. (Currently only triggered on Bose Frames.)
- ConnectUI / SensorServiceUI: Replaced all text meshes with TextMeshPro for crisp font rendering on all resolutions.
- ConnectUI / SensorServiceUI: Updated audio/visuals to match Native SDK screen design.
- iOS: Removed unused assets and files in iOS Native Frameworks.
- USB: Added support for additional Frames Alto variants in the USB Provider.
- iOS: Bridge rebuilt with Native SDK. (v4.0.22)
- Android: Bridge rebuilt with Native SDK. (v4.0.16)

Fixes

- Unity: Added pragmas to hide 0649 warnings in Unity console for private, serialized fields.
- Unity: Fixed bug with Bose.Wearable AssemblyDefinition where importing the plugin on newer versions of Unity could result in unsafe flag in CSC file being ignored and causing compile errors.
- Android: Remove calls to JNI::Call that are deprecated in 2019.x and instead use either sbyte or int where appropriate.
- iOS: Fixed bug on iOS where certain devices would not show up again in the search window after connecting and disconnecting earlier that application session.
- iOS: Fixed bug with XcodePostBuildProcessor that would prevent moving the SDK to any location in the Assets folder.
- Demo: Cannot move sound cage when "Info" popup is visible.

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**

v4.0.5

Release Date

11/13/2019

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

SDK/API Changes

- Allow compilation support on Android SDK 21+. Runtime support: Android SDK 26+.
- Allow compilation support on iOS 11+. Runtime support: iOS 12+.

Additions and Improvements

- A new `SensorServiceUIPanel` script and standalone Prefab have been added that allows a developer to warn users when their connected device has its sensor service suspended or resumed. For more details and instructions on usage, see Sensor Service Events.
- Greatly improved performance for the USB provider on Mac and Windows while playing in the Unity Editor.
- Significantly improved the connection UX for the `WearableConnectionUIPanel` and its Prefab for discovering and informing user how to grant/provide permissions and services that are required for the *Bose AR SDK for Unity* to function properly.
- Updated Android Bridge to use Android SDK 4.0.15.
- Updated iOS Bridge to use iOS SDK 4.0.20.

Fixes

- Fixed missing image in **About** pop-up window.
- Resolved an issue where the USB Provider wasn't working for some users on Windows.

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**

v4.0.4

Release Date

10/10/2019

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

SDK/API Changes

- New functionality: Reconnect to last device. This form of reconnection requires a device to have previously successfully connected, is significantly faster than the "auto-reconnect" mode, and will persist until the last connected device is found and an attempt is made. For more information, please see the Discovery and Connection documentation.

- Android: Due to changes in Android 10, our SDK will now check/request `FineLocation` instead of `CoarseLocation` where applicable.

Additions and Improvements

- Added support for Android 10.
- Added ability to cancel the connection process.
- Added SDK license as text file in the Documentation folder. Also made available via the menu: **Tools > Bose Wearable > Help > License**.

Fixes

- Fixed several issues with logging for iOS.
- Added fix for Bluetooth Permission flow on iOS 13 where a denied permission would result in the device search UI hanging. This will now result in an immediate connection failure.

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**

v4.0.3

Release Date

9/23/2019

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

SDK/API Changes

- The `Plugins/` directory has been relocated to sit within the `Bose/Wearable` directory. Please read the Upgrade Guide on how to safely update your project to this version.
- Removed `Device.isConnected`. Please use `WearableControl.Instance.IsDeviceConnected`.

Fixes

- Updated known devices for USB Provider. Resolves connection failures for users who had a Bose AR device but could not connect over USB.

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**

v4.0.2

Release Date

9/20/2019

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

SDK/API Changes

- `WearableSensor` and `WearableGesture` now have public `Id` properties to make them easier to identify if you store them in a list.
- Deprecated `Device.isConnected`. Please use `WearableControl.Instance.IsDeviceConnected`.

Additions and Improvements

- Added support for iOS 13.
- `GestureDetector` will now trigger a device reconfiguration if necessary in `OnEnable` and `OnDisable`.
- Additional iOS SDK Logging will be printed to console when "Debug Logging" is enabled on `WearableControl`.

Fixes

- Disconnecting on the "Firmware Update Available" connection screen is now properly detected.

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**

v4.0.1

Release Date

8/27/2019

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

SDK/API Changes

- `GestureDetector` now will properly subscribe/unsubscribe for the given gesture when enabled/disabled. Previously this was handled during Awake/Destroy.

Additions and Improvements

- Demos: Building the demo from the **Tools > Bose Wearable > Build Wearable Demo** menu item will now show the build in Finder/Explorer when the build completes. (This may be disabled in the Preferences window.)
- SDK: Added ability to simulate a sensor or gesture configuration failure in the Debug Provider.
- SDK: Confirmed Unity 2019.2 support. (No changes were necessary.)
- Models: Updated textures for both Bose Frames variants.
- Added more setup/troubleshooting documentation for the USB Provider.

Fixes

- Fixed editor stability and connection issues with the USB Provider.
- Fixed a casing issue that would cause failed compilations on Unity 2018.3.0 - 2018.3.3.
- Wearable Model Loader will respect the parent scale upon instantiation.
- Fixed devices not showing up when disconnecting and re-searching.
- Fixed issue where the connection panel could become stuck open.
- iOS: Fixed an intermittent crash when reconnecting to device.
- iOS: Fixed logging (At times, garbage was being printed).

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**

v4.0.0

Release Date

7/17/2019

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

SDK/API Changes

- All API that had been previously marked obsolete has been removed. Please see previous release notes for guidance on updating your API if compile errors arise.
- The project has been heavily reorganized for future development. You *must* follow the Upgrade Guide to upgrade to this version.

Additions and Improvements

- Demos: Reduced loading times between demos.

- SDK: Cleaned up shared asset usage across demos and included modules.
- SDK: Documentation Menu Item will attempt to load the local PDF or navigate to the website if the PDF cannot be found.
- SDK: Refactored editor-only constants to a separate class, marginally overall reducing memory load at runtime.
- SDK: RotationMatcher now has the capability to define an UpdateInterval which will be resolved against a discovered WearableRequirement, or applied to a generated one.
- SDK: Ability to enable debug logging exists in every provider inspector, and defaults to off. (Additional debug information can be useful when debugging connection flows for custom UI/UX.)

Fixes

- Fixed RSSI (Signal Strength) not updating on iOS.
- Fixed runtime NRE due to Compiler Override auto-generation.
- Documentation: Updated images to reflect visuals of current inspectors.
- Demos: Fixed Info Panel for Debug Demo.

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**

v0.16.0

Release Date

7/9/2019

This release is the last major beta release. All items marked obsolete will be removed in the next major release. Please update your app's API usage accordingly.

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

SDK/API Changes

- Success and Failure callbacks have been removed from `WearableControl.ConnectToDevice`, please use the `ConnectionStatusChanged` event for similar notifications.
- Support for the Proxy Provider has been removed.

Additions and Improvements

- SDK: Added support for Active Noise Reduction (ANR) & Controllable Noise Cancellation (CNC) for Supported Devices.

- Demos: Improved SFX for Gesture Demo
- Demos: Simplified Basic Demo UX
- DebugPanel: Added accuracy labels for Accelerometer and Gyroscope in the Debug Panel.
- Added automated process to manage creation of compiler override files for Unity 2017.4, 2018.4 and 2019.1.

Fixes

- Demos: Fixed Splash Logos
- Demos: Fixed several safeArea issues with demo content.
- Demos: Additional safeties around disconnected devices in demos and during load.
- Demos: Ensured consistency of button sizes across Main Menu
- Demos: Resolved lighting issues in Basic Demo.
- ConnectUIPanel: Fixed icon resolutions, removed unused assets.
- ConnectUIPanel: Fixed device buttons in search screen not resetting their size.
- ConnectUIPanel: Added safety around invalid auto-reconnect timeout values.
- ModelLoader: Fixed NRE when attempting to load a model when no device is connected.
- ModelLoader: Optimized texture usage and compression settings.
- Configuration: Fixed race condition where a locked update would block a second update.
- iOS: Fixed a few crashes occurring during connection.
- iOS: Cancelling during Secure Pairing can now safely recover and re-pair.
- Android: Fixed error spam on device disconnect.
- USB Provider: Fixed USB Audio that was not being enabled on a subset of Bose Frames.
- SDK: Fix Broken Documentation Link in MenuItem.
- SDK: Removed references that would have caused LINQ to be included at runtime.
- Added protection for duplicate WearableControl or WearableConnectUIPanel objects.

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**

v0.15.0

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 >= 1.1.4

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**

v0.13.0

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**