Technical Documentation

Creative - Game Development

EMG SENSOR PLUGIN (Unity)

Jan. 31, 2018

Angel Rodriguez

OVERVIEW

- EMG SensorCore script takes an analog readout from a Limbitless Arduino EMG sensor (2017 version) via USB port and grants access to it from inside of a Unity project running on PC, MacOS, or Unix.
- Automatically handles the Arduino firmata, USB COM port connection, and general sensor settings from inside of a SensorCore monobehaviour script which optionally survives between scenes in the project. Allows automatic connection and retrieval of a single Sensor from any USB port, including mid-game.
- Designed to be customizable and simple to access. Static instance access to the active SensorCore script allows any script in the project access without a variable reference. Goal is simplify of porting a game to the sensor.
- Has built-in smoothing and support for single-point or dual-point sensor thresholds and sensor caps for a variety of visualizations.
- SensorTrigger script allows attachment of a variety of event triggers to sensor behaviour for easy gameplay integration.
- Package includes prefabs for Canvas UI meters that are fully customizable and configurable to different sensor representations to fit any project. Also includes example scene with all sensor types running.

TABLE OF CONTENTS

1. INITIAL SETUP

A IMPORTING PACKAGE

In order to use the EMG Sensor Plugin you must import the *unitypackage* file it to a new or existing project.

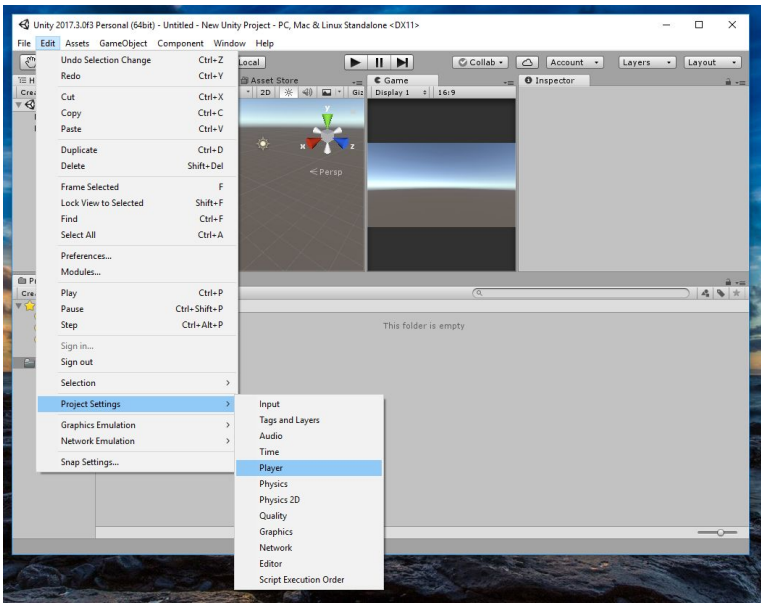
The *unitypackage* file can be found **here**.

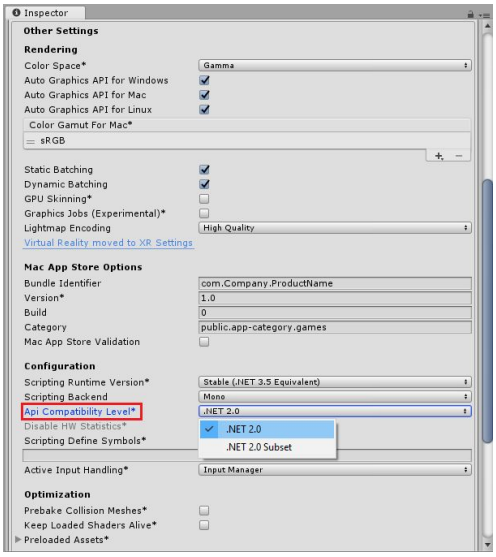
Importing the project requires Unity version 4 or higher, however it is tested in Unity 2017.3.0f3 and a newer or older version may have minor errors regarding unknown or obsolete keywords.

The directions for initial setup are:

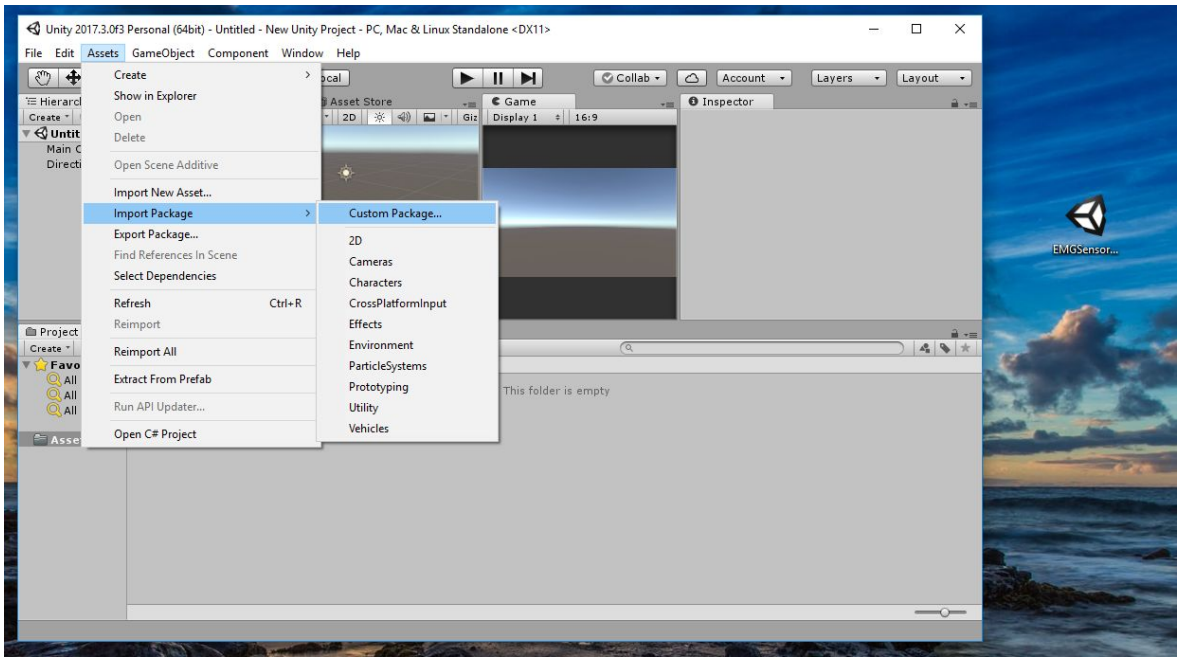
1. First, the .NET subset of the project needs to be changed. In the editor, navigate to *Edit -> Project Settings -> Player*.

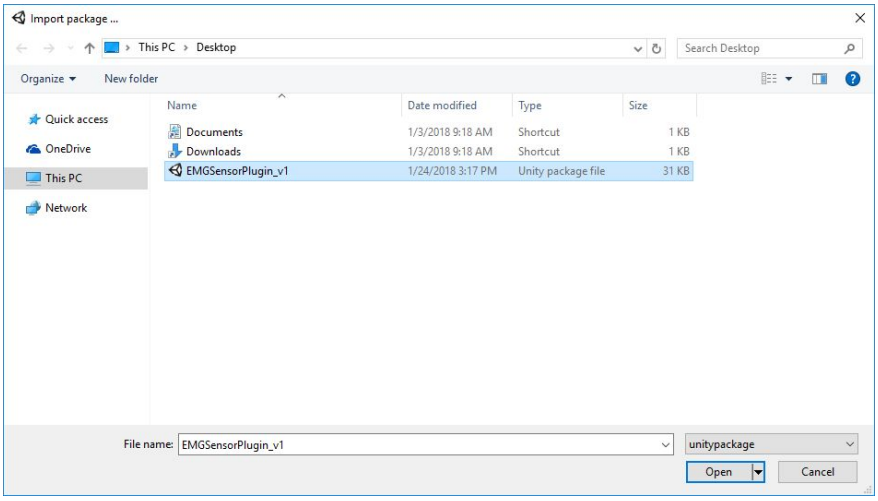
2. In the *PlayerSettings* inspector window that appears, scroll down to the *Api Compatibility Level* and ensure it's set to **.Net 2.0**, not the Subset.





3. To import, in your Unity project navigate to *Assets -> Package -> Custom Package...*

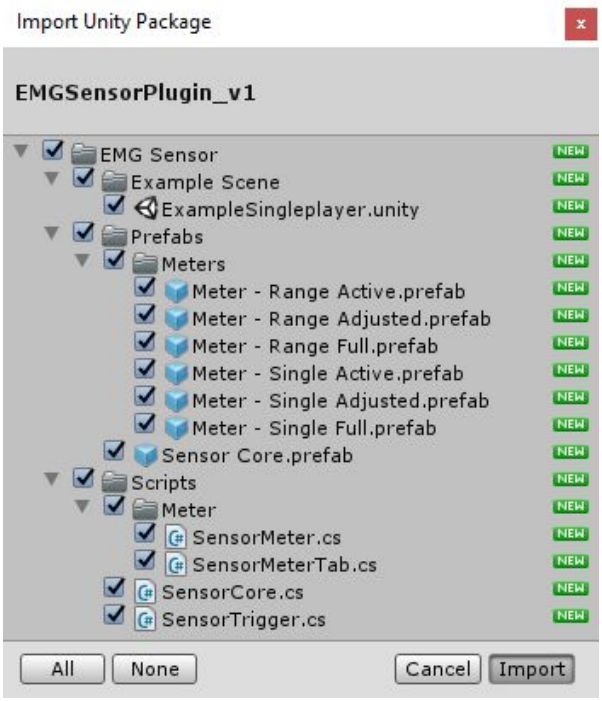




4. In the pop-up window, navigate to the *unitypackage* file and select *Open.*

Alternatively, you can drag the *unitypackage* file into the *Assets* panel in the Unity editor.

5. The resulting pop-up lists the contents of the *unitypackage* file. Ensure that all of the items are checked, or select the *All* button. Then, click *Import*.

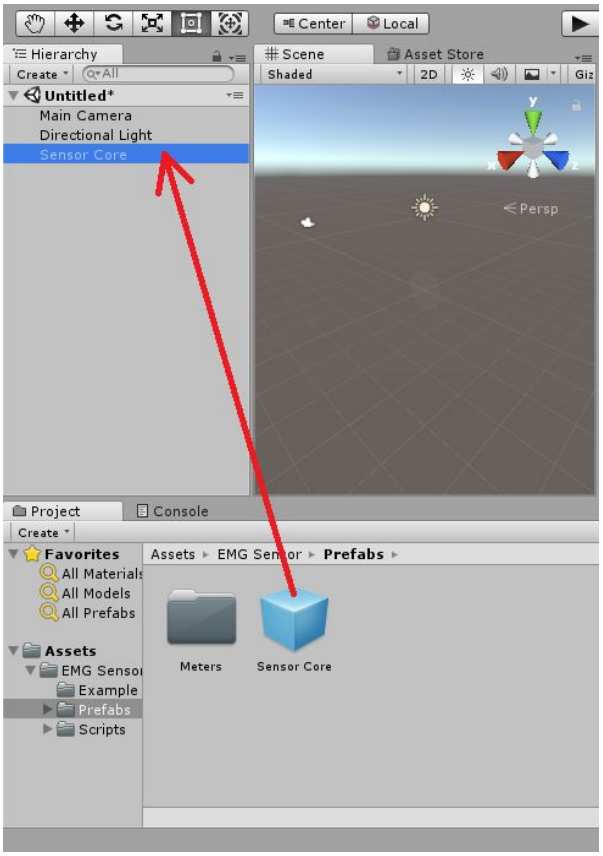


The project's folder will now contain the base folder *EMG Sensor* and its subfolders which contain everything you need.

An example scene is contained in:
EMG Sensor/Example Scene

B INITIAL SETUP

Adding the SensorCore to a scene is simple:



Navigate to the folder *EMG Sensor/Prefabs* and drag the *Sensor Core* prefab object into the Scene window or Scene Hierarchy panel. It will appear in the Scene Hierarchy panel with blue text.

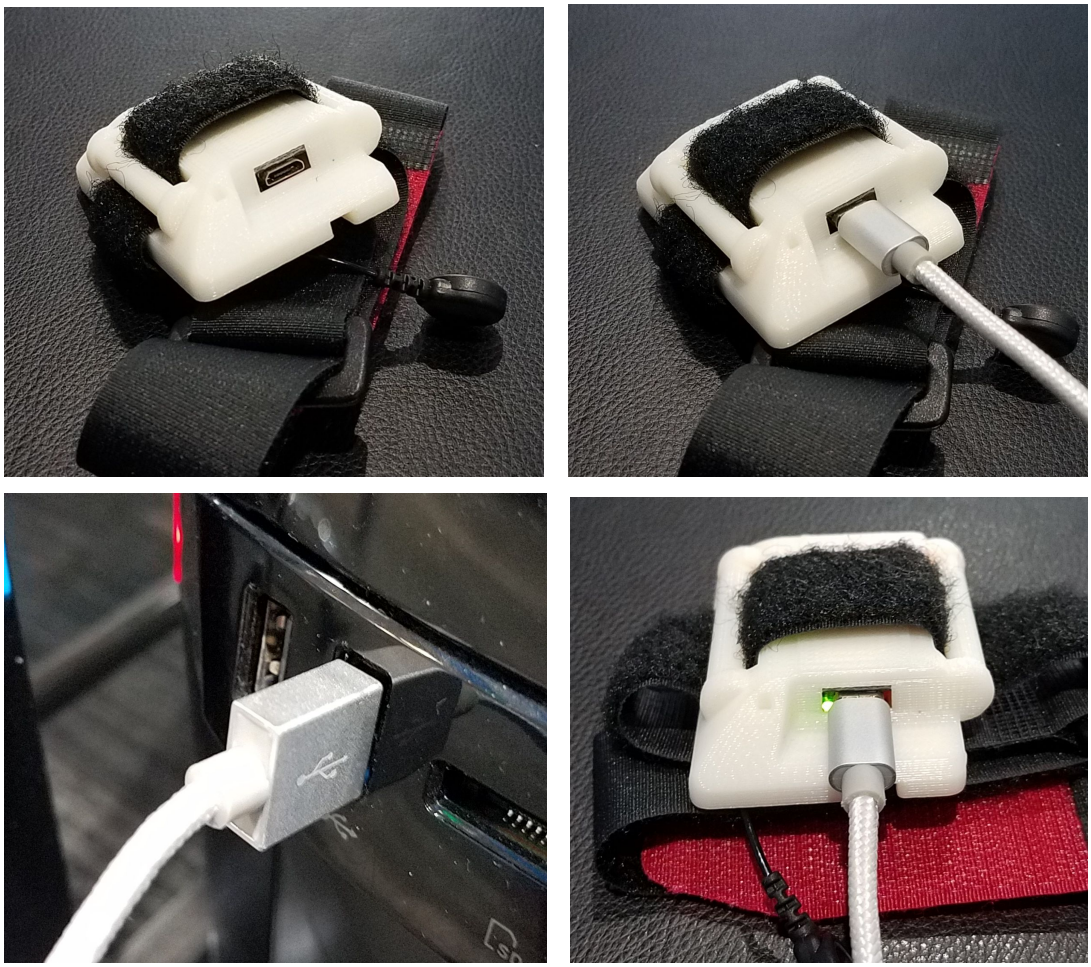
...That's it! The SensorCore with default settings is now in your project and running.

By default it will automatically connect to a USB Arduino EMG sensor and record its readings. The SensorCore will also not be destroyed when going between scenes, and will replace any existing SensorCore in the scenes that it survives in to. Therefore, you can freely add a SensorCore to each scene in the project.

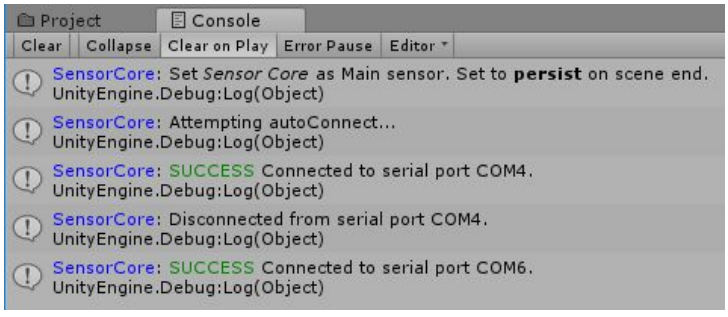
C TESTING SENSOR

The next step is to test the SensorCore to ensure that it's reading input from the arduino EMG sensor.

1. Gather the Arduino EMG sensor and a microUSB cable. Plug the microUSB into the sensor then connect it to any working USB port on the project computer. Ensure that the sensor's LED lights are turned on.



2. Inside of the Unity scene with the *SensorCore* script added to the scene hierarchy, press the **Play** button in the editor and review the Console panel when connecting and disconnecting the sensor. Due to the nature of the arduino sensor it may require being removed and plugged in again to be detected. Test different USB ports on the



PC. If no connections are made, try restarting Unity.

---

2. SENSOR CORE

A SENSOR CORE OVERVIEW

The **SensorCore** is the primary script in the EMG Sensor package. It contains the Arduino communication code (firmata) and connection methods. The analog **raw reading**, which is an integer number between 0 and 1024, is accessed by the SensorCore each frame, stored into a public variable, and interpreted against sensor attributes like a flex or rest threshold. All sensor events and anything that is based on sensor input will be somehow polling that reading from the SensorCore.

As such, it is required to be inside of any scene that needs to use the sensor at all. By default the script is a **Singleton**, which means that the same instance of SensorCore will survive when a new scene is loaded. Any other SensorCore that loads in will be automatically destroyed if there's an active surviving SensorCore. Therefore, only a SensorCore at the earliest necessary scene is strictly required, but it's recommended to have a SensorCore in each scene to allow for isolated testing of that scene. The active SensorCore can be accessed statically (without reference from any project script) through `SensorCore.Main` .

The SensorCore also has a smoothing algorithm which dampens sudden changes in the sensor reading and enables better flex emulation through a button press in ranged mode (where the distance between the meter thresholds is important). The smoothing magnitude is the speed at which the smoothed meter moves towards the raw value, with the smoothing curve naturally slowing down the magnitude as it approaches the raw value.

B PUBLIC VARIABLES

VARIABLE	VALUE	DESC.
`Main`	**SensorCore**	**The currently active static instance of SensorCore.**
`smoothingMagnitude`	Float [0-1]	The magnitude of smoothing on the adjusted meter reading. Larger smoothing catches up to the raw reading more quickly.
`smoothingCurve`	Float [0-1]	When smoothing, adjusts the effect of distance on the smoothing to curve the approach. A higher curve value slows down on proximity to the raw reading, while a low curve linearly approaches the raw with little curving. Use a higher curve for smaller sensor movements.
`sensorAdjustedCap`	Float [0-1024]	The maximum value that the sensor reading will display in adjusted meter mode. This is used to narrow the sensor focus to only the values that will be used by the sensor (they tend to stay below 400).

VARIABLE	VALUE	DESC.
`flexThreshold`	Float [0-1]	The position of the Flex Threshold of the meter. A reading above the threshold is considered a maximum flex in single or range modes.
`restThreshold`	Float [0-1]	The position of the Rest Threshold of the meter. A reading below the threshold is considered a rest in range modes.
`flexKeys`	String array	Array of keys which cause the meter to flex. Smoothing recommended in range mode.
`isConnected`	Boolean	If true, the SensorCore is connected to and reading from an Arduino sensor in a COM port.
`isFlexEmulating`	Boolean	If true, designated key-presses in `flexKeys` are being accepted as a way to max out the sensor (flex).
`readingRaw`	Float [0-1024]	The raw pin value taken from the sensor. Usually only used when testing the sensor reading, as it is unlikely that the full range of 0 to 1024 will be of sufficient accuracy.
`readingRawSmoothed`	Float [0-1024]	The ratio of the sensor reading, when smoothing is applied. This value chases the ratio of the unsmoothed raw value according to the smoothing attributes.
`readingAdjusted`	Float [0-1]	The adjusted ratio of the sensor when bound to the sensorAdjustedCap for accuracy. A value of 0 is empty and a value of 1 is at the sensorAdjustedCap. Usually used when adjusting the sensor meter's thresholds or checking the reading.
`readingAdjusted Smoothed`	Float [0-1]	The adjusted ratio of the sensor when bound to the sensorAdjustedCap for accuracy, with smoothing applied according to the smoothing attributes.

C PUBLIC METHODS

These methods can be called at any time with a reference to the SensorMeter, or by using `SensorCore.Main` as a public static reference to the active SensorCore instance.

METHOD	PARAMETERS	RETURNS	DESC.
`isFlexing`	`bool isSmoothed`	Boolean	Returns true if the sensor reading is at or above the flex threshold. `isSmoothed` determines whether the reading used is the raw or smoothed reading (default `true`).
`isResting`	`bool isSmoothed`	Boolean	Returns true if the sensor reading is at or below the rest threshold. `isSmoothed` determines whether the reading used is the raw or smoothed reading (default `true`).
`GetFlexFull`	`bool isSmoothed`	Float [0-1]	Returns the sensor reading percentage with a max of 1024. `isSmoothed` determines whether the reading used is the raw or smoothed reading (default `true`).
`GetFlex Adjusted`	`bool isSmoothed`	Float [0-1]	Returns the sensor reading percentage with a max of `sensorAdjustedCap`. `isSmoothed` determines whether the reading used is the raw or smoothed reading (default `true`).
`GetFlexActive`	`bool isSingle` `bool isSmoothed`	Float [0-1]	Returns the sensor reading percentage with a max of `flexThreshold`. If `isSingle`, has min value of restThreshold. `isSmoothed` determines whether the reading used is the raw or smoothed reading (default `true`).
`EmulateFlex Frame`	--	--	When called sets the raw reading to 1024, bypassing the actual sensor reading. This emulates a maximum reading for the desired frame only, which is used to emulate a flex. A continuous emulated flex requires this called each frame.

D SENSOR MODES

The *SensorCore* and through extension the *SensorMeter* break sensor data down into six modes for gameplay derivation, with each being optionally smoothed. They're based on two points of variance: the threshold type (count) and the range focus.

The first threshold type is **Single** meaning it only compares the sensor reading against a single point: the flex threshold. This is used for games with a single input used by entering (going past the flex threshold), exiting (lowering back down below it), and remaining in the restThreshold zone. While getting a flex percentage up to and beyond the single threshold is possible, the nature of the individual player and sensor hardware make a consistent gauge of the lower limit of the sensor reading impossible to accurately predict, for example. Therefore only flex threshold event triggers are recommended in single mode.

The second threshold type is **Range** which uses both a flex threshold and a rest threshold. A ranged sensor mode allows the range between the configurable rest and flex thresholds to contain usable analog gameplay information with discrete event triggering. It is able to guarantee data usability inside of the range by establishing both a floor and a ceiling customized to the indivdual and hardware. This is a more complex but far more diverse input set which can guage a percentage of flex inside of the focus zone, rest events and flex events each (enter, exit, and states). Having a game with several actions being performed by the sensor with additional input will likely require a range mode sensor.

The range focus of the meter also determines how the reading is interpreted by the game. There's three types of sensor range focuses: full, adjustable, and active.

- **Full**: The sensor has a maximum reading of 1024, and a full range focus compares the sensor reading (raw or smoothed) against the maximum value. This range is rarely useful for gameplay as each individual and hardware has differences in what ranges they feasibly reach, which will rarely ever approach the sensor maximum.
- **Adjusted**: Adjusting the range adds a _Sensor Adjusted Cap_ which sets a new maximum value for the sensor. The sensor can still generate a reading above the cap, but it will only be seen as the cap value when polled inside this mode. The adjusted mode is best for showing the sensor range for the purposes of adjusting the flex or rest thresholds, as it allows more accurate manipulation and visualizion by cutting out the unused ranges of the sensor meter. It's recommended to allow the option of dynamically adjusting the sensor cap to allow for abnormally high or low sensor ranges.
- **Active**: The active focus is best for deriving gameplay as it compares the sensor reading against the desired thresholds. In single mode the active focus begins at 0 and has a maximum value of the flex threshold. In range mode the minimum is the rest threshold and the flex is similarly the maximum vlaue. The result is that the player flexes to "fill the meter" and generate a flex action, or rests below the rest threshold to be considered resting (empty meter) in ranged mode. Active mode is best for deriving and visually representing the sensor as a gameplay element as readibly as possible.

3. SENSOR TRIGGER

A SENSOR TRIGGER OVERVIEW

The *SensorTrigger* script can be added to a gameObject that contains a *SensorCore*. In its inspector pane the designer can drag and drop public method references from other scripts and have those methods be called when a sensor state is triggered, just like a Unity *EventTrigger* component.

For example, a *PlayerController* action like "`Jump()`" could be dragged into the `OnFlexEnter` trigger on the *SensorTrigger* inspector. Then wach time the sensor detects a new flex it will automatically call the `Jump` method inside of *PlayerController*.

The goal is to more easily trigger methods to expedite porting and ease of development without requiring the designer to manually check the SensorCore readings against the thresholds.

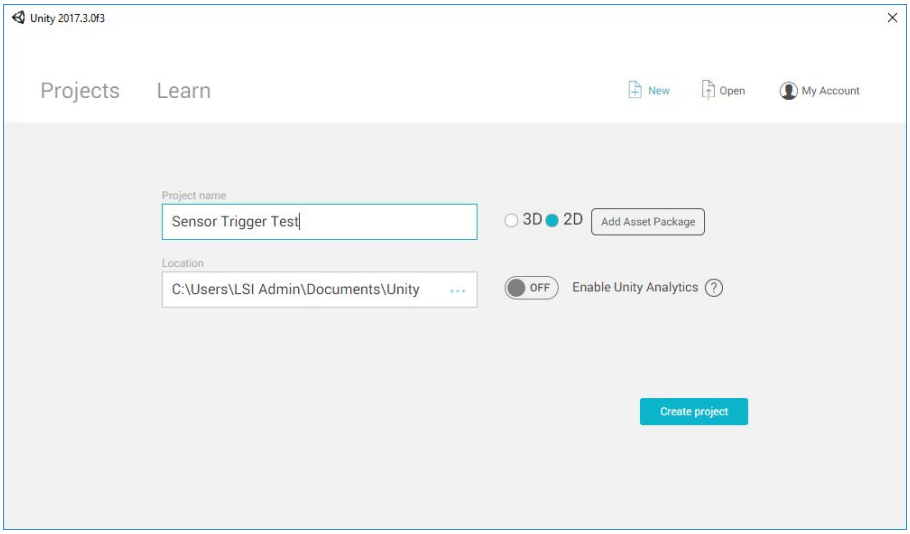
The supplied trigger events that methods can be added to are:

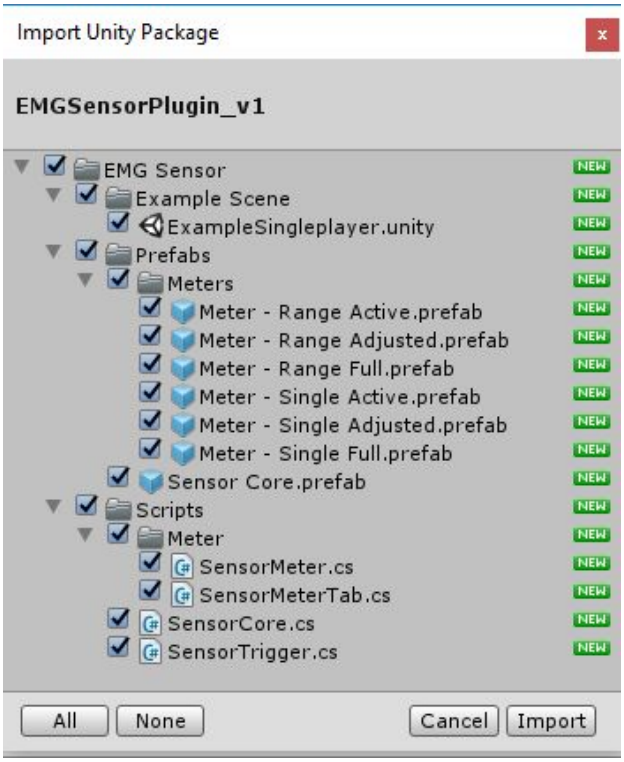
- **`OnDownturn:`** Called when the sensor reading decreases after previously increasing.
- **`OnUpturn`**: Called when the sensor reading increasing after previously decreasing.
- **`OnFlexEnter`**: Called when the sensor reading reaches or goes past the flex threshold after previously being below it.
- **`OnFlexExit`**: Called when the sensor reading goes below the flex threshold after previously being at or above it.
- **`OnRestEnter`**: Called when the sensor reading goes at or below the rest threshold after previously being above it.
- **`OnRestExit`**: Called when the sensor reading reaches above the rest threshold after previously being at or below it.

B SENSOR TRIGGER EXAMPLE

Setting up an event to trigger with the *SensorTrigger* script is simple. Below we'll make a new project that includes a *SensorCore* and *SensorTrigger* which changes the color of an image element on flex.

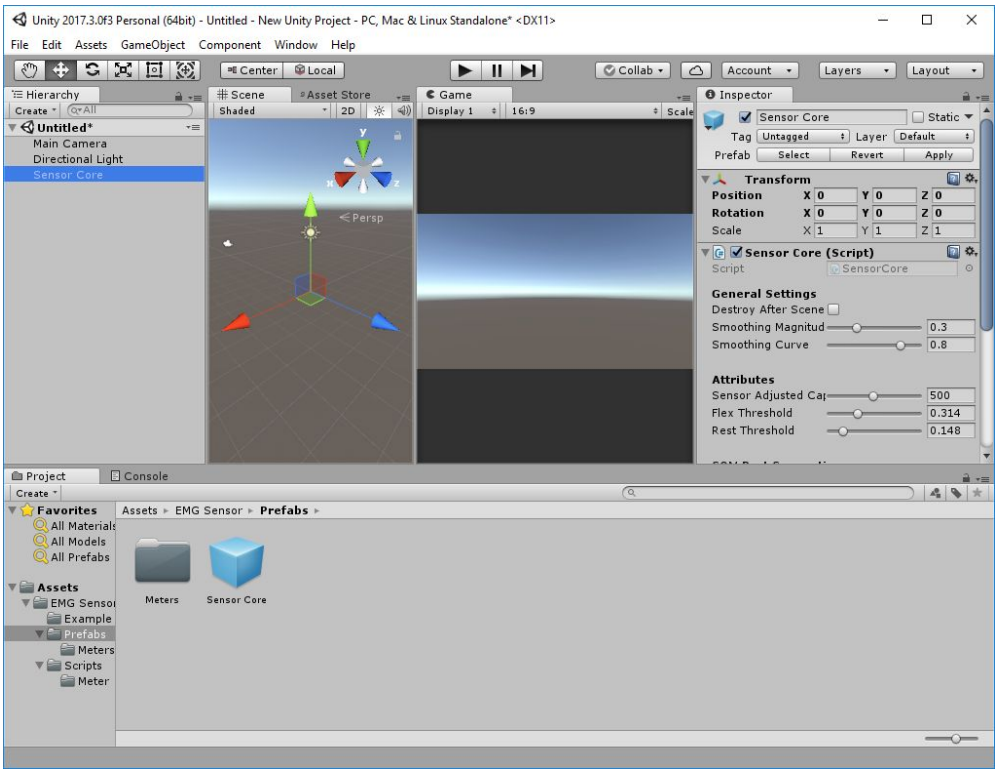
1. Start a blank Unity Project.





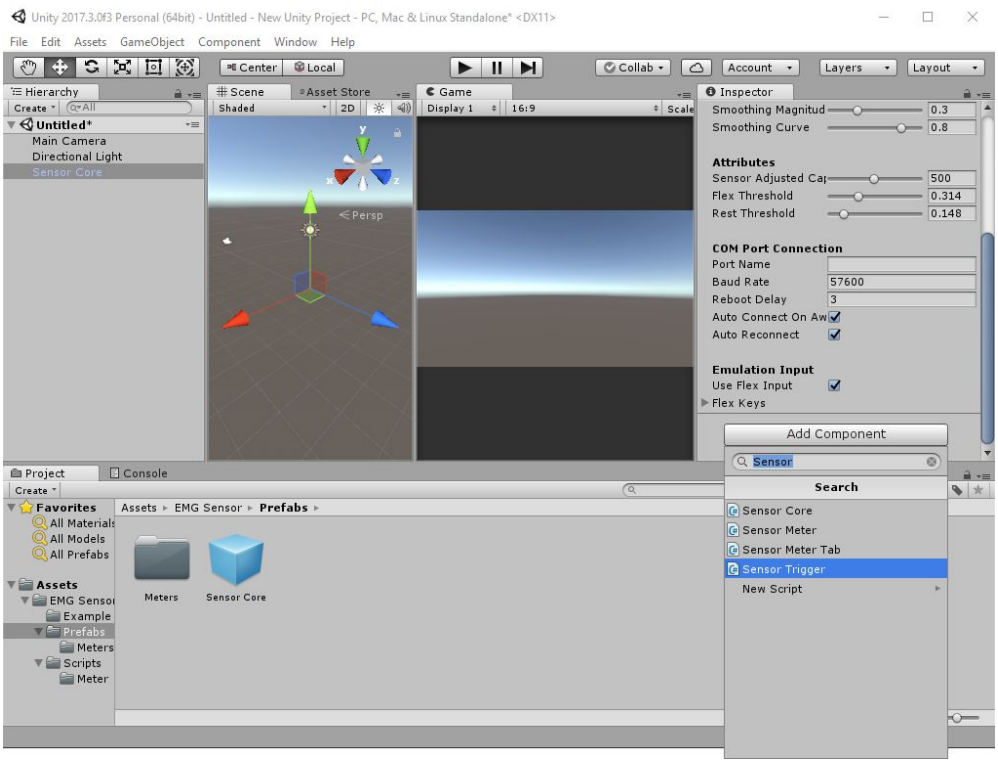
2. Follow the instructions in section 1.A (Importing Package) to import the *SensorCore* package.

3. Drag a Sensor Core object from the Assets/EMG Sensor/Prefabs folder into the scene heirarchy.

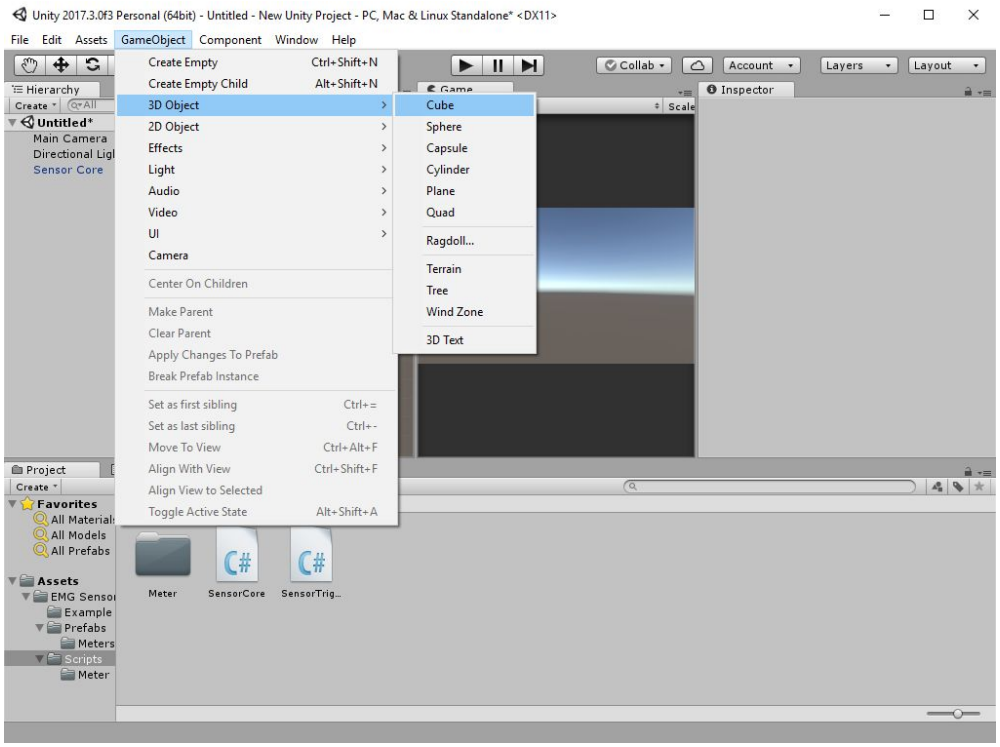


4. In the Sensor Core object's inspector pane, scroll down and click the button that reads "Add Component". In the window that appears' searchbar, search for the Sensor Trigger script and click it to add it to the object.

Alternatively, you can drag the SensorTrigger script from Assets/EMG Sensor/Scripts into the inspector.

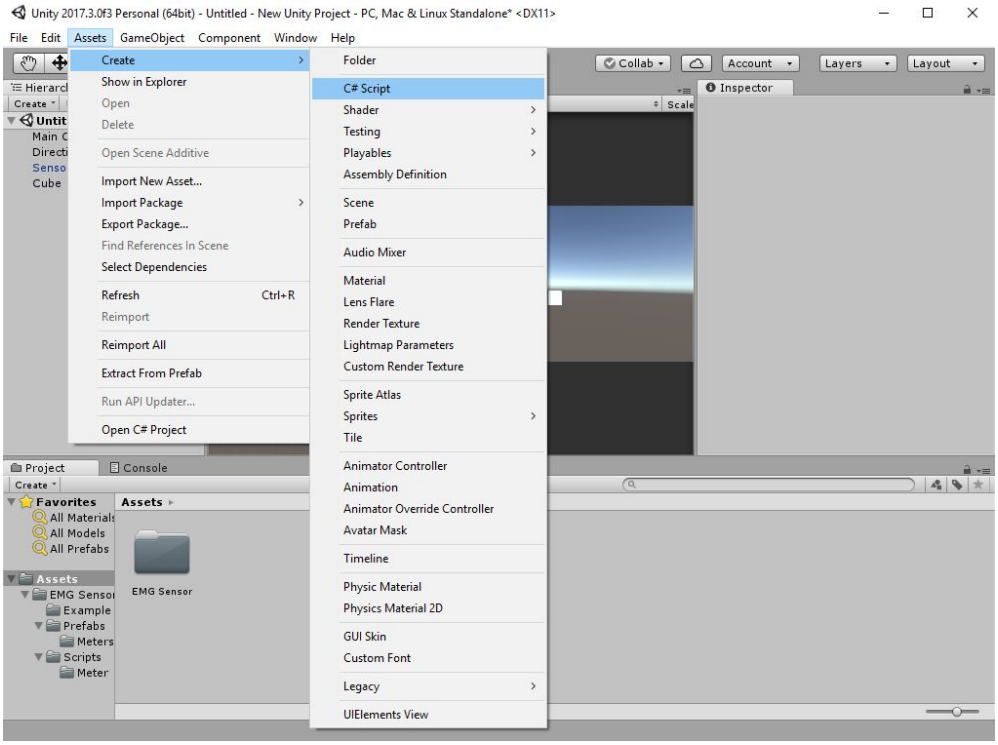


5. Add a new 3D cube primitive to the scene by navigating to GameObject -> 3D Object -> Cube, or by right clicking in the scene heirarchy and selecting 3D Object -> Cube.



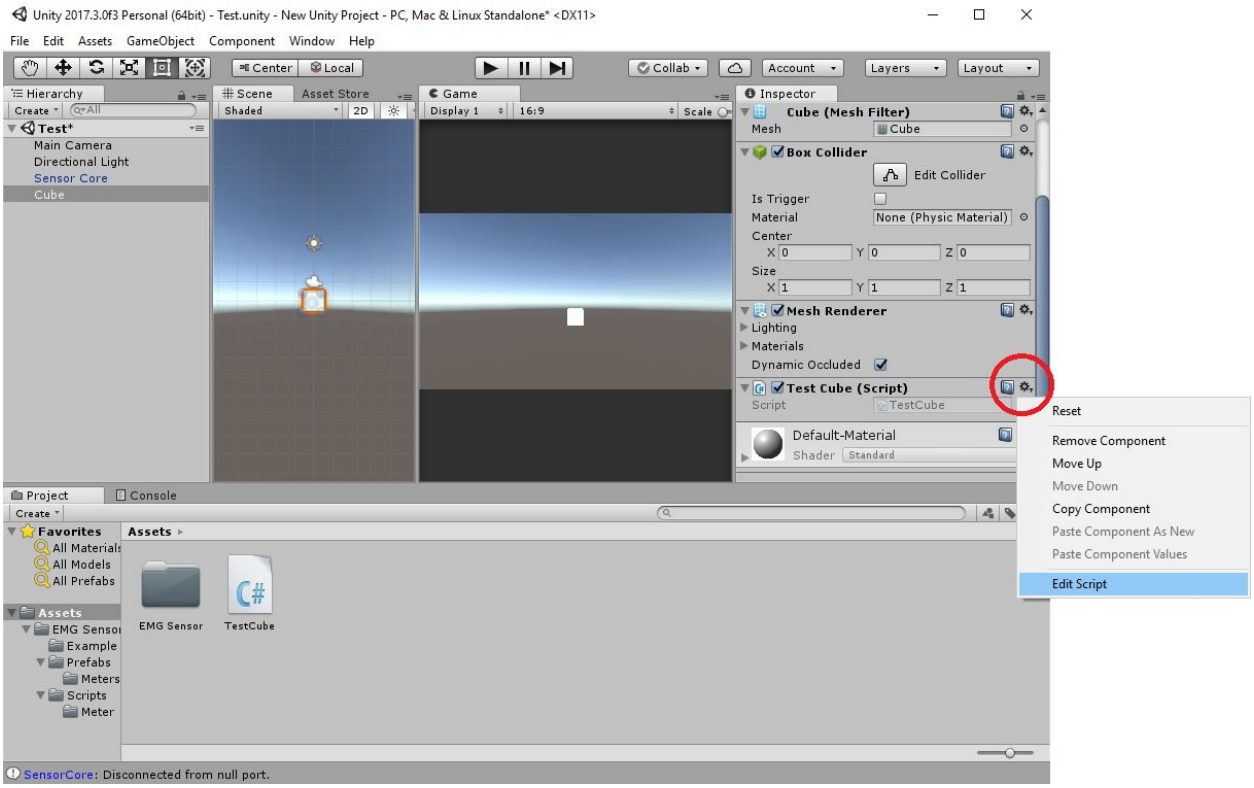
6. Add a new C# script by navigating to Assets -> Create -> C# Script, or alternatively by right clicking in the Assets pane and selecting Create -> C# Script.

Name it "TestCube" if you'd like.



7. Add the TestCube script to the new 3D Cube object in the scene (see step 4).

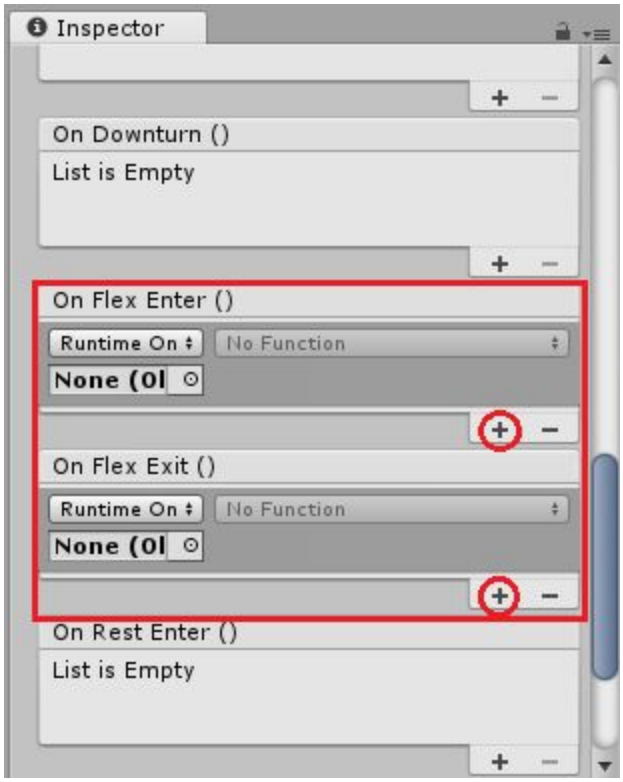
8. Open the TestCube.cs script by clicking the gear icon on thr top right of the newly added TestCube component in the Cube object's inspector pane, then selecting "Edit Script." Alternatively, double-click the TestCube.cs script in the Assets pane.



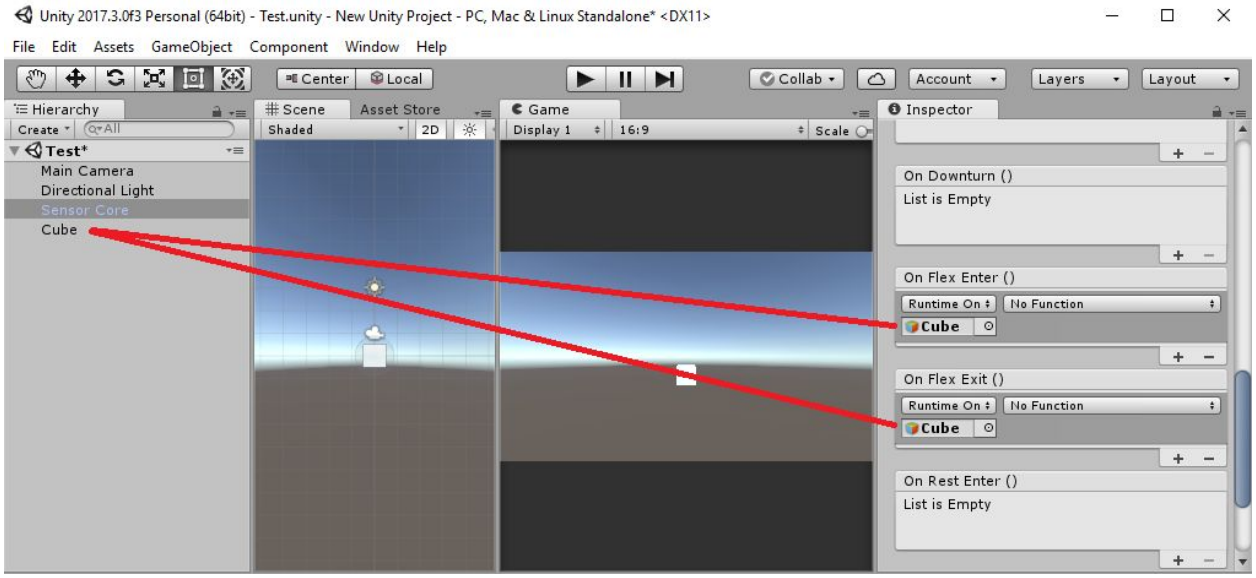
9. In the script editor that appears (Visual C++ or Monodevelop typically, it does not make a difference to this example), edit the script to include the following:

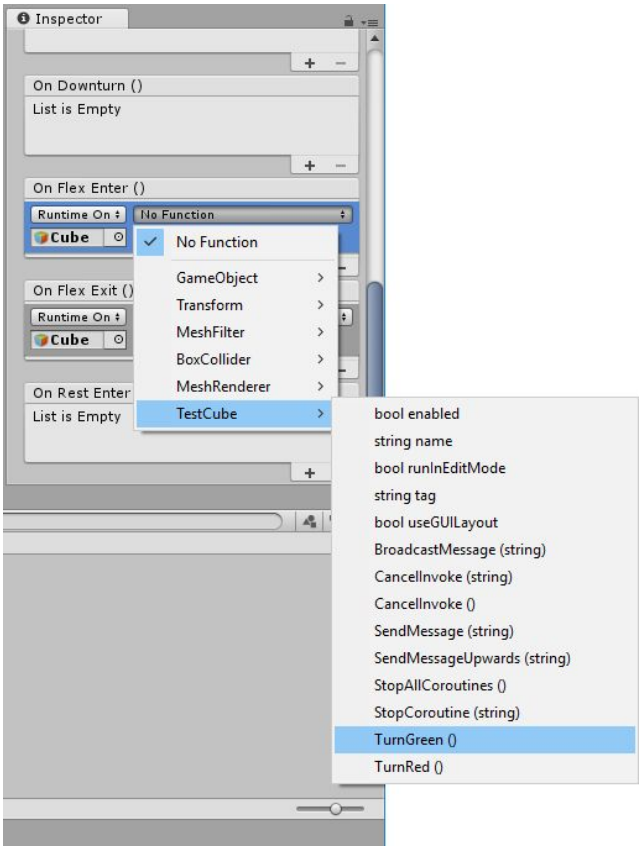
```
public class TestCube : MonoBehaviour
{
	private MeshRenderer refRenderer;

	void Start ()
	{
		refRenderer = GetComponent<MeshRenderer>();
	}

	public void TurnGreen()
	{
		refRenderer.material.color = Color.green;
	}

	public void TurnRed()
	{
		refRenderer.material.color = Color.red;
	}
}
```

10. Select the Sensor Core object in the heirarchy, which should have a SensorTrigger script component attached in step 4. In the inspector pane, in the SensorTrigger component, add two new event triggers for **OnFlexEnter** and **OnFlexExit** triggers by clicking the + (plus) button at the bottom of the respective event subpanes.



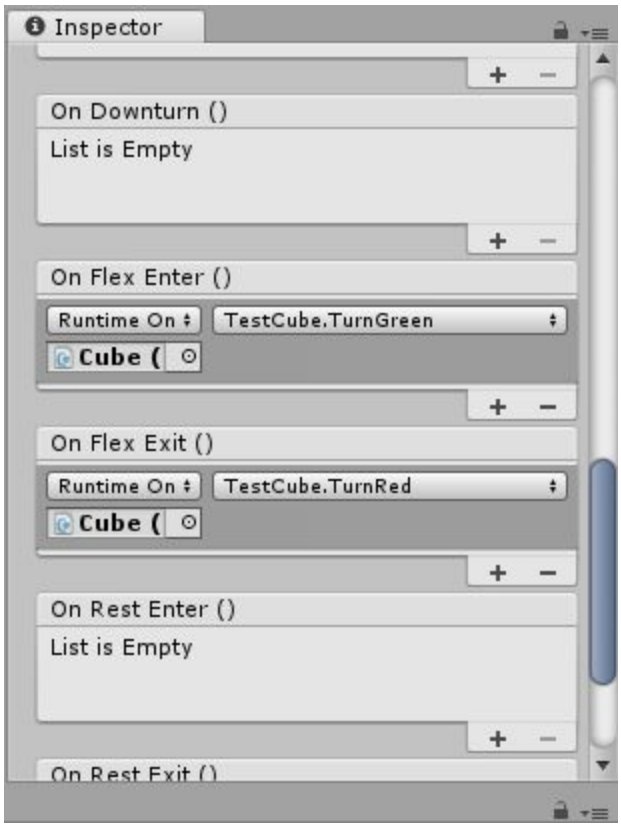
11. In the object reference space, for each event subpane, drag the Cube object from the heirarchy pane into the space to give each event a reference to the Cube object.

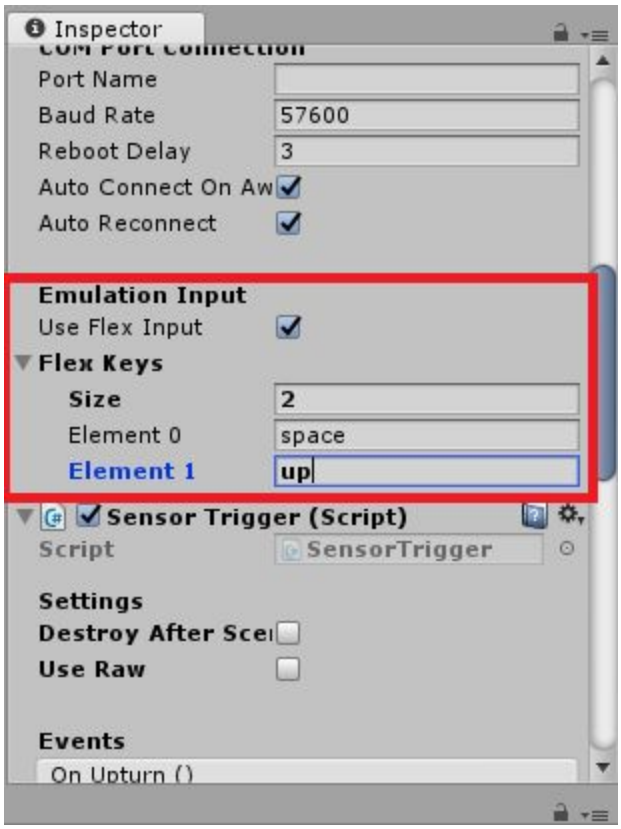




12. In the function selection drop down for **On Flex Enter()** choose TestCube -> TurnGreen ().

13. In the function selection drop fown for **On Flex Exit()** choose TestCube -> TurnRed ().



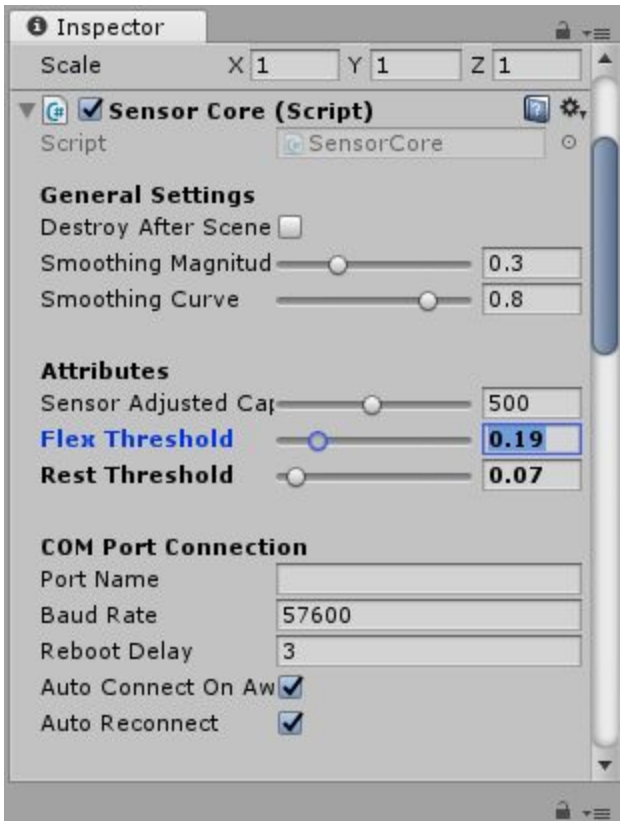


14. To prepare flex input, either plug in an arudino sensor (see Initial Setup guide in 1.b for instructions) or set up input emulation in the SensorCore component in the insepctor pane (see left).

By default *Use Flex Input* is set to true, and Flex Keys includes one key: spacebar.

To add an additional Flex Key, drop down the Flex Keys sub menu and change the *Size* field to the desired legnth. Then, add the Unity Key name of the key or button desired. In the example on the left, the up arrow key was added to the Flex Key array.

15. That's it. Press the play button on at the top of the Unity editor to play the scene in the Game window. Flexing up to the flex threshold on the sensor (ensure it is set appropriately in the SensorCore inspector pane using the FlexThreshold slider, see right) or pressing the flex emulation keys will trigger the Cube to turn green or red appropriately.



Any public script function can be added to the SensorTrigger event and be used to call functions in a scene based on SensorCore reading state.

Experiment with the other sensor triggers to get a feel for the sensor's response and timing when flexed.

---

4. SENSOR METER

A OVERVIEW

Additional scripts **Sensor Meter** and **Sensor Meter Tab** are included as part of the EMG Sensor plugin to provide additional development ease and portability. The SensorMeter can be configured in multiple modes, supports both single and ranged modes, and isdesigned to be both an example and a customizable framework for creating or inspiring other visualizations.

The example scene "*ExampleSensors.unity*" contains a set of 6 sensors to showcase how they interpret the sensor differently.

It also includes Sensor Meter Tab script-enabled buttons to have a simple working implementation of a directional and mouse-draggable adjustment slider for the rest and flex thresholds.

Ideally, a game that wants to quickly use the SensorCore could drag in the core and have arudino support, then drag in a SensorMeter for an adjustment scene (using SensorMeterTab) and one for gameplay, both of which will automatically be capable of representing the sensor data.

The EMG Sensor Package includes the siz example sensors as prefabs in the file directory.

B MODES

A SensorMeter component works by generating a float value between 0 and 1 inclusive, which represents a ratio of the sensor's reading compared against a certain maximum.That ratio can then be used by a gameObject to scale a visual element. It can be configured to 6 modes which represent how that value is derived. All of the meter modes are affected by the *smoothed* attribute, which uses the smoothed SensorCore reading to generate the meter (on by default).

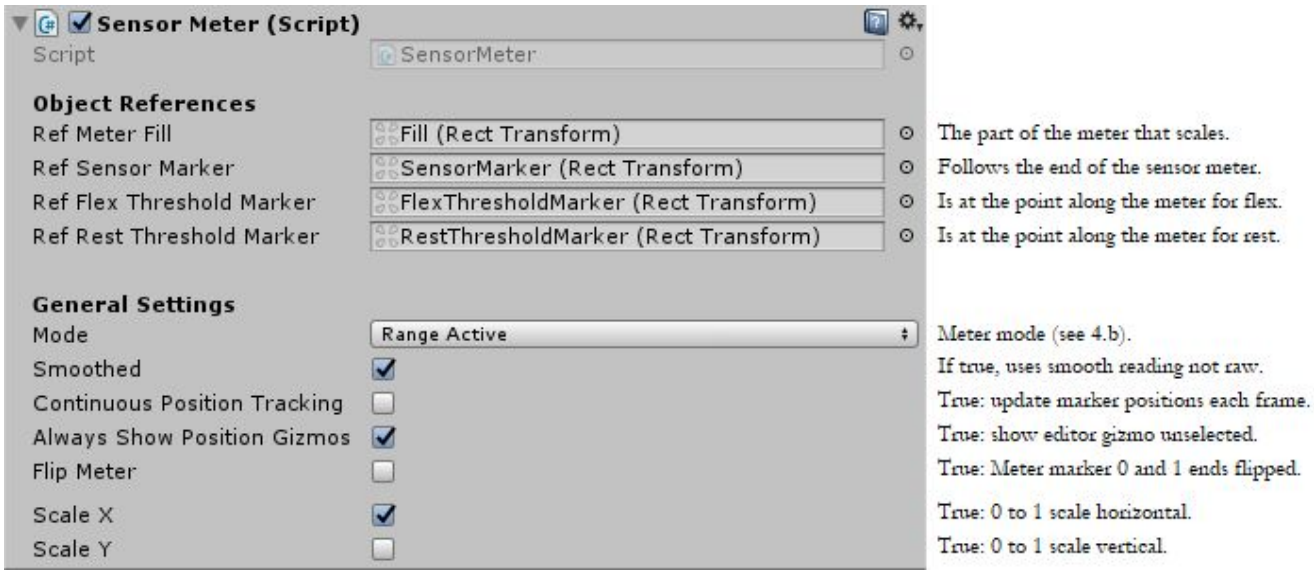
The modes are:

- **`Single Full:`** Visualizes the entire sensor range from 0 to 1024. Can display the flex threshold's absolute position along the sensor range. Useful for sensor testing.
- **`Single Adjustable`**: Sets the ratio value from 0 to the SensorCore's sensorAdjustedCap value as the maximum. Any reading above the cap remains at a full bar (value of 1). Displays the flex threshold, which is unable to be above the sensorAdjsutedCap. Most useful for allowing the player to adjust the threshold value (using the Sensor Meter Tab, for example). Usually put alongside an active mode meter.
- **`Single Active`**: Sets the value ratio from 0 to the flex threshold. In this mode a filled bar represents a flex. The flex threshold, being the cap, cannot be visualized along the meter in this mode for the purposes of adjusting it's value. However, this is the best the mode for a single-point threshold game where the game's sensor action is determined by flexing (e.g. "filling the bar").
- **`Range Full:`** Visualizes the entire sensor range from 0 to 1024. Can display the rest and flex threshold's absolute position along the sensor range. Useful for sensor testing.
- **`Range Adjustable`**: Sets the ratio value from 0 to the SensorCore's sensorAdjustedCap value as the maximum. Any reading above the cap remains at a full bar (value of 1). Displays the rest and flex thresholds, which are unable to be above the sensorAdjsutedCap. Most useful for allowing the player to adjust the threshold values (using Sensor Meter Tabs, for example). Usually put alongside an active mode meter.
- **`Range Active`**: Sets the value ratio with a minimum sensor value of the rest threshold and a maximum of the flex threshold. Therefore any sensor reading at or below the rest threshold is a value of 0 (empty bar) and any sensor reading at or below the flex threshold is a value of 1 (full bar). Most useful for displaying gameplay after the sensor thresholds have been adjusted, as having the rest and flex be the ends of the meter make adjusting them impossible to visualize.

C CONFIGURATION

The **SensorMeter** has several configurable properties in its inspector and is designed with ease of customizability (showcased in the prefab's heirarchy).

Script Inspector Controls:



Prefab Heirarchy:

The heirarchy for the meters is where the visual customization comes in.

- *Outer / Inner* are only there as an example and be removed or replaced.
- *Fill* can be any scaleable element. Set pivot accordingly (see below).
- The three markers (*flex, rest, sensor*) are optional position markers.
    - Set as references in SensorMeter (above).
    - Can be renamed, changed, not used, or added to freely.

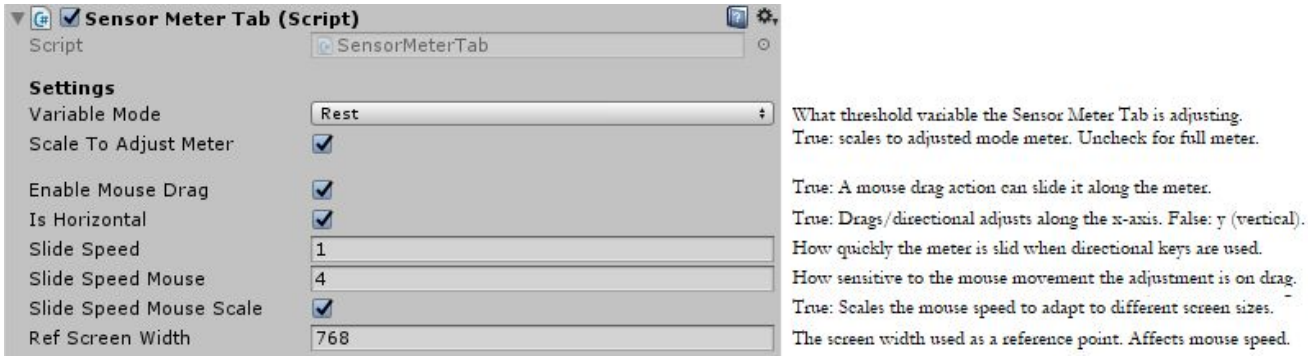
Fill Element Considerations:

The meter fill is the element that the sensor will scale from 0 to maxScale. That comes with a few considerations, however:

- The fill meter can be any scale in x, y, or z, not just a scale of 1, but it must be at the scale desired to be considered "full" in the scene because that scale is recorded to be the maximum at scene start.
- The "Pivot" options of the rectTransform must be set to reflect the direction that the meter scales in.
- For a 3D meter the fill should be parented to an empty gameObject that's at the desired pivot.

D METER TAB

SensorMeters can be extended with a **Sensor Meter Tab**-enabled marker. The markers need to be children of the corresponding threshold marker (if the meter is in 'Rest' variable mode then it must be parented to the *restThresholdMarker* that is referenced in the *SensorMeter* inspector). Then, dragging or moving the corresponding default direction axes (horizontal/vertical) while selected, will adjust the threshold and simultaneously update it's position to show the change like a slider.



Just like the other elements of a *SensorMeter,* a *SensorMeterTab* is designed to be customizable. It utilizes a standard Unity *UI.Button* element (meaning it must be used on a *Canvas* meter, not a 3D meter) which can be cistomized with button events and customized transitions. Any number of children parented to the marker will also move with the marker to visualized the approprite threshold and can provide audio, particles, animations, etc.