Open Source VR Project

build passing API Documentation

OSVR Gesturing Installation Guide

> Requires the Unity software version 2019.1 (or above).
>
> Requires the Oculus Integration plugin from the Unity Asset Store.

Getting Started

Documentation

Visit Full Documentation for a detailed API

Setting up the project

- Create a new project in the Unity software version 2019.1 using 3D Template or open an existing project.
- Ensure `Virtual Reality Supported` is checked:
  - In the Unity software select `Main Menu -> Edit -> Project Settings` to open the `Project Settings` window.
  - Select `Player` from the left hand menu in the `Project Settings` window.
  - In the `Player` settings panel expand `XR Settings`.
  - In `XR Settings` ensure the `Virtual Reality Supported` option is checked.
- Ensure the project `Scripting Runtime Version` is set to `.NET 4.x Equivalent`:
  - In the Unity software select `Main Menu -> Edit -> Project Settings` to open the `Project Settings` inspector.
  - Select `Player` from the left hand menu in the `Project Settings` window.
  - In the `Player` settings panel expand `Other Settings`.
  - Ensure the `Scripting Runtime Version` is set to `.NET 4.x Equivalent`.
- Download `Oculus Integration Plugin`
  - Navigate to the Unity Asset Store (`Ctrl+9`)
  - Search for `Oculus Integration`
  - Download the package and import the assets.
- Download the `OSVR Gesture Framework` asset from the Unity Asset Store
  - If the Gesture Framework is downloaded before Oculus Integration, various scripts will not be loaded correctly
  - Open the `Gestures/ExampleScenes/GestureDemoScene` scene.

Controls

The system requires a concrete implementation of IController to track positions and determine when a Gesture should be active. In the example scene, the GameObject "Right Hand" holds a `TouchController` component, which works for Oculus Rift Touch Controllers. In the Inspector, the option `controllerType` will let you choose which hand is used, and the option `gestureActiveButton` lets you choose which button must be held to activate the Gesture. The defaults are RTouch (Right Touch Controller) and Primary Index Trigger (the trigger resting under the index finger of the RTouch Controller).

Getting Started

In the `TrackerSetup.cs` file, code is written to demonstrate how to set up the Gesture tracking

```
GestureMonitor tracker = gameObject.AddComponent<GestureMonitor>();
```

Create the Gesture Monitor Component responsible for tracking and detecting Gestures

```
tracker.controller = GetComponent<TouchController>();
```

Set the instance of an IController to track.

```
tracker.lineRenderer = GetComponent<LineRenderer>();
```

Attach a LineRenderer if wanted

```
tracker.AddGesture("Square", new SquareGesture());
```

Add Gestures to be tracked

```
tracker.AddGestureCompleteCallback(GestureComplete);
```

Set what happens when a Gesture is completed

```
tracker.AddGestureFailedCallback(GestureFailed);
```

Set what happens when a Gesture is failed

```
tracker.AddGestureStartCallback(GestureStart);
```

Set what happens when a Gesture is started

```
tracker.setEnabled("Square", true)
```

Although Gestures are enabled by default, use this method to toggle the tracking state of individual Gestures

Building Gestures

Gestures are defined in the XY plane, and consist line segments, arcs, and points, called "Checks". When the user draws a shape, each of these Checks will be queried with the positional data passed in, to determine whether the Gesture has been completed or not. Currently, creating Gestures much be done in code, there is not a solution to do so in the Unity Editor or Unity Inspector.

```
new Gesture()
```

Start with a base Gesture

```
new Gesture().AddOnceChecks(new List<Check>{})
```

Add a series of Checks where each segment much only be satisfied once

```
new Gesture().AddSequentialChecks(new List<Check>{})
```

Add a series of Checks where each segment much be passed in a specified order

```
new LineCheck(new Vector3(0,0,0), new Vector3(1,1,0), .5f)
```

Create a new Line segment check that goes from (0,0,0) to (1,1,0) and has a tolerance of .5f

```
new RadiusCheck(new Vector3(0,0,0), .5f)
```

Create a new Radius check centered at (0,0,0) with a radius of .5f.

```
new ArcCheck(new Vector3(0, 0, 0), 90, new Vector3(0,1,0))
```

Create a new Arc check that traces out the path from (0,0,0) to (1, 1, 0) (90 degrees around the circle defined by the starting point (0,0,0) and the center (0,1,0)).

```
new Gesture().SetNormalizer(new FittedNormalizer(new Vector3(-1,-1,0), new Vector3(1,1,0)))
```

All Gestures must have a Normalizer attached, and FittedNormalizer is the standard for shapes with height and width. The Normalizer must take in bounds that completely encompass the bounds of the Checks that make up the Gesture. In the example above, the leftmost point of the Gesture is at `x=-1`, and the rightmost point is at `x=1`. When adding the Normalizer, it is very important to completely encompass the Gesture without making the bounding box too large, or else the Gesture will not be detected as intended.

`new Gesture().AddEvent(UnityAction<GestureMetaData> action)`

Each Gesture can have its own Event attached that will run when the Gesture is detected at any time. This call takes in a function with a single parameter of GestureMetaData.

Visualization

In order to see how the Gestures will look (and check your construction), navigate to `Tools/Gesture Visualization` while the game is running. Using the dropdown to select the correct Gesture name will give a rough depiction of what each Gesture will look like.