Intel® RealSense™ SDK

Hand Tracking Tutorial
Using Unity* Software

With the Intel® RealSense™ SDK, you have access to robust, natural human-computer interaction (HCI) algorithms such as face tracking, finger tracking, gesture recognition, speech recognition and synthesis, fully textured 3D scanning and enhanced depth augmented reality.

Using the SDK and Unity* software you can create Windows* applications and games that offer innovative user experiences.

This tutorial shows how to enable the hand tracking module of the SDK and use its features, such as hand and finger tracking, robust 22-point skeletal joint tracking, and gesture recognition.

The module can also display event alert notifications in your application, and a basic rendering utility is provided to visualize the data.

Contents

- **Overview**
  - Hand Tracking Data
  - Gesture Tracking
  - Hand Alert Notification
  - Blob Extractor
  - Contour Extractor
  - Data Smoothing Utility
- **Code Sample Files**
- **Creating a Session for Hand Tracking**
- **Initializing the Pipeline**
  - Hand Tracking
  - Gesture Tracking Configuration
  - Alert Event Notification Configuration
- **Stream Hand Tracking data**
  - Joint data
  - Gesture Data
  - Alert Event Notification Data
- **Rendering the Frame**
- **Cleaning up the Pipeline**
- **Running the Code Sample**
- **To learn more**

Overview

The Intel RealSense SDK contains input/output modules and algorithm modules. In this tutorial, you'll see how to use one of the algorithm modules: the hand tracking module. To learn more about I/O modules, see the Capturing Raw Streams Unity tutorial.

The hand module handles the tracking of hands in the scene, allowing you to reconstruct the 3D skeleton of the hands, recognize various hand gestures and get notifications for interesting events. The **PXCMHandModule** interface is used to set up the configuration and output data of up to two hands. The **PXCMHandConfiguration** can be used to set up different tracking options, enable gesture and alert notification and select the required outputs.

Hand Tracking Data

Applications can locate and return a **hand data** for up to two hands through the **PXCMHandData** interface.

The hand data is exposed to the application through the **iHand** interface that provides the following data:

Mass Center: Image and world coordinates of the calculated hand image mass center.

Extremity Points: Special tracking points such as the closest, left-most, top-most, right-most, bottom-most and center points which form the boundaries of the mask of the hand silhouette.

Body Side: Whether it is a left or right hand.

Palm Orientation: Estimation of where the hand is facing.

Tracked Joints: Positions and rotations of the user's hand in world and image coordinates.

Normalized Joints: Posture of user's hand without changing dimensions of the hand, i.e., the distances between each joint (bone-length) are always the same.

Finger Data: Degree of foldedness and the radius of a particular fingertip.

Segmentation Image: A mask of the hand silhouette allowing to separate a hand from the background image.

Hand Openness: (0-100 value) indicating if all fingers are completely folded to all fingers fully spread.

Gesture Tracking

Using the **PXCMHandData** interface, you can also access gesture information. Certain defined hand movements, also known as hand-gesture can be enabled for detection.

Gesture Name	Illustration	Description
spreadfingers		The hand is open facing the camera with fingers pointing upwards.
fist		Fingers folded into a fist.
tap		Hand moving towards the camera (along z-axis) as if one is pressing a button.
thumb_down		The hand is closed with the thumb finger pointing down.
thumb_up		The hand is closed with the thumb finger pointing up.
two_fingers_pinch_open		The thumb finger and the index finger touch with vertical orientation of the hand.
v_sign		Index and middle fingers extended in upwards direction.

full_pinch		All fingers extended and touching the thumb. The pinched fingers should be pointing to the screen.
swipe		Swipe to the left with the right hand or swipe to the right with the left hand. The palm is facing the side and fingers are more or less towards the camera. The hand swiping must be inside the camera's field of view.
wave		Move the hand to the left and then the right. Repeat the action if necessary.

Hand Alert Notification

This module helps notify your application of certain useful information:

Hand detection and tracking: Inform the application that a hand is detected and is tracked.

Hand calibration: Inform the application that hand calibration data is available.

Tracking Boundaries: Inform the application when the tracked hand goes out or is about to go out of the tracking boundaries.

Blob Extractor

The blob extractor provides information about image blobs within certain depth value range. The blob extractor does not handle joint tracking or gesture tracking.

Contour Extractor

The contour extractor provides information about the contour of the image blobs. Just like the blob extractor, contour extractor does not handle joint tracking or gesture tracking.

Data Smoothing Utility

The data smoothing utility can be used to reduce data noise. The utility interface enables data types for 1D, 2D, 3D data points with support for Stabilizer smoother, Weighted smoother, Quadratic smoother and Spring smoother algorithms.

Code Sample Files

You can use either procedural calls (used in this tutorial) or event callbacks to capture hand data, and code samples for both are listed in Table 1. Using event callbacks is usually preferred when developing console applications; procedural calls are often used for GUI applications.

The HandRenderer.cs file, provided with the code samples, contains the **HandRender** utility class that renders the hand tracking data. It is provided with this tutorial so that you do not have to create your own hand rendering algorithm.

Executable files (.exe) are provided in the Debug subfolder in the code sample directory.

Table 1: Hand Tracking Code Samples

Code Sample	**For explanation, see:**
Hand tracking using procedural calls Code sample file: HandTracking.cs	This Tutorial. Also see Hand Tracking using the SenseManager Procedural Functions section of the SDK Reference Manual.
Hand tracking using event callbacks	Hand Tracking using the SenseManager Callback Functions section of the SDK Reference Manual.
Gesture recognition using procedural or events	Gesture Recognition Data section of the SDK Reference Manual.
Alert notification using procedural calls or events	Handle Alert Notification section of the SDK Reference Manual.

Creating a Session for Hand Tracking

The SDK core is represented by two interfaces:

- **PXCMSession:** manages all of the modules of the SDK
- **PXCMSenseManager:** organizes a pipeline by starting, stopping, and pausing the operations of its various modalities.

The first step when creating an application that uses the Intel RealSense SDK is to create a session. A session can be created explicitly by creating an instance of **PXCMSession**. Each session maintains its own pipeline that contains the I/O and algorithm modules.

Another way of creating a session is by creating an instance of the **PXCMSenseManager** using **CreateInstance**. The PXCMSenseManager implicitly creates a session internally. Do this in the Start function before calling the Update method.

```
/// <summary>
/// Use this for initialization
/// Unity function called on the frame when a script is enabled
/// just before the Update method is called the first time.
/// </summary>
void Start () {

    /* Initialize a PXCMSenseManager instance */
    psm = PXCMSenseManager.CreateInstance();
    if (psm == null){
        Debug.LogError("SenseManager Initialization Failed");
        return;
    }

}
```

Initializing the Pipeline

Hand Tracking

1. Enable hand tracking using **EnableHand**.
2. Retrieve an instance of hand module using **QueryHand** on the instance of PXCMSenseManager.
3. Initialize the pipeline using **Init**.

Note: You can use sts of type **pxcmStatus** for error checking.

```
// Unity Start function
{
  /* Enable the hand tracking module*/
     sts = psm.EnableHand();
     if (sts != pxcmStatus.PXCM_STATUS_NO_ERROR) Debug.LogError("PXCSenseManager.EnableHand: " + sts);

     /* Retrieve an instance of hand to configure */
     handAnalyzer = psm.QueryHand();
     if (handAnalyzer == null) Debug.LogError("PXCSenseManager.QueryHand");

     /* Initialize the execution pipeline */
     sts = psm.Init();
     if (sts != pxcmStatus.PXCM_STATUS_NO_ERROR)
     {
        Debug.LogError("PXCMSenseManager.Init Failed");
        OnDisable();     // Clean-up
        return;
      }

}
```

Create an instance of **PXCMHandConfiguration** using the hand module to configure the hand.

```
PXCMHandConfiguration config = handAnalyzer.CreateActiveConfiguration();
```

Gesture Tracking Configuration

Enable all the gestures using **EnableAllGestures** on the PXCMHandConfiguration instance. You can also enable specific gestures using EnableGesture.

```
config.EnableAllGestures();
```

Alert Event Notification Configuration

Similarly, enable all alerts using **EnableAllAlerts**. Refer to EnableAlert to enable specific alerts for the hand module.

```
config.EnableAllAlerts();
```

After configuring the hand module, apply the changes to the active configuration using **ApplyChanges**. Dispose any instance of **PXCMHandConfiguration** using **Dispose** on it.

```
config.ApplyChanges();
config.Dispose();
```

Stream Hand Tracking data

1. Perform all processing in the **Update** function, which Unity software calls every frame.
2. Acquire frames in a loop using **AcquireFrame(false,0)**:
   a. TRUE to wait for all processing modules to be ready in a given frame; else
   b. FALSE whenever any of the processing modules signal.
3. First retrieve the hand module using **QueryHand** and then use **Update** on a **HandData** instance to update the hand data.
4. At the end of **Update** function make sure to release the acquired frame using **ReleaseFrame** to process the next frame.

```
/// <summary>
/// Update is called every frame, if the MonoBehaviour is enabled.
/// </summary>
void Update () {

    /* Make sure PXCMSenseManager Instance is Initialized */
    if (psm == null) return;

    /* Wait until any frame data is available true(aligned) false(unaligned) */
    if (psm.AcquireFrame(false,0) != pxcmStatus.PXCM_STATUS_NO_ERROR) return;

    /* Retrieve am instance of hand tracking Module */
    handAnalyzer = psm.QueryHand();
    if (handAnalyzer != null)
    {
       /* Retrieve an instance of hand tracking Data */
       PXCMHandData _outputData = handAnalyzer.CreateOutput();
       if (_outputData != null)
       {
          _outputData.Update();

          //Retrieve joint data, gesture recognition data and alert notification data
          //refer to next section
       }
    }

     /* Realease the frame to process the next frame */
    psm.ReleaseFrame();
}
```

Joint data

1. Retrieve the number of detected hands in the current frame using **QueryNumberOfHands** on the updated hand data instance.
2. Loop through the total number of detected hands.
3. Use **QueryHandData** to populate the **PXCHandData::iHand** instance**,** with appropriate access order type.
4. On a given instance of **iHand** you can loop through and **QueryTrackedJoint** with a specific **JointType** to extract individual joints.

- Make sure to declare the two dimensional array joints and initialize the one dimensional arrays of handids and bodysSides in the Mono Behavior class.

```
//Update Unity function
{
        //AcquireFrame

        /* Retrieve Hand Joints*/
        joints = new PXCMHandData.JointData[2, PXCMHandData.NUMBER_OF_JOINTS];
        for (int i = 0; i < _outputData.QueryNumberOfHands(); i++)
        {
           PXCMHandData.IHand _handData;
           _outputData.QueryHandData(PXCMHandData.AccessOrderType.ACCESS_ORDER_FIXED, i,   out
        _handData);
           for (int j = 0; j < PXCMHandData.NUMBER_OF_JOINTS; j++)
              _handData.QueryTrackedJoint((PXCMHandData.JointType)j, out joints[i, j]);

           handIds[i] = _handData.QueryUniqueId();
           bodySides[i] = _handData.QueryBodySide();
        }

        // Render the joints:  explained in rendering the frame section

        // Release Frame
}
```

Gesture Data

1. Iterate through all the fired gestures using **QueryFiredGesturesNumber** and populate the gesture data using **QueryFiredGestureData**.
2. Access an **iHand** instance associated with a gesture using **QueryHandDataById** on a hand data instance.
3. Use **QueryBodySide** on the retrieved iHand instance to detect whether the gesture was fired by a right or left hand.

```
//Update Unity function
{
        //AcquireFrame

        /* Retrieve Gesture Data */
        PXCMHandData.GestureData _gestureData;
        for (int i = 0; i < _outputData.QueryFiredGesturesNumber(); i++)
        {
                 if (_outputData.QueryFiredGestureData(i, out _gestureData) ==
                pxcmStatus.PXCM_STATUS_NO_ERROR)
                {
                        // Display the gestures:  explained in rendering the frame section
                 }
        }
        // Release Frame

}
```

Alert Event Notification Data

1. Iterate through fired alerts in the current frame using **QueryFiredAlertsNumber** and populate the alert data using **QueryFiredAlertData**.
2. Using **alerData.label** you can compare in a switch statement all available alerts under the **PXCMFaceData** alert type.

```
//Update Unity function
{
        //AcquireFrame

        /* Retrieve Alert Data */
        PXCMHandData.AlertData _alertData;
        for (int i = 0; i < _outputData.QueryFiredAlertsNumber(); i++)
        {
            if (_outputData.QueryFiredAlertData(i, out _alertData) == pxcmStatus.PXCM_STATUS_NO_ERROR)
            {
                // Display the alerts:  explained in rendering the frame section
             }
        }
}
```

Rendering the Frame

A render utility class has been provided to render various hand tracking data in Unity software.

- Look over the **HandRenderer** class's **DisplayAlerts**, **DisplayGestures**, **DisplayJoints** functions to see some of the ways to use the Intel RealSense SDK to render visuals in Unity software.
- Although **DisplaySmoothenedJoints** utilizes tutorial specific smoothing of joint data, the SDK provides a better smoothing component, PXCMDataSmoothing. For data smoothing example, please refer to the HandsViewer Unity* Project in the **RSSDK/framework/Unity** directory.

Cleaning up the Pipeline

After your application is done capturing and rendering samples, you must "clean up". This is done in the **OnDisable** function, which the Unity software calls right before the behavior is disabled.

1. Check to see if **PXCMSenseManager** is already released.
2. **Dispose** the handAnalyzer instance.
3. If not, release any session and processing module instances using **Dispose()** on the **PXCMSenseManager** instance.

```
/// <summary>
/// Unity function that is called when the behaviour becomes disabled () or inactive.
/// Used for clean-up in the end
/// </summary>
void OnDisable()
{
   handAnalyzer.Dispose();
   if (psm == null) return;
   psm.Dispose();
}
```

Running the Code Sample

You can run the Unity* tutorial code sample by building and running the HandsTracking scene in Unity.

Figure 1 shows a rendered image frame that displays the hand tracking data as the 22 tracked joints, alert data, and gesture data along with the Body side of each gesture.

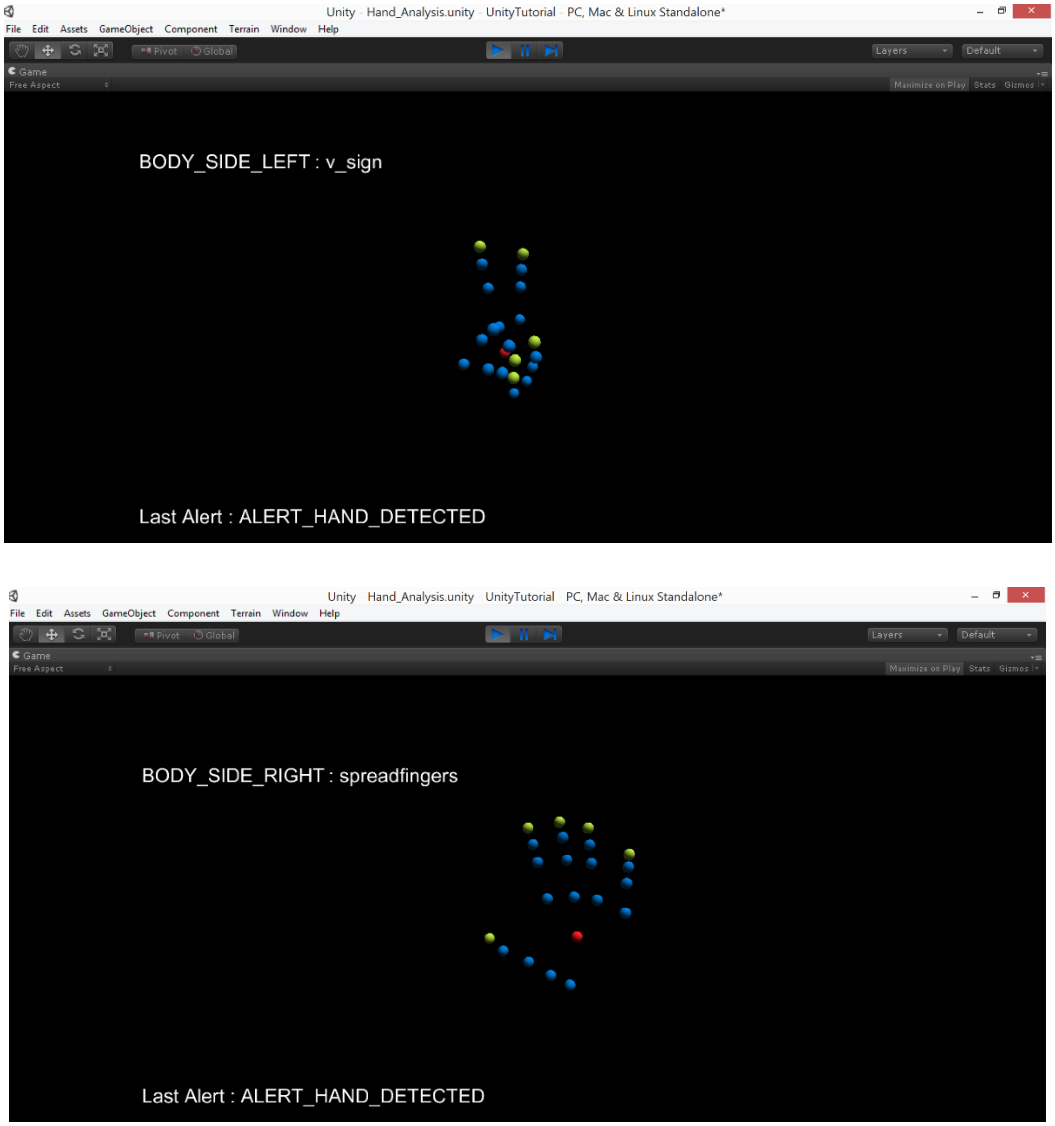

Figure 1. Rendered Hand Tracking data, Alert data, and Gesture data

To learn more

- The SDK Reference Manual is your complete reference guide and contains API definitions, advanced programming techniques, frameworks, and other need-to-know topics.
- The Intel RealSense SDK allows the hand tracking module to adapt to a user's hand over time. Check out how to save **hand calibration data** for specific users in the Hand Calibration Data section of the SDK Reference Manual.
- For more ways to capture Hand Data refer to its member functions.