Face Tracking Tutorial
Using Unity* Software

Intel® RealSense™ SDK

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 face tracking algorithms, namely: face location detection, landmark detection, pose detection, and expression detection. The module also provides a basic rendering utility to view the output data.

Contents

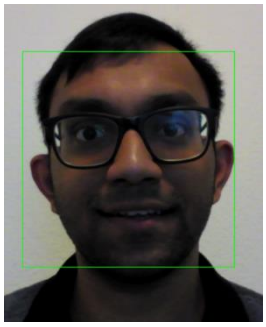
Overview

The Intel RealSense SDK supports input/output modules and algorithm modules. In this tutorial, you'll see how to use the algorithms in the face tracking module to collect and render face location and landmark detection data and to collect and display pose detection and expression detection data on your screen.

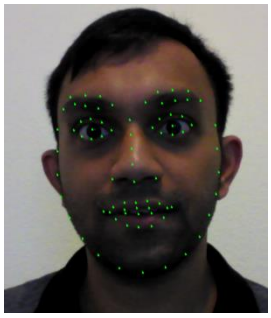
The **PXCMFaceModule** interface of the SDK is the representation of the face tracking module. **PXCMFaceData** interface abstracts the face location detection, landmark detection, pose detection, and expression detection algorithms of a tracked face. You can use this interface to collect data acquired by the abstracted algorithms.

Each of the five face tracking algorithms collect a specific type of data:

Face Location Detection – Applications can locate and return a face (or multiple faces) inside a rectangle. This data can be used to count how many faces are in the current video sample and find their general locations.



Landmark Detection – This data, comprising 78 key landmark points, is often used to further identify separate features (eyes, mouth, eyebrows, etc.) of a detected face. One example usage is to determine a user's eye location in applications that change perspectives based on where the user is looking on the screen or using the feature points to create a face avatar.



Pose Detection – This data estimates the head orientation (in degrees) of a face once it is detected. Head orientation is measured three ways as: roll, pitch, and yaw (see Figure 1). Your application can use this data in many ways. For example, the perspective of the scene may change based on the user's head orientation to simulate a 3D effect.

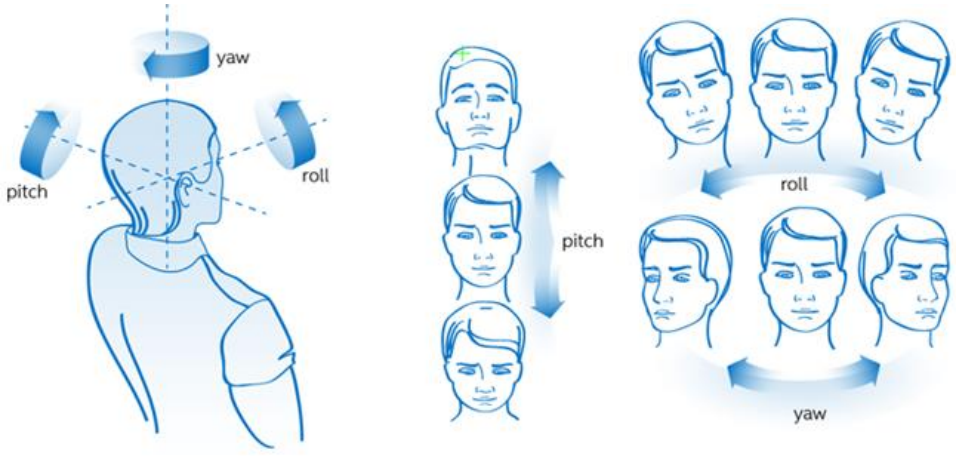


Figure 1. Head orientation measured as: roll, pitch, and yaw

Expression Detection – This module calculates the scores for a few supported expressions such as eye-closed, eye-brow turning up, etc.

Face Recognition – This feature compares the current face with a set of reference pictures in a recognition database to determine the user's identification.

Face Alert Notification– This feature helps notify the application with useful information on the face status, such as; face detected, face lost, face occluded or if face is out of camera's field of view.

Code Sample Files

You can use either procedural calls (used in this tutorial) or event callbacks to capture face 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 facerenderer.cs file, provided with the code samples, contains the **FaceRender** utility class that renders the face tracking data. It is provided with the SDK so that you do not have to create your own face rendering algorithm.

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

Table 1: Face Tracking Code Samples

Code Sample	**For explanation, see:**
Face tracking algorithms using procedural calls Code Sample File: FaceTracking.cs	This Tutorial. Also see Face Tracking using the SenseManager Procedural Functions section of the SDK Reference Manual.
Face tracking algorithms using event callbacks	Face Tracking using the SenseManager Callback Functions section of the SDK Reference Manual.

Creating a Session for Face 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

1. Enable the face tracking using **EnableFace**.
2. Query the face Module instance using **QueryFace** on the **PXCMSenseManager** instance to configure the face tracking module.
3. Initialize the pipeline using **Init**.
4. Retrieve **PXCMFaceData** instance from the face module using **CreateOutput**.
5. Retrieve an active configuration from the face module using **CreateActiveConfiguration**.
6. Set the tracking mode using **SetTrackingMode**.

You can also determine if you want to enable 2D face tracking, which captures only color input data or 3D face tracking, which captures color and depth data using SetTrackingMode.

```
// Unity Start function
{

    /* Enable the face tracking module*/
    sts = psm.EnableFace();
    if (sts != pxcmStatus.PXCM_STATUS_NO_ERROR) Debug.LogError("PXCSenseManager.EnableFace: " + sts);

    /* Retrieve an instance of face to configure */
    faceAnalyzer = psm.QueryFace();
    if (faceAnalyzer == null) Debug.LogError("PXCSenseManager.QueryFace");

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

    /* Retrieve a PXCMFaceConfiguration instance from a face to enable Gestures and Alerts */
    PXCMFaceConfiguration config = faceAnalyzer.CreateActiveConfiguration();

}
```

Face Detection Configuration

Enable the detection algorithm by setting **detection.isEnabled** to **true**.

```
config.detection.isEnabled = true;
```

Face Landmark Detection Configuration

Enable the landmark detection algorithm by setting **landmarks.isEnabled** to **true**.

```
config.landmarks.isEnabled = true;
```

Face Expression Detection Configuration

1. Enable the expression detection algorithm.
2. Enable a specific face expression.

Refer to EnableAllExpressions to enable all face expressions.

```
config.QueryExpressions().Enable();
config.QueryExpressions().EnableExpression(PXCMFaceData.ExpressionsData.FaceExpression.EXPRESSION_MOUTH_OPEN);
```

Face Pose Detection Configuration

Enable the pose detection algorithm by setting **pose.isEnabled** to **True**.

```
config.pose.isEnabled = true;
```

Face Alert Event Notification Configuration

Enable the alerts using **EnableAllAlerts**. Refer to EnableAlert to enable specific alerts.

```
config->EnableAllAlerts();
```

Apply the Configurations

Apply changes to the active configuration using **ApplyChanges** and Dispose the active face configuration.

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

Although this tutorial sample is meant for one face you can enable individual configuration algorithms for multiple faces setting the **MaxTrackedFaces**.

Streaming the Face Tracking Algorithms

1. Perform all processing in the **Update** function, which Unity software calls every frame.
2. Acquire frames in a loop using **AcquireFrame(true)**:
   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 face module using **QueryFace** and then use **Update** on a **FaceData** instance to update the hand data.
4. On the FaceData **QueryNumberOfDetectedFaces** to iterate through every face detected.
5. Retrieve a face tracked using **QueryFaceByIndex** and check if it's a valid face.
6. At the end of **Update** function make sure to **Dispose** the **FaceData** and 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(true) != pxcmStatus.PXCM_STATUS_NO_ERROR) return;

    /* Retrieve am instance of face tracking Module */
    faceAnalyzer = psm. QueryFace ();
    if (faceAnalyzer!= null) {
       /* Retrieve an instance of face tracking Data */
       PXCMFaceData _outputData = faceAnalyzer.CreateOutput();
       if (_outputData != null) {
          _outputData.Update();
          for (int i = 0; i < _outputData.QueryNumberOfDetectedFaces(); i++){
             PXCMFaceData.Face _iFace = _outputData.QueryFaceByIndex(i);
             if (_iFace != null){
                    //Retrieve face detection data, landmark data, expression data, pose data per face
             }
          }// End of for loop
          /* Retrieve alert data */
       }
      _outputData.Dispose();
     }
     /* Realease the frame to process the next frame */
    psm.ReleaseFrame();
}
```

Face Detection Data

1. Retrieve the detection data using **QueryDetection**.
2. Use **QueryBoundingRect** to get the tracked face rectangle dimensions.
3. Use SetDetection to pass the rectangle to render.

```
// for loop that iterates through detected faces
{               /* Retrieve Detection Data */
                  PXCMFaceData.DetectionData detectionData = _iFace.QueryDetection();
                  if (detectionData != null)
                  {
                     PXCMRectI32 rect;
                     if (detectionData.QueryBoundingRect(out rect)) {
                          faceRenderer.SetDetectionRect(rect);
                  }

 }
```

Face Landmark Detection Data

1. Retrieve the landmark data using **QueryLandmarks**.
2. Use **QueryNumPoints** to initialize the **landmarkPoints** array.
3. Use **QueryPoints** to retrieve the 78 landmark points from the landmark data.
4. Use SetLandmark to pass the landmark array to render.

```
// for loop that iterates through detected faces
{               /* Retrieve 78 Landmark Points */
                  PXCMFaceData.LandmarksData LandmarkData = _iFace.QueryLandmarks();
                  if (LandmarkData != null)
                  {
                        PXCMFaceData.LandmarkPoint[] landmarkPoints = new
                        PXCMFaceData.LandmarkPoint[MaxPoints];
                        if(LandmarkData.QueryPoints(out landmarkPoints))
                                    faceRenderer.DisplayJoints2D(landmarkPoints);
                  }

 }
```

Face Expression Detection Data

1. Retrieve the expression data using **QueryExpressions**.
2. **QueryExpression** with the specific expression type to retrieve **FaceExpressionResult**.
3. Use SetExpression to pass the expressionResult and expression type to render.

```
// for loop that iterates through detected faces
{                              /* Retrieve Expression Data */
          PXCMFaceData.ExpressionsData expressionData = _iFace.QueryExpressions();
                if (expressionData != null)
                {
                   PXCMFaceData.ExpressionsData.FaceExpressionResult expressionResult;
           if(expressionData.QueryExpression(PXCMFaceData.ExpressionsData.FaceExpression.EXPRESSION_MO
UTH_OPEN, out expressionResult))
                      {
                                  faceRenderer.DisplayExpression(expressionResult,
                              PXCMFaceData.ExpressionsData.FaceExpression.EXPRESSION_MOUTH_OPEN);
                      }
                }

 }
```

Face Pose Detection Data

1. Retrieve the pose data using **QueryPose**.
2. Use **QueryPoseQuaternions** to fill the pose quaternion data structure.
3. Use SetPose to pass the pose quaternion data to render.

```
// for loop that iterates through detected faces
{               /* Retrieve Pose Data */
                 PXCMFaceData.PoseData poseData = _iFace.QueryPose();
                 if (poseData != null)
                 {
                    PXCMFaceData.PoseQuaternion poseQuaternion;
                    if (poseData.QueryPoseQuaternion(out poseQuaternion)){
                                faceRenderer.DisplayPoseQuaternion(poseQuaternion);
                    }
                 }
 }
```

Face Alert Event Notification Data

1. Create an instance of **PXCFaceData.AlertData**.
2. Iterate through fired alerts using **QueryFiredAlertsNumber** and populate the alert data using **QueryFiredAlertData**.
3. Using **alertData.label** you can compare in a switch statement all available alerts under the **PXCFaceData** alert type.

```
// Update function
{
        //AcquireFrame
         //Update output data

        /* Retrieve Alert Data */
        PXCMFaceData.AlertData _alertData;
        for (int i = 0; i < _outputData.QueryFiredAlertsNumber(); i++)
          if (_outputData.QueryFiredAlertData(i, out _alertData) == pxcmStatus.PXCM_STATUS_NO_ERROR)
                faceRenderer.DisplayAlerts(_alertData);

        //ReleaseFrame

 }
```

Rendering the Frame

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

- The SDK also provides a smoothing component, PXCMDataSmoothing.
- Look over the **FaceRenderer** class's **DisplayJoints2D**, **SetDetectionRect**, **DisplayPoseQuaternion**, **DisplayExpression**, **DisplayAlerts** functions to see some of the ways to use the Intel RealSense SDK to render visuals in Unity software.
- Please read the Unity Capturing Raw Streams tutorial for detailed step-by-step instructions to render the color feed as a texture in Unity software.

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 Unity software calls right before the behavior is disabled.

1. Check to make sure the **PXCMSenseManager** is already released.
2. 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()
{
   faceAnalyzer.Dispose();
   if (psm == null) return;
   psm.Dispose();
}
```

Running the Code Sample

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

Figures 2 and 3 show a rendered color image frame that displays the face location detection data as a rectangle around the found face and the landmark detection data as the 78 collected landmark points, Mouth Open Expression data, and the face pose represented in Quaternions.

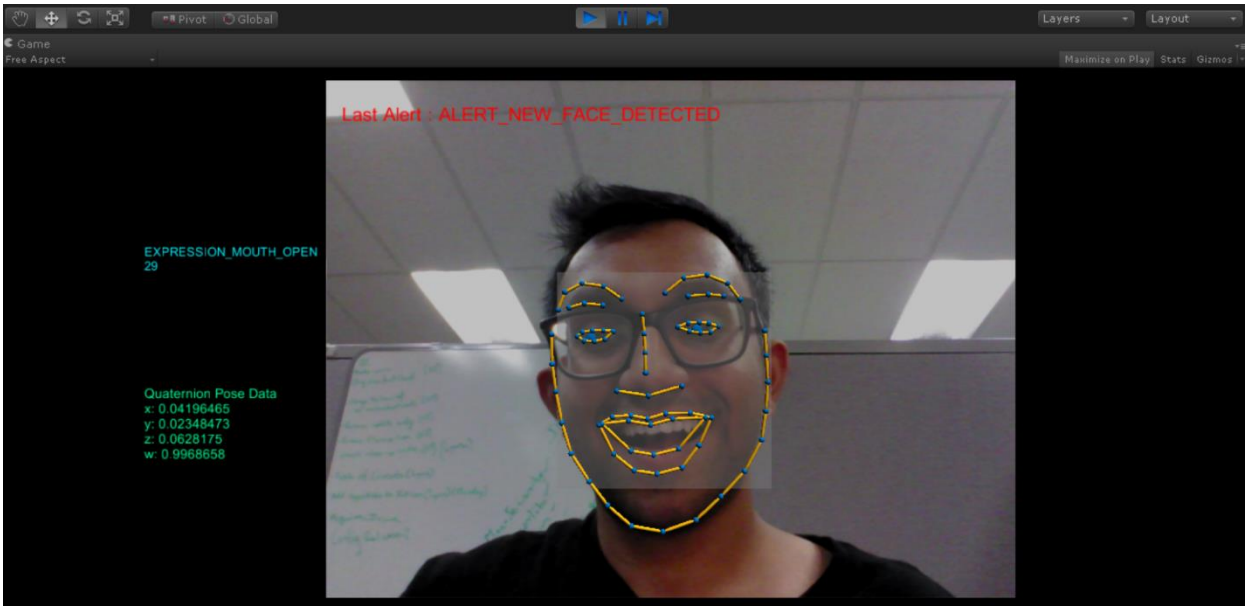


Figure 2. Rendered Color Image output when the face is detected

Figure 3. Rendered Color Image output when the face is occluded- see the ALERT_FACE_OCCLUDED message

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.

- To learn about Face Recognition, a feature that compares the current face with a set of reference pictures in a recognition database to determine the user's identification please to refer to Face Recognition section of the SDK Reference Manual.

- You can also tweak the device settings to get the best face tracking results by changing certain device properties.

```
PXCMCapture.Device device = psm.QueryCaptureManager().QueryDevice();
if (device)
{
   device.SetDepthConfidenceThreshold(1);
   device.SetIVCAMMotionRangeTradeOff(21);
   device.SetIVCAMFilterOption(6);
}
```