



# INTEL<sup>®</sup> PERCEPTUAL COMPUTING SDK

---

## Reference Manual Gesture Modules

API Version 1.0



## LEGAL DISCLAIMER

THIS DOCUMENT CONTAINS INFORMATION ON PRODUCTS IN THE DESIGN PHASE OF DEVELOPMENT.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

INTEL MAY MAKE CHANGES TO SPECIFICATIONS AND PRODUCT DESCRIPTIONS AT ANY TIME, WITHOUT NOTICE. DESIGNERS MUST NOT RELY ON THE ABSENCE OR CHARACTERISTICS OF ANY FEATURES OR INSTRUCTIONS MARKED "RESERVED" OR "UNDEFINED." INTEL RESERVES THESE FOR FUTURE DEFINITION AND SHALL HAVE NO RESPONSIBILITY WHATSOEVER FOR CONFLICTS OR INCOMPATIBILITIES ARISING FROM FUTURE CHANGES TO THEM. THE INFORMATION HERE IS SUBJECT TO CHANGE WITHOUT NOTICE. DO NOT FINALIZE A DESIGN WITH THIS INFORMATION.

THE PRODUCTS DESCRIBED IN THIS DOCUMENT MAY CONTAIN DESIGN DEFECTS OR ERRORS KNOWN AS ERRATA WHICH MAY CAUSE THE PRODUCT TO DEVIATE FROM PUBLISHED SPECIFICATIONS. CURRENT CHARACTERIZED ERRATA ARE AVAILABLE ON REQUEST.

CONTACT YOUR LOCAL INTEL SALES OFFICE OR YOUR DISTRIBUTOR TO OBTAIN THE LATEST SPECIFICATIONS AND BEFORE PLACING YOUR PRODUCT ORDER.

COPIES OF DOCUMENTS WHICH HAVE AN ORDER NUMBER AND ARE REFERENCED IN THIS DOCUMENT, OR OTHER INTEL LITERATURE, MAY BE OBTAINED BY CALLING 1-800-548-4725, OR BY VISITING INTEL'S WEB SITE [HTTP://WWW.INTEL.COM](http://www.intel.com).

ANY SOFTWARE SOURCE CODE REPRINTED IN THIS DOCUMENT IS FURNISHED UNDER A SOFTWARE LICENSE AND MAY ONLY BE USED OR COPIED IN ACCORDANCE WITH THE TERMS OF THAT LICENSE ANY SOFTWARE SOURCE CODE REPRINTED IN THIS DOCUMENT IS FURNISHED UNDER A SOFTWARE LICENSE AND MAY ONLY BE USED OR COPIED IN ACCORDANCE WITH THE TERMS OF THAT LICENSE

INTEL, THE INTEL LOGO, INTEL CORE, INTEL MEDIA SOFTWARE DEVELOPMENT KIT (INTEL MEDIA SDK) ARE TRADEMARKS OR REGISTERED TRADEMARKS OF INTEL CORPORATION OR ITS SUBSIDIARIES IN THE UNITED STATES AND OTHER COUNTRIES.

MPEG IS AN INTERNATIONAL STANDARD FOR VIDEO COMPRESSION/DECOMPRESSION PROMOTED BY ISO. IMPLEMENTATIONS OF MPEG CODECS, OR MPEG ENABLED PLATFORMS MAY REQUIRE LICENSES FROM VARIOUS ENTITIES, INCLUDING INTEL CORPORATION.

\*OTHER NAMES AND BRANDS MAY BE CLAIMED AS THE PROPERTY OF OTHERS.

COPYRIGHT © 2011-2013, INTEL CORPORATION. ALL RIGHTS RESERVED.



## Optimization Notice

Intel compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the "Intel Compiler User and Reference Guides" under "Compiler Options." Many library routines that are part of Intel compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel compilers, associated libraries and associated development tools may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20110307



## Table of Contents

Intel® Perceptual Computing SDK .....	1
Document Conventions .....	1
Acronyms and Abbreviations .....	1
Programming Guide .....	2
General Procedure .....	3
Geometric Nodes and Hand Modeling .....	5
Pose and Gesture Recognition .....	6
Blob Image and Data .....	8
Alert Events .....	9
Coordinates .....	11
Interface and Function Reference .....	13
PXCGesture .....	13
ProcessImageAsync .....	14
QueryBlobData .....	14
QueryBlobImage .....	15
QueryGestureData .....	16
QueryNodeData .....	16
QueryProfile .....	17
QueryUser .....	18
SetProfile .....	18
SubscribeAlert .....	19
SubscribeGesture .....	20
PXCGesture::Alert::Handler .....	20



OnAlert .....	21
PXCGesture::Gesture::Handler .....	21
OnGesture.....	21
<b>Structure Reference .....</b>	<b>23</b>
PXCGesture::Alert .....	23
PXCGesture::Blob.....	23
PXCGesture::GeoNode .....	24
PXCGesture::Gesture .....	26
PXCGesture::ProfileInfo .....	27
<b>Enumerator Reference .....</b>	<b>29</b>
PXCGesture::Alert::Label .....	29
PXCGesture::Blob::Label.....	30
PXCGesture::GeoNode::Label .....	30
PXCGesture::GeoNode::Openness .....	32
PXCGesture::GeoNode::Side.....	32
PXCGesture::Gesture::Label .....	33



## Intel® Perceptual Computing SDK

The Intel® Perceptual Computing SDK is a library of pattern detection and recognition algorithm implementations exposed through standardized interfaces. The library's purpose is to lower barriers to using these algorithms and shift the application developers' focus from coding the algorithm details to innovating on the usage of these algorithms for next generation human computer experience.

This document describes the gesture modules of the Intel® Perceptual Computing SDK Application Programming Interface (API). The other Perceptual Computing SDK Manuals that are released with the SDK describe different perceptual computing algorithms and their API definitions.

### Document Conventions

The SDK API uses the Verdana typeface for normal prose. With the exception of section headings and the table of contents, all code-related items appear in the Courier New typeface (`pxcStatus`). Hyperlinks appear in underlined boldface, such as **PXCSession**.

### Acronyms and Abbreviations

<b>API</b>	Application Programming Interface
<b>SDK</b>	Software Development Kit
<b>SP</b>	Synchronization Point



## Programming Guide

The SDK gesture recognition module takes RGB, depth, or IR streams as input, and returns the recognized gestures or any intermediate tracking results.

The SDK module provides four types of processing results: blob information, geometric node tracking result, pose/gesture notification, and alert notification.

Blobs are those intermediate results that represent the intermediate image processing results on the raw input. See the [Blob::Label](#) enumerator for supported blobs. A typical blob is the scene blob [LABEL\\_SCENE](#), which is the processed depth image of the camera view. The processing may include subsampling; thus, a blob resolution may be smaller than the input depth image. The module returns the blob image as well as other blob-related parameters, as defined in the [Blob](#) structure. The SDK module provides the blob information after processing every input frame.

Geometric nodes are skeleton joints on a human body or those of a localized body part. For example, the joint can be [LABEL\\_ELBOW\\_PRIMARY](#) to represent the left elbow, or [LABEL\\_HAND\\_PRIMARY| LABEL\\_FINGER\\_RING](#) to represent the left ring fingertip. See the [GeoNode::Label](#) enumerator for supported labels. The SDK gesture recognition module returns the positions of tracked geometric nodes and other useful information. See the [GeoNode](#) structure for details. The SDK module provides the geometric node information after processing every input frame.

The SDK module recognizes a set of predefined poses and gestures, and returns the recognition results. Poses are static hand and finger positions defined to deliver certain meanings. For example, [LABEL\\_POSE\\_PEACE](#) is a pose that delivers the meaning of peace or victory. To perform the pose, fully extend the index and middle fingers and close the rest. See the [Gesture::Label](#) enumerator for predefined poses. Gestures are a set of changing poses (or patterns) over a duration of time. The SDK module uses callbacks to notify the application when the module recognizes a specific pose or gesture. The SDK notifies a recognized pose twice, the first time for the pose to be active and the second time for the pose to be inactive. The SDK module notifies the application when the module recognizes a specific gesture only once, that is, at the end of the gesture.

The SDK module sends alerts to the application when the module detects errors. For example, if the tracked object is about to move out of the camera field of view, the SDK module sends notification that the object has moved out of field of view. See the [Alert::Label](#) enumerator for alert definitions. The application may want to notify the user to move accordingly via some non-intrusive visual queues.

## General Procedure

When using the pipeline interface `UtilPipeline`, as illustrated in Example 1, the application uses the following procedure for finger tracking and gesture recognition:

- The application needs to call the `EnableGesture` function to enable finger tracking and gesture recognition. This is usually done in the constructor of the `UtilPipeline` derived class. See the `UtilPipeline` definition for details.
- Optionally, the application can override the `OnGestureSetup` function to fine-tune any parameters during the module initialization.
- If the application needs to receive pose/gesture notification, the application can override the `OnGesture` function. Similarly, the application can override the `OnAlert` function to receive alert notification.
- At each frame, the application can use the `QueryGesture` function to obtain the [PXCGesture](#) instance and call into its member functions to access the geometric node details or blob details.

```
class MyPipeline: public UtilPipeline {
...
    MyPipeline(...):UtilPipeline(...) {
        ...
        EnableGesture();
        ...
    }
    virtual void OnGestureSetup(PXCGesture::Profile *profile) {}
    virtual void PXCAPI OnGesture(PXCGesture::Gesture *data) {}
    virtual void PXCAPI OnAlert(PXCGesture::Alert *data) {}
    virtual bool OnNewFrame(void) {
        ...
        QueryGesture() →QueryNodeData (...);
        ...
        QueryGesture() →QueryBlobData (...);
        ...
    }
...
}
```

**Example 1: Finger Tracking and Gesture Processing in UtilPipeline**



If the application directly accesses the [PXCGesture](#) interface, the application should follow the following procedure:

- **Locate a module implementation:** The application uses the `PXCSession::CreateImpl` function to create an instance of the [PXCGesture](#) interface, as illustrated by Example 2. See the `CreateImpl` function for additional ways to locate a module implementation.

```
PXCGesture *gesture=0;
session->CreateImpl(PXCGesture::CUID, (void**) &gesture);
```

#### Example 2: Create a PXCGesture Instance

- **Initialize the module:** The two functions for module initialization are [QueryProfile](#) and [SetProfile](#). The former function returns available configurations. The latter sets one as the current active configuration. In Example 3, the application queries the first supported configuration; uses it to locate an input device that can provide data; and then initializes the module with the configuration. Note that the gesture module input needs are specified as part of the [ProfileInfo](#) structure, which may be a depth image, and/or a color image. The utility class `UtilCapture` will locate an input device that matches the gesture module data needs.

```
PXCGesture::ProfileInfo pinfo;
gesture->QueryProfile(0, &pinfo);
UtilCapture capture(&session);
capture.LocateStreams(&pinfo.inputs);
gesture->SetProfile(&pinfo);
```

#### Example 3: Initialize a Gesture Module

Additionally, the application can set up a gesture notification handler ([SubscribeGesture](#)), or an alert notification handler ([SubscribeAlert](#)). See the *Pose and Gesture Recognition* section and the *Alert Events* section for more details.

- **Data Processing Loop:** In the loop, the application passes samples from the input device to the gesture module for tracking and recognition. The application calls the [ProcessImageAsync](#) function to deliver the samples, as illustrated in Example 4. Note that the samples from the input device may contain multiple images such as a color image and a depth image. The application can use `PXCSmartArray<PXCIImage>` set of functions to simplify programming.

If the application needs to switch context, the application calls the [ProcessImageAsync](#) function with a `NULL` pointer to reset any tracking states saved in the gesture module.



```
PXCSmartArray<PXCIImage> images;
PXCSmartSPArray sps(2);
for (;;) {
    // Get samples from input device and pass to the gesture module
    capture.ReadStreamAsync(images.ReleaseRefs(), sps.ReleaseRef(0));
    gesture->ProcessImageAsync(images, sps.ReleaseRef(1));
    sps.SynchronizeEx();
    // Tracking or recognition results are ready. Now process them
    ...
}
```

**Example 4: Gesture Module Data Processing Loop**

## Geometric Nodes and Hand Modeling

The SDK module tracks geometric nodes for every frame. The application can use the [QueryNodeData](#) function to retrieve a single or multiple geometric node data (adjacent in labeling), as illustrated in Example 5. In the former case, if the specified geometric node data is not available, the function returns `PXC_STATUS_ITEM_UNAVAILABLE`. In the latter case, the `body` field of the [GeoNode](#) structure is [LABEL\\_ANY](#) if the corresponding node data is not available.

Note that the geometric node label always consists of two parts: a body label and a local details label. In the example, the body label is [LABEL\\_BODY\\_HAND\\_PRIMARY](#) and the local details label is [LABEL\\_FINGER\\_THUMB](#). See the [GeoNode::Label](#) enumerator for definitions of additional labels.

```
/* Retrieve individual node data */
PXCGesture::GeoNode thumb_data;
gesture->QueryGeoNode(0, PXCGesture::GeoNode::LABEL_BODY_HAND_PRIMARY | PXCGesture::GeoNode::LABEL_FINGER_THUMB, &thumb_data);

/* Retrieve data of all five fingers in a single call */
PXCGesture::GeoNode hand_data[5];
gesture->QueryGeoNode(0, PXCGesture::GeoNode::LABEL_BODY_HAND_PRIMARY | PXCGesture::GeoNode::LABEL_FINGER_THUMB, 5, hand_data);
```

**Example 5: Retrieve Geometric Node Data**



The [GeoNode](#) structure provides the following details as summarized in Table 1:

Node Data	Typical Usages
Basic information such as time stamp, user, body, side, confidence	Identify the geometric node or associate the node with some other nodes.
Position in image and world coordinates	Draw the hand skeleton, manipulate a virtual object, or use as a base for application-specific pose/gesture recognition.
Radius for fingertips	Use as an indicator of the fingertip volume for virtual object manipulation.
Normal vector for hand/palm center	Calculate the hand/palm orientation.
Hand openness state and openness value	Use as a simple hand open/close pose or combine with other methods to determine the hand states.
Hand center of mass	Determine the detection of a hand and its general position. Not meant for accuracy positioning.

**Table 1: Geometric Node Data and Their Typical Usages**

## Pose and Gesture Recognition

There are a few ways that an application can retrieve pose or gesture data.

If an application uses the pipeline interface `UtilPipeline`, the application can simply overwrite the `OnGesture` function, which provides event notification when a pose/gesture is recognized, as illustrated in Example 6.

```
class MyPipeline: public UtilPipeline {
    virtual void PXCAPI OnGesture(PXCGesture::Gesture *data) {
        // process pose/gesture details
    }
}
```

**Example 6: Pose/Gesture Processing in UtilPipeline**



If an application directly uses the [PXCGesture](#) interface, the application can subscribe to the gesture notification by using the [SubscribeGesture](#) function, as shown in Example 7. The function takes an event handler [OnGesture](#) that the application must implement, as shown in Example 8. The application must make sure that the event handler instance is valid when the event notification occurs. To unsubscribe, the application simply calls the [SubscribeGesture](#) function with a `NULL` pointer.

```
...
    MyGestureHandler handler;
    gesture->SubscribeGesture(100,&handler); // subscription
...
    gesture->SubscribeGesture(NULL);        // unsubscription
...
```

**Example 7: Subscribe and Unsubscribe to Pose/Gesture Notification**

```
class MyGestureHandler: public PXCGesture::Gesture::Handler {
public:
    virtual void PXC_API OnGesture(PXCGesture::Gesture *data) {
        // process pose/gesture details
    }
}
```

**Example 8: Pose/Gesture Handler Implementation**

Alternatively, an application can use the [QueryGestureData](#) function to retrieve any active poses or gestures of the current frame. There may be more than one pose or gesture available. Use the zero-based index to enumerate all poses/gestures, as illustrated in Example 9.

```
for (int i=0;;i++) {
    PXCGesture::Gesture gdata;
    pxcStatus sts=gesture->QueryGestureData(0,
        PXCGesture::GeoNode::LABEL_ANY,i,&gdata);
    if (sts<PXC_STATUS_NO_ERROR) break;
    // process the pose/gesture data in gdata.
}
```

**Example 9: Retrieve Active Poses or Gesture Data**



The SDK gesture module signals pose and gesture events from the same triggering geometric node in sequential order. For example, if the user performs a peace pose followed by a swipe left gesture, the SDK gesture module signals the following events:

1. [LABEL\\_POSE\\_PEACE](#), and set the active field of the [Gesture](#) structure to be true.
2. [LABEL\\_POSE\\_PEACE](#), and set the active field of the [Gesture](#) structure to be false.
3. [LABEL\\_NAV\\_SWIPE\\_LEFT](#), and set the active field of the [Gesture](#) structure to be true.

## Blob Image and Data

The application can call the [QueryBlobImage](#) function to obtain the blob image and the [QueryBlobData](#) function to obtain any blob data details, as illustrated in Example 10. If certain blobs contain more than one piece of data or images, use the zero-based index to retrieve all data and images.

```
PXCGesture::Blob bdata;  
gesture->QueryBlobData(LABEL_SCENE, 0, &bdata);  
PXCIImage *bimage;  
gesture->QueryBlobImage(LABEL_SCENE, 0, &bimage);
```

### Example 10: Retrieve Blob Data and Image

The blob [LABEL\\_SCENE](#) is the processed camera view of the scene, or the label map, for hand and finger tracking. The image is in 8-bit gray scale. The objects in the image are labeled with different pixel values. The `labelBackground`, `labelLeftHand`, and `labelRightHand` fields in the [Blob](#) structure indicate the pixel value, or -1 if the corresponding object does not exist. As illustrated in Figure 1, the label map shows a hand with pixel value 0 and the background as pixel value 1. The application can extract the hand out of the background by retrieving the zero-value pixels from the image.

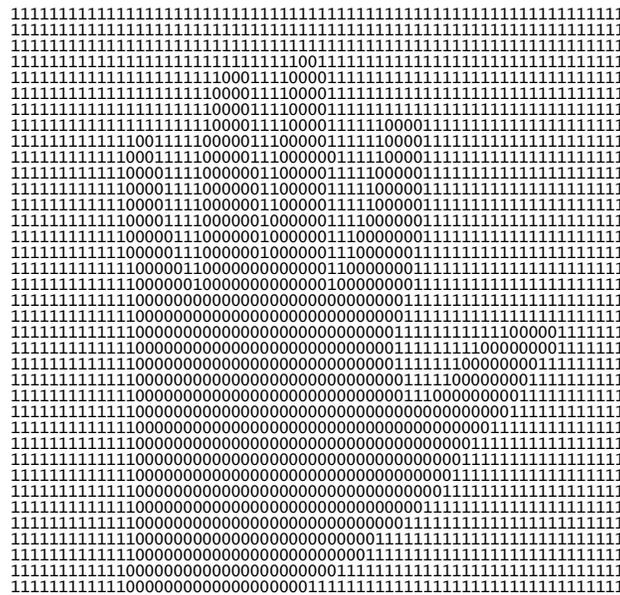


Figure 1: The LABEL\_SCENE Label Map (labelBackground=1, labelLeftHand=0 & labelRightHand=-1)

## Alert Events

If an application uses the pipeline interface `UtilPipeline`, the application can overwrite the `OnAlert` function to receive alert notification, as illustrated in Example 11:

```

class MyPipeline: public UtilPipeline {
...
    virtual void PXCAPI OnAlert(PXCGesture::Alert *data) {
        // process alert details
    }
...
}

```

Example 11: Alert Processing in UtilPipeline

If an application directly uses the [PXCGesture](#) interface, the application can use the [SubscribeAlert](#) function to subscribe to alert notification for tracking related events, as shown in Example 12. The function takes the event handler [OnAlert](#) that the application must implement, as shown in Example 13. The application must make sure that the event handler instance is valid when the event notification occurs. To unsubscribe, the application simply calls the [SubscribeAlert](#) function with a `NULL` pointer.

```
...
    MyAlertHandler handler;
    gesture→SubscribeAlert(&handler);           // subscription
...
    gesture→SubscribeAlert(NULL);             // unsubscription
...
```

#### Example 12: Subscribe and Unsubscribe to Alert Notification

```
class MyAlertHandler: public PXCGesture::Alert::Handler {
public:
    virtual void PXC_API OnAlert(PXCGesture::Alert *data) {
        // process alert details
    }
}
```

#### Example 13: Alert Handler Implementation

There are two types of tracking events: field of view events and geometric node tracking events. The field of view events indicate some erroneous conditions that the application may want to instruct the user to correct.

For example, [LABEL\\_FOV\\_LEFT](#) indicates that the user's hand intercepts the left border of the camera's field of view. It's recommended that the application implements some visual clues to ask the user to move a bit to the right. When the user moves, the SDK module will send a [LABEL\\_FOV\\_OK](#) event notification to indicate the recovery of the erroneous condition.

In certain cases, the SDK module may send multiple field of view events in a single event notification as a bit-OR'ed value. For example, if the user's hand intercepts both the left and top border of the camera's field of view, the SDK module will signal as [LABEL\\_FOV\\_LEFT | LABEL\\_FOV\\_TOP](#).

The geometric node tracking events signal when certain geonodes are in tracking or lost tracking. The SDK module sends the [LABEL\\_GEONODE\\_ACTIVE](#) event when a geometric node comes in tracking, and the [LABEL\\_GEONODE\\_INACTIVE](#) event when the geometric node is lost tracking. In Example 14, the application prints messages when the left/right hand is in tracking or lost tracking. The application can customize the geometric node alert notification by changing the `nodeAlerts` field of the [ProfileInfo](#) structure during the SDK module initialization, as illustrated in Example 15.



```
class MyAlertHandler: public PXCGesture::Alert::Handler {
    virtual void PXCAPIC OnAlert(PXCGesture::Alert *data) {
        if (data->label==PXCGesture::Alert::LABEL_GEONODE_ACTIVE)
            wprintf(L"Found hand id=%x\n", data->label);
        if (data->label==PXCGesture::Alert::LABEL_GEONODE_INACTIVE)
            wprintf(L"Lose hand id=%x\n", data->label);
    }
}
```

**Example 14: Handling Hand Found/Lost**

```
PXCGesture::ProfileInfo pinfo;
gesture->QueryProfile(0, &pinfo);
pinfo.nodeAlerts=PXCGesture::GeoNode::LABEL_BODY_HAND_PRIMARY |
                PXCGesture::GeoNode::LABEL_BODY_HAND_SECONDARY;
gesture->SetProfile(&pinfo);
```

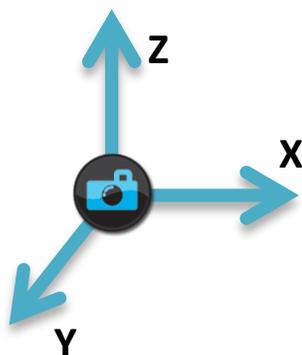
**Example 15: Configure Geometric Node Alert Events**

## Coordinates

The SDK interface definitions use two coordinate systems: the image coordinates and the world coordinates.

The image coordinates refer to the pixel  $(x, y)$  in the depth image pictures, where  $x$  is in the range of 0 to  $w-1$ , where  $w$  is the image width, and  $y$  is in the range of 0 to  $h-1$ , where  $h$  is the image height. For example, in the [GeoNode::Data](#) structure, the image coordinate parameters refer to the depth image coordinates.

The world coordinates are 3D coordinates  $(x, y, z)$  relative to the camera. The reference point  $(0, 0, 0)$  is the camera's location. When facing the camera, the  $x$  axis points to the right of the camera, the  $y$  axis points away from the camera to the object in front, and the  $z$  axis points up, as illustrated in Figure 2. The values  $x$ ,  $y$ ,  $z$  are in meters.



**Figure 2: World Coordinate System**



## Interface and Function Reference

This section describes SDK functions and their operations.

In each function description, only commonly used status codes are documented. The function may return additional status codes in certain case. See the `pxcStatus` enumerator for a list of all status codes.

### PXCGesture

The `PXCGesture` interface provides member functions to perform gesture recognition. The application can create this interface using the `CreateImpl` function with the interface identifier `PXCGesture : :CUID`.

The `PXCGesture` interface exposes the following member functions:

Member Functions	Description
<a href="#"><u>QueryProfile</u></a>	Return the algorithm configurations.
<a href="#"><u>SetProfile</u></a>	Set the algorithm configuration.
<a href="#"><u>QueryUser</u></a>	Return active user identifiers during tracking.
<a href="#"><u>QueryBlobData</u></a>	Return the parameter details of the blob.
<a href="#"><u>QueryBlobImage</u></a>	Return the blob image.
<a href="#"><u>QueryNodeData</u></a>	Return the details of the geometric node.
<a href="#"><u>QueryGestureData</u></a>	Return the current active post/gesture.
<a href="#"><u>ProcessImageAsync</u></a>	Process the inputs, track geometric nodes, and recognize the gestures.
<a href="#"><u>SubscribeAlert</u></a>	Subscribe to the alert events to receive alert notifications.
<a href="#"><u>SubscribeGesture</u></a>	Subscribe to the gesture events to receive notification of gestures during processing.



## ProcessImageAsync

### Syntax

```
pxcStatus ProcessImageAsync (PXCImage *images [], PXCScheduler::SyncPoint **sp);
```

### Parameters

images	The array of required input streams (supporting a maximum of three channels), as described in the <a href="#">ProfileInfo</a> configuration parameters.
sp	The SP, to be returned.

### Description

This function feeds the input streams to the gesture module for tracking and recognition. The array of the input streams must be big enough to provide the required streams and in the right order as reported in the algorithm configuration.

This function is asynchronous. The intermediate tracking results are not available until after the application synchronizes the SP.

If `images=NULL`, this function resets the module state. The application must synchronize any previous operations before performing a state reset.

### Return Status

<code>PXC_STATUS_NO_ERROR</code>	The function returned successfully.
----------------------------------	-------------------------------------

### Change History

This function was introduced in SDK API 1.0.

## QueryBlobData

### Syntax

```
pxcStatus QueryBlobData (Label label, pxcU32 idx, Data *data);
```

### Parameters

label	The blob label. See the <a href="#">Label</a> enumerator for definitions.
idx	Zero-based index to enumerate all available blobs with the same blob label.



`data` The blob data in the [Data](#) structure, to be returned.

## Description

This function returns the blob details.

If there is more than one blob with the same label, the application may increase the `idx` value to retrieve all available blobs, until the function returns `PXC_STATUS_ITEM_UNAVAILABLE`.

## Return Status

`PXC_STATUS_NO_ERROR` The function returned successfully.

`PXC_STATUS_ITEM_UNAVAILABLE` The blob was not found.

## Change History

This function was introduced in SDK API 1.0.

## QueryBlobImage

### Syntax

```
pxcStatus QueryBlobImage(Label label, pxcU32 idx, PXCImage **image);
```

### Parameters

<code>label</code>	The blob label. See the <a href="#">Label</a> enumerator for definitions.
<code>idx</code>	Zero-based index to enumerate all available blobs with the same blob label.
<code>image</code>	The blob image, to be returned.

### Description

This function returns the blob image.

If there is more than one blob with the same label, the application may increase the `idx` value to retrieve all available blobs, until the function returns `PXC_STATUS_ITEM_UNAVAILABLE`.

The application must release the returned image instance after use.

### Return Status

`PXC_STATUS_NO_ERROR` The function returned successfully.



PXC\_STATUS\_ITEM\_UNAVAILABLE The blob was not found.

## Change History

This function was introduced in SDK API 1.0.

## QueryGestureData

### Syntax

```
pxcStatus QueryGestureData(pxcUID user, GeoNode::Label body, pxcU32 idx, Gesture *data);
```

### Parameters

user	Reserved; must be zero.
body	The body that performs the pose/gesture. Use <a href="#">PXCGesture::GeoNode::LABEL_ANY</a> for any available body part.
idx	Zero-based index to enumerate all active gestures.
data	The gesture details to be returned. See the <a href="#">Gesture</a> structure for details.

### Description

This function returns the current active gesture details. If there is more than one gesture, use the zero-based index to enumerate all poses/gestures until the function returns `PXC_STATUS_ITEM_UNAVAILABLE`.

### Return Status

PXC_STATUS_NO_ERROR	The function returned successfully.
PXC_STATUS_ITEM_UNAVAILABLE	The geometric node is not available.

## Change History

This function was introduced in SDK API 1.0.

## QueryNodeData

### Syntax



```
pxcStatus QueryNodeData (pxcUID user, Label body, Data *data);  
pxcStatus QueryNodeData (pxcUID user, Label body, pxcU32 ndata, Data  
*data_array);
```

## Parameters

user	Reserved; must be zero.
body	The geometric node label. See the <a href="#">Label</a> enumerator for definitions.
data	The geometric node details to be returned. See the <a href="#">Data</a> structure for details.
ndata	The number of geometric nodes to be retrieved.
data_array	An array of the geometric node structure.

## Description

This function returns a single geometric node detail, or an array of the geometric node details. In the latter case, the array contains the geometric node details with increasing body labels starting with the variable `body`.

## Return Status

PXC_STATUS_NO_ERROR	The function returned successfully.
PXC_STATUS_ITEM_UNAVAILABLE	The geometric node is not available.

## Change History

This function was introduced in SDK API 1.0.

## QueryProfile

### Syntax

```
pxcStatus QueryProfile (pxcU32 pidx, ProfileInfo *pinfo);  
pxcStatus QueryProfile (ProfileInfo *pinfo);
```

### Parameters

pidx	The zero-based configuration index. Use <code>WORKING_PROFILE</code> , or the <code>pidx</code> omitted version, to retrieve the current working configuration.
------	---



`pinfo`

The [ProfileInfo](#) structure to retrieve the configuration parameters.

### Description

This function returns the algorithm configuration parameters.

### Return Status

`PXC_STATUS_NO_ERROR` The function returned successfully.

`PXC_STATUS_ITEM_UNAVAILABLE` The specified configuration was not found.

### Change History

This function was introduced in SDK API 1.0.

## QueryUser

### Syntax

```
pxcStatus QueryNode (pxcU32 idx, pxcUID *user);
```

### Parameters

`idx` The zero-based index for enumerating active users.

`user` The user identifier, to be returned.

### Description

This function returns the user identifiers in active tracking.

### Return Status

`PXC_STATUS_NO_ERROR` The function returned successfully.

`PXC_STATUS_ITEM_UNAVAILABLE` There are no more users.

### Change History

This function was introduced in SDK API 1.0.

## SetProfile

### Syntax



```
pxcStatus SetProfile(ProfileInfo *pinfo);
```

## Parameters

<code>pinfo</code>	The configuration parameters in the <a href="#">ProfileInfo</a> structure.
--------------------	--

## Description

This function sets the algorithm configuration parameters. The values in the [ProfileInfo](#) structure do not have to exactly match what the [QueryProfile](#) function returns.

## Return Status

<code>PXC_STATUS_NO_ERROR</code>	The function returned successfully.
<code>PXC_STATUS_PARAM_UNSUPPORTED</code>	There were unsupported parameters in the configuration.

## Change History

This function was introduced in SDK API 1.0.

## SubscribeAlert

### Syntax

```
pxcStatus SubscribeAlert(Alert::Handler *handler);
```

### Parameters

<code>handler</code>	The alert handler to process any event that the algorithm generates, or zero to unsubscribe to event notification. See the <a href="#">Alert::Handler</a> interface for definitions.
----------------------	--

### Description

This function subscribes to the alert event notifications. The application must implement the alert handler, in the [Alert::Handler](#) interface.

The application can unsubscribe to the event notification by calling this function with a `NULL` pointer.

### Return Status

<code>PXC_STATUS_NO_ERROR</code>	The function returned successfully.
----------------------------------	-------------------------------------



## Change History

This function was introduced in SDK API 1.0.

## SubscribeGesture

### Syntax

```
pxcStatus SubscribeGesture(pxcU32 threshold, Gesture::Handler *handler);
```

### Parameters

threshold	The gesture maturity threshold, from 0 to 100, at which the SDK module should trigger a gesture notification. Reserved; must be 100.
handler	The gesture event handler <a href="#">Gesture::Handler</a> , or NULL to unsubscribe to gesture event notification.

### Description

This function subscribes to the gesture recognition event notifications.

The application can unsubscribe to the event notification by calling this function with a `NULL` pointer.

### Return Status

`PXC_STATUS_NO_ERROR` The function returned successfully.

## Change History

This function was introduced in SDK API 1.0.

## PXCGesture::Alert::Handler

The `Handler` interface implements the alert event handling. The application must implement this interface.

The `Handler` interface exposes the following member functions:

Member Functions	Description
<code>OnAlert</code>	This SDK invokes this function when some alert occurs.





## Parameters

`gesture`

The [Gesture](#) structure that contains details of the recognized gesture.

## Description

The SDK invokes this `onGesture` function when the SDK recognizes a gesture. For poses, the SDK invokes this function twice once for active and the other for inactive.

The application should not implement any lengthy processing that blocks execution in this function.

## Change History

This function was introduced in SDK API 1.0.

## Structure Reference

In the following structure references, all reserved fields must be zero.

### PXCGesture::Alert

#### Definition

```
struct Alert {  
    pxcU64          timeStamp;  
    pxcUID          user;  
    GeoNode::Label body;  
    Label           label;  
    pxcU32          reserved[3];  
};
```

#### Members

timeStamp	The time stamp when the alert occurs, in 100 ns.
user	The user identifier that triggers the alert, zero if not available.
body	The body that triggers the alert, zero if not available. See the <a href="#">GeoNode::Label</a> enumerator for definitions.
label	The alert label. See the <a href="#">Label</a> enumerator for definitions.

#### Description

The `Alert` structure describes the current alert information.

#### Change History

This structure is introduced in SDK API 1.0.

### PXCGesture::Blob

#### Definition



```
struct Blob {  
    pxcU64          timeStamp;  
    Label           name;  
    pxcU32          labelBackground;  
    pxcU32          labelLeftHand;  
    pxcU32          labelRightHand;  
    pxcU32          reserved[26];  
};
```

## Description

The `Blob` structure describes the details of a blob.

## Members

<code>timestamp</code>	The time stamp of the blob, in 100 ns.
<code>name</code>	The blob name identifier. See the <a href="#">Label</a> enumerator for definitions.
<code>labelBackground</code>	The pixel value of the background image in the label map.
<code>labelLeftHand</code>	The pixel value of the left hand in the label map.
<code>labelRightHand</code>	The pixel value of the right hand in the label map.

## Change History

This structure is introduced in SDK API 1.0.

## PXCGesture::GeoNode

### Definition

```
struct GeoNode {  
    pxcU64          timeStamp;  
    pxcUID          user;  
    Label           body;  
    Side           side;  
    pxcU32          confidence;  
    PXCPPoint3DF32 positionWorld;  
    PXCPPoint3DF32 positionImage;  
    pxcU32          reserved2[4];  
  
    union {
```



```
pxcU32      reserved3[16];
struct { // fingertip
    pxcF32    radiusWorld;
    pxcF32    radiusImage;
};
struct { // hand
    PXCPPoint3DF32    massCenterWorld;
    PXCPPoint3DF32    massCenterImage;
    PXCPPoint3DF32    normal;
    pxcU32            openness;
    Openness         opennessState;
};
};
};
```

## Description

The `GeoNode` structure describes the details of a geometric node.

## Members

<code>timeStamp</code>	The current time stamp, in 100 ns.
<code>user</code>	The user identifier; reserved.
<code>body</code>	The body part identifier; see the <a href="#">Label</a> enumerator for definitions.
<code>side</code>	The body side identifier; see the <a href="#">Side</a> enumerator for definitions.
<code>confidence</code>	The confidence score from 0 to 100.
<code>positionWorld</code>	The node position in world coordinates.
<code>positionImage</code>	The node position in image space coordinates, in $(x, y, d)$ , where $(x, y)$ are coordinates in the depth image, and $d$ is the distance to the camera in meters.

### Fingertip-specific parameters:

<code>radiusWorld</code>	The volume of a fingertip node in 3D in meters.
<code>radiusImage</code>	The volume of a fingertip node in 2D in pixels.

### Hand/Palm center-specific parameters:

<code>normal</code>	The vector that is perpendicular to the hand (palm center) plane.
<code>openness</code>	The value from 0 to 100 to indicate the level of palm openness.
<code>opennessState</code>	The hand openness state; see the <a href="#">Openness</a> enumerator for definitions.



`massCenterWorld` The mass of center of a hand in world space coordinates.

`massCenterImage` The mass of center of a hand in image space coordinates.

## Change History

This structure is introduced in SDK API 1.0.

## PXCGesture::Gesture

### Definition

```
struct Gesture {
    pxcU64          timeStamp;
    pxcUID          user;
    GeoNode::Label body;
    Label           label;
    pxcU32          confidence;
    pxcBool         active;
    pxcU32          reserved[9];
};
```

### Description

The `Gesture` structure describes the details of a recognized gesture.

### Members

<code>timestamp</code>	The time stamp that the gesture ends, in 100 ns.
<code>user</code>	The user that performs this gesture, or zero if not available.
<code>body</code>	The body part that performs the gesture, or zero if not available. See the <a href="#">GeoNode::Label</a> enumerator for definitions.
<code>label</code>	The gesture label. See the <a href="#">Label</a> enumerator for details.
<code>confidence</code>	The confidence score from 0 to 100.
<code>active</code>	For poses only, the boolean value indicates whether the pose is active or inactive. The SDK calls back twice for poses, once for pose active and the other for pose inactive.

## Change History



This structure is introduced in SDK API 1.0.

## PXCGesture::ProfileInfo

### Definition

```
struct ProfileInfo {
    PXCCapture::VideoStream::DataDesc    inputs;
    Gesture::Set                          sets;
    GeoNode::Label                       bodies;
    Blob::Label                          blobs;
    Alert::Label                          alerts;
    GeoNode::Label                       nodeAlerts;
    pxcU32                                activationDistance;
    pxcU32                                reserved[6];
};
```

### Description

The `ProfileInfo` structure describes the configuration parameters of the SDK gesture recognition algorithm.

### Members

sets	Gesture notification sets in a bit-OR'ed value. The SDK will send gesture notifications only to the listed sets.
bodies	Trackable geometric body nodes in a bit-OR'ed value. The SDK will only track those listed bodies, unless the specified gesture set requires additional geometric node tracking.
blobs	Blobs in a bit-OR'ed value. The SDK will only return the listed blobs, data, and image.
alerts	Alerts in a bit-OR'ed value. The SDK will only signal the listed alerts.
nodeAlerts	Geometric nodes in a bit-OR'ed value. The SDK will only track these geometric nodes in the <a href="#">LABEL GEONODE ACTIVE/ LABEL GEONODE INACTIVE</a> alerts.
inputs	The required input stream formats. See the <code>PXCCapture::VideoStream::DataDesc</code> structure for details.



`activationDistance` The hand activation distance in centimeters.

### **Change History**

This structure is introduced in SDK API 1.0.



# Enumerator Reference

## PXCGesture::Alert::Label

### Description

The `Label` enumerator itemizes the supported alert messages. The SDK delivers the field of view series of alerts in a bit-OR'ed value.

### Name/Description

<code>LABEL_ANY</code>	This is not an alert but a value that the application can use to initialize any alert variable.
<code>LABEL_FOV_LEFT</code>	Signals when a body intercepts the left border of field of view.
<code>LABEL_FOV_RIGHT</code>	Signals when a body intercepts the right border of field of view.
<code>LABEL_FOV_TOP</code>	Signals when a body intercepts the top border of field of view.
<code>LABEL_FOV_BOTTOM</code>	Signals when a body intercepts the bottom border of field of view.
<code>LABEL_FOV_BLOCKED</code>	Signals when the camera field of view is completely blocked.
<code>LABEL_FOV_OK</code>	Signals when the camera field of view is recovered from any of the above field of view alerts.
<code>LABEL_GEONODE_ACTIVE</code>	Signals when a geometric node is in tracking.
<code>LABEL_GEONODE_INACTIVE</code>	Signals when a geometric node is lost tracking.

### Change History

This enumerator is introduced in SDK API 1.0.

### Remarks

The application may customize alert signalling by setting the `alerts` and `nodeAlerts` fields of the [ProfileInfo](#) structure. The SDK only sends alerts if they are described in the configuration.



## PXCGesture::Blob::Label

### Description

The `Label` enumerator itemizes the supported blobs.

### Name/Description

<code>LABEL_ANY</code>	Any blob, or not defined.
<code>LABEL_SCENE</code>	The blob image is the label map that the camera sees and the algorithm works on. The label map is a 8-bit gray image. Each tracking object is labeled separately by the pixel values. See the <a href="#">Blob</a> structure for definitions of specific object pixel values.

### Change History

This enumerator is introduced in SDK API 1.0.

## PXCGesture::GeoNode::Label

### Description

The `Label` enumerator itemizes the supported geometric nodes. Any geometric node is a bit-OR'ed value of a body label and a detailed label. The body label describes where the node is in terms of full-body skeletons, and the detailed label describes additional local-scope details. For example, a thumb fingertip is described as `LABEL_BODY_HAND_PRIMARY | LABEL_FINGER_THUMB`.

### Name/Description

<code>LABEL_ANY</code>	Any node, or not defined.
<code>LABEL_MASK_BODY</code>	The mask to obtain only the body label from a geometric node label.
<code>LABEL_MASK_DETAILS</code>	The mask to obtain only the detailed label from a geometric node label.

#### Full-body labels

<code>LABEL_BODY_ELBOW_PRIMARY</code>	The first tracked elbow in the field of view, not distinguishing left and right.
<code>LABEL_BODY_ELBOW_SECONDARY</code>	The second tracked elbow in the field of view, not



	distinguishing left and right.
LABEL_BODY_HAND_PRIMARY	The first tracked hand in the field of view, not distinguishing left and right.
LABEL_BODY_HAND_SECONDARY	The second tracked hand in the field of view, not distinguishing left and right.
LABEL_BODY_ELBOW_LEFT	The left elbow. See the <b>Remarks</b> section for known limitation.
LABEL_BODY_ELBOW_RIGHT	The right elbow. See the <b>Remarks</b> section for known limitation.
LABEL_BODY_HAND_LEFT	The left hand. See the <b>Remarks</b> section for known limitation.
LABEL_BODY_HAND_LEFT	The left hand. See the <b>Remarks</b> section for known limitation.

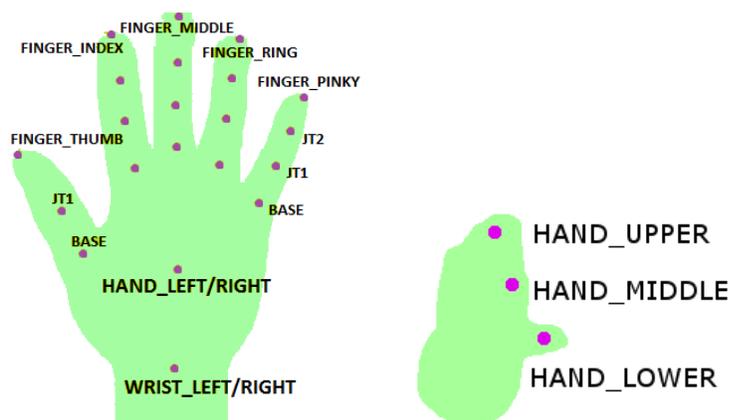
#### Hand detailed labels

LABEL_HAND_FINGERTIP	The fingertip or the furthest point from the palm center. With a hand fully opened, the fingertip is usually the middle fingertip. With the hand closed, the fingertip may point to one of the finger bases.
LABEL_HAND_UPPER	The upper point in a mittened hand model. See the <b>Remarks</b> section for details.
LABEL_HAND_MIDDLE	The middle point in a mittened hand model. See the <b>Remarks</b> section for details.
LABEL_HAND_LOWER	The lower point in a mittened hand model. See the <b>Remarks</b> section for details.
LABEL_FINGER_THUMB	The thumb fingertip.
LABEL_FINGER_INDEX	The index fingertip.
LABEL_FINGER_MIDDLE	The middle fingertip.
LABEL_FINGER_RING	The ring fingertip.
LABEL_FINGER_PINKY	The pinky fingertip.

#### Change History

This enumerator is introduced in SDK API 1.0.

#### Remarks



If the SDK is unable to determine the body side, the geometric node suffix **LEFT** and **RIGHT** refers to the first and second tracked objects, respectively. The application can use the `side` field in the [GeoNode](#) structure to further confirm the body side.

## PXCGesture::GeoNode::Openness

### Description

The `Openness` enumerator itemizes hand openness states.

### Name/Description

LABEL_OPENNESS_ANY	The hand openness state is unknown.
LABEL_OPEN	The hand is open.
LABEL_CLOSE	The hand is closed.

### Change History

This enumerator is introduced in SDK API 1.0.

## PXCGesture::GeoNode::Side

### Description

The `side` enumerator itemizes geometric node body side information.



## Name/Description

LABEL_UNKNOWN	The body side is unknown.
LABEL_LEFT	The geometric node is part of the left body.
LABEL_RIGHT	The geometric node is part of the right body.

## Change History

This enumerator is introduced in SDK API 1.0.

## PXCGesture::Gesture::Label

### Description

The `Label` enumerator itemizes predefined gestures. The enumerator uses the bit-OR'ed values of a gesture set enumerator and the detailed label to uniquely describe a gesture.

### Name/Description

LABEL_ANY	Any gesture or gesture not defined.
LABEL_MASK_SET	Use this mask to retrieve the set information of a gesture label.
LABEL_MASK_DETAILS	Use this mask to retrieve the detailed label information of a gesture label.

### Gesture Set Definitions

SET_HAND	Common hand gestures
SET_NAVIGATION	Common navigation gestures.
SET_POSE	Common pose gestures.
SET_CUSTOMIZED	Any customized gestures

### Detailed Navigation Gestures

LABEL_NAV_SWIPE_LEFT	Extend your fingers and swipe from right to left.
LABEL_NAV_SWIPE_RIGHT	Extend your fingers and swipe from left to right.
LABEL_NAV_SWIPE_UP	Extend your fingers and swipe from bottom to top.



LABEL\_NAV\_SWIPE\_DOWN      Extend your fingers and swipe from top to bottom.

#### Detailed Hand Gestures

LABEL\_HAND\_WAVE      Extend the index and the little finger and rest the thumb finger on the middle finger.

LABEL\_HAND\_CIRCLE      Extend all fingers and move the hand in a circle.

#### Detailed Pose Gestures

LABEL\_POSE\_THUMB\_UP      The thumb finger is up. Other fingers are curved.

LABEL\_POSE\_THUMB\_DOWN      The thumb finger is down. Other fingers are curved.

LABEL\_POSE\_PEACE      The index and middle fingers are extended forming a "V" letter. Other fingers are curved.

LABEL\_POSE\_BIG5      Extend all fingers open as if posing for number five.

### Change History

This enumerator is introduced in SDK API 1.0.

### Remarks

Swipes are basic navigation gestures. However, it is technically challenging to recognize swipes accurately. There are many cases a left swipe is exactly like a right swipe from the camera's view point. To avoid confusion, the user should perform the swipe gestures as follows:

*Imagine there is a virtual plane about 12 inches away from the camera. The swipes must first go into the plane, travel inside the plane, for example, from left to right, and then go out of the plane.*