



OpenNI2 Coding Instructions



Table of Contents

Overview	3
OpenNI Class.....	4
Basic Access to Devices	4
Basic Access to Video Streams	4
Event Driven Access to Devices	4
Error Information.....	5
Version Information	5
Device Class	5
Basic Operation	5
Constructor	6
Device::open()	6
Device::close().....	6
Device::isValid()	6
Obtaining Information on a Device	6
Specific Device Capabilities.....	7
Registration.....	7
FramSync.....	8
General Capabilities.....	8
File Devices	8
Recorder Class	8
PlaybackControl Class.....	9
VideoStream Class.....	10
Basic Functionality of VideoStreams.....	10
Creating and Initializing the VideoStream class	10
Polling based data reading	10
Event based data reading	11
Obtaining Information about VideoStreams	11
SensorInfo & VideoMode	11
Field of View	11
Min and Max Pixel Values.....	11
Configuring VideoStreams	11
VideoFrameRef Class	12
Accessing Frame Data	12



Overview

The OpenNI 2.0 API provides access to OpenNI2 compatible depth sensors. It allows an application to initialize a sensor and receive depth, RGB, and IR video streams from the device. It provides a single unified interface to sensors and .ONI recordings created with depth sensors.

Getting access to the depth streams requires the use of four main classes. Each of these classes will be discussed in detail in its own chapter:

- 1) **openni::OpenNI** - Provides a single static entry point to the API. Also provides access to devices, device related events, version and error information. Required to enable you to connect to a Device.
- 2) **openni::Device** - Provides an interface to a single sensor device connected to the system. Requires OpenNI to be initialized before it can be created. Devices provide access to Streams.
- 3) **openni::VideoStream** - Abstracts a single video stream. Obtained from a specific Device. Required to obtain VideoFrameRefs.
- 4) **openni::VideoFrameRef** - Abstracts a single video frame and related meta-data. Obtained from a specific Stream.

In addition to these main classes, various supporting classes and structures are provided for holding specific types of data. The classes are mainly used to encapsulate data:

Sensor Configuration Classes:

openni::DeviceInfo - This class records device wide configuration settings, including the device name, URI, USB VID/PID descriptors and vendor name.

openni::SensorInfo - This class stores the configuration settings that apply to a given sensor. A “sensor” in this context is either an IR camera, RGB camera, or depth camera. A device may contain several sensors.

openni::VideoMode - This class stores the resolution, framerate, and pixel format of a frame. It is used by VideoStream to set and track settings, by VideoFrameRef to track these settings, and by SensorInfo to provide a list of all valid modes.

openni::CameraSettings - Stores the settings for an RGB camera. Allows you to Enable/Disable auto white balance and auto exposure. Firmware support is necessary.

Data Storage Classes / Structures:

openni::Version - Stores a software version. Used by OpenNI to report its version.

openni::RGB888Pixel - This structure stores a single color pixel value.

openni::DepthPixel - This structure stores depth pixel value.

openni::Grayscale16Pixel - This structure stores IR pixel value.

openni::Array< T > - OpenNI provides a simple Array class that wraps the primitive arrays containing image data.

openni::CoordinateConverter - Allows conversion between Real World and Depth coordinates.

openni::Recorder - Provides to facilitate recording VideoStream data to an ONI file.

openni::DeviceConnectedListener - Connects listener device.

openni::openDeviceDisconnectedListener - Disconnect listener device.

openni::DeviceStateChangedListener - Change the state of listener device.



OpenNI Class

The first of the main classes that makes up OpenNI 2.0 is `openni::OpenNI`. This class provides a static entry point to the API. It is used to provide access to all devices in the system. It also makes available various device connection and disconnection events, as well as providing functions that allow for polling based access of all data streams.

Basic Access to Devices

The `OpenNI` class provides a static entry point to the API in the form of the `OpenNI::initialize()` function. This function initializes all available sensor drivers and scans the system for available devices. Any application using `OpenNI` should call this function prior to any other use of the API.

Once the `initialize` function has been run, it will become possible to create `Device` objects and use them to communicate with actual sensor hardware. The function `OpenNI::enumerateDevices()` returns a list of all available devices connected to the system.

When an application is ready to exit, the `OpenNI::shutdown()` function should be called to shut down all drivers and properly clean up.

Basic Access to Video Streams

A system of polling for stream access can be implemented by using the `OpenNI::waitForAnyStream()` function. This function takes a list of streams as one of its arguments. When called, it blocks until any of the streams in the list have new data available. It then returns a status code and indicates which stream has data available. This function can be used to implement a polling loop around new data being available.

Event Driven Access to Devices

The `OpenNI` class provides a framework for accessing devices in an event driven manner. `OpenNI` defines three events: `onDeviceConnected`, `onDeviceDisconnected`, and `onDeviceStateChanged`. An `onDeviceConnected` event is generated whenever a new device is connected and available through `OpenNI`. An `onDeviceDisconnected` event is generated when a device is removed from the system. An `onDeviceStateChanged` event is called whenever any settings of the `Device` are changed.

Listener classes can be added or removed from the list of event handlers by using the following methods:

```
Status OpenNI::addDeviceConnectedListener ( DeviceConnectedListener* pListener )
Status OpenNI::addDeviceDisconnectedListener( DeviceDisconnectedListener* pListener )
Status OpenNI::addDeviceStateChangedListener(DeviceStateChangedListener* pListener)
OpenNI::removeDeviceConnectedListener(DeviceConnectedListener* pListener)
```



```
OpenNI::removeDeviceDisconnectedListener(DeviceDisconnectedListener* pListener)  
OpenNI::removeDeviceStateChangedListener(DeviceStateChangedListener* pListener)
```

All three events provide a pointer to a `openNI::DeviceInfo` object. This object can be used to get details and identification of the device referred to by the event. Additionally, the `onDeviceStateChanged` event provides a pointer to a `DeviceState` object that can be used to see the new state of the device:

```
OpenNI::DeviceConnectedListener { virtual void onDeviceConnected (const DeviceInfo *)=0 }  
OpenNI::DeviceDisconnectedListener { virtual void onDeviceDisconnected (const DeviceInfo *)=0 }  
OpenNI::DeviceStateChangedListener { virtual void onDeviceStateChanged (const DeviceInfo *,  
DeviceState)=0 }
```

Error Information

Many functions in the SDK have a return type of “Status”. When an error occurs, status will contain a code that can be logged or displayed to a user. `OpenNI::getExtendedError()` method returns additional, human-readable information about the error.

Version Information

Version information for the API is provided by `OpenNI::getVersion()`. This function returns the version of the API that the application is presently interacting with.

Device Class

The `openni::Device` class provides an interface to a single physical hardware device (via a driver). It can also provide an interface to a simulated hardware device via a recorded ONI file taken from a physical device. The basic purpose of `Devices` is to provide `Streams`. The `Device` object is used to connect to and configure the underlying file or hardware device, and then `Streams` are created from that device.

Before the `Device` class can be connected to a hardware device, the device must be physically connected to a PC, and a driver must be installed. Drivers for OpenNI2 compatible sensors are installed along with OpenNI 2.0. If connecting to an ONI file instead of a physical device, it is only required that the ONI recording be available on the system running the application, and that the application have read access to this file. It is also required that the `openni::OpenNI::initialize()` function has been called prior to connecting to any devices. This will initialize the drivers and make the API aware of any devices connected.

Basic Operation



Constructor

The constructor for the Device class takes no arguments, and does not connect the Device to a physical hardware device. It simply creates the object in memory so that other functions can be called.

Device::open()

This function can be used to connect a Device to a physical hardware device. The open() function takes a single argument – the URI of a device. This function returns a status code indicating either success or what error occurred.

The simplest use of this function is to call it with the constant `openni::ANY_DEVICE` as the URI. Using this constant will cause the system to connect to any active hardware device. This is most useful when it can be safely assumed that there is exactly one active hardware device attached to the system.

If multiple sensors are attached to the system, then you should first call `OpenNI::enumerateDevices()` to obtain a list of all active devices. Then, find the desired device in the list, and obtain its URI by calling `DeviceInfo::getUri()`. Use the output from this function as the calling parameter for `Device::open()` in order to open that specific device.

If opening a file rather than a physical device, then the argument should be the path to the .ONI file.

Device::close()

The close() function properly shuts down the hardware device. As a best practice, any device that is opened should be closed. This will leave the driver and hardware device in a known state so that future applications will not have difficulty connecting to them.

Device::isValid()

The isValid() function can be used to determine whether there is currently an active device connected to this Device object.

Obtaining Information on a Device

It is possible to obtain basic information about a device. Information available includes name, vendor string, uri, and USB VID/PID. The `openni::DeviceInfo` class is provided to contain all of this information. It provides getter functions for each available information item. To obtain the `DeviceInfo` for a given device, call the `Device::getDeviceInfo()` function.

A device may be comprised of several sensors. For example, an OpenNI2 compatible device has an IR sensor, a color sensor and a depth sensor. Color sensor is not necessary since some devices do not contain this part. Each sensor



matches one stream type in OpenNI2. Streams can be opened on any existing sensor.

It is possible to get a list of sensors available from a device. The `Device:hasSensor()` function can be used to query whether a device provides a specific sensor. Possible sensors include:

`SENSOR_IR` – The IR video sensor

`SENSOR_COLOR` – The RGB-Color video sensor

`SENSOR_DEPTH` – The depth video sensor

If the desired sensor is available, then the `Device:getSensorInfo()` function can be used to get specific info on it. This will be encapsulated by a `SensorInfo` object. `SensorInfo` provides getters for the sensor type, and an array that contains all available video modes. Individual video modes are encapsulated by the `VideoMode` class.

Specific Device Capabilities

Registration

Some devices produce both depth and image streams. Usually, these streams are produced using two different physical cameras. Since the cameras are located at separate points in space, they will provide images of the same scene from two different angles. This results in objects in one image stream appearing to have a different apparent position from the same object in the other image stream. If the geometric relationship between the two cameras and the distance to the object in question are both known, then it is possible to mathematically transform one of the images to make it appear to have been taken from the same vantage point as the other. This process is referred to as Registration. Generally, in order to keep the color map's form, this process usually calculates the depth data, and makes depth map close to color map (D2C - depth to color).

The `Device` object provides the `isImageRegistrationSupported()` function to test whether the specific device it is connected to supports Registration. If Registration is supported, then `getImageRegistrationMode()` can be used to query the current status of this feature, and `setImageRegistrationMode()` can be used to set it to a specific value. The `openni::ImageRegistrationMode` enumeration provides the possible values that can be passed to these get and set functions:

`IMAGE_REGISTRATION_OFF` - Hardware registration features are disabled.

`IMAGE_REGISTRATION_DEPTH_TO_IMAGE` - The depth image is transformed to have the same apparent vantage point as the RGB image.

Note that since the two sensors will have areas where their field of view (FOV) does not overlap, there will generally be an area to one side of the Depthmap that is not shown as a result of enabling this feature. It is also common to see “shadows” or “holes” in the Depthmap where there are sudden edges in the depth geometry.



FramSync

When both a depth and color video stream is available, it is possible that the individual frames from each stream will not be exactly synchronized with each other. This can show up as a slight difference in frame rate, or a slight phase difference in frame arrival time, even when frame rate is exactly matched.

Some devices provide the capability to perform hardware synchronization on the two frames, in order to obtain frames that are separated from each other in time by some guaranteed maximum. Usually, this maximum is much less than the time between frames. This capability is referred to as FrameSync.

To enable or disable this ability, call the `setDepthColorSyncEnabled()`.

General Capabilities

Some devices have capabilities and settings other than FrameSync and Registration. OpenNI 2.0 provides a means to activate these with the `setProperty()` and `getProperty()` functions. The `setProperty()` function takes a numerical ID for the property, and a data value to set that ID to. The `getProperty()` function returns the current value stored at a specific ID.

File Devices

OpenNI 2.0 provides the ability to record the output of a device to a file (called an ONI file, and usually having the file extension `.oni`). This recording will optionally include all streams produced by the device.

Open file device by calling `Device::open()`. The difference between open device and open file device is that file device uses file path as URI. This functionality can be very useful for algorithm debugging. Live scenes are generally difficult or impossible to reproduce exactly. By using the recording functionality, the exact input can be fed to more than one algorithm, enabling debugging and performance comparisons. The functionality can also be useful for automated testing of applications, and for situations where insufficient cameras are available for all developers on a project – developers without a camera can develop and test code on a file recording. Finally, recordings can facilitate technical support by allowing a remote support representative to view the exact output from a customer camera, making it easier to spot problems.

The `PlaybackControl` class is used to access any file specific functionality for file devices. See the chapter devoted to that class for more information. To facilitate writing general purpose code that deals with both files and physical devices, the `Device::isFile()` function has been provided. This allows applications to determine whether a Device was created from a file before attempting to use a `PlaybackControl`.

Recorder Class

A simple Recorder class is provided to facilitate recording VideoStream data to an ONI file. ONI files are OpenNI's standard for recording the output of depth sensors. They can contain one or more streams of information (e.g. a depth and color stream recorded simultaneously by a PrimeSense sensor). They also contain the settings of the Device used to create that information. It is possible to instantiate Device objects from this file and then interact with them as if



they were physical devices.

Setting up a recorder

- 1) construct the recorder by calling its default constructor. This is no different than instantiating any other class.
- 2) call the Recorder::create() function on that Recorder, and provide a file name to record to. Any errors in creating and writing to the file will be returned as status codes from the create function.
- 3) provide the data streams to be recorded. This is done using Recorder::attach() function to attach the recorder to a given VideoStream. If you would like to record more than one stream, simply call attach multiple times, once for each VideoStream to be added.

Recording

After video streams are attached, recording will not commence until the Recorder::start() function is called. Once start() has been called, every frame generated by the streams recorded will be written to the ONI file. When you have finished recording, call the Recorder::stop() function to end the recording. Calling the Recorder::destroy() function will free any memory used by the recorder and ensure that files are written to disk properly.

Playback

ONI files can be played back by many OpenNI applications and utilities. To connect to them directly from application code, open a file device and read from it. Additional playback controls can be accessed by using the PlaybackControl object from file device (see PlaybackControl Class).

PlaybackControl Class

There are some actions that are only possible when dealing with a recorded file. These actions include seeking within a stream, determining how long a recording is, looping the recording, and changing playback speed. This functionality has been encapsulated by the PlaybackControl class.

Initializing

To use the PlaybackControl class, you must first instantiate and initialize a Device class from a file. Once a valid file Device has been created, you can acquire its internal PlaybackControl object by calling Device::getPlaybackControl(). The Device::IsFile() function can be used to determine whether a given Device was created from a file, if this is unknown

seek

Two functions are provided to allow seeking within a recording.

The PlaybackControl::seek() function takes a VideoStream pointer and a frameID as inputs. It then sets playback of the recording to frame indicated. If there are multiple streams in a recording, all streams will be set to the same point in time as the stream indicated – a specific stream is required as input only to provide context for the frameID.

The PlaybackControl::getNumberOfFrames() function can be used to determine how long a recording is. This is useful primarily to determine valid targets for the Seek function. It takes a stream pointer as input, and returns the number of frames in the recording for that specific stream. Note that it is possible for different streams in a recording to have differing number of frames, since frames will not always be synchronized.

Playback Speed

It is possible to vary the playback speed of a recording. This is useful when testing an algorithm with a large input data set, since it allows results to be obtained faster.



The `PlaybackControl::setSpeed()` function takes a floating point value as input. The input value is interpreted as a multiple of the speed the recording was made at. For example, if a recording was of a 30fps stream, and a value of 2.0 was passed to `setSpeed()`, then the stream would play back at 60fps. If a value of 0.5 was passed, the stream would play back at 15fps.

Setting the speed to 0.0 will cause the stream to run as fast as the host system is able to run. Setting the speed to -1 will cause the stream to be read manually – which is to say that the stream will be paused until a frame is read by the applications. Placing the recording in manual mode, and then reading in a tight loop, will accomplish something very similar to setting the speed to 0.0. Setting speed to 0.0 is therefore primarily useful for programs that used event driven data reading.

The `PlaybackControl::getSpeed()` function will provide the most recent value that was set with `setSpeed()` (ie, the active speed value).

Playback Looping

A playback looping function has been provided by API. The `PlaybackControl::setRepeatEnabled()` function can be used to turn looping on and off. If a value of `TRUE` is passed to `setRepeatEnabled`, then the recording will start over at the first frame after the last frame is read. If a value of `FALSE` is passed, then no more frames will be generated after the recording is over.

`PlaybackControl::getRepeatEnabled()` can be used to query the current repeat value.

VideoStream Class

The `VideoStream` class encapsulates all data streams created by the `Device` class. It allows you to request that specific data flows be started, stopped, and configured. It also allows for configuration of parameters that exist at the stream (as opposed to `Device`) level.

Basic Functionality of VideoStreams

Creating and Initializing the VideoStream class

Calling the default constructor of the `VideoStream` class will create an empty, uninitialized `VideoStream` object. Before it can be used, this object must be initialized with the `VideoStream::create()` function. The `create()` function requires a valid initialized device. Once created, you should call the `VideoStream::start()` function to start the flow of data. A `VideoStream::stop()` function is provided to stop the flow of data.。

Polling based data reading

Once a `VideoStream` has been created, data can be read from it directly with the `VideoStream::readFrame()` function. If new data is available, this function will provide access to the most recent `VideoFrameRef` generated by the `VideoStream`. If no new frame is ready yet, then this function will block until a new frame is ready.



Note that if reading from a recording with looping turned off, this function will block forever once the last frame has been reached.

Event based data reading

It is also possible to read the data from a VideoStream in an event driven manner. To do this, you must create a class that extends the VideoStream::Listener class. This class should implement a function called onNewFrame(). Once you create this class, instantiate it, and pass it to the VideoStream::addListener() function. When a new frame is available, the onNewFrame() function of your listener will be called. You will still be required to call readFrame().

Obtaining Information about VideoStreams

SensorInfo & VideoMode

The SensorInfo and VideoMode classes are provided to keep track of information about VideoStreams. A VideoMode encapsulates the frame rate, resolution, and pixel format of a VideoStream. SensorInfo contains the type of sensor used to produce a VideoStream, and a list of VideoMode objects that each contains a valid set of parameters for the stream. By iterating through the list of VideoMode objects, it is possible to determine all possible modes for the sensor producing a given VideoStream.

Use the VideoStream::getSensorInfo to obtain the sensor info object that corresponds to a given VideoStream.

Field of View

Functions are provided to determine the field of view of the sensor used to create a VideoStream. Use the getHorizontalFieldOfView() and getVerticalFieldOfView() functions to determine the field of view used to create a stream. This value will be reported in radians.

Min and Max Pixel Values

For depth streams it, it is often useful to know the minimum and maximum possible values that a pixel can contain. Use the getMinPixelValue() and getMaxPixelValue() functions to obtain this information.

Configuring VideoStreams

Video Modes

It is possible to set the frame rate, resolution, and pixel type of a given stream. To do this, use the setVideoMode() function. Before doing this, you will first need to obtain the SensorInfo for the VideoStream to be configured, so that you can choose a valid VideoMode.



Cropping

If a given sensor supports cropping, the VideoStream provides a means to control it. Use the VideoStream::isCroppingSupported() to determine whether a sensor supports cropping.

Mirroring

Mirroring causes the VideoStream to appear as if seen in a mirror – ie, the image is transformed by reflecting all pixels across the vertical axis. To enable or disable mirroring, use the VideoStream::setMirroringEnabled() function. Pass in TRUE to turn on mirroring, and FALSE to turn it off. The current mirroring setting can be queried by using the getMirroringEnabled() function.

General Properties

At the firmware level, most sensor settings are stored as address/value pairs. These can be manipulated directly with the SetProperty and GetProperty functions. These functions are used internally by the SDK to implement cropping, mirroring, etc. They will not need to be used frequently by application code, since most of the useful properties are wrapped by friendlier functions.。

VideoFrameRef Class

The VideoFrameRef class encapsulates all data related to a single frame read from a VideoStream. It is the basic class that VideoStream uses to return each new frame. It provides access to the underlying array that contains the frame data, as well as any metadata that is required to work with the frame.

VideoFrameRef objects are obtained from calling VideoStream::readFrame().

VideoFrameRef data can come from IR cameras, RGB cameras, or depth cameras. If required, the getSensorType() function can be used to determine which type of sensor generated the frame. This will return a SensorType, an enumeration that provides a code for each possible sensor type.

Accessing Frame Data

The VideoFrameRef includes the VideoFrameRef::getData() function that returns a pointer directly to the underlying frame data. This will be a void pointer, so it must be cast using the data type of the individual pixels in order to be properly indexed.

Metadata-

Several items of metadata are provided with each frame to facilitate working with the data itself.

Cropping data-

The VideoFrameRef knows the cropping settings for the VideoStream used to create it. It is possible to determine the



origin of the cropping window, size of the cropping window, and whether cropping was enabled when the frame was generated. The functions to do this include: `getCropOriginX()`, `getCropOriginY()`, `getCroppingEnabled()`. The cropping window size will be equal to the size of the frame if cropping is enabled, so the method for determining that is the same as the method for determining frame resolution.

TimeStamp-

Each frame of data will be stamped with a timestamp. This value is measured in microseconds from some arbitrary zero value. The difference in time stamps between two frames from the same stream will be the time difference between those frames. All streams in the same device are guaranteed to use the same zero, so differences between time stamps can also be used to compare frames from different streams.

The current implementation of OpenNI 2.0 is to start the timestamp zero as the time of the first frame of the stream. This is not guaranteed to be the case in every implementation, however, so application code should only use timestamp deltas. The timestamp value itself should not be used as any kind of absolute time reference.

Frame Index-

In addition to timestamps, frames are provided sequential Frame Index numbers. This is useful for determining sequence of frames, and for knowing how many frames came between two frames. If two streams have been synchronized by using the `Device::setColorDepthSync()` function, then the frame indexes of corresponding frames is guaranteed to match.

If synchronization is not enabled, then frame indexes are NOT guaranteed to match. In this case, it is more useful to use the timestamp to determine where frames are in relation to each other.

Video Modes-

`VideoFrameRef::getVideoMode()` can be used to determine the video mode settings of the sensor that created the frame at the time of its creation. This information includes the pixel format and resolution of the image, as well as the frame rate the camera was running at when the image was taken.

Data Size-

The `getDataSize()` function can be used to determine the size of all the data contained in the image array. This is useful if you need to allocate a buffer to store frame, or a number of frames. Note that this is the data size for entire array. Use `VideoMode::getPixelFormat()` to determine the size of individual array elements.

Image Resolution-

For convenience, `getHeight()` and `getWidth()` functions are provided to easily determine the resolution of the frame. This data could also be obtained with `VideoFrameRef::getVideoMode().getResolutionX()` and `VideoFrameRef::getVideoMode().getResolutionY()`, but these values are required very frequently, so the above calls would be inefficient and awkward.

Data Validity-

The `VideoFrameRef::isValid()` function is provided to determine whether a `VideoFrameRef` contains actual valid data. This function will return false if called between the initial construction of the `VideoFrameRef` and the first time data is loaded from an actual `VideoStream`.

**Sensor Type-**

The type of sensor used to generate the frame data can be determined by calling `getSensorType()`. This will return a `SensorType` – an enumeration that assigns constants to each possible sensor type. Possible values include:

`SENSOR_IR` – for an image taken with an IR camera

`SENSOR_COLOR` – for an image taken with an RGB camera

`SENSOR_DEPTH` – for an image taken with a depth sensor

Array Stride-

The stride of the array containing the frame can be obtained with the `getStrideInBytes()` function. This provides the size of each row of the data array in bytes. This is primarily useful to allow two dimensional indexing of the image data.