User Guide

Table of Contents

License Notice

OpenNI is written and distributed under the GNU Lesser General Public License (LGPL) which means that its source code is freely-distributed and available to the general public.

You can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or any later version.

OpenNI is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details: http://www.gnu.org/licenses/.

Overview

Natural Interaction

The term Natural Interaction (NI) refers to a concept where Human-device interaction is based on human senses, mostly focused on hearing and vision. Human device NI paradigms render such external peripherals as remote controls, keypads or a mouse obsolete. Examples of everyday NI usage include:

- Speech and command recognition, where devices receive instructions via vocal commands.
- Hand gestures, where pre-defined hand gestures are recognized and interpreted to activate and control devices. For example, hand gesture control enables users to manage living room consumer electronics with their bare hands.
- Body Motion Tracking, where full body motion is tracked, analyzed and interpreted for gaming purposes.

What is OpenNI?

OpenNI (Open Natural **I**nteraction) is a multi-language, cross-platform framework that defines APIs for writing applications utilizing Natural Interaction. OpenNI APIs are composed of a set of interfaces for writing NI applications. The main purpose of OpenNI is to form a standard API that enables communication with both:

- Vision and audio sensors (the devices that 'see' and 'hear' the figures and their surroundings.)
- Vision and audio perception middleware (the software components that analyze the audio and visual data that is recorded from the scene, and comprehend it). For example, software that receives visual data, such as an image, returns the location of the palm of a hand detected within the image.

OpenNI supplies a set of APIs to be implemented by the sensor devices, and a set of APIs to be implemented by the middleware components. By breaking the dependency between the sensor and the middleware, OpenNI's API enables applications to be written and ported with no additional effort to operate on top of different middleware modules ("write once, deploy everywhere"). OpenNI's API also enables middleware developers to write algorithms on top of raw data formats, regardless of which sensor device has produced them, and offers sensor manufacturers the capability to build sensors that power any OpenNI compliant application.

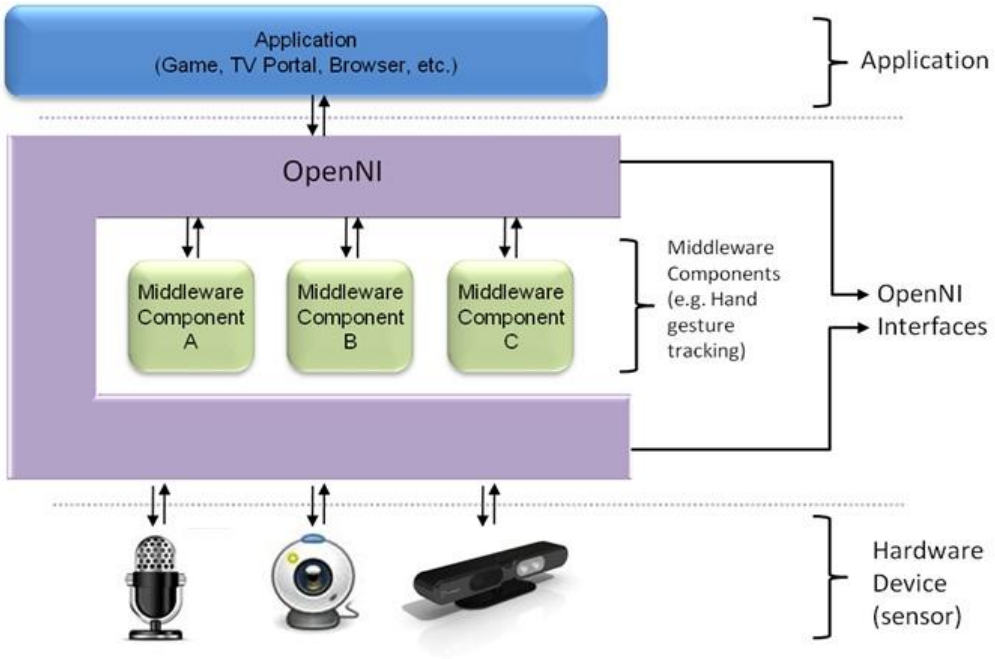
The OpenNI standard API enables natural-interaction application developers to track real-life (3D) scenes by utilizing data types that are calculated from the input of a sensor (for example, representation of a full body, representation of a hand location, an array of the pixels in a depth map and so on). Applications can be written regardless of the sensor or middleware providers.

OpenNI is an open source API that is publicly available at www.OpenNI.org.

Abstract Layered View

Figure 1 below displays a three-layered view of the OpenNI Concept with each layer representing an integral element:

- Top: Represents the software that implements natural interaction applications on top of OpenNI.
- Middle: Represents OpenNI, providing communication interfaces that interact with both the sensors and the middleware components, that analyze the data from the sensor.
- Bottom: Shows the hardware devices that capture the visual and audio elements of the scene.



Concepts

Modules

The OpenNI Framework is an abstract layer that provides the interface for both physical devices and middleware components. The API enables multiple components to be registered in the OpenNI framework. These components are referred to as modules, and are used to produce and process the sensory data. Selecting the required hardware device component, or middleware component is easy and flexible.

The modules that are currently supported are:

Sensor Modules

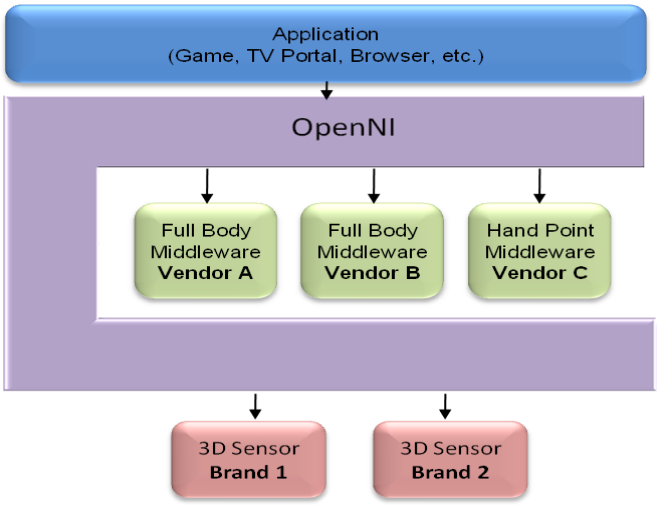
- **3D sensor**
- **RGB camera**
- **IR camera**
- **Audio device** (a microphone or an array of microphones)

Middleware components

- **Full body analysis middleware**: a software component that processes sensory data and generates body related information (typically data structure that describes joints, orientation, center of mass, and so on).
- **Hand point analysis middleware:** a software component that processes sensory data and generates the location of a hand point
- **Gesture detection middleware**: a software component that identifies predefined gestures (for example, a waving hand) and alerts the application.
- **Scene Analyzer middleware:** a software component that analyzes the image of the scene in order to produce such information as:
  - The separation between the foreground of the scene (meaning, the figures) and the background
  - The coordinates of the floor plane
  - The individual identification of figures in the scene.

Example

The illustration below displays a scenario in which 5 modules are registered to work with an OpenNI installation. Two of the modules are 3D sensors that are connected to the host. The other three are middleware components, including two that produce a person's full-body data, and one that handles hand point tracking.



Modules, whether software or actual devices that wish to be OpenNI compliant, must implement certain interfaces.

Production Nodes

"Meaningful"3D data is defined as data that can comprehend, understand and translate the scene. Creating meaningful 3D data is a complex task. Typically, this begins by using a sensor device that produces a form of raw output data. Often, this data is a depth map, where each pixel is represented by its distance from the sensor. Dedicated middleware is then used to process this raw output, and produce a higher-level output, which can be understood and used by the application.

OpenNI defines **Production Nodes**, which are a set of components that have a productive role in the data creation process required for Natural Interaction based applications. Each production node encapsulates the functionality that relates to the generation of the specific data type. These production nodes are the fundamental elements of the OpenNI interface provided for the applications. However, the API of the production nodes only defines the language. The logic of data generation must be implemented by the modules that plug into OpenNI.

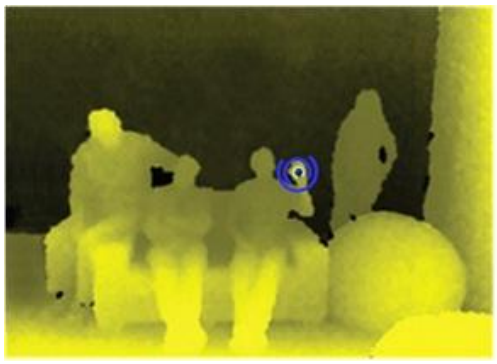
For example, there is a production node that represents the functionality of generating hand-point data. The logic of hand-point data generation must come from an external middleware component that is both plugged into OpenNI, and also has the knowledge of how to produce such data.

In principal, each production node is a standalone unit that generates a specific type of data, and can provide it to any object, whether it be another production node, or the application itself. However, typically some production nodes always use other production nodes that represent lower level data types, analyze this lower level data and produce higher level data for the application.

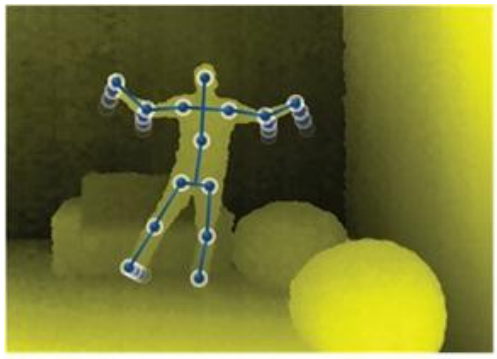
Example
*The application wants to track the motion of a human figure in a 3D scene. This requires a production node that provides body data for the application, or, in other words, a **user generator**. This specific user generator obtains its data from a **depth generator**. A depth generator is a production node that is implemented by a sensor, which takes raw sensory data from a depth sensor (for example, a stream of X frames per second) and outputs a depth map.*

Common examples of higher level output are as described and illustrated below:

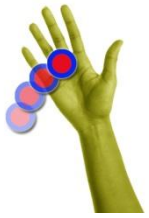
- The location of a user's hand.
  The output can be either the center of the palm (often referred to as 'hand point') or the finger tips.



- The identification of a figure within the scene.
  The output is the current location and orientation of the joints of this figure (often referred to as 'body data').



- The identification of a hand gesture (for example, waving).
  The output is an alert to the application that a specific hand gesture has occurred.



Production Node Types

Each production node in OpenNI has a type and belongs to one of the following categories:

- Sensor-Related Production Nodes
- Middleware-Related Production Nodes

The production node types that are currently supported in OpenNI are:

Sensor-Related Production Nodes

- **Device**: A node that represents a physical device (for example, a depth sensor, or an RGB camera). The main role of this node is to enable device configuration.
- **Depth Generator:** A node that generates a depth-map. This node should be implemented by any 3D sensor that wishes to be certified as OpenNI compliant.
- **Image Generator**: A node that generates colored image-maps. This node should be implemented by any color sensor that wishes to be certified as OpenNI compliant
- **IR Generator:** A node that generates IR image-maps. This node should be implemented by any IR sensor that wishes to be certified as OpenNI compliant.
- **Audio Generator:** A node that generates an audio stream. This node should be implemented by any audio device that wishes to be certified as OpenNI compliant.

Middleware-Related Production Nodes

- **Gestures Alert Generator**: Generates callbacks to the application when specific gestures are identified.
- **Scene Analyzer**: Analyzes a scene, including the separation of the foreground from the background, identification of figures in the scene, and detection of the floor plane.
  The Scene Analyzer's main output is a labeled depth map, in which each pixel holds a label that states whether it represents a figure, or it is part of the background.
- **Hand Point Generator**: Supports hand detection and tracking. This node generates callbacks that provide alerts when a hand point (meaning, a palm) is detected, and when a hand point currently being tracked, changes its location.
- **User Generator**: Generates a representation of a (full or partial) body in the 3D scene.

For recording purposes, the following production node types are supported:

- **Recorder:** Implements data recordings
- **Player:** Reads data from a recording and plays it
- **Codec:** Used to compress and decompress data in recordings

Production Chains

As explained previously, several modules (middleware components and sensors) can be simultaneously registered to a single OpenNI implementation. This topology offers applications the flexibility to select the specific sensor devices and middleware components with which to produce and process the data.

What is a production chain?

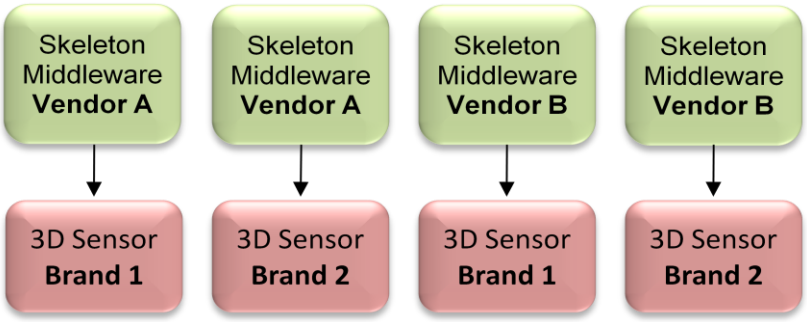
In the Production Nodes section, an example was presented in which a **user generator** type of production node is created by the application. In order to produce body data, this production node uses a lower level depth generator, which reads raw data from a sensor. In the example below, the sequence of nodes (user generator => depth generator), is reliant on each other in order to produce the required body data, and is called a **production chain**.

Different vendors (brand names) can supply their own implementations of the same type of production node.

Example:
*Brand A provides an implementation (a module) of user generator middleware. Brand B provides separate middleware that implements a user generator. Both generators are available to the application developer. OpenNI enables the application to define which modules, or **production chain**, to use. The OpenNI interface enumerates all possible production chains according to the registered modules. The application can then choose one of these chains, based on the preference for a specific brand, component, or version and so on, and create it.*
***Note:** An application can also be non-specific, and request the first enumerated production chain from OpenNI.*

Typically, an application is only interested in the top product node of each chain. This is the node that outputs the required data on a practical level, for example, a hand point generator. OpenNI enables the application to use a single node, without being aware of the production chain beneath this node. For advanced tweaking, there is an option to access this chain, and configure each of the nodes.

For example, if we look at the system illustration that was presented earlier, it described multiple registered modules and devices. Once an application requests a user generator, OpenNI returns the following four optional production chains to be used to obtain body data:



The above illustration shows a scenario in which the following modules were registered to OpenNI:

- Two body middleware components, each being different brands.
- Two 3D sensors, each being two different brands

This illustration displays the four optional production chains that were found for this implementation. Each chain represents a possible combination of a body middleware component and a 3D sensor device. OpenNI offers the application the option to choose from the above four production chain alternatives.

Capabilities

The Capabilities mechanism supports the flexibility of the registration of multiple middleware components and devices to OpenNI. OpenNI acknowledges that different providers may have varying capabilities and configuration options for their production nodes, and therefore, certain non-mandatory extensions are defined by the OpenNI API. These optional extensions to the API are called **Capabilities,** and reveal additional functionality, enabling providers to decide individually whether to implement an extension. A production node can be asked whether it supports a specific capability. If it does, those functions can be called for that specific node.

OpenNI is released with a specific set of capabilities, with the option of adding further capabilities in the future. Each module can declare the capabilities it supports. Furthermore, when requesting enumeration of production chains, the application can specify the capabilities that should be supported as criteria. Only modules that support the requested capability are returned by the enumeration.

Currently supported capabilities:

- **Alternative View:** Enables any type of map generator (depth, image, IR) to transform its data to appear as if the sensor is placed in another location (represented by another production node, usually another sensor).
- **Cropping:** Enables a map generator (depth, image, IR) to output a selected area of the frame as opposed to the entire frame. When cropping is enabled, the size of the generated map is reduced to fit a lower resolution (less pixels). For example, if the map generator is working in VGA resolution (640x480) and the application chooses to crop at 300x200, the next pixel row will begin after 300 pixels. Cropping can be very useful for performance boosting.
- **Frame Sync:** Enables two sensors producing frame data (for example, depth and image) to synchronize their frames so that they arrive at the same time.
- **Mirror:** Enables mirroring of the data produced by a generator. Mirroring is useful if the sensor is placed in front of the user, as the image captured by the sensor is mirrored, so the right hand appears as the left hand of the mirrored figure.
- **Pose Detection:** Enables a user generator to recognize when the user is posed in a specific position.
- **Skeleton:** Enables a user generator to output the skeletal data of the user. This data includes the location of the skeletal joints, the ability to track skeleton positions and the user calibration capabilities.
- **User Position:** Enables a Depth Generator to optimize the output depth map that is generated for a specific area of the scene.
- **Error State:** Enables a node to report that it is in "Error" status, meaning that on a practical level, the node may not function properly.
- **Lock Aware:** Enables a node to be locked outside the context boundary. For more information, see Sharing Devices between Applications and Locking Nodes.

Generating and Reading Data

Generating Data

Production nodes that also produce data are called Generators, as discussed previously. Once these are created, they do not immediately start generating data, to enable the application to set the required configuration. This ensures that once the object begins streaming data to the application, the data is generated according to the required configuration. Data Generators do not actually produce any data until specifically asked to do so. The **xn::Generator::StartGenerating()** function is used to begin generating. The application may also want to stop the data generation without destroying the node, in order to store the configuration, and can do this using the **xn::Generator::StopGenerating** function.

Reading Data

Data Generators constantly receive new data. However, the application may still be using older data (for example, the previous frame of the depth map). As a result of this, any generator should internally store new data, until explicitly requested to update to the newest available data.

This means that Data Generators "hide" new data internally, until explicitly requested to expose the most updated data to the application, using the **UpdateData** request function. OpenNI enables the application to wait for new data to be available, and then update it using the **xn::Generator::WaitAndUpdateData()** function.

In certain cases, the application holds more than one node, and wants all the nodes to be updated. OpenNI provides several functions to do this, according to the specifications of what should occur before the UpdateData occurs:

- **xn::Context::WaitAnyUpdateAll()**: Waits for any node to have new data. Once new data is available from any node, all nodes are updated.
- **xn::Context::WaitOneUpdateAll()**: Waits for a specific node to have new data. Once new data is available from this node, all nodes are updated. This is especially useful when several nodes are producing data, but only one determines the progress of the application.
- **xn::Context::WaitNoneUpdateAll()**: Does not wait for anything. All nodes are immediately updated.
- **xn::Context::WaitAndUpdateAll()**: Waits for all nodes to have new data available, and then updates them.

The above four functions exit after a timeout of two seconds. It is strongly advised that you use one of the **xn::Context::Wait[…]UpdateAll()** functions, unless you only need to update a specific node. In addition to updating all the nodes, these functions have the following additional benefits:

- If nodes depend on each other, the function guarantees that the "needed" node (the lower-level node generating the data for another node) is updated before the "needing" node.
- When playing data from a recording, the function reads data from the recording until the condition is met.
- If a recorder exists, the function automatically records the data from all nodes added to this recorder.

Mock Nodes

OpenNI provides a mock implementation for nodes. A mock implementation of a node does not contain any logic for generating data. Instead, it allows an outside component (such as an application, or another node implementation) feed it configuration changes and data. Mock nodes are rarely required by the application, and are usually used by player nodes to simulate actual nodes when reading data from a recording.

Sharing Devices between Applications and Locking Nodes

In most cases, the data generated by OpenNI nodes comes from a hardware device. A hardware device can usually be set to more than one configuration. Therefore, if several applications all using the same hardware device are running simultaneously, their configuration must be synchronized.

However, usually, when writing an application, it is impossible to know what other applications may be executed simultaneously, and as such, synchronization of the configuration is not possible. Additionally, sometimes it is essential that an application use a specific configuration, and no other.

OpenNI has two modes that enable multiple applications to share a hardware device:

- **Full Sharing** (default): In this mode, the application declares that it can handle any configuration of this node. OpenNI interface enables registering to callback functions of any configuration change, so the application can be notified whenever a configuration changes (by the same application, or by another application using the same hardware device).
- **Locking Configuration**: In this mode, an application declares that it wants to lock the current configuration of a specific node. OpenNI will therefore not allow "Set" functions to be called on this node. If the node represents a hardware device (or anything else that can be shared between processes), it should implement the "Lock Aware" capability, which enables locking across process boundaries.

Note: When a node is locked, the locking application receives a lock handle. Other than using this handle to unlock the node, the handle can also be used to change the node configuration without releasing the lock (in order that the node configuration will not be "stolen" by another application).

Licensing

OpenNI provides a simple licensing mechanism that can be used by modules and applications. An OpenNI context object, which is an object that holds the complete state of applications using OpenNI, holds a list of currently loaded licenses. This list can be accessed at any stage to search for a specific license.

A license is composed of a vendor name and a license key. Vendors who want to use this mechanism can utilize their own proprietary format for the key.

The license mechanism is used by modules, to ensure that they are only used by authorized applications A module of a particular vendor can be installed on a specific machine, and only be accessible if the license is provided by the application using the module. During the enumeration process, when OpenNI searches for valid production chains, the module can check the licenses list. If the requested license is not registered, the module is able to hide itself, meaning that it will return zero results and therefore not be counted as a possible production chain.

OpenNI also provides a global registry for license keys, which are loaded whenever a context is initialized. Most modules require a license key from the user during installation. The license provided by the user can then be added to the global license registry, using the **niLicense** command-line tool, which can also be used to remove licenses.

Additionally, applications sometimes have private licenses for a module, meaning that this module can only be activated using this application (preventing other applications from using it).

General Framework Utilities

In addition to the formal OpenNI API, a set of general framework utilities is also published, intended mainly to ease the portability over various architectures and operating systems. The utilities include:

- A USB access abstract layer (provided with a driver for Microsoft Windows)
- Certain basic data type implementation (including list, hash, and so on)
- Log and dump systems
- Memory and performance profiling
- Events (enabling callbacks to be registered to a specific event)
- Scheduling of tasks

Those utilities are available to any application using OpenNI. However, these utilities are not part of standard OpenNI, and as such, backwards compatibility is only guaranteed to a certain extent.

Recording

Recordings are a powerful debug tool. They enable full capture of the data and the ability to later stream it back so that applications can simulate an exact replica of the situation to be debugged.

OpenNI supports recordings of the production nodes chain; both the entire configuration of each node, and all data streamed from a node.

OpenNI has a framework for recording data and for playing it back (using mock nodes). It also comes with the **nimRecorder** module, which defines a new file format (".ONI") - and implements a Recorder node and a Player node for this format.

Production Node Error Status

Each production node has an error status, indicating whether it is currently functional. For example, a device node may not be functional if the device is disconnected from the host machine. The default error state is always OK, unless an **Error Status** capability is implemented. This capability allows the production node to change its error status if an error occurs. A node that does not implement this capability always has a status of "OK".

An application can check the error status of each node although it mostly only needs to know if any node has an error status, and is less interested which node (other than for user notification purposes). In order to receive notifications about a change in the error status of a node, the application can register to a callback that will alert of any change in a node's error status.

OpenNI aggregates the error statuses of all the nodes together into a single error status, called Global Error Status. This makes it easier for applications to find out about the current state of a node or nodes. A global error status of **XN_STATUS_OK** means that all the nodes are OK. If only one node has an error status, that error status becomes the global error status (for example, if one sensor is disconnected, the OpenNI global error status is **XN_STATUS_DEVICE_NOT_CONNECTED**). If more than one node has an error status, the global error status is **XN_STATUS_MULTIPLE_NODES_ERROR**. In such a situation, the application can review all nodes and check which one has an error status, and why.

Backwards Compatibility

OpenNI declares full backwards compatibility. This means that every application developed over any version of OpenNI, can also work with every future OpenNI version, without requiring recompilation.

On a practical level, this means that each computer should ideally have the latest OpenNI version installed on it. If not this, then the latest OpenNI version required by any of the applications installed on this computer. In order to achieve this, we recommend that the application installation should also install OpenNI.

Getting Started

Supported Platforms

OpenNI is available on the following platforms:

- Windows XP and later, for 32-bit only
- Linux Ubuntu 10.10 and later, for x86

Main Objects

The Context Object

The context is the main object in OpenNI. A context is an object that holds the complete state of applications using OpenNI, including all the production chains used by the application. The same application can create more than one context, but the contexts cannot share information. For example, a middleware node cannot use a device node from another context. The context must be initialized once, prior to its initial use. At this point, all plugged-in modules are loaded and analyzed. To free the memory used by the context, the application should call the shutdown function.

Metadata Objects

OpenNI Metadata objects encapsulate a set of properties that relate to specific data alongside the data itself. For example, typical property of a depth map is the resolution of this map (for example, the number of pixels on both an X and a Y axis). Each generator that produces data has its own specific metadata object.

In addition, the metadata objects play an important role in recording the configuration of a node at the time the corresponding data was generated. Sometimes, while reading data from a node, an application changes the node configuration. This can cause inconsistencies that may cause errors in the application, if not handled properly.

Example
A depth generator is configured to produce depth maps in QVGA resolution (320x240 pixels), and the application constantly reads data from it. At some point, the application changes the node output resolution to VGA (640x480 pixels). Until a new frame arrives, the application may encounter inconsistency where calling the ***xn::DepthGenerator::GetDepthMap()*** *function will return a QVGA map, but calling the* ***xn::DepthGenerator::GetMapOutputMode()*** *function will return that the current resolution is a VGA map. This can result in the application assuming that the depth map that was received is in VGA resolution, and therefore try to access nonexistent pixels.*

The solution is as follows: Each node has its metadata object, that records the properties of the data when it was read. In the above case, the correct way to handle data would be to get the metadata object, and read both the real data (in this case, a QVGA depth map) and its corresponding resolution from this object.

Configuration Changes

Each configuration option in OpenNI interfaces comprises the following functions:

- A **Set** function for modifying the configuration.
- A **Get** function for providing the current value.
- **Register** and **Unregister** functions, enabling registration for callback functions to be called when this option changes.

Data Generators

Map Generator

The basic interface for all data generators that produce any type of map.

Main functionalities:

- **Output Mode property**: Controls the configuration by which to generate the map
- **Cropping capability**
- **Alternative Viewpoint capability**
- **Frame Sync capability**

Depth Generator

An object that generates a depth map.

Main Functionalities:

- **Get depth map**: Provides the depth map
- **Get Device Max Depth**: The maximum distance available for this depth generator
- **Field of View property**: Configures the values of the horizontal and vertical angles of the sensor

- **User Position capability**

Image Generator

A Map Generator that generates a color image map.

Main Functionalities:

- **Get Image Map**: Provides the color image map
- **Pixel format property**

IR Generator

A map generator that generates an IR map.

Main Functionality:

- Get IR Map: Provides the current IR map

Scene Analyzer

A map generator that gets raw sensory data and generates a map with labels that clarify the scene.

Main Functionalities:

- Get Label Map: Provides a map in which each pixel has a meaningful label (i.e. figure 1, figure 2, background, and so on)
- Get Floor: get the coordinates of the floor plane

Audio Generator

An object that generates Audio data.

Main Functionalities:

- **Get Audio Buffer**
- **Wave Output Modes property**: Configure the audio output, including sample rate, number of channels and bits-per-sample

Gesture Generator

An object that enables specific body or hand gesture tracking

Main Functionalities:

- **Add/Remove Gesture**: Turn on/off a gesture. Once turned on, the generator will start looking for this gesture.
- **Get Active Gestures**: Provides the names of the gestures that are currently active
- **Register/Unregister Gesture callbacks**
- **Register/Unregister Gesture change**

Hand Point Generator

An object that enables hand point tracking.

Main Functionalities:

- **Start/Stop Tracking**: Start/stop tracking a specific hand (according to its position)

- **Register/Unregister Hand Callbacks**: The following actions will generate hand callbacks:
  - When a new hand is created
  - When an existing hand is in a new position
  - When an existing hand disappears

User Generator

An object that generates data relating to a figure in the scene.

Main Functionalities:

- **Get Number of Users**: Provides the number of users currently detected in the scene
- **Get Users**: Provides the current users
- **Get User CoM**: Returns the location of the center of mass of the user
- **Get User Pixels**: Provides the pixels that represent the user. The output is a map of the pixels of the entire scene, where the pixels that represent the body are labeled User ID.
- **Register/Unregister user callbacks**: The following actions will generate user callbacks:
  - When a new user is identified
  - When an existing user disappears

Creating an empty project that uses OpenNI

1. Open a new project or an existing one with which you want to use OpenNI.
2. In the Visual Studio menu, open the Project menu and choose Project properties.
3. In the C/C++ section, under the **General node, select =>Additional Include Directories** and add "$(OPEN_NI_INCLUDE)". This is an environment variable that points to the location of the OpenNI **Include** directory. (The default location is: C:\Program files\OpenNI\Include.)
4. In the Linker section, under the General node, select Additional Library Directories and add "$(OPEN_NI_LIB)". This is an environment variable that points to the location of the OpenNI include directory. (The default location is: C:\Program files\OpenNI\Lib.)
5. In the Linker section, under the Input node, select Additional Dependencies and add OpenNI.lib.
6. If you wish to use an XML file to configure OpenNI, you can start from the basic XML file that can be found in the OpenNI Data folder. (The default location is: C:\Program files\OpenNI\Data.) For further information about OpenNI xml scripts, see Xml Scripts.
7. Ensure that you add the **Additional Include** and **Library** directories to both your Release and Debug configurations.
8. Your code files should include XnOpenNI.h if using the C interface, or XnCppWrapper.h if using the C++ interface.

Basic Functions: Initialize, Create a Node and Read Data

The following code illustrates the basic functionality of OpenNI. It initializes a Context object, and creates and reads data from a single Depth node.

```
XnStatus nRetVal = XN_STATUS_OK;

xn::Context context;

// Initialize context object
nRetVal = context.Init();
// TODO: check error code

// Create a DepthGenerator node
xn::DepthGenerator depth;
nRetVal = depth.Create(context);
// TODO: check error code

// Make it start generating data
nRetVal = context.StartGeneratingAll();
// TODO: check error code

// Main loop
while (bShouldRun)
{
    // Wait for new data to be available
    nRetVal = context.WaitOneUpdateAll(depth);
    if (nRetVal != XN_STATUS_OK)
    {
        printf("Failed updating data: %s\n",
xnGetStatusString(nRetVal));
        continue;
    }

    // Take current depth map
    const XnDepthPixel* pDepthMap = depth.GetDepthMap();

    // TODO: process depth map
```

```
}

// Clean-up
context.Shutdown();
```

Enumerating Possible Production Chains

The following code demonstrates how to fine control the enumeration process. It enumerates Production Chains for producing User output, reduces the options using a basic query, and then chooses the first of all the possibilities.

```
// Build a query object
xn::Query query;
nRetVal = query.SetVendor("MyVendor");
// TODO: check error code

query.AddSupportedCapability(XN_CAPABILITY_SKELETON);
// TODO: check error code

// Enumerate
xn::NodeInfoList possibleChains;
nRetVal = context.EnumerateProductionTrees(XN_NODE_TYPE_USER, &query,
possibleChains, NULL);
// TODO: check error code

// No errors so far. This means list has at least one item. Take the
first one
xn::NodeInfo selected = *possibleChains.Begin();

// Create it
nRetVal = context.CreateProductionTree(selected);
// TODO: check error code

// Take the node
xn::UserGenerator userGen;
nRetVal = selected.GetInstance(userGen);
// TODO: check error code

// Now we can start to use it
```

Understanding why enumeration failed

Sometimes an application enumerates for a specific node, and receives zero results. An obvious reason would be that no module implementing the node type was installed, although there are other possible reasons, including that a module may be installed but have no license, or a required hardware device is currently disconnected.

OpenNI enables the application to acquire a full list of modules that failed to enumerate, including why each one failed, by using the **xn::EnumerationErrors** function.

The following code attempts to create a Hands Generator node, and if enumeration fails, checks all errors:

```
xn::EnumerationErrors errors;

xn::HandsGenerator handsGen;

nRetVal = context.CreateAnyProductionTree(XN_NODE_TYPE_HANDS, NULL,
handsGen, &errors);

if (nRetVal == XN_STATUS_NO_NODE_PRESENT)

{

    // Iterate over enumeration errors, and print each one

    for (xn::EnumerationErrors::Iterator it = errors.Begin(); it !=
errors.End(); ++it)

    {

        XnChar strDesc[512];

        xnProductionNodeDescriptionToString(&it.Description(), strDesc,
512);

        printf("%s failed to enumerate: %s\n",
xnGetStatusString(it.Error()));

    }

    return (nRetVal);

}

else if (nRetVal != XN_STATUS_OK)

{

    printf("Create failed: %s\n", xnGetStatusString(nRetVal));

    return (nRetVal);

}
```

Working with Depth, Color and Audio Maps

The following code creates a depth generator, checks if it can generate VGA maps in 30 FPS, configures it to that mode, and then reads frames from it, printing out the middle pixel value:

```
XnStatus nRetVal = XN_STATUS_OK;


Context context;
```

```
nRetVal = context.Init();
// TODO: check error code

// Create a depth generator
DepthGenerator depth;
nRetVal = depth.Create(context);
// TODO: check error code

// Set it to VGA maps at 30 FPS
XnMapOutputMode mapMode;
mapMode.nXRes = XN_VGA_X_RES;
mapMode.nYRes = XN_VGA_Y_RES;
mapMode.nFPS = 30;
nRetVal = depth.SetMapOutputMode(mapMode);
// TODO: check error code

// Start generating
nRetVal = context.StartGeneratingAll();
// TODO: check error code

// Calculate index of middle pixel
XnUInt32 nMiddleIndex =
      XN_VGA_X_RES * XN_VGA_Y_RES/2 + // start of middle line
      XN_VGA_X_RES/2;                 // middle of this line

while (TRUE)
{
      // Update to next frame
      nRetVal = context.WaitOneUpdateAll(depth);
      // TODO: check error code

      const XnDepthPixel* pDepthMap = depth.GetDepthMap();
      printf("Middle pixel is %u millimeters away\n",
            pDepthMap[nMiddleIndex]);
}

// Clean up
```

```
context.Shutdown();
```

Working with the Skeleton

The following code shows how to identify when a new user is detected, look for a pose for that user, calibrate the user when they are in the pose, and track them.

Specifically, it prints out the location of the user's head, as they are tracked.

```
#define POSE_TO_USE "Psi"
xn::UserGenerator g_UserGenerator;

void XN_CALLBACK_TYPE
User_NewUser(xn::UserGenerator& generator,
             XnUserID nId, void* pCookie)
{
  printf("New User: %d\n", nId);
  g_UserGenerator.GetPoseDetectionCap().StartPoseDetection(POSE_TO_USE,
                                                           nId);
}
void XN_CALLBACK_TYPE
User_LostUser(xn::UserGenerator& generator, XnUserID nId,
              void* pCookie)
{}
void XN_CALLBACK_TYPE
Pose_Detected(xn::PoseDetectionCapability& pose, const XnChar* strPose,
              XnUserID nId, void* pCookie)
{
  printf("Pose %s for user %d\n", strPose, nId);
  g_UserGenerator.GetPoseDetectionCap().StopPoseDetection(nId);
  g_UserGenerator.GetSkeletonCap().RequestCalibration(nId, TRUE);
}
void XN_CALLBACK_TYPE
Calibration_Start(xn::SkeletonCapability& capability, XnUserID nId,
                  void* pCookie)
{
  printf("Starting calibration for user %d\n", nId);
}
void XN_CALLBACK_TYPE
Calibration_End(xn::SkeletonCapability& capability, XnUserID nId,
```

```
                XnBool bSuccess, void* pCookie)
{
  if (bSuccess)
  {
    printf("User calibrated\n");
    g_UserGenerator.GetSkeletonCap().StartTracking(nId);
  }
  else
  {
    printf("Failed to calibrate user %d\n", nId);
   g_UserGenerator.GetPoseDetectionCap().StartPoseDetection(
                                                        POSE_TO_USE,
                                                        nId);
  }
}

void main()
{
  XnStatus nRetVal = XN_STATUS_OK;

  xn::Context context;
  nRetVal = context.Init();
  // TODO: check error code

  // Create the user generator
  nRetVal = g_UserGenerator.Create(context);
  // TODO: check error code

  XnCallbackHandle h1, h2, h3;
  g_UserGenerator.RegisterUserCallbacks(User_NewUser, User_LostUser,
                                        NULL, h1);
  g_UserGenerator.GetPoseDetectionCap().RegisterToPoseCallbacks(
      Pose_Detected, NULL, NULL, h2);
  g_UserGenerator.GetSkeletonCap().RegisterCalibrationCallbacks(
      Calibration_Start, Calibration_End, NULL, h3);

  // Set the profile
```

```
  g_UserGenerator.GetSkeletonCap().SetSkeletonProfile(
      XN_SKEL_PROFILE_ALL);
  // Start generating
  nRetVal = context.StartGeneratingAll();
  // TODO: check error code

  while (TRUE)
  {
    // Update to next frame
    nRetVal = context.WaitAndUpdateAll();
    // TODO: check error code
    // Extract head position of each tracked user
    XnUserID aUsers[15];
    XnUInt16 nUsers = 15;
    g_UserGenerator.GetUsers(aUsers, nUsers);
    for (int i = 0; i < nUsers; ++i)
    {
      if (g_UserGenerator.GetSkeletonCap().IsTracking(aUsers[i]))
      {
        XnSkeletonJointPosition Head;
        g_UserGenerator.GetSkeletonCap().GetSkeletonJointPosition(
           aUsers[i], XN_SKEL_HEAD, Head);
        printf("%d: (%f,%f,%f) [%f]\n", aUsers[i],
               Head.position.X, Head.position.Y, Head.position.Z,
               Head.fConfidence);
      }
    }
  }

  // Clean up
  context.Shutdown();
}
```

Working with Hand Point

The following code shows how to look for hand gestures, and once a gesture is identified, to start tracking that hand.

```
#define GESTURE_TO_USE "Click"

xn::GestureGenerator g_GestureGenerator;
xn::HandsGenerator g_HandsGenerator;

void XN_CALLBACK_TYPE
Gesture_Recognized(xn::GestureGenerator& generator,
                   const XnChar* strGesture,
                   const XnPoint3D* pIDPosition,
                   const XnPoint3D* pEndPosition, void* pCookie)
{
      printf("Gesture recognized: %s\n", strGesture);
      g_GestureGenerator.RemoveGesture(strGesture);
      g_HandsGenerator.StartTracking(*pEndPosition);
}
void XN_CALLBACK_TYPE
Gesture_Process(xn::GestureGenerator& generator,
                const XnChar* strGesture,
                const XnPoint3D* pPosition,
                XnFloat fProgress,
                void* pCookie)
{}

void XN_CALLBACK_TYPE
Hand_Create(xn::HandsGenerator& generator,
            XnUserID nId, const XnPoint3D* pPosition,
            XnFloat fTime, void* pCookie)
{
  printf("New Hand: %d @ (%f,%f,%f)\n", nId,
         pPosition->X, pPosition->Y, pPosition->Z);
}
void XN_CALLBACK_TYPE
Hand_Update(xn::HandsGenerator& generator,
            XnUserID nId, const XnPoint3D* pPosition,
            XnFloat fTime, void* pCookie)
{
}
```

```
void XN_CALLBACK_TYPE
Hand_Destroy(xn::HandsGenerator& generator,
             XnUserID nId, XnFloat fTime,
             void* pCookie)
{
  printf("Lost Hand: %d\n", nId);
  g_GestureGenerator.AddGesture(GESTURE_TO_USE, NULL);
}

void main()
{

  XnStatus nRetVal = XN_STATUS_OK;

  Context context;
  nRetVal = context.Init();
  // TODO: check error code

  // Create the gesture and hands generators
  nRetVal = g_GestureGenerator.Create(context);
  nRetVal = g_HandsGenerator.Create(context);
  // TODO: check error code

  // Register to callbacks
  XnCallbackHandle h1, h2;
  g_GestureGenerator.RegisterGestureCallbacks(Gesture_Recognized,
                                              Gesture_Process,
                                              NULL, h1);
  g_HandsGenerator.RegisterHandCallbacks(Hand_Create, Hand_Update,
                                         Hand_Destroy, NULL, h2);

  // Start generating
  nRetVal = context.StartGeneratingAll();
  // TODO: check error code

  nRetVal = g_GestureGenerator.AddGesture(GESTURE_TO_USE);
```

```
  while (TRUE)
  {
    // Update to next frame
    nRetVal = context.WaitAndUpdateAll();
    // TODO: check error code
  }

  // Clean up
  context.Shutdown();
}
```

Working with Audio Generators

As detailed earlier, Audio Generators accumulate data until a call to UpdateData() is made, and the entire accumulated audio buffer is returned to the application. The size of the audio buffer may differ from one call to another, and the application should always call the **xn::AudioGenerator::GetDataSize()** function to get the current size of the buffer.

The following code creates an audio generator, configures it to CD quality, and then constantly reads data from it:

```
Context context;
nRetVal = context.Init();
// TODO: check error code

AudioGenerator audio;
nRetVal = audio.Create(context);
// TODO: check error code

XnWaveOutputMode waveMode;
waveMode.nSampleRate = 44100;
waveMode.nChannels = 2;
waveMode.nBitsPerSample = 16;
nRetVal = audio.SetWaveOutputMode(waveMode);
// TODO: check error code

while (TRUE)
{
      // Update to next data
      nRetVal = context.WaitOneUpdateAll(audio);
```

```
        // TODO: check error code

        // Get the audio buffer
        const XnUChar* pAudioBuf = audio.GetAudioBuffer();
        XnUInt32 nBufSize = audio.GetDataSize();

        // Queue the buffer for playing
}

// Clean up
context.Shutdown();
```

Recording and Playing Data

Recording

To record, an application should create a Recorder node, and set its destination (the file name to which it should write). The application should then add to the recorder node, every node it wants to record. When adding a node to the recorder, the recorder reads its configuration and records it. It also registers to every possible event of the node, so that when any configuration change takes place, it is also recorded.

Once all required nodes are added, the application can read data from the nodes and record it. Recording of data can be achieved either by explicitly calling the **xn::Recorder::Record()** function, or by using one of the **UpdateAll** functions (see Reading Data).

Applications that initialize OpenNI using an XML file can easily record their session without any change to the code. All that is required is that they create an additional node in the XML file for the recorder, add nodes to it, and when the application calls one of the **UpdateAll** functions, recording will occur.

The following code generates a depth generator, and then records it:

```
// Create a depth generator
DepthGenerator depth;
nRetVal = depth.Create(context);
// TODO: check error code

// Start generating
nRetVal = context.StartGeneratingAll();
// TODO: check error code

// Create Recorder
Recorder recorder;
```

```
nRetVal = recorder.Create(context);
// TODO: check error code

// Init it
nRetVal = recorder.SetDestination(XN_RECORD_MEDIUM_FILE,
"c:\\temp\\tempRec.oni");
// TODO: check error code

// Add depth node to recording
nRetVal = recorder.AddNodeToRecording(depth, XN_CODEC_16Z_EMB_TABLES);
// TODO: check error code

while (TRUE)
{
      // Update to next frame (this will also record that frame)
      nRetVal = context.WaitOneUpdateAll(depth);
      // TODO: check error code

      // Do application logic
}
```

Playing

To play a file recording, use the **xn::Context::OpenFileRecording()** function. OpenNI will open the file, create a mock node for each node in the file, and populate it with the recorded configuration.

An application may take the nodes it needs by calling the **xn::Context::FindExistingNode()** function, and use them normally.

Note: Nodes created by the player are locked, and cannot be changed, as the configuration must remain according to the recorded configuration.

Applications that initialize OpenNI using an XML file can easily replace their input. This means that instead of reading from a real-time device, they read from a recording by replacing the nodes in the XML file with a recording element (see Recording).

The following code opens up a recording file, and takes the depth generator that was created for this purpose:

```
Context context;
nRetVal = context.Init();
// TODO: check error code
```

```
// Open recording
nRetVal = context.OpenFileRecording("c:\\temp\\tempRec.oni");
// TODO: check error code

// Take the depth node (we assume recording contains a depth node)
DepthGenerator depth;
nRetVal = context.FindExistingNode(XN_NODE_TYPE_DEPTH, depth);
// TODO: check error code

// Add regular application logic
```

Node Configuration

An application will usually want to fully configure a node prior to beginning to stream data. For this reason, OpenNI defines a flow in which configuration can take place, and once all configurations are set, the **xn::Generator::StartGenerating()** function of the node can be called, and data streaming can begin.

The following code creates a depth generator, configures it to VGA resolution, 30 FPS, and then starts it:

```
// Create a DepthGenerator node
xn::DepthGenerator depth;
nRetVal = depth.Create(context);
// TODO: check error code

XnMapOutputMode outputMode;
outputMode.nXRes = 640;
outputMode.nYRes = 480;
outputMode.nFPS = 30;
nRetVal = depth.SetMapOutputMode(outputMode);
// TODO: check error code

// We're done configuring it. Make it start generating data
nRetVal = context.StartGeneratingAll();
// TODO: check error code

// Main loop
while (bShouldRun)
```

```
{
    // Wait for new data to be available
    nRetVal = context.WaitOneUpdateAll(depth);
    if (nRetVal != XN_STATUS_OK)
    {
        printf("Failed updating data: %s\n",
xnGetStatusString(nRetVal));
        continue;
    }

    // Take current depth map
    const XnDepthPixel* pDepthMap = depth.GetDepthMap();

    // TODO: process depth map
}
```

Configuration Using XML file

OpenNI supports using XML as a configuration script. The configuration XML script can be used for creating nodes and configuring them, as well as for configuring the context itself (adding license keys, etc.). an XML script can be executed by calling **xn::Context::RunXmlScript()** and passing it the XML script as a string, or by calling **xn::Context::RunXmlScriptFromFile()** and passing it an XML file to load.

The XML must have one single root node named OpenNI. Under this node there can be three optional sections: Licenses, Log and Production Nodes.

Licenses

This section can provide additional license keys to be registered. The element name should be "Licenses", and it should contain a list of elements, each named "License" with two string attributes: "vendor" and "key". Each such element actually calls **xn::Context::AddLicense()**. For example:

```
<Licenses>

    <License vendor="vendor1" key="key1"/>

    <License vendor="vendor2" key="key2"/>

</Licenses>
```

Log

This section can configure the OpenNI log system. The element name should be "Log". It can contain the following optional attributes:

- writeToConsole: "True" or "false" (default). Determines if the log should be written to the application console.
- writeToFile: "true" or "false" (default). Determines if the log should be written to a file. This file is located under a Log folder that is created under working directory.
- writeLineInfo: "true" (default) or "false". Determines if every log entry should also contain the file name and line info from which it was written.

Additionally, it can also contain the following elements:

- LogLevel with the attribute values set to 0 (verbose), 1 (info), 2 (warnings) or 3 (errors, default). This determines the minimum severity of the log to be written.
- Masks with a list of mask elements, each of which determines if a specific mask is on or off.
- Dumps with a list of dump elements, each of which determines if a specific dump is on or off.

For example:

```
<Log writeToConsole="false" writeToFile="false" writeLineInfo="true">
    <LogLevel value="3"/>
    <Masks>
        <Mask name="ALL" on="true" />
    </Masks>
    <Dumps>
        <Dump name="SensorTimestamps" on="false" />
    </Dumps>
</Log>
```

Production Nodes

This section allows the creation and configuration of nodes. The element name should be "ProductionNodes", and it can have several child-elements performing various tasks:

Global Mirror

The "ProductionNodes" element can contain an element called "GlobalMirror" which sets the global mirror (**xn::Context::SetGlobalMirror()**), according to the "on" attribute ("true" or "false").

For example:

```
<ProductionNodes>
    <GlobalMirror on="true" />
</ProductionNodes>
```

Recordings

The "ProductionNodes" element may contain an element called "Recording" that instructs it to open a recording. For now, OpenNI supports file recordings using the "file" attribute:

```
<Recording file="c:\myfile.oni" />
```

Nodes

The "ProductionNodes" element can contain one or more elements named "Node". Each such element asks OpenNI to enumerate and create a node (similar to the **xn::Context::CreateAnyProductionTree()** function). The "Node" element should have a string attribute named "type" which will indicate the type of the node to be enumerated.

The type can be one of the following:

- Device (XN_NODE_TYPE_DEVICE)
- Depth (XN_NODE_TYPE_DEPTH)
- Image (XN_NODE_TYPE_IMAGE)
- IR (XN_NODE_TYPE_IR)
- Audio (XN_NODE_TYPE_AUDIO)
- Gesture (XN_NODE_TYPE_GESTURE)
- User (XN_NODE_TYPE_USER)
- Scene (XN_NODE_TYPE_SCENE)
- Hands (XN_NODE_TYPE_HANDS)
- Recorder (XN_NODE_TYPE_RECORDER)

Additionally, the "Node" element can have an optional "name" string attribute, which will hold the requested name of the created node.

Queries

The "Node" element can also declare a query that will be used when enumerating for this node. It is done by adding a "Query" element to the "Node" element, which can have the following child-elements:

- "Vendor": Specifies the requested node vendor
- "Name": Specifies the requested node name
- "MinVersion": Specifies the requested node minimum version
- "MaxVersion": Specifies the requested node maximum version.
- "Capabilities": Specifies a list of capabilities that the node must support, each under a "Capability" sub-element.
- "MapOutputModes": Specifies a list of map output modes that should be supported by the map generator, each under a "MapOutputMode" object, that contains three attributes: "xRes", "yRes" and "fps".
- "MinUserPositions": Specifies the minimum number of user positions supported by a depth generator with the "UserPosition" capability.
- **"NeededNodes"**: Specifies that only production trees containing specific nodes are valid. Those nodes are declared using a sub-element named "Node".

If more than one such element is present, all conditions are checked using the "AND" operator.

For example, the following code will try to create a depth node, supplied by **vendor1**, named **name1**, from version 1.0.0.0 to 3.1.0.5, supporting the "UserPosition" and "Mirror" capabilities, a 30 FPS output mode VGA, at least 2 user positions, including user position that uses the "MyDevice" node.

```
<Node type="Depth" name="MyDepth">
    <Query>
        <Vendor>vendor1</Vendor>
        <Name>name1</Name>
        <MinVersion>1.0.0.0</MinVersion>
        <MaxVersion>3.1.0.5</MaxVersion>
        <Capabilities>
            <Capability>UserPosition</Capability>
            <Capability>Mirror</Capability>
        </Capabilities>
        <MapOutputModes>
            <MapOutputMode xRes="640" yRes="480" FPS="30"/>
        </MapOutputModes>
        <MinUserPositions>2</MinUserPositions>
        <NeededNodes>
            <Node>MyDevice</Node>
        </NeededNodes>
    </Query>
</Node>
```

Configuration

Each "Node" element can also contain a list of configuration changes to be performed. This list should be placed under a "Configuration" element. The sub-elements of the "Configuration" element will be executed serially. Those commands can be:

- **"Mirror"**, with an attribute "on" set to "true" or "false". Executes the xn::MirrorCapability::SetMirror() function. Only relevant for generators supporting the "Mirror" capability.
- **"MapOutputMode"**, with 3 attributes: "xRes", "yRes" and "fps". Executes the xn::MapGenerator::SetMapOutputMode() function. Only relevant for map generators (depth, image, IR and scene).
- **"WaveOutputMode"**, with 3 attributes: "sampleRate", "bitsPerSample" and "channels". Executes the xn::AudioGenerator::SetWaveOutputMode() function. Only relevant for audio generators.

- **"Cropping"**, with 5 attributes: "enabled", "xOffset", "yOffset", "xSize", "ySize". Executes the xn::CroppingCapability::SetCropping() function. Only relevant to map generators (depth, image, IR and scene), which support the "Cropping" capability.
- **"PixelFormat"**. Can have the one of the following values: "RGB24", "YUV422", "Grayscale8" or "Grayscale16". Executes the xn::ImageGenerator::SetPixelFormat() function. Only relevant for image generators.
- **"UserPosition"**, which has the attribute "index" and two sub-elements: "Min" and "Max", each has 3 attributes: "x", "y" and "z". Executes the xn::UserPositionCapability::SetUserPosition() function. Only relevant for depth generators supporting the "UserPosition" capability.
- **"FrameSync"**, which contains the name of the node to frame sync with. Executes the xn::FrameSyncCapability::FrameSyncWith() function. Only relevant for generators that support the "FrameSync" capability.
- **"AlternativeViewPoint"**, which contains the name of the node to set viewpoint to. Executes the xn::AlternativeViewPointCapability::SetViewPoint() function. Only relevant for generators that support the "AlternativeViewPoint" capability.
- **"RecorderDestination"**, which contains two attributes: "medium" (currently, only "File" is supported), and "name", which should hold the file name. Executes the xn::Recorder::SetDestination() function. Only relevant for recorder nodes.
- **"AddNodeToRecording"**, which contains two attributes: "name" and "codec". Executes the xn::Recorder::AddNodeToRecording() function. Only relevant for recorder nodes.
- **"Property"**, which contains 3 attributes: "type", "name" and "value". Type can be "int", "real", or "string", which executes the xn::ProductionNode::SetIntProperty(), xn::ProductionNode::SetRealProperty() or xn::ProductionNode::SetStringProperty() functions.

In addition, the application can request that this node be locked (preventing any configuration change to this node once configuration is done) by using the "lock" attribute, and setting it to "true" or "false" (default). This calls the **xn::ProductionNode::LockForChanges()** function.

The following example creates three nodes: image, depth and audio.

- The image node is configured to use QVGA output of 60 FPS, with an RGB24 pixel format. It also sets a cropping area, and turns on the mirror.
- The Depth node is configured to use VGA output of 30 FPS. It also sets the position of the user to a binding box located between the following sets of coordinates: [128, 128, 500] and [600, 400, 2000]. The depth node also configures a special property, proprietary to the "VendorX" vendor.
- The audio is configured to be sampled at 44100 Hz, in stereo mode and at 16-bit per sample. Enumeration takes place only for nodes that support those configurations.

```
<ProductionNodes>
    <Node type="Image">
        <Query>
            <MapOutputModes>
                <MapOutputMode xRes="320" yRes="240" FPS="60"/>
            </MapOutputModes>
```

```
            <Capabilities>
                <Capability>Cropping</Capability>
                <Capability>Mirror</Capability>
            </Capabilities>
        </Query>
        <Configuration>
            <MapOutputMode xRes="320" yRes="240" FPS="60"/>
            <PixelFormat>RGB24</PixelFormat>
            <Cropping enabled="true" xOffset="28" yOffset="28"
xSize="200" ySize="160" />
            <Mirror on="true" />
        </Configuration>
    </Node>
    <Node type="Depth">
        <Query>
            <Vendor>VendorX</Vendor>
            <MapOutputModes>
                <MapOutputMode xRes="640" yRes="480" FPS="30"/>
            </MapOutputModes>
            <Capabilities>
                <Capability>UserPosition</Capability>
            </Capabilities>
        </Query>
        <Configuration>
            <MapOutputMode xRes="640" yRes="480" FPS="30"/>
            <UserPosition index="0">
                <Min x="128" y="128" z="500"/>
                <Max x="600" y="400" z="2000"/>
            </UserPosition>
            <Property type="int" name="VendorXDummyProp" value="3" />
        </Configuration>
    </Node>
    <Node type="Audio">
        <Configuration>
            <WaveOutputMode sampleRate="44100" bitsPerSample="16"
channels="2" />
        </Configuration>
    </Node>
```

```
</ProductionNodes>
```

Start Generating

By default, when all nodes under the "ProductionNodes" element are created and configured, a call is made to the **xn::Context::StartGeneratingAll()** function. If the application requires a different behavior, it can place the "startGenerating" attribute containing "true" or "false", on any node, and also on the "ProductionNodes" element (which defines whether or not to start to generate all). For example, the following will create two nodes: image and depth, but only start to generate the depth node:

```
<ProductionNodes startGenerating="false">
    <Node type="Image" />
    <Node type="Depth" startGenerating="true" />
</ProductionNodes>
```

Building and Running a Sample Application

OpenNI is provided with certain samples, which are located in the 'Samples' folder, with their binaries under 'Samples\Bin\Debug' or 'Samples\Bin\Release'. Most samples use an XML file to configure OpenNI. This XML file can be found at '%OPEN_NI_INSTALL_DIR%\Data\SamplesConfig.xml'.

Note: on Linux, some samples (like NiViewer, NiSimpleViewer, NiUserTracker) need the GLUT library in order to compile and run. Install freeglut3-dev or equivalent. Other samples need the mono WinForms library. Install libmono-winforms2.0-cil (or mono-complete).

To build and run a sample application:

1. Ensure that you have the latest Microsoft Platform SDK installed. This can be downloaded this using the following hyperlink: Microsoft's Platform SDK Web Install.
2. Open Windows Explorer (or your preferred file navigator), and browse to the OpenNI installation directory, the default location of which is: **C:\Program Files\OpenNI**.
3. In the OpenNI directory, browse to: **Samples\NiSimpleViewer**.
4. Open the **NiSimpleViewer_2008.vcproj** project file, and build the application.
5. After you have successfully built the project, but before you try to run it, please ensure that the **SamplesConfig.xml** file is correctly configured, according to the following specifications:
    a. Navigate to the **Data** directory, the default location of which is **C:\Program Files\OpenNI\Data**.
    b. Use a text editor program to open the **SamplesConfig.xml** to be edited.

Throughout the sample applications tutorial you will encounter use of relative paths in the sample application source files.

Note: When your application is executed from within a debugging environment, such as Microsoft Visual Studio, paths which are not absolute may not be resolved relative to the output executable.

In order for the sample application to execute correctly in debug mode, you should modify the Working Directory to be the sub-folder in which the executable is located. You can also use Visual Studio's macro instead of setting an actual path.

Note: To set the value, use **"Project Properties"->"Debugging"->"Working Directory"**.

NiSimpleRead

NiSimpleRead is a basic sample that configures OpenNI using the SamplesConfig XML file, then uses the depth generator node. The application loops to read new frames from the depth generator, and prints out the depth value of the middle pixel. The sample is created when the user presses 'ESC'.

NiSimpleCreate

NiSimpleCreate demonstrates how to create a production node programmatically in code, rather than using the SamplesConfig XML file. After creating a depth node, it reads from the node in the same way as NiSimpleRead.

NiCRead

NiCRead is a sample that is exactly the same as NiSimpleRead, other than the fact that it demonstrates the use of the C interface, rather than the C++ interface.

NiSimpleViewer

NiSimpleViewer is a small OpenGL application that draws the depth maps and the image maps to the screen. It configures OpenNI using the SamplesConfig XML, but requires both depth and color images to be present, both with the same resolution, and with the image node set to RGB24 format. The application creates a histogram of the depth map and draws the frame using this, to enable better visibility of the depth map.

The following keys can be used to control the application:

Key	Description
1	Converts to OVERLAY mode, drawing a depth map on top of the image map. It also sets depth viewpoint to image viewpoint (using the Alternative viewpoint capability).
2	Draws only depth. It also turns off alternative viewpoint.
3	Draws only image. It also turns off alternative viewpoint.
Esc	Closes the application

NiSampleModule

NiSampleModule is a sample for writing a module that is OpenNI compliant. It implements a depth node supporting the mirror capability. Before using this, this module must be registered using the niReg utility. It should also be deregistered afterwards, otherwise applications may receive this when they require depth nodes.

NiConvertXToONI

NiConvertXToONI opens any recording, takes every node within it, and records it to a new ONI recording. It receives both the input file and the output file from the command line.

NiRecordSynthetic

NiRecordSynthetic demonstrates how to open a recording, perform a form of transformation on the data within it, and re-record this data.

NiViewer

NiViewer shows how to display depth, image and IR maps, and play audio, in addition to demonstrating a wide set of configurations. NiViewer has two modes: If a file name appears in the command-line, it will open this file as a recording. Otherwise, it will configure OpenNI using the SamplesConfig XML file.

NiViewer is automatically associated with the .ONI file extension, for opening OpenNI recordings. The following keys can be used to control the application:

Key	Description
1	Shows depth only, in histogram mode
2	Shows depth only, in psychedelic mode (centimeters)
3	Shows depth only, in psychedelic mode (millimeters)
4	Shows depth only, in rainbow mode
5	Shows depth masked image, meaning image pixels that don't have depth values are blacked out.
6	Background removal mode
7	Shows depth and image (or IR), side by side.
8	Shows depth on top of image (or IR)
9	Shows transparent depth on top of image (or IR)
0	Shows rainbow depth on top of image (or IR)
=	Shows image (or IR) only
`	Shows depth standard deviation
p	Toggles pointer mode on/off. When pointer mode is on, additional depth info is displayed regarding currently pointed pixel.
f	Toggles Full Screen / Window mode
?	Toggles help screen on/off
m	Toggles mirror on/off
/	Resets all cropping
s	Start recording
d	Start recording in 5 seconds

x	Stop recording
c	Capture current frame to files
z	Start/Stop collecting statistics about depth pixels
o	Pause/Play
l	Seek one frame forward (recordings only)
L	Seek 10 frames forward (recordings only)
k	Seek one frame backwards (recordings only)
K	Seek 10 frames backwards (recordings only)
;	Read one single frame and pause
Esc	Closes the application

Additionally, the mouse can be used. Clicking the right-button of the mouse opens up a menu through which many configurations can be changed. Using the left mouse button can block selection, by holding it down over one part of a frame, then moving it and releasing the key, causes the node to be cropped, if the Cropping capability is supported.

NiBackRecorder

niBackRecorder is a command line tool, which stores frames in memory in a cyclic buffer. Clicking "D" sends a request to dump this cyclic buffer to an ONI file. In effect, it saves the last certain number of seconds, according to how it has been configured.

Usage

```
niBackRecorder time <seconds> [depth [qvga|vga]] [image [qvga|vga]]
[verbose] [mirror <on|off>] [registration] [framesync] [outdir
<directory>]
```

The following option is mandatory:

- **Time**: Number of seconds to dump each time

The following options can be used:

- **Depth**: Sets the resolution of the depth to either QVGA or VGA. If not specified, depth is off. If no resolution is specified, QVGA is used.
- **Image**: Sets the resolution of the image, to either QVGA or VGA. If not specified, image is off. If no resolution is specified, QVGA is used.
- **Verbose**: Turns on the log
- **Mirror**: Sets the mirror mode. If not specified otherwise, it uses whatever was configured.
- **Registration**: Changes the depth to match the image.
- **Framesync**: Synchronizes between depth and image
- **Outdir**: The location where the oni files should be created. The default is the execution directory.

Note: Keep in mind the amount of memory used to store the frames.

Configuration	**Size**

1 second, QVGA depth	30*320*240*2B = 4500KB
1 second, QVGA image	30*320*240*3B = 6750KB
1 second, VGA depth	30*640*480*2B = 18000KB
1 second, VGA image	30*640*480*3B = 27000KB

NiUserTracker

NiUserTracker shows how to use the User Generator, with its pose detection and skeleton capabilities.

Each figure identified in the scene is colored in a different color, and the User Generator searches for the calibration pose. Once the figure is in the calibration pose, calibration is performed on that figure. When the calibration is successfully completed, a skeletal representation of the figure exists.

Key	Description
b	Toggle background pixels
x	Toggle all pixels
s	Toggle skeleton (for calibrated users)
i	Toggle user label
l	Toggle skeleton (for calibrated users)
p	Pause/Start
Esc	Closes the application

Troubleshooting

OpenNI provides a simple log mechanism. Each log entry consists of a severity (error, warning, info or verbose), a mask (arbitrary string, usually the name of the component logging this entry) and a message. Any component (or application) that uses OpenNI can write log entries.

Log entries can be output to a file, the console, or both.

The log system can be configured using API calls or Xml scripts (see Configuration Using XML file). Available configuration options set the destination of the log, and filter it according to severity or to specific masks.

Note: The special mask "ALL" controls whether **all** masks are written.

When failing to start any OpenNI based application, the most useful thing to do is to turn on logs. This is also a reason why it is better for an application to be configured using an XML script. The script will have turned the log off by default, and when needed, a user can turn it back on. The log may contain the reason why the application fails to load, but even if it does not contain this reason, sending the log to the developer of a failing component could mean a lot in terms of pinpointing the problem.

Glossary

Term	Description
Image Map	An array of pixels that represent an image.
IR Image Map	An image map where each pixel represents the brightness of that pixel in grayscale.
Depth Map	An image map where each pixel is represented by its distance from the sensor.
Color Image Map	An image map where each pixel is represented by an RGB value.
Hand Point	The location of the palm of a hand in the scene.
Gesture	Expressing an instruction through bodily movements.
Calibration	The act of capturing and analyzing the various proportions and measurements of the figure's body, to optimize the specific tracking of its movements.
Calibration Pose	A pose that the figure is requested to hold for several seconds, to enable the software to calculate the calibration data of this specific user (for example, a Psi pose).