Kinect Image Processing

ECE 479/579 Project

3/25/2012

Tonderai Nemarundwe & Jabeer Ahmed

Introduction

Goals

- The goal of the project was to be able to track human users via a Kinect sensor
- To create an Avatar that mimics user movements recorded from Kinect
- To recognize gestures**

Background Information

- The method we used is based on Natural Interaction (NI) which refers to a concept of Human-device interaction.
- Body motion is tracked in our project
- OpenNI (Open Natural Interaction) is the cross-platform framework that we get our APIs from.

How things work (Simple overview)

- Kinect is a 3D sensor node that creates 3D data/depth map where each point is distance from Kinect
- OpenNI middle-ware processes and creates production nodes from raw Kinect data
- User motion/gestures are tracked by generators
- Application layer can render images/execute based on this data

Theoretical Information

RGB Camera: The RGB camera embedded in the device is a regular USB video camera (640x480). It can be used for simple imaging to object detection and tracking. Using OpenCV haar classifiers and objection detection algorithms it is possible to detect faces, facial features such as eyes, nose, mouth etc, and hands.

Figure 1: Kinect devices: RGB camera, infrared laser projector, monochrome CMOS camera, multi-array MIC and motorized tilt

3D depth Sensor (Infrared laser projector and monochrome CMOS camera pair): Common misconception is that the depth sensor on the kinect device is a laser range finder. It is actually an *Active Triangulation system* consisting of a camera-projector pair and uses S*tructured Light Illumination.* An active triangulation system is where a projector (*infrared projector*)projects a structured pattern of light on to the targeted scene, which is then imaged from a different point of view with a camera (*monochrome CMOS camera*) and triangulated for depth information, figure 2 is an illustration of this technique. Benefit of this feature is that no point by point scanning needs to be performed, instead the entire scene can be captured at once and reconstructed. Also, since Kinect has its own invisible light source, it can perform depth imaging even in dimly lit enviornment.

Multi-Array Microphone: The Kinect is also equipped with an array of microphones and the OpenNI library comes with algorithms for performing audio processing and speech recognition.

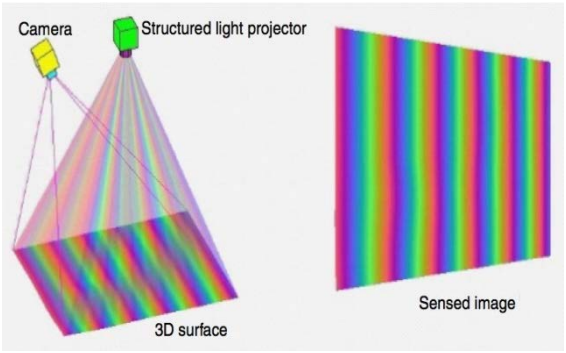


Figure 2 Structured light illumination technique. A projector illuminates the scene with structured light, which is then captured by the camera from a different perspective. Then the captured image is scanned for matching points between projected image and captured image. These points are then triangulated to recover depth image.

Body Parts Detection and Skeleton Tracking

Objects such as human head, faces or hands are easier to detect due to uniformity in their key features, which is why it is sometimes possible to detect and track these features using regular colored image. But, other body parts such as elbow, shoulders, torso, legs etc. are harder to detect due to their increased variance in size, shape, texture (e.g. clothing) and silhouette ambiguity due to similarity with the background. This can be easily overcome using depth data for feature detection.

Like most object detection and tracking algorithms, OpenNI's skeleton tracker algorithm applies a combination of probabilistic guessing and machine learning algorithm. The steps involved in their algorithm are as follows:

1. A large data base of body poses are captured performing all realistic poses. The set of subjects used for database building represent a wide range of body sizes, shape, age, clothing and other features such as hair style and facial hair.
2. Poses are then clustered into groups where each group if most distinct from each other to generalize a pose.
3. A deep *Randomized decision forests classifier* is trained using the pose data set consisting of thousands of pose depth image set. Jamie Shotton et al., demonstrated that using a really large number of image set it is possible to eliminate overestimation.
4. This trained classifier is later used for detection, pose estimation and tracking.

Figure 3, shows the relation between accuracy in pose estimation versus the number of training images, as reported by *Jamie et al*. In the OpenNI implementation, approx. 500K images were used for training, which makes it a very robust system.

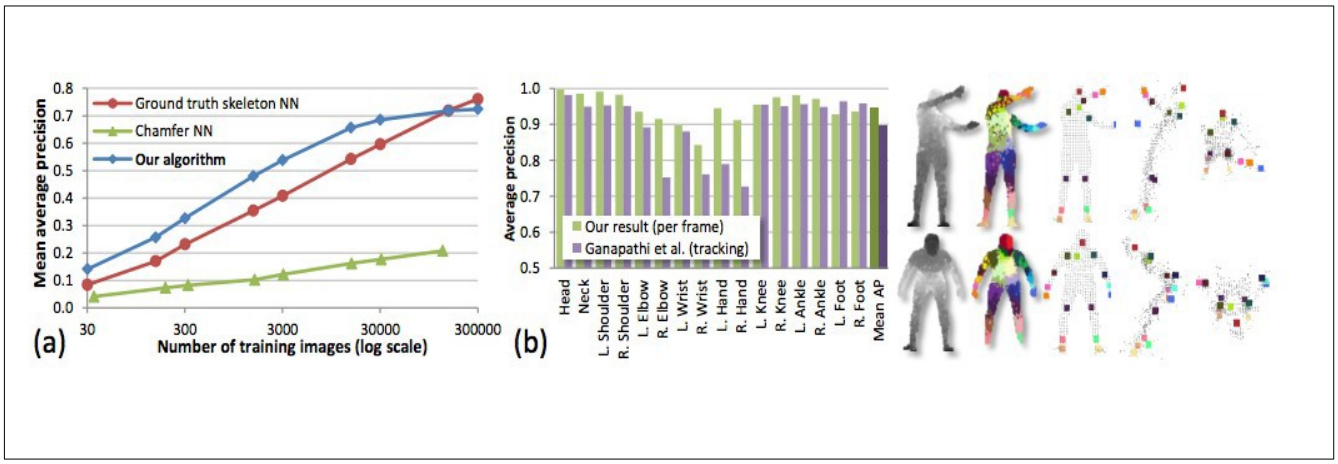


Figure 3 a) Mean precision in joint location estimation vs. number of images used to generate classifier. b) Average precision for body parts

Initialization

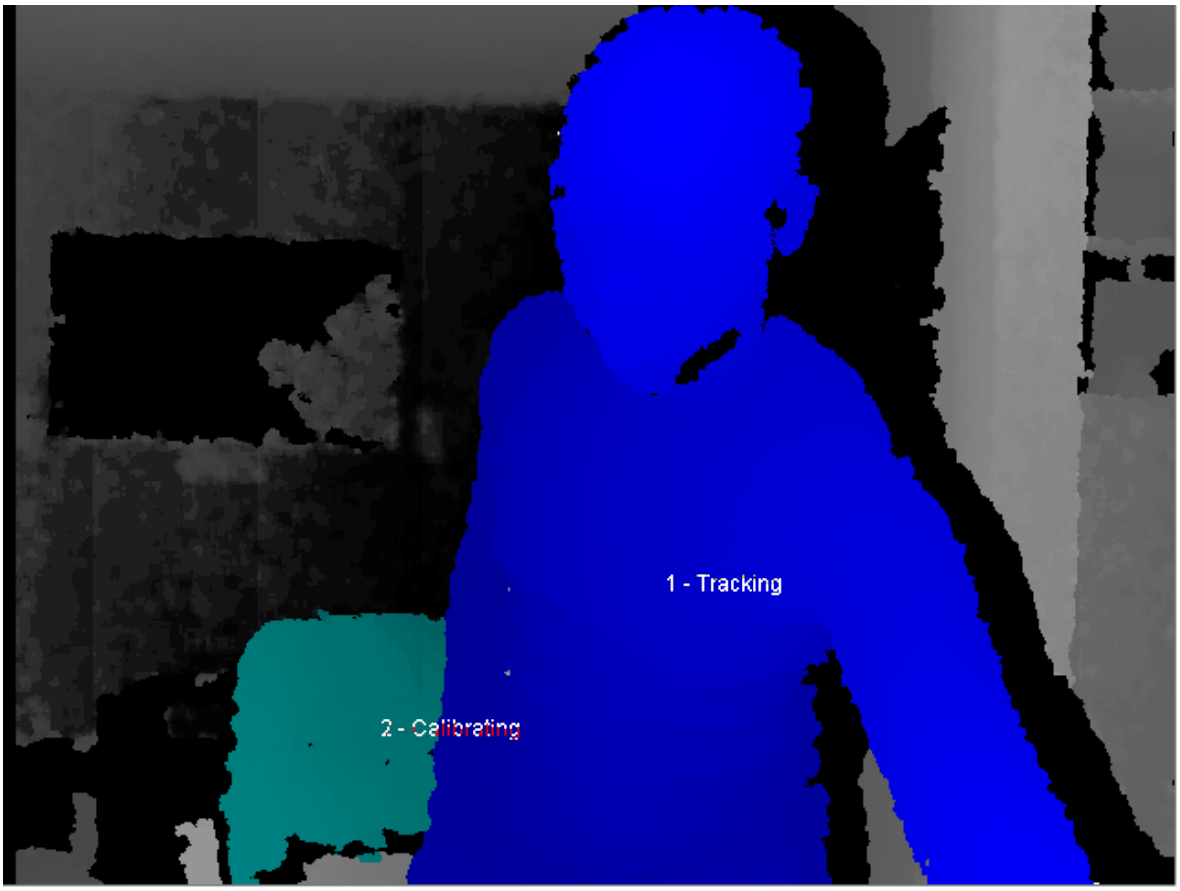


Figure 4 Calibration and first output

- Application calibrates and creates a hash map for every user.
- The hash map is used to keep track of 3D points of user and joints
- The 3D points are used to create a converted real world projective of user

Skeleton Tracking

- The skeleton tracking is based on the joint coordinates created by the NUI middle ware
- Based on confidence level of the neural net algorithm implemented by NITE, the joint will be created if it is above a threshold
- To draw the skeleton, it is a matter of drawing lines that join the joints created by NUI middle ware

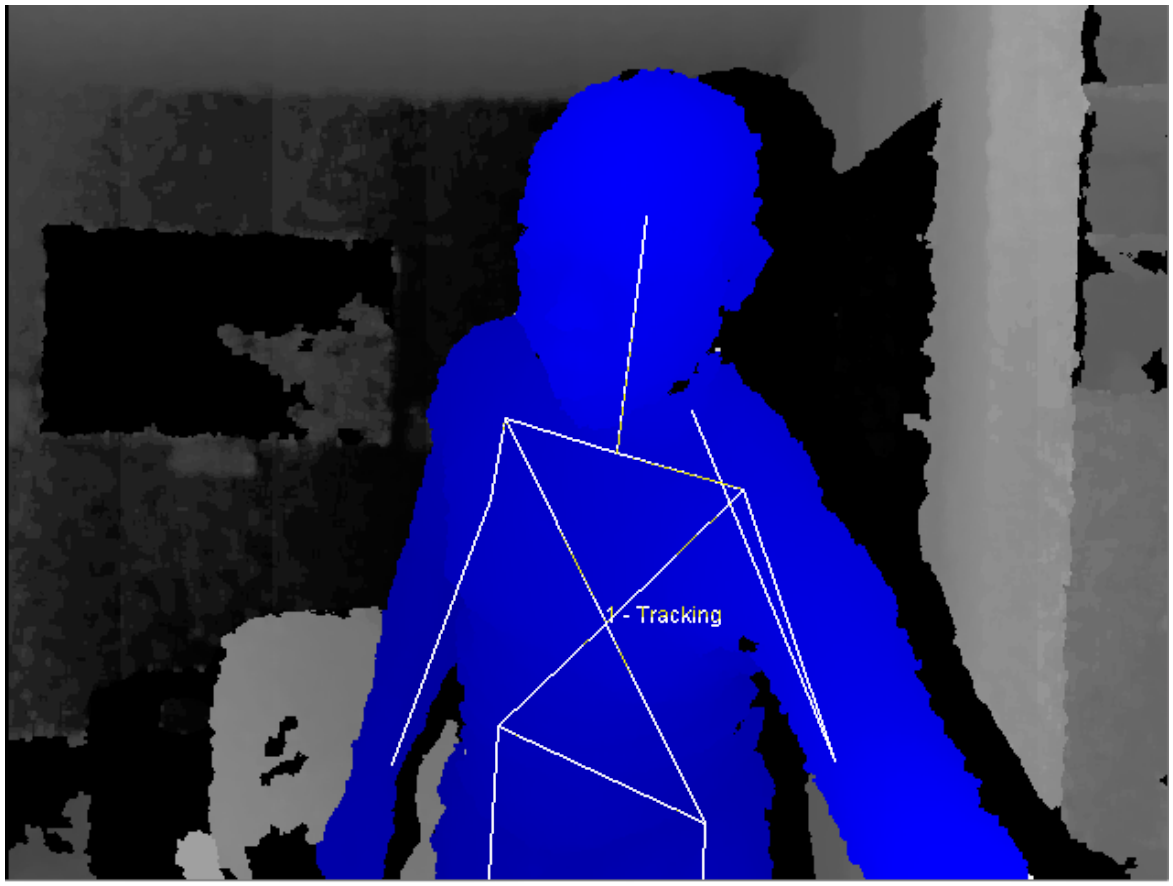


Figure 5 Skeleton Tracking

Hand tracking

There are a couple of ways to do hand tracking:

- Using hand joint information or using NITE hand tracker
- Using joints doesn't seem to be as accurate as NITE hand generator
- Using OpenCV and Kinect it is also possible to recognize fingers and track finger tips. These methods are based on depth *thresholding*, *contour extraction*, *approximating contours* and using *convex hull* mathematics to get the vertices of the fingertips if their interior angle is small enough

More information on convex hull: http://mathworld.wolfram.com/ConvexHull.html

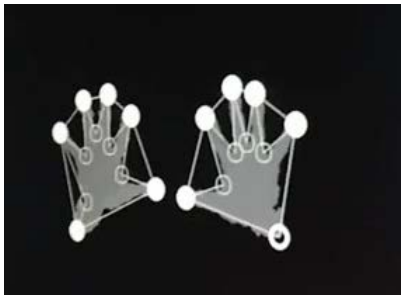


Figure 6 Fingertip detection

Figure 7 Hand tracking using Joint information

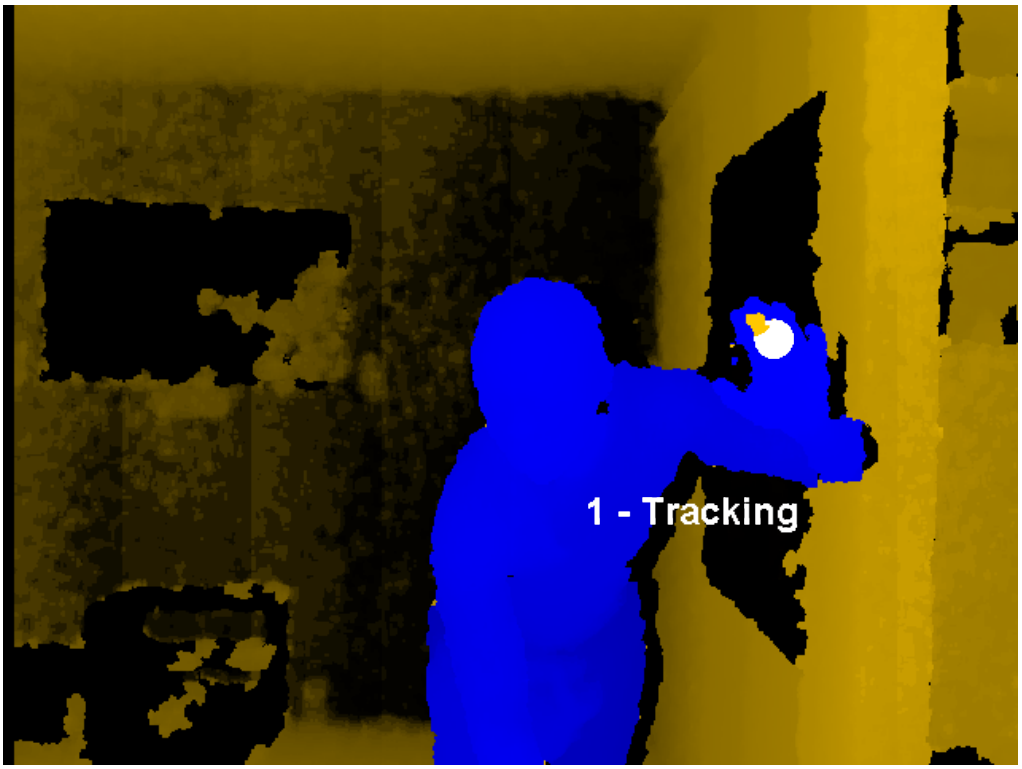


Figure 8 Hand tracking using hand generator production node

Conclusion

Working with the Kinect was a great experience. It is a very powerful sensor that is a great addition to the Robotics community. It is a great innovation by Microsoft that is beginning to change the way people can interact with technology. Based on physical movements and speech, a new technological phenomenon is in effect thanks to the Kinect. Our project provides a framework for a future developer to work on and it is easy to follow once you have read the appendix sections of this report.

To be able to control a Robot's motion with the Kinect, the next developer has to use the 3D points created and stored in the Hash maps and translate them into actual target points for a robot. We proved this by creating an avatar and used these 3D points to animate the Avatar.

The microphone array in the Kinect is a great piece of hardware that can also be used in project. With available speech processing APIs from Microsoft, voice control for the robot would be easy to implement using a Kinect. The robot can also locate the person who is giving the command and actually approach them using digital signal processing techniques.

Gesture recognition is already implemented in our project. There are some basic gestures already pre-written by NITE. This functionality is accessible through instantiating the appropriate objects from the NITE APIs. A list of recognizable gestures can be obtained by looking in the PrimeSense documentation. To recognize your own gestures, it would be a simple matter of creating a data set of these gestures (which in theory are combinations of xyz coordinates & time). A neural net or some form of machine learning would then have to be implemented for a smooth application.

Source code for our project is located on Google code: http://code.google.com/p/ece479-skel-tracker/

Using Eclipse and SVN checkout would be the best way to get started.

Appendix 1: Getting Started With Kinect

Prerequisites

Figure 9: Kinect and Adapter

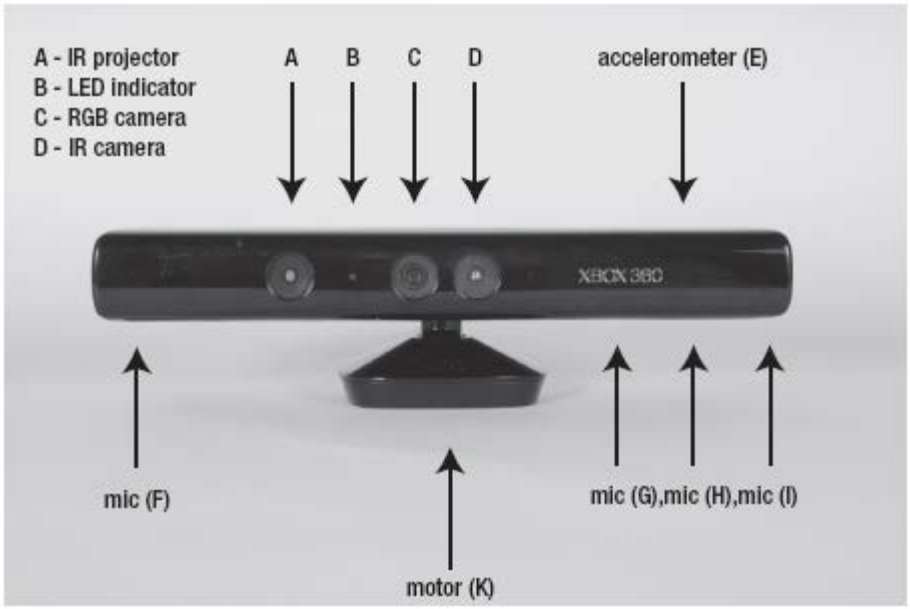


Figure 10:Kinect external component identification

- The above picture shows all the basic parts that make up a Kinect. It has four microphones which make it record quadraphonic sound. With digital signal processing in software, it is possible to do background filtering, noise detection as well as determining relative position of a person speaking within a room.
- The Kinect also has a standard webcam and an infrared camera.

- There is also a laser which works in conjunction with the infrared camera to derive the exact position in space of everything in the field of view.

Installing Drivers and Middleware

Need to install in order,

1. OpenNI
2. Sensor Kinect drivers (Prime Sense)
3. NITE (available from OpenNI website)

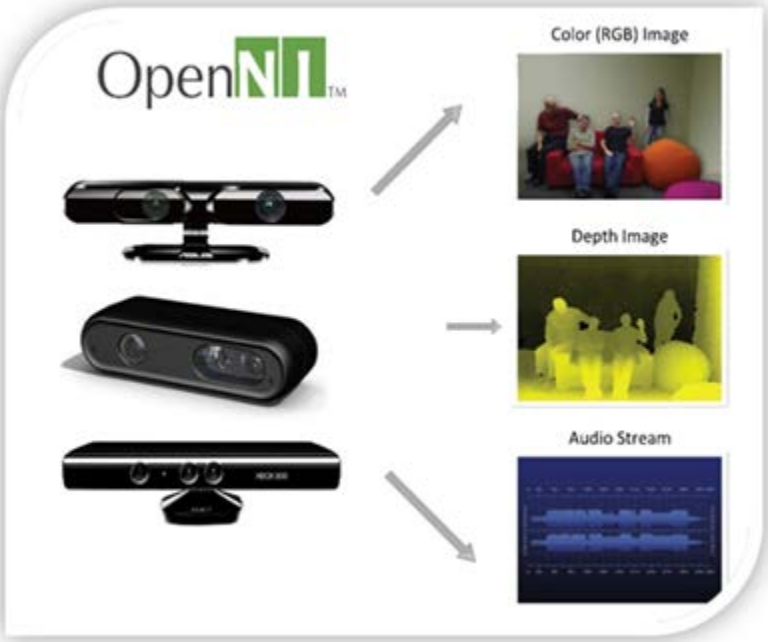
OpenNI stable master tree:
https://github.com/OpenNI/OpenNI
OpenNI latest unstable branch:
https://github.com/OpenNI/OpenNI/tree/unstable

PrimeSense Sensor Module for OpenNI stable master tree:
https://github.com/PrimeSense/Sensor
PrimeSense Sensor Module for OpenNI latest unstable branch:
https://github.com/PrimeSense/Sensor/tree/unstable

Wiki: http://wiki.openni.org

Binaries are available at:
http://www.openni.org/Downloads/OpenNIModules.aspx
(The "OpenNI Compliant Hardware Binaries" section)

How the Kinect with OpenNI works



The Kinect sensor is a streaming device. It basically sends via high speed USB communication the following types of data:

- RGB Data
- Depth Data
- Audio Capture

What you need to know about OpenNI

Using OpenNI means you will have to be accessing Prime Sense's NITE middleware to get meaningful information from the device, including motion tracking support, user recognition and gesture support..

The middleware contains the following components that allow the device to make sense of the scene the sensor is observing. Note this is not like a mouse where you have limited directional information to read or events to fire. You have to create your own “events” for your applications to understand and use what the Kinect is sending you..

NITE MIDDLEWARE *KEY* COMPONENTS

Full body component: 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).

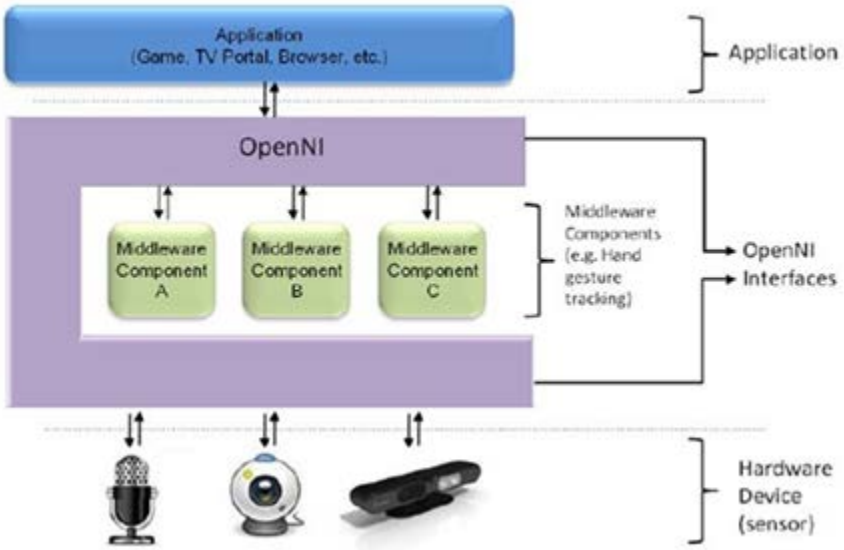
Hand point analysis component: A software component that processes sensory data and generates the location of a hand point

Gesture detection component: A software component that identifies predefined gestures (for example, a waving hand) and alerts the application.

Scene Analyzer component: A software component that analyzes the image of the scene in order to produce such information as:

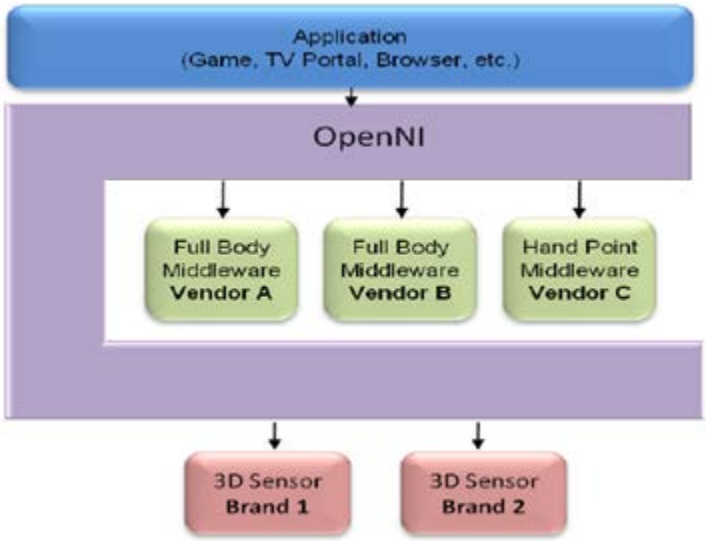
- The separation between the foreground of the scene (meaning, the figures) and the background
- The coordinates of the floor plane
- The individual identification of figures in the scene.

A Top down View of the Sensor and Interaction with the Middleware



Why OpenNI is a very well designed standard

OpenNI supports sensors and natural interaction devices from multiple vendors.



In fact with OpenNI you could have multiple sensors working within your application from different vendors and extended middleware to provide enhanced tracking.

Teaching the Sensor how to see and what to observe

How do we use the NITE Middleware to get back meaning information back about what our sensor/sensors are observing?

Meaningful Data Points

"Meaningful" 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. NITE middleware is used to process this raw output, and produce a higher-level output, which can be understood and used by the application.

Production Nodes

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.

Sensor-Specific Production Node Types

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.

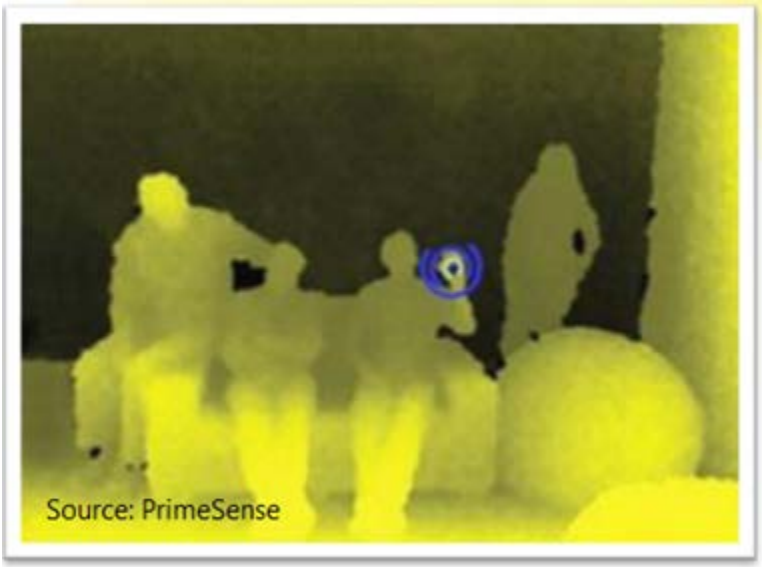
Depth Generator: A node that generates a depth-map. This node should be implemented by any 3D sensor that wishes to be certified as OpenNI compliant.

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-Specific Production Node Types



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. The scene analyzer is probably the busiest of all the middleware components.

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.

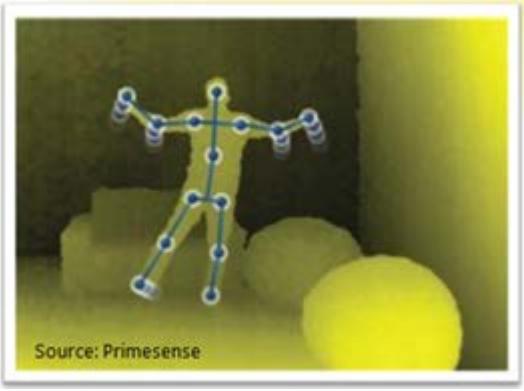
User Generator: Generates a representation of a (full or partial) body in the 3D scene. Remember more than one user can be recognized and logged in at one time. For a user to be recognized and calibrated he normally must appear first in the "PSI" position extending his limbs as the Greek alphabet symbol Ψ. In other software like the X-Box Kinect a hand waiving gesture is used to recognize the user.

Recording the following are supported:.

Recorder: Implements data recordings

Player: Reads data from a recording and plays it

Codec: Used to compress and decompress data in recordings



What is a Production Chain?

In order to produce body data, this production node uses a lower level depth generator, which reads raw data from a sensor.

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.

In order to produce body data, this production node uses a lower level depth generator, which reads raw data from a sensor.

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.

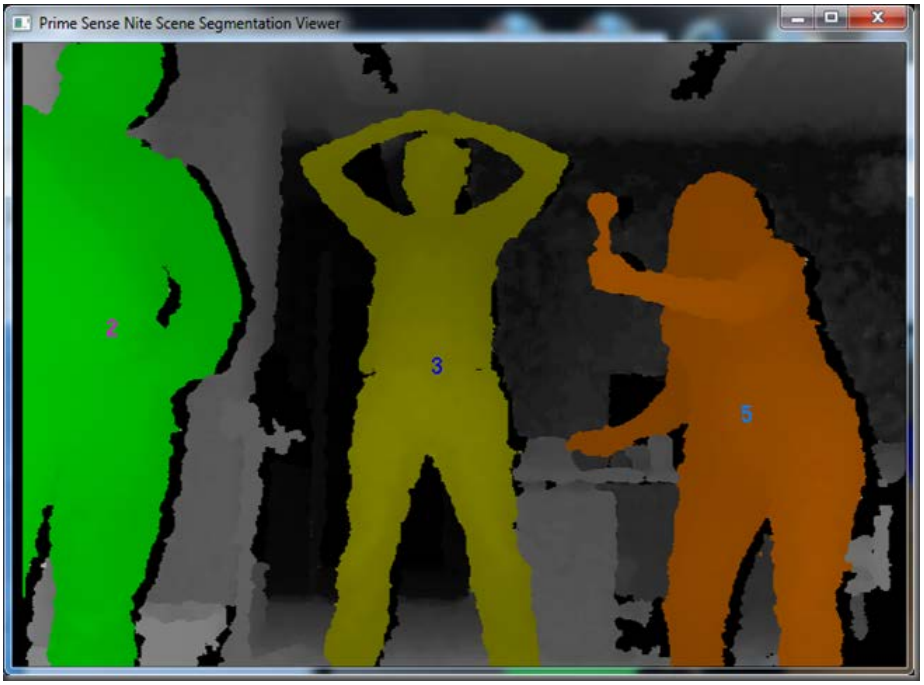
Typically, an application is only interested in the top product node of each chain. This is the node that outputs the required data on a practical level,.

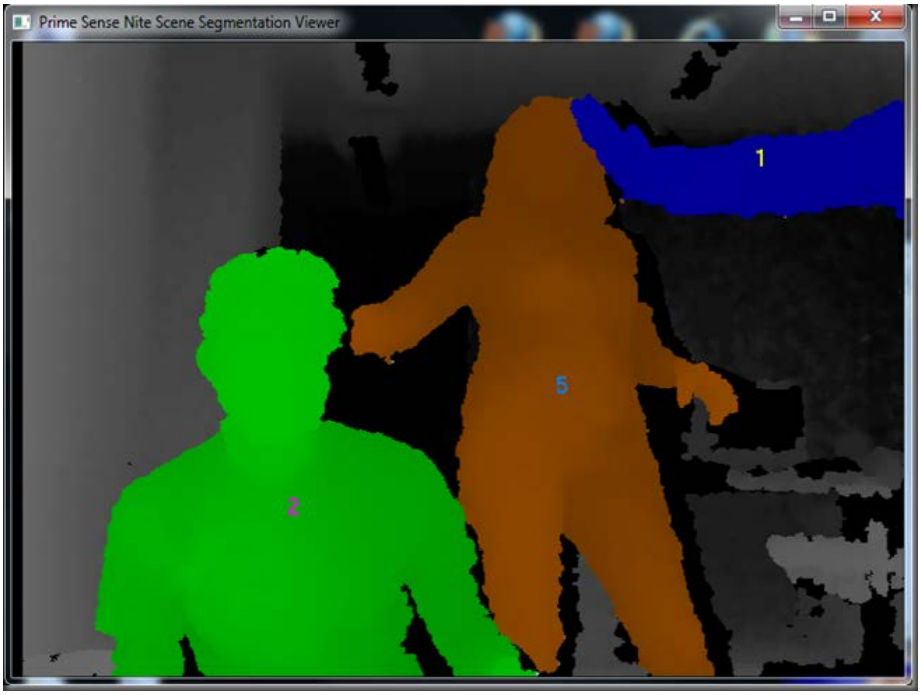
For example, a hand point generator. OpenNI enables the application to use a single node, without being aware of the production chain beneath this node. For advanced tweaking, there is an option to access this chain, and configure each of the nodes.

Appendix 2: Software Development

- The main thing is to add the dlls located in the directories created by the installation of OpenNI and PrimeSense. The location of these two, usually in Program Files in Windows contains a lib and include directory. The necessary dlls, library objects and jars are all located in there.
- Your project has to be able to reference these files and it will work.

Below is sample program included with OpenNI install.





Code walk through

SimpleViewer - sample program (C++)

This section describes the SimpleViewer sample program. This sample program uses a DepthGenerator node and ImageGenerator node to build an accumulative histogram from depth values.

Global Declaration Block

The following definition is for the path to an OpenNI XML script file for inputting and building a stored production graph. The *production graph* is a network of *production nodes* and is the principal OpenNI object model. The identifies blobs as hands or human users.

```
#define SAMPLE_XML_PATH "../../../../Data/SamplesConfig.xml"
```

The following declares the array for the histogram array that is a key part of this sample program. (This is not OpenNI specific.)

```
float g_pDepthHist[MAX_DEPTH];
```

The following declaration block declares the OpenNI objects required for building the OpenNI production graph.

```
Context g_context;
ScriptNode g_scriptNode;
DepthGenerator g_depth;
ImageGenerator g_image;
DepthMetaData g_depthMD;
ImageMetaData g_imageMD;
```

Each of these declarations is described separately in the following paragraphs.
A **Context** object is a workspace in which the application builds an OpenNI production graph.
The **ScriptNode** object loads an XML script from a file or string, and then runs the XML script to build a production graph.
The **DepthGenerator** node generates a depth map. Each map pixel value represents a distance from the sensor.
The **ImageGenerator** node generates color image maps of various formats, such as the RGB24 image format. Call its **SetPixelFormat()** method to set the image format to be generated.
The **DepthMetaData** object provides a **frame object** for the **xn::DepthGenerator** node. A **generator node's** frame object contains a generated data frame and all its associated properties. This frame object, comprising the data frame and its properties, is accessible through the node's metadata object.
The **ImageMetaData** object provides a **frame object** for the **xn::ImageGenerator** node. This metadata object is associated with an ImageGenerator node in the same way as the DepthMetaData object is associated with a DepthGenerator node.

Main code

The declarations at the top of the main program collect and report status and errors from any of the OpenNI functions.

```
XnStatus rc;
EnumerationErrors errors;
```

Use Script to Set up a Context and Production Graph

The **InitFromXmlFile()** method is a shorthand combination of two other initialization methods — **Init()** and then **RunXmlScriptFromFile()** — which initializes the context object and then creates a production graph from an XML file. The XML script file describes all the nodes you want to create. For each node description in the XML file, this method creates a node in the production graph.

```
rc = g_context.InitFromXmlFile(SAMPLE_XML_PATH, g_scriptNode, &errors);
```

Verify Existence of Nodes in the Sample Script File

This is verification code to check that OpenNI found at least one node definition in the script file. The program continues execution only if at least one node definition if found.

```
if (rc == XN_STATUS_NO_NODE_PRESENT)
{
    XnChar strError[1024];
    errors.ToString(strError, 1024);
    printf("%s\n", strError);
    return (rc);
}
else if (rc != XN_STATUS_OK)
{
    printf("Open failed: %s\n", xnGetStatusString(rc));
    return (rc);
}
```

Get a DepthGenerator Node from the Production Graph

Assuming that the above call to **InitFromXmlFile()** succeeded, a production graph is then created.
The **FindExistingNode()** method in the following code block tries to get a reference to any one of the production nodes. This call specifies XN_NODE_TYPE_DEPTH to get a reference to a **xn::DepthGenerator** "DepthGenerator" node. A DepthGenerator node generates a depth map as an array of pixels, where each pixel is a depth value representing a distance from the sensor in millimeters. A reference to the node is returned in the depth parameter.

```
rc = g_context.FindExistingNode(XN_NODE_TYPE_DEPTH, g_depth);
```

The code block that follows the FindExistingNode() call just checks that OpenNI found a DepthGenerator node in the production graph.

```
if (rc != XN_STATUS_OK)
{
    printf("No depth node exists! Check your XML.");
    return 1;
}
```

Get a DepthGenerator Node from the Production Graph

The following code is similar to the previous code block, but this time the FindExistingNode() method call gets a reference to an **ImageGenerator** node. Assuming that an ImageGenerator node was found, a reference to it is returned in the g_image parameter.

```
rc = g_context.FindExistingNode(XN_NODE_TYPE_IMAGE, g_image);
if (rc != XN_STATUS_OK)
{
    printf("No image node exists! Check your XML.");
    return 1;
}
```

Get the DepthGenerator's Data

In the following statement, the latest data generated is placed in an 'easy-to-access' buffer. In OpenNI terminology: the node's getMetaData() method gets the node's data that is designated as 'metadata to be placed in the node's metadata object'. The code copies the node's frame data and configuration to a metadata object (depthMD). This metadata object is then termed the 'frame object'.

```
g_depth.GetMetaData(g_depthMD);
```

Get the ImageGenerator's Latest Data

This works the asme as for the DepthGenerator as above.

```
g_image.GetMetaData(g_imageMD);
```

Checking for Unsupported Mode or Format

The following code block checks for Hybrid mode requirement. Hybrid mode isn't supported in this sample. This check accesses some attributes of the frame data's associated configuration properties: FullXRes() and FullYRes() are the full frame resolution, i.e., the entire field-of-view, ignoring cropping of the FOV in the scene.

```
if (g_imageMD.FullXRes() != g_depthMD.FullXRes() || g_imageMD.FullYRes() != g_depthMD.FullYRes())
{
    printf ("The device depth and image resolution must be equal!\n");
    return 1;
}
```

The following code block checks that the selected pixel format is RGB24. Other formats are not supported.if (g_imageMD.PixelFormat() != XN_PIXEL_FORMAT_RGB24)

```
if (g_imageMD.PixelFormat() != XN_PIXEL_FORMAT_RGB24)
{
    printf("The device image format must be RGB24\n");
    return 1;
}
```

Initializing the Texture Map

The dimensions of the Texture Map buffer are calculated by rounding the full frame resolution of the DepthGenerator data frame. full frame resolution is accessed through **xn::MapMetaData::FullXRes()** "FullXRes()" and **xn::MapMetaData::FullYRes()**"FullYRes()" (again, both accessed through the metadata frame object).

```
g_nTexMapX = (((unsigned short)(g_depthMD.FullXRes()-1) / 512) + 1) * 512;
g_nTexMapY = (((unsigned short)(g_depthMD.FullYRes()-1) / 512) + 1) * 512;
g_pTexMap = (XnRGB24Pixel*)malloc(g_nTexMapX * g_nTexMapY * sizeof(XnRGB24Pixel));
```

glutDisplay() callback - Display Control

Significant OpenNI programming is performed inside the glutDisplay() callback.

Read the Frame Objects

The following code blocks read the frame objects from the DepthGenerator and ImageGenerator nodes.
The xn::Context::WaitAnyUpdateAll() "WaitAnyUpdateAll()" method in the following statement waits any node to have generated a new data frame. The method then makes the data frames of all nodes in the entire production graph available for getting. The application can then get the data (for example, using a metadata GetData() method). This method has a timeout.

```
nRetVal = context.WaitOneUpdateAll(depth);
```

The following code block calls the GetMetaData() methods of each of the two generator nodes to get the nodes' frame data from the frame objects – depthMD and g_imageMD, as already explained earlier.
The code then calls the Data() methods of each of the frame objects to get pointers – pDepth and pImage – into their respective map buffers. All further access to the data from the DepthGenerator and ImageGenerator nodes are through these frame objects.

```
g_depth.GetMetaData(g_depthMD);
g_image.GetMetaData(g_imageMD);
const XnDepthPixel* pDepth = g_depthMD.Data();
```

```
const XnUInt8* pImage = g_imageMD.Data();
```

Scale the Images

The following code uses the FullXRes() to calculate the scaling factor between the depth map and the GL window. FullXRes() gets the full frame resolution, i.e., the entire field-of-view, ignoring cropping of the FOV in the scene.

```
unsigned int nImageScale = GL_WIN_SIZE_X / g_depthMD.FullXRes();
```

Using the Depth Values to Build an Accumulative Histogram

The following code block uses the depth values to build an accumulative histogram of frequency of occurrence of each depth value. The *pDepth pointer accesses each value in the depth buffer. It then uses the value as an index into the g_pDepthHist histogram array.

```
xnOSMemSet(g_pDepthHist, 0, MAX_DEPTH*sizeof(float));


unsigned int nNumberOfPoints = 0;
for (XnUInt y = 0; y < g_depthMD.YRes(); ++y)
{
    for (XnUInt x = 0; x < g_depthMD.XRes(); ++x, ++pDepth)
    {
        if (*pDepth != 0)
        {
            g_pDepthHist[*pDepth]++;
            nNumberOfPoints++;
        }
    }
}
for (int nIndex=1; nIndex<MAX_DEPTH; nIndex++)
{
    g_pDepthHist[nIndex] += g_pDepthHist[nIndex-1];
}
if (nNumberOfPoints)
{
    for (int nIndex=1; nIndex<MAX_DEPTH; nIndex++)
    {
        g_pDepthHist[nIndex] = (unsigned int)(256 * (1.0f - (g_pDepthHist[nIndex] / nNumberOfPoints)));
    }
}
```