Kinect Chapters 1 & 2. Kinect Imaging

[**Note**: all the code for this chapter is available online at http://fivedots.coe.psu.ac.th/~ad/kinect/; only important fragments are described here.]

The following chapters are about controlling Microsoft's Kinect Sensor using Java. This chapter introduces the Kinect, explains how to install the developer software on Windows (which can be a bit tricky), and looks at basic imaging (depth detection, camera, and infrared (IR)). Future chapters will look at point clouds, skeletal user tracking in 2D and 3D, tilt motor control, hand tracking, basic gesture support, and gesture-controlled GUI components.

For readers already familiar with Kinect development, I'll be using OpenNI and NITE, not OpenKinect, CLNUI, or Microsoft's Kinect for Windows SDK. The 3D features (in the point cloud and skeletal tracking examples) are implemented using Java 3D.

I'll be using the Java API released with OpenNI and NITE in mid-July 2011; I won't be utilizing Processing libraries such as Simple-OpenNI.

1. The Kinect Sensor Hardware

The Kinect Sensor (Figure 1) was originally intended to be a motion sensing input device for the Xbox 360, allowing the user to control games via gestures and spoken commands. Key hardware components are a RGB camera, depth sensor, multi-array microphones, a tilt motor, and a three-axis accelerometer.

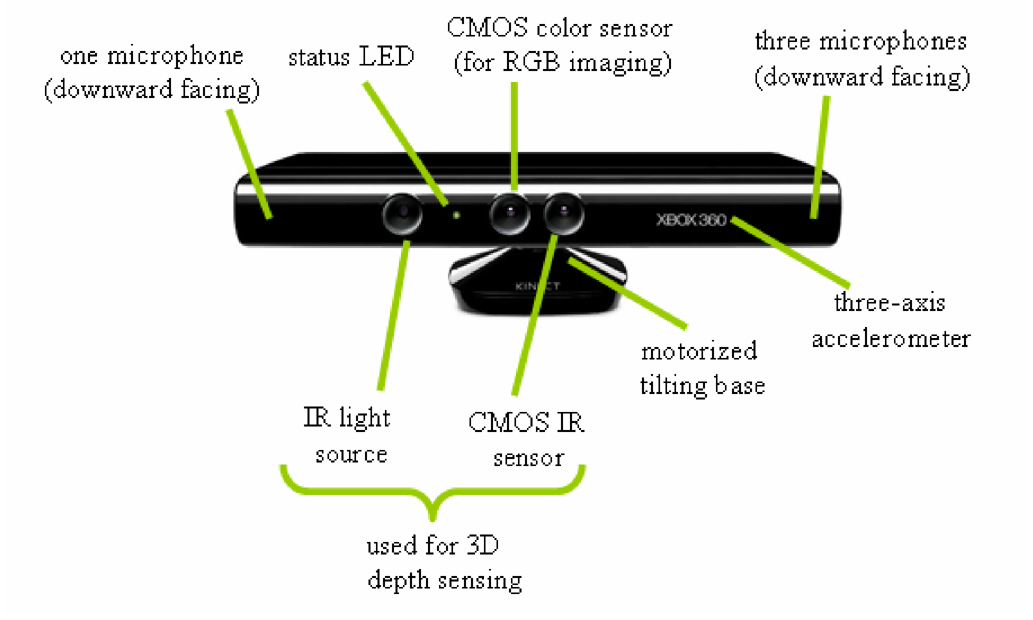

one microphone (downward facing)
status LED
CMOS color sensor (for RGB imaging)
three microphones (downward facing)
three-axis accelerometer
motorized tilting base
IR light source
CMOS IR sensor
used for 3D depth sensing

Figure 1. The Kinect Sensor.

Soon after its launch in November 2010, third party developers started writing software to allow the Kinect to be used on platforms other than just the Xbox (e.g. Windows, Linux, Mac), and Microsoft eventually released a Windows 7-based SDK in June 2011.

The Kinect's depth sensor consists of an infrared (IR) light source, a laser that projects a pattern of dots, that are read back by a monochrome CMOS IR sensor. The sensor detects reflected segments of the dot pattern and converts their intensities into distances. The resolution of the depth dimension (along the z-axis) is about one centimeter, while spatial resolution (along the x- and y- axes) is in millimeters.

Each frame generated by the depth sensor is at VGA resolution (640 × 480 pixels), containing 11-bit depth values which provides 2,048 levels of sensitivity. The output stream runs at a frame rate of 30 Hz.

The RGB video stream also utilizes VGA resolution and a 30 Hz frame rate.

The audio array consists of four microphones, with each channel processing 16-bit audio at a sampling rate of 16 kHz. The hardware includes ambient noise suppression.

The Kinect draws 12 watts, quite a bit more than 2.5 watts provided by a standard USB port. This means that it's necessary to plug the device into both the PC's USB port *and* into a separate power supply for the Kinect to function properly.

Microsoft suggests that you allow about 6 feet (1.8 meters) of empty space between you and the sensor. Standing too close, or in a cluttered environment, confuses the depth sensor.

The iFixit website has a detailed teardown of the Kinect's hardware (http://www.ifixit.com/Teardown/Microsoft-Kinect-Teardown/4066/1), with plenty of interesting pictures of its internals.

2. Kinect Development Software

There are four main Kinect development libraries: OpenKinect's libfreenect (http://openkinect.org/wiki/Main_Page), CLNUI (http://codelaboratories.com/nui/), OpenNI (http://www.openni.org/), and Microsoft's Kinect for Windows SDK (http://research.microsoft.com/en-us/um/redmond/projects/kinectsdk/default.aspx).

libfreenect is derived from a reverse-engineered/hacked Kinect driver, which works across Windows, Linux and MacOS X.

CLNUI (http://codelaboratories.com/nui/) is aimed at Windows only, but allows multiple Kinects to work together.

It's really not fair to compare libfreenect or CLNUI with OpenNI, which is intended to act as an interface between a variety of sensor drivers (including a Kinect driver) and middleware for manipulating the sensor data in a higher-level manner (e.g. for recognizing user gestures as commands). Figure 2 shows OpenNI's three-layer format (based on an image from the OpenNI user guide at http://www.openni.org/documentation).

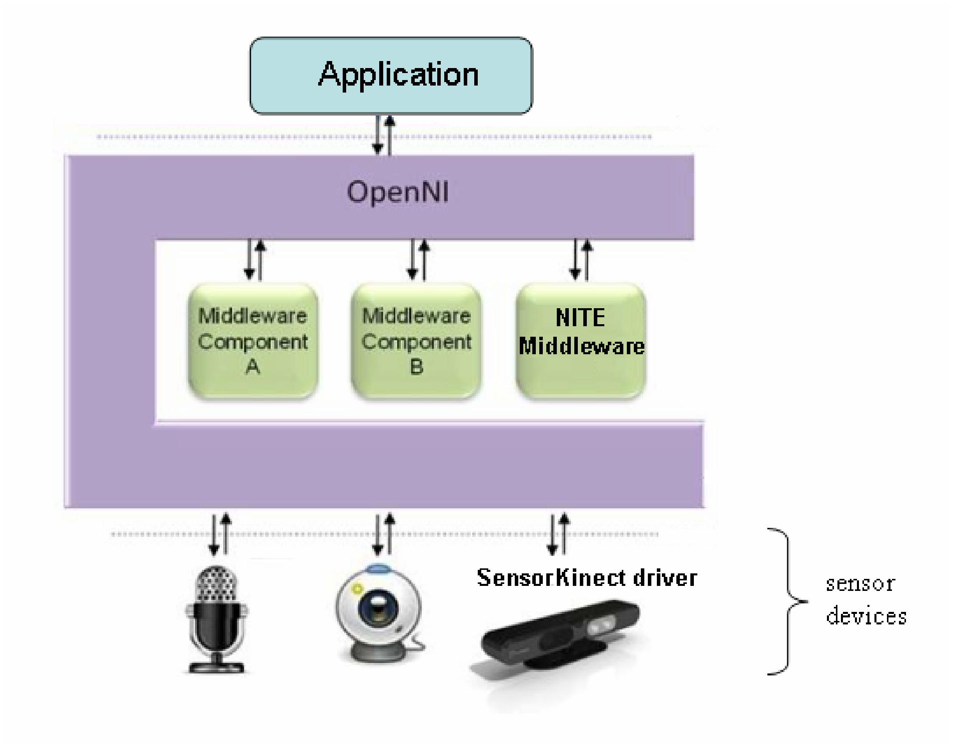


Figure 2. The Layers in the OpenNI Framework.

I don't need OpenNI 's multi-device, multi-middleware capabilities, as I'm only utilizing the Kinect and NITE middleware (which is explained below). OpenNI's Kinect driver is called SensorKinect, and is derived from open sourced (LGPL) code from PrimeSense, the company behind Kinect's depth sensor technology.

One advantage of libfreenect is API support for controlling the Kinect's tilt motor, status light, and accelerometer, which is missing from the SensorKinect driver. However, it's easy to add the necessary support using USB programming techniques I employed in the missile launcher and robot arm chapters. I'll describe the details in chapter 6.

An important reason for using OpenNI is its support for middleware. The NITE library understands different hand movements as gesture types based on how 'hand points' change over time. NITE gestures include pushing, swiping, holding steady (i.e. no hand movement), waving, and hand circling.

OpenKinect and OpenNI users have created numerous interesting, fun examples. Dedicated sites for Kinect hacking projects include http://www.kinecthacks.com/, http://kinect.dashhacks.com/, http://www.kinecthacks.nl/, and http://kinect-hacks.net/. More general maker sites with Kinect sections include http://hackaday.com/category/kinect-hacks/ and http://www.engadget.com/tag/kinect,hack YouTube is a good source for ideas and inspiration – search for "kinect" and "hacks".

2.1. The 800-pound Gorilla Enters

Eight months after the Kinect's release, Microsoft released a Kinect SDK (http://research.microsoft.com/en-us/um/redmond/projects/kinectsdk/download.aspx), which covers much the same ground as OpenNI. The low-level API gives access to the depth sensor, image sensor, and microphone array, while higher-level features include skeletal tracking, audio processing, and integration with the Windows speech recognition API.

The main area where the SDK wins over OpenNI is audio. Although OpenNI contains audio-related APIs, none of them are currently implemented (July 2011). The situation is somewhat better for OpenKinect, which has proof-of-concept audio code at https://github.com/OpenKinect/libfreenect/tree/master/examples (micview.c and wavrecord.c).

A nice aspect of Kinect audio support in Microsoft's SDK is that it treats one of the microphones as a USB audio device. This makes it possible for the Java sound API to connect to it and capture audio input. I'll discuss how to create a Microphone class for Java in NUI Chapter ??.

Other pluses for Microsoft's SDK are its extensive documentation and examples, and its ease of installation on Windows 7. As we'll see, both the documentation and installation steps for OpenNI are fairly good, but could be improved.

The main drawback of the SDK is that it only works on Windows 7, not XP or any non-windows platform. The SDK is free, but limited to non-commercial purposes. Also, applications must be written in C++, C#, or Visual Basic; naturally, I want to use Java ☺. These are the main reasons for my not programming with the SDK in these Kinect chapters.

2.2. Programming the Kinect in Java

Both libfreenect and OpenNI have Java and Processing bindings. Processing is a sketching language (http://processing.org/), that is translated into Java at compile time, so it's fairly easy to utilize Processing libraries in Java programs.

libfreenect lists two Java wrappers at http://openkinect.org/wiki/Main_Page, of which the JNA version comes with more information. A Processing binding called, somewhat confusingly, openkinect is at http://www.shiffman.net/p5/kinect/.

OpenNI has an excellent Processing binding called Simple-OpenNI (http://code.google.com/p/simple-openni/), which comes with good examples, clear installation instructions, and has an active forum.

If you only want to employ OpenNI's kinect driver, SensorKinect, then JKinect is a suitable Java wrapper for the driver. In a similar vein, there's a Java binding for the CLNUI Kinect driver, called CLNUI 4 Java (http://www.pixelnerve.com/v/2010/11/19/clnui-4-java-kinect/)

I'm not using any of these libraries because, at the start of July 2011, the OpenNI and NITE releases finally came with standard Java bindings. This means that I can program the Kinect without adding extra third-party libraries.

3. Kinect Installation

Installing OpenNI and NITE currently involves *three* separate packages, which must be installed in the correct order, onto a system that doesn't contain older OpenNI versions or other Kinect drivers (such as libfreenect or CLNUI). In addition, the installed libraries require some manual configuration before they'll work correctly.

This may seem like a pain, but the situation in the early days of Kinect development (i.e. 6 months ago) was much trickier, and the installation steps are getting easier. It's likely that the manual configuration hassles that I'm about to recount will be gone by the time you read this.

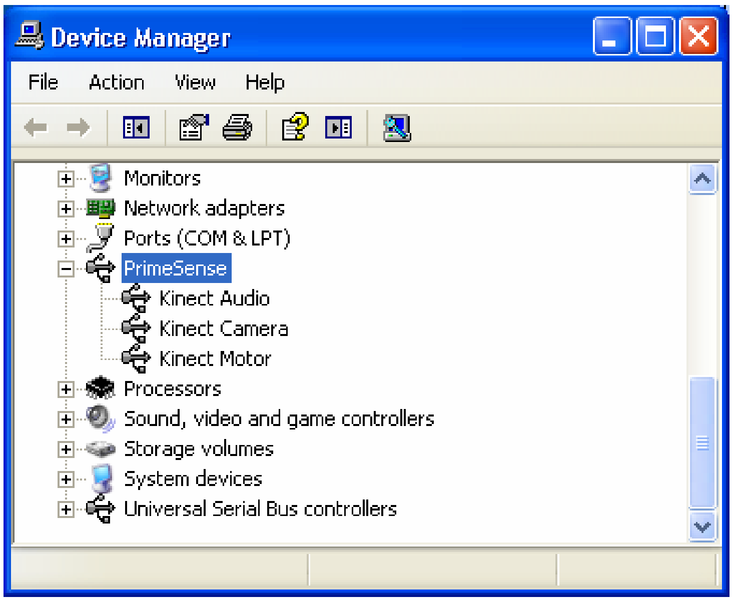
3.1. Cleaning Up First

Before you install the OpenNI and NITE packages, remove old versions and any Kinect drivers.

There are many freeware tools that can help with Windows cleanliness. I use Revo Uninstaller (http://www.revouninstaller.com/) to remove software since it also deletes unnecessary directories and menu items. CCleaner is also good (http://www.piriform.com/CCLEANER) for cleaning the Windows registry.

Any leftover bits of OpenNI and NITE will probably be located in C:\Program Files\PrimeSense (home to the SensorKinect and NITE code) and C:\Program Files\OpenNI. Both these directories should be deleted with extreme prejudice.

Also check the Device Manager, accessible via the Control Panel >> System >> Hardware tab (or by running devmgmt.msc from the command line). When the Kinect is plugged into the USB port *and* into the mains, then Kinect-related entries will appear on the Device manager list. Look for (at least) three drivers, for the Kinect's camera, motor and audio. You'll usually find them under a "PrimeSense" heading, or perhaps "PrimeSensor", as shown in Figure 3. The audio driver might be under "Other devices" or perhaps "Microsoft Kinect".



```
+ Monitors
+ Network adapters
+ Ports (COM & LPT)
- PrimeSense
    Kinect Audio
    Kinect Camera
    Kinect Motor
+ Processors
+ Sound, video and game controllers
+ Storage volumes
+ System devices
+ Universal Serial Bus controllers
```

Figure 3. The Kinect Drivers shown by the Device Manager.

All drivers that mention "Kinect" or "Xbox" should be uninstalled.

3.2. Downloading the Packages

The three Windows packages I downloaded in the middle of July 2011 were:

1. the *unstable* version of the Kinect driver, SensorKinect, version 5.0.3.4
2. the *unstable* version of OpenNI, development version 1.3.2.3
3. the stable version of NITE, development version 1.4.0.5

By the time you read this, these versions numbers will probably have moved on. When in doubt, download the latest SensorKinect driver, and read its documentation about which OpenNI and NITE versions are compatible.

The SensorKinect driver is available at GitHub in https://github.com/avin2/SensorKinect/. GitHub stores SensorKinect in two branches: a master branch for the current stable version, and a separate unstable branch.

I retrieved the *unstable* version: "PrimeSense Sensor Module for OpenNI (Version 5.0.3.4 Unstable version - Jul 19th 2011)". The naming is a bit confusing since the SensorKinect driver is a fork of the PrimeSense driver and uses a similar numbering scheme for its versions. I went for the unstable version since it had fixed some bugs in the stable version.

Various SensorKinect installers are in the Bin\ folder of the unzipped download; for example, "SensorKinect-Win-OpenSource32-5.0.3.4.msi", but **don't** install anything yet.

Make sure to look at the "README" and "CHANGES" files in the unzipped SensorKinect folder for useful installation information. For instance, the README in my download included links to the versions of OpenNI and NITE that would work with the driver. It also explained that OpenNI had to be installed before the SensorKinect driver.

OpenNI and NITE can be downloaded from http://www.openni.org/downloadfiles/opennimodules/.The OpenNI Modules page points to the two packages I need:

1. "OpenNI Binaries" is the OpenNI download page, which is divided into stable and unstable branches. I retrieved "OpenNI Unstable Build for Windows x86 (32-bit) v1.3.2.3 Development Edition", which shouldn't be confused with the x64 (64-bit) or Redist versions.
2. "OpenNI Compliant Middleware Binaries" is the NITE download page, which is also divided into stable and unstable branches. I retrieved "PrimeSense NITE Stable Build for Windows x86 (32-bit) v1.4.0.5 Development Edition" from the stable branch. Again, don't get confused with the x64 (64-bit) or Redist versions.

3.3. Installation and Configuration

Rather unintuitively, it's necessary to install the packages in the order:

1. OpenNI (OpenNI-Win32-1.3.2.3-Dev.msi)

2. SensorKinect drive (SensorKinect-Win-OpenSource32-5.0.3.4.msi)
3. NITE (NITE-Win32-1.4.0.5-Dev.msi)

The msi files will automatically store the development kits in C:\Program Files\OpenNI\ and C:\Program Files\PrimeSense\ (for SensorKinect and NITE).

Before getting Windows to install the hardware drivers, it's necessary to plug the Kinect into the PC and into the mains. Three drivers, for the motor, audio, and camera, need to be installed, and it's best if you specify their location rather than letting Windows' Hardware 'Wizard' search itself. All the drivers are in C:\Program Files\PrimeSense\SensorKinect\Driver. The Hardware Wizard will execute three times, once for each driver.

Once the drivers are in place, check the Device Manager – it should look something like Figure 3.

A recent 'innovation' of OpenNI's installation process is to automatically add a Java CLASSPATH environment variable to Windows. I prefer to invoke Java with an explicit classpath argument in calls to javac.exe and java.exe, and so don't need the environment variable. More to the point, it's addition to the OS breaks some of my other Java code. Go to the System control panel, and delete the variable via the Advanced tab.

3.4. Fixing the XML Configuration Files

The OpenNI and NITE installations come with precompiled executables that are useful for testing the Kinect and software. However, four XML configuration files must be modified before the programs will work correctly: three files (Sample-Scene.xml, Sample-Training.xml, and Sample-User.xml) in C:\Program Files\PrimeSense\NITE\Data and SampleConfig.xml in C:\Program Files\OpenNI\Data.

A free PrimeSense license key must be added to each XML file:

```
<License vendor="PrimeSense" key="0KOIk2JeIBYClPWVnMoRKn5cdY4="/>
```

Also, all the <Node> terms must have a <MapOutputMode> attribute added to their <Configuration> subterm:

```
<MapOutputMode xRes="640" yRes="480" FPS="30"/>
```

This specifies the default resolution and frame rate of the Kinect's CMOS devices.

The changes to SamplesConfig.xml are shown in bold below:

```
<OpenNI>
  <Licenses>
    <License vendor="PrimeSense" key="0KOIk2JeIBYClPWVnMoRKn5cdY4="/>
  </Licenses>
  <Log writeToConsole="false" writeToFile="false">
    <!-- 0 - Verbose, 1 - Info, 2 - Warning, 3 - Error (default) -->
    <LogLevel value="3"/>
    <Masks>
```

```
      <Mask name="ALL" on="true"/>
    </Masks>
    <Dumps>
    </Dumps>
  </Log>
  <ProductionNodes>
    <Node type="Depth" name="Depth1">
      <Configuration>
        <MapOutputMode xRes="640" yRes="480" FPS="30"/>
        <Mirror on="true"/>
      </Configuration>
    </Node>
    <Node type="Image" name="Image1" stopOnError="false">
      <Configuration>
        <MapOutputMode xRes="640" yRes="480" FPS="30"/>
        <Mirror on="true"/>
      </Configuration>
    </Node>
    <!--
    <Node type="Audio" name="Audio1"/>
    -->
  </ProductionNodes>
</OpenNI>
```

3.5. Testing OpenNI and NITE

There are several programs in C:\Program Files\OpenNI\Samples\Bin\Release that can be used to test OpenNI, including NiSimpleRead.exe, NiSimpleViewer.exe, NiViewer.exe, and NiUserTracker.exe.

NiSimpleRead produces a stream of text showing the depth measurement for the center of the Kinect's viewing area. For instance:

```
Frame 65 Middle point is: 3019. FPS: 30.556658
```

Wave your hand in front of the Kinect, to see the distance change.

NiSimpleViewer displays a depth map, shown in Figure 4 (OpenNI uses the word 'map' instead of image in its documentation.)

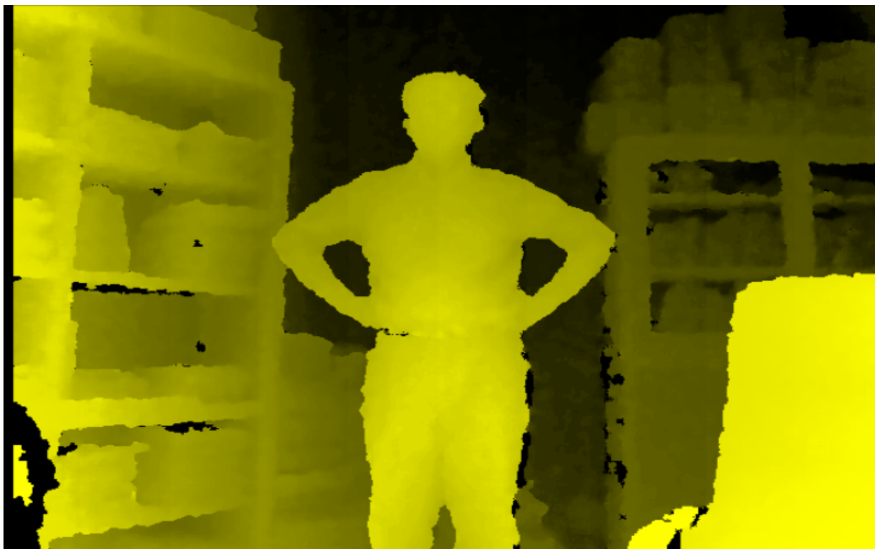


Figure 4. NiSimpleViewer's Depth Map.

NiViewer can display different map combinations (see Figure 5); type '?' to get a help screen of commands.

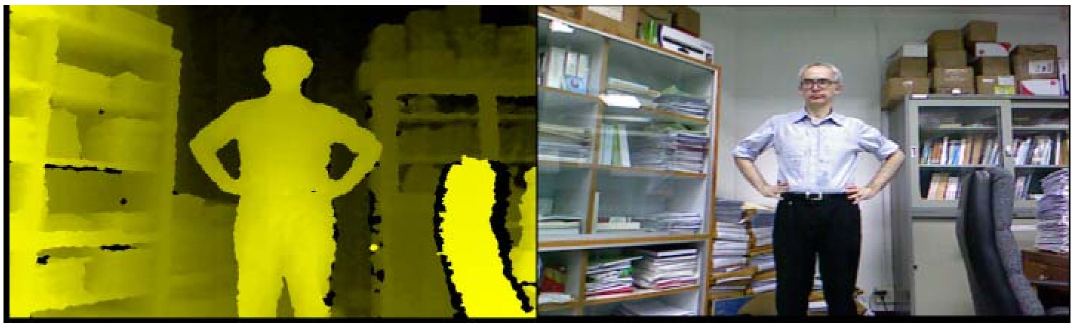


Figure 5. NiViewer's Maps Display.

NiUserTracker illustrates pose detection and skeleton capabilities, as shown in Figure 6.

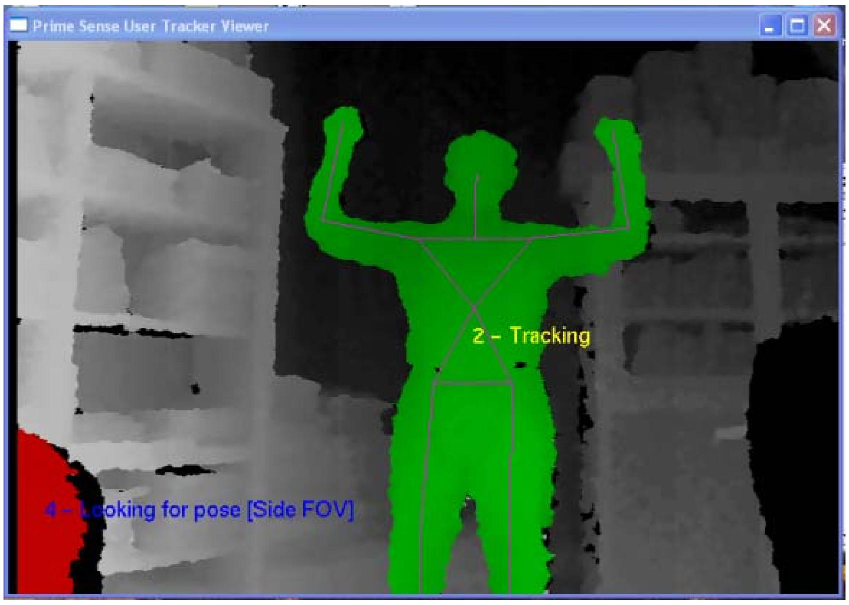

Prime Sense User Tracker Viewer
2 - Tracking
4 - Looking for pose [Side FOV]

Figure 6. NiUserTracker tracking a user.

NiUserTracker works best if the Kinect can 'see' all (or most) of the user. Also, the skeleton (the thin lines inside the body shape) only appear after the person has stood still with both arms raised, in the so-called 'psi' pose shown in Figure 6, for a few seconds.

These examples, and others, are briefly explained in the OpenNI User Guide, in the OpenNI documentation folder.

The NITE examples are in C:\Program Files\PrimeSense\NITE\Samples\Bin\Release, with the simplest probably being Sample-Boxes.exe. It uses a variety of NITE gestures ('wave', 'push', 'swipe', 'steady') to select between boxes (see Figure 7).

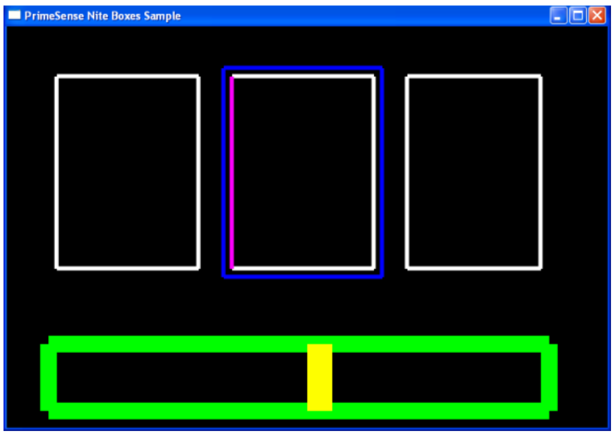


Figure 7. Selecting Boxes.

The user starts the selection process by waving his hand in front of the Kinect, which can take several seconds to notice the gesture.

Sample-Boxes, and the other examples, are briefly explained in the NITE Controls User Guide in the NITE documentation folder.

4. Documentation and Other Sources of Help

Aside from the documentation that comes with the OpenNI and NITE downloads, the sample code is a great source of ideas. The directories C:\Program Files\OpenNI\Samples and C:\Program Files\PrimeSense\NITE\Samples contain the source code for the programs, which is mostly in C/C++, but there are also some Java examples.

The OpenNI forum at Google Groups is an excellent source of information on OpenNI and NITE (http://groups.google.com/group/openni-dev). The Wiki maintained by the OpenKinect group is also valuable for background on the Kinect hardware and communication protocols (http://openkinect.org/wiki/Main_Page). There's also an OpenKinect forum at Google (http://groups.google.com/group/openkinect/topics).

Another interesting read is the Programming Guide for Microsoft's Kinect SDK (http://research.microsoft.com/en-us/um/redmond/projects/kinectsdk/docs/programmingguide_kinectsdk.pdf), which explains some of the concepts behind the Kinect, although the programming details aren't applicable to OpenNI or NITE.

The Kinect hack sites which I mentioned earlier have technical sections (http://www.kinecthacks.com/, http://kinect.dashhacks.com/, http://www.kinecthacks.nl/, and http://kinect-hacks.net/).

Java Programming Documentation

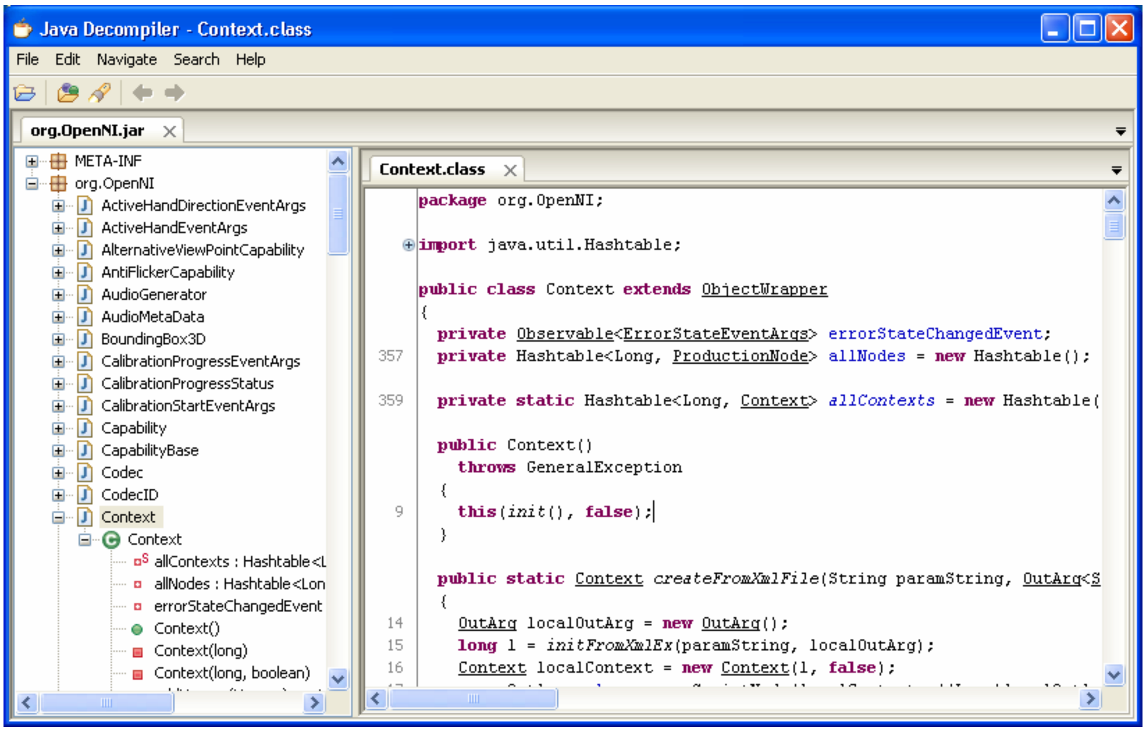
The Java APIs for OpenNI and NITE were introduced in July 2011 and currently (September 2011) there's not much information about them.

The OpenNI samples include four Java examples: HandTracker.java, SimpleRead.java, SimpleViewer.java, and UserTracker.java. The last three are the equivalents of NiSimpleRead.exe, NiSimpleViewer.exe, and NiUserTracker.exe, which I described above. The NITE samples include Boxes.java, a Java version of Sample-Boxes.exe from above.

A good source of "almost-Java" examples is the Simple-OpenNI Processing library at http://code.google.com/p/simple-openni/. There's a Simple-OpenNI forum at http://groups.google.com/group/simple-openni-discuss.

The compiled OpenNI and NITE Jar files containing the Java interface to the C API are very useful as documentation (if you use the right tools). They are located at C:\Program Files\OpenNI\Bin\org.OpenNI.jar and C:\Program Files\PrimeSense\NITE\Bin\ com.primesense.NITE.jar.

JAR files can be decompiled, and I highly recommend the freeware JD-GUI decompiler from http://java.decompiler.free.fr/?q=jdgui for this task. It displays decompiled source code organized by classes, methods, and fields, with hypertext-like links between the classes. Figure 8 shows JD-GUI's view of the Context class in org.OpenNI.jar.



```
META-INF
org.OpenNI
  ActiveHandDirectionEventArgs
  ActiveHandEventArgs
  AlternativeViewPointCapability
  AntiFlickerCapability
  AudioGenerator
  AudioMetaData
  BoundingBox3D
  CalibrationProgressEventArgs
  CalibrationProgressStatus
  CalibrationStartEventArgs
  Capability
  CapabilityBase
  Codec
  CodecID
  Context
    Context
      allContexts : Hashtable<L
      allNodes : Hashtable<Lon
      errorStateChangedEvent
      Context()
      Context(long)
      Context(long, boolean)
```

```
package org.OpenNI;

import java.util.Hashtable;

public class Context extends ObjectWrapper
{
  private Observable<ErrorStateEventArgs> errorStateChangedEvent;
  private Hashtable<Long, ProductionNode> allNodes = new Hashtable();

  private static Hashtable<Long, Context> allContexts = new Hashtable(

  public Context()
    throws GeneralException
  {
    this(init(), false);
  }

  public static Context createFromXmlFile(String paramString, OutArg<S
  {
    OutArg localOutArg = new OutArg();
    long l = initFromXmlEx(paramString, localOutArg);
    Context localContext = new Context(l, false);
```

Figure 8. JD-GUI's view of the org.OpenNI .Context class

It's possible to save the decompiled source as Java text files, which allows it to be further investigated using ESS-MODEL to produce class diagrams (http://essmodel.sourceforge.net/). Unfortunately, ESS-MODEL hasn't been updated in a while, so can't correctly translate Java generics into diagrams. Nevertheless, I've used ESS-MODEL to produce all the class diagrams in this chapter.

5. OpenNI Concepts in Java

The main functional element in OpenNI is the production node. A node may represent a sensor's functionality (e.g. depth detection, infrared, audio) or a higher-level abstraction (e.g. user positioning, hand tracking). An OpenNI application always starts by initializing the nodes it needs for its particular task.

In the Java API, the ProductionNode class is subclassed to create various kinds of nodes (see Figure 9).

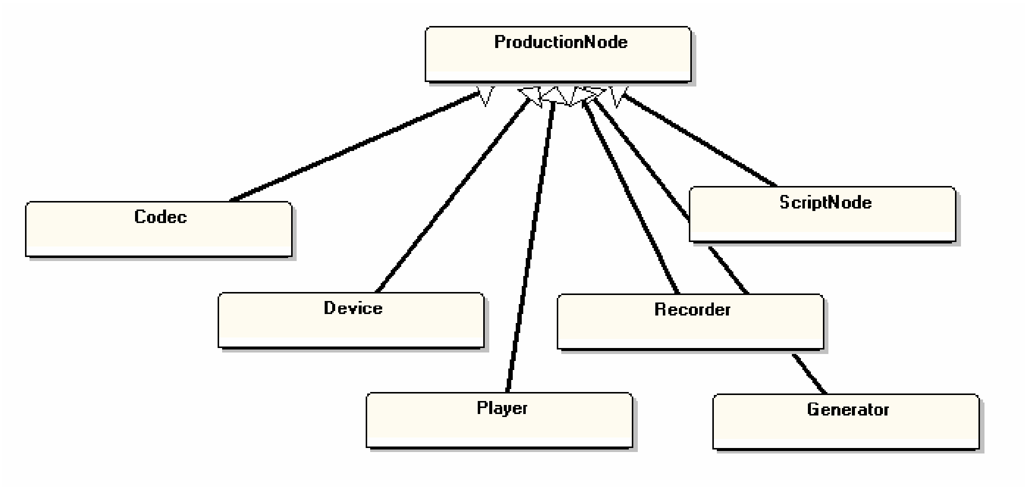


Figure 9. Subclasses of the ProductionNode Class.

For example, the Device class is for device configuration, while the Recorder, Player, and Codec classes are for data recording. However, most programs tends to use subclasses of Generator, which represent sensor and middleware features (see Figure 10).

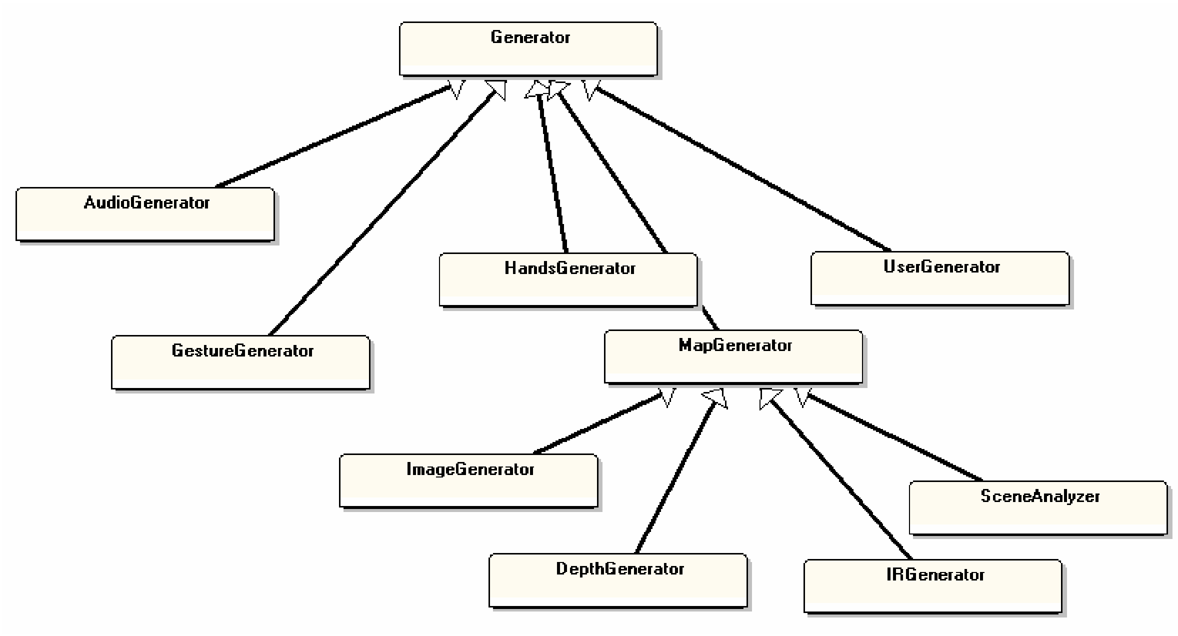


Figure 10. Subclasses of the Generator Class.

The MapGenerator subclasses typically output streams of images (which are called maps in OpenNI).

The sensor-related Generator subclasses are:

AudioGenerator: for producing an audio stream. This class, and other audio related classes, are currently not implemented (July 2011).

DepthGenerator: for creating a stream of depth maps;

ImageGenerator: for creating colored image maps;

IRGenerator: for creating infrared image maps.

The middleware-related classes are:

GesturesGenerator: for recognizing hand gestures, such as waving and swiping;

HandsGenerator: for hand detection and tracking;

SceneAnalyzer: for separating the foreground from the background in a scene, labeling the figures, and detecting the floor. The main output is a stream of labeled depth maps;

UserGenerator: generates a representation of a (full or partial) body in the scene.

5.1. Generating Data and Metadata

The map generators (the depth, image, IR and scene generators in Figure 10) make their data available as subclasses of Map and MapMetaData (see Figure 11). Metadata are data properties (as opposed to the data itself), such as map resolution.

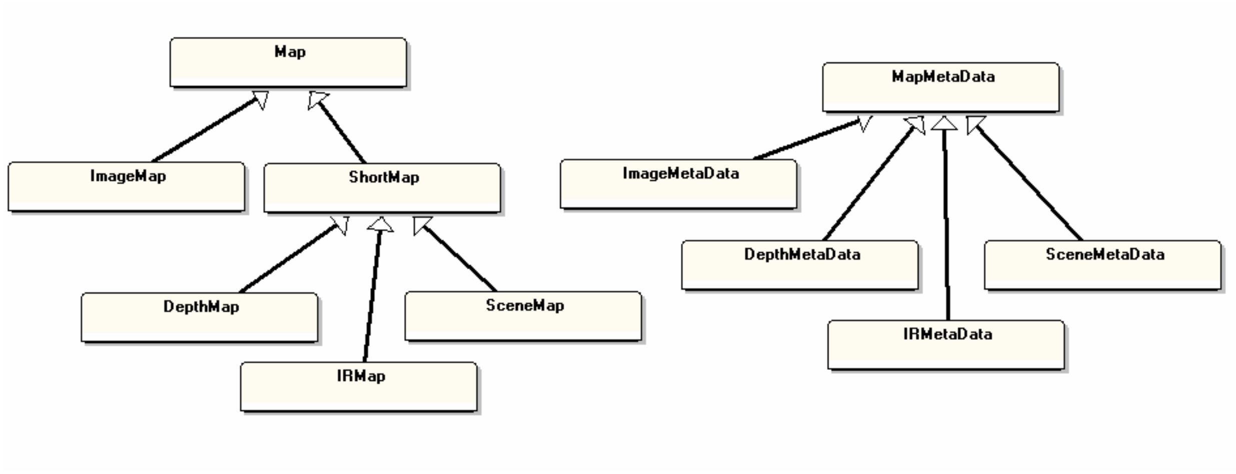


Figure 11. Map and MapMetaData Subclasses

5.2. Listeners

Generators can also interact with user programs via listeners (called callbacks in OpenNI). The MapGenerators tend to use StateChangedObservable listeners, while other generators (e.g. for hands and gesture recognition) use Observable<EventArgs> listeners, with different EventArgs subclasses (shown in Figure 12).

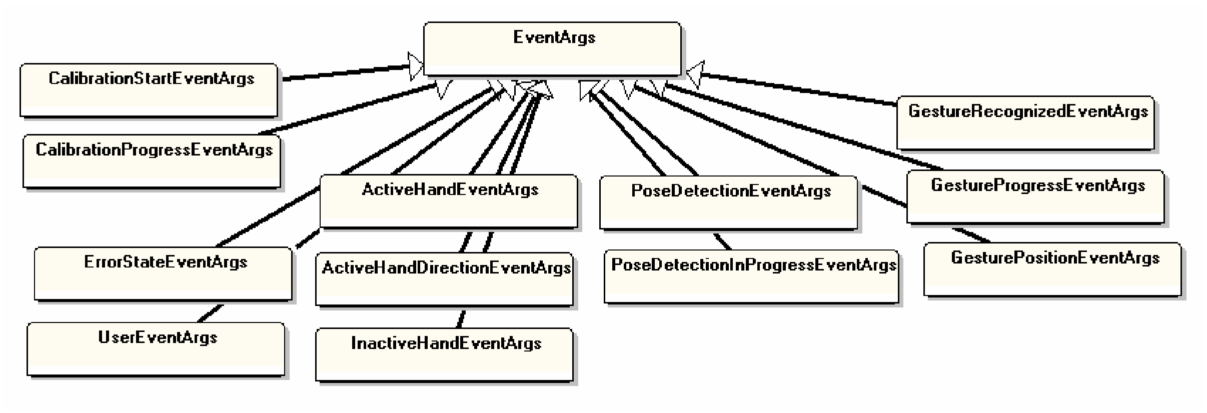


Figure 12. EventArgs and its Subclasses.

I've arranged the classes in Figure 12 so related types of event arguments are close together. For instance, there are several types of event generated for hand movement, poses, and gestures. I'll describe these in later chapters.

5.3. Capabilities

OpenNI is intended to work with different sensors, so the capabilities of an instantiated production node may vary. For instance, some webcams may allow an image map to be cropped. Each capability is represented by a subclass of CapabilityBase, as shown in Figure 13.

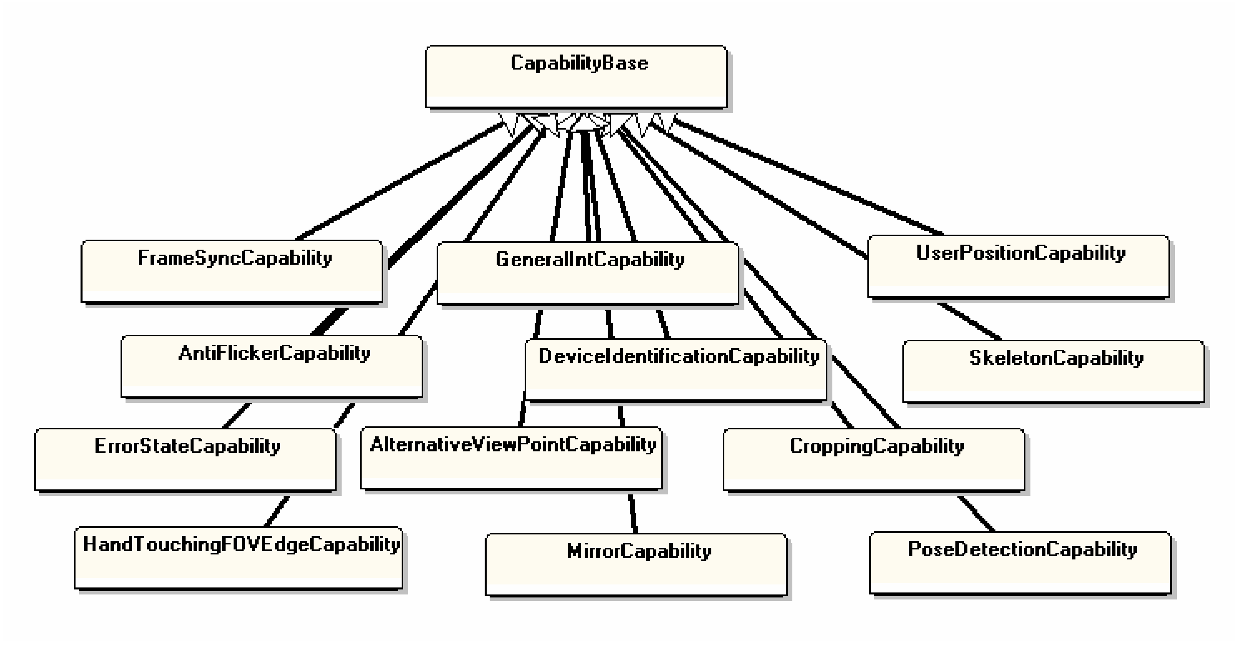


Figure 13. The Capability Classes.

The purpose of some of these classes:

AlternativeViewPointCapability: Enables a map generator (e.g. depth, image, IR) to transform its data to appear as if the sensor is in another location. This is useful when two maps must be superimposed, and need aligning.

CroppingCapability: Enables a map generator (e.g. depth, image, IR) to output a selected area of the frame instead of the entire thing. Cropping can boost performance.

FrameSyncCapability: Enables two sensors to synchronize their frame data (e.g. depth and image), so they show the same thing when they are displayed together.

MirrorCapability: Enables mirroring of the data produced by a generator so, for example, the user's right hand appears on the left. This makes it easier to interact with the image when it's displayed on-screen.

Since I'll only be using the Kinect sensor and NITE middleware, I'll not often utilize the capability classes; instead, I'll simply assume that the standard set of Kinect features are available. However, I will employ AlternativeViewPointCapability in version 6 of the ViewerPanel class in section 12.1.

6. Working with Maps

In the rest of this chapter, I'll look at how to manipulate depth, image, and IR maps, using subclasses of MapGenerator (i.e. DepthGenerator, ImageGenerator, IRGenerator), subclasses of Map for the data (i.e. DepthMap, ImageMap, IRMap) and subclasses of MapMetaData for the metadata (i.e. DepthMetaData, ImageMetaData and IRMetaData). In other words, I'll be exploring some of the classes shown in Figures 10 and 11.

The discussion is based around six variations of a OpenNIViewer application which draws the current map on a JPanel inside a JFrame. Figure 14 shows the version that displays a grayscale depth map.

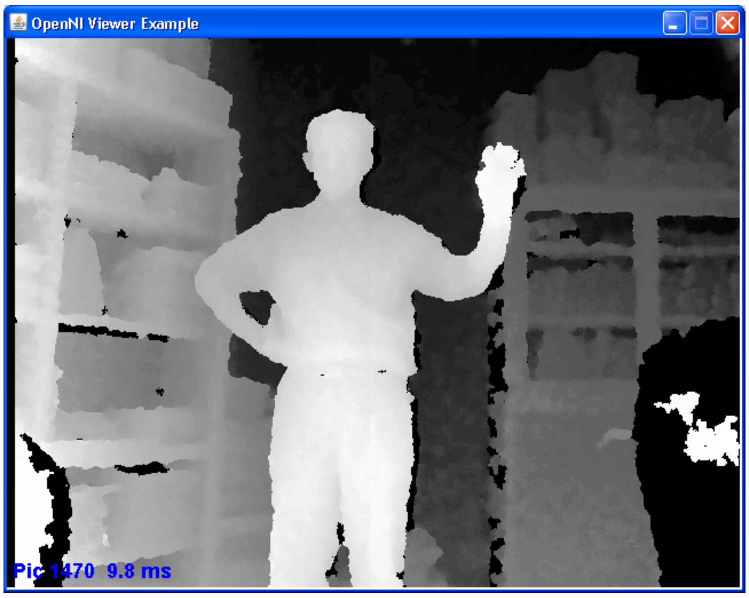


Figure 14. Viewing a Grayscale Depth Map.

The six versions of OpenNIViewer all have the same class structure, shown in Figure 15.

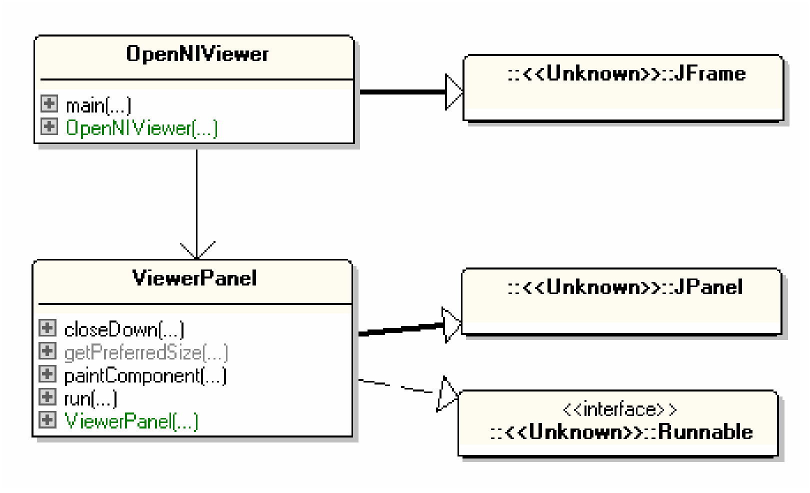


Figure 15. UML Class Diagrams for OpenNIViewer.

OpenNIViewer is a JFrame which places a ViewerPanel object at its center. All the OpenNI processing takes place inside ViewerPanel using a thread to regularly update and draw the map onto the panel. As a consequence, OpenNIViewer is the same in all versions of the viewer; and the interesting OpenNI work is done by six versions of ViewerPanel. These versions are:

Version 1: draws a grayscale depth map specified using an XML configuration file. Version 1 is shown in Figure 14.

Version 2: also draws a grayscale depth map, but created at run time using a Context object. The output is identical to Figure 14.

Version 3: shows a camera image map.

Version 4: renders an IR map.

Version 5: draws a colorized depth map.

Version 6: shows a combined colorized depth and camera image.

I'll describe these different versions of ViewerPanel in the rest of this chapter.

7. Version 1. Grayscale Depth Map and XML

The first version of ViewerPanel is based on the SimpleViewer example in the OpenNI distribution (in OpenNI\Samples\SimpleViewer.java). The production nodes are initialized with information loaded from an XML file, and a continuously updating grayscale depth map is displayed. Darker shades of gray correspond to distances further away from the Kinect, although pitch black also means that the sensor was unable to calculate depths for points too close to the Kinect (e.g. the right hand lower edge of Figure 14).

The XML file (SamplesConfig.xml) is:

```
<OpenNI>
  <Licenses>
```

```
    <License vendor="PrimeSense" key="0KOIk2JeIBYClPWVnMoRKn5cdY4="/>
  </Licenses>
  <Log writeToConsole="true" writeToFile="false">
    <!-- 0 - Verbose, 1 - Info, 2 - Warning, 3 - Error (default) -->
    <LogLevel value="3"/>
    <Masks>
      <Mask name="ALL" on="true"/>
    </Masks>
    <Dumps>
    </Dumps>
  </Log>
  <ProductionNodes>
    <Node type="Depth" name="Depth1">
      <Configuration>
        <MapOutputMode xRes="640" yRes="480" FPS="30"/>
        <Mirror on="true"/>
      </Configuration>
    </Node>
<!--
    <Node type="Image" name="Image1" stopOnError="false">
      <Configuration>
        <MapOutputMode xRes="640" yRes="480" FPS="30"/>
        <Mirror on="true"/>
      </Configuration>
    </Node>
-->
  </ProductionNodes>
</OpenNI>
```

The file originally defined two production node for depth and image generation, but I've commented out the image generator information which isn't needed here.

The XML starts with a license line and also specifies that errors be sent to the command window (writeToConsole="true") which is useful during debugging.

The depth generator is set to return a 640 x 480 sized image, at a rate of 30 frames/sec, and image mirroring is turned on.

The ViewerPanel constructor loads the XML settings, creating a Context object in the process. The context holds the production nodes used by the application. DepthGenerator and DepthMetaData objects are created, as well as various image data structures

```
// globals
private static final int MAX_DEPTH_SIZE = 10000;

private static final String SAMPLE_XML_FILE = "SamplesConfig.xml";

// image variables
private byte[] imgbytes;
private BufferedImage image = null;   // displays the depth image
private int imWidth, imHeight;
private float histogram[];   // for the depth values
private int maxDepth = 0;    // largest depth value

// OpenNI
private Context context;
private DepthMetaData depthMD;
```

```
public ViewerPanel()
{
  setBackground(Color.WHITE);

  df = new DecimalFormat("0.#");  // 1 dp
  msgFont = new Font("SansSerif", Font.BOLD, 18);

  try {
    // create a context
    OutArg<ScriptNode> scriptNode = new OutArg<ScriptNode>();
    context = Context.createFromXmlFile(SAMPLE_XML_FILE, scriptNode);

    DepthGenerator depthGen = DepthGenerator.create(context);
    depthMD = depthGen.getMetaData();
         // use depth metadata to access depth info
         // (avoids bug with DepthGenerator)
  }
  catch (GeneralException e) {
    System.out.println(e);
    System.exit(1);
  }

  histogram = new float[MAX_DEPTH_SIZE];
                  // used for depth values manipulation

  imWidth = depthMD.getFullXRes();
  imHeight = depthMD.getFullYRes();
  System.out.println("Image dimensions (" + imWidth + ", " +
                                          imHeight + ")");

  // create empty image data array
  imgbytes = new byte[imWidth * imHeight];

  // create empty BufferedImage of correct size and type
  image = new BufferedImage(imWidth, imHeight,
                            BufferedImage.TYPE_BYTE_GRAY);

  new Thread(this).start();   // start updating the panel's image
} // end of ViewerPanel()
```

DepthGenerator and DepthMetaData objects are created in order to program around a bug in DepthGenerator. Unfortunately, DepthGenerator.getDepthMap() doesn't return a map (image), but the same information can be accessed via DepthMetaData.getData().

The histogram[] array is an important non-OpenNI data structure, which is used to record depth values and to convert them into integers between 0 and 255. These integers are later treated at 8-bit grayscale values, and used to initialize the grayscale BufferedImage for the depth picture.

7.1. Updating the Panel

The ViewerPanel's run() method repeatedly updates the depth image, and draws it onto the panel.

The simplest way of implementing the update-redraw-sleep looping behavior of run() is to utilize one of the Context class' wait() methods which suspends execution until new data is available, and then updates all the relevant production nodes.

```
// globals
private volatile boolean isRunning;

// used for the average ms processing information
private int imageCount = 0;
private long totalTime = 0;


public void run()
{
  isRunning = true;
  while (isRunning) {
    try {
      context.waitAnyUpdateAll();
    }
    catch(StatusException e)
    {  System.out.println(e);
       System.exit(1);
    }
    long startTime = System.currentTimeMillis();
    updateDepthImage();
    imageCount++;
    totalTime += (System.currentTimeMillis() - startTime);
    repaint();
  }

  // close down
  try {
    context.stopGeneratingAll();
  }
  catch (StatusException e) {}
  context.release();
  System.exit(0);
}  // end of run()
```

There are several Context wait() methods, including:

- Context.waitAnyUpdateAll(): it waits for any node to produce new data. Once data is available, all the application's nodes are updated.
- Context.waitOneUpdateAll(ProductionNode node): it waits for the specified node to output new data. Once data is available from this node, all the nodes are updated. This method is useful when several nodes produce data, but only one determines the progress of the application.
- Context.waitAndUpdateAll(): it waits for all the nodes to generate new data, then updates them.

The wait-update loop in run() doesn't use Thread.sleep() to impose a fixed frame rate on each iteration of the loop. The loop executes as quickly as the production nodes are updated.

The termination of the loop (by setting isRunning to false) is followed by the cleanup of the context before the program exits.

7.2. From Depth Values to Image Pixels

updateDepthImage() retrieves the current depth map (using the depth metadata), and builds grayscale image data in two steps.

```
// globals
private byte[] imgbytes;
private float histogram[];


private void updateDepthImage()
{
  ShortBuffer depthBuf = depthMD.getData().createShortBuffer();
                           // current depths map
  calcHistogram(depthBuf);   // convert depths to ints
  depthBuf.rewind();

  // store ints as bytes in imgbytes[] pixel array
  while (depthBuf.remaining() > 0) {
    int pos = depthBuf.position();    // pixel position of depth
    short depth = depthBuf.get();     // depth measure
    imgbytes[pos] = (byte) histogram[depth];
             // store depth's grayscale at depth's pixel pos
  }
}  // end of updateDepthImage()
```

First, calcHistogram() converts the depth measurements (millimeter distances from the Kinect) into integers in the range 0-255, which are stored in histogram[].

The histogram[] array is used in a rather complex way in updateDepthImage(). Its easiest to think of its integer values as 8-bit grayscales, while its indices are depth measurements. For instance, if histogram[100] == 255, then it means that the 100mm depth is represented by the color white.

In updateDepthImage(), histogram[] and the depth buffer are used to initialize the imgbytes[] array. The pixel position of each depth measurement is read from the buffer, and that depth's grayscale from histogram[]. The grayscale is stored (as a byte) at that pixel position in imgbytes[].

At render time (which I describe in the next subsection), the bytes in imgbytes[] are used to fill the 2D data buffer in the grayscale depth BufferedImage.

calcHistogram() converts depths measurements to grayscales (integers in the range 0 to 255) in four stages.

```
// globals
private float histogram[];
private int maxDepth = 0;    // largest depth value


private void calcHistogram(ShortBuffer depthBuf)
{
  // reset histogram[]  (stage 1)
  for (int i = 0; i <= maxDepth; i++)
    histogram[i] = 0;
  // store depth counts in histogram[];
  // a depth (an integer mm value) is used as an index
  // into the array (stage 2)
  int numPoints = 0;
  maxDepth = 0;
  while (depthBuf.remaining() > 0) {
    short depthVal = depthBuf.get();
    if (depthVal > maxDepth)
      maxDepth = depthVal;
    if ((depthVal != 0)  && (depthVal < MAX_DEPTH_SIZE)){
                 // skip histogram[0]
      histogram[depthVal]++;
      numPoints++;
    }
  }

  // convert into a cummulative depth count (skipping histogram[0])
  for (int i = 1; i <= maxDepth; i++)    // stage 3
    histogram[i] += histogram[i - 1];

  // convert cummulative depth into integers (0-255); stage 4
  if (numPoints > 0) {
    for (int i = 1; i <= maxDepth; i++)   // skip histogram[0]
      histogram[i] =
        (int) (256 * (1.0f - (histogram[i] / (float) numPoints)));
  }
}  // end of calcHistogram()
```

1. The first stage resets the histogram[] array.
2. In the second stage, the number of pixels at each distance (or depth) from the camera is stored in histogram[]. Each depth (an integer millimeter value) is used as an index into the array. For example, histogram[100] will record the number of pixels that are 100mm from the Kinect.
3. The third stage accumulates the histogram[] counts, so that each histogram[] cell contains the number of the pixels at a certain depth *or less* from the Kinect. For instance, histogram[100] will record the number of pixels that are 100mm or less from the camera. Because the totals are cummulative, histogram[maxDepth] will hold the total number of pixels in the depth map.
4. The last stage transforms the cummulative counts into integer values between 0 and 255. Each histogram[] value is transformed into a fraction by dividing its value (the number of pixels equal or less than a certain depth) by the total number of pixels. This will mean that histogram[maxDepth] will be mapped to 1.0. Each fraction is subtracted from 1.0, so that larger numbers will be mapped to a fraction closer to 0. Finally, each fraction is transformed into an integer between 0 and 255. These integers will later be treated as 8-bit grayscales, and so greater depths will have smaller grayscale values, and so be rendered nearer to black.

calcHistogram() treats histogram[0] as a special case. It's used to represent "no value", which occurs when a pixel is too close to the Kinect to be assigned a depth. Since histogram[0] always stores 0, it will be mapped to black when imgbytes[] is converted into grayscale image data.

7.3. Rendering the Depth Map.

ViewerPanel's paintComponent() writes imgbytes[]'s pixel data into a BufferedImage before drawing it. To follow the steps, it helps to understand how a BufferedImage is built from component parts, as shown in Figure 16.

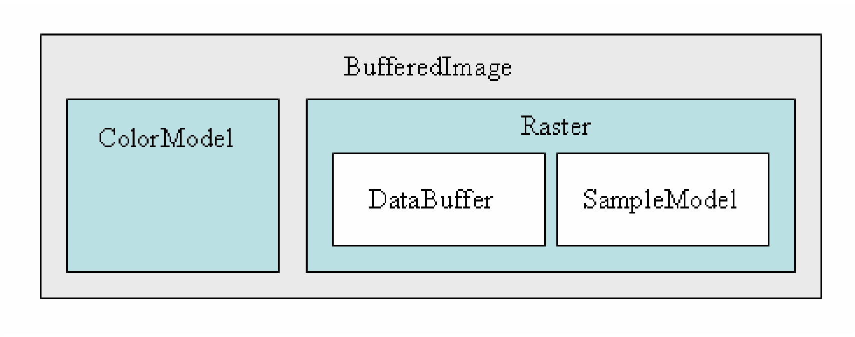


Figure 16. The Main Components of a BufferedImage.

Every BufferedImage consists of a raster and a color model. The pixel data is stored in the raster, while the color model maps pixel values to colors.

Inside a raster, a data buffer contains pixel *samples*, stored in an array of bytes or as integers. Every pixel consists of one or more samples, which together represent a color. A grayscale image, for example, will store one sample for each pixel. An RGB image will store three samples per pixel.

The other part of the raster, the sample model, knows how to extract samples from a pixel in the data buffer.

The color model interprets a pixel's samples as color. In a grayscale image, which has one sample per pixel, the color model maps a sample to a color between black and white. In an RGB image, the color model uses the three samples in a pixel as red, green, and blue components.

The BufferedImage used in paintComponent() has its color and sample models set in ViewerPanel's constructor when its type is set to TYPE_BYTE_GRAY:

```
image = new BufferedImage(imWidth, imHeight,
                                 BufferedImage.TYPE_BYTE_GRAY);
```

paintComponent()'s task is to convert the 1D imgbytes[] array into a data buffer, and use it to initialize the raster component of the image.

```
// globals
private byte[] imgbytes;
private BufferedImage image = null;
private int imWidth, imHeight;


public void paintComponent(Graphics g)
{
  super.paintComponent(g);
  Graphics2D g2 = (Graphics2D) g;

  // store image pixel array in the BufferedImage
  DataBufferByte dataBuffer =
            new DataBufferByte(imgbytes, imWidth * imHeight);
  Raster raster = Raster.createPackedRaster(dataBuffer,
                                  imWidth, imHeight, 8, null);
  image.setData(raster);
  if (image != null)
    g2.drawImage(image, 0, 0, this);

  writeStats(g2); // write statistics info at bottom left of panel
} // end of paintComponent()
```

8. Version 2. Grayscale Depth Map without XML

It's possible to initialize the application's context without using an XML configuration file. The building of a context from a script node using Context.createFromXmlFile() can be replaced by the explicitly creation of a context and its production nodes.

configOpenNI(), shown below, builds a Context object from scratch, which requires a valid license with "vendor" and "key" string attributes. In this example, only a single DepthGenerator production node is added to the context.

```
// globals
private Context context;
private DepthMetaData depthMD;


private void configOpenNI()
// create context and depth generator
{
  try {
    context = new Context();

    // add the NITE License
    License license = new License("PrimeSense",
               "0KOIk2JeIBYClPWVnMoRKn5cdY4=");   // vendor, key
    context.addLicense(license);

    DepthGenerator depthGen = DepthGenerator.create(context);
    MapOutputMode mapMode = new MapOutputMode(640, 480, 30);
                                       // xRes, yRes, FPS
    depthGen.setMapOutputMode(mapMode);

    // set Mirror mode for all
    context.setGlobalMirror(true);

    context.startGeneratingAll();
    System.out.println("Started context generating...");

    depthMD = depthGen.getMetaData();
         // use depth metadata to access depth info
  }
  catch (Exception e) {
    System.out.println(e);
    System.exit(1);
  }
}  // end of configOpenNI()
```

The DepthGenerator object must have its x- and y- axis resolutions set, along with its frame rate. Image mirroring is also switched on.

After the production nodes have been added to the context, they are started via a call to Context.startGeneratingAll().

9. Version 3: Camera Image Map

This version of ViewerPanel displays a RGB camera image instead of a depth map, as illustrated by Figure 17.

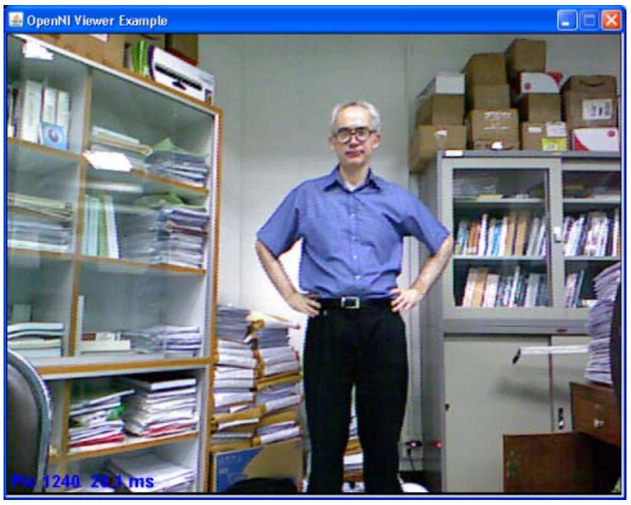


Figure 17. Viewing a RGB Camera Image Map.

The ViewerPanel constructor calls configureNITE() to build a Context object, but creates an ImageGenerator production node instead of a DepthGenerator.

```
// globals
private Context context;
private ImageGenerator imageGen;


private void configOpenNI()
// create context and image generator
{
  try {
    context = new Context();

    // add the NITE License
    License license = new License("PrimeSense",
                    "0KOIk2JeIBYClPWVnMoRKn5cdY4=");   // vendor, key
    context.addLicense(license);

    imageGen = ImageGenerator.create(context);

    MapOutputMode mapMode = new MapOutputMode(640, 480, 30);
                                          // xRes, yRes, FPS
    imageGen.setMapOutputMode(mapMode);
    imageGen.setPixelFormat(PixelFormat.RGB24);
    // set Mirror mode for all
    context.setGlobalMirror(true);

    context.startGeneratingAll();
    System.out.println("Started context generating...");

    ImageMetaData imageMD = imageGen.getMetaData();
    imWidth = imageMD.getFullXRes();
    imHeight = imageMD.getFullYRes();
  }
  catch (Exception e) {
    System.out.println(e);
    System.exit(1);
  }
}  // end of configOpenNI()
```

Unlike for depth generation where I had to use a DepthMetaData object to obtain a map, the camera image can be obtained directly from the ImageGenerator object.

9.1. Updating the Image

ViewerPanel still spawns a thread as in earlier versions, and the run() method is almost unchanged:

```
public void run()
{
  isRunning = true;
  while (isRunning) {
    try {
      // context.waitAnyUpdateAll();
      context.waitOneUpdateAll(imageGen);
                  // wait for 'one' ('any' is safer)
    }
    catch(StatusException e)
    {  System.out.println(e);
       System.exit(1);
    }
    long startTime = System.currentTimeMillis();
    updateImage();
    imageCount++;
    totalTime += (System.currentTimeMillis() - startTime);
    repaint();
  }

  // close down
  try {
    context.stopGeneratingAll();
  }
  catch (StatusException e) {}
  context.release();
  System.exit(1);
}  // end of run()
```

There are two changes. I use Context.waitOneUpdateAll() to wait for a node update, although it is safer to stick with Context.waitAnyUpdateAll() in most programs, and call updateImage() to create a BufferedImage.

```
// globals
private ImageGenerator imageGen;
private BufferedImage image;


private void updateImage()
// get image data as bytes; convert to an image
{
  try {
    ByteBuffer imageBB = imageGen.getImageMap().createByteBuffer();
    image = bufToImage(imageBB);
  }
  catch (GeneralException e) {
    System.out.println(e);
  }
}  // end of updateImage()
```

bufToImage() maps the sequence of bytes in the imageBB ByteBuffer into a two-dimensional array of integers, representing the image's pixels.

The final image will be of type BufferedImage.TYPE_INT_ARGB, which stores four bytes (samples) per integer (pixel). The bytes are arranged as shown in Figure 18.

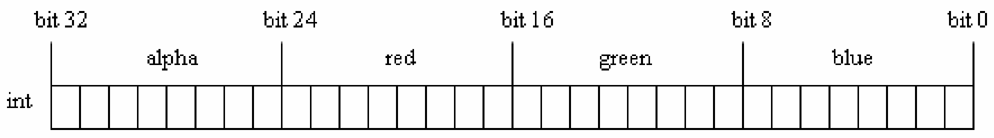


Figure 18. A TYPE_INT_ARGB Pixel.

The format packs 8-bit alpha, red, green, and blue samples into the 32-bit integer.

The bufToImage() code:

```
private BufferedImage bufToImage(ByteBuffer pixelsRGB)
/* Transform the ByteBuffer of pixel data into a BufferedImage
   Converts RGB bytes to ARGB ints with no transparency.
*/
{
  int[] pixelInts = new int[imWidth * imHeight];
  int rowStart = 0;
      // rowStart will index the first byte in each row;
      // start with first row, and moves across and down

  int bbIdx;               // index into ByteBuffer
  int i = 0;               // index into pixels int[]
  int rowLen = imWidth * 3;    // number of bytes in each row
  for (int row = 0; row < imHeight; row++) {
    bbIdx = rowStart;
    for (int col = 0; col < imWidth; col++) {
      int pixR = pixelsRGB.get( bbIdx++ );
      int pixG = pixelsRGB.get( bbIdx++ );
      int pixB = pixelsRGB.get( bbIdx++ );
      pixelInts[i++] =     // A R G B format; A not used (FF)
         0xFF000000 | ((pixR & 0xFF) << 16) |
         ((pixG & 0xFF) << 8) | (pixB & 0xFF);
    }
    rowStart += rowLen;   // move to next row
  }

  // create a BufferedImage from the pixelInts[] data
  BufferedImage im =  new BufferedImage(imWidth, imHeight,
                                BufferedImage.TYPE_INT_ARGB);
  im.setRGB(0, 0, imWidth, imHeight, pixelInts, 0, imWidth);
  return im;
}  // end of bufToImage()
```

The 2D pixelInts[] array is filled a row at a time from pixelsRGB. For each integer in the array, three values are read from pixelsRGB, and then converted into the red, green, and blue bytes in the integer. The alpha byte is set to 0xFF which means that the color is not transparent.

Once the integer array, pixelInts[], has been populated, it is used as the data buffer for a BufferedImage. The image's sample and color models are derived from TYPE_INT_ARGB (whereas the depth image used TYPE_BYTE_GRAY).

9.2. Drawing the Image

Once the hard work of buffer-to-BufferedImage conversion has been done by bufToImage(), paintComponent() only has to draw the image.

```
// globals
private BufferedImage image;


public void paintComponent(Graphics g)
// Draw the camera image and statistics info

  super.paintComponent(g);
  Graphics2D g2 = (Graphics2D) g;

  if (image != null)
    g2.drawImage(image, 0, 0, this);

  writeStats(g2);
} // end of paintComponent()
```

10. Version 4: Infrared (IR) Map

This version of ViewerPanel displays the Kinect's IR image, as illustrated in Figure 19.

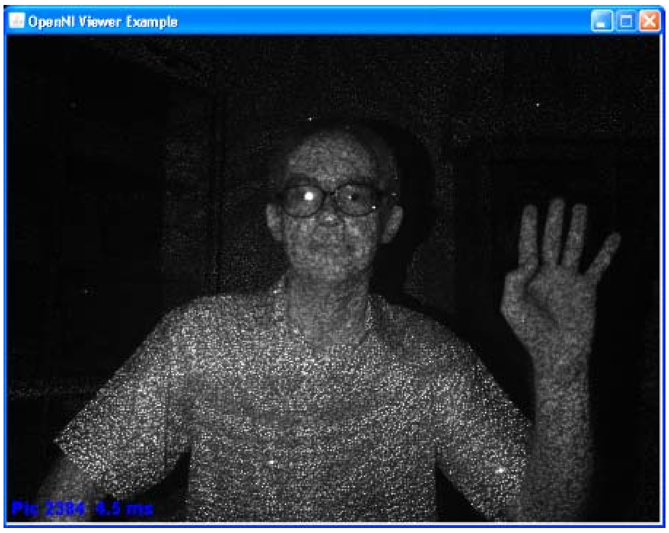


Figure 19. Viewing an IR Map.

The IR light source is a laser that projects an irregular pattern of dots. The monochrome CMOS IR sensor measures the dots' “time of flight” after they reflect off objects in the scene. In the same way as sonar, knowing how long the light takes to return, the Kinect can calculate how far away an object is located. IR is used to reduce the problem of ambient light affecting the calculations.

The Kinect encodes patterns in the IR light, which are deformed when they reflect back. The detected changes help the Kinect generate finer depth graduations.

The overall screen pattern (which is easy to see when OpenNIViewer is running) is a 3x3 grid, with each grid cell containing many, many little dots, with a brighter reference dot at its middle. Each grid cell has a slightly different average grayscale level compared to its neighbors.

The ViewerPanel constructor calls configureNITE() to create a Context object, and adds an IRGenerator production node to the context.

```
// globals
private Context context;
private IRGenerator irGen;
private int imWidth, imHeight;


private void configOpenNI()
{
  try {
    context = new Context();

    // add the NITE License
    License license = new License("PrimeSense",
                  "0KOIk2JeIBYClPWVnMoRKn5cdY4=");   // vendor, key
    context.addLicense(license);

    irGen = IRGenerator.create(context);

    MapOutputMode mapMode = new MapOutputMode(640, 480, 30);
                                        // xRes, yRes, FPS
    irGen.setMapOutputMode(mapMode);
    // set Mirror mode for all
    context.setGlobalMirror(true);

    context.startGeneratingAll();
    System.out.println("Started context generating...");

    IRMetaData irMD = irGen.getMetaData();
    imWidth = irMD.getFullXRes();
    imHeight = irMD.getFullYRes();
  }
  catch (Exception e) {
    System.out.println(e);
    System.exit(1);
  }
}  // end of configOpenNI()
```

10.1. Updating the Image

The run() method in ViewerPanel is unchanged from previous examples, except for calling updateIRImage() to update the image.

```
// global
private IRGenerator irGen;
private BufferedImage image = null;

private void updateIRImage()
{
  try {
    ShortBuffer irSB = irGen.getIRMap().createShortBuffer();

    // scan the IR data, storing the min and max values
    int minIR = irSB.get();
    int maxIR = minIR;
    while (irSB.remaining() > 0) {
      int irVal = irSB.get();
      if (irVal > maxIR)
        maxIR = irVal;
      if (irVal < minIR)
        minIR = irVal;
    }
    irSB.rewind();

    // convert the IR values into 8-bit grayscales
    image = createGrayIm(irSB, minIR, maxIR);
  }
  catch(GeneralException e)
  {  System.out.println(e);  }
}  // end of updateIRImage()
```

Before the IR buffer is converted into a grayscale image, its smallest and largest values are recorded. In my test environment, the IR ranged from 0 to around 1020. This range is utilized in createGrayIm() to 'scale' the IR data so it spans the grayscale range (0-255).

```
// globals
```

```
private static final int MIN_8_BIT = 0;
private static final int MAX_8_BIT = 255;
       // for mapping the IR values into a 8-bit range


private BufferedImage createGrayIm(ShortBuffer irSB,
                                          int minIR, int maxIR)
{
  // create a grayscale image
  BufferedImage image = new BufferedImage(imWidth, imHeight,
                                      BufferedImage.TYPE_BYTE_GRAY);
  // access the image's data buffer
  byte[] data = ((DataBufferByte)
                  image.getRaster().getDataBuffer()).getData();

  float displayRatio = (float)
                    (MAX_8_BIT - MIN_8_BIT)/(maxIR - minIR);

  // scale the converted IR data over the grayscale range;
  int i = 0;
  while (irSB.remaining() > 0) {
    int irVal = irSB.get();
    int out;
    if (irVal <= minIR)
      out = MIN_8_BIT;
    else if (irVal >= maxIR)
      out = MAX_8_BIT;
    else
      out = (int) ((irVal - minIR)* displayRatio);
    data[i++] = (byte) out;   // store in the data buffer
  }
  return image;
}  // end of createGrayIm()
```

The sample and color models in the image are derived from TYPE_BYTE_GRAY, in the same way as the depth image in versions 1 and 2.

A Flickering Bug

During testing, the IR image would sometimes flicker excessively. This problem would disappear after I ran an application that uses the Kinect's camera, such as version 3 of ViewerPanel or OpenNI's NiViewer example.

11. Version 5: A Colored Depth Map.

This version of ViewerPanel displays the Kinect's depth map once again, but this time in glorious Technicolor, as illustrated by Figure 20.

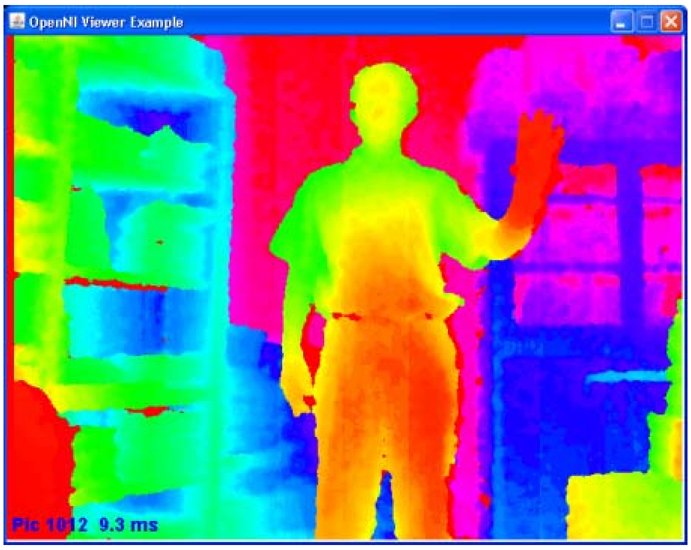


Figure 20. Viewing a Colored Depth Map.

The trick to coding this version of ViewerPanel is *not* to modify the data, which still uses one byte to represent a pixel, but to change the BufferedImage's color model. I use the IndexColorModel class which treats each byte as an index (between 0 and 255) into an array containing RGB colors (a so-called colormap).

Building a colormap containing 256 colors is quite a chore, but the ColorUtils library (http://code.google.com/p/colorutils/) can generate several predefined maps. Figure 20 employs the colormap returned by ColorUtils.getSpectrum(), which colorizes the white to black range using the red-to-violet spectrum. This means that objects in the depth map close to the Kinect will be rendered red or orange, while those further away will be green, blue, or violet. Those without depth measures (perhaps because they are too close to the Kinect) will be rendered as red.

It's worth mentioning that this approach to colorizing the depth map can be used on the IR map, which also uses one byte per pixel.

11.1. Creating a Color Model

The IndexColorModel object is created with genColorModel().

```
// globals
private static final int NUM_COLORS = 256;

private IndexColorModel colModel;    // used for colorizing


private IndexColorModel genColorModel()
// generate a index color model for the BufferedImage
{
  byte[] r = new byte[NUM_COLORS];    // reds
  byte[] g = new byte[NUM_COLORS];    // greens
  byte[] b = new byte[NUM_COLORS];    // blues

  Color[] cs = ColorUtils.getSpectrum(NUM_COLORS, 0, 1);
             // colors will range from red to violet
  for (int i=0; i < NUM_COLORS; i++) {
    r[i] = (byte) cs[i].getRed();
    g[i] = (byte) cs[i].getGreen();
    b[i] = (byte) cs[i].getBlue();
  }
  return new IndexColorModel(8, NUM_COLORS, r, g, b);
                // 8 = no of bits in pixel
}  // end of genColorModel()
```

The IndexColorModel object needs to be initialized with the bit-size of a pixel, and arrays of red, green, and blue components for the RGB colors in the model. These arrays are filled by accessing the Color array returned by ColorUtils.getSpectrum().

genColorModel() is called in ViewerPanel's constructor, where the model is assigned to a global variable.

```
// global
private IndexColorModel colModel;

// in ViewerPanel()
colModel = genColorModel();
```

Other ColorUtils methods can return colormaps with restricted spectra, such as:

```
Color[] cs = ColorUtils.getRedtoBlueSpectrum(NUM_COLORS);

Color[] cs = ColorUtils.getRedtoYellowSpectrum(NUM_COLORS);

Color[] cs = ColorUtils.getColorsBetween(Color.RED,
                                     Color.GREEN, NUM_COLORS);
```

11.2. Creating the Depth Map

Since the depth data still uses 8-bit pixels, none of the code for filling histogram[], and using it to initialize imgbytes[], needs to be changed from version 1 of ViewerPanel:

```
// globals
private byte[] imgbytes;
private float histogram[];

private void updateDepthImage()
/* build a new histogram of 8-bit depth values, and convert it to
   image pixels (stored as bytes); unchanged from version 1 */
{
  ShortBuffer depth = depthMD.getData().createShortBuffer();
  calcHistogram(depth);
  depth.rewind();

  while (depth.remaining() > 0) {
    int pos = depth.position();
    short depth = depth.get();
    imgbytes[pos] = (byte) histogram[depth];
  }
}  // end of updateDepthImage()
```

11.3. Rendering the Image

paintComponent() builds a BufferedImage from the current image data in imgbytes[] and the indexed color model, and then displays it.

```
// globals
private byte[] imgbytes;
private IndexColorModel colModel;


public void paintComponent(Graphics g)
// Draw the colorized depth image and statistics info
{
  super.paintComponent(g);
  Graphics2D g2 = (Graphics2D) g;

  BufferedImage image = createColorImage(imgbytes, colModel);
  if (image != null)
    g2.drawImage(image, 0, 0, this);

  writeStats(g2);
} // end of paintComponent()
```

createColorImage() uses the color model to create a compatible raster (see Figure 16 to recall the components of a BufferedImage). The raster's data buffer is exposed, filled with the data, and the resulting BufferedImage is returned.

```
private BufferedImage createColorImage(byte[] imgbytes,
                                    IndexColorModel colModel)
/* Create BufferedImage using the image data
   and indexed color model */
{
  // create suitable raster that matches the color model
  WritableRaster wr =
      colModel.createCompatibleWritableRaster(imWidth, imHeight);

  // copy image data into data buffer of raster
  DataBufferByte dataBuffer = (DataBufferByte) wr.getDataBuffer();
  byte[] destPixels = dataBuffer.getData();
  System.arraycopy(imgbytes, 0, destPixels, 0, destPixels.length);

  // create BufferedImage from color model and raster
  return new BufferedImage(colModel, wr, false, null);
}  // end of createColorImage()
```

12. Version 6: Combined Depth and Image Maps

This version of ViewerPanel combines the Kinect's image map and a colorized depth map, as illustrated by Figure 21.

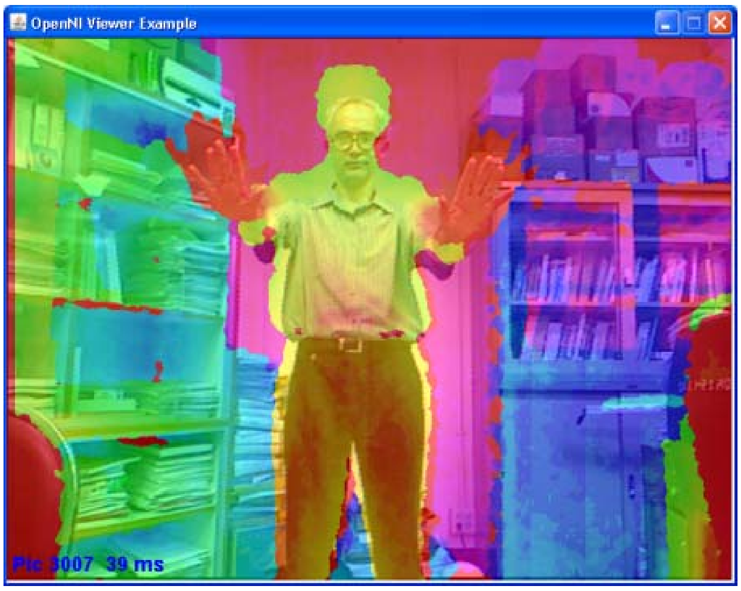


Figure 21. Combined Depth and Image Maps.

The program is essentially a merging of the depth coloring code from version 5 of ViewerPanel, with the webcam code from version 3. The coloring is slightly more complicated since it must include an alpha channel so that the webcam image can be seen behind the colored depth image.

Figure 21 highlights a problem with combining the images – they don't match up! This is due to the Kinect utilizing physically separated CMOS sensors to produce the depth and image maps.

OpenNI includes an AlternativeViewPointCapability API to adjust the depth generator's viewpoint so it aligns with the image generator. Unfortunately, the Java API in my version of OpenNI doesn't work. There is an AlternativeViewPointCapability class in the API, but the methods I need aren't defined as public, and there's no get method for accessing the AlternativeViewPointCapability object in the depth generator.

Hopefully, these issues will be fixed in the OpenNI version you're using, so I'll describe some code snippets for using AlternativeViewPointCapability, based on my extrapolating from similar C/C++ fragments.

My non-aligned combination can be compared with an aligned version by running OpenNI's NiViewer example. Pressing 8 displays an aligned depth/camera image, like that shown in Figure 22.

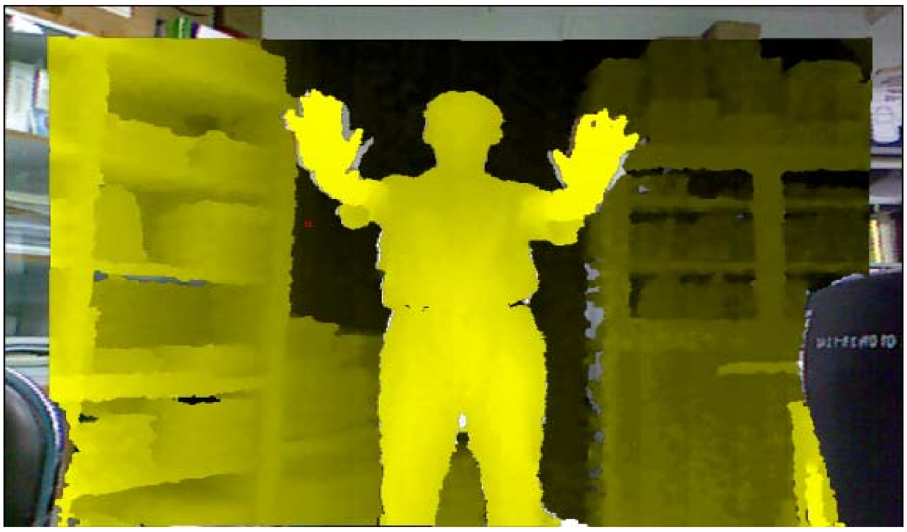


Figure 22. NiViewer's Combined Depth and Image Maps.

The depth image appears to be scaled down around the midpoint of the screen.

12.1. Creating a Color Model

The IndexColorModel object is created with a modified version of genColorModel() from version 5 which adds an alpha channel to the indexed colors.

```
// globals
private static final int NUM_COLORS = 256;
private IndexColorModel colModel;     // used for colorizing


private IndexColorModel genColorModel()
{
  byte[] r = new byte[NUM_COLORS];    // reds
  byte[] g = new byte[NUM_COLORS];    // greens
  byte[] b = new byte[NUM_COLORS];    // blues
  byte[] a = new byte[NUM_COLORS];    // alphas

  Color[] cs = ColorUtils.getSpectrum(NUM_COLORS, 0, 1);
             // colors will range from red to blue

  for (int i=0; i < NUM_COLORS; i++) {
    r[i] = (byte) cs[i].getRed();
    g[i] = (byte) cs[i].getGreen();
    b[i] = (byte) cs[i].getBlue();
    a[i] = (byte) 0x88;     // translucency
  }
  return new IndexColorModel(8, NUM_COLORS, r, g, b, a);
}  // end of genColorModel()
```

The value for the transparency setting (0x88) was chosen at random, and is the same for all the colors in the model.

12.2. Initializing the Context

The context must be assigned both a depth generator and image generator. As in previous depth generation examples, I'll use a DepthMetaData object to obtain the depth map.

```
// globals
private Context context;
private DepthMetaData depthMD;
private ImageGenerator imageGen;


private void configOpenNI()
{
  try {
    context = new Context();

    // add the NITE License
    License license = new License("PrimeSense",
                    "0KOIk2JeIBYClPWVnMoRKn5cdY4=");   // vendor, key
    context.addLicense(license);

    DepthGenerator depthGen = DepthGenerator.create(context);
    imageGen = ImageGenerator.create(context);

    MapOutputMode mapMode = new MapOutputMode(640, 480, 30);
                                      // xRes, yRes, FPS
    depthGen.setMapOutputMode(mapMode);
    imageGen.setMapOutputMode(mapMode);
    imageGen.setPixelFormat(PixelFormat.RGB24);

    // set Mirror mode for all
    context.setGlobalMirror(true);

    context.startGeneratingAll();
    System.out.println("Started context generating...");

    depthMD = depthGen.getMetaData();
         // use depth metadata to access depth info
         // (avoids bug with DepthGenerator)
  }
  catch (Exception e) {
    System.out.println(e);
    System.exit(1);
  }
}  // end of configOpenNI()
```

The depth image should be aligned with the camera image after the generators (depthGen and imageGen) have been created. One C/C++ approach, translated to Java, is:

```
AlternativeViewPointCapability altViewCap =
               depthGen.getAlternativeViewpointCapability();
altViewCap.setViewpoint(imageGen);
boolean isSet = altViewCap.isViewpointAs(imageGen);  //check result
```

The problem is that the Java DepthGenerator class doesn't have a getAlternativeViewpointCapability() method to return its AlternativeViewPointCapability object. Also, both setViewpoint() and isViewpointAs() are declared private in AlternativeViewPointCapability.

There may be an alternative way to create the AlternativeViewPointCapability object, by using:

```
AlternativeViewPointCapability altViewCap =
                 new AlternativeViewPointCapability(depthGen);
```

This doesn't raise any errors at compile time, but it's still not possible to call setViewpoint() and isViewpointAs().

The use of any OpenNI capability should be preceded by a test to see whether the hardware supports that feature. For the alternative viewpoint, this test is:

```
boolean hasAltView =
            depthGen.isCapabilitySupported("AlternativeViewPoint");
```

This code correctly returns true, which suggests that the problems with the Java methods are just an oversight in this version of the API.

12.3. Updating the Panel

The run() method calls two update() methods, one for the camera image and one for the depth image. In addition, it uses Context.waitAndUpdateAll() to wait for *both* production nodes to generate new data before updating them.

```
// globals
private int imageCount = 0;
private long totalTime = 0;
private Context context;
private volatile boolean isRunning;


public void run()
{
  isRunning = true;
  while (isRunning) {
    try {
      context.waitAndUpdateAll();
      // wait for all nodes to have new data, then update them
    }
    catch(StatusException e)
    {  System.out.println(e);
       System.exit(1);
    }
    long startTime = System.currentTimeMillis();
    updateImage();
    updateDepthImage();
    imageCount++;
    totalTime += (System.currentTimeMillis() - startTime);
    repaint();
  }
  // close down
  try {
    context.stopGeneratingAll();
  }
  catch (StatusException e) {}
  context.release();
  System.exit(0);
}  // end of run()
```

The update methods are almost unchanged from earlier versions. updateImage() does the same task as the same-named method in version 3: it extracts the image data from the map as an array of bytes, and converts it to a BufferedImage. updateDepthImage() is identical to earlier versions (e.g. in version 1), storing the depth image data in the imgbytes[] global.

12.4. Rendering the Images

paintComponent() must draw two images, first the camera image and then the translucent depth image over the top of it.

```
// globals
private byte[] imgbytes;
private BufferedImage cameraImage = null;
private IndexColorModel colModel;     // used for colorizing


public void paintComponent(Graphics g)
{
  super.paintComponent(g);
  Graphics2D g2 = (Graphics2D) g;

  // draw the camera image first
  if (cameraImage != null)
    g2.drawImage(cameraImage, 0, 0, this);

  // draw the translucent depth image
  BufferedImage depthImage = createColorImage(imgbytes, colModel);
  if (depthImage != null)
    g2.drawImage(depthImage, 0, 0, this);

  writeStats(g2);
} // end of paintComponent()
```

createColorImage() is identical to the same method in version 5 – it creates a BufferedImage using the image data and indexed color model. Since the model's colors all have alpha channels (see genColorModel() above), then the resulting image will be translucent.