Kinect Chapter 15. Using the Kinect's Microphone Array

The previous chapter was about speech recognition, with audio recorded from the PC's microphone rather than the Kinect. Frankly, a bit of a cheat, but still useful.

This chapter shows how to capture sound from the Kinect's microphone array, so there's no longer any need to feel guilty about using an extra microphone. The trick is to install audio support from Microsoft's Kinect SDK which lets Windows 7 treat the array as a standard multichannel recording device. Care must be taken when mixing this audio driver with OpenNI, but the payoff is that the microphone array becomes visible to Java's sound API. The main drawback is that this only works on Windows 7 since Microsoft's SDK only supports that OS.

My Java examples start with several tools for listing audio sources and their capabilities, such as the PC's microphone and the Kinect array. I'll also describe a simple audio recorder that reads from a specified source and saves the captured sound to a WAV file.

One of the novel features of Microsoft's SDK is support for *beamforming* – the ability to calculate the direction of an audio source. It's possible to duplicate this using the Java SoundLocalizer application developed by Laurent Calmes (http://www.laurentcalmes.lu).

I finish with another version of my audio-controlled Breakout game, this time employing the Kinect's microphone array instead of the PC's mike.

1. Installing the "Microsoft SDK for Kinect" Audio Driver

My Windows 7 test machine started with all three PrimeSense drivers listed in the Device Manager control panel. I uninstalled *and deleted* the PrimeSense Audio driver, leaving just the camera and motor drivers (see Figure 1).

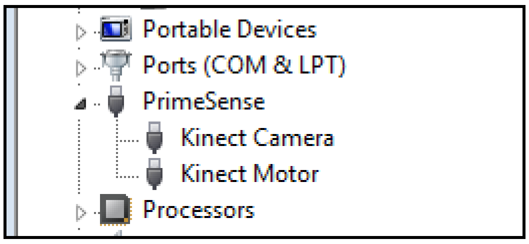


Figure 1. The PrimeSense Drivers without Audio.

Note that only uninstalling the driver isn't usually enough, since Windows has a habit of automatically trying to reinstall drivers for unrecognized hardware. The easiest way to avoid this irritating behavior is to specify that the driver be deleted when you request its de-installation. This appears as an option in the uninstallation dialog box.

I downloaded the latest version of Microsoft's SDK from http://www.microsoft.com/en-us/kinectforwindows/develop/overview.aspx, and ran its installer (making sure the Kinect was unplugged from my PC). Afterwards, I plugged the Kinect back in, and the Device Manager showed a number of changes (see Figure 2).

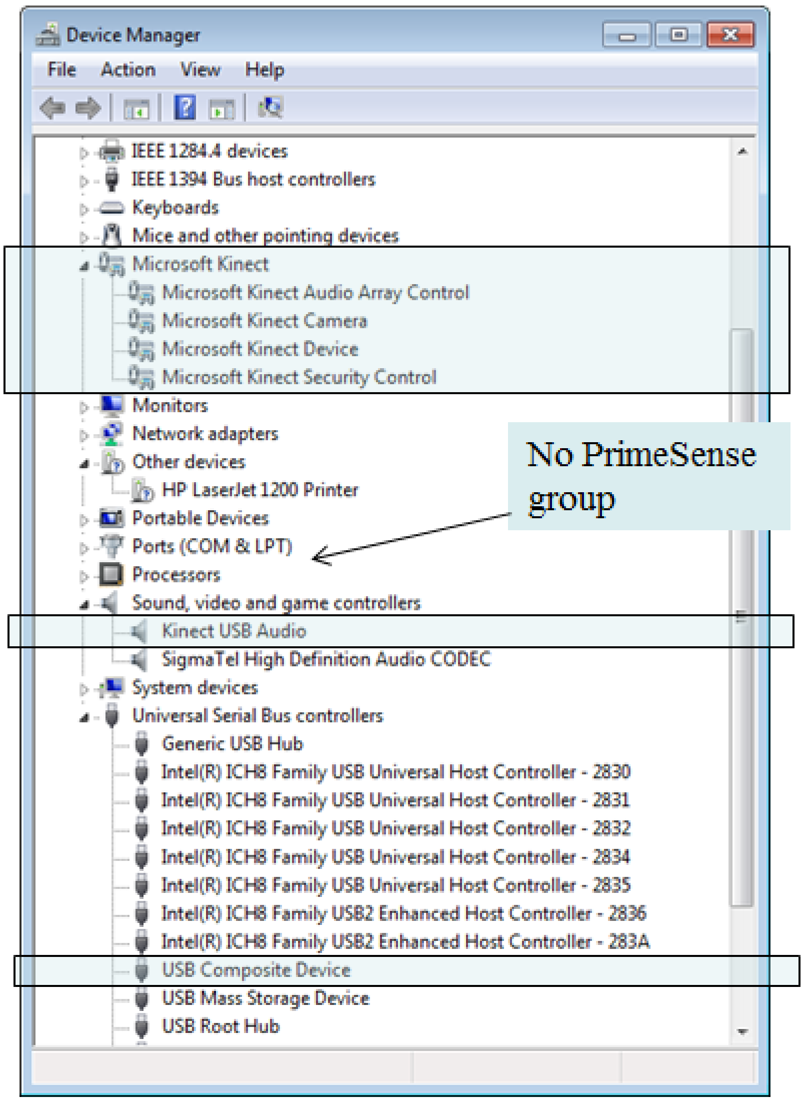
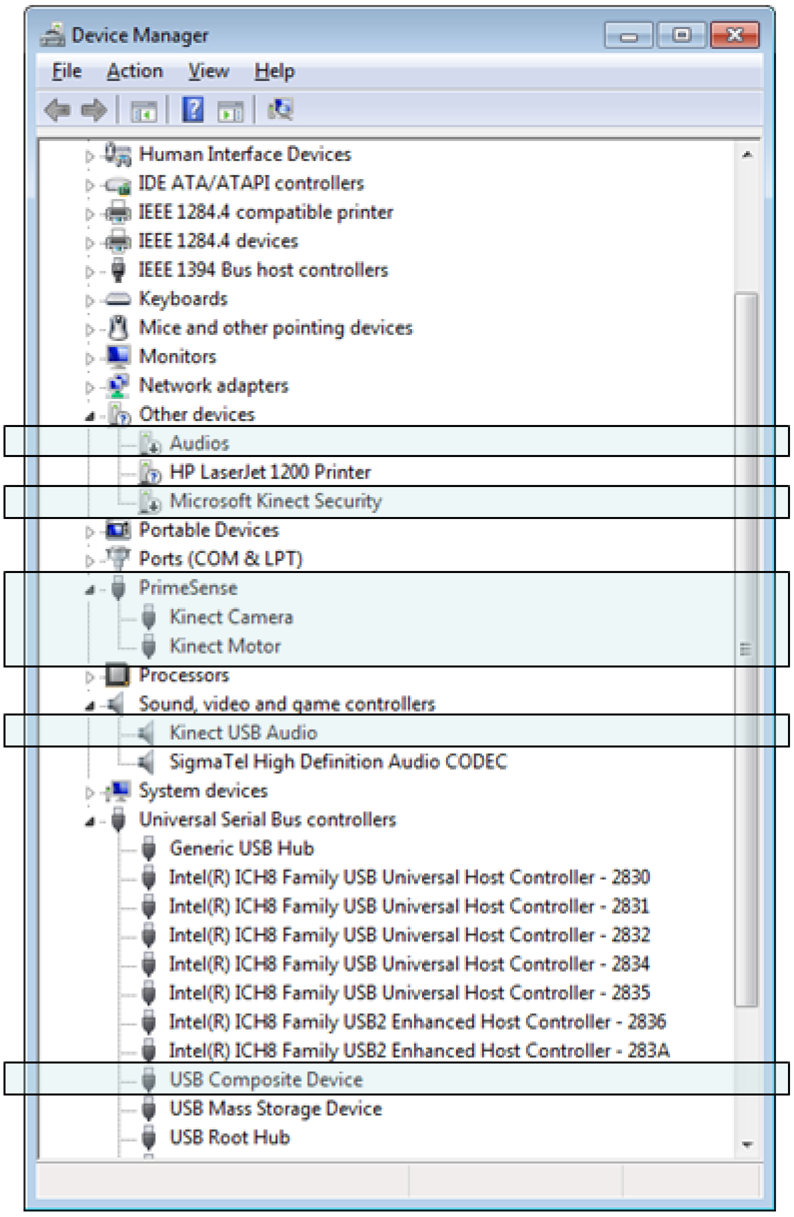

Device Manager
File Action View Help
IEEE 1284.4 devices
IEEE 1394 Bus host controllers
Keyboards
Mice and other pointing devices
Microsoft Kinect
Microsoft Kinect Audio Array Control
Microsoft Kinect Camera
Microsoft Kinect Device
Microsoft Kinect Security Control
Monitors
Network adapters
Other devices
HP LaserJet 1200 Printer
Portable Devices
Ports (COM & LPT)
Processors
Sound, video and game controllers
Kinect USB Audio
SigmaTel High Definition Audio CODEC
System devices
Universal Serial Bus controllers
Generic USB Hub
Intel(R) ICH8 Family USB Universal Host Controller - 2830
Intel(R) ICH8 Family USB Universal Host Controller - 2831
Intel(R) ICH8 Family USB Universal Host Controller - 2832
Intel(R) ICH8 Family USB Universal Host Controller - 2834
Intel(R) ICH8 Family USB Universal Host Controller - 2835
Intel(R) ICH8 Family USB2 Enhanced Host Controller - 2836
Intel(R) ICH8 Family USB2 Enhanced Host Controller - 283A
USB Composite Device
USB Mass Storage Device
USB Root Hub
No PrimeSense group

Figure 2. The Device Manager After Installing the Microsoft SDK.

The Microsoft SDK consists of four drivers under the "Microsoft Kinect" heading, a "Kinect USB Audio" sound driver, and a "USB Composite Device". Also, the PrimeSense group has disappeared.

I need the SDK's sound driver and the USB driver, but can uninstall and delete the four drivers in the Microsoft Kinect group. The uninstallation dialog window for each driver in the group will include an option box to delete the driver – make sure it's selected every time, otherwise the OS will obsessively keep trying to reinstall the drivers.

Once the four drivers have been deleted, unplug and replug the Kinect into your PC. Several changes occur in the Device Manager, as shown in Figure 3.



```
Human Interface Devices
IDE ATA/ATAPI controllers
IEEE 1284.4 compatible printer
IEEE 1284.4 devices
IEEE 1394 Bus host controllers
Keyboards
Mice and other pointing devices
Monitors
Network adapters
Other devices
    Audios
    HP LaserJet 1200 Printer
    Microsoft Kinect Security
Portable Devices
Ports (COM & LPT)
PrimeSense
    Kinect Camera
    Kinect Motor
Processors
Sound, video and game controllers
    Kinect USB Audio
    SigmaTel High Definition Audio CODEC
System devices
Universal Serial Bus controllers
    Generic USB Hub
    Intel(R) ICH8 Family USB Universal Host Controller - 2830
    Intel(R) ICH8 Family USB Universal Host Controller - 2831
    Intel(R) ICH8 Family USB Universal Host Controller - 2832
    Intel(R) ICH8 Family USB Universal Host Controller - 2834
    Intel(R) ICH8 Family USB Universal Host Controller - 2835
    Intel(R) ICH8 Family USB2 Enhanced Host Controller - 2836
    Intel(R) ICH8 Family USB2 Enhanced Host Controller - 283A
    USB Composite Device
    USB Mass Storage Device
    USB Root Hub
```

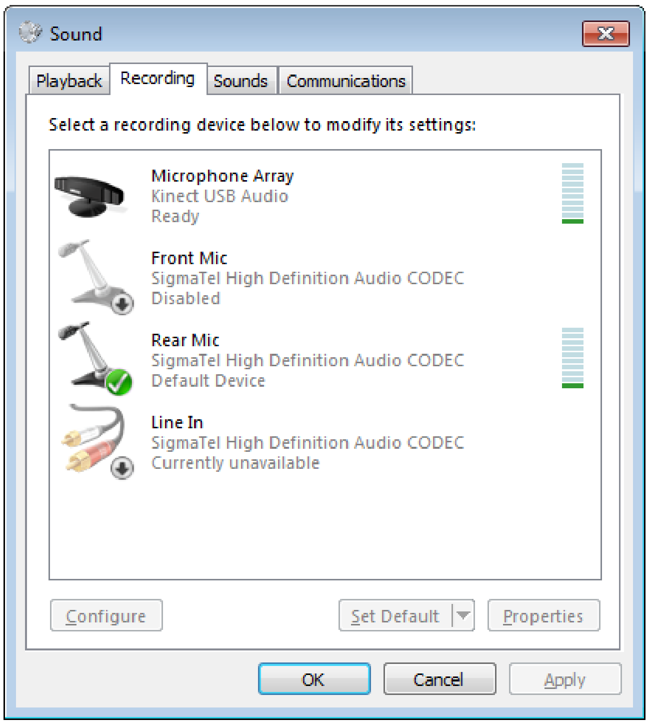
Figure 3. The Device Manager After Deleting the Microsoft Kinect Group.

The PrimeSense group reappears in the Device Manager list, and "Audios" and "Microsoft Kinect Security" entries appear under the "Other devices" heading, both showing errors. Both drivers should be disabled by right clicking on them, but don't delete them.

We've finished the installation of the Kinect array audio driver; time for testing.

2. Testing the Audio Driver

The simplest way of testing the driver is via the Recording tab of the Control Panel's "Sound" control (see Figure 4).



Select a recording device below to modify its settings:

Microphone Array
Kinect USB Audio
Ready

Front Mic
SigmaTel High Definition Audio CODEC
Disabled

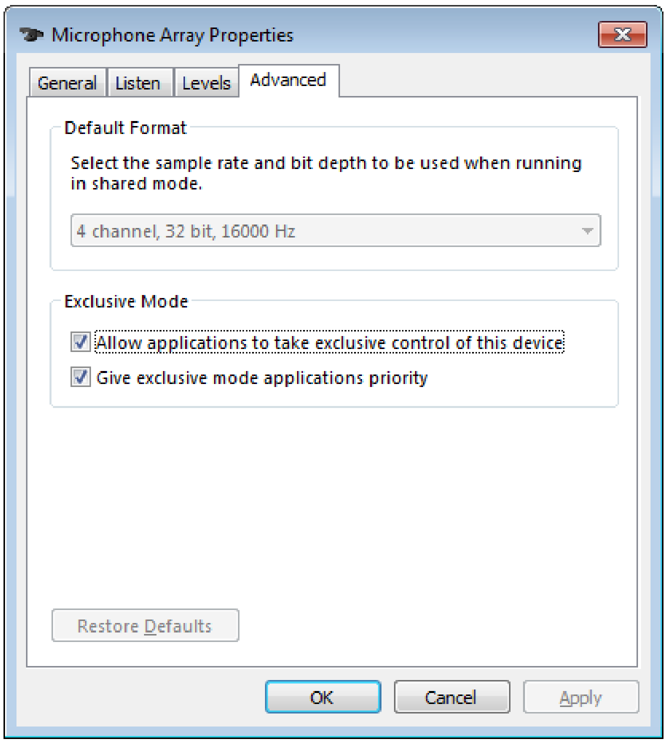
Rear Mic
SigmaTel High Definition Audio CODEC
Default Device

Line In
SigmaTel High Definition Audio CODEC
Currently unavailable

Figure 4. The Sound Control Panel Recording Tab.

Figure 4 should be compared to Figure 2 of the last chapter – a "Microphone Array" input device has joined the rear microphone and the other devices – the array's icon shows that it represents the Kinect. The dark green bars on the right of the array and Rear Mic rows mean that both devices are picking up sound. At this stage, it's a good idea to unplug the PC microphone, so only the Kinect is enabled.

Clicking on the "Microphone Array" row makes the dialog box's "Properties" button active. Clicking it leads to a Properties window with tabs for setting the volume and checking the array's format settings (as shown on the Advanced tab in Figure 5).


Microphone Array Properties

General | Listen | Levels | Advanced

Default Format

Select the sample rate and bit depth to be used when running in shared mode.

4 channel, 32 bit, 16000 Hz

Exclusive Mode

☑ Allow applications to take exclusive control of this device

☑ Give exclusive mode applications priority

Figure 5. The Microphone Array Input Format.

The format information states that the microphone array can supply four channels of 32 bit audio at 16 KHz. This isn't surprising since the Kinect array is made up of four microphones, exposed to the World in Figure 6.

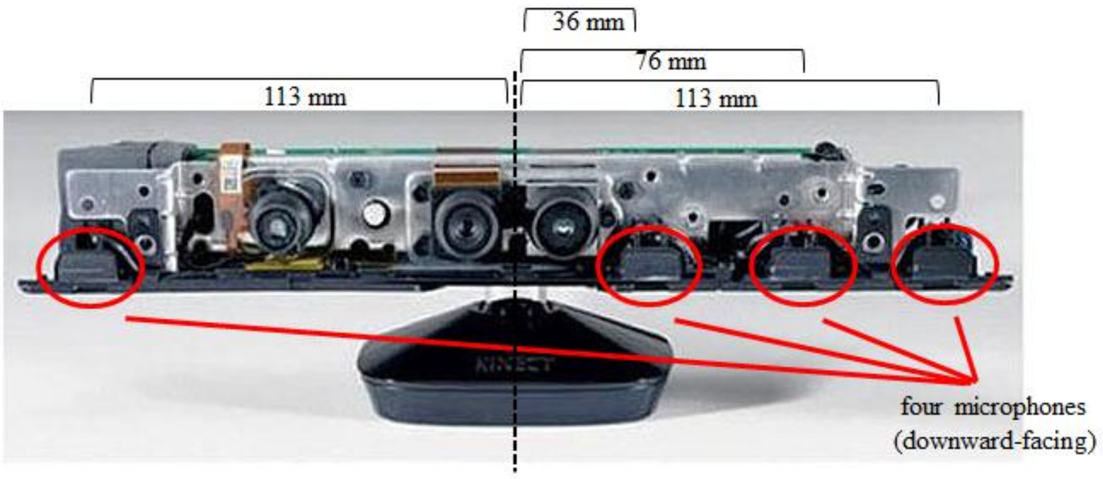
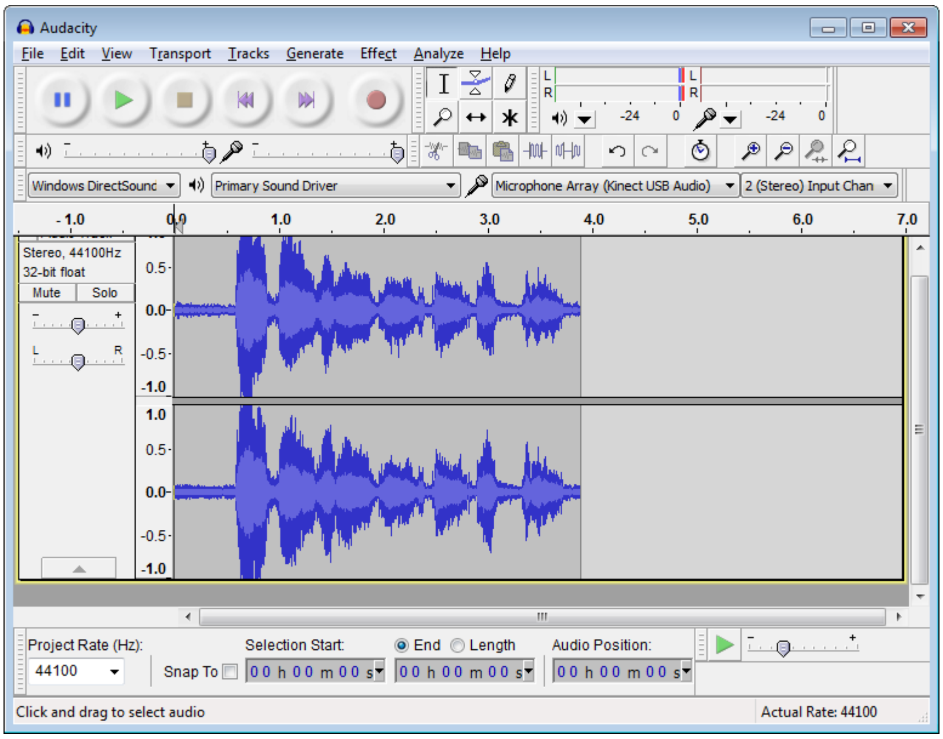


Figure 6. A Kinect Sensor Exposed!

The microphone are located along the Kinect's main axis, one 113 mm to the left of the base, three others to the right.

Unfortunately, if you've using a standard PC then there's some bad news. Most consumer-level sound cards only offer two-channel (stereo) Analogue to Digital conversion (ADC), which means that you'll only be able to record stereo from the microphone array. The easiest way to check is to record a piece of speech, and look at its visualization inside a audio editing tool, such as Audacity (http://audacity.sourceforge.net/). Figure 7 shows a short recording from the Kinect microphones.



Windows DirectSounc
Primary Sound Driver
Microphone Array (Kinect USB Audio)
2 (Stereo) Input Chan
Stereo, 44100Hz
32-bit float
Project Rate (Hz): 44100
00 h 00 m 00 s
00 h 00 m 00 s
00 h 00 m 00 s
Click and drag to select audio
Actual Rate: 44100

Figure 7. An Audacity Recording Using the Kinect.

Sound capture in Audacity is started by selecting from a list next to the microphone icon, and pressing record (the red circular button). I chose "Microphone Array", and was given the options of mono or stereo recording only. A close look at the recording in Figure 7 shows that the two channels are not identical.

An obvious question is which two channels are being recorded, since there are four microphones on the Kinect? Unfortunately, the answer appears to depend on the sound card, and I was unable to find any information for my antiquated SigmaTel board. However, the choice only matters when we utilize beamforming later.

3. Background on the Java Sound API

Java's sound API has been around since the early days of v1.1, when it developed a reputation for being confusingly low-level and somewhat bug-ridden. The latter problem went away with successive releases of JDK versions, with a major improvement in Java 1.5. However, the sound API remains low-level, mainly because it has to deal with such a variety of sound hardware and drivers, and support so many features, including audio playback, recording, manipulation, and MIDI synthesis.

Fortunately, I only need audio capture in this chapter, simplifying matters somewhat.

Java's AudioSystem class is the top-level entry point for the sound API, giving users access to Mixer objects. Java represents a sound card with many Mixer objects, each one corresponding to a functional component of the hardware and/or its drivers. For instance, there will be Mixer objects for the various forms of sound card input, different outputs, and for the hardware controls.

A Mixer acts as a container for simpler sound objects, created using the SourceDataLine, TargetDataLine, Clip and Port classes.

SourceDataLine represents an output stream for playing sounds, while a TargetDataLine is used to capture incoming audio. (These two classes have rather misleading names, in my opinion.)

A Clip object is an audio file loaded completely into memory, which can be played. I won't be using Clip in this chapter

A Port is a container for audio controls (e.g. for controlling gain, pan, reverb, sample rate). I'll divide ports into two types – input ports for recording-related controls (e.g. gain and sample rate) and output ports for the playback controls (e.g. volume).

3.1. Listing all the Mixers

My ListMixers application lists all the Mixer objects known to Java. For example, the output on my Win7 test machine is as follows:

```
> java ListMixers
Default mixer: Primary Sound Driver, version Unknown Version
No. mixers: 10

1. Name: Primary Sound Driver
   Description: Direct Audio Device: DirectSound Playback
   Lines: SourceDataLine; Clip;

2. Name: Speakers (SigmaTel High Definition Audio CODEC)
   Description: Direct Audio Device: DirectSound Playback
   Lines: SourceDataLine; Clip;

3. Name: Headphones (SigmaTel High Definition Audio CODEC)
   Description: Direct Audio Device: DirectSound Playback
   Lines: SourceDataLine; Clip;

4. Name: Primary Sound Capture Driver
   Description: Direct Audio Device: DirectSound Capture
   Lines: TargetDataLine;

5. Name: Rear Mic (SigmaTel High Definit
   Description: Direct Audio Device: DirectSound Capture
   Lines: TargetDataLine;

6. Name: Microphone Array (Kinect USB Au
   Description: Direct Audio Device: DirectSound Capture
   Lines: TargetDataLine;

7. Name: Port Speakers (SigmaTel High Definit
   Description: Port Mixer
   Lines: Output Port;

8. Name: Port Headphones (SigmaTel High Defin
   Description: Port Mixer
   Lines: Output Port;

9. Name: Port Rear Mic (SigmaTel High Definit
   Description: Port Mixer
   Lines: Input Port; Output Port;

10. Name: Port Microphone Array (Kinect USB Au
   Description: Port Mixer
   Lines: Input Port; Output Port;
```

There are 10 mixers, each with a name and description. The "Lines" line lists what sound objects the mixer contains. The Line class is inherited by every other sound class, and so "line" is often used to refer to the different sound objects in a mixer.

The easiest way to understand these mixers is to separate them into four groups based on their lines:

1. Mixers for playing output offer SourceDataLine and Clip lines. Mixers 1, 2, and 3 link to my machine's speakers and headphones.
2. A mixer for output controls contains an output Port. For example, mixer numbers 7 and 8.
3. Mixers for capturing input support TargetDataLine. Mixer numbers 4, 5, and 6 utilize the rear microphone and the Kinect array.
4. A mixer for input controls contains an input Port. Mixer numbers 9 and 10 both have an input Port (and a mysterious output Port, which I'll discuss shortly).

Java sound provides "Primary Sound" wrappers for the PC's sound card, which are present across all platforms. In the list above, these wrappers are called "Primary Sound Driver" and "Primary Sound Capture Driver" (mixer numbers 1 and 4). I won't be using these wrappers, preferring to access a mixer by its hardware/driver name (e.g. "SigmaTel" or "Kinect" in my case).

Two patterns appear when we look at the mixer names – each SourceDataLine mixer has a corresponding output port mixer, and each TargetDataLine mixer has an input port mixer. This means that if a user wants to change the control setting of a particular input or output line (e.g. increase the recording volume) that he needs to access the corresponding control in the associated input or output port.

This separation of lines and controls is standard Java Mixer design, although SourceDataLine and TargetDataLine objects can offer their own controls.

The Implementation of ListMixers

The top-level of ListMixers.java employs AudioSystem.getMixerInfo() to obtain information on all the Mixer objects in the system. Inside a loop, each Mixer object is investigated in more detail:

```
public static void main(String[] args)
{
  Mixer defaultMixer = AudioSystem.getMixer(null);
  if (defaultMixer == null) {
    System.out.println("Audio system unavailable");
    return;
  }
  else
    System.out.println("Default mixer: " +
                       defaultMixer.getMixerInfo());

  Mixer.Info[] mis = AudioSystem.getMixerInfo();
  System.out.println("No. mixers: " + mis.length);
  for (int i = 0; i < mis.length; i++) {
    Mixer mixer = AudioSystem.getMixer(mis[i]);
    System.out.println();
    AudioUtils.printInfo((i+1), mixer);
    printLines(mixer);
  }
}  // end of main()
```

The AudioUtils class is my library of useful sound methods; printInfo() prints the name and description associated with a mixer:

```
// in AudioUtils
public static void printInfo(int idNum, Mixer mixer)
{
  Mixer.Info mi = mixer.getMixerInfo();
  System.out.println("" + idNum + ". Name: " +  mi.getName() );
  System.out.println("   Description: " + mi.getDescription() );
}  // end of printInfo()
```

Back in ListMixers, the printLines() method reports on what sound objects (lines) are supported by a mixer.

```
private static void printLines(Mixer mixer)
// print the line types supported by this mixer
{
  String support = "";
  if (AudioUtils.isSourceDataLine(mixer))
    support += "SourceDataLine; ";

  if (AudioUtils.isClip(mixer))
    support += "Clip; ";

  if (AudioUtils.isTargetDataLine(mixer))
    support += "TargetDataLine; ";

  if (AudioUtils.isInputPort(mixer))
    support += "Input Port; ";

  if (AudioUtils.isOutputPort(mixer))
    support += "Output Port; ";

  System.out.println("   Lines: " + support);
}  // end of printLines()
```

Five AudioUtils methods test for different types of line:

```
// globals
private static Line.Info sdInfo =
                  new Line.Info(SourceDataLine.class);
private static Line.Info tdInfo =
                  new Line.Info(TargetDataLine.class);
private static Line.Info clipInfo = new Line.Info(Clip.class);
private static Line.Info portInfo = new Line.Info(Port.class);


public static boolean isSourceDataLine(Mixer mixer)
// used for playback of audio
{ return mixer.isLineSupported(sdInfo);  }


public static boolean isTargetDataLine(Mixer mixer)
// used to capture incoming audio
{ return mixer.isLineSupported(tdInfo);  }


public static boolean isClip(Mixer mixer)
{ return mixer.isLineSupported(clipInfo);  }


public static boolean isInputPort(Mixer mixer)
// a port on a source line is for recording controls
{
  Line.Info[] sli = mixer.getSourceLineInfo();
                             // sources for the mixer
  for (Line.Info lineInfo : sli) {
    if (lineInfo instanceof Port.Info)
      return true;
  }
  return false;
}  // end of isInputPort()


public static boolean isOutputPort(Mixer mixer)
// a port on a target line is for playback controls
{
  Line.Info[] tli = mixer.getTargetLineInfo();
                         // targets for the mixer
  for (Line.Info lineInfo : tli) {
    if (lineInfo instanceof Port.Info)
      return true;
  }
  return false;
}  // end of isOutputPort()
```

isInputPort() and isOutputPort() are slightly more complicated than the other methods in order to check whether a mixer port is "input" or "output". An input port is stored as a source line for the mixer (not to be confused with a SourceDataLine), and an output port is a target line for the mixer (not to be confused with a TargetDataLine).

3.2. Focusing on Audio Capture

One way to reduce the mixer output is to only list the mixers associated with audio capture, which is the purpose of the CaptureMixers.java example, shown executing below:

```
> java CaptureMixers
Is there microphone support? true
Is there a microphone source? true

Looking for Capture Mixers...

1. Name: Primary Sound Capture Driver
   Description: Direct Audio Device: DirectSound Capture
   Target line: interface TargetDataLine supporting 8 audio formats,
and buffers of at least 32 bytes

2. Name: Rear Mic (SigmaTel High Definit
   Description: Direct Audio Device: DirectSound Capture
   Target line: interface TargetDataLine supporting 8 audio formats,
and buffers of at least 32 bytes

3. Name: Microphone Array (Kinect USB Au
   Description: Direct Audio Device: DirectSound Capture
   Target line: interface TargetDataLine supporting 8 audio formats,
and buffers of at least 32 bytes

4. Name: Port Rear Mic (SigmaTel High Definit
   Description: Port Mixer
   Source line: MICROPHONE source port
   Target line: Master Volume target port

5. Name: Port Microphone Array (Kinect USB Au
   Description: Port Mixer
   Source line: MICROPHONE source port
   Target line: Master Volume target port
```

Only five capture-related mixers are shown, compared to the 10 mixers reported by ListMixers. However, the output includes more details on the TargetDataLine formats and the controls managed by the input ports. Both input port mixers contain an input port for controlling the microphone (as you'd expect), and an output port for master volume. The purpose of the output ports is quite mysterious since no audio is being output to the microphones.

CaptureMixers begins by using AudioSystem and the Port.Info classes to check for the presense of a microphone. It then reports all the capture-related mixers:

```
public static void main(String[] args)
{
  Mixer defaultMixer = AudioSystem.getMixer(null);
  if (defaultMixer == null) {
    System.out.println("Audio system unavailable");
    return;
  }

  System.out.println("Is there microphone support? " +
         AudioSystem.isLineSupported(Port.Info.MICROPHONE));
  System.out.println("Is there a microphone source? " +
         Port.Info.MICROPHONE.isSource());
  System.out.println();

  System.out.println("Looking for Capture Mixers...\n");
  int numCaptures = reportCaptures();
  if (numCaptures == 0)
    System.out.println("None found");
}  // end of main()
```

A capture mixer is one which supports either a TargetDataLine (for incoming audio) or supports an input port on a mixer source line (for recording controls).

```
private static int reportCaptures()
{
  int numCaptures = 0;
  for (Mixer.Info mi : AudioSystem.getMixerInfo()) {
    Mixer mixer = AudioSystem.getMixer(mi);
    if (AudioUtils.isTargetDataLine(mixer) ||
        AudioUtils.isInputPort(mixer)) {
      numCaptures++;
      printMixerInfo(numCaptures, mixer);
    }
  }
  return numCaptures;
}  // end of reportCaptures()
```

Mixers are selected using the AudioUtils.isTargetDataLine() and AudioUtils.isInputPort() methods described earlier.

printMixerInfo() prints the mixer's name, description, and its source and target line information.

```
private static void printMixerInfo(int idNum, Mixer mixer)
{
  AudioUtils.printInfo(idNum, mixer);
  for (Line.Info lineInfo : mixer.getSourceLineInfo())
    System.out.println("   Source line: " + lineInfo);
  for (Line.Info lineInfo : mixer.getTargetLineInfo())
    System.out.println("   Target line: " + lineInfo);
  System.out.println();
}  // end of printMixerInfo()
```

3.3. Naming a Capture Source

One way of further narrowing the capture information is to use the hardware/driver name of the mixer (i.e. on my machine "SigmaTel" or "Kinect") to filter out the "Primary Sound" and other drivers. This relies on the convention that a mixer is named after its hardware or driver, which seems to be standard practice.

My NamedCaptureMixers application prints information on the capture mixers whose names contains the string supplied on the command line. Its output when supplied with "kinect" is shown below.

```
> java NamedCaptureMixers kinect
Looking for capture mixers whose name contains: "kinect"

1. Name: Microphone Array (Kinect USB Au
   Description: Direct Audio Device: DirectSound Capture
   Target line: interface TargetDataLine supporting 8 audio formats,
and buffers of at least 32 bytes
   1: PCM_UNSIGNED unknown sample rate, 8 bit, mono, 1 bytes/frame,
   2: PCM_SIGNED unknown sample rate, 8 bit, mono, 1 bytes/frame,
   3: PCM_SIGNED unknown sample rate, 16 bit, mono, 2 bytes/frame,
little-endian
```

```
   4: PCM_SIGNED unknown sample rate, 16 bit, mono, 2 bytes/frame,
big-endian
   5: PCM_UNSIGNED unknown sample rate, 8 bit, stereo, 2 bytes/frame,
   6: PCM_SIGNED unknown sample rate, 8 bit, stereo, 2 bytes/frame,
   7: PCM_SIGNED unknown sample rate, 16 bit, stereo, 4 bytes/frame,
little-endian
   8: PCM_SIGNED unknown sample rate, 16 bit, stereo, 4 bytes/frame,
big-endian


2. Name: Port Microphone Array (Kinect USB Au
   Description: Port Mixer
   Source line: MICROPHONE source port
   1 Compound Control: Master Volume - values below
     1 Boolean Control: Mute Control with current value: false
     2 Float Control: Volume with current value: 0.8799878  (range:
0.0 - 1.0)

   Target line: Master Volume target port
   1 Boolean Control: Mute Control with current value: false
   2 Float Control: Volume with current value: 0.8799878  (range: 0.0
- 1.0)
   3 Compound Control: Master Volume - values below
     1 Boolean Control: Mute Control with current value: false
     2 Float Control: Volume with current value: 0.8799878  (range:
0.0 - 1.0)
```

As we'd expect from the earlier call to CaptureMixers, only two mixers are reported – the one holding the eight TargetDataLine formats for incoming audio streams, and the port mixer for the recording controls of the Kinect array.

The TargetDataLines information shows that only a maximum of two input channels from the Kinect are available to Java even though four channels are passed to the sound card (see Figure 5). In other words, my sound card can only process stereo or mono input. Also, no details are given on the supported sample rates, but we know from the Kinect's driver that 16 KHz is one of them (again see Figure 5).

NamedCaptureMixers lists all the controls managed by a port, along with their current settings. The microphone port has controls for mute and volume, and currently isn't muted with a high volume setting. These settings can be confirmed by going to the OS'es Sound control panel, and checking their properties tab, as in Figure 8.

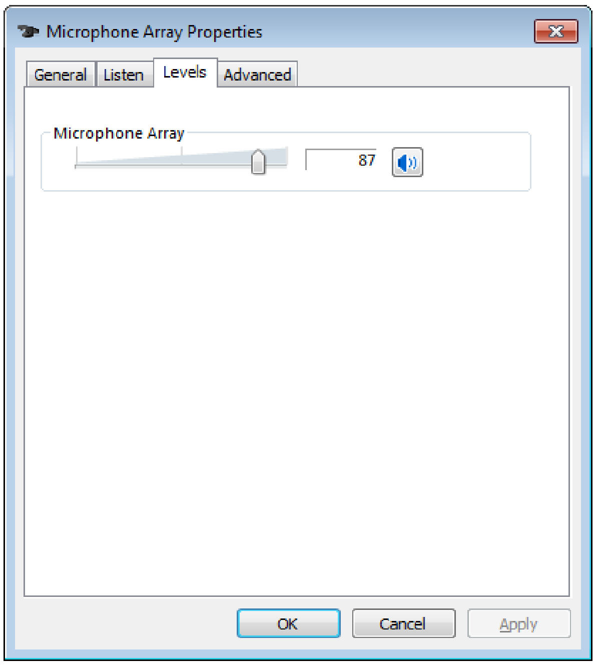


Figure 8. "Levels" Properties for the Kinect in the Sound Control Panel.

NamedCaptureMixers calls reportNamedCaptures() to loop through the mixers, selecting based on the mixer name and if AudioUtils.isTargetDataLine() or AudioUtils.isInputPort() returns true.

```
private static int reportNamedCaptures(String nm)
{
  int numCaptures = 0;
  for (Mixer.Info mi : AudioSystem.getMixerInfo()) {
    Mixer mixer = AudioSystem.getMixer(mi);
    if (AudioUtils.containsName(mixer, nm)) {
      if (AudioUtils.isTargetDataLine(mixer)) {
        numCaptures++;
        printTDLInfo(numCaptures, mixer);
      }
      else if (AudioUtils.isInputPort(mixer)) {
        numCaptures++;
        printInputPortInfo(numCaptures, mixer);
      }
    }
  }
  return numCaptures;
}  // end of reportNamedCaptures()
```

printTDLInfo() lists the supported audio formats for each TargetDataLine:

```
private static void printTDLInfo(int idNum, Mixer mixer)
{
  AudioUtils.printInfo(idNum, mixer);

  for (Line.Info info : mixer.getTargetLineInfo()) {
    System.out.println("    Target line: " + info);
    DataLine.Info di = (DataLine.Info) info;
    printAudioFormats(di.getFormats());
  }
  System.out.println();
}  // end of printTDLInfo()


private static void printAudioFormats(AudioFormat[] formats)
{
  for (int i = 0; i < formats.length; i++)
    System.out.println("    " + (i+1) + ": " + formats[i]);
  if (formats.length == 0)
    System.out.println("    [no formats]");
  System.out.println();
}  // end of printAudioFormats()
```

printInputPortInfo() lists all the supported controls on the mixer's source lines. At least one, and most likely all, of these lines will be input ports.

```
private static void printInputPortInfo(int idNum, Mixer mixer)
{
  AudioUtils.printInfo(idNum, mixer);
  for (Line.Info lineInfo : mixer.getSourceLineInfo()) {
    System.out.println("    Source line: " + lineInfo);
    accessControls(mixer, lineInfo);
  }
  for (Line.Info lineInfo : mixer.getTargetLineInfo()) {
    System.out.println("    Target line: " + lineInfo);
    accessControls(mixer, lineInfo);
  }
  System.out.println();
}  // end of printInputPortInfo()
```

accessControls() must open the port before its controls can be examined, then closes it afterwards.

```
private static void accessControls(Mixer mixer, Line.Info info)
{
  Line line = null;
  try {
    line = mixer.getLine(info);
    line.open();
    printControls(line.getControls());
  }
  catch (LineUnavailableException e) {
    System.out.println("Line could not be opened");
  }
  finally {
    if (line != null)
      line.close();
  }
}  // end of accessControls()
```

printControls() iterates through the array of Control objects, calling printControl() on each one:

```
private static void printControls(Control[] controls)
{
  for (int i = 0; i < controls.length; i++)
```

```
    printControl("   ", (i+1), controls[i]);
  if (controls.length == 0)
    System.out.println("  [no controls]");
  System.out.println();
}  // end of printControls()
```

printControl() is a little complicated since there are four types of Java control (boolean, compound, enum, and float) which must be examined in slightly different ways. In particular, compound and enum contain other controls, so printControl() must recursively examine their elements.

```
private static void printControl(String indent, int id,
                                            Control control)
{ String type = control.getType().toString();

  if (control instanceof BooleanControl) {
    BooleanControl ctrl = (BooleanControl) control;
    System.out.println(indent + id + " Boolean Control: " + ctrl);
  }
  else if (control instanceof CompoundControl) {
    CompoundControl ctrl = (CompoundControl) control;
    Control[] ctrls = ctrl.getMemberControls();
    System.out.println(indent + id + " Compound Control: " +
                                   type + " - values below");
      for (int i=0; i < ctrls.length; i++)
      printControl(indent + "  ", (i+1), ctrls[i]);
  }
  else if (control instanceof EnumControl) {
    EnumControl ctrl = (EnumControl) control;
    Object[] values = ctrl.getValues();
    Object value = ctrl.getValue();
    System.out.println(indent + id + " Enum Control: " +
                                   type + " - values below");
               // choose from a set of objects
    for (int i=0; i < values.length; i++) {
      if (values[i] instanceof Control)
        printControl(indent + "  ", (i+1), (Control) values[i]);
      else
        System.out.println(indent + "  Values[" + (i+1) + "]: " +
                 ((values[i] == value) ? "*" : "") + values[i]);
    }
  }
  else if (control instanceof FloatControl) {
    FloatControl ctrl = (FloatControl) control;
    System.out.println(indent + id + " Float Control: " + ctrl);
  }
  else
    System.out.println(indent + id + " Control: " + control);
}   // end of printControl()
```

4. Recording from a Microphone

My MikeRecorder application is started from the command line with the (partial) name of the required TargetDataLine mixer (e.g. "kinect", "sigmaTel").

If you're not sure about the mixer names on your machine, then they can be listed with CaptureMixers from section 3.2, "Focusing on Audio Capture". In my case, the rear microphone mixer is item 2 (with the name "Rear Mic (SigmaTel High Definit") and the Kinect array is item 3 (with the name "Microphone Array (Kinect USB Au").

If a line is found that contains the user's string, then a simple GUI application starts, as shown in Figure 9.

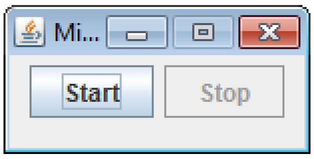
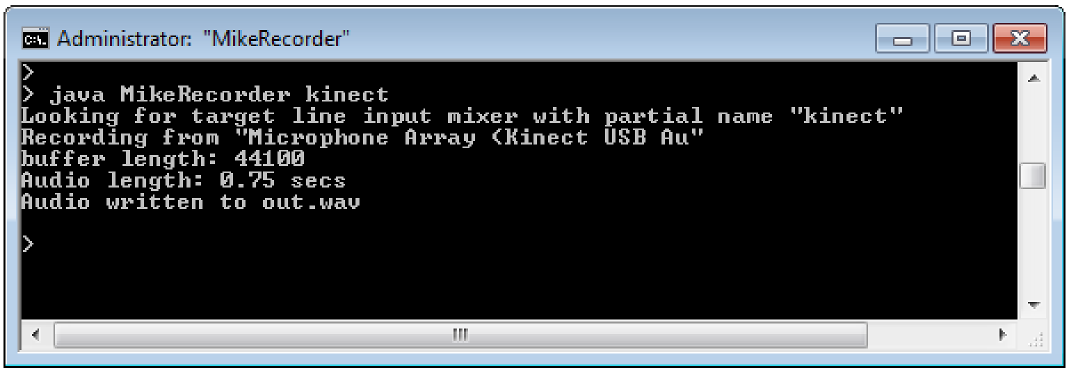


Figure 9. The MikeRecorder Application.

Recording commences when the user presses "Start", and finishes when the user presses the "Stop" button. The recording is automatically saved into a WAV file called out.wav, and the application exits.

The duration of the recorded clip is reported on the command line (see Figure 10).



```
>
> java MikeRecorder kinect
Looking for target line input mixer with partial name "kinect"
Recording from "Microphone Array (Kinect USB Au"
buffer length: 44100
Audio length: 0.75 secs
Audio written to out.wav

>
```

Figure 10. MikeRecorder on the Command Line.

The user has to open the saved file in a separate audio application (e.g. Audacity) in order to playback the recording. Another feature that's missing is a volume control. If the volume needs changing, you can do it via the "Levels" Properties dialog in the Sound Control panel (shown in Figure 8).

MikeRecorder can be supplied with any mixer name, and so it's possible to use it to record from the PC's microphone or the Kinect.

The application's class diagrams are shown in Figure 11.

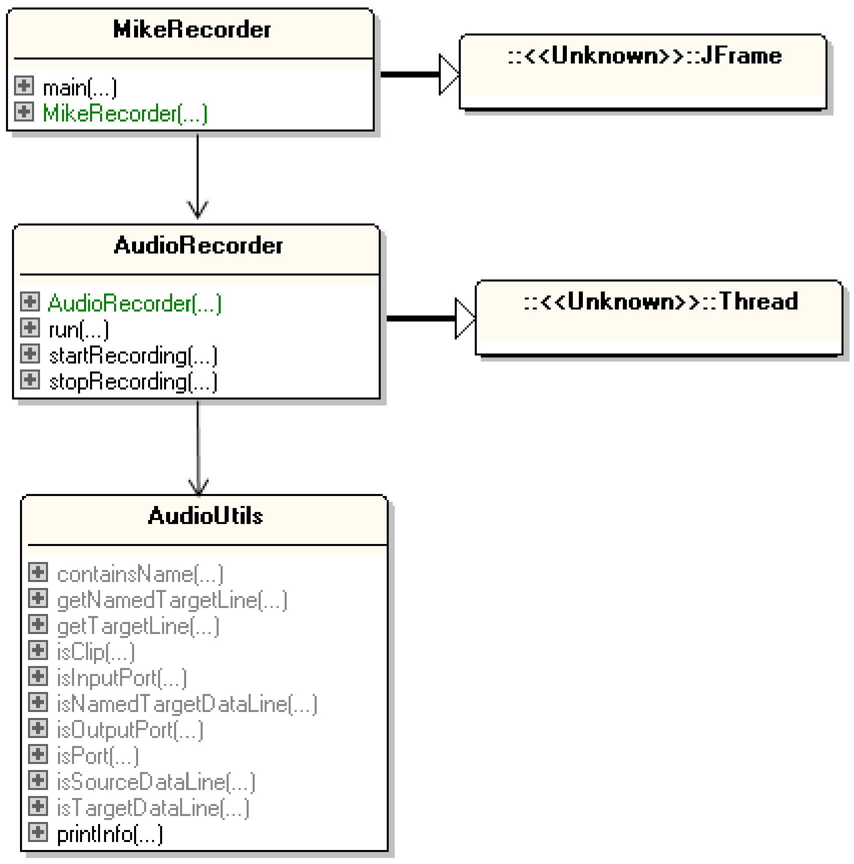


Figure 11. Class Diagrams for MikeRecorder.

The MikeRecorder class sets up the GUI, and audio capture is managed by AudioRecorder using some of the library methods in AudioUtils.

The recording is carried out by a separate thread inside AudioRecorder so that MikeRecorder's GUI doesn't lock up, making it impossible for a "Stop" button press to be processed.

4.1. Initializing the Recorder

The AudioRecorder() constructor has a single task – to initialize a TargetDataLine coming from the named mixer. There's usually multiple TargetDataLine formats available for a mixer (we saw that the Kinect mixer offers 8 formats according to NamedCaptureMixers in section 3.3), so part of the initialization involves specifying which format is required.

```
// globals
private String outFnm;    // file used to save the recording
private AudioFormat audioFormat;
private TargetDataLine tdLine = null;      // audio input line


public AudioRecorder(String inMixName, String fnm)
{
  outFnm = fnm;

  // record at CD quality:
  //    PCM 44.1 kHz, 16 bit, stereo, signed, big endian
  audioFormat =  new AudioFormat( 44100.0F, 16, 2, true, false);

  // initialize target line coming from the named microphone
  // with the audio format
  tdLine = initMike(inMixName, audioFormat);
  if (tdLine == null) {
    System.out.println("No suitable target line input mixer found");
    System.exit(1);
  }
}  // end of AudioRecorder()
```

A fairly safe bet for most sound cards is to use a CD quality PCM (pulse code modulation) audio format, to record stereo 16-bit audio at 44.1 kHz. A look back at the output from NamedCaptureMixers confirms that this format is available, although there's no information on the sample rate.

initMike() tries to find a suitable mixer using three slightly different calls to AudioUtils.getTargetLine().

```
// global
private static final String MIKE_NAME =
            "Primary Sound Capture Driver";  // in Windows
  //        "Built-in Microphone";           // MAC OS X
  //        "plughw:0,0";                    // Linux


private TargetDataLine initMike(String nm, AudioFormat audioFormat)
{
  System.out.println("Looking for target line input mixer
                             with partial name \"" + nm + "\"");
  TargetDataLine tdLine = AudioUtils.getTargetLine(nm, audioFormat);

  if (tdLine == null) {
    System.out.println("\nLooking for standard mike: \"" +
                                             MIKE_NAME + "\"");
    tdLine = AudioUtils.getTargetLine(MIKE_NAME, audioFormat);
  }

  if (tdLine == null) {
    System.out.println("\nLooking for any target line input mixer");
    tdLine = AudioUtils.getTargetLine("", audioFormat);
                         // i.e. ignore name constraint
  }

  return tdLine;
}  // end of initMike()
```

First a search is made for an input mixer that uses the supplied name. If that fails a search is carried out for a mixer using a common name stored in MIKE_NAME. If that turns up nothing, then an input mixer is chosen independent of its name. A suitable name for MIKE_NAME depends on the OS, so I've included examples for Windows, Mac OS X, and Linux.

The retrieval of the TargetDataLine for the specified mixer name and audio format is handled by AudioUtils.getTargetLine().

```
// in AudioUtils
public static TargetDataLine getTargetLine(String nm,
                                     AudioFormat audioFormat)
{
  Mixer mixer = getNamedTargetLine(nm);
  if (mixer == null) {
    System.out.println("No target line mixer found for \"" +
                                                  nm + "\"");
    return null;
  }
  System.out.println("Recording from \"" +
                        mixer.getMixerInfo().getName() + "\"");

  TargetDataLine tdLine = null;
  DataLine.Info info =
           new DataLine.Info(TargetDataLine.class, audioFormat);
  try {
    tdLine = (TargetDataLine) mixer.getLine(info);
                        // request a suitable line from the mixer
    tdLine.open(audioFormat);   // open it for reading
  }
  catch (LineUnavailableException e) {
    System.out.println("Unable to access the capture line");
  }
  return tdLine;
}  // end of getTargetLine()
```

First getTargetLine() is called to find the named mixer which contains a TargetDataLine.

```
// in AudioUtils
public static Mixer getNamedTargetLine(String nm)
{
  for (Mixer.Info mi : AudioSystem.getMixerInfo()) {
    Mixer mixer = AudioSystem.getMixer(mi);
    if (isNamedTargetDataLine(mixer, nm))
      return AudioSystem.getMixer(mi);
  }
  return null;
}  // end of getNamedTargetLines()
```

This uses isNamedTargetDataLine(), which I described earlier, to select the first suitable mixer.

Back in getTargetLine(), a DataLine.Info object is created for the TargetDataLine class and the supplied audio format, and this is used to query the retrieved mixer. If the audio format is acceptable to the mixer, then a matching TargetDataLine is returned. This is opened so that audio capture can commence, and the line is returned to initMike() in AudioRecorder.

4.2. Starting and Stopping the Recording

When the user presses the "Start" button in the GUI (see Figure 9), the startRecording() method is called in AudioRecorder to commence the recording thread.

```
// globals
private String outFnm;
private AudioFormat audioFormat;


public void startRecording()
// called from the "Start" button in the GUI
{ this.start();  }


public void run()
{
  byte[] audioBytes = recordBytes();

  // convert bytes to audio stream
  ByteArrayInputStream bais = new ByteArrayInputStream(audioBytes);
  AudioInputStream ais = new AudioInputStream(bais, audioFormat,
            audioBytes.length / audioFormat.getFrameSize());

  // report audio duration
  long millisecs = (long) ((ais.getFrameLength() * 1000) /
                                 audioFormat.getFrameRate());
  System.out.println("Audio length: " +
                (millisecs / 1000.0) + " secs");

  saveAudio(ais, outFnm);
  System.exit(0);
}  // end of run()
```

The actual job of recording is delegated to recordBytes(), which continues until the user presses the "Stop" button in the GUI. The button press triggers a call to AudioRecorder.stopRecording(), which sets a global boolean, isRunning, to false.

```
// global
private volatile boolean isRunning  = true;

public void stopRecording()
{ isRunning = false; }
```

This causes recordBytes() to return to run(), which exits after reporting the audio's duration and saving the clip to a file. The audio is converted from a byte array into an audio stream to make these two tasks easier. Saving is particularly straightforward with AudioSystem.write()

```
private void saveAudio(AudioInputStream ais, String outFnm)
// save audio stream to a WAV file
{
  try {
    AudioSystem.write(ais, AudioFileFormat.Type.WAVE,
                                       new File(outFnm));
    System.out.println("Audio written to " + outFnm);
  }
  catch (IOException e)
  {  System.out.println("Unable to write audio to " + outFnm); }
}  // end of saveAudio()
```

4.3. Recording Chunks of Sound

recordBytes() reads chunks of audio data from the TargetDataLine and stores them in an ever-growing ByteArrayOutputStream. When the recording is stopped, this stream is converted into a byte array and returned.

```
// globals
private TargetDataLine tdLine = null;      // audio input line
private volatile boolean isRunning  = true;


private byte[] recordBytes()
{
  // create byte stream for holding recorded input
  ByteArrayOutputStream out = new ByteArrayOutputStream();

  // create data buffer for holding input audio
  byte[] data = new byte[tdLine.getBufferSize() / 5];

  // read the data from the target data line, write to byte stream
  int numBytesRead = 0;
  tdLine.start();    // begin audio capture
  while (isRunning) {
    if ((numBytesRead = tdLine.read(data, 0, data.length)) == -1)
      break;
    out.write(data, 0, numBytesRead);
  }

  // end of the recording - stop and close the target data line
  tdLine.stop();
  tdLine.close();

  // return audio as a byte array
return out.toByteArray();
}  // end of recordBytes()
```

The size of the input buffer (the data[] array) should be some fraction of the line's buffer size, so the OS can more easily switch between this recording thread and filling the TargetDataLine.

TargetDataLine.read() takes three arguments: a byte array, an offset into the array (typically 0), and the number of bytes of input data that should be read. It returns the number of bytes that were actually read into the byte array. ByteArrayOutputStream.write() adds the contents of the array to the end of the stream.

The recording can be stopped in two ways, either because of a TargetDataLine.read() error, or if the global isRunning boolean is set to false in stopRecording().

5. Beamforming

One of the novel features of Microsoft's Kinect SDK is *beamforming* (also called sound localization) – the ability to determine the direction of an audio source by analyzing differences between the audio streams created by the microphone array.

The SDK doesn't supply the distance to an audio source, just its current angle in the XZ plane relative to the center of the Kinect. An 0 angle indicates that the sound is coming from in front of the sensor, a positive angle means that the sound is coming from the user's left, and negative places the sound on the right (as shown in Figure 12).

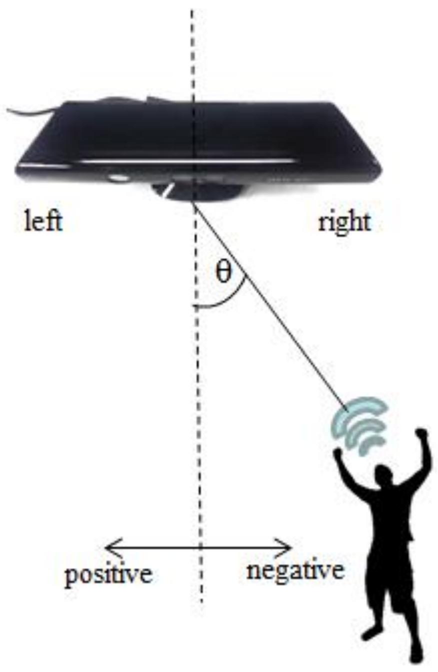


Figure 12. Beamforming with the Kinect.

The mathematics behind beamforming is rather complex but the simple case, involving two monaural inputs, is easier. I'll explain *binaural* sound localization with the help of Figure 13, which involves two monaural microphones, d meters apart.

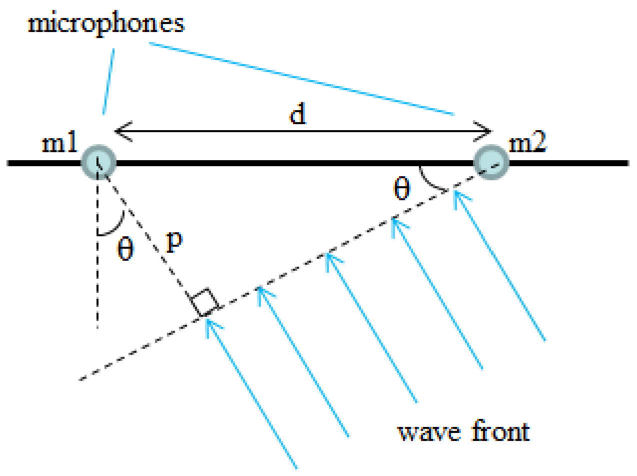


Figure 13. Binaural Beamforming.

A sound's wave front arrives rotated by θ angles to the perpendicular to a line joining the two microphones, m1 and m2. The microphone on the left (m1) receives the sound a little later than the one on the right (m2) because the left edge of the wave has to travel an extra distance p. This can be expressed as the equation:

$\sin \theta = p / d$

The delay can also be defined in terms of sound velocity and time:

$c = p / \Delta t$

c the speed of sound (in m/s), and Δt is the time delay (in seconds) between the wave reaching m2 and m1.

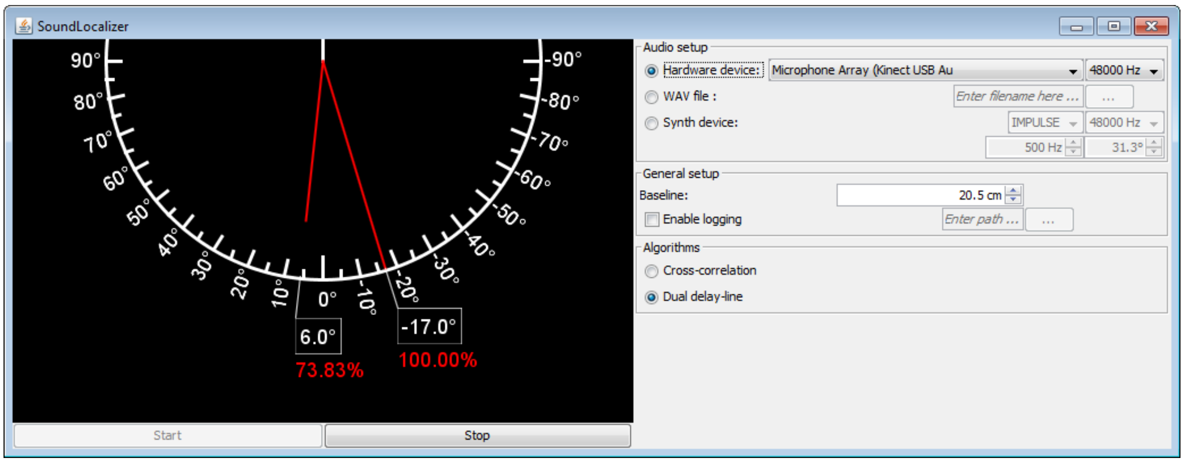
These two equations can be combined to produce an expression for θ:

$\theta = \arcsin(c \cdot \Delta t / d)$

The tricky part of this equation is determining Δt, which is typically achieved by comparing the wave fronts arriving at m1 and m2. They essentially have the same wave form, but the wave arriving at m1 will be time-shifted by a Δt amount compared to the wave at m2.

5.1. A Java Library for Binaural Localization

Laurent Calmes has developed a Java GUI application that implements two forms of binaural sound localization (http://www.laurentcalmes.lu/), using either cross-correlation or dual delay-lines to calculate the time shift (Δt) and therefore the wave angle θ. His application can utilizes stereo input from any Java mixer, which means that I can connect the Kinect sensor to it, as shown in Figure 14.



Audio setup
[✓ Hardware device:] Microphone Array (Kinect USB Au 48000 Hz
WAV file : Enter filename here ... ...
Synth device: IMPULSE 48000 Hz 500 Hz 31.3°

General setup
Baseline: 20.5 cm
Enable logging Enter path ... ...

Algorithms
Cross-correlation
[✓ Dual delay-line]

Figure 14. Binaural Localization in Java.

It's too small to see in Figure 14, but the "Audio Setup" area at the top-right has been initialized to use the Kinect microphone array. Two sound sources are currently being detected, with varying confidence levels: one to the user's left at 6.0 degrees with 73.83% confidence, and another to the right at -17 degrees with 100% confidence (which is me whistling). It's common for the localizer to pick up multiple sound sources with varying levels of confidence. In this example, the extra sound iscoming from the room's air conditioner.

It's possible to switch between the two localization algorithms (cross-correlation and dual delay-lines), but I didn't detect much difference in the results. In the application's documentation, Calmes says that the cross-correlation algorithm is fastest, while dual delay-lines is more accurate.

5.2. Using Calmes' SoundLocalizer as a Library

Calmes' SoundLocalizer application makes beamforming with the Kinect possible in Java, but I'd like to access the computed data (i.e. the audio angles and confidence levels) without the GUI, to make it easier to combine beamforming with other Kinect code.

Unfortunately Calmes' application doesn't come with a separate beamforming library, but he does supply class documentation and I employed the JD-GUI decompiler (http://java.decompiler.free.fr/?q=jdgui) to further examine his code. The GUI code is mostly located in his SoundLocalizerWindow class, and it was fairly easy to pull out the non-GUI parts from there.

One minor problem was that Calmes had (inadvertently) defined one of his important classes, HWDevice, to have a private constructor. I changed it to be public, and created a new JAR file, SL.jar, containing the rest of his code unchanged. This JAR can be found at my Kinect website (http://fivedots.coe.psu.ac.th/~ad/kinect/) in this chapter's code area.

My SimpleSoundLoc.java application doesn't have a GUI, instead printing the detected angles and their confidence levels to standard output. It's hardwired to connect to the Kinect mixer, and uses the dual delay-lines algorithm (but its easy to change to the cross-correlation version). Typical output:

```
> java -cp "SL.jar;." SimpleSoundLoc
Found "Microphone Array (Kinect USB Au"
(angle, conf): (-27.0, 1.00) (-9.5, 0.85) (13.5, 0.62) (-61.0, 0.58)
(angle, conf): (-25.5, 1.00) (-10.5, 0.97) (11.5, 0.74)
(angle, conf): (-25.5, 1.00) (-11.0, 0.96) (11.0, 0.67) (-63.0, 0.53)
(angle, conf): (-22.5, 1.00) (-12.5, 0.95) (9.5, 0.57)
(angle, conf): (-11.5, 1.00) (-19.5, 0.99)
(angle, conf): (-12.5, 1.00) (9.0, 0.69)
(angle, conf): (-12.5, 1.00) (9.5, 0.76) (28.5, 0.57)
(angle, conf): (-12.5, 1.00) (9.0, 0.73) (29.0, 0.58)
(angle, conf): (-13.5, 1.00) (9.0, 0.70) (27.5, 0.54)
(angle, conf): (-13.0, 1.00) (24.0, 0.52) (28.5, 0.52)
(angle, conf): (-12.5, 1.00) (9.0, 0.67) (29.5, 0.52)
(angle, conf): (-12.5, 1.00)
   :
```

Each line reports the currently detected sound sources – negative angles are to the right, positive angles to the left, as in Figure 12. Each line lists all the currently detected angles in decreasing order of confidence. Usually the correct source is the 100% confidence result, but occasionally it drops to second or third place in a noisy environment.

The application is terminated by the user typing ctrl-c.

I've deliberately made SimpleSoundLoc simple so it's clear how to utilize Calmes' classes. Initially, a HWDevice object is created (akin to an input mixer), and one of the sound localization algorithms is assigned to the device. Then a loop is entered which repeatedly calculates and reports the detected angles and confidence levels.

```
private static final String INPUT_MIXER = "kinect";
                         // this could be "sigmatel"


public static void main(String[] args) throws Exception
{
  Mixer.Info mi = getTarget(INPUT_MIXER);

  double tfSize = 0.062;       // timeframe size (in secs)
  AudioFormat af = new AudioFormat(16000.0f, 16, 2, true, false);
            // sampleRate, sample size in bits,
            // no. of channels, is signed,  little Endian

  HWDevice hwd = new HWDevice(tfSize, af, mi);
                    // timeframe, audio format, mixer
  hwd.init();

  LocalizationAlgorithm alg = initAlg(tfSize, hwd);

  List<LocalizationAlgorithm.Az> peaks;
     // list of detected sound sources in decreasing conf order
  while (true) {
    peaks = alg.compute();
    reportPeaks(peaks);
  }
}  // end of main()
```

The Java mixer representing the Kinect is obtained by using its partial name (i.e. "kinect"). I employ AudioUtils.getNamedTargetLine() which was introduced back in the MikeRecorder application, in section 4.1.

```
private static Mixer.Info getTarget(String nm)
{
  Mixer mixer =  AudioUtils.getNamedTargetLine(nm);
  if (mixer == null) {
    System.out.println("No mixer using \"" + nm +"\" found");
    System.exit(1);
  }

  Mixer.Info mi = mixer.getMixerInfo();
  System.out.println("Found \"" + mi.getName() + "\"");
  return mi;
}  // end of getTarget()
```

The main purpose of getTarget() is to convert the returned Mixer object into a Mixer.Info instance required by Calmes' HWDevice constructor back in main().

Initializing the Sound Localization Algorithms

My initAlg() method hides the code for creating a DDLAlgorithm object which implements the dual delay-lines algorithm.

```
private static LocalizationAlgorithm initAlg(double tfSize,
                                             HWDevice hwd)
                                      throws Exception
// uses the dual-delay line algorithm
{
  int tfSizeSpls = (int) Math.round(tfSize *
                           hwd.getFormat().getSampleRate());
  int logSize = FFT.log2(tfSizeSpls);
  int fftSize = 1 << logSize;
  if (fftSize < tfSizeSpls)
    fftSize <<= 1;

  DDLAlgorithm mAlg = new DDLAlgorithm(0.226, 361, 0.88,
                                         fftSize, tfSize, hwd);
    /* baseline dist in metres, number of delays in the delay line,
       coincidence map time integration time constant in secs,
       size of the FFT in samples, timeframe size in secs,
       audio device
    */
  mAlg.init();
  return mAlg;
}  // end of initAlg()
```

I have to admit that I don't fully understand the numerical arguments for the DDLAlgorithm constructor. I copied most of them from Calmes' decompiled SoundLocalizerWindow class. However, Calmes does a good job of explaining his code in the Java documentation and at his website, so I realized that I needed to change the first argument of the DDLAlgorithm constructor, the baseline distance, which corresponds to the d variable in Figure 13.

Unfortunately, it's not clear what value I should assign to d, since the Kinect has four microphones, at varying distances from each other (see Figure 6), and I don't know how those four sources are combined by the sound card into the two channels reaching Java. I decided to assume that the channels came from the microphones at either end of the Kinect sensor, which are 226mm apart. This choice seems to produce fairly accurate angle values at runtime.

I also tried out Calmes' cross correlation algorithm inside initAlg(), which is defined like so:

```
private static LocalizationAlgorithm initAlg(double tfSize,
                                              HWDevice hwd)
                                    throws Exception
// uses the cross-correlation algorithm
{
  CorrelationAlgorithm mAlg =
         new CorrelationAlgorithm(0.226, 0.01, tfSize, hwd);
     /* baseline dist in meters, integration time in secs,
        timeframe size in secs, audio device
     */
  mAlg.init();
  return mAlg;
  }  // end of initAlg()
```

Once again, I changed the first constructor argument, the baseline distance, to 0.226 m.

I couldn't detect any difference between the results for these two algorithms, and both appear to execute at about the same speed.

Reporting the Sound Sources

Back in the main(), LocalizationAlgorithm.compute() is repeatedly called to store the current sound angles and their confidence levels in a list. reportPeaks() loops through that list printing out the data, which is already sorted by confidence level.

```
private static void reportPeaks(List<LocalizationAlgorithm.Az> peaks)
{
  System.out.print("(angle, conf): ");
  for (LocalizationAlgorithm.Az az : peaks) {
    double rotAng = az.getAzimuth();  // angle
    double conf = az.getHeight();     // confidence value
    System.out.printf("(%.1f, %.2f) ", rotAng, conf);
  }
  System.out.println();
}  // end of reportPeaks()
```

The list contains LocalizationAlgorithm.Az objects which each hold an angle and its confidence value.

6. Back to Breakout

My main motivation for this chapter was to recode the Spoken Breakout game of the previous chapter so that I could play it without holding a microphone. I'll still utilize speech recognition, but with its audio input coming from the Kinect.

Figure 15 shows the Kinect version of Spoken Breakout, which looks much the same as before, except that I've no longer got a microphone in my hand. I can still use voice commands such as "pause", "resume", "exit", and "move faster".

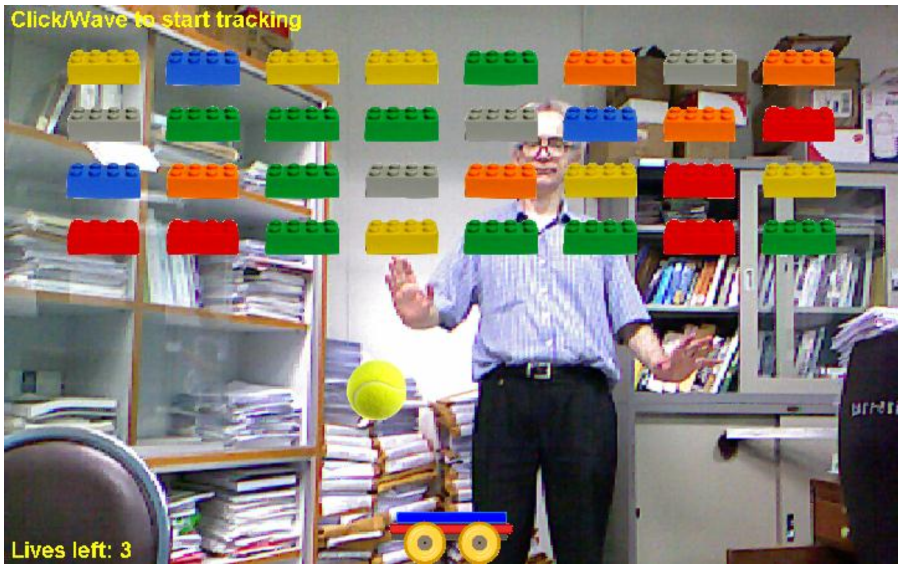


Figure 15. Playing Spoken Breakout with Kinect Audio.

Figure 16 shows the class diagrams for this version of the game, with the modified portions shaded in blue.

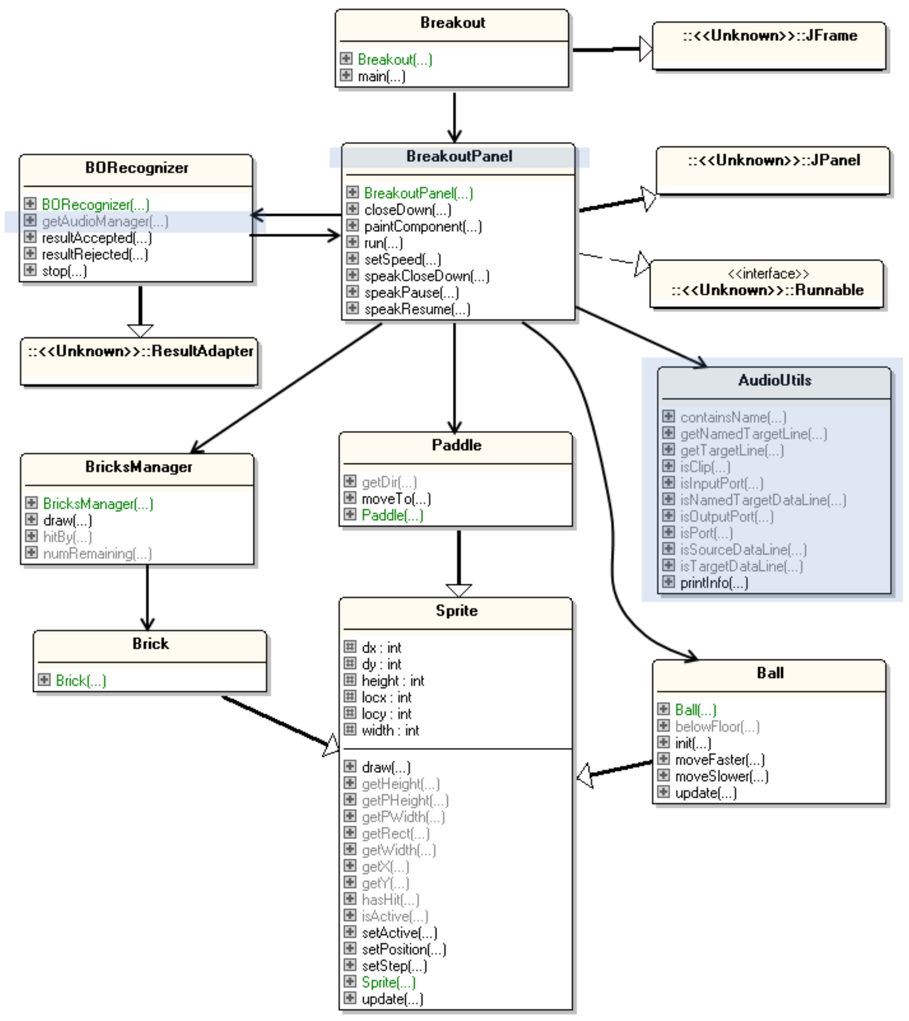


Figure 16. Class Diagrams for the Kinect Spoken Breakout Game.

The largest change is the addition of the AudioUtils library class, which I've been using throughout this chapter. I need it for accessing the TargetDataLine coming from the Kinect. There's one new public method in BORecognizer, and a new private method in BreakoutPanel. The rest of the application is unchanged from Chapter 14.

6.1. Modifying the Recognizer

The BORecognizer class still creates a recognizer that supports English, loads the command grammar, and starts the listener which invokes methods over in BreakoutPanel. The only addition is a public method for returning a reference to the recognizer's audio manager:

```
// global
private Recognizer rec;


public AudioManager getAudioManager()
{  return rec.getAudioManager();  }
```

The AudioManager class in JSAPI only supports the addition and removal of audio listeners, but TalkingJava extends it in some useful ways, as explained next.

6.2. Connecting the Kinect to the Speech Recognizer

BreakoutPanel is still the heart of the application, managing the initialization of OpenNI, and updating and drawing the game onto the panel. Its new task is to connect the TargetDataLine coming from the Kinect into the speech recognizer. Unfortunately, audio stream redirection isn't supported in standard JSAPI, but is available as an extended feature of TalkingJava.

The redirection is carried out by useKinectMike(), which is called at the end of BreakoutPanel's constructor after a BORecognizer object has been created:

```
// global
private BORecognizer rec;    // the speech recognizer


public BreakoutPanel(JFrame top)
{
  // : same code as in chapter 14

  rec = new BORecognizer(this);
  useKinectMike(rec);

  new Thread(this).start();   // start updating the panel
}  // end of BreakoutPanel()
```

Inside useKinectMike(), the TalkingJava-specific class CGAudioManager connects a suitable TargetDataLine coming from the Kinect into the speech recognizer.

```
private void useKinectMike(BORecognizer rec)
{
  CGAudioManager audioMan = (CGAudioManager)rec.getAudioManager();

  // get a suitable target line coming from the Kinect
  TargetDataLine tdLine =  AudioUtils.getTargetLine("kinect",
                                 audioMan.getAudioFormat());
  if (tdLine == null) {
    System.out.println("Kinect input not found");
    System.exit(1);
  }

  try {
    // connect Kinect's target line to recognizer's audio manager
    AudioLineSource source = new AudioLineSource(tdLine);
    audioMan.setSource(source);
    source.startSending();

    System.out.println("\nSay something please...");
  }
  catch(Exception e) {
    e.printStackTrace();
  }
}  // end of useKinectMike()
```

TalkingJava's CGAudioManager implements the AudioManager, AudioObject, AudioSink, and AudioSource interfaces. AudioSink and AudioSource aren't part of JSAPI; they contain methods for redirecting audio IO streams.

AudioUtils.getTargetLine() (from section 4.1) is used to find the Java mixer for the Kinect, and a TargetDataLine is selected based on the audio format used by recognizer.

AudioLineSource is TalkingJava's wrapper for a TargetDataLine which allows it to be redirected by the audio manager. There are similar wrappers for redirecting files, sockets, and JMF media, both as sources and as sinks for the speech recognizer.

The redirection is set up with CGAudioManager.setSource(), and audio data starts flowing with the call to CGAudioManager.startSending().

7. More on Java Sound

As I mentioned earlier, the Java sound APIs are quite large, dealing with both sampled sound and MIDI, both for recording, playback, and manipulation. I've only focused on the basics of audio capture in this chapter, but if you want to find out more, there are plenty of resources available. The official Java tutorial has a good section on sound at http://docs.oracle.com/javase/tutorial/sound/, and the Java Sound Resources site (http://www.jsresources.org/) offers many examples and FAQs.

One of the best articles on Java sound is "Wired for Sound" by Sing Li (http://www.developerfusion.com/article/84314/wired-for-sound/) which discusses basic concepts and includes a karaoke example with a volume control for the voice recorder part. Unfortunately, the source code seems to have disappeared from that site.

Dick Baldwin has written a selection of online articles about Java sound at http://www.dickbaldwin.com/tocadv.htm in the "Java Sound" section towards the bottom of that page. Of particular relevance is "Capturing Microphone Data into an Audio File" at http://www.developer.com/java/other/article.php/2105421.

A selection of small sound examples can be found at the Java2s site, http://www.java2s.com/Tutorial/Java/0120__Development/0880__Audio.htm

There's an active forum on Java sound at https://forums.oracle.com/forums/forum.jspa?forumID=942