Chapter N13. 3D Sound with JOAL

This chapter interrupts my discussion of non-standard input devices programming so I can talk about JOAL (https://joal.dev.java.net/), a Java wrapper around OpenAL. OpenAL, the OpenAudio Library, is a cross-platform API for programming 2D and 3D audio (http://www.openal.org/).

I'll introduce JOAL (and OpenAL) by developing a JOALSoundMan class that simplifies the creation of spatial 3D sound effects, and I'll demonstrate it's use with a few simple, non-graphical examples. In chapter 14 ??, I'll utilize JOALSoundMan to add 3D sound to Java 3D, and it'll crop up again in chapter 17 ??, managing the audio effects in my JOGL code.

1. Why JOAL and Java 3D?

In chapter 1, I mentioned that Java 3D supports 2D and 3D sound. The API includes three sound-related classes: BackgroundSound for ambient sound (sound that's audible everywhere in the scene), PointSound for a sound source located at a particular spot, and ConeSound for a point source aimed in a specific direction.

PointSound and ConeSound are *spatial* sound classes, since their volume, and the audio mix coming from the left and right speakers depends on the sound's location in relation to the listener (which is usually the camera's viewpoint). Another factor is the relative velocities of the sound source and listener if they're moving.

The bad news is that PointSound and ConeSound contain some nasty bugs, which led to their demotion to optional parts of Java 3D starting with version 1.3.2. Also, the Java 3D development team wanted the classes reimplemented so they didn't rely on the Headspace audio engine, third-party software not maintained by Sun. They hoped to persuade a kindly Java 3D community member to do this work, using JOAL.

The end's almost in sight – in July 2006, David Grace posted JoalMixer to the org.jdesktop.j3d.audioengines.joal branch of the j3d-incubator project (at https://j3d-incubator.dev.java.net/). It includes revised PointSound, ConeSound, and BackgroundSound classes, built with JOAL.

Unfortunately, JoalMixer arrived too late for the Java 3D 1.5 release, and can only be utilized at the moment by recompiling the Java 3D sources. For gallant users interested in this approach, there are detailed instructions in the java.net Java 3D forum thread on spatialized audio at http://forums.java.net/jive/thread.jspa?threadID=4638&start=0&tstart=105.

I decided *not* to be gallant, and roll my own JOAL code instead (a JOALSoundMan class). It offers ambient and point sounds, and is reusable across Java, Java 3D, and JOGL applications, without requiring any recompilation of those APIs.

 Andrew Davison © 2007

2. Background on OpenAL and JOAL

JOAL is a thin layer of Java over OpenAL, so its 2D and 3D audio features are really those of OpenAL. For that reason, I'll talk about OpenAL first, and then JOAL.

OpenAL supports the construction of a sound application containing a collection of audio sources located in 3D space, heard by a single listener. The space doesn't need a graphical representation in the application.

The main OpenAL entities are the buffer, source, and listener.

- A *buffer* stores sound information, typically a sound clip loaded from a WAV file. There may be many buffers in a program.
- A *source* is a point in the 3D space which emits a sound in all directions. A source isn't the audio sample, but the location where the sample is played. Each source must refer to a buffer, to have a sound to play. It's possible to connect several sources to the same buffer.
- An application has a single *listener*, which represents the user in the scene. Listener properties (position, and perhaps velocity) are combined with the properties of each source (position, velocity, etc.) to determine how the source's audio is heard.

OpenAL is available on a wide range of platforms, including Windows, OS X, Linux, the PS2, and Xbox. The API is hardware-independent but utilizes hardware support if the underlying sound card has it. There's a growing list of commercial games employing OpenAL, including Doom 3, Unreal Tournament 2004, and Battlefield 2, and it has a wide following in the open-source games world (see the list at http://www.openal.org/titles.html).

OpenAL is currently maintained by Creative Labs, perhaps best known for their Sound Blaster line of audio cards.

The Windows version of OpenAL (and JOAL) support Creative Lab's EAX and EFX technologies (Environmental Audio eXtensions and EFfects eXtensions). EAX offers reverberations and low-pass filtering effects, while EFX adds filtering at the OpenAL Source level. EAX and EFX require sound card support, so they aren't widely available.

OpenAL's main website is at http://www.openal.org/, which includes various ports of the library, and documentation. You don't need the software since we're using JOAL, but the OpenAL 1.1 Programmer's Guide and specification are worth a read.

An excellent series of OpenAL programming articles can be found at devmaster.net (http://www.devmaster.net/articles.php?catID=6), and there's more links at http://www.openal.org/links.html. Although these tutorials are written in C for OpenAL, JOAL methods and OpenAL functions are so similar that the information is still helpful.

There are two official OpenAL mailing lists archived at http://opensource.creative.com/pipermail/openal/ and http://opensource.creative.com/pipermail/openal-devel/. There's a Nabble forum for OpenAL at http://www.nabble.com/OpenAL-f14243.html.

 Andrew Davison © 2007

2.1. What about JOAL?

I'm not using OpenAL directly, but rather the Java wrapper, JOAL. Nearly all the necessary JOAL software can be found at https://joal.dev.java.net/, so there's no need to download the OpenAL libraries. I'll explain the gory installation details shortly.

The devmaster.net OpenAL tutorials have been 'translated' into JOGL (at https://joal-demos.dev.java.net) except for one on OggVorbis streaming. Starfire Research also has some brief, but good, JOAL examples, starting at http://www.starfireresearch.com/services/java/docs/joal.html.

The main JOAL forum is at http://www.javagaming.org/forums/index.php?board=26.0. There's also a JOAL list at the lwjgl site, http://lwjgl.org/forum/viewforum.php?f=10. The focus is on the LightWeight Java Games Library (lwjgl), but there's plenty of general information as well. If these forums don't answer your questions, then you should consider searching through the OpenAL lists that I mentioned above.

2.2. Installing JOAL

I downloaded the Windows version of JOAL, release build 1.1.0, dated December 22nd, from https://joal.dev.java.net/, via the "Documents & files" menu. The file (joal-1.1.0-windows-i586.zip) contains two JAR files, joal.jar and gluegen-rt.jar, and two DLLs, joal-native.dll and gluegen-rt.dll. Other useful downloads are the API documentation (joal-1.1.0-docs.zip) and the source code for the demos (joal-demos-src.zip).

joal.jar and gluegen-rt.jar should be copied into <JAVA_HOME>\jre\lib\ext and <JRE_HOME>\lib\ext. On my WinXP test machine, they're the directories c:\Program Files\Java\jdk1.6.0\jre\lib\ext and c:\Program Files\Java\jre1.6.0\lib\ext.

joal-native.dll and gluegen-rt.dll should be placed in a directory of your choice (I chose d:\joal).

Andrew Davison © 2007

Figure 1 shows the two DLLs in my d:\joal directory.

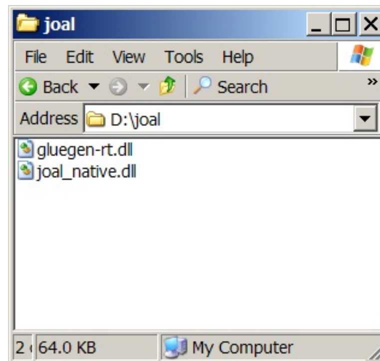

Figure 1. The DLLs Needed for JOAL.

These DLLs are needed at runtime, so every call to java.exe requires a command line option to supply the path to the libraries. For instance, to run the JOAL application JoalExample1.java needs:

```
> java -Djava.library.path="d:\joal" JoalExample1
```

I'm lazy so I use a DOS batch file instead:

```
> runJoal JoalExample1
```

runJoal.bat contains:

```
@echo off
echo Executing %1 with JOAL...
java -Djava.library.path="d:\joal" %1 %2
echo Finished.
```

The %1 and %2 arguments allow at most two command line arguments to be passed to the java.exe call.

3. Managing JOAL Sounds

Most JOAL applications look the same: there's an initialization phase, then the buffers, their sources, and the listener are created. During the program's execution, the sources and/or listener are moved around, causing the sound output to change. At termination, the source and buffers are deleted.

I decided to package up these stages into a JOALSoundMan class in order to hide the repetitive manipulation of the low-level buffers, sources, and listener data structures.

JOALSoundMan has three groups of public methods:

- general-purpose: JOALSoundMan(), cleanUp()
- buffer and source related: load(), setPos(), play(), pause(), stop()
- listener related: moveListener(), setListenerPos(), turnListener(), getX(), getZ(), getAngle()

1. General. JOALSoundMan() initializes JOAL, and creates a listener located at (0,0,0) facing along the -z axis. cleanUp() deletes any source and buffer data structures created during the application's execution.

2. Buffer and Source. load() loads a specified WAV file, and creates a buffer for it. The buffer is then associated with a new source, located at the origin. A second version of load() includes a (x, y, z) coordinate for positioning the source. Every source is assigned a name which is used to refer to it in other methods.

setPos() changes the position of a named source. play() plays the named source, pause() pauses the source, and stop() stops it.

3. Listener. moveListener() moves the listener by a specified x- and z- step. setListenerPos() moves the listener to the new (x, 0, z) coordinate. It's not possible to change the listener's y-axis position, a restriction that simplifies the coding.

turnListener() turns the listener a specified number of degrees around its y-axis. There's no way to rotate the listener around the x- or z- axes. getX(), getY(), and getAngle() returns the listener's current position and y-axis angle.

I'll explain these methods in more detail in the following sections.

3.1. Initializing JOAL

The JOALSoundMan constructor creates two HashMaps, initializes OpenAL, and creates the listener.

```
// global stores for the sounds
private HashMap<String, int[]> buffersMap;   // (name, buffer) pairs
private HashMap<String, int[]> sourcesMap;  // (name, source) pairs


public JOALSoundMan()
{
  buffersMap = new HashMap<String, int[]>();
  sourcesMap = new HashMap<String, int[]>();

  initOpenAL();
  initListener();
} // end of JOALSoundMan()
```

JOALSoundMan assumes that every sound will have it own JOAL buffer and source. The two HashMaps store them, indexed by the sound's name.

initOpenAL() sets up a link to OpenAL via the ALut library (so named to remind OpenGL programmers of GLUT).

```
// globals
private AL al;  // to access OpenAL


private void initOpenAL()
{
  try {
    ALut.alutInit();    // creates an OpenAL context
```

```
    al = ALFactory.getAL();    // used to access OpenAL
    al.alGetError();           // clears any error bits

    // System.out.println("JOAL version: " + Version.getVersion());
  }
  catch (ALException e) {
    e.printStackTrace();
    System.exit(1);
  }
} // end of initOpenAL()
```

The System.out.println() call to display JOAL's version is commented out in initOpenAL() since the Version class isn't present in JOAL 1.1.0, although it was available in the previous beta releases.

3.2. Initializing the Listener

JOALSoundMan.initListener() places the listener at the origin, looking along the -z axis. Figure 2 illustrates the set-up: the triangle is the listener, with its apex pointing towards a "look at" point. The view is from above, looking down onto the XZ plane.

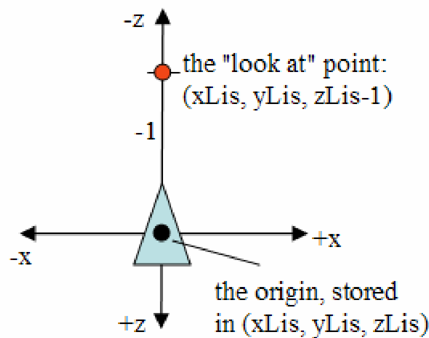

Figure 2. The Listener's Orientation.

setListener() stores the listener's position in the globals xLis, yLis, and zLis. The "look at" point is one unit along the -z axis, at (xLis, yLis, zLis-1).

```
// global listener info
private float xLis, yLis, zLis;    // current position
private float[] oriLis;   // orientation


private void initListener()
// position and orientate the listener
{
  xLis = 0.0f; yLis  = 0.0f; zLis = 0.0f;
  al.alListener3f(AL.AL_POSITION, xLis, yLis, zLis);
                        // position the listener at the origin

  al.alListener3i(AL.AL_VELOCITY, 0, 0, 0);   // no velocity

  oriLis = new float[] {xLis, yLis, zLis-1.0f,  0.0f, 1.0f, 0.0f};
     /* the first 3 elements are the "look at" point,
```

```
        the second 3 are the "up direction" */
  al.alListenerfv(AL.AL_ORIENTATION, oriLis, 0);
} // end of initListener()
```

The listener properties are set with calls to AL.alListenerXX() methods, where XX denotes the data type assigned to the property. The first argument of the method is the property being affected.

The listener's orientation is defined in terms of the “look at” point and a vector for the “up” direction. In initListener(), “up” is the +y axis. Both values are stored in a global oriLis[] array, and assigned to the AL.AL_ORIENTATION property.

It’s worth noting that oriLis[0], oriLis[1], and oriLis[2] are the “look at” point’s x-, y-, and z- values. The x- and z- values will be changed later as the listener moves around.

Where are the Listener's Ears?

The location of the listener's ears becomes important when the listener or sources move, since they dictate the volume and distribution of sound emitted by the speakers. The location of the ears, and even their number, isn't defined by the OpenAL specification; the details are left to the implementation.

 Andrew Davison © 2007

A listener in a standard PC environment with stereo speakers, or headphones, has two ears mapped to the speakers (headphones) as shown in Figure 3.

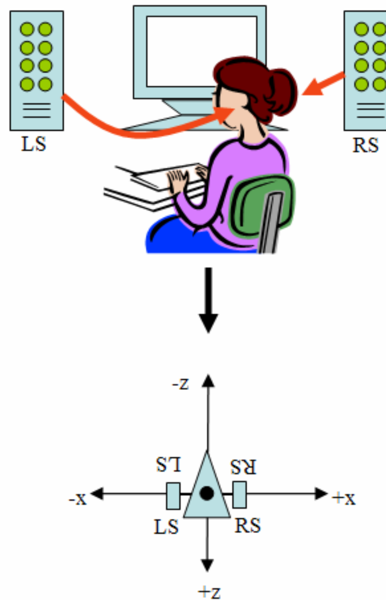

Figure 3. The Listener's Ears.

LS is the left speaker, RS the right hand one.

By default in OpenAL, the listener starts at the origin, and faces along the –z axis, so that its left and right ears are mapped to the left and right speakers. This means that most of the code in setListener() is unnecessary, since it duplicates the default position and orientation employed in JOAL. I included it just to be on the safe side.

3.3. JOAL Clean Up

Before the JOAL application terminates, it should stop playing sounds, and delete any buffers and sources.

```
public void cleanUp()
{
  Set<String> keys = sourcesMap.keySet();
  Iterator<String> iter = keys.iterator();

  String nm;
  int[] buffer, source;
  while(iter.hasNext()){
    nm = iter.next();

    source = sourcesMap.get(nm);
    System.out.println("Stopping " + nm);
    al.alSourceStop(source[0]);
    al.alDeleteSources(1, source, 0);

    buffer = buffersMap.get(nm);
    al.alDeleteBuffers(1, buffer, 0);
  }
```

```
  ALut.alutExit();
} // end of cleanUp()
```

The names associated with the sources are converted into an Iterator, which loops through the buffer and source HashMaps using Al.alDeleteBuffers() and Al.alDeleteSources() to delete the entries. Playing sounds are stopped with Al.alSourceStop().

The ALut.alutExit() method shuts down OpenAL, and closes the output device.

3.4. Loading a Sound

A sound is loaded in two stages: first it's converted into a JOAL buffer, then a JOAL source, linked to that buffer, is placed in the scene. The buffer and source references are stored in the global HashMaps, using the sound's name as the key.

```
public boolean load(String nm, boolean toLoop)
{
  if (sourcesMap.get(nm) != null) {
    System.out.println(nm + " already loaded");
    return true;
  }

  int[] buffer = initBuffer(nm);
  if (buffer == null)
    return false;

  int[] source = initSource(nm, buffer, toLoop);
  if (source == null) {
    al.alDeleteBuffers(1, buffer, 0);
              // no need for the buffer anymore
    return false;
  }

  if (toLoop)
    System.out.println("Looping source created for " + nm);
  else
    System.out.println("Source created for " + nm);

  buffersMap.put(nm, buffer);
  sourcesMap.put(nm, source);
  return true;
} // end of loadSource()
```

Buffer creation is handled by initBuffer(), while initSource() creates the source.

The toLoop boolean is used in initSource() to specify whether the source should play its sound repeatedly.

Making a Buffer

The WAV file is loaded into several data arrays, then the buffer is initialized with those arrays

```
// global
private final static String SOUND_DIR = "Sounds/";
          // where the WAV files are stored

private int[] initBuffer(String nm)
{
  // create arrays for holding various WAV file info
  int[] format = new int[1];
  ByteBuffer[] data = new ByteBuffer[1];
  int[] size = new int[1];
  int[] freq = new int[1];
  int[] loop = new int[1];

  // load WAV file into the data arrays
  String fnm = SOUND_DIR + nm + ".wav";
  try {
    ALut.alutLoadWAVFile(fnm, format, data, size, freq, loop);
  }
  catch(ALException e) {
    System.out.println("Error loading WAV file: " + fnm);
    return null;
  }
  // System.out.println("Sound size = " + size[0]);
  // System.out.println("Sound freq = " + freq[0]);


  // create an empty buffer to hold the sound data
  int[] buffer = new int[1];
  al.alGenBuffers(1, buffer, 0);
  if (al.alGetError() != AL.AL_NO_ERROR) {
    System.out.println("Could not create a buffer for " + nm);
    return null;
  }

  // store data in the buffer
  al.alBufferData(buffer[0], format[0], data[0], size[0], freq[0]);

  // ALut.alutUnloadWAV(format[0], data[0], size[0], freq[0]);
            // not in API anymore
  return buffer;
}  // end of initBuffer()
```

ALut only offers ALut.alutLoadWAVFile() at present, so a buffer is restricted to being a stereo or mono WAV file.

The size[] and freq[] data arrays contain information on the size and frequency of the loaded file. The commented-out println()'s show how to access it:

```
// System.out.println("Sound size = " + size[0]);
// System.out.println("Sound freq = " + freq[0]);
```

An empty buffer is created with AL.alGenBuffers(), then filled with the data from the arrays with Al.alBufferData().

Earlier versions of JOAL used ALut.alutUnloadWAV() to release the WAV file. The call is no longer required, and has been removed from the API.

initBuffer()'s result is an array holding a reference to the buffer, or null.

Making a Source

The source is positioned at (0,0,0), and linked to the buffer that was just created. The source may play repeatedly, depending on the toLoop argument.

```
private int[] initSource(String nm, int[] buf, boolean toLoop)
{
  // create a source (a point in space that emits a sound)
  int[] source = new int[1];
  al.alGenSources(1, source, 0);
  if (al.alGetError() != AL.AL_NO_ERROR) {
    System.out.println("Error creating source for " + nm);
    return null;
  }

  // configure the source
  al.alSourcei(source[0], AL.AL_BUFFER, buf[0]);  // bind buffer
  al.alSourcef(source[0], AL.AL_PITCH, 1.0f);
  al.alSourcef(source[0], AL.AL_GAIN, 1.0f);
  al.alSource3f(source[0], AL.AL_POSITION, 0.0f, 0.0f, 0.0f);
                    // position the source at the origin
  al.alSource3i(source[0], AL.AL_VELOCITY, 0, 0, 0);  // no velocity
  if (toLoop)
    al.alSourcei(source[0], AL.AL_LOOPING, AL.AL_TRUE);  // looping
  else
    al.alSourcei(source[0], AL.AL_LOOPING, AL.AL_FALSE); //play once

  if (al.alGetError() != AL.AL_NO_ERROR) {
    System.out.println("Error configuring source for " + nm);
    return null;
  }

  return source;
} // end of initSource()
```

An empty source is created with AL.alGenSources(), and its various attributes are set via calls to AL.alSourceXX(). The most important is the AL.AL_BUFFER attribute which links the source to the buffer.

initSource() returns an array holding the source reference, or null.

Figure 4 shows the source located at the origin, as viewed from above.

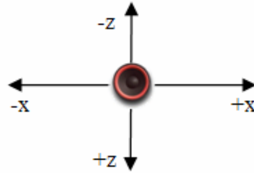

Figure 4. A Source in 3D Space.

The point source emits sound in every direction.

3.5. Positioning a Source

A source is easily moved by changing the (x, y, z) coordinate in its AL.AL_POSITION attribute. setPos() does the task:

```
// globals
private HashMap<String, int[]> sourcesMap;


public boolean setPos(String nm, float x, float y, float z)
// move the nm sound to (x,y,z)
{
  int[] source = (int[]) sourcesMap.get(nm);
  if (source == null) {
    System.out.println("No source found for " + nm);
    return false;
  }

  al.alSource3f(source[0], AL.AL_POSITION, x, y, z);
  return true;
}  // end of setPos()
```

setPos(), and the other source-related methods, use the sound's name as a key into sourcesMap. The retrieved source has its position adjusted.

Positioning a source is such a common task that JOALSoundMan offers a variant of load() which employs setPos().

```
public boolean load(String nm, float x, float y, float z,
                                   boolean toLoop)
{ if (load(nm, toLoop))
    return setPos(nm, x, y, z);
  else
    return false;
}
```

3.6. Playing, Stopping, and Pausing a Source

A source can be played, stopped, and paused with AL.alSourcePlay(), AL.alSourceStop(), and AL.alSourcePause(). The JOALSoundMan methods for playing, stopping, and pausing a source are all similar: they use the sound's name to find the source, then call the relevant AL method. For example, JOALSoundMan.play():

```
public boolean play(String nm)
{
  int[] source = (int[]) sourcesMap.get(nm);
  if (source == null) {
    System.out.println("No source found for " + nm);
    return false;
  }

  System.out.println("Playing " + nm);
  al.alSourcePlay(source[0]);
  return true;
}  // end of play()
```

Calling play() on a stopped source will restart it, but resumes a paused source.

3.7. Moving the Listener

It's useful to be able to move the listener in different ways: either with an (x, z) step added to the listener's current position, or by supplying it with an entirely new (x, z) location. Neither approach changes the listener's y-axis position (at 0).

moveListener() performs the step-based move by utilizing the positional method, setListenerPos(), to do most of the work.

```
// globals
private float xLis, yLis, zLis;    // listener's current position


public void moveListener(float xStep, float zStep)
// move the listener by a (x,z) step
{
  float x = xLis + xStep;
  float z = zLis + zStep;
  setListenerPos(x, z);
}  // end of moveListener()
```

 Andrew Davison © 2007

Changing the listener's position also requires an update to its orientation. The trick is to calculate the x- and z- axis offsets of the listener's move, and apply them to the "look at" point. This is illustrated in Figure 5.

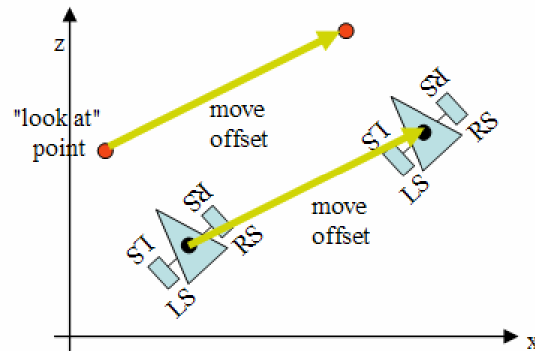

Figure 5. Moving the Listener and its "look at" Point.

The moveListener() method:

```
public void setListenerPos(float xNew, float zNew)
// position the listener at (xNew,zNew)
{
  float xOffset = xNew-xLis;
  float zOffset = zNew-zLis;

  xLis = xNew;  zLis = zNew;
  al.alListener3f(AL.AL_POSITION, xLis, yLis, zLis);

  /* keep the listener facing the same direction by
     moving the "look at" point by the (x,z) offset */
  oriLis[0] += xOffset;
  oriLis[2] += zOffset;
     // no change needed to y-coord in oriLis[1]
  al.alListenerfv(AL.AL_ORIENTATION, oriLis, 0);
}  // end of setListenerPos()
```

There's no need to manipulate y-axis values, since the listener only moves over the XZ plane.

 Andrew Davison © 2007

3.8. Turning the Listener

Turning the listener is complicated by having to turn the “look at” point as well. However, the calculations are simplified by only permitting the listener to rotate around the y-axis.

Figure 6 shows what happens when the listener rotates by angleLis degrees.

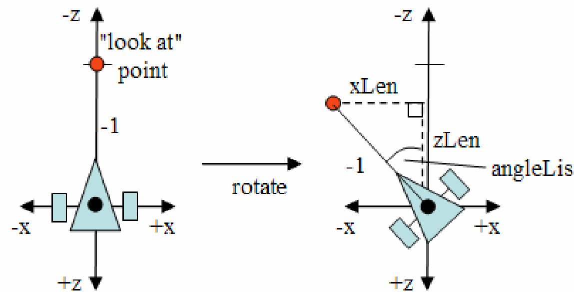

Figure 6. Rotating the Listener and the “look at” Point.

xLen and zLen are added to the listener’s position to get the new “look at” point. This approach is implemented in turnListener():

```
// globals
private float xLis, yLis, zLis;    // current position
private float[] oriLis;   // orientation
private int angleLis = 0;


public void turnListener(int degrees)
// turn the listener anti-clockwise by degrees amount
{
  angleLis += degrees;

  double angle = Math.toRadians(angleLis);
  float xLen = -1.0f * (float) Math.sin(angle);
  float zLen = -1.0f * (float) Math.cos(angle);

  /* face in the (xLen, zLen) direction by adding the
     values to the listener position */
  oriLis[0] = xLis+xLen; oriLis[2] = zLis+zLen;
  al.alListenerfv(AL.AL_ORIENTATION, oriLis, 0);
}  // end of turnListener()
```

angleLis is a global storing the listener’s total rotation away from its starting direction along the -z axis. The user supplies an angle change which is added to angleLis.

4. Using JOALSoundMan

In the rest of this chapter, I'll go through several small examples showing how JOALSoundMan can move a source, and translate and rotate the listener. None of these use 3D (or 2D) graphics, although the last one utilizes a simple GUI.

I also use JOALSoundMan in a Java 3D example in the next chapter, and with JOGL in chapter 17 ??.

4.1. Moving a Source

In MovingSource.java, a source is placed at (0,0,0), at the same spot as the listener (which by default starts at the origin and faces along the -z axis). Gradually the source is moved along the -z axis, away from the listener, causing the repeating sound to fade away. Figure 7 shows the situation diagrammatically.

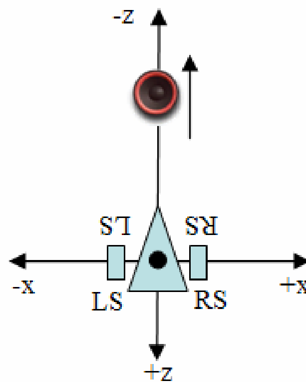

Figure 7. Moving the Source.

The complete MovingSource.java program:

```
public class MovingSource
{
  public static void main(String[] args)
  {
    if (args.length != 1) {
      System.out.println("Usage: runJOAL MovingSource <WAV name>");
      System.exit(1);
    }
    String soundName = args[0];

    JOALSoundMan soundMan = new JOALSoundMan();
    // the listener is at (0,0,0) facing along the -z axis

    if (!soundMan.load(soundName, true))
      System.exit(1);
    // default position for sound is (0,0,0)
    soundMan.play(soundName);

    // move the sound along the -z axis
    float step = 0.1f;
    float zPos = 0.0f;
```

```
    for(int i=0; i < 50; i++) {
      zPos -= step;
      soundMan.setPos(soundName, 0, 0, zPos);
      try {
        Thread.sleep(250);    // sleep for 0.25 secs
      }
      catch(InterruptedException ex) {}
    }

    // soundMan.stop(soundName);
    soundMan.cleanUp();
  }  // end of main()

} // end of MovingSource class
```

MovingSource must be supplied with the name of a WAV file (e.g. FancyPants.wav):

```
> runJOAL MovingListener FancyPants
```

JOALSoundMan.load() loads the sound from the Sounds/ subdirectory, and positions the source at the origin. JOALSoundMan.play() starts it playing, and the while loop gradually moves it with repeated calls to JOALSoundMan.setPos().

After the loop finishes, the sound could be stopped with JOALSoundMan.stop(), but JOALSoundMan.cleanUp() does that anyway.

4.2. Moving the Listener

Another way of making a source fade away is to leave it alone and move the listener instead. This is demonstrated by MovingListener.java, which starts with the same configuration as MovingSource.java (the source and listener both at the origin), but incrementally shifts the listener along the +z axis. This is shown in Figure 8.

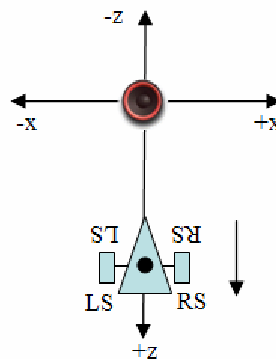

Figure 8. Moving the Listener.

The main() method for MovingListener.java:

```
public static void main(String[] args)
{
  if (args.length != 1) {
    System.out.println("Usage: runJOAL MovingListener <WAV name>");
    System.exit(1);
```

```
  }
  String soundName = args[0];

  JOALSoundMan soundMan = new JOALSoundMan();
  // the listener is at (0,0,0) facing along the -z axis

  if (!soundMan.load(soundName, true))
    System.exit(1);
  // default position for sound is (0,0,0)
  soundMan.play(soundName);

  // move the listener along the z axis
  for(int i=0; i < 50; i++) {
    soundMan.moveListener(0, 0.1f);
    try {
      Thread.sleep(250);    // sleep for 0.25 secs
    }
    catch(InterruptedException ex) {}
  }
  soundMan.cleanUp();
}  // end of main()
```

The only difference from MovingSource.java is that the listener is translated in 0.1 steps along the +z axis by calling JOALSoundMan.moveListener().

4.3. Moving the Listener Between Sources

MovingListener2.java translates the listener away from one source towards another; Figure 9 shows most of the details.

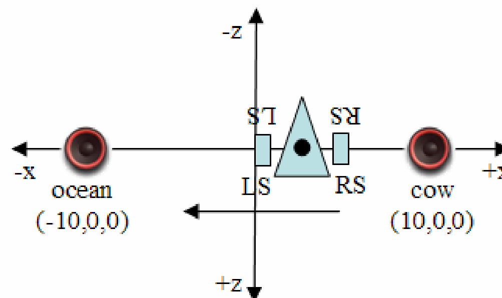

Figure 9. Moving the Listener Between Sources.

The “cow” source repeatedly moos at (10,0,0), and the listener starts at the same spot. It then travels incrementally along the x-axis towards (-10,0,0) where an “ocean” source is playing.

The mooing will initially be the loudest sound coming from the speakers, but will slowly fade way, being replaced by the increasingly louder “ocean”. The orientation of the listener means that both ears receive the same amount of sound, so the fade-out of the cow and fade-in of the ocean are the same for both speakers.

The main() method of MovingListener2.java:

```
public static void main(String[] args)
```

```
{
  JOALSoundMan soundMan = new JOALSoundMan();

  float xPos = 10.0f;
  soundMan.moveListener(xPos, 0);
  // the listener is at (xPos,0,0) facing along the -z axis

  // cow at (xPos,0,0)
  if (!soundMan.load("cow", xPos,0,0, true))
    System.exit(1);
  soundMan.play("cow");

  // ocean at (-xPos,0,0)
  if (!soundMan.load("ocean", -xPos,0,0, true))
    System.exit(1);
  soundMan.play("ocean");

  // move the listener from cow to ocean
  float xStep = (2.0f * xPos)/40.0f;
  for(int i=0; i < 40; i++) {
    soundMan.moveListener(-xStep, 0);
    try {
      Thread.sleep(250);   // sleep for 0.25 secs
    }
    catch(InterruptedException ex) {}
  }
  soundMan.cleanUp();
}  // end of main()
```

4.4. Turning the Listener

This example examines how the listener's orientation affects the speakers output.

As shown in Figure 10, the listener begins by facing along the -z axis as usual, but positioned at (1,0,0). The "FancyPants" sound plays repeatedly off to its right. Then the listener is slowly rotated in an anti-clockwise direction in a full circle.

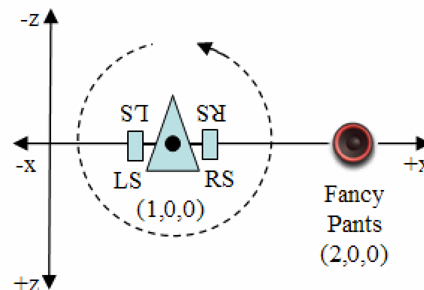

Figure 10. Rotating the Listener.

The effect on the audio is more interesting than previously because the balance of sound changes between the left and right ears (the left and right speakers).

"FancyPants" is initially loudest for the right ear/speaker (when the listener's angle is 0). As the listener turns, the volume decreases in the right ear and increases in the left,

until at 90 degrees, the sound is the same from both speakers. As the listener continues turning towards 180 degrees, the sound in the right ear decreases almost to nothing, and becomes louder in the left ear. At 180 degrees, the left ear is nearest to the source and the sound is at its loudest there. Thereafter, the volume starts increasing again in the right ear. At 270 degrees, the listener is facing the source, and the sound has the same volume from both speakers once again. After that, the left speaker gets quieter until the listener returns to its starting orientation at 360 degrees.

The main() function of TurningListener.java:

```
public static void main(String[] args)
{
  if (args.length != 1) {
    System.out.println("Usage: runJOAL TurningListener <WAV name>");
    System.exit(1);
  }
  String soundName = args[0];

  JOALSoundMan soundMan = new JOALSoundMan();
  // the listener is at (0,0,0) facing along the -z axis
  soundMan.moveListener(1,0);   // now at (1,0,0)

  if (!soundMan.load(soundName, 2,0,0, true))    // at (2,0,0)
    System.exit(1);
  soundMan.play(soundName);

  // rotate listener anti-clockwise
  for(int i=0; i < 60; i++) {
    soundMan.turnListener(6);  // 6 degrees each time
    try {
      Thread.sleep(250);   // sleep for 0.25 secs
    }
    catch(InterruptedException ex) {}
  }
  soundMan.cleanUp();
}  // end of main()
```

The only new coding feature here is the call to JOALSoundMan.turnListener() inside the loop.

 Andrew Davison © 2007

4.5. JOAL and Swing

There's nothing preventing JOAL from being used in Java applications with GUIs. The SingleSource.java example plays, suspends and stops a JOAL source using Swing buttons. Figure 11 shows the GUI.

Figure 11. The SingleSource Application.

The SingleSource() constructor loads the source with JOALSoundMan, and sets up the GUI.

```
// globals
private JOALSoundMan soundMan;
private String soundName;


public SingleSource(String nm)
{
  super("Single Static Source");

  soundMan = new JOALSoundMan();
  soundName = nm;

  if (!soundMan.load(soundName, true))
    System.exit(1);

  buildGUI();

  addWindowListener( new WindowAdapter() {
    public void windowClosing(WindowEvent e)
    { soundMan.cleanUp();
      System.exit(0);
    }
  });

  pack();
  setResizable(false);    // fixed size display
  setVisible(true);
}  // end of SingleSource()
```

The window closing event is caught so that JOALSoundMan.cleanUp() can be called before termination.

buildGUI() sets up three buttons, whose action listeners are the class itself. When any of the buttons are pressed, SingleSource's actionPerformed() method is called.

```
// globals
private JButton playButton, stopButton, pauseButton;


public void actionPerformed(ActionEvent e)
```

 Andrew Davison © 2007

```
{
  if (e.getSource() == playButton)
    soundMan.play(soundName);
  else if (e.getSource() == stopButton)
    soundMan.stop(soundName);
  else if (e.getSource() == pauseButton)
    soundMan.pause(soundName);
}  // end of actionPerformed
```

The relevant play(), stop(), or sound() method is called in JOALSoundMan.

5. Other Source Types

As I mentioned earlier, Java 3D includes three sound-related classes: BackgroundSound for ambient sound, PointSound for a sound source located at a particular place in the scene, and ConeSound for a point source aimed in a specific direction. All of these can be implemented using JOAL, as David Grace's JoalMixer illustrates (https://j3d-incubator.dev.java.net/).

JOALSoundMan creates point sound sources, so how much work is needed for it to support ambient and cone sounds?

5.1. Ambient Sounds

For ambient sounds, JOALSoundMan can utilize an OpenAL 'feature': stereo WAV files will not be played positionally. This means that calls to JOALSoundMan.setPos() for a stereo source will have no effect on the sound coming from the speakers. This can be tested by calling the MovingListener.java example with a stereo WAV file:

```
> runJOAL MovingListener FancyPantsS
```

FancyPantsS.wav is a stereo version of the mono FancyPants.wav, which I created using the WavePad audio editing tool (http://www.nch.com.au/wavepad/).

Another way of playing sounds ambiently is to use Java's own sound classes (e.g. AudioClip), or perhaps the SoundsPlayer class I developed in chapter 12 ??, since they don't use positional effects.

A third approach is to 'connect' the source to the listener, so that it moves as the listener does. Then the source's volume and speaker distribution will stay the same no matter where the listener moves to.

The AL.AL_SOURCE_RELATIVE attribute changes the source's position to be *relative* to the listener. For example:

```
al.alSourcei(source[0], AL.AL_SOURCE_RELATIVE, AL.AL_TRUE);
al.alSource3f(source[0], AL.AL_POSITION, 0.0f, 0.0f, 0.0f);
```

The source will always be at the same spot as the listener.

Andrew Davison © 2007

5.2. Cone Sounds

A cone sound is a point source with four additional parameters. A typical cone is shown in Figure 12.

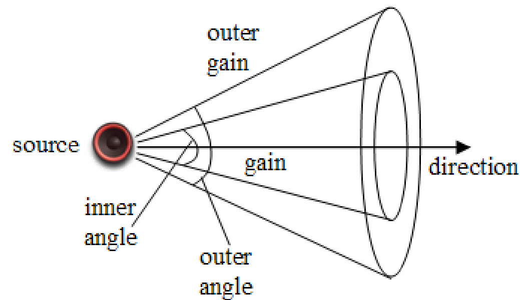

Figure 12. A Cone Sound.

Two cones are defined in terms of inner and outer angles around a central direction vector. The inner cone plays the sound using the AL.AL_GAIN volume setting (often 1.0f), but the volume tails off in the outer cone, decreasing to the AL.AL_CONE_OUTER_GAIN value (typically 0.0f) at the outer cone's edges.

The following code snippet shows how the source's attributes are set:

```
al.alSourcef(source[0], AL.AL_CONE_INNER_ANGLE, innerAngle);
al.alSourcef(source[0], AL.AL_CONE_OUTER_ANGLE, outerAngle);
al.alSourcef(source[0], AL.AL_GAIN, 1.0f);
al.alSourcef(source[0], AL.AL_CONE_OUTER_GAIN, 0.0f);
al.alSource3f(source[0], AL.AL_DIRECTION, x, y, z);
```

innerAngle and outerAngle are specified in degrees (e.g. 30.0f and 45.0f). The (x, y, z) direction should be a vector, such as (0,0,1) to make the cone point along the +z axis.

6. Summary

The chapter has discussed JOAL, a Java wrapper around OpenAL, a popular library for programming with 2D and 3D audio.

I developed a JOALSoundMan class which hides the low-level details of buffer, source, and listener creation, and I demonstrated its use with several simple examples.

JOALSoundMan will appear again in the next chapter when I employ it to add 3D sound to a Java 3D example, and in chapter 17 ?? when it does something similar for a JOGL program.

Andrew Davison © 2007