3D Sound with SoundSystem

About SoundSystem

The SoundSystem library was created to provide a simple, common interface to a variety of 3rd party sound and codec libraries, and to simplify switching between them on the fly. It is designed to make playing sound effects in 3D space very simple.

SoundSystem is designed to be expandable. Additional sound and codec library plug-ins can be added with relatively little effort.

There are some important terms which I will use throughout this guide, so before I begin, I will go over them now.

- A sample is the sound contained in an audio file.
- A buffer stores the sound bytes for a sample. It may contain the entire sound, or a smaller chunk of data.
- A source is a point in 3D space which emits a sound. A source is not the sample itself, but rather the location where the sample is played.
- A point-source changes as its position in 3D space changes. Close point-sources are louder than those which are further-away. Point-sources to the right or left play louder out of the respective speaker.
- An ambient-source plays at a constant volume regardless of its position in 3D space. Ambient-sources are used for things like background music.
- A streaming source plays a sample in small pieces rather than all at once. Streaming sources are useful for really long samples. A streaming source can be either an ambient-source or point-source.
- Stream-buffers contain small sequential chunks of a sample which are queued to play on a streaming source.
- A channel is the link to an underlying sound library (such as OpenAL or JavaSound), through which sources are played
- The listener is the point in 3D space where sources are heard from. The listener is able to move around in 3D space, and the orientation can be changed by defining the look-at direction and the up-direction.
- Attenuation is how a source's volume fades with distance.
- Rolloff attenuation is a realistic attenuation model, which uses a rolloff-factor to determine how quickly a source's volume fades. A higher rolloff-factor causes a source to fade more rapidly, and a source with zero attenuation never fades (i.e. it is an ambient-source).
- Linear attenuation is less realistic than rolloff-attenuation, but it allows the user to specify a maximum fade-distance, beyond which a source's volume is zero.

Getting Started

Let’s begin by learning how the core SoundSystem library works. It can be downloaded for free at http://www.paulscode.com. While you are at the paulscode website, you will also see various expansions of SoundSystem (such as SoundSystemJPCT), which are designed to make using the library even easier. These expansions automatically link with common external plug-ins, and they provide a whole bunch of extra methods to make source creation easier than using the core SoundSystem library alone, by automatically using default settings any time a parameter is not specified. However, I believe it is important to learn how to use the core library itself first before you will be able to take full advantage of all the great features. Since all the expansions extend the base SoundSystem class, everything I go over here can be applied to them as well.

I will start with a basic demo application. The complete source code and resources for all the tutorials in this book can be downloaded from the paulscode website. For this application, you will need the core library JAR (SoundSystem.jar) and a MIDI music file (beethoven.mid is used in this example).

By default, your sound file must be compiled into the application JAR in a package named “Sounds”, so be sure to place beethoven.mid there.

First, import the library at the top of your program:

```
import paulscode.sound.SoundSystem;
```

Initializing the SoundSystem is simple – just create a SoundSystem object:
Important Note: There may only be one SoundSystem instance loaded at a time in your program.

```
SoundSystem mySoundSystem = new SoundSystem();
```

At this point, there are several methods you could use for creating a source, but the ‘backgroundMusic’ method is probably best suited for music. This method automatically creates and begins playing an ambient source, which is what you would normally want for background music. I will go into the other source creation methods in much more detail later. For now, we will create a new ambient source called “Cool Music”, which will loop a file called “beethoven.mid”:

```
mySoundSystem.backgroundMusic( "Cool Music", "beethoven.mid", true );
```

“Cool Music” is now the name that can be used to refer to this new source. This parameter may be any non-empty string. It is important to note that no two sources are allowed to have the same name. Using “true” for the “toLoop” parameter means that when you play the source, it will keep repeating over and over until it is stopped. If you use “false” for this parameter instead, the source would play only once then stop.

When finished with the SoundSystem, it is important to call the cleanup() method. This shuts down whatever sound library has been loaded and removes references to all instantiated objects used by the SoundSystem:

```
mySoundSystem.cleanup();
```

This is very important, because it shuts down whatever library plug-in is active, and frees any resources it is using. Failure to call the cleanup() method when using OpenAL may cause rogue audio artifacts to play continuously on some audio hardware, even after an application has been shut down, forcing the user to reboot. It has even been known to cause some machines to crash, so don’t forget to call cleanup() when you are done with the sound system!

Example Application #1

(Creates the SoundSystem, plays some music, then shuts down)

```
// Author: Paul Lamb (http://www.paulscode.com)
import paulscode.sound.SoundSystem;

/**
 *  Creates the SoundSystem, plays some music, then shuts down.
 */
public class Example_1
{
    public static void main( String[] args )
    {
        new Example_1();
    }

    public Example_1()
    {
        // Instantiate the SoundSystem:
        SoundSystem mySoundSystem = new SoundSystem();

        // Create a new ambient, looping source called "Cool Music":
        mySoundSystem.backgroundMusic( "Cool Music", "beethoven.mid", true );

        // Wait for 10 seconds:
        sleep( 10 );

        // Shut down:
        mySoundSystem.cleanup();
    }

    public void sleep( long seconds )
    {
        try
        {
            Thread.sleep( 1000 * seconds );
        }
        catch( InterruptedException e )
        {}
    }
}
```

When running this application, you might have noticed the following output:

```
Starting up SoundSystem...
Initializing No Sound
    (Silent Mode)
```

This means that we have not linked with any external sound library plug-ins, so SoundSystem is unable to play any sounds. So why were we able to hear the music then, you may wonder? It’s because we used a MIDI file. As with most other sound libraries, SoundSystem makes a clear distinction between MIDI and everything else (thus, “No Sound, Silent Mode” refers to everything except MIDI). MIDI music is played in a different way than other sources, and it doesn’t require a sound library or codec plug-in to work. Therefore, the core SoundSystem library without any additional plug-ins is capable of playing MIDI files (with the extension ‘.mid’ or ‘.midi’), but nothing else. MIDI files are extremely small compared to other sound formats, so they can be an excellent option if memory is a concern (such as in an applet).

While we are on the subject of MIDI, I should mention that it has one important limitation: Only one MIDI file can be played at a time. This is a limitation of the MIDI synthesizers available in Java, and cannot be avoided until (or if) I write my own synthesizer. In order to enforce this basic rule, SoundSystem will only let you have one MIDI source in your program. If you create additional MIDI sources, SoundSystem will simply remove the previous ones, ensuring that only one exists at a time.

One final note about the MIDI source: because it is special, it is not classified as “normal” or “streaming” and it plays through its own special channel. So creating a MIDI source using the newSource method will have the same effect as using the newStreamingSource method to create it.

Adding Plug-ins

Source code in this chapter uses the LibraryJavaSound and CodecWav plug-ins, which can be downloaded from http://www.paulscode.com.

As mentioned in the previous chapter, the core SoundSystem library without any plug-ins is only capable of playing MIDI files. It would be pretty lame if that were the only thing we could ever do. Fortunately, this is where the concept of plug-ins comes in to play. Plug-ins give the SoundSystem library access to various 3rd party libraries.

The first type of plug-in is the codec. Codec plug-ins tell SoundSystem how to read data from various file formats, such as .wav or .ogg. You will generally need a codec plug-in for each format you want to be able to play. Each codec will have its own advantages and disadvantages, so it is generally a good idea to choose codecs based on the needs of a particular project.

The second type of plug-in is the Library. Library plug-ins provide the SoundSystem with an interface for accessing various sound libraries, such as LWJGL OpenAL or Java Sound. You must use a Library plug-in before SoundSystem will be able to produce any sound. You will generally need one Library plug-in for each library you want to use. Linking with more than one Library plug-in is generally a good idea, so there will be a backup option in case one library isn't compatible on the end user's computer.

There is one important thing to keep in mind when choosing plug-ins for your project. They generally have their own individual license agreements, separate from the SoundSystem License, so be sure to read their documentation to learn about any restrictions involved with using them.

To link with the plug-ins, you'll first need to import the required classes at the top of your program:

```
import paulscode.sound.SoundSystemConfig;
import paulscode.sound.SoundSystemException;
import paulscode.sound.libraries.LibraryJavaSound;
import paulscode.sound.codecs.CodecWav;
```

To link with the LibraryJavaSound plug-in:

```
try
{
    SoundSystemConfig.addLibrary( LibraryJavaSound.class );
}
catch( SoundSystemException e )
{
    System.err.println( "error linking with the LibraryJavaSound plug-in" );
}
```

The above method is pretty self explanatory. You are just telling SoundSystem about the library plug-in. If there was some problem linking with the plug-in, a SoundSystemException will be thrown. You can add multiple library plug-ins by calling the SoundSystemConfig.addLibrary() method for each one. You should always link with the library plug-ins at the top of your program before you instantiate the SoundSystem.

To link with the CodecWav plug-in:

```
try
{
    SoundSystemConfig.setCodec( "wav", CodecWav.class );
}
catch( SoundSystemException e )
{
    System.err.println("error linking with the CodecWav plug-in" );
}
```

The SoundSystemConfig.setCodec() method is used to associate a file format with the codec used to read data from it. So in the above example, we are telling SoundSystem to use the CodecWav plug-in to read from any files with the extension “wav”.

(there is no example application for this chapter)

Choosing a Sound Library

Source code in this chapter uses the LibraryLWJGLOpenAL and LibraryJavaSound plug-ins, which can be downloaded from http://www.paulscode.com.

As I mentioned in the last chapter, an important feature of SoundSystem is its ability to link with a variety of different 3rd party sound libraries. An important thing to remember when choosing library plug-ins is that not every sound library is compatible on every computer. The most compatible library plug-in is LibraryJavaSound, but it is not as nice as LibraryLWJGLOpenAL. The sample code in this chapter uses the LibraryLWJGLOpenAL and LibraryJavaSound plug-ins, which can be downloaded from the paulscode website.

SoundSystem has a method for checking whether a particular library is compatible on a user’s computer:

```
aLCompatible = SoundSystem.libraryCompatible( LibraryLWJGLOpenAL.class );
jSCompatible = SoundSystem.libraryCompatible( LibraryJavaSound.class );
```

After determining which libraries are compatible, one can be loaded:

```
Class libraryType;

if( aLCompatible )
    libraryType = LibraryLWJGLOpenAL.class;   // OpenAL
else if( jSCompatible )
    libraryType = LibraryJavaSound.class;  // Java Sound
else
    libraryType = Library.class;  // "No Sound, Silent Mode"

try
{
    mySoundSystem = new SoundSystem( libraryType );
}
catch( SoundSystemException sse )
{
    // Shouldn’t happen, but it is best to prepare for anything
    sse.printStackTrace();
    return;
}
```

If compatibility checking seems too complicated, don’t worry. If you use the first SoundSystem constructor without parameters as we did in the first example application, it is not necessary to check for library compatibility at all. SoundSystem has its own built-in compatibility checking, and it will attempt to initialize the first library plug-in, and if that fails it will try the next one, and so on. If none of them are compatible, it uses “No Sound, Silent Mode” as a last resort.

So if we want SoundSystem to try OpenAL first, and use JavaSound as a backup option, simply add the plug-ins in that order:

```
SoundSystemConfig.addLibrary( LibraryLWJGLOpenAL.class );
SoundSystemConfig.addLibrary( LibraryJavaSound.class );
```

It is also simple to switch between libraries on the fly:

```
try
{
    mySoundSystem.switchLibrary( LibraryJavaSound.class );
}
catch( SoundSystemException sse )
{
    // Java Sound was not compatible on the user’s machine
    sse.printStackTrace();
    return;
}
```

You can obtain information about which library is currently loaded:

```
Class currentLibrary = SoundSystem.currentLibrary();

String title = SoundSystemConfig.getLibraryTitle( currentLibrary );
String description = SoundSystemConfig.getLibraryDescription( currentLibrary );
```

(there is no example application for this chapter)

Sources

Source code in this chapter uses the LibraryLWJGLOpenAL, LibraryJavaSound, CodecWav, and CodecJOgg plug-ins, which can be downloaded from http://www.paulscode.com. A .wav and a .ogg file are also needed (explosion.wav and beats.ogg are used) for the example application.

There are two types of sources: normal and streaming (technically there is a third type used specifically for MIDI, as I mentioned in an earlier chapter). The reason there are two different types of sources is that typically there is a maximum sample size that can be played at one time (for Java Sound it is usually between 5 – 15 seconds). Some audio files need to be longer than that (such as background music). What SoundSystem does for these longer files is “stream” the audio information to a channel in chunks. This is done on a single separately running thread which handles all active streaming sources. All this is done in the background, so you don’t have to worry too much about the details involved with streaming, just that longer sounds should be used with streaming sources instead of normal ones.

By default, there are 4 streaming channels. For most soundcards this leaves 28 normal channels. Since all sources are played through channels, this means that there can be up to 4 streaming sources and 28 normal sources playing at one time. Normally, it is not advisable to change these numbers, since there are several different problems that can arise if these values are set improperly. If there is a good reason to change these values, it can be done using two static methods found in the SoundSystemConfig class:

```
SoundSystemConfig.setNumberStreamingChannels( 5 );  // up to 5 streams at once
SoundSystemConfig.setNumberNormalChannels( 27 );
```

In most cases, you should be safe as long as the total number of streaming channels plus normal ones adds up to 32. If they add up to more than 32, Java Sound may throw an exception when too many sources try to play at once. Calling the above two methods should only be done BEFORE creating the SoundSystem object. It is important to note that Sound System will only create as many channels as the user’s sound card will allow, so setting the channel numbers to a particular value does not guarantee that there will always be that many channels available on every computer.

Sources are further divided into two categories: priority or non-priority. The difference between the two is that a priority source will not be overwritten by another source if there are not enough channels.

For example, let’s say there are 4 streaming channels, and the background music is a priority source. Now say 4 additional non-priority streaming sources try to play at the same time, the first three will fill up the remaining 3 streaming channels. To play the last source, one streaming source will have to be stopped to free up a channel. Since the background music was a priority source, it will be left alone, and instead one of the non-priority sources will be stopped instead.

One important thing to note is that if you make all sources priority sources, then they will never be stopped to make room for newer sources to play. This could result in some sources never being played, so it is a generally good idea to only use priority sources for things that absolutely must never be cut off (like background music, streaming speech, and things like that.)

Sources can be further divided into looping and non-looping sources. A looping source will keep playing over and over again until it is told to stop, while a non-looping source will only play once. Looping sources are generally used for things like background music and environment sounds like crickets and such. Most sources are going to be non-looping, and will play once when a certain event happens (such as an explosion).

The final attribute that all sources have is an attenuation model. Identifiers for the three different attenuation models are found in the SoundSystemConfig class:

ATTENUATION_NONE
ATTENUATION_ROLLOFF
ATTENUATION_LINEAR

The first is "no attenuation". As indicated in the first chapter, a source with no attenuation is called an ambient source. Ambient sources will sound the same regardless of how far away they are. Ambient sources are good to use for things like background music.

The second attenuation model is called "rolloff attenuation". This is the most realistic attenuation model. When using this attenuation model, you can choose a "rolloff-factor" for each source. This is a float value between 0 and 1. The lower the number, the longer it takes for a source to fade with distance. If the rolloff factor is set to zero, the source will be an ambient source.

The last attenuation model is called "linear attenuation". This attenuation model is not as realistic as rolloff attenuation, but it can be very useful in some cases. For example, if you want to ensure that a source's volume will reach zero at a certain distance, then this is the attenuation model to use. In linear attenuation, you can choose a "fade distance" – the distance at which a source becomes silent.

Now that we have gone over all the terminology used for creating sources, let's look at the newSource method:

```
boolean priority = false;
String sourcename = "Source 1";
String filename = "explosion.wav";
boolean loop = false;
float x = 0;
float y = 0;
float z = 0;
int aModel = SoundSystemConfig.ATTENUATION_ROLLOFF;
float rFactor = SoundSystemConfig.getDefaultRolloff();

mySoundSystem.newSource( priority, sourcename, filename, loop, x, y, z, aModel,
                         rFactor );
```

In the example above, I defined all the parameters ahead of time, so that it is easy to see where they go in the newSource method. You will also notice that I am using a method for obtaining the default value for "rolloff factor". Other methods for obtaining and changing default settings can be found in the SoundSystemConfig class.

Another method to use for creating sources is the newStreamingSource method. The parameters for this method are exactly the same as the ones for the newSource method:

```
mySoundSystem.newStreamingSource( priority, sourcename, filename, loop, x, y,
                                  z, aModel, rFactor );
```

Since playing background music is so common, the SoundSystem class has a method (as we saw in the first example application) specifically designed for making this quick and easy, called backgroundMusic(),:

```
mySoundSystem.backgroundMusic( "Music 1", "beats.ogg", true );
```

The above line of code automatically creates a streaming, looping, priority source and starts playing it.

The last two types of methods for creating new sources are quickPlay and quickStream. These methods automatically generate a name for the new source, and immediately start playing. Additionally, they set the new source to "temporary", which means it will be automatically removed after it finishes playing. These methods are designed to eliminate the headache of trying to keep track of numerous sources. They make it easy to simply play a sound effect at a particular 3D position, and forget about it.

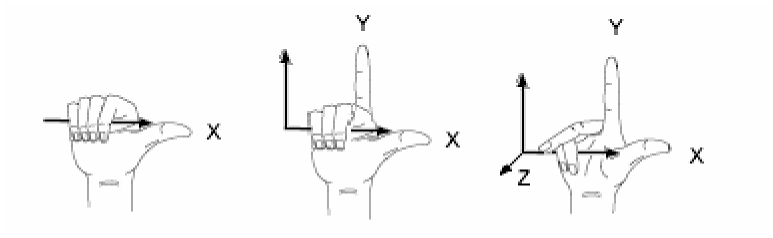
```
boolean priority = false;
String filename = "explosion.wav";
boolean loop = false;
float x = 0;
float y = 0;
float z = 0;
int aModel = SoundSystemConfig.ATTENUATION_ROLLOFF;
float rFactor = SoundSystemConfig.getDefaultRolloff();

mySoundSystem.quickPlay( priority, filename, loop, x, y, z, aModel, rFactor );
```

These methods take similar parameters to newSource and newStreamingSource.

Additionally, any source can be made into a temporary or non-temporary source by simply calling the setTemporary method:

```
mySoundSystem.setTemporary( sourcename, false );
```

One final note: SoundSystem uses the right-handed Cartesian 3D coordinate system commonly seen in geometry textbooks and the like:



As you can see from the diagram, “up” is +Y, “right” is +X, and “forward” is –Z.

To demonstrate the power and ease of the SoundSystem library, here is an example which plays some background music and explosions in 3D:

Example Application #2

(Plays background music and explosions in 3D)

```
import paulscode.sound.SoundSystem;
import paulscode.sound.SoundSystemConfig;
import paulscode.sound.SoundSystemException;
import paulscode.sound.libraries.LibraryLWJGLOpenAL;
import paulscode.sound.libraries.LibraryJavaSound;
import paulscode.sound.codecs.CodecWav;
import paulscode.sound.codecs.CodecJOgg;

/**
 *  Plays background music and explosions in 3D.
 **/
public class Example_2
{
    public static void main( String[] args )
    {
        new Example_2();
    }

    public Example_2()
    {
        // Load some library and codec plug-ins:
        try
        {
            SoundSystemConfig.addLibrary( LibraryLWJGLOpenAL.class );
            SoundSystemConfig.addLibrary( LibraryJavaSound.class );
            SoundSystemConfig.setCodec( "wav", CodecWav.class );
            SoundSystemConfig.setCodec( "ogg", CodecJOgg.class );
        }
        catch( SoundSystemException e )
        {
            System.err.println("error linking with the plug-ins" );
        }
        // Instantiate the SoundSystem:
        SoundSystem mySoundSystem = new SoundSystem();
        // play some background music:
        mySoundSystem.backgroundMusic( "Music 1", "beats.ogg", true );

        // wait a bit before playing the explosions:
        sleep( 2000 );
        // play 15 explosions, right and left:
        for( int x = 0; x < 15; x++ )
        {
            // If x is divisible by 2, play to the right:
            if( x % 2 == 0 )
                mySoundSystem.quickPlay( false, "explosion.wav", false,
                                         20, 0, 0,
                                         SoundSystemConfig.ATTENUATION_ROLLOFF,
                                         SoundSystemConfig.getDefaultRolloff()
                                        );
            // Otherwise play to the left:
            else
                mySoundSystem.quickPlay( false, "explosion.wav", false,
                                         -20, 0, 0,
                                         SoundSystemConfig.ATTENUATION_ROLLOFF,
                                         SoundSystemConfig.getDefaultRolloff()
                                        );

            // wait a bit so the explosions don't all start at once
            sleep( 125 );
        }
        // Wait a few seconds:
        sleep( 10000 );

        // Shut down:
        mySoundSystem.cleanup();
    }

    public void sleep( long milliseconds )
    {
        try
        {
            Thread.sleep( milliseconds );
        }
        catch( Exception e )
        {}
    }
}
```

Once a source has been created, there are several methods to use to manipulate it. The play, stop, pause, and rewind methods all take the source's name as a parameter:

```
mySoundSystem.play( "Source 1" );
```

The setPosition method is used to move a source around in 3D space:

```
mySoundSystem.setPosition( "Source 1", 20, 0, 0 );
```

Each source's volume can be changed individually (float value between 0 and 1):

```
mySoundSystem.setVolume( "Source 1", 0.5f );
```

And the "master volume can be changed (affecting all sources including MIDI):

```
mySoundSystem.setMasterVolume( 0.5f );
```

Important Note: Setting a volume to zero should mean completely silent. However for about 50% of users the volume of the MIDI source can never become completely silent (it will get quieter, but not silent). Unfortunately, this is a limitation of the synthesizers generally available in Java, and can not be avoided until (or if) I write my own MIDI synthesizer in the future.

Another useful feature of the SoundSystem library is its ability to transition streaming sources (as well as the MIDI source) from one sample to another. SoundSystem has a couple of methods for doing this (note: these methods do not work for normal "non-streaming" sources).

The first sample-transition method is used to queue up one or more samples to play in sequence. SoundSystem will wait until the currently playing sample is finished, then immediately play the next sample in the queue, and so on:

```
mySoundSystem.queueSound( "Cool Music", "musicLoop.mid" );
```

When the last sample in the queue finishes playing, it will either loop that last sample if the source is a looping source, or stop if the source is non-looping. This method is especially useful for music "lead-ins" (a music intro, followed by a portion that loops nicely) to keep background music from awkwardly starting over every time it reaches the end.

The next sample-transition method is used to fade out the currently playing sample, and then begin playing another one:

```
mySoundSystem.fadeOut( "Music 1", "nextsong.ogg", 5000 );
```

The third parameter is the number of milliseconds for the currently playing sample to fade to silent before switching to the new sample. If the second parameter is empty or null, the source will simply stop after fading out. This method is useful for smoothly transitioning between background music. For example different levels of a game world might play different background music, and you don't want to abruptly stop one song to switch to another one.

The final sample-transition method is used to fade out the currently playing sample, and then fade in the next one:

```
mySoundSystem.fadeOutIn( "Music 1", "nextsong.ogg", 5000, 2000 );
```

This method does almost the same thing as the fadeout method, except that the extra parameter lets you specify how many milliseconds to fade the next sample in (rather than starting it at full volume).

Both of the above "fade" methods will remove anything that that was added to the sample queue by the queueSound() method. These methods may be more desirable than the queueSound method when using codecs that tend to have a noticeable lag while transitioning between samples (such as CodecWav and CodecJOgg). Also remember that on some systems, MIDI volume will never become completely silent, which may make these fade methods less desirable for use with MIDI music. **Important Note:** The fade methods may not work well with some library and codec plug-ins (such as LibraryJavaSound and CodecJOrbis), causing the volume to fade in a jerky manner. If you do experience jerky fading, the following method may help correct the problem:

```
mySoundSystem.checkFadeVolumes();
```

This method would normally be called somewhere in the main game loop. To optimize frame-rates, it is probably not necessary to call this method every frame, but rater at some acceptable "granularity" (play around to see what sounds acceptable in a particular situation).

One more thing to mention about sources, is where they get their sample data from. What happens behind the scenes is that the first time the Sound System encounters a filename, it loads the file and sticks the entire sample data into a buffer (unless it is for a streaming source). Every normal source which plays that sound effect will read the data from that buffer, so there are not multiple copies of the sample data floating around and eating up extra memory.

The SoundSystem class has a method called loadSound. It can be used to manually pre-load sample data from audio files. By pre-loading the audio files at one place in a program, this prevents a slight lag that can occur the first time a sound file is encountered which hasn't been loaded yet:

```
mySoundSystem.loadSound( "explosion.wav" );
```

It is important to note that all normal sounds are loaded entirely into memory (streaming sources, of course, are not pre-loaded). Because of this, the SoundSystem class has a method for unloading a sample from memory if it is no longer needed (for example different levels in a game could have their own unique sound effects that don't need to stick around in memory when the player is not in one of those levels):

```
mySoundSystem.unloadSound( "explosion.wav" );
```

The last topic related to sources is MIDI. SoundSystem supports music files in the MIDI format (with the extension '.mid' or '.midi'), and you can specify a MIDI file using any of the source-creation methods mentioned above. It is important to note, however, that MIDI files are handled in a special way. Firstly, only one MIDI source may exist at a time. If you attempt to create more than one MIDI source, the subsequent sources will automatically remove the earlier ones, so there will only ever be one MIDI source at a time. It doesn't matter if you specify the MIDI source as 'normal' or 'streaming', both will behave the same way. The MIDI source plays on its own special type of channel, so playing MIDI does not use up one of the normal or streaming channels.

The Listener

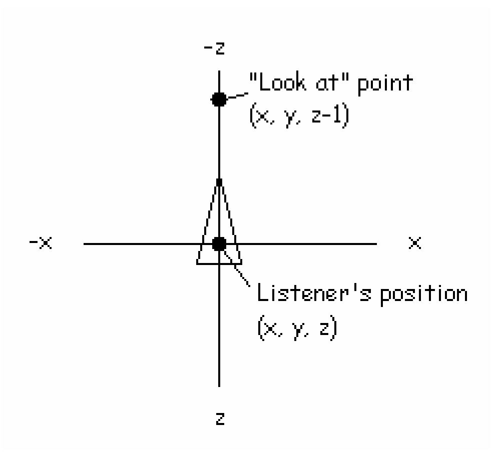
The last important concept to understand about the core SoundSystem Library is the listener. This is probably the most difficult concept to understand. The listener is basically where the player is listening to things from. It is defined by a position in 3D space and an orientation. The position is straight-forward, however orientation can be a bit tricky to understand at first. It consists of two normalized vectors (a normalized vector means that its length is 1). The first vector indicates the "look-at direction" (direction the listener is facing), and the second vector indicates the "up-direction" (is the listener rotated sideways, upside-down, etc.) One way to think about this is to first imagine that your head is located at some coordinates 3D space. Then imagine which direction your head is turned towards. Finally, imagine how far your head is tilted to either side. These are basically the kinds of things which you have to keep in mind when setting the listener's orientation.

The best way to picture the listener's default position and orientation is that you are positioned at the origin (0, 0, 0), facing into the monitor, head straight. The vectors that represent this are:

Position:	(0, 0, 0)
Look-at:	(0, 0, -1)
Up:	(0, 1, 0)

As the listener's position changes, the values for look-at and up remain the same. Imagine yourself walking around the room without turning your body or head. You would always be looking in the same *relative* direction (i.e., you are NOT always looking at the same spot in 3D).

Here is a visual representation of the listener as viewed from above:



The SoundSystem class has two methods for changing the listener's position. The first moves the listener relative to its current position:

```
mySoundSystem.moveListener( 0, 0, -100 );
```

The above command moves the listener 100 units in the –z direction (into the screen), relative to its current position. So if the listener had been located at (10, 10, 300), it would now be located at (10, 10, 200).

The second method sets the listener's actual position in 3D space:

```
mySoundSystem.setListenerPosition( 0, 0, -100 );
```

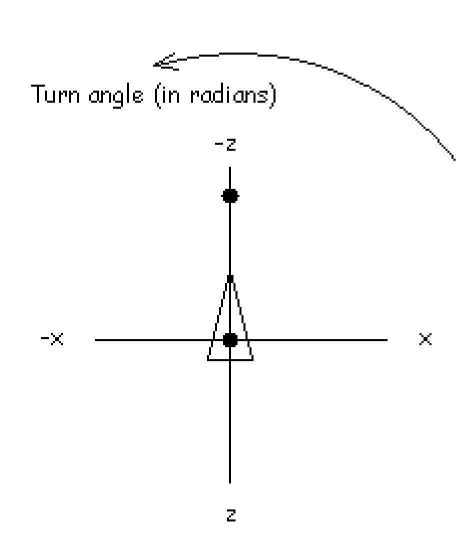
The above command places the listener at position (0, 0, -100).

The listener's orientation can be set using the setListenerOrientation method:

```
mySoundSystem.setListenerOrientation( 0, 0, -1, 0, -1, 0 );
```

The above command turns the listener upside-down, looking into the screen.

One method of moving through 3D space is the "first-person shooter" perspective. Because this is so common, there are two methods specifically designed to make this type of movement easier. Rather than manually changing the look-at direction, they instead allow you to use an angle to turn the listener. The angle is a float value in radians (not degrees). Angles are "counterclockwise around the y-axis":



The first method turns the listener relative to its current angle:

```
mySoundSystem.turnListener( (float) Math.PI );
```

The above command turns the listener pi radians counterclockwise relative to its current angle. So if the listener's angle had been 1.0 radians, it would now be about 4.14 radians.

The second method sets the listeners actual angle:

```
mySoundSystem.setListenerAngle( (float) Math.PI );
```

Example Application #3
(Demonstrates turning the listener)

```
import paulscode.sound.SoundSystem;
import paulscode.sound.SoundSystemConfig;
import paulscode.sound.SoundSystemException;
import paulscode.sound.libraries.LibraryLWJGLOpenAL;
import paulscode.sound.libraries.LibraryJavaSound;
import paulscode.sound.codecs.CodecWav;
import paulscode.sound.codecs.CodecJOgg;

/**
 *  Demonstrates turning the listener.
 **/
public class Example_3
{
    public static void main( String[] args )
    {
        new Example_3();
    }

    public Example_3()
    {

        // Load some library and codec plug-ins:
        try
        {
            SoundSystemConfig.addLibrary( LibraryLWJGLOpenAL.class );
            SoundSystemConfig.addLibrary( LibraryJavaSound.class );
            SoundSystemConfig.setCodec( "wav", CodecWav.class );
            SoundSystemConfig.setCodec( "ogg", CodecJOgg.class );
        }
        catch( SoundSystemException e )
        {
            System.err.println("error linking with the plug-ins" );
        }
        // Instantiate the SoundSystem:
        SoundSystem mySoundSystem = new SoundSystem();

        // play looping explosions to the right:
        mySoundSystem.quickPlay( false, "explosion.wav", true,
                                 20, 0, 0,
                                 SoundSystemConfig.ATTENUATION_ROLLOFF,
                                 SoundSystemConfig.getDefaultRolloff() );

        // Turn the listener around:
```

```
        for( float angle = 0; angle < (float) Math.PI * 2; angle += .02 )
        {
            mySoundSystem.setListenerAngle( angle );
            sleep( 20 );
        }

        // Shut down:
        mySoundSystem.cleanup();
    }

    public void sleep( long milliseconds )
    {
        try
        {
            Thread.sleep( milliseconds );
        }
        catch( Exception e )
        {}
    }
}
```

Unfortunately, as you can imagine, manipulating the listener can be extremely complicated, and the subject is really beyond the scope of this book.

Conclusion

At this point, you should now have a basic understand of the core SoundSystem Library, and you are ready to begin using one of the SoundSystem extensions. Things are about to get a whole lot simpler from this point on. The extensions automatically take care of most of the messy stuff you saw in the above examples, letting you more or less forget about sound and focus on other concepts in your game. When you choose an extension for your project, be sure to read any guides that come with it. If this book made absolutely no sense to you, don't worry. I mainly wrote it to provide a basic overview of the core SoundSystem library, and to be used as a simple reference to use when problems arise. If you have any comments or suggestions about this book or about the SoundSystem library in general, feel free to post a thread on the forums at http://www.paulscode.com.

Legal Concerns

References used in this book:

jPCT
The jPCT API (http://www.jpct.net)

LWJGL
The Lightweight Java Gaming Library (http://www.lwjgl.org)

J-Ogg
The J-Ogg library (http://www.j-ogg.de), copyrighted by Tor-Einar Jarnbjo

J-Orbis
The J-Orbis library (http://www.jcraft.com/jorbis/)

The SoundSystem License: