Sound

Computer Games Development

CSCU9N6

Sound Concepts

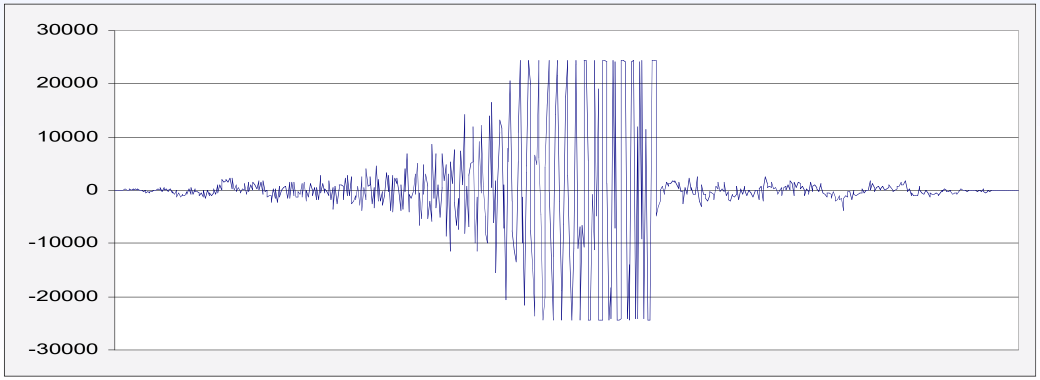
- Sound Generation
- Mixing Sound
- Playing a Sound in Java
- Sound Effects / Filters
- Playing Multiple Sounds
- Music

DGJ – Chapter 4, p163-p220

Sound

Sound is a 1 dimensional signal of amplitude (volume) which varies over time.

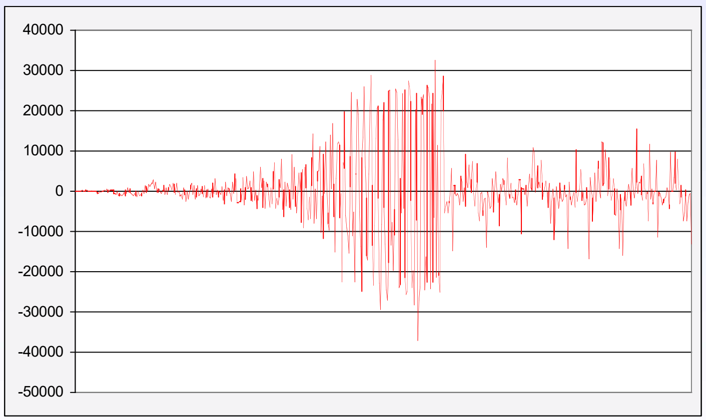
- The amplitude at a given point in time determines the pressure on your ear drum
- The higher the amplitude, the louder the sound and the more your ear drum is pressed in
- Your ear drum flexes 1000's of times a second
- Tiny hairs sitting inside a fluid filled spiral inside your ear (the cochlea) respond to different frequencies of sound

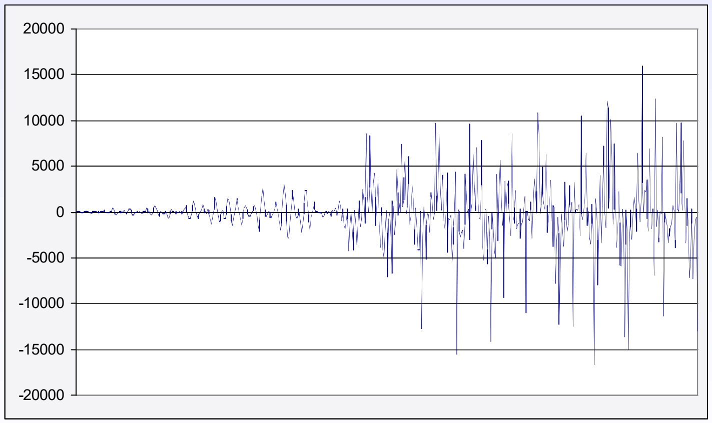


Sound Mixing

When we hear sound, it is a blend of many signals all combined together. Our ears and brain do a rather amazing job of splitting it into it's original parts.

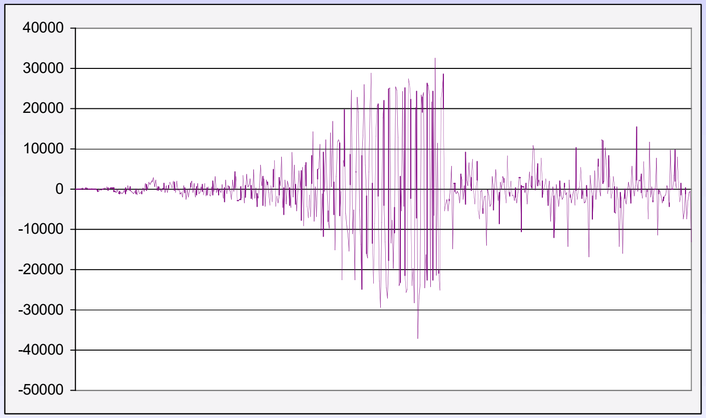
Take the following 2 sounds:



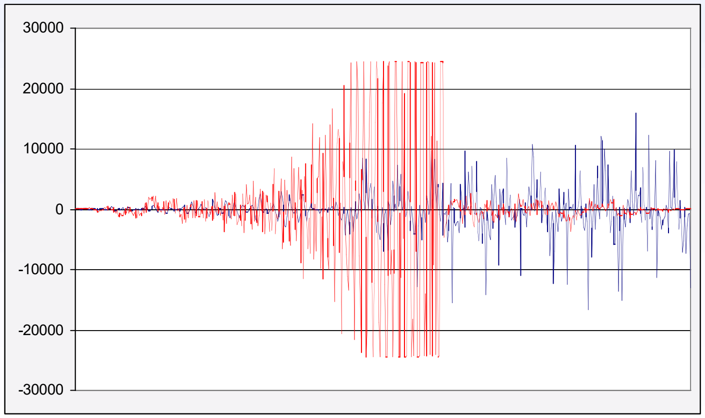


Sound Mixing

If heard together, your ear receives this:



But your brain is able to pick it apart to determine that it was formed from this:



Playing a Sound in Java

We require the following ingredients...

- File : A reference to the file to be played
- AudioInputStream : The stream of sound data
- AudioFormat : Information about the format of the data
  - 16/32bit, sample rate (e.g. 44,100Hz), mono or stereo
- Line (or subclass e.g. DataLine) : A way of connecting the AudioInputStream to the Java sound system
- Clip : A way of controlling the playback of the sound

Why so complicated?

- The breakdown of the sound into its components provides us with a lot of flexibility
- We can create our own sound effects / filters / mixers
  - This would not be possible if all you could do was play a sound file without access to its contents

Playing a Sound - Example

```
import java.io.*;
import javax.sound.sampled.*;

public class SoundPlay {

	public static void main(String[] args) {
		SoundPlay s = new SoundPlay();
		s.play("sounds/groovey.wav");
		System.exit(0); // Java Sound bug fix...
	}

	public boolean play(String filename)
	{
		try {
			File file = new File(filename);
			AudioInputStream stream = AudioSystem.getAudioInputStream(file);
			AudioFormat			format = stream.getFormat();
			DataLine.Info info = new DataLine.Info(Clip.class, format);
			Clip clip = (Clip)AudioSystem.getLine(info);
			clip.open(stream);
			clip.start();
			Thread.sleep(100); // Give it a chance to start playing....
			while (clip.isRunning()) { Thread.sleep(100); }
			clip.close();
		}
		catch (Exception e) { return false;			}
		return true;
	}
}
```

Sound Effects / Filters

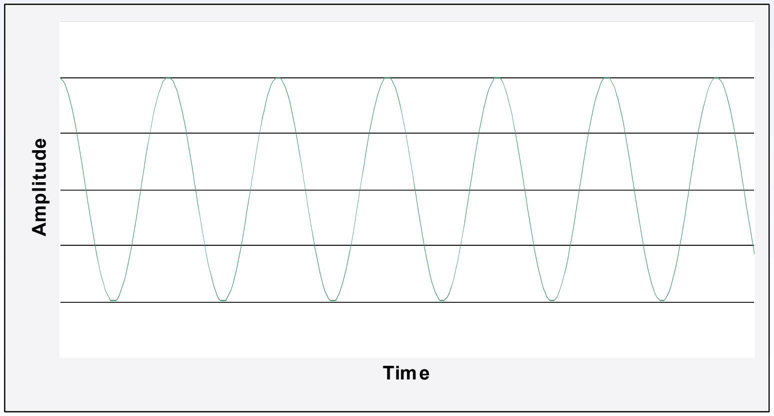
A Sound Filter can be applied to a sound source to alter the content of the source in some way. A common filter could be an echo, muffle or reverberation.

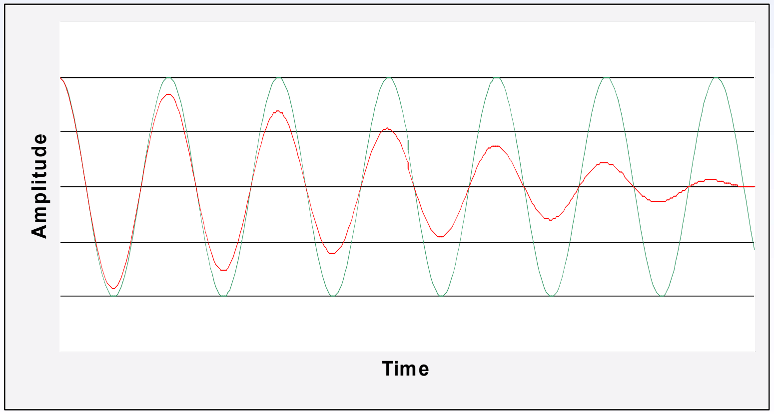
- You could record all the different types of sound you might wish to play before loading them into your game, however this is not very flexible.
- A better solution is to record a standard sample and then apply a particular filter to provide the distortion you want at run time (e.g. Muffle a sound if it passes through a wall).
- This becomes more important the bigger and more dynamic a game is, particularly with 3D games. In 3D it is usually not possible to anticipate all possible sounds (with distortions applied) that a user might hear.
- Sound filters are the transforms of the sound world

Sound Filters

A sound filter will take a data stream (usually in bytes) and change the contents of the stream in some way.

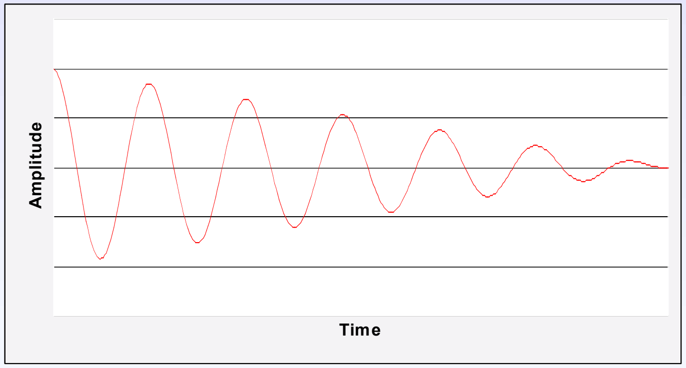
A simple fade filter would steadily reduce the amplitude (volume) of a signal as it passed through the filter.

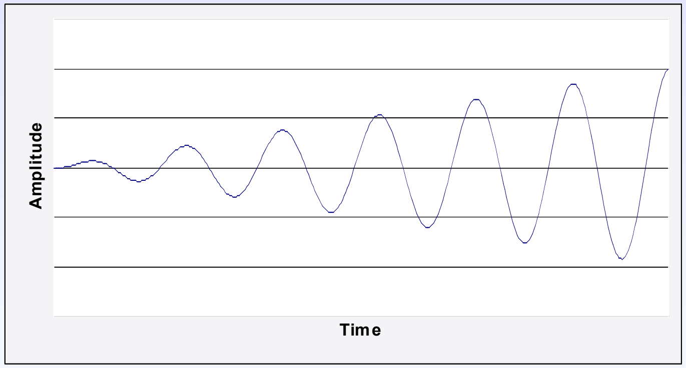




Reverse Filter

An alternative filter could be to reverse the signal (play it backwards) such that the signal that is at the start, becomes the one at the end:





Implementing a Sound Filter

In Java, we can implement a sound filter by extending the FilteredSoundStream class.

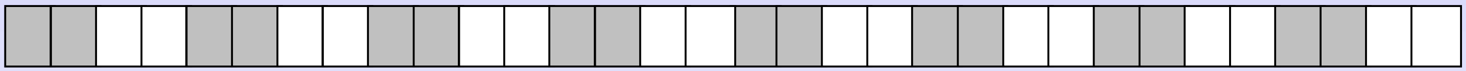
- We add our own tricks to a new 'read' method.

We also need a couple of methods to get and set the sound data since 'WAV' files are little-endian.

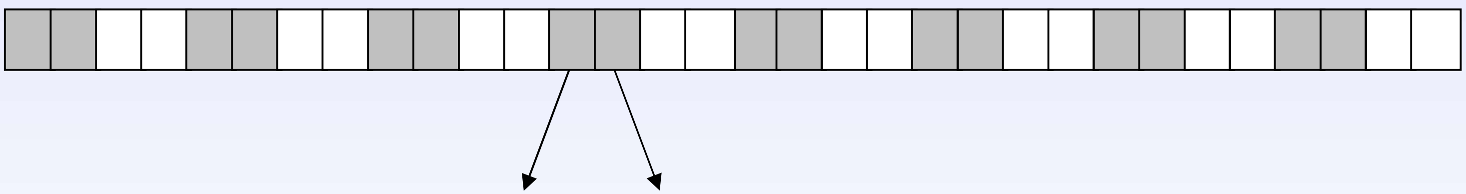
- Big endian data has the most significant byte first
- This is what you are used to
- Little endian data has the bytes switched round with the least significant byte first

Example: Big Endian vs Little Endian

For example, take a data stream of bytes coming from a sound source (e.g. WAV file):



We are going to read this source 2 bytes at a time since there are 16 bits (2 bytes) per sample.

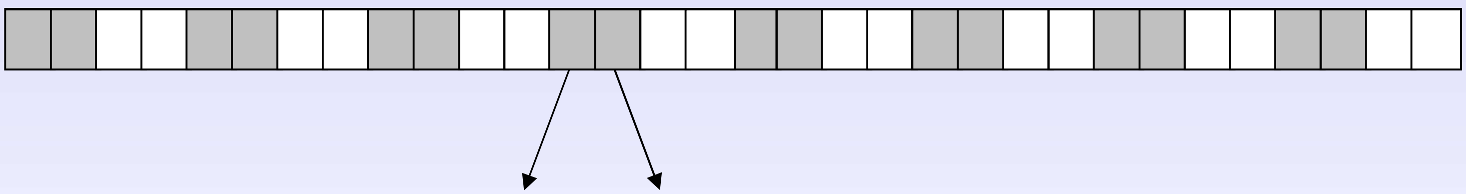


0	0	0	0	0	0	1	0		1	1	0	0	0	0	1	1

In the Big-endian world (that you are familiar with), you would make the 16 bit number 0000001011000011.

Example: Big Endian vs Little Endian

In the Little-endian world (of WAV files), you have to flip the bytes around to get the correct 16 bit number. You would read them in order as before...



0	0	0	0	0	0	1	0		1	1	0	0	0	0	1	1

Then switch them around to make the correct number (1100001100000010).

1	1	0	0	0	0	1	1		0	0	0	0	0	0	1	0

FadeFilterStream - 1

```
import java.io.*;

public class FadeFilterStream extends FilterInputStream {

    FadeFilterStream(InputStream in)
    {
        super(in);
    }

    // The following 2 methods work for 16 bit values only.
    // You will
    public short getSample(byte[] buffer, int position)
    {
        return (short) (((buffer[position+1] & 0xff) << 8) | (buffer[position] & 0xff));
    }

    public void setSample(byte[] buffer, int position, short sample)
    {
        buffer[position] = (byte)(sample & 0xFF);
        buffer[position+1] = (byte)((sample >> 8) & 0xFF);
    }
```

FadeFilterStream - 2

```
public int read(byte [] sample, int offset, int length) throws IOException
        {
                int bytesRead = super.read(sample,offset,length);
                float change = 2.0f * (1.0f / (float)bytesRead);
                float volume = 1.0f;
                short amp=0;

                // Loop through the sample 2 bytes at a time
                for (int p=0; p<bytesRead; p = p + 2)
                {
                        amp = getSample(sample,p);
                        amp = (short)((float)amp * volume);
                        setSample(sample,p,amp);
                        volume = volume - change;
                }
                return length;
        }
}
```

FadeFilterStream - 3

```
import java.io.*;
import javax.sound.sampled.*;

public class FadePlay {

	public static void main(String[] args) {
		FadePlay s = new FadePlay();
		s.play("sounds/voice.wav");
		System.exit(0); // Java Sound bug fix...
	}

	public boolean play(String filename)  {
		try {
			File file = new File(filename);
			AudioInputStream stream = AudioSystem.getAudioInputStream(file);
			AudioFormat format = stream.getFormat();
			FadeFilterStream filtered = new FadeFilterStream(stream);
			AudioInputStream f = new AudioInputStream(filtered,format,stream.getFrameLength());
			DataLine.Info info = new DataLine.Info(Clip.class, format);
			Clip clip = (Clip)AudioSystem.getLine(info);
			clip.open(f);
			clip.start();
			Thread.sleep(100);
			while (clip.isRunning()) { Thread.sleep(100); }
			clip.close();
		}
		catch (Exception e) { return false;			}
		return true;
	}
}
```

A Sound Filter Class

In practice, you should create a SoundFilter class which is passed to a FilterInputStream to achieve the desired filter. The FadeFilterStream class shows the principle behind manipulating a sound signal, however for a more detailed example with a better class structure, see DGJ, Chapter 4 p163-185.

Multiple Sounds

The previous examples did not return control until they had finished playing a sound. This is not very useful.

- What if you want to play multiple sounds?
- What if the player wants to move?
- What about updating the screen?

Threads to the rescue!

- Each sound can be played in a separate thread
- We can start a sound playing, then go back to listening for keyboard input or animating movements
- Imagine the alternative...

Multiple Sounds - Example 1

```
import java.io.*;
import javax.sound.sampled.*;

public class ThreadPlay extends Thread {

    String filename;    // The name of the file to play
    boolean finished;   // A flag showing that the thread has finished

    ThreadPlay(String fname) {
        filename = fname;
        finished = false;
    }

    public static void main(String[] args) {
        ThreadPlay s1 = new ThreadPlay("sounds/groovey.wav");
        ThreadPlay s2 = new ThreadPlay("sounds/mad.wav");

        s1.start();
        s2.start();

        while (!s1.finished || !s2.finished);

        System.exit(0); // Java Sound bug fix...
    }
```

Multiple Sounds – Example 2

```
public void run() {
    // This used to be called play but now it's a thread,
    // we rename it to 'run' since this is where the action is.
    // Apart from the use of 'filename' it is the same as before.
    try {
        File file = new File(filename);
        AudioInputStream stream = AudioSystem.getAudioInputStream(file);
        AudioFormat        format = stream.getFormat();
        DataLine.Info info = new DataLine.Info(Clip.class, format);
        Clip clip = (Clip)AudioSystem.getLine(info);
        clip.open(stream);
        clip.start();
        Thread.sleep(100);
        while (clip.isRunning()) { Thread.sleep(100); }
        clip.close();
    }
    catch (Exception e) {    }
    finished = true;
  }
}
```

Music

Background music can be useful to add atmosphere to a game scene. Some games change the music just before something bad is about to happen...

Options

- Uncompressed
  - CD, WAV
- Compressed
  - LA, FLAC
- Lossy Compressed
  - MP3, Ogg Vorbis, Real
- Music Notation
  - MIDI

Sound Formats

Uncompressed

- CD Audio, WAV
- Sound signal is saved at the same rate it was recorded
- Highest sound quality
- Largest file size

Compressed

- LA : Lossless Audio
- FLAC : Free Lossless Audio Codec (Coder / Decoder)
- Decoded sound signal is the same quality as original recording (no loss)
- Encoded file is significantly smaller than original recording (~50%)
- Requires decompression before it can be sent to sound device (uses extra CPU time)

Sound Formats

Lossy Compression

- MP3, Ogg Vorbis, WMA, Real
- Sound data is re-sampled and compressed, throwing away higher frequencies we are less likely to notice
- The more data thrown away, the more likely you will notice
  - A little test…
- Very useful for streaming over the internet / portable music devices

MIDI

- Music is stored as a form of notation
  - Multiple tracks, for each track: instrument bank, note on, note off, volume
- Very compact, files are a few K rather than MB
- Reproduced sound dependent on quality of sound bank for the reproduced instrument
- Allows a track recorded via a keyboard to be played back as a guitar…
- Only records music – not vocals

Playing MIDI in Java

MIDI allows you to adapt music in response to game play.

You need:

- Sequence
  - the MIDI data encoding the musical score
- Sequencer
  - reads the sequence and sends it to the synthesiser

Soundbanks

- A MIDI sequencer uses a soundbank (a database of sound samples) to play each note listed in the MIDI file.
- The default soundbank in Java is not great but you can download a better version from :

  http://java.sun.com/products/java-media/sound/soundbanks.html

MIDI sound tracks

- http://www.partnersinrhyme.com/midi/index.shtml

Playing MIDI in Java - Example

```
import java.io.*;
import javax.sound.midi.*;

public class PlayMIDI {

        public static void main(String[] args) throws Exception {

                PlayMIDI            player = new PlayMIDI();
                player.play("sounds/music.midi");
                System.exit(0);
        }

        public void play(String filename) throws Exception        {
                // Get a reference to the MIDI data stored in the file
                Sequence score = MidiSystem.getSequence(new File(filename));
                // Get a reference to a sequencer that will play it
                Sequencer seq = MidiSystem.getSequencer();

                // Open the sequencer and play the sequence (score)
                seq.open();
                seq.setSequence(score);
                seq.start();
                while (seq.isRunning()) { Thread.sleep(100); }
                seq.close();
        }
}
```

Altering MIDI in Java

MIDI tracks

- A MIDI file will contain a number of tracks, one for each instrument
- We can select or deselect the tracks we would like to play using the *setTrackSolo* and *setTrackMute* methods for the Sequencer class
  - setTrackSolo(int track, boolean solo)
    - If *solo* is true, only play this *track* (and other tracks with *solo* set to true)
  - setTrackMute(int track, boolean mute)
    - If *mute* is true, silence the given *track*

Tempo

- You can alter the tempo of the musical piece
  - setTempoFactor(float factor)
    - A factor > 1 speeds up the music, a factor < 1 slows it down
- Demo...

Manipulating MIDI

Why would we do this?

- A MIDI file could contain a theme tune with many different instruments
- A particular group of instruments would give a certain feel to the theme music
- You can turn on and off the relevant instrument tracks to get the required feel
- Changing the tempo, will either calm the player or create a sense of urgency