Sound Effects and Music

Chapter 4

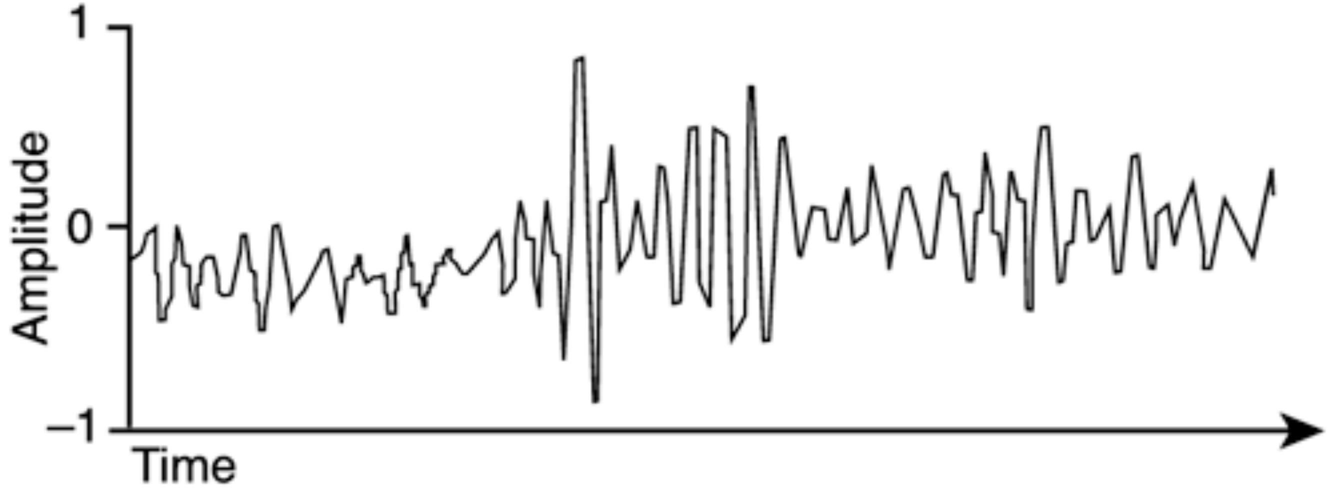
Content

Introduction

- When playing a game sound effects might be there but you don’t hear them.
- You expect to hear them.
- Sound is important part of a game.
- We will learn basics of playing sound and then move to effects.
- Create a sound manager.
- Learn to play music with dynamic changes.

Sound Basics

- Sound is vibration through a medium.
- Your eardrums pick up the vibration and signal your brain, which interprets it as sound.
- Vibration through the air creates pressure fluctuations.
- Faster fluctuations create a higher sound wave frequency, leading you to hear a higher pitch.



Sound Basics (2)

- Digital sound, such as that in CD audio and many computer sound formats, contains sound as a series of discrete samples of the sound's amplitudes.
- Sample rate is amount of samples stored per second (e.g. CD 44,100 Hz)
- Higher sample rates result in a more accurate audio representation, and lower sample rates mean poorer quality but a smaller file size.
- Samples are typically 16 bits, giving 65,536 amplitude possibilities.
- Multichannel sound.

Java Sound API (2)

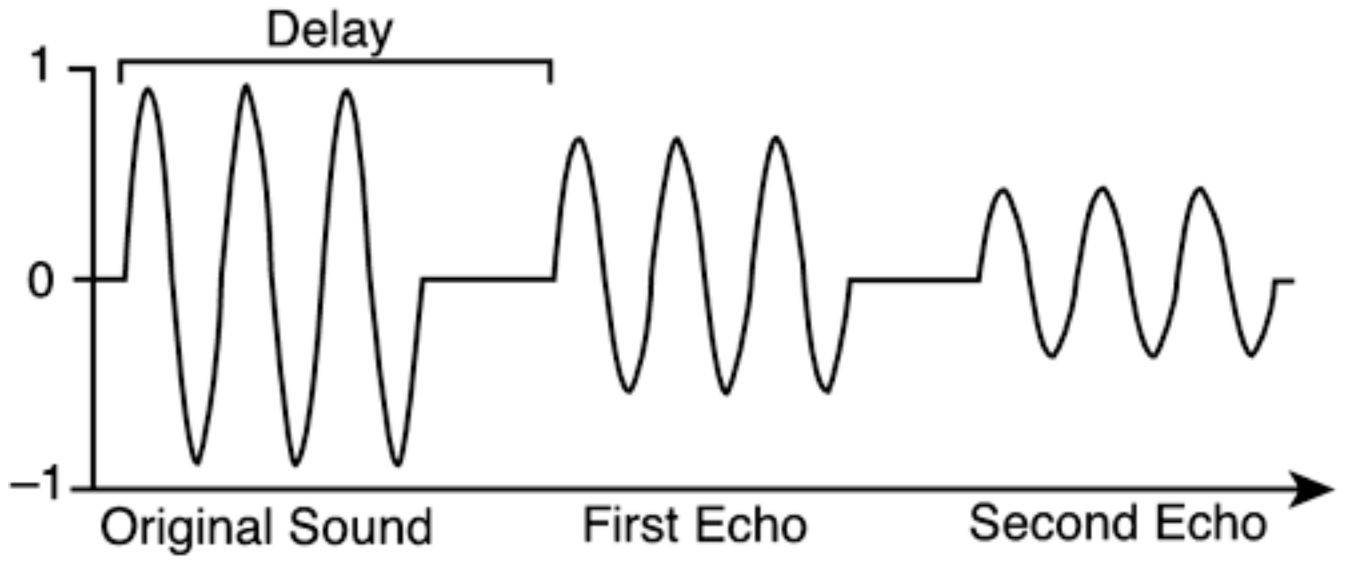
- Some sound programs that you can use to create, record, and edit sounds are
  - Pro Tools (www.avid.com),
  - Cool Edit (www.oldversion.com/windows/cool-edit-pro/),
  - GoldWave (www.goldwave.com), and
  - Audacity (audacity.sourceforge.net).

Creating a Real-Time Sound Filter Architecture

- *Sound filters* are simple audio processors that can modify existing sound samples, usually in real time. Sound filters, also known as *digital signal processors*, are used in audio production all the time—for example, to add distortion to a guitar or an echo to a voice.
- Sound filters can make your game more dynamic.
  - Game in which the player is wandering around a cave, an echo would be appropriate.
  - Or, if a rocket is whizzing past the player, you could make the sound shift from the left to right speaker.
- In this section, you'll create a real-time sound filter architecture and create two filters: an echo filter and a simulated 3D sound filter.

Creating a Real-Time Echo Filter

- Two elements of an echo: delay and decay.
- The delay is how long it takes for the echo to occur. The decay is how much quieter the echo was compared to the original sound.
- In the case of you shouting on top of the mountain, the delay was about one second, and the decay was probably less than 50%. A decay value of 100% means the echo never dies out.



Creating a Real-Time Echo Filter (2)

- The SoundFilter class is format-agnostic. It doesn't matter whether the sound you are filtering has a 44,100Hz sample rate or an 8,000Hz sample rate. So, you can't simply tell the echo filter the amount of time to delay.
- Instead, you tell it how many samples to delay. For a 44,100Hz sample rate and a one-second delay, tell the filter to delay by 44,100 samples.
- Remember, the delay counts from the beginning of the sound, not the end.
  - So, if you want to delay one second after a 44,100Hz sound is finished, set the delay to the number of samples in the sound plus 44,100.
  - Usually, though, you'll want the same echo delay across all sounds, which means that with longer sounds, the first echo might kick into action while the original sound is still playing.

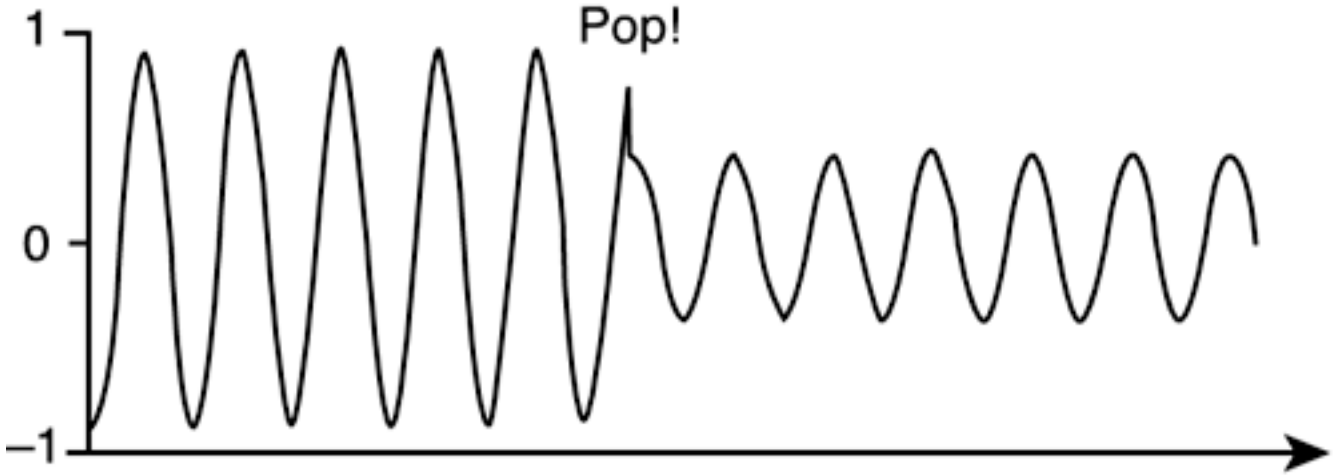
Emulating 3D Sound

- Many different effects are used to create 3D sound. Here are some of the most common ones:
  - Make sound diminish with distance so the farther away a sound source is, the quieter it is.
  - Pan sounds to the appropriate speaker so a sound source on the left of the player is played in the left speaker, and a sound source on the right of the player is played in the right speaker. This can also be extended to four-speaker surround sound.
  - Apply room effects so sound waves bounce off walls, creating echoes and reverberation.
  - Apply the Doppler effect so a sound source's movement affects its pitch. For instance, a sound source quickly moving toward the player has a higher pitch than a stationary one. If you've ever heard a fire engine or train whiz past you, you've experienced this effect.

Idea Behind Creating a 3D Filter

- Using Java Sound's Controls, you can dynamically change the volume and pan of a sound. Unfortunately, the Controls don't have the best quality for real-time use, and using them for such can create clicks or "wobbly" sounds. Therefore, you'll do the volume change yourself instead of using Controls. This means it is your task to avoid clicks and pops.

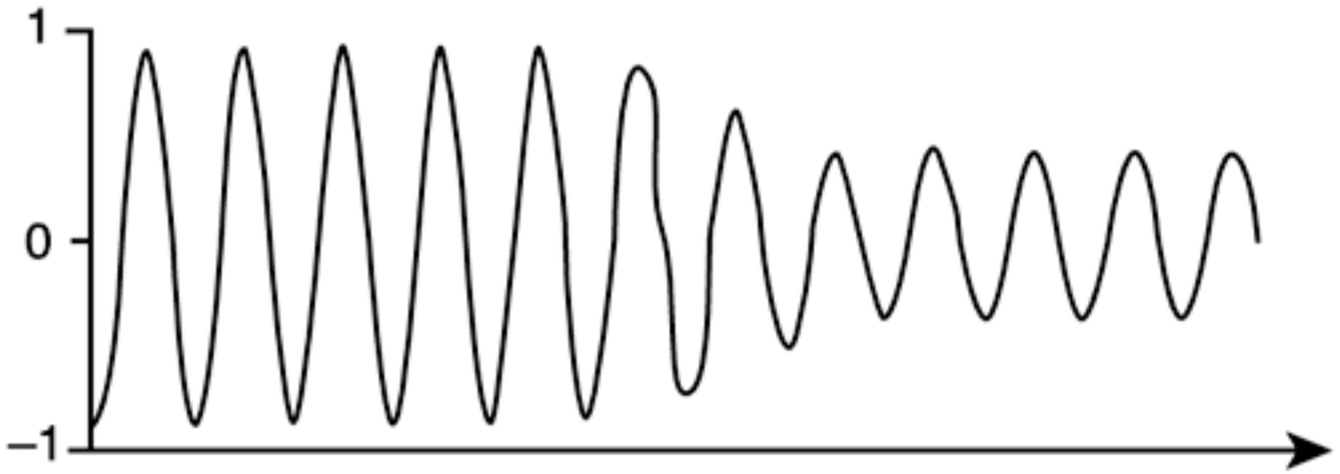
- Clicks and pops can happen when the volume of a sound abruptly changes.



Abruptly changing the volume of a sound can result in "pops."

Idea Behind Creating a 3D Filter (2)

- To avoid this in your 3D filter, whenever you need to change the volume of a sound, be sure to gradually change the volume over time



Gradually changing the volume of a sound creates a more natural listening experience.

- Because your samples are stored in 16-bit signed format, to change the volume, just multiply each sample by a factor.

```
sample = (short)(sample * volume);
```

Implementing a 3D Filter

- The **Filter3d** class
  - Modifies the volume of a sound to make it get quieter with distance.
  - It keeps track of two Sprite objects: one as a sound source and one as a listener. The farther away the source Sprite is from the listener Sprite, the quieter the sound is.
  - Distance is measured using the Pythagorean Theorem:

$$A^2+B^2=C^2$$

- Maximum distance sound
  - If the listener is more than the maximum distance from the sound source, the sound isn't heard at all. The sound's volume is scaled linearly from 0 to the maximum distance.

Implementing a 3D Filter (3)

- In Filter3d you create a diminishing-with-distance effect—the minimum amount necessary for a 3D effect.

- If you wanted to add panning, you would need to update the SoundFilter architecture to enable you to turn a mono sound into a stereo sound. Then you could calculate the pan you wanted from -1 to 1, where -1 is the left speaker, 1 is the right speaker, and 0 is in the middle. Finally, the samples for each speaker could be calculated as follows:

```
        short sampleLeft = (short)(sample * (1-pan));
        short sampleRight = (short)(sample * (1+pan));
```

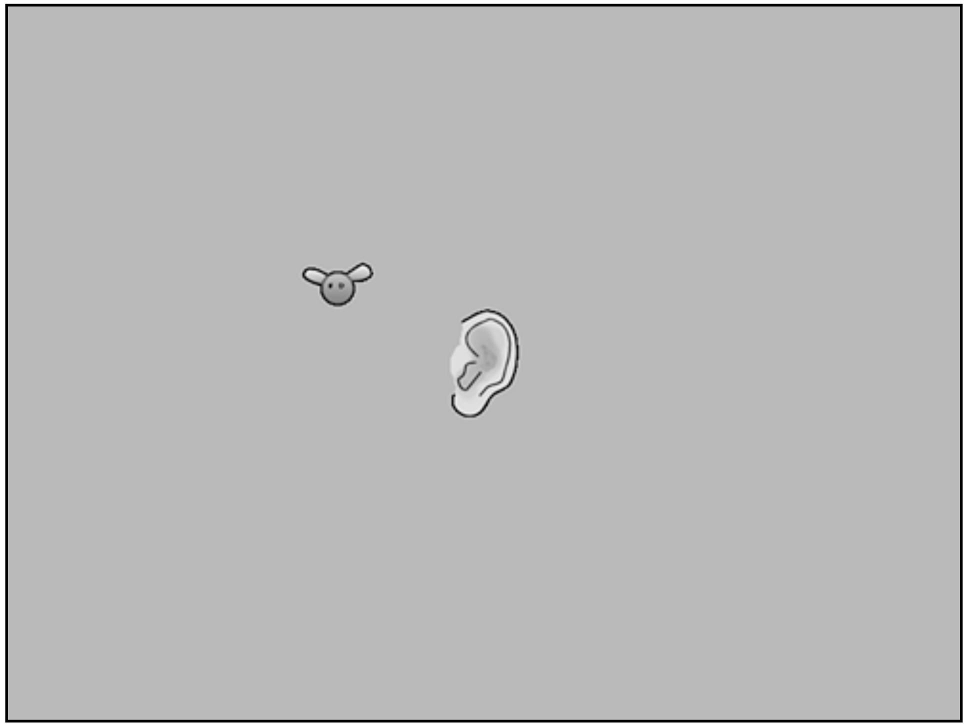
- For a 2D game, the pan could be determined by the position where a sound source is on the screen. A sound source on the left side of the screen would play in the left speaker, and so on.

Trying Out the 3D Filter

- The diminishing-with-distance effect is cool enough for many games; you can try it out by creating the **Filter3dTest** class.

- The Filter3dTest class is a graphical demo that enables you to move a fly around on the screen using the mouse.

- Along with the fly is a virtual "ear" in the middle of the screen. The fly makes a buzzing sound, and the closer the fly is to the ear, the louder the buzzing sound is.



Source code: Filter3dTest.java

Playing Music

- Music can change the player's emotions toward elements of the game (e.g. happy music for easier levels, dramatic music for more difficult levels).
- When you've decided on the type of music you want, the next step is to figure out where the music comes from. Games typically play music in one of three ways:
  - Streaming music from an audio track CD
  - Playing compressed music such as MP3 or Ogg Vorbis
  - Playing MIDI music

Playing CD Audio

- Some CD-ROM games have plain old Red Book Audio (the standard audio CD format) right on the CD.
  - The benefit here is you get great-quality sound and it's easy to implement: Just tell the sound card to start playing the CD, without any other involvement from the game.
  - The other cool aspect is that players can slip the game CD into their audio CD player and groove to the game's tracks.
- Unfortunately, CD audio takes up a lot of space, typically around 30MB for a three-minute song. If you have four three-minute songs, that's 120MB of space that could be used for more graphics or bigger levels. In addition, the Java Sound implementation doesn't support playback from a CD, so this option is out for you.

Playing MP3 and Ogg Vorbis

- The second option is compressed music. MP3 and Ogg Vorbis formats are much smaller than that for CD audio, typically around 3MB for a three-minute song, and have near-CD quality. They've become increasingly more popular in games.

- The drawback here is that decoding MP3 or Ogg Vorbis files takes quite a chunk of processor power. Sound cards can't play compressed music directly, so the music is decoded while it's played. This means the extra processor use can interfere with other parts of your game. On faster, modern machines, the decoding won't be noticeable, but on slower, older machines, the decoding could take 20% to 40% of the processor or more. That could make other parts of the game, such as animation, seem slow or jerky.

- If the processor time doesn't matter to your game, you'll need to get an MP3 or Ogg Vorbis Java decoder.

Playing MP3 and Ogg Vorbis (2)

```
// create the format used for playback
// (uncompressed, 44100Hz, 16-bit, mono, signed, little-endian)
AudioFormat playbackFormat =
    new AudioFormat(44100, 16, 1, true, false);

// open the source file
AudioInputStream source =
    AudioSystem.getAudioInputStream(new File("music.ogg"));

// convert to playback format
source = AudioSystem.getAudioInputStream(playbackFormat, source);
```

- Also, be sure not to load the samples into memory. A compressed sound file might take up only 1MB for each minute of music, but the uncompressed samples would take 10MB for each minute. Instead, play any large sounds directly from the AudioInputStream, which streams the sound from disk.

- But which one should your game use, MP3 or Ogg Vorbis? Although MP3 is incredibly popular, Ogg Vorbis is license-free and claims better sound quality. The people playing your game won't notice or care, so go with whichever one suits your needs better. You can find more information on Ogg Vorbis at www.xiph.org/ogg/vorbis. Also, be sure to look into MP3 licensing issues at www.mp3licensing.com.

Playing MIDI Music

- Finally, there's MIDI music. MIDI music isn't sampled music like other sound formats
- It works more like a composer's sheet music, giving instructions on which note to play on each instrument. The audio system synthesizes to each note according to the pitch, instrument, and volume, among other parameters.
- Because MIDI files contain instructions instead of samples, MIDI files are very small compared to sampled sound formats and are often measured in kilobytes rather than megabytes.
- Because the music is synthesized, the quality might not be as high as that of sampled music. Some instruments won't sound realistic, or the music might sound a little too mechanical. A creative musician can usually mask the deficiencies of MIDI music, however.

Playing MIDI Music (2)

- The Java Sound API synthesizes MIDI music through the use of a sound bank, which is a collection of instruments. Unfortunately, although the Java SDK includes a sound bank, the Java runtime does not. If a sound bank isn't found, the hardware MIDI port, which has unreliable timing in Java Sound, is used. For this reason, it's recommended to use a sound bank.

- The Java SDK includes a minimal-quality sound bank, and you can download higher-quality sound banks from *http://java.sun.com/products/java-media/sound/soundbanks.html* and include them with your game.

- The Java Sound API provides MIDI sound capabilities in the **javax.sound.midi** package. To play MIDI music, you need two objects in this package: Sequence and Sequencer. A Sequence object contains the MIDI data, and a Sequencer sends a Sequence to the MIDI synthesizer. Here's an example of playing a MIDI file:

Creating Adaptive Music

- Adaptive music is music that changes based on the state of the game.
- For instance, if the player is battling a large number of enemies, the music might be fast and loud. Conversely, the music might be quiet when the player is walking around exploring rooms alone.
- The change in music could happen at any time—for example, the player could be strolling along one second, and then 100 robots could be trying to kill him the next. So, changing the music smoothly can be a challenge.
- You can adapt songs to the game state in two ways:
  - Change songs
  - Modify the song currently playing

Creating Adaptive Music (2)

- Because the actions of a player can change at any time, changing songs is a more difficult task. The change can't be abrupt, or it will be distracting. Songs can be designed so that they have "change points" to signify places where a song change can occur.

- Also, songs need to transition smoothly. To do this, the first song can fade out while the next song is fading in. Also, while the first song is fading out, its tempo could change to match the tempo of the next song.

- Or, you could just take the easy way out and insert a sound of a scratching phonograph needle.

Creating Adaptive Music (3)

- The second option is to simply modify the exiting song. You can do this by changing the tempo or volume, or adding another instrument to it.

- Adding or taking away an instrument is easy to do with MIDI sequences. MIDI music is typically organized into tracks, with each track playing a specific instrument. For instance, there might be one track for guitar, one for keyboards, and so on.

- You can mute or unmute a track with one method:

```
sequencer.setTrackMute(trackNum, true);
```

- Here, trackNum is an integer representing the track number you want to mute. If you don't know what track belongs to what instrument in your MIDI file, you might have to experiment by muting each track, one by one.

```
Source code: MidiTest.java
```

Summary

- In this chapter, you learned the basics of sound, sound filters, and music.
- You made real-time echo and pseudo-3D filters, and you created a cool sound manager to handle game sound effects.
- You also made some adaptive music and learned how to play back MP3 and Ogg Vorbis sound files.
- Combining this knowledge with graphics and interactivity from the previous chapters, you're ready to create your own game.