Java game programming

Sound

2010

Fayolle Pierre-Alain

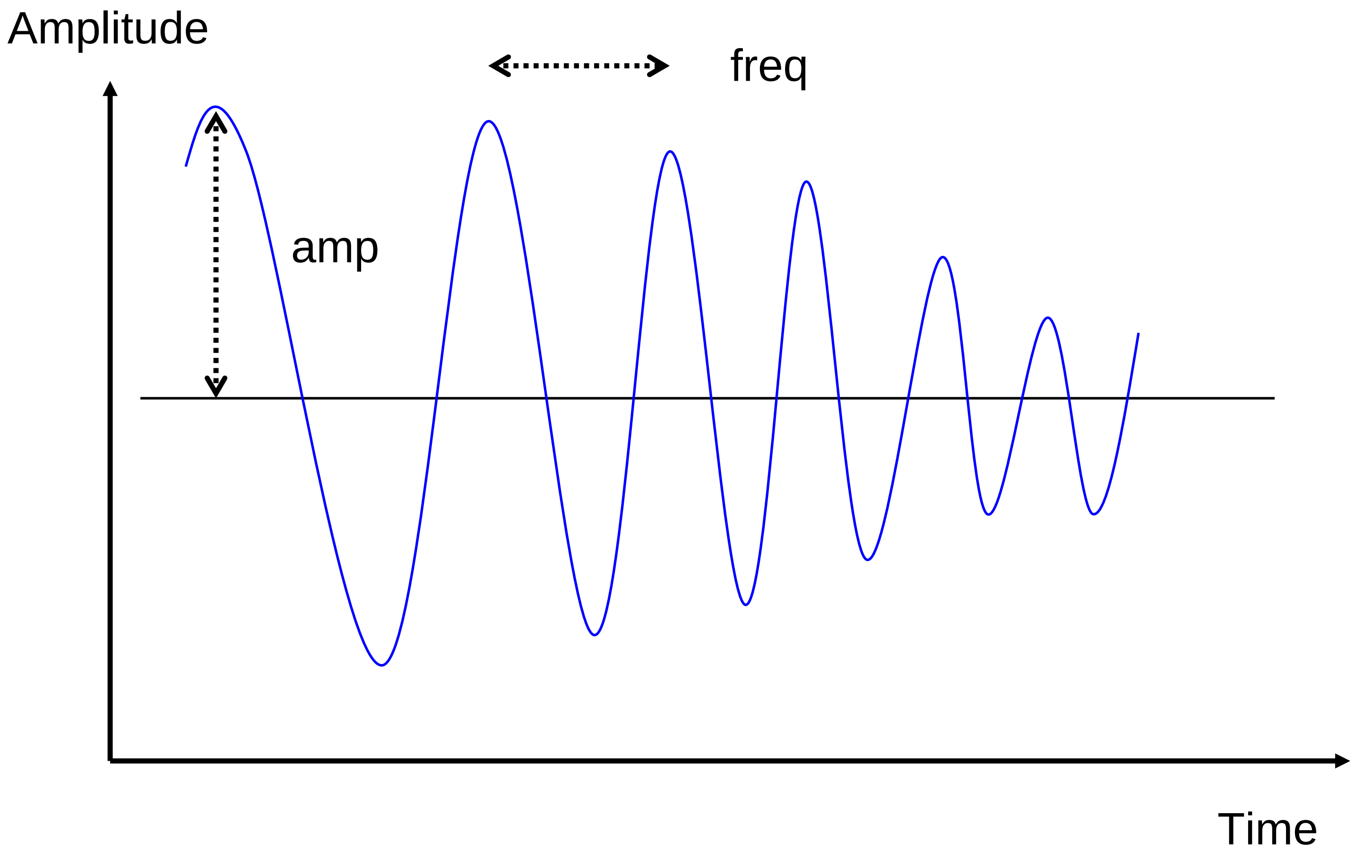
Plan

- Sound basics
- Java Sound API
- Playing sound
- Filtering sound
- 3D sound with JOAL
- Playing music

Sound basics

- A sound is a wave travelling through a medium (air, water, …..)
- It is a wave of alternating pressure deviations from the equilibrium pressure of the medium the wave is travelling in
- Molecules in the medium are oscillating, periodically displaced by the sound wave

Sound basics: wave



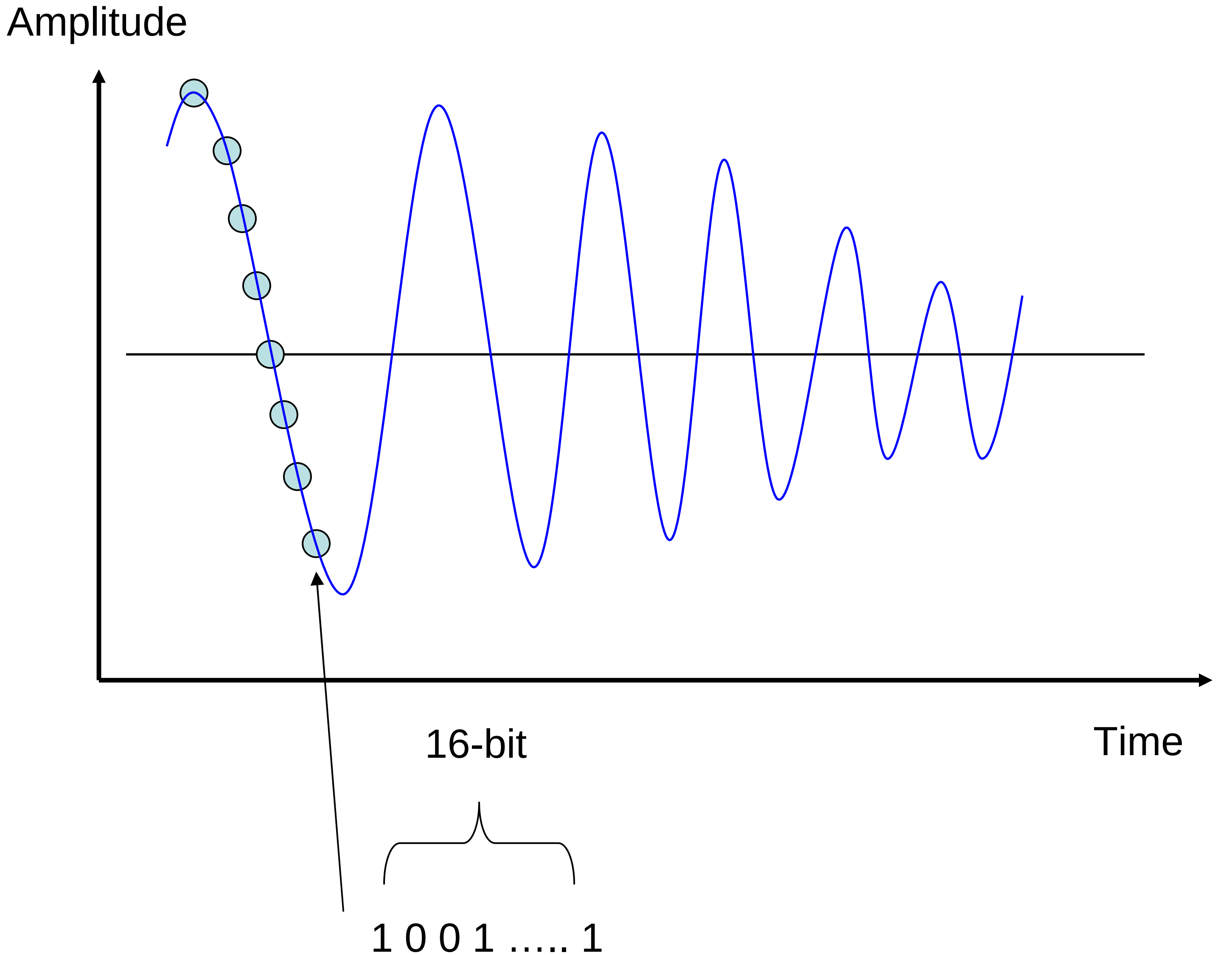
Sound basics

- Sound waves have usual characteristics of waves: frequency, amplitude, …..
- Sound waves with higher amplitude corresponds to louder sound
- Higher frequency leads to higher pitch

Sampling

- Digital sound (for example: computer sound formats) consists of a series of discrete samples of the sound's amplitudes
- the sampling is done at two levels:
  - The number of points taken along the time axis
  - The number of bits used to represent the amplitude along the vertical axis
- The first one is called the sampling rate and corresponds to the number of samples stored per second (unit: Hertz)
  - Audio CD have sampling rate of 44.1 kHz
- The second one corresponds to the number of bits used to encode the amplitude (quantization)

Sampling



Java sound API

- Interfaces and classes for capturing, processing and playing sampled sound data are in the *javax.sound.sampled* package
- This sound API was made available starting with the Java 2 Platform
- The API supports three sampled audio file formats: AIFF, AU, and WAV

Java sound API

- Supported file formats:
  - WAV (or WAVE): usually associated with Windows PC
  - AIFF: often associated with Macs
  - AU: often associated with UNIX systems
- The Java API can handle:
  - 8-bit and 16-bit audio data (quantization)
  - Mono / stereo
  - Sample rates from 8 kHz to 48 kHz

Java sound API essentials: Data

- Formatted audio data: data formats and file formats
- Data formats: tells you how to interpret a series of bytes of raw sampled audio data
- Data formats is represented by a *AudioFormat* object which stores: encoding techniques, num of channels, sample rate, num bits per sample, frame rate, etc
- Some information like frame rate depends on the encoding technique

Java sound API essentials: Data

- File formats:
  - Specify the structure of a sound file
  - Various information (name, length)
  - Format of the raw audio data
- In Java, a file format is represented by an object of type *AudioFileFormat* containing:
  - File type (WAV, etc)
  - File's length (in bytes)
  - Length of the audio data
  - AudioFormat object specifying the data format

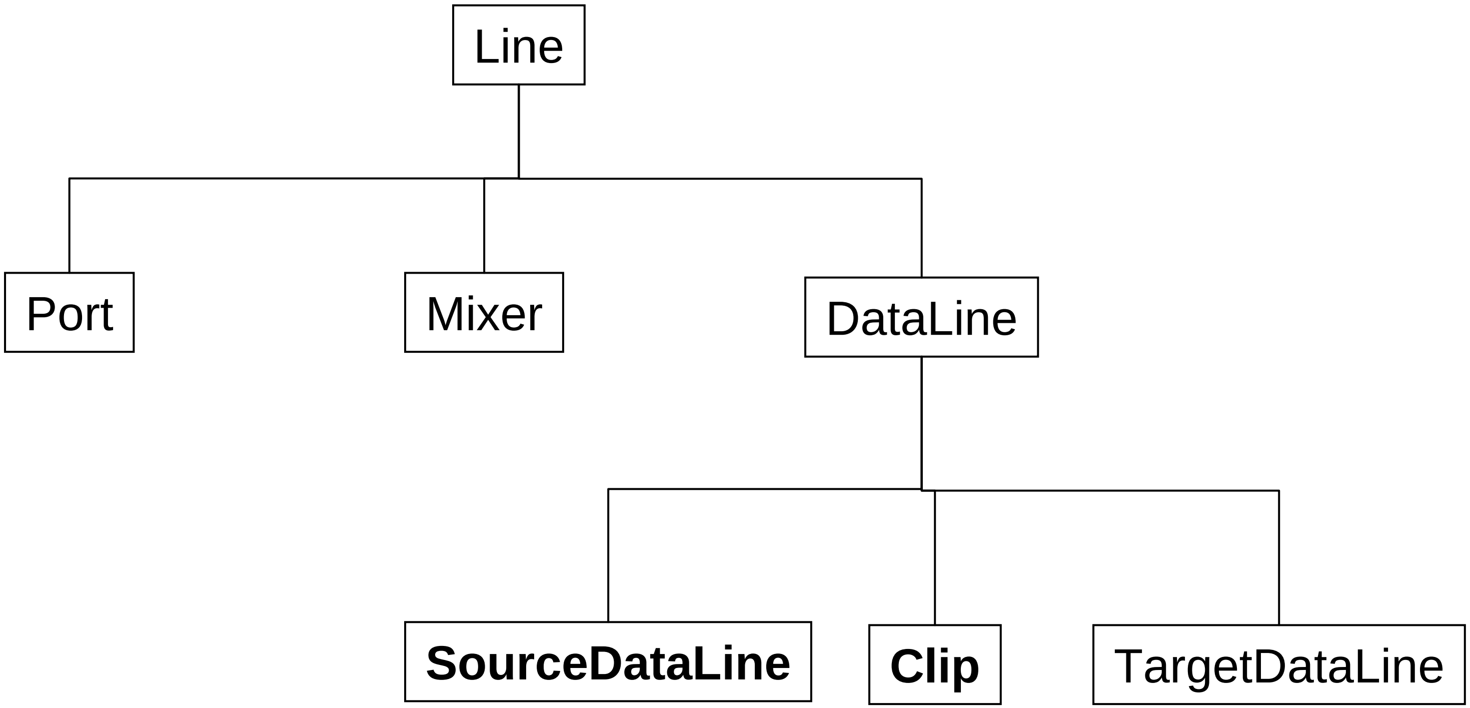
Java sound API essentials: Data

- The *AudioSystem* class provides methods for read / write sounds in the different file formats
- Possibility to access the content through an *AudioInputStream*, a subclass of Java's *InputStream*, as a series of bytes to be read sequentially

Java sound API essentials: Line

- An element of digital audio pipeline
- A path for moving audio in and out the system
- A line can be used to receive audio from the sound system
  – For example: from a microphone
- A line can be used to send audio to the sound system
  – For example: send audio to the sound system to play
- Line is an interface with several sub-interfaces (of interest: SourceDataLine and Clip)

Line class hierarchy



SourceDataLine

- A data line to which data (sound) may be written
- An application writes audio bytes to a source data line
- The source data line handles the buffering of the data and delivers to the an output port (ex: speaker on a sound card) through the mixer
- The mixer (class *Mixer*) is used to mix samples from different sources (if needed)
- An object of type SourceDataLine is obtained by the method getLine() in the class AudioSystem

SourceDataLine (2)

- Applications playing (or mixing) audio should send data quickly enough to prevent the buffer from underflowing
- When it happens, it causes discontinuities in the audio (perceived by the user as clicks)
- Writing to a SourceDataLine object is done with the method write(byte[], int off, int len)

Clip

- Clip interface is used when audio data can be loaded prior to being played
- Data is pre-loaded and has known length so it is possible to:
  - Loop through the data
  - Start playing at any position
- Clip can be obtained with the method getLine() of the class *AudioSystem* (same as SourceDataLine)
- Audio stream is loaded with the method open()
- Playback is started by the method start()

Difference between Clip and SourceDataLine

- Clip is used if the sample audio is *known in advance* because the sound data needs to be specified at one time before the playback
- SourceDataLine is used if the sample audio can’t be known in advance

Accessing audio system ressources

- Class AudioSystem serves as an entry point for accessing sampled audio ressources
- Query AudioSystem to get information and obtain ressources
- Example:
  - Get access to installed mixers
  - Get access to lines (even without dealing with mixers)
  - Translate audio data from one format to another
  - Translate between audio file and audio stream. Read and write files using different format

Example: Opening a Sound File

- Open a sound file (with known file format)
- Get an input audio stream using method getAudioInputStream from AudioSystem

```
File f = new File(“test.wav”);
AudioInputStream s = AudioSystem.getAudioInputStream(f);
```

Example: Playing a sound using a Clip

- Get the data format of the audio input stream with method getFormat()
- Get a line from the audio system using getLine(Line.Info) and specifying the proper line info
- Load the audio stream with open(AudioInputStream)
- Start the playback with start() method

Playing a sound using a Clip (2)

```
File f = new File(“test.wav”);
AudioInputStream s = AudioSystem.getAudioInputStream(f);
AudioFormat format = s.getFormat();
DataLine.Info info = new DataLine.Info(Clip.class, format);
Clip clip = (Clip)AudioSystem.getLine(info);
clip.open(s);
clip.start();
```

Example: Playing sound using a SourceDataLine

- The first steps (up to loading data to the line) are similar
- Opening a SourceDataLine is done with: open(AudioFormat f)
- Start playing sound by invoking the method start() and repeatedly writing to the line's playback buffer

Playing sound using a SourceDataLine (2)

```
File f = new File("test.wav");

AudioInputStream s = AudioSystem.getAudioInputStream(f);
AudioFormat format = s.getFormat();

DataLine.Info info = new DataLine.Info(SourceDataLine.class,
format);
SourceDataLine line = (SourceDataLine)AudioSystem.getLine(info);

line.open(format);
line.start();

int total, numBytesRead;
byte[] buffer = new byte[bufferSize]
while (total < totalToRead && !stopped)}
    numBytesRead = s.read(buffer, 0, bufferSize);
    if (numBytesRead == -1) break;
    total += numBytesRead;
    line.write(buffer, 0, numBytesRead);
}
```

Sound filtering

- Filtering generally consists of some linear transformation of a number of surrounding samples around the current sample of the input signal
- In this case, filters modify existing sound samples
- It is usually applied in real-time
- It is used to make games more dynamic by adding acoustic effects
  - For example: making the sound going from the left speaker to the right speaker when the character crosses the scene
  - Making echo when the character is in a dungeon

Sound filtering (2)

- In terms of code design: all filters will subclass an abstract Filter class and override the doFilter() method
- Instead of applying the filter directly on raw data (i.e. array of bytes), it will be embedded in a class subclassing FilterInputStream

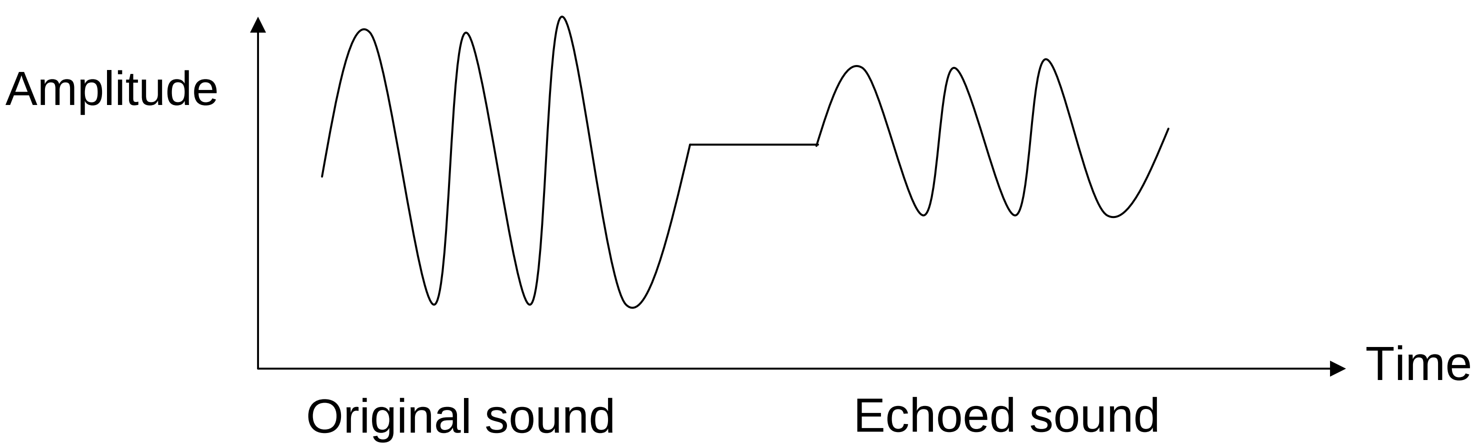
```
public class FilterSoundStream extends FilterInputStream {
  private Filter filter;
  public FilterSoundStream(Filter f, InputStream is) {
    …..
  }

  public int read(byte[] sample, int off, int len) {
    // override read to apply the filter on sample after it
    // is read from the stream
  }
}


public abstract class Filter {
  public abstract void doFilter(byte[] s, int off, int len);
  …..
}
```

Example of filter: echo

- An echo effect is obtained by repeating the same sound after some delay
- Usually the echo's intensity is also decreasing in time (the sound gets quieter than the original sound)



Example of filter: echo (2)

- The echoed sound can start while the original sound is still playing
- The decrease in amplitude is expressed as a percentage of the original sound
- The delay (shift along time) is expressed in number of samples (i.e. the next echoed sound will start at sample i on the curve)

Example of filter: echo (3)

- The filtered sound (i.e. echoed sound) is longer than the original
- After reading the audio sound and loading it into a byte buffer, this buffer is artificially increased
  - The first bytes correspond to the original sound
  - Later bytes will store the echoed sound (initially set to 0)
- To simulate the echo, an intermediate buffer is used
  - Its size is the number of samples used to simulate the delay
  - Originally this buffer contains 0
  - When we iterate through the sound buffer, the value of the sound buffer are saved in this intermediate buffer
  - short echoSample = (short)(oldSample + decay * buffer[currBufferPos])

3D sound

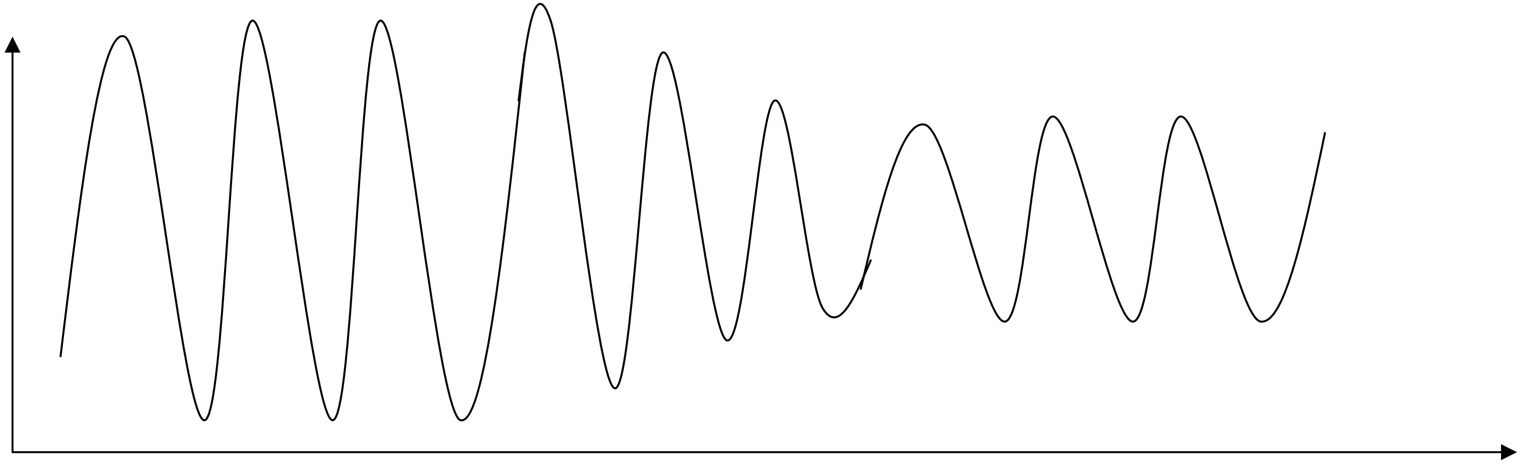
- Also called directional hearing
- Position virtual sound sources in 3D space
- Immerse the player in a 3D environment:
  - Possibility to hear sound coming from the left, right, back
- Techniques for 3D sound include:
  - Make sound decrease with distance (farther the source, quieter the sound)
  - Pan sound to the appropriate speaker (in case of several speakers available)
  - Room effects: echoes, reverberation, etc
  - Doppler effect

Doppler effect

- Doppler effect is the change of frequency of a wave for an observer moving relative to the wave source
- Example: a vehicle with siren approaches and passes an observer
- Received frequency is higher (compared to the emitted frequency) during the approach, identical when passing by and lower during the recession

Distance Filter

- It is possible to create a simple filter that simulates a “distance” effect by gradually scaling the volume of a sound when a listener move away from a source
- The sound gets quieter when the listener moves away from a source
- Volume needs also to be gradually changed to avoid sound artifacts
- New sample = old sample * volume



JOAL: Java bindings for OpenAL

- OpenAL is a cross-platform 3D audio API for use in gaming and virtual reality
- JOAL are Java bindings to allow calling functions from the OpenAL library from the Java language
- JOAL provides methods allowing you to:
  - Read sampled audio data into buffers
  - Associate these buffers with audio sources (defined by their space coordinates and eventually speed)
  - Specify listener by her orientation
- Effect such as volume decreasing with distance or Doppler effect can be easily implemented using the method provided by this library

Playing music

- In addition to sound effects, it is possible to add a background music to the game
- Music can be changed for each level of the game and can be used to reflect the intensity of the game
  – Example: the pace of the music can accelerate when bad guys arrive
- Games play music using one of the following ways:
  – Streaming music from an audio track on a CD
  – Playing compressed music (MP3)
  – Playing MIDI music

MP3 / vorbis

- Both MP3 and vorbis are compressed digital audio encoding format
- They are using lossy data compression, i.e. they are using data compression method which discards some of the data
- The use of lossy compression algorithm is used to reduce the amount of data required to represent the audio while still sounding like the original audio sound
- To play MP3 encoded files or vorbis encoded files, decoders are needed
- By default, the standard Java API does not provide such decoder
- The Java Media Framework API (JMF), which is an optional package, adds support for MP3
- Support for vorbis is provided by some external libraries such as: J-Ogg (www.j-ogg.de), or JOrbis (www.jcraft.com)
- Because the decoding is performed by the CPU, usage of MP3 / vorbis audio into game has some performance cost

MIDI music

- Sampled audio is a direct representation of sound
- MIDI can be thought as a a recipe for creating a musical sound
- It gives instructions on which note to play, on which instrument and under which conditions
- Program that can create, edit and perform MIDI files are named *sequencer*
- *Synthesizers* interpret MIDI events and produce the corresponding audio output

MIDI music (2)

- MIDI files are relatively small compared to files containing sampled sound
- Quality may not be as high as sampled music (because the music is synthesized)
- The music may also sound mechanical

MIDI capabilities of Java API

- Java API provides MIDI sound capabilities in the *javax.sound.midi* package (starting with Java 2)
- Important classes are *Sequencer* and *Sequence*
- A Sequence object contains MIDI data
  - A data structure containing aggregates of MidiEvents, organized in time
  - MidiEvents are encapsulation of MidiMessages
- A Sequencer sends a Sequence to a MIDI synthesizer

Sequencer interface

- Provides methods for:
  - Loading sequence data from a MIDI file or a sequence object
  - Saving currently loaded sequence to a MIDI file
  - Starting and stopping playback and recording
  - Querying and setting the synchronization and timing parameters. Ex: plays at different tempos, mute tracks

Example: obtaining a Sequencer and playing a MIDI file

- Get a sequence for the MIDI file:
  - Sequence sequence = MidiSystem.getSequence(new File(filename));
- Get acces to a sequencer through the MidiSystem class and open it:
  - Sequencer = MidiSystem.getSequencer(); sequencer.open();
- Play the MIDI sequence:
  - sequencer.setSequence(sequence); sequencer.start();

Example: Looping through a MIDI file

- By default the sequencer plays the sequence only once.
- To play the music in a loop, we need to start again the sequencer upon receipt of a MIDI message signaling the end of the track
- To do that, our class needs to implement the interface *MetaEventListener*
- And override the method meta(MetaMessage m)

```
public MidiPlayer implements MetaEventListener {
  .....

  public void meta(MetaMessage m) {
   if (m.getType() == 47) {
       sequencer.start()
    }
  }

  .....

}
```

Modifying MIDI sequences

- In a game, we may want to make the music evolve with different situations:
  - Ex: increase tempo when more enemies
- It is easy to do that by modifying MIDI sequences
- Example of possible editing include:
  - Modify the tempo of the music (increase / decrease)
  - Change the instrument being used (remove / add tracks)

Changing the playback speed

- A sequence's speed is indicated by its tempo
- Changing the tempo can be done by the following Sequencer methods:
  - public void setTempoInBPM(float bpm) which sets the tempo to bpm in beats per minute
  - public void setTempoInMPQ(float mpq) which sets the tempo to mpq in microseconds per quarter note
  - public void setTempoFactor(float factor) which scale the current speed by factor
- Changing the tempo factor does not affect the current nominal speed

Muting or soloing tracks in the sequence

- The sequencer lets the user choose which tracks should sound during the playback
- For each track, you can decide to turn it on or off with the sequencer method: setTrackMute(int, boolean)
- The first argument corresponds to the track that you want to mute and the second argument to the state (true->mute, false->unmute)
- Mute requests can fail if the first argument is greater than the max number of available track or if the sequencer does not support muting
- The number of tracks can be obtained from the sequence with the method getTracks();

Summary

- Sound basics: sound wave, sampled audio
- Load and play a sound available as a sampled audio (ex: WAV, AIFF, AU)
- Apply special effects to the sound with filters
- Music:
  - Compressed sampled audio format (MP3, vorbis)
  - MIDI
- Load, play and modify MIDI files