Lecture 28

Game Audio
The Role of Audio in Games

Engagement

- **Entertains** the player
  - Music/Soundtrack
- **Enhances the** **realism**
  - Sound effects
- **Establishes** **atmosphere**
  - Ambient sounds
- **Other reasons?**
The Role of Audio in Games

Feedback

- **Indicate** off-screen action
  - Indicate player should move

- **Highlight** on-screen action
  - Call attention to an NPC

- **Increase** reaction time
  - Players react to sound faster

- **Other reasons?**
History of Sound in Games

• Arcade games
• Early handhelds
• Early consoles
Early Sounds: *Wizard of Wor*
History of Sound in Games

- Basic Sounds
  - Arcade games
  - Early handhelds
  - Early consoles

- Recorded Sound Samples
  - Starts w/ MIDI
  - 5th generation
  - 5th generation (Playstation)
  - Early PCs

Game Audio
History of Sound in Games

- Basic Sounds
  - Arcade games
  - Early handhelds
  - Early consoles

- Recorded Sound Samples
  - Starts w/ MIDI
  - 5th generation (Playstation)
  - Early PCs
  - Early PCs

- Some Variability of Samples
  - Sample selection
  - Volume
  - Pitch
  - Stereo pan
History of Sound in Games

- **Basic Sounds**
  - Arcade games
  - Early handhelds
  - Early consoles

- **Recorded Sound Samples**
  - Starts w/ MIDI
  - 5th generation (Playstation)
  - Early PCs

- **Some Variability of Samples**
  - Sample selection
  - Volume
  - Pitch
  - Stereo pan

- **More Variability of Samples**
  - Multiple samples
  - Reverb models
  - Sound filters
  - Surround sound
The Technical Challenges

- Sound **formats** are not (really) cross-platform
  - It is not as easy as choosing MP3
  - Android, iOS favor different formats

- Sound playback **APIs** are not standardized
  - LibGDX is a layer over many different APIs
  - So behavior is not the same on all platforms

- Sound playback crosses **frame boundaries**
  - Mixing sound with animation has challenges
### File Format vs Data Format

<table>
<thead>
<tr>
<th><strong>File Format</strong></th>
<th><strong>Data Format</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>• The data storage format</td>
<td>• The actual audio encoding</td>
</tr>
<tr>
<td>• Has data other than audio</td>
<td>• Basic audio codec</td>
</tr>
<tr>
<td>• Many have many encodings</td>
<td>• Bit rate (# of bits/unit time)</td>
</tr>
<tr>
<td>• .caf holds MP3 and PCM</td>
<td>• Sample rate (digitizes an analog signal)</td>
</tr>
<tr>
<td><strong>Examples:</strong></td>
<td><strong>Examples:</strong></td>
</tr>
<tr>
<td>• .mp3, .wav</td>
<td>• MP3, Linear PCM</td>
</tr>
<tr>
<td>• .aac, .mp4, .m4a (Apple)</td>
<td>• AAC, HE-AAC, ALAC</td>
</tr>
<tr>
<td>• .flac, .ogg (Linux)</td>
<td>• FLAC, Vorbis</td>
</tr>
</tbody>
</table>
# Data Formats and Platforms

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>iOS</th>
<th>Android</th>
</tr>
</thead>
<tbody>
<tr>
<td>MP3</td>
<td>You know what this is</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>(HE-)AAC</td>
<td>A lossy codec, Apple’s MP3 alternative</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Linear PCM</td>
<td>Completely uncompressed sound</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MIDI</td>
<td><strong>NOT SOUND</strong>; Data for an instrument</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Vorbis</td>
<td>Xiph.org’s alternative to MP3</td>
<td>Maybe</td>
<td>Yes</td>
</tr>
<tr>
<td>FLAC</td>
<td>Xiph.org’s alternative lossless codec</td>
<td>Maybe</td>
<td>Yes</td>
</tr>
<tr>
<td>ALAC</td>
<td>Apple’s lossless codec (but compressed)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>iLBC</td>
<td>Internet low bit-rate codec (VOIP)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>IMA4</td>
<td>Super compression for 16 bit audio</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>$\mu$-law</td>
<td>Like PCM, but optimized for speech</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
The Associated File Formats

<table>
<thead>
<tr>
<th>Format</th>
<th>File Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>MP3</td>
<td>.mp3</td>
</tr>
<tr>
<td>(HE-)AAC</td>
<td>.aac, .mp4, .m4a</td>
</tr>
<tr>
<td>Linear PCM</td>
<td>.wav</td>
</tr>
<tr>
<td>MIDI</td>
<td>.mid</td>
</tr>
</tbody>
</table>

- Any other file format is **not cross-platform**
- Apple/iOS is pushing the `.caf` file
  - Stands for Core Audio Format
  - Supports MP3, (HE-)AAC, PCM, ALAC, etc…
  - But not cross-platform
The Associated File Formats

<table>
<thead>
<tr>
<th>Format</th>
<th>File Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>MP3</td>
<td>.mp3</td>
</tr>
<tr>
<td>(HE-)AAC</td>
<td>.aac, .mp4, .m4a</td>
</tr>
<tr>
<td>Linear PCM</td>
<td>.wav Uncompressed</td>
</tr>
<tr>
<td>MIDI</td>
<td>.mid</td>
</tr>
</tbody>
</table>

- Any other format is not (completely) cross-platform
- Apple/iOS is pushing the (proprietary) .caf file
  - Stands for Core Audio Format
  - Supports MP3, (HE-)AAC, PCM, ALAC, etc…
- OGG has become a popular format for gaming
Linear PCM Format

- Sound data is an array of **sample** values

| 0.5 | 0.2 | -0.1 | 0.3 | -0.5 | 0.0 | -0.2 | -0.2 | 0.0 | -0.6 | 0.2 | -0.3 | 0.4 | 0.0 |

- A sample is an **amplitude** of a sound wave

- Values are normalized -1.0 to 1.0 (so they are floats)
Linear PCM Format

- Sound data is an array of \textit{sample} values

\begin{tabular}{|cccccccccccccc|}
\hline
0.5 & 0.2 & -0.1 & 0.3 & -0.5 & 0.0 & -0.2 & -0.2 & 0.0 & -0.6 & 0.2 & -0.3 & 0.4 & 0.0 \\
\hline
\end{tabular}

- A sample is an \textit{amplitude} of a sound wave

- Sometimes encoded as shorts or bytes MIN to MAX
Linear PCM Format

- Sound data is an array of **sample** values

| 0.5 | 0.2 | -0.1 | 0.3 | -0.5 | 0.0 | -0.2 | -0.2 | 0.0 | -0.6 | 0.2 | -0.3 | 0.4 | 0.0 |

- Magnitude of the amplitude is the volume
  - 0 is lowest volume (silence)
  - 1 is maximum volume of sound card
  - Multiply by number 0 to 1 to change global volume
Linear PCM Format

- Sound data is an array of **sample** values

<p>| | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>0.2</td>
<td>-0.1</td>
<td>0.3</td>
<td>-0.5</td>
<td>0.0</td>
<td>-0.2</td>
<td>-0.2</td>
<td>0.0</td>
<td>-0.6</td>
<td>0.2</td>
<td>-0.3</td>
</tr>
<tr>
<td>0.4</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Magnitude of the amplitude is the volume
  - 0 is lowest volume (silence)
  - 1 is maximum volume of sound card
  - Multiply by number 0 to 1 to change global volume
Linear PCM Format

- Samples are organized into (interleaved) **channels**

<table>
<thead>
<tr>
<th>0.5</th>
<th>0.2</th>
<th>-0.1</th>
<th>0.3</th>
<th>-0.5</th>
<th>0.0</th>
<th>-0.2</th>
<th>-0.2</th>
<th>0.0</th>
<th>-0.6</th>
<th>0.2</th>
<th>-0.3</th>
<th>0.4</th>
<th>0.0</th>
</tr>
</thead>
</table>

- Each channel is essentially a **speaker**
  - Mono sound has one channel
  - Stereo sound has two channels
  - 5.1 surround sound is **six** channels

- A **frame** is set of simultaneous samples
  - Each sample is in a separate frame

Game Audio
Linear PCM Format

• The sample rate is frames per second

1 second

# frames

• Example: 0.5 seconds of stereo at 44.1 kHZ
  • 0.5 s * 44100 f/s = 22050 frames
  • 2 samples/frame * 22050 frames = 44100 samples
  • 4 bytes/sample * 44100 samples = 176.4 kBytes

• 1 minute of stereo CD sound is 21 MB!
Playing Sound Directly

Game Loop

PCM data buffer

Sound Card

Game Audio
Playing Sound Directly

Game Loop

Write PCM chunk to buffer

PCM data buffer

Sound Card
Direct Sound in LibGDX: AudioDevice

- /**
  * Writes the array of float PCM samples to the audio device.
  *
  * This method blocks until they have been processed.
  */
  *\n  *\n  *\n  void writeSamples(float[] samples, int offset, int numSamples)

- /**
  * Writes array of 16-bit signed PCM samples to the audio device.
  *
  * This method blocks until they have been processed.
  */
  *\n  *\n  *\n  void writeSamples(short[] samples, int offset, int numSamples)
Direct Sound in LibGDX: AudioDevice

- /**
  * Writes the array of float PCM samples to the audio device.
  *
  * This method blocks until they have been processed.
  */
  
  void writeSamples(float[] samples, int offset, int numSamples)

- /**
  * Writes array of 16-bit signed PCM samples to the audio device.
  *
  * This method blocks until they have been processed.
  */
  
  void writeSamples(short[] samples, int offset, int numSamples)

Requires separate audio thread
The Latency Problem

- Buffer is really a *queue*
  - Output from queue front
  - Playback writes to end
  - Creates a *playback delay*

- **Latency**: amount of delay
  - Some latency must exist
  - Okay if latency $\leq$ framerate
  - **Android latency is $\sim$90 ms!**

- Buffering is a necessary evil
  - Keeps playback smooth
  - Allows real-time *effects*
Playing Sound Directly

Choice of buffer size is important!

- **Too large**: long latency until next sound plays
- **Too small**: buffers swap too fast, causing audible pops
How Streaming Works

- All sound cards only play PCM data
  - Other files (MP3 etc.) are decoded into PCM data
  - But the data is *paged-in* like memory in an OS

- This is how OGG is added to most engines

![Diagram of sound streaming](image)
How Streaming Works

- **Sound**: Sound asset that is *preloaded* as full PCM
- **Music**: Sound asset that is *streamed* as PCM pages

*Page size* set by file format

*Chunk size* set by audio API
Handling Multiple Sounds

PCM Data

PCM Data

PCM Data

PCM Data

PCM Data

Literally!

Sound Card
Handling Multiple Sounds

- Can create values outside of -1 to 1
  - This causes clipping/distortion
  - Common if many simultaneous sounds
- Audio engineer must balance properly

Literally!
We Want Something Simpler!

- Want ability to **play** and **track** sounds
  - Functions to load sound into card buffer
  - Functions to detect if sound has finished

- Want ability to **modify** active sounds
  - Functions for volume and pitch adjustment
  - Functions for stereo panning (e.g. left/right channels)
  - Functions to pause, resume, or loop sound

- Want ability to **mix** sounds together
  - Functions to add together sound data quickly
  - Background process for dynamic volume adjustment
We Want Something Simpler!

- Want ability to **play** and **track** sounds
  - Functions to load sound into card buffer
  - Functions to detect if sound has finished

- Want ability to **modify** active sound
  - Functions for volume and pitch adjustment
  - Functions for stereo panning (e.g. left/right channels)
  - Functions to pause, resume, or loop sound

- Want ability to **mix** sounds together
  - Functions to add together sound data quickly
  - Background process for dynamic volume adjustment

This is the purpose of a **sound engine**
Standard Industry Sound Engines

- **OpenAL**
  - Created in 2000 by Loki Software for Linux
  - Was an attempt to make a sound standard
  - Loki went under; last stable release in 2005
  - Still used heavily in the Indie space

- **FMOD/WWISE**
  - Industry standard for game development
  - Mobile support is possible but not easy
  - Not free; but no cost for low-volume sales
## The LibGDX Sound Classes

<table>
<thead>
<tr>
<th>Sound</th>
<th>Music</th>
</tr>
</thead>
</table>
| • Primary method is `play()`  
  • Returns a long integer  
  • Represents sound *instance*  
  • `loop()` is a separate method  
  • Has **no public constructor**  
  • Use `Audio.newSound(f)`  
  • Audio can cache/preload  
  • Must dispose when done | • Primary method is `play()`  
  • This is a *void* method  
  • Only allows **one instance**  
  • `loop` is an attribute of music  
  • Has **no public constructor**  
  • Use `Audio.newMusic(f)`  
  • Audio can cache the file  
  • Must dispose when done |
Playing a Sound

- Playback may include **multiple sounds**
  - Sounds may play simultaneously (offset)
  - Simultaneous sounds may be same asset
  - **Asset (source) vs. Instance (playback)**

- Playback crosses **frame boundaries**
  - It may span multiple animation frames
  - Need to know when it stops playing
  - May need to stop (or pause) it early
Classic Model: Channels

Channel
Channel
Channel
...
Channel
Channel
Classic Model: Channels

Engine has fixed number of channels (historically 24)
**Classic Model: Channels**

- **Engine has fixed number of channels (historically 24)**
- **Load sound into channel to play it**
- **Channel**
- **Channel**
- **Channel**
- ...
Classic Model: Channels

Engine has fixed number of channels (historically 24)

Load sound into channel to play it
Playing a Sound with Channels

- **Request** a sound channel for your asset
  - If none is available, sound fails to play
  - Otherwise, it gives you an ID for a channel

- **Load** asset into the channel (but might stream)

- **Play** the sound channel
  - Playing is a property of the channel, not asset
  - Channel has other properties, like volume

- **Release** the channel when the sound is done
  - This is usually done automatically
Need to remember channel id
The Sound API

- /**
  * @return channel id for sound playback
  *
  * If no channel is available, returns -1
  * @param volume The sound volume
  * @param pitch The pitch multiplier (>1 faster, <1 slower)
  * @param pan The speaker pan (-1 full left, 1 full right)
  */
  
  public long play(float volume, float pitch, float pan);

- public void stop(long audioID);

- public void resume(long audioID);

- public void setLooping(long audioID, boolean loop);

- Public void setVolume(long audioID, float volume);
The Sound API

- /**
  * @return channel id for sound playback
  * If no channel is available, returns -1
  * @param volume  The sound volume
  * @param pitch     The pitch multiplier (>1 faster, <1 slower)
  * @param pan       The speaker pan (-1 full left, 1 full right)
  */
  public long play(float volume, float pitch, float pan);

- public void stop(long audioID);
- public void resume(long audioID);
- public void setLooping(long audioID, boolean loop);
- Public void setVolume(long audioID, float volume);

Returns available channel id

Need to remember channel id
Why This is Undesirable

- Tightly couples architecture to sound engine
  - All controllers need to know this channel id
  - Playback must communicate the id to all controllers

- Instances usually have a *semantic meaning*
  - **Example**: Torpedo #3, Ship/crate collision
  - Meaning is independent of the channel assigned
  - Would prefer to represent them by this meaning

- **Solution**: Refer to instances by *keys*
The SoundController Class (Lab 4)

- /**
  * @return true if the given sound could be played
  *
  * @param key the reference key for the sound effect
  * @param file the sound effect file to play
  * @param loop whether to loop indefinitely
  * @param volume the sound volume
  */

  public boolean play(string key, string file, bool loop, float volume);

- public void stop(string key);

- public void isActive(string key);

- Other methods I forgot to write

Refer to instance logically
Stopping Sounds

- Would like to know when a sound is finished
  - To free up the channel (if not automatic)
  - To stop any associated animation
  - To start a follow-up sound

- Two main approaches
  - **Polling**: Call an `isPlaying()` method
  - **Callback**: Pass a function when play

*Cannot do in android.media*
Stopping Sounds

- Would like to know when a sound is finished
  - To free up the channel (if not automatic)
  - To stop any associated animation
  - To start a follow-up sound

- Two main approaches
  - **Polling**: Call an `isPlaying()` method
  - **Callback**: Pass a function when play

- LibGDX cannot tell you anything!!!
SoundController: The Ugly Hacks

- /**
  * Sets the maximum # of frames a sound can run
  */
  public void setTimeLimit(long timelimit);

- /**
  * Sets the number of frames before a key can be reused
  */
  public void setCoolDown(long cooldown);

- /**
  * Sets the maximum # of sounds per animation frame
  */
  public void setFrameLimit(int framelimit);
SoundController: The Ugly Hacks

- /**
  * Sets the maximum # of frames a sound can run
  */
  public void setTimeLimit(long timelimit);

- /**
  * Sets the number of frames before a key can be reused
  */
  public void setCoolDown(long cooldown);

- /**
  * Sets the maximum # of sounds per animation frame
  */
  public void setFrameLimit(int framelimit);
SoundController: The Ugly Hacks

- /**
  * Sets the maximum # of frames a sound can run
  */
  public void setTimeLimit(long timelimit);

- /**
  * Sets the number of frames before a key can be reused
  */
  public void setCoolDown(long cooldown);

- /**
  * Sets the maximum # of sounds per animation frame
  */
  public void setFrameLimit(int framelimit);

Garbage collect done sounds

Prevent stopping recent sounds

Game Audio
SoundController: The Ugly Hacks

- /**
  * Sets the maximum # of frames a sound can run
  */
  public void setTimeLimit(long timelimit);

- /**
  * Sets the number of frames before a key can be reused
  */
  public void setCoolDown(long cooldown);

- /**
  * Sets the maximum # of sounds per animation frame
  */
  public void setFrameLimit(int framelimit);

- Garbage collect done sounds
- Prevent stopping recent sounds
- Limit overhead of changing mixer graph
Problem with the Channel Model

- All controls are embedded in the channel
  - **Example:** Volume, looping, play position
  - Restricted to a *predetermined* set of controls

- Modern games want *custom sound-processing*
  - User defined sound filters (low pass, reverb)
  - Advanced equalizer support
  - Support for surround and 3D sound
  - Procedural sound generation
DSP Processing: The Mixer DAG
Idea: Sound Instance is a Sub-DAG
Idea: Sound Instance is a Sub-DAG

Source → Effect → Mixer → Effect → Source

Sound Instance

Sound Engine

Load instance into DAG slot
Idea: Sound Instance is a Sub-DAG

Channel model is a special case of this DAG
Example: UDK Kismet
Example: FMOD
Idea: Sound Instance is a Sub-DAG

Android support is why LibGDX cannot do this.
Summary

• Audio design is about creating soundscapes
  • Music, sound effects, and dialogue
  • Combining sounds requires a sound engine

• Cross-platform support is a problem
  • Licensing issues prevent a cross-platform format
  • Very little standardization in sound APIs

• Best engines use digital signal processing (DSP)
  • Mixer graph is a DAG supporting sound effects
  • Android prevents us from doing this in LibGDX