

Lecture 9

Game Audio

History of Sound in Games

Basic Sounds

- Arcade games
- Early handhelds
- Early consoles

History of Sound in Games

Basic
Sounds



Recorded
Sound
Samples

Sample = pre-recorded audio

- Arcade games
- Early handhelds
- Early consoles
- Starts w/ MIDI
- 5th generation
(Playstation)
- Early PCs

History of Sound in Games

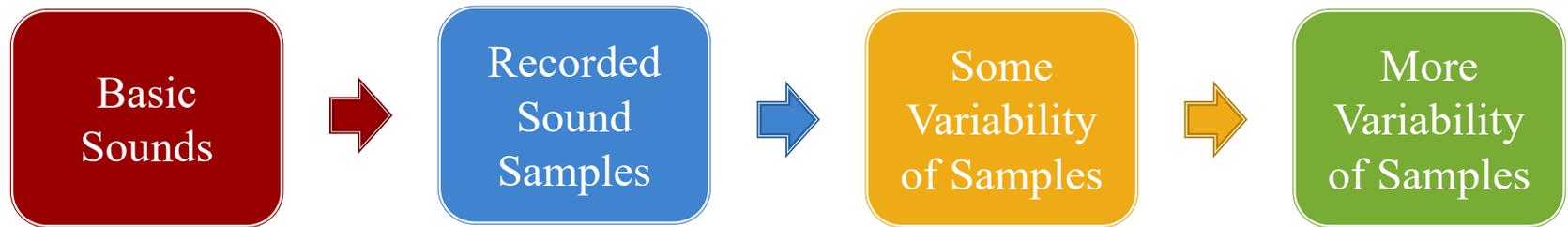


- Arcade games
- Early handhelds
- Early consoles

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

- Sample selection
- Volume
- Pitch
- Stereo pan

History of Sound in Games



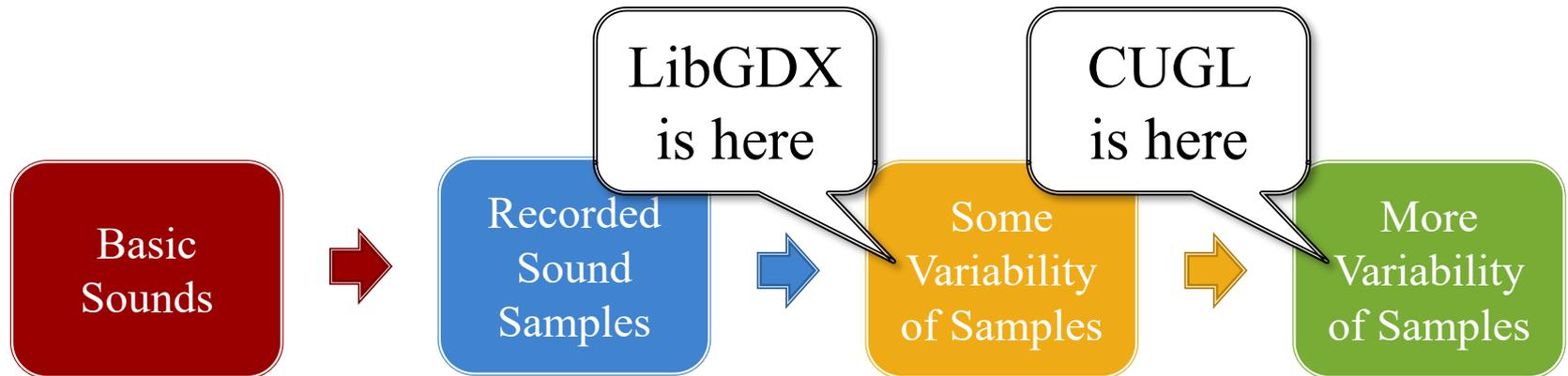
- Arcade games
- Early handhelds
- Early consoles

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

- Sample selection
- Volume
- Pitch
- Stereo pan

- Multiple samples
- Reverb models
- Sound filters
- Surround sound

History of Sound in Games



- Arcade games
- Early handhelds
- Early consoles
- Starts w/ MIDI
- 5th generation (Playstation)
- Early PCs
- Sample selection
- Volume
- Pitch
- Stereo pan
- 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
 - Different platforms favor different formats
- Sound playback **APIs** are not standardized
 - LibGDX & CUGL are layered over many APIs
 - Behavior is not the same on all platforms
- Sound playback crosses **frame boundaries**
 - Mixing sound with animation has challenges

File Format vs Data Format

File Format

- The data storage format
 - Has data other than audio
- Many have many encodings
 - .caf holds MP3 *and* PCM
- **Examples:**
 - .mp3, .wav, .aiff
 - .aac, .mp4, .m4a (Apple)
 - .flac, .ogg (Linux)

Data Format

- The actual audio encoding
 - Basic audio codec
 - Bit rate (# of bits/unit time)
 - Sample rate (digitizes an analog signal)
- **Examples:**
 - MP3, Linear PCM
 - AAC, HE-AAC, ALAC
 - FLAC, Vorbis

Game Audio Formats

Format	Description	File Formats
Linear PCM	Completely uncompressed sound	.wav, .aiff
MP3	A popular compressed, lossy codec	.mp3, .wav
Vorbis	Xiph.org's alternative to MP3	.ogg
FLAC	Xiph.org's compressed, lossless codec	.flac, .ogg
MIDI	NOT SOUND ; Data for an instrument	.midi
(HE-)AAC	A lossy codec, Apple's MP3 alternative	.aac, .mp4, .m4a
ALAC	Apple's lossless codec (but compressed)	.alac, .mp4, .m4a

MP3 largely avoided due to patent issues.

Game Audio Formats

Format	Description	File Formats
Linear PCM	Completely uncompressed sound	.wav, .aiff
MP3	Apple's	.mp3, .wav
Vorbis	Xiph.	.ogg
FLAC	Xiph.org's compressed, lossless codec	.flac, .ogg
MIDI	NOT SOUND ; Data for an instrument	.midi
(HE-)AAC	A lossy codec, Apple's MP3 alternative	.aac, .mp4, .m4a
ALAC	Apple's lossless codec (but compressed)	.alac, .mp4, .m4a

Supported in CUGL

MP3 largely avoided due to patent issues.

Which Formats Should You Choose?

- **Question 1:** Streaming or no streaming?
 - Audio gets large fast; music often streamed
 - But streaming creates overhead; bad for sound fx
 - Few engines support WAV streams (LibGDX & CUGL do)
- **Question 2:** Lossy or lossless compression?
 - Music can be lossy; sound fx not so much
 - Only FLAC and WAV are standard lossless
- **Question 3:** How many channels (speakers) needed?
 - MP3 channel is *stereo only*
 - Others support many channels (e.g. 7.1 surround)

Which Formats Should You Choose?

- **Question 1:** Streaming or no streaming?

- Audio gets large fast; music often streamed

- But

- Few

- **Question 2:**

- Mu

- Onl

Sound FX: Linear PCM/WAV

Music: OGG Vorbis

- **Question 3:** How many channels (speakers) needed?

- MP3 channel is *stereo only*

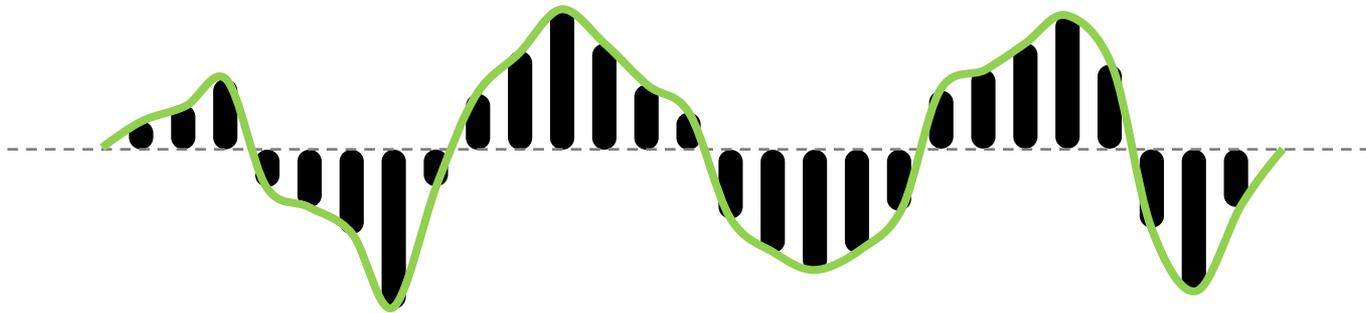
- Others support many channels (e.g. 7.1 surround)

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



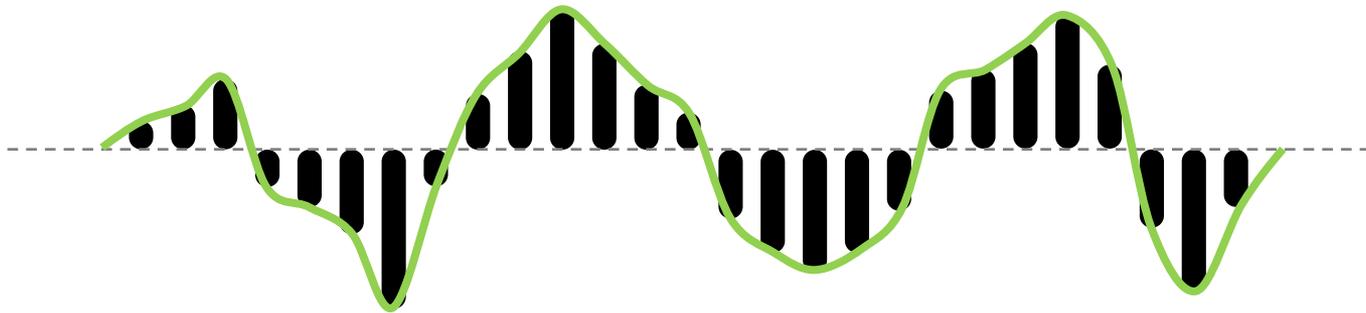
- Values are normalized -1.0 to 1.0 (so they are floats)

Linear PCM Format

- Sound data is an array of **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

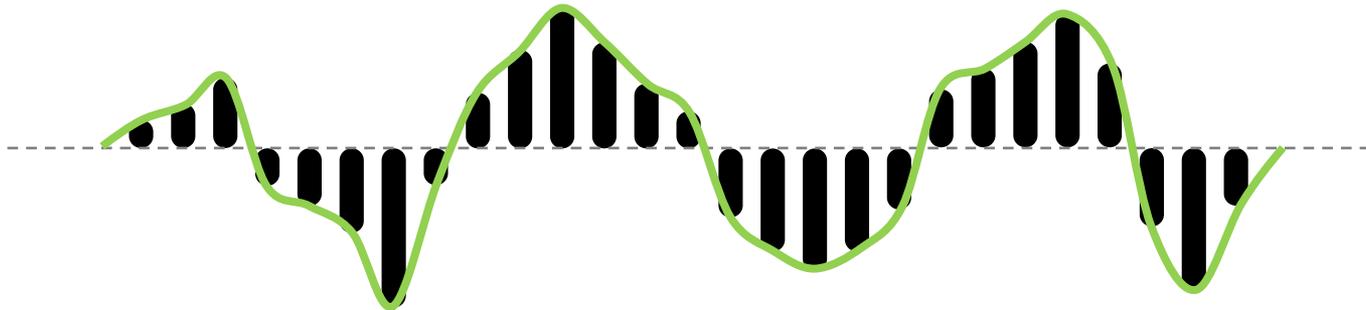


- Sometimes encoded as shorts or bytes MIN to MAX

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
-----	-----	------	-----	------	-----	------	------	-----	------	-----	------	-----	-----

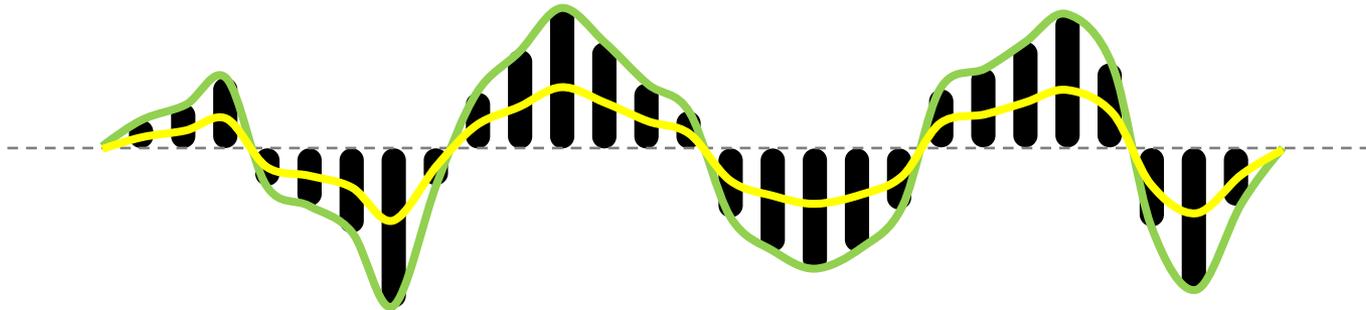


- Magnitude of the amplitude is the volume
 - 0 is lowest volume (silence)
 - 1 is maximum volume of sound card
 - Multiply by number 0 to 1 to change global volume

Linear PCM Format

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

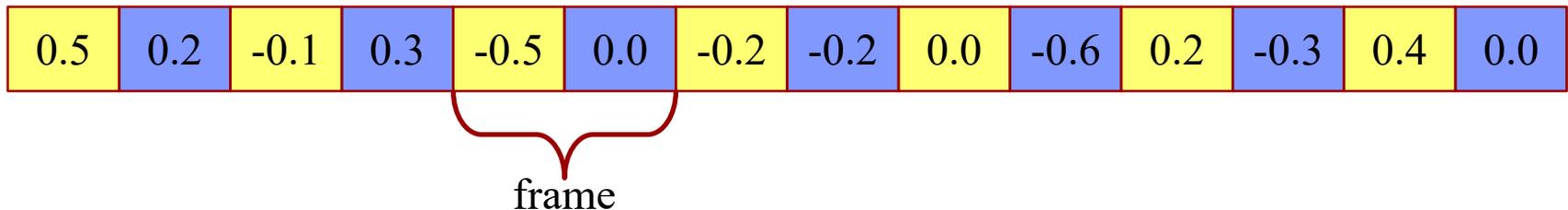
0.5	0.2	-0.1	0.3	-0.5	0.0	-0.2	-0.2	0.0	-0.6	0.2	-0.3	0.4	0.0
-----	-----	------	-----	------	-----	------	------	-----	------	-----	------	-----	-----



- 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

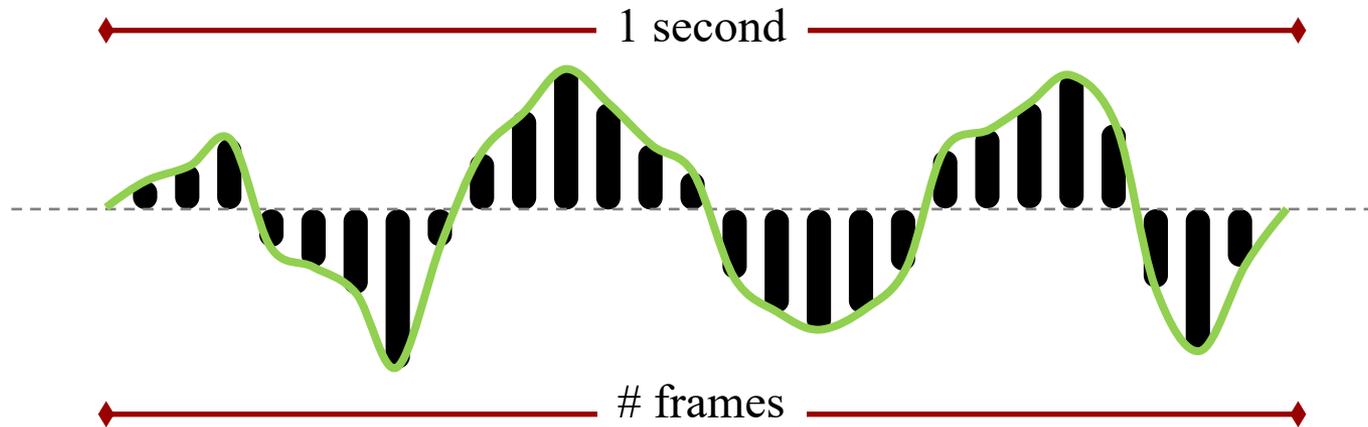
- Samples are organized into (interleaved) **channels**



- Each channel is essentially a **speaker**
 - Mono sound has one channel
 - Stereo sound has two channels
 - 7.1 surround sound is *eight* channels
- A **frame** is set of simultaneous samples
 - Each sample is in a separate frame

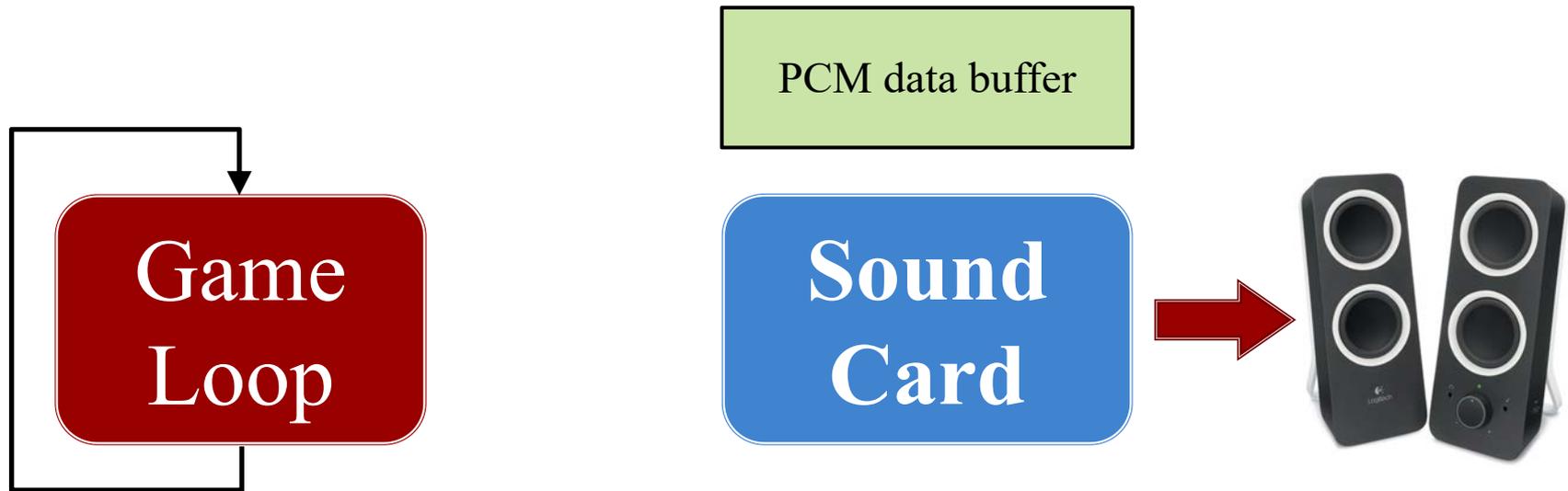
Linear PCM Format

- The sample rate is frames per second

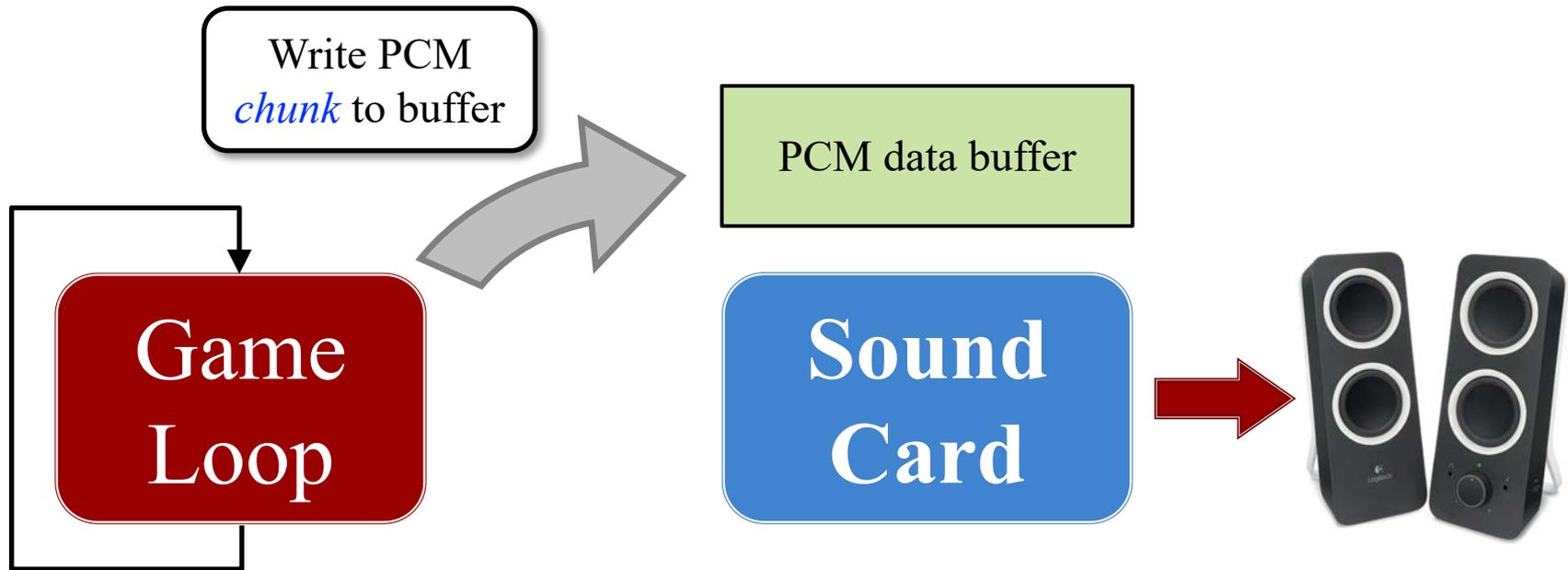


- **Example:** 0.5 seconds of stereo at 44.1 kHz
 - $0.5 \text{ s} * 44100 \text{ f/s} = 22050 \text{ frames}$
 - $2 \text{ samples/frame} * 22050 \text{ frames} = 44100 \text{ samples}$
 - $4 \text{ bytes/sample} * 44100 \text{ samples} = 176.4 \text{ kBytes}$
- 1 minute of stereo CD sound is 21 MB!

Playing Sound Directly



Playing Sound Directly



Direct Sound in CUGL: AudioNode

- Class representing an audio **source instance**
 - Not the same as Sound, which is an *asset*
 - sound->`createNode()` returns an instance node
 - Plug node into an AudioOutput (device)
- Data is read from method

```
/**
```

```
* Reads up to the specified number of frames into the given buffer
```

```
*
```

```
* @param buffer    The read buffer to store the results
```

```
* @param frames    The maximum number of frames to read
```

```
*/
```

```
Uint32 AudioNode::read(float* buffer, Uint32 frames);
```

Direct Sound in CUGL: AudioNode

- Class representing an audio **source instance**

- Not the same as Sound, which is an *asset*

- sound->`createNode()` returns an

- Plug node into an AudioOutput

Called in separate
audio thread

- Data is read from method

```
/**
```

```
* Reads up to the specified number of frames into the given buffer
```

```
*
```

```
* @param buffer    The read buffer to store the results
```

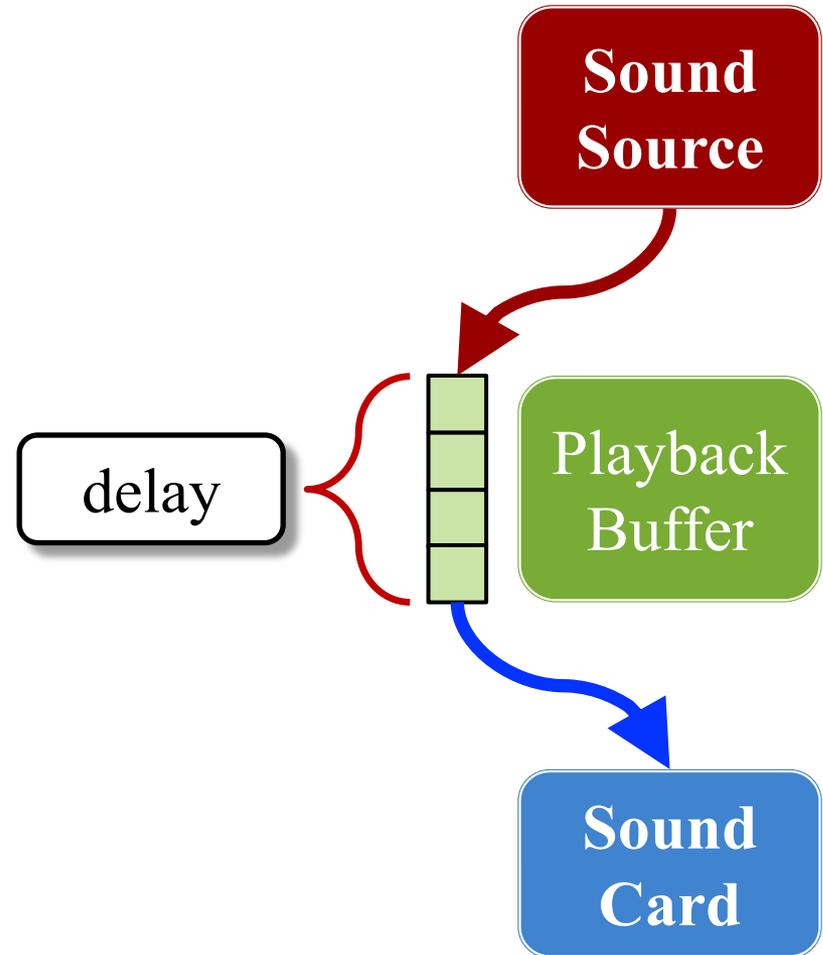
```
* @param frames    The maximum number of frames to read
```

```
*/
```

```
Uint32 AudioNode::read(float* buffer, Uint32 frames);
```

The Latency Problem

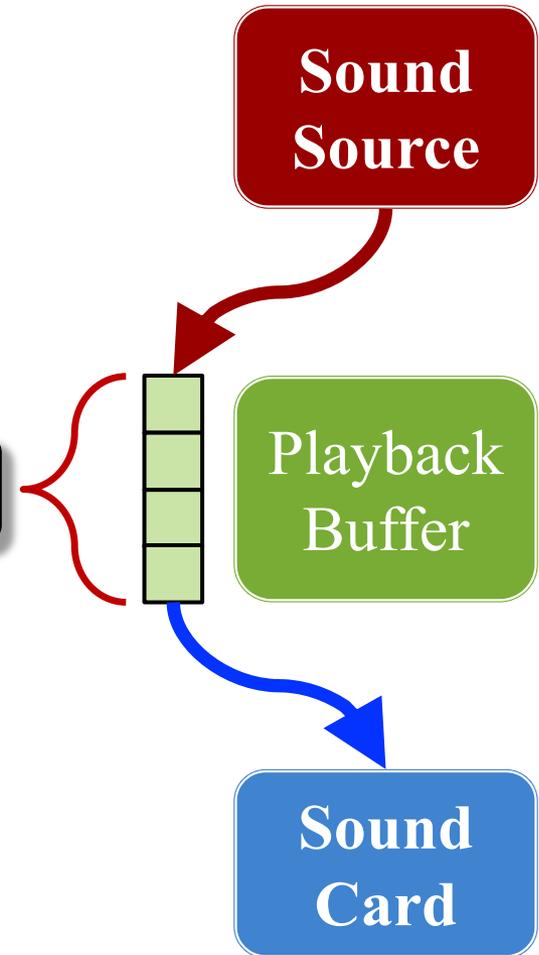
- Buffer is really a *queue*
 - Output from queue front
 - Playback writes to end
 - Creates a *playback delay*
- **Latency**: amount of delay
 - Some latency must exist
 - Okay if latency \leq framerate
 - CUGL target: **12 ms**
- Buffering is a necessary evil
 - Keeps playback smooth
 - Allows real-time *effects*



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 frame
 - CUGL target: **12 ms**
- Buffering is a necessary evil
 - Keeps playback smooth
 - Allows real-time *effects*

But SDL3 abstraction makes hard to measure

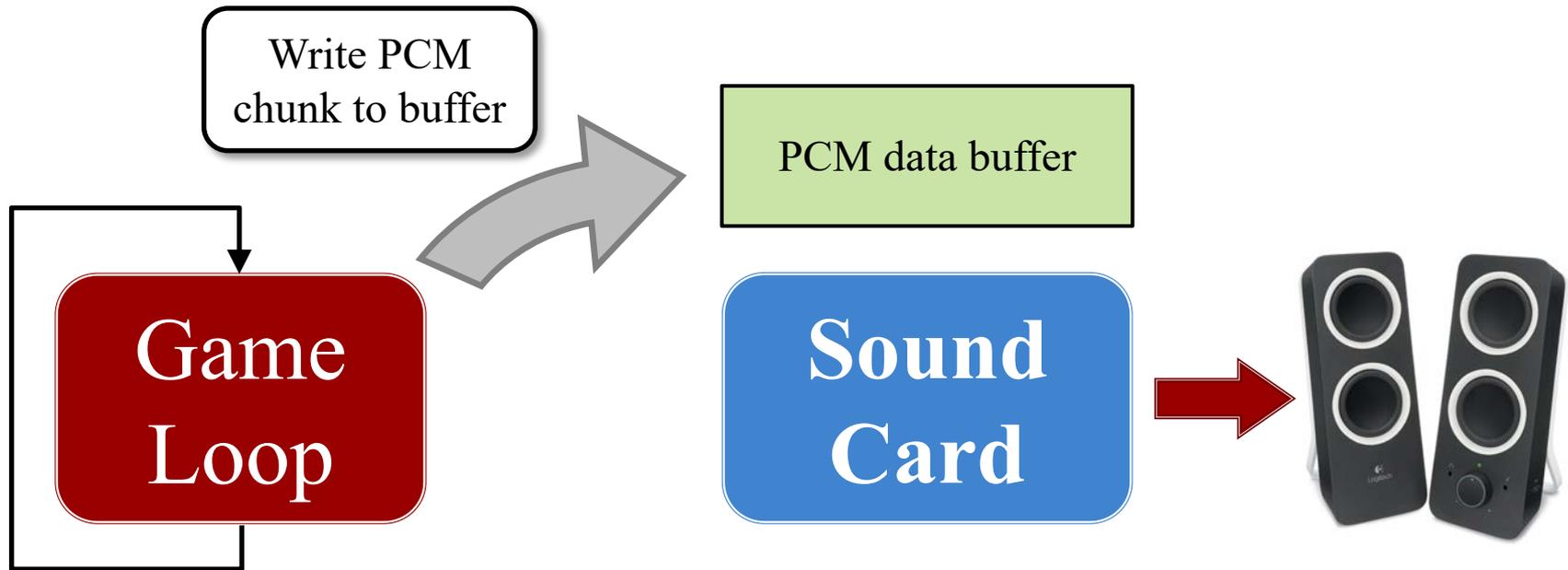


Aside: Calibration

- Rhythm games need **calibration**
 - Measure when player hears beat
 - AND when they see an image
 - Similar to a “clap board”
- **Experimental** measurement of *latency*
 - But the results can be very noisy
 - Players may be **late**, even with simple beats
 - Players may anticipate beat and be **early**!
- Requires a lot of fine tuning



Playing Sound Directly

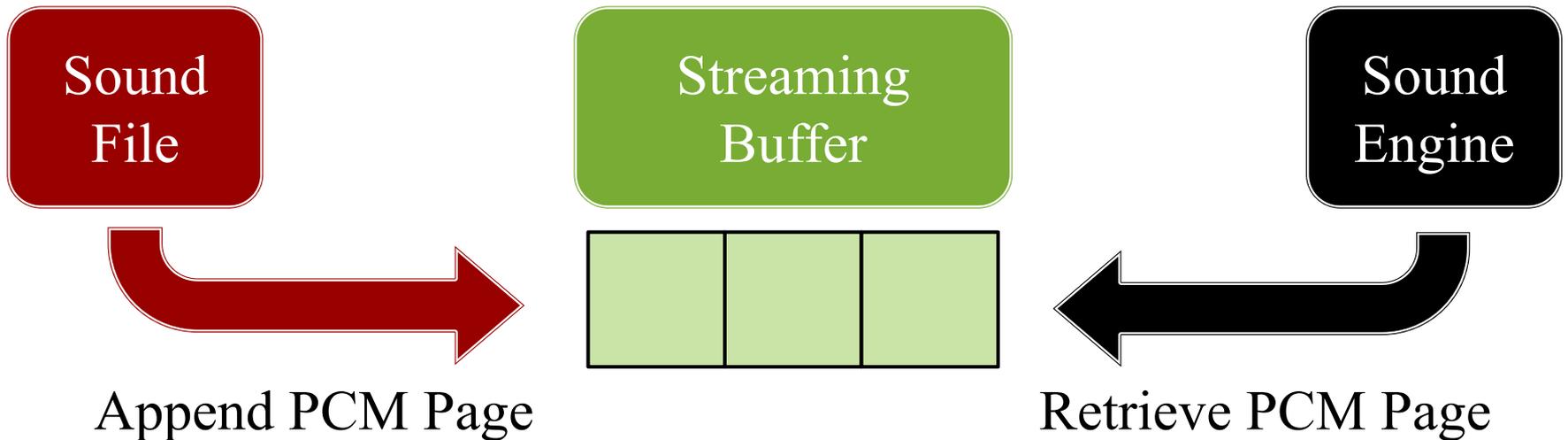


Choice of buffer size is important!

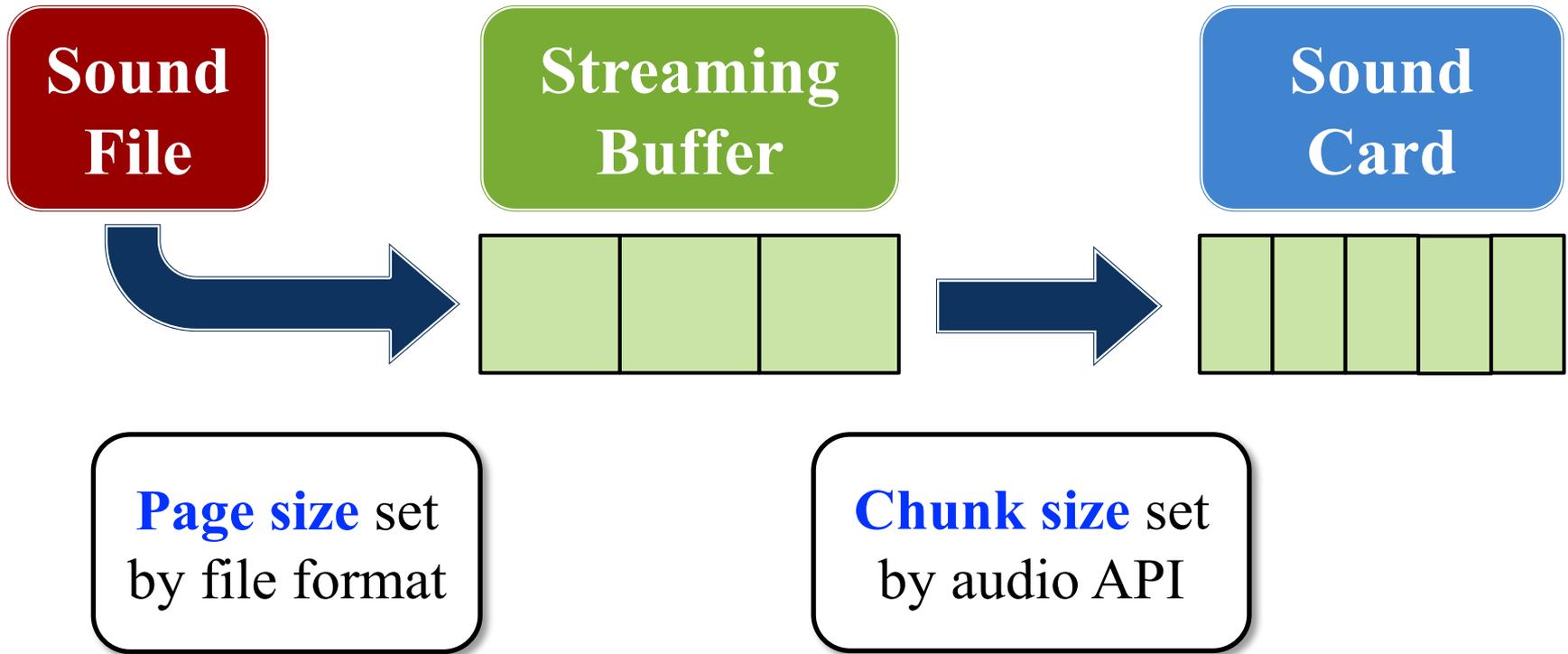
- **Too large:** *long* latency until next sound plays
- **Too small:** buffers swap too fast, causing audible pops

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
- Why LibGDX/CUGL can stream WAV files too!

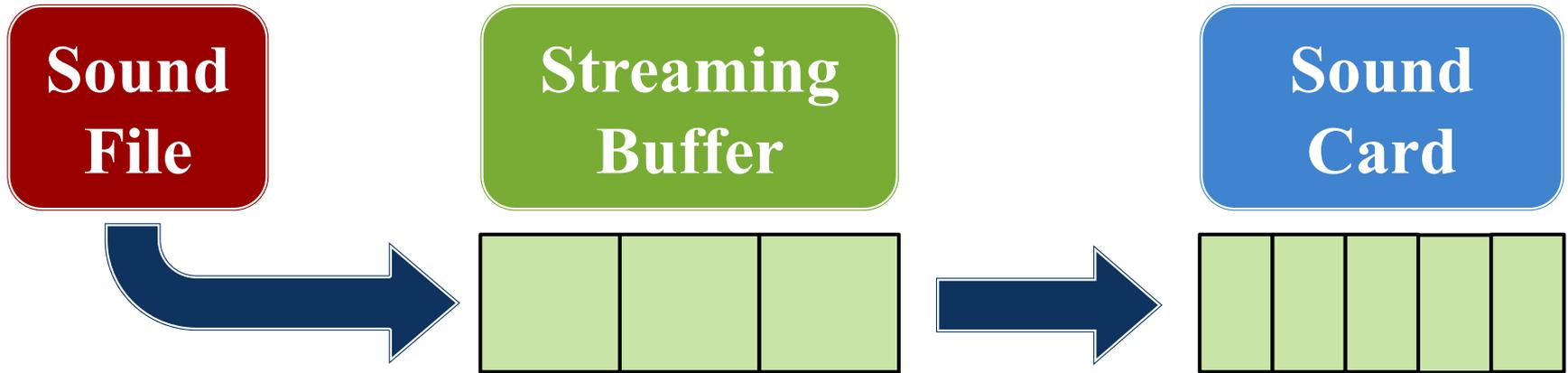


How Streaming Works



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

How Streaming Works



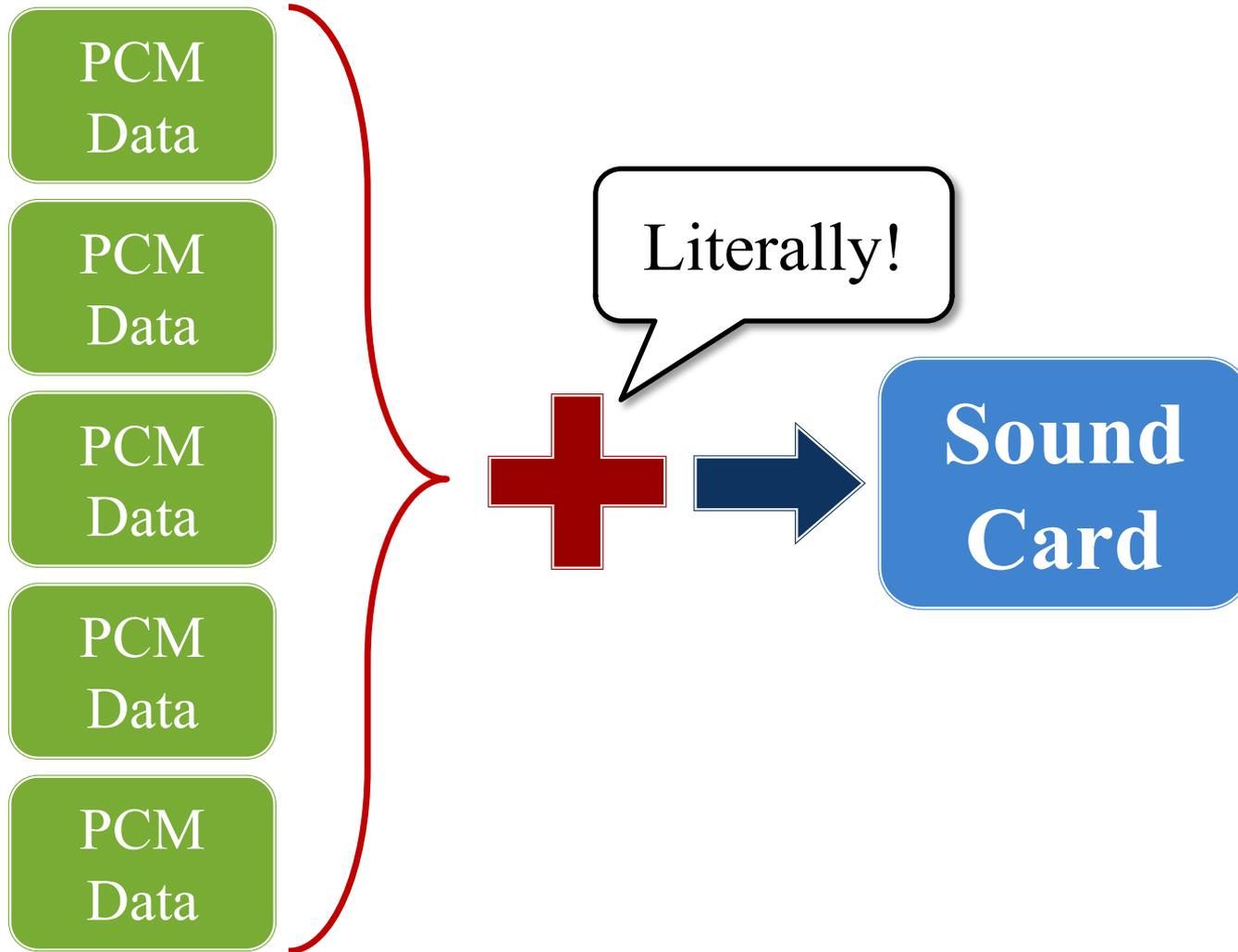
Page size set by file format

Chunk size set by audio API

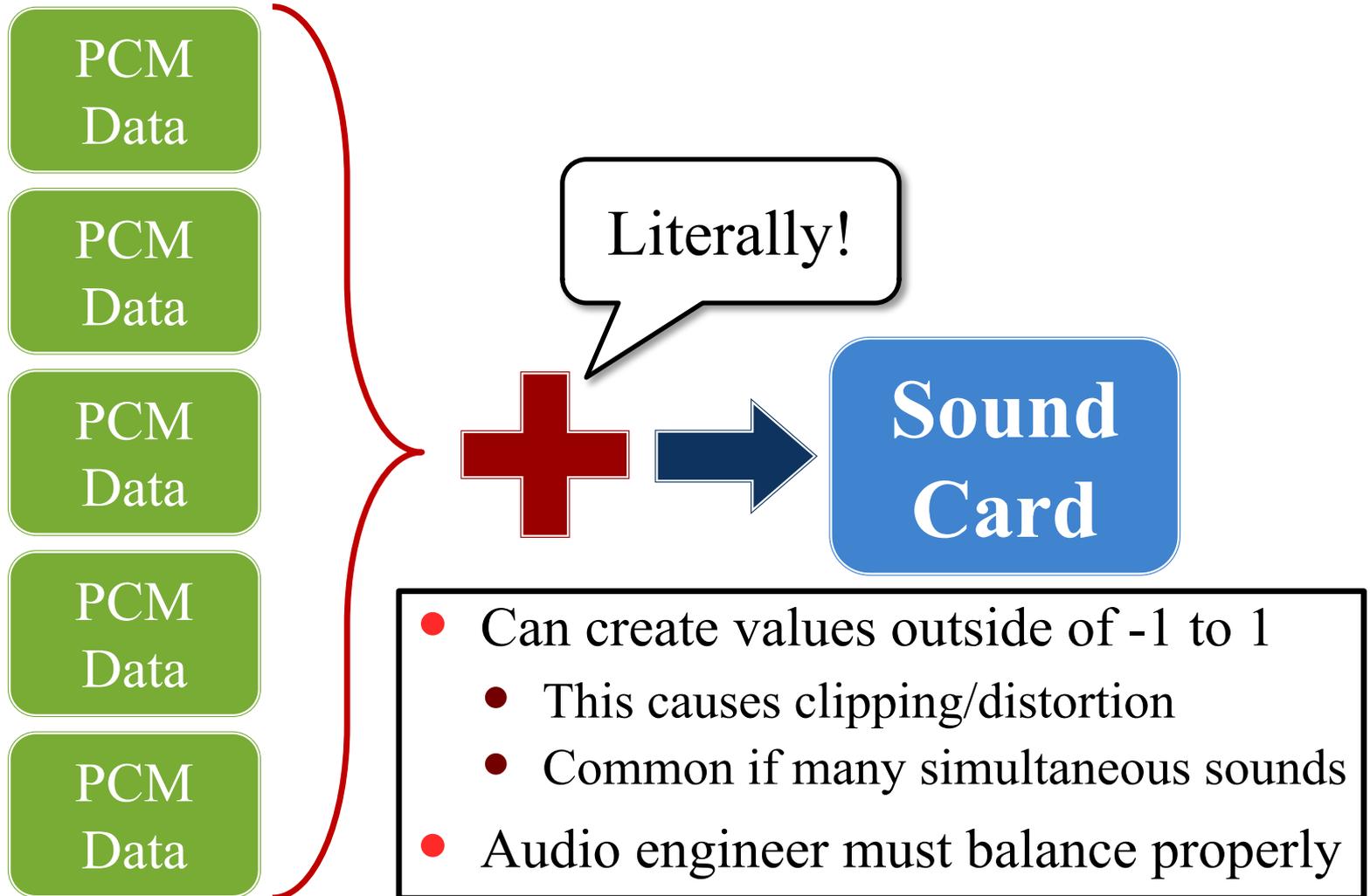
- **Sound:** Sound as full PCM
- **Music:** Sound as PCM pages

LibGDX distinction; less true in CUGL

Handling Multiple Sounds



Handling Multiple Sounds



Why is Mixing Hard?

- 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

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 sounds

- Functions to

This is the purpose of a **sound engine**

pause, resume, or loop sound

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

The SDL Sound Engine

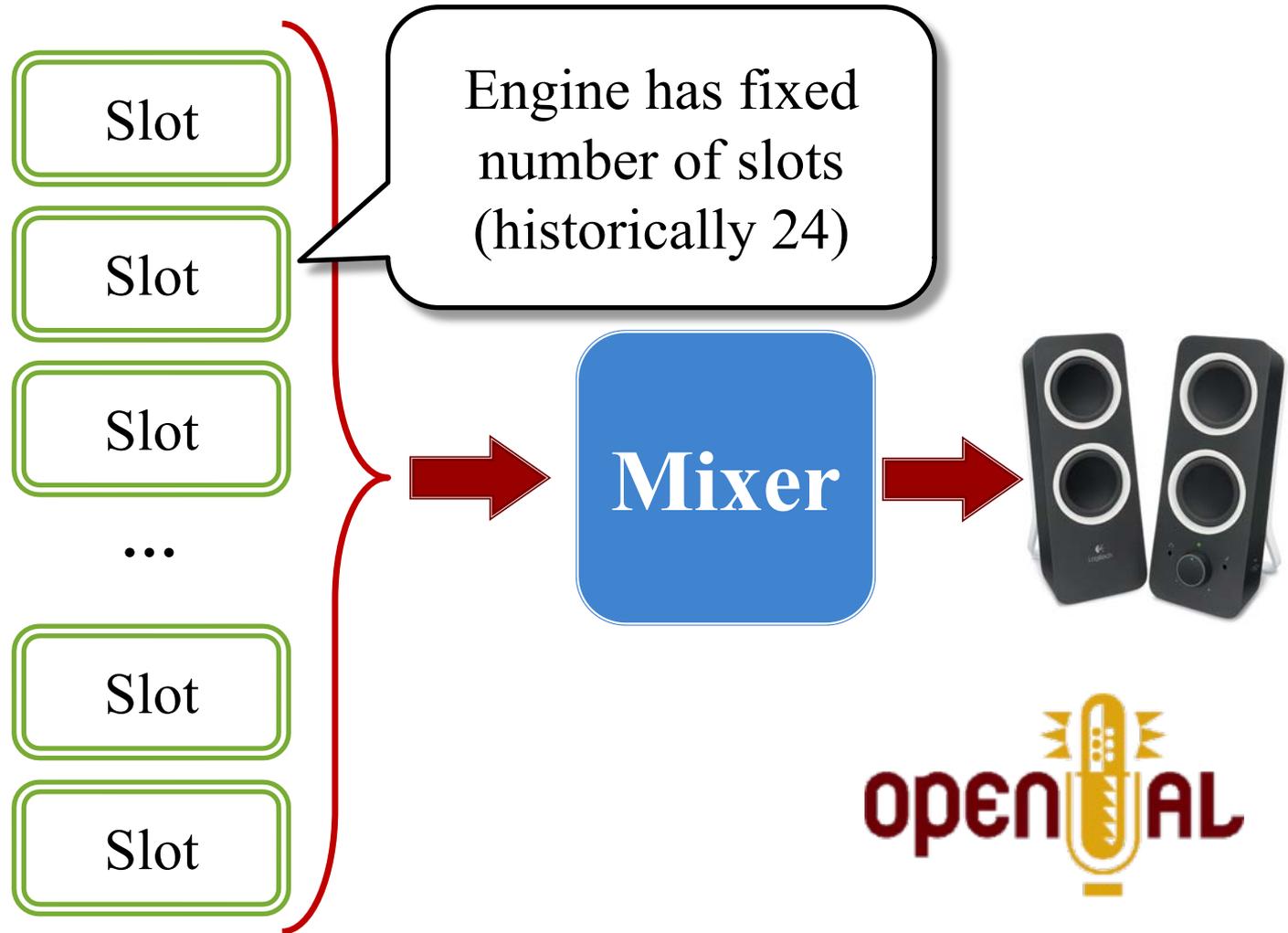
- CUGL is on top of SDL
 - SDL has its own audio API
 - Works on all platforms
- But it is an **extremely** low-level API
 - Fill the buffer with linear PCM data
 - Either pull (callback) or push (queue)
 - No support for non-WAV audio formats
 - No support for mixing, pausing, or **anything**



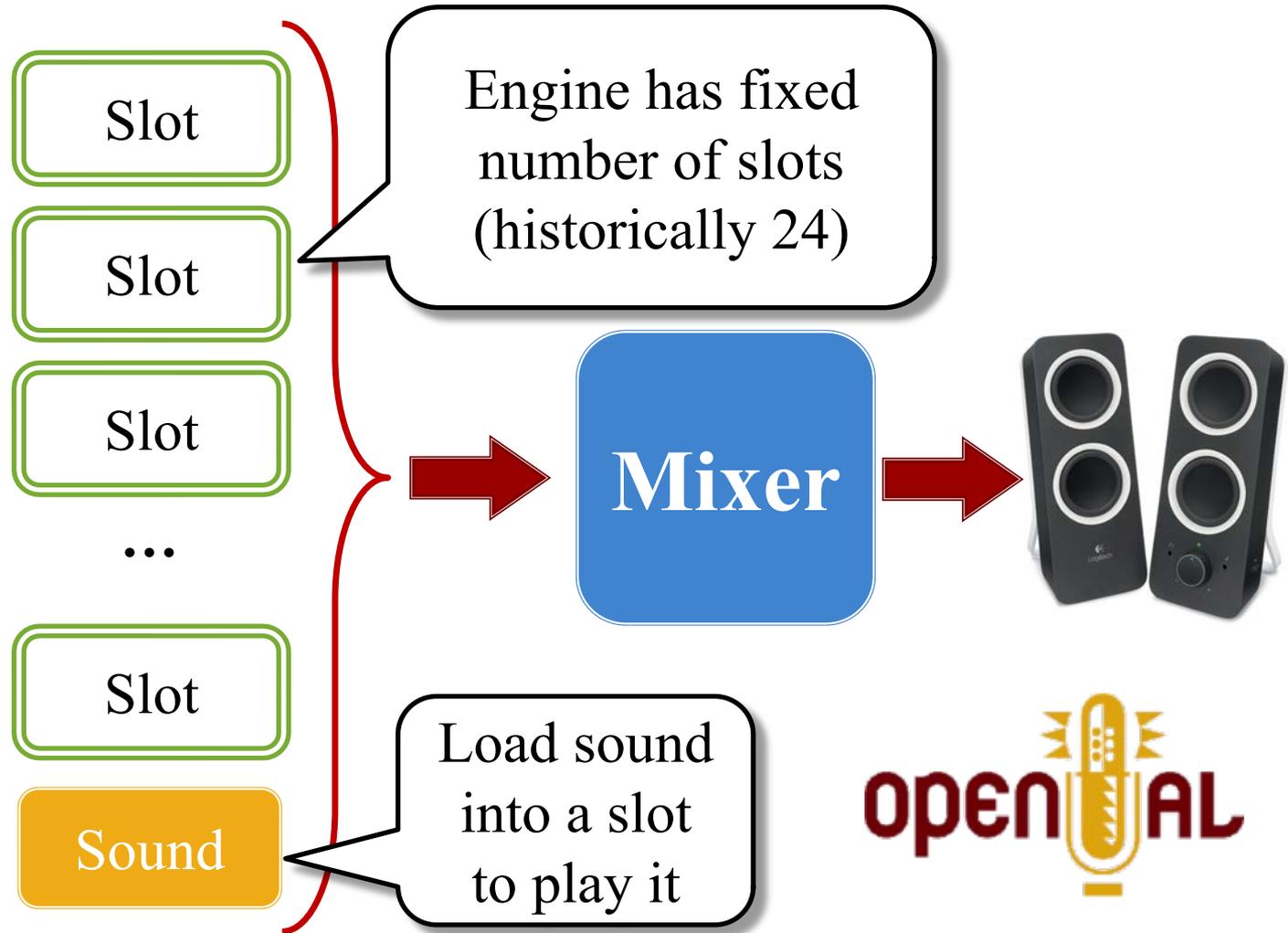
Solution: CUGL Audio Classes

- **AudioEngine:** Playing sound effects
 - Built on the the OpenAL model
 - Very easy to use and understand
 - Designed for simultaneous sounds
- **AudioQueue:** Playing music sequences
 - Accessed from the AudioEngine
 - Creates seamless playback queues
 - Ideal for long-running music loops

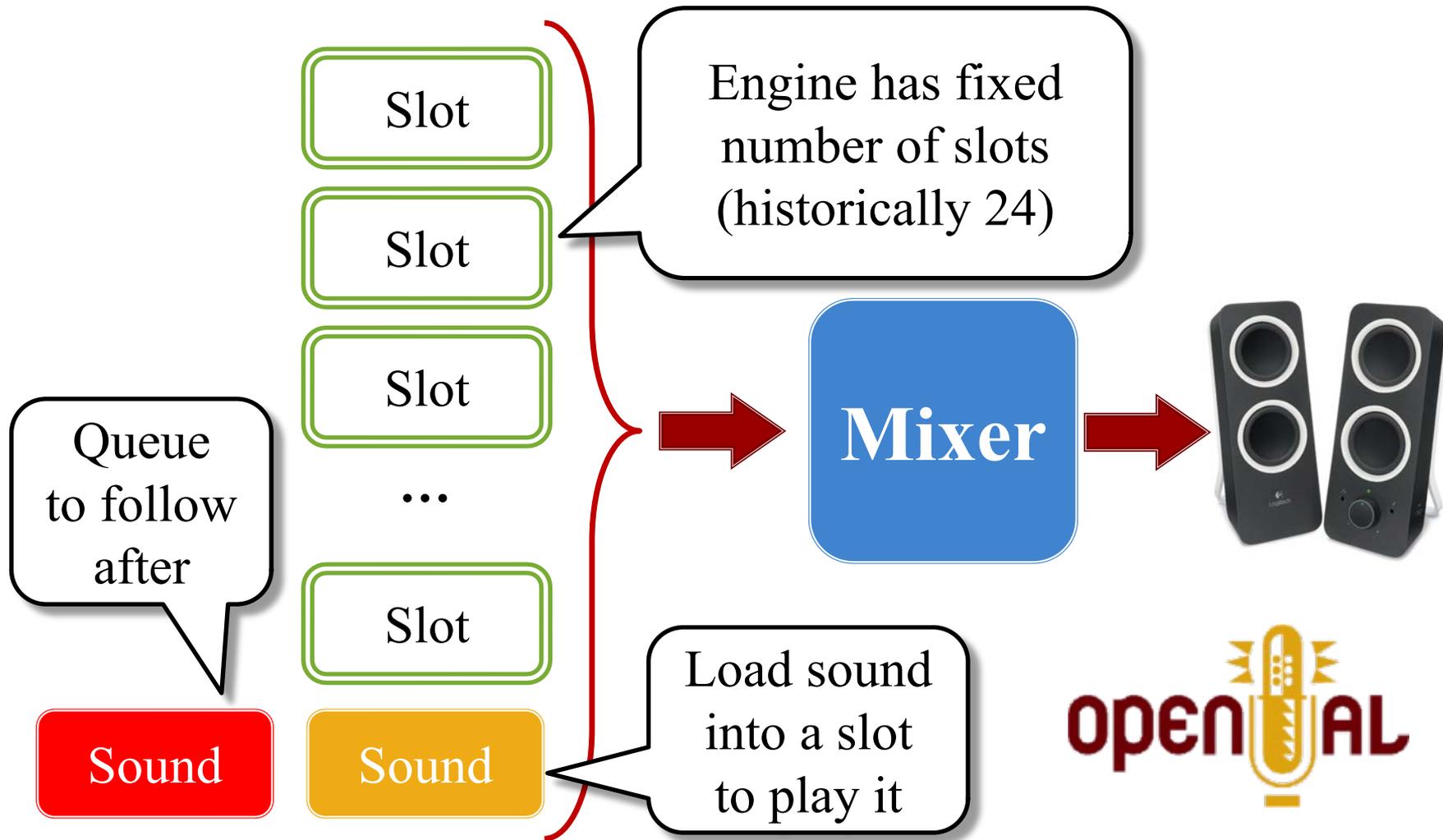
Classic Model: Playback Slots



Classic Model: Playback Slots



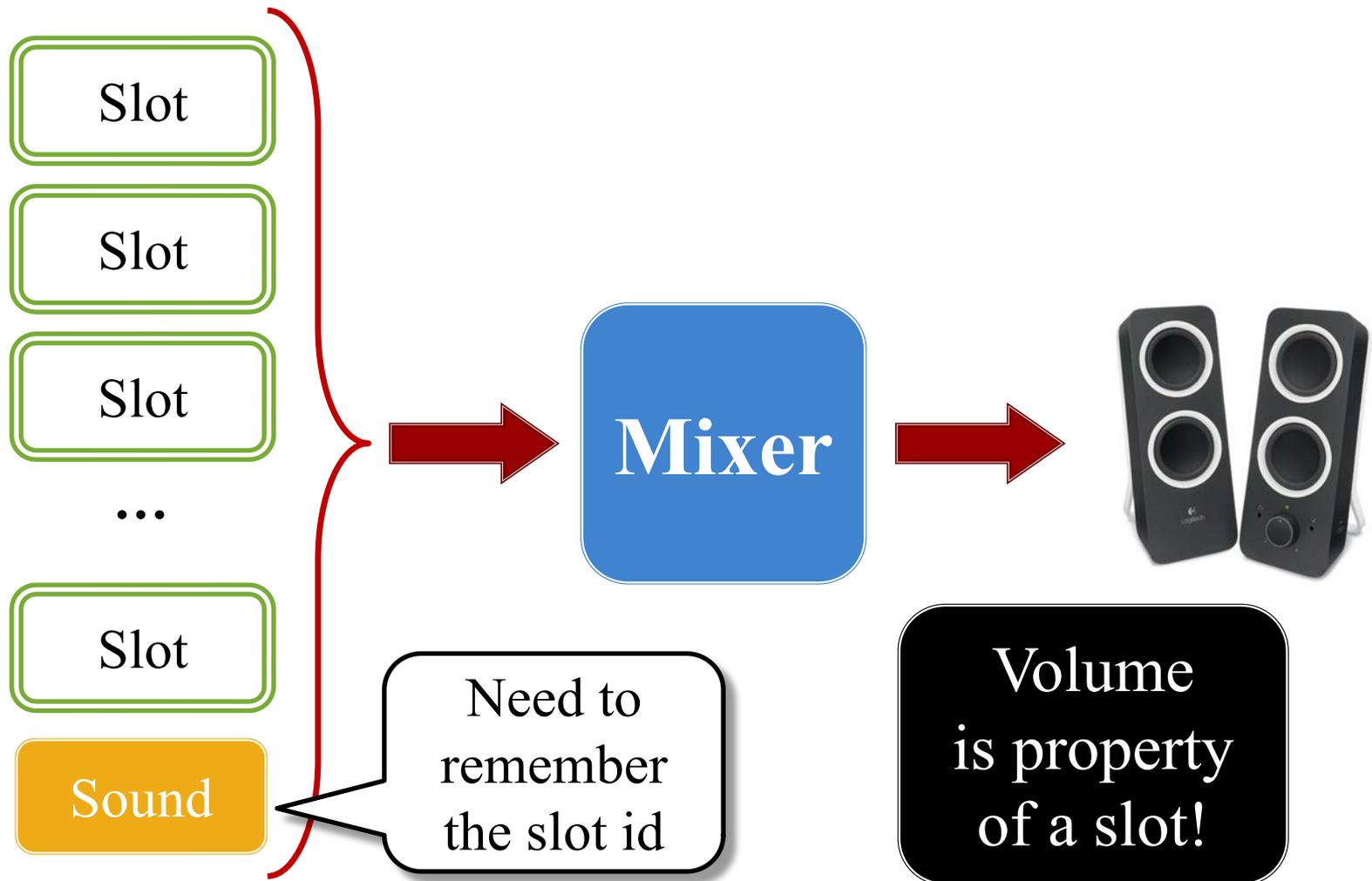
Classic Model: Playback Slots



Playing a Sound with Slots

- **Request** a playback slot for your asset
 - If none is available, sound fails to play
 - Otherwise, it gives you an id for the slot
- **Load** asset into the slot (but might stream)
- **Play** the playback slot
 - Playing is a property of the slot, not asset
 - Playback slot has other properties, like volume
- **Release** the slot when the sound is done
 - This is usually done automatically

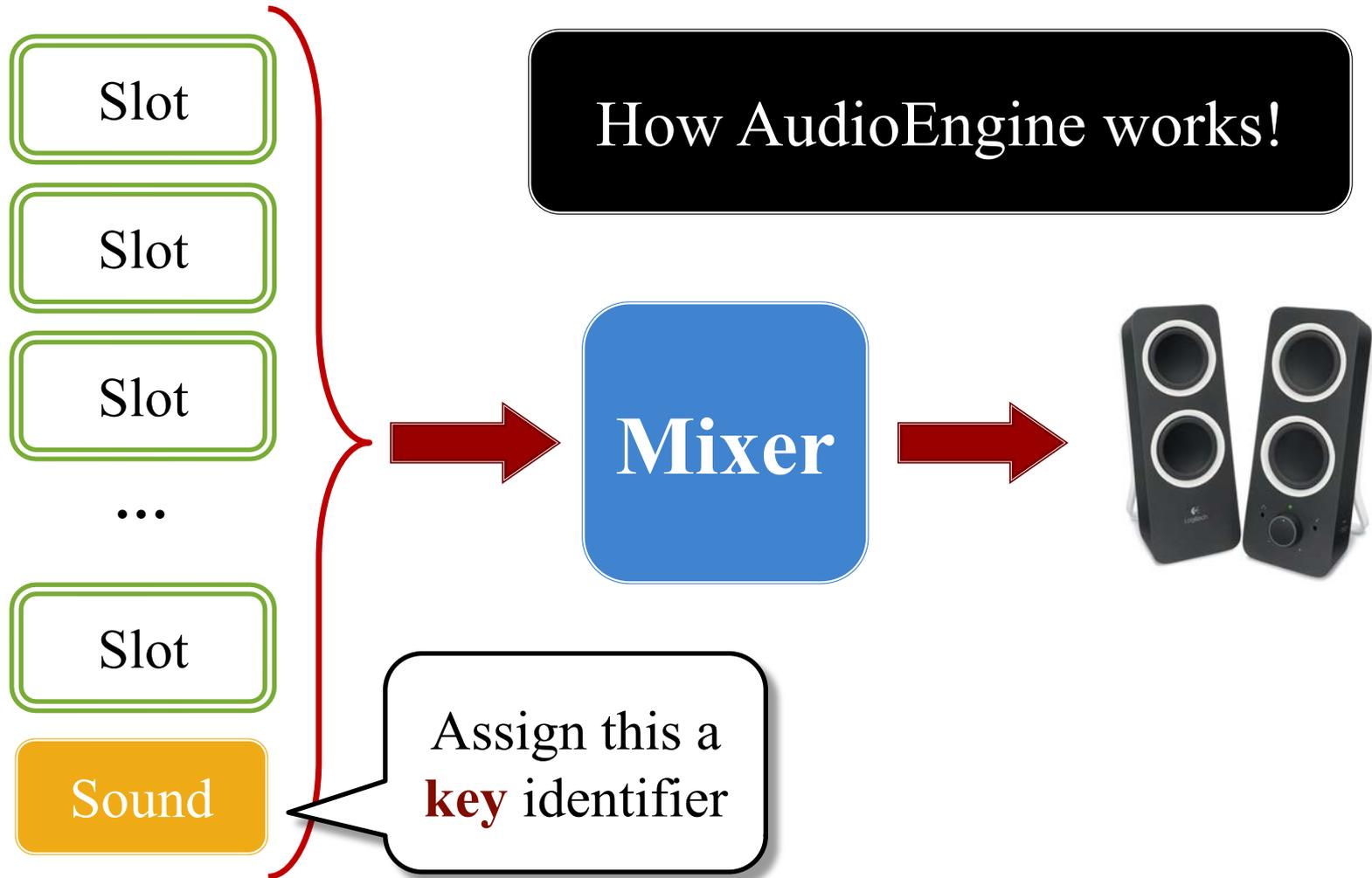
Application Design



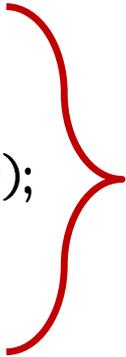
Why This is Undesirable

- Tightly couples architecture to sound engine
 - All controllers need to know this playback slot id
 - Playback must communicate id to all controllers
- Instances usually have a *semantic meaning*
 - **Example:** Torpedo #3, Ship/crate collision
 - Meaning is independent of the slot assigned
 - Would prefer to represent them by this meaning
- **Solution:** Refer to instances by *keys*

Application Design



The AudioEngine API

- ```
/**
 * Plays the given sound, and associates it with the specified key.
 *
 * @param key the reference key for the sound effect
 * @param sound the sound effect file to play
 * @param loop whether to loop indefinitely
 * @param volume the sound volume
 */
void play(const string key, const std::shared_ptr<Sound>& sound);
```
  - ```
void stop(const string key);
```
 - ```
void setVolume(const string key, float volume);
```
  - ```
void getState(const string key);
```
- 
- Refer to
instance
logically

Stopping Sounds

- Would like to know when a sound is finished
 - To free up the slot (if not automatic)
 - To stop any associated animation
 - To start a follow-up sound
- Two main approaches
 - **Polling**: Call an `isPlaying()` method/function
 - **Callback**: Pass a listener to the engine
- **AudioEngine** allows both approaches

Gapless Playback

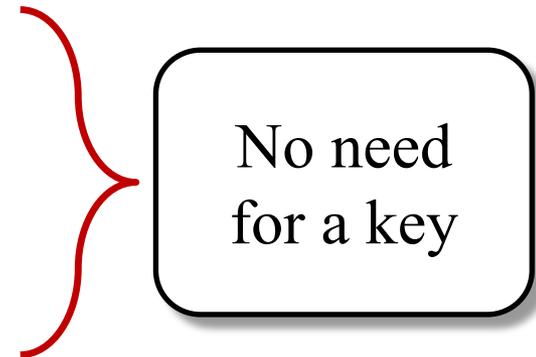
- Gapless playback requires a **queue**
 - Queue immediately plays next sound on completion
 - Ideally with some **crossfade** to prevent pops
- Supported by class **AudioQueue**
 - Built on top of AudioEngine; use **allocQueue()** method
 - Permanently takes over a slot for the queue
 - Can have multiple queues – as many as there are slots
 - But no simultaneity guarantee between queues
- AudioQueue is *kind of* similar to AudioEngine
 - But no need for keys, as there is only one slot

The AudioQueue API

- ```
/**
 * Adds the given sound to the queue, to play when possible.
 *
 * @param sound the sound effect file to play
 * @param loop whether to loop indefinitely
 * @param volume the sound volume
 * @param fade number of seconds to fade in
 */
```

```
void enqueue(const std::shared_ptr<Sound>& sound);
```

- ```
void advance(unsigned int steps);
```
- ```
void setVolume(float volume);
```
- ```
void getState();
```

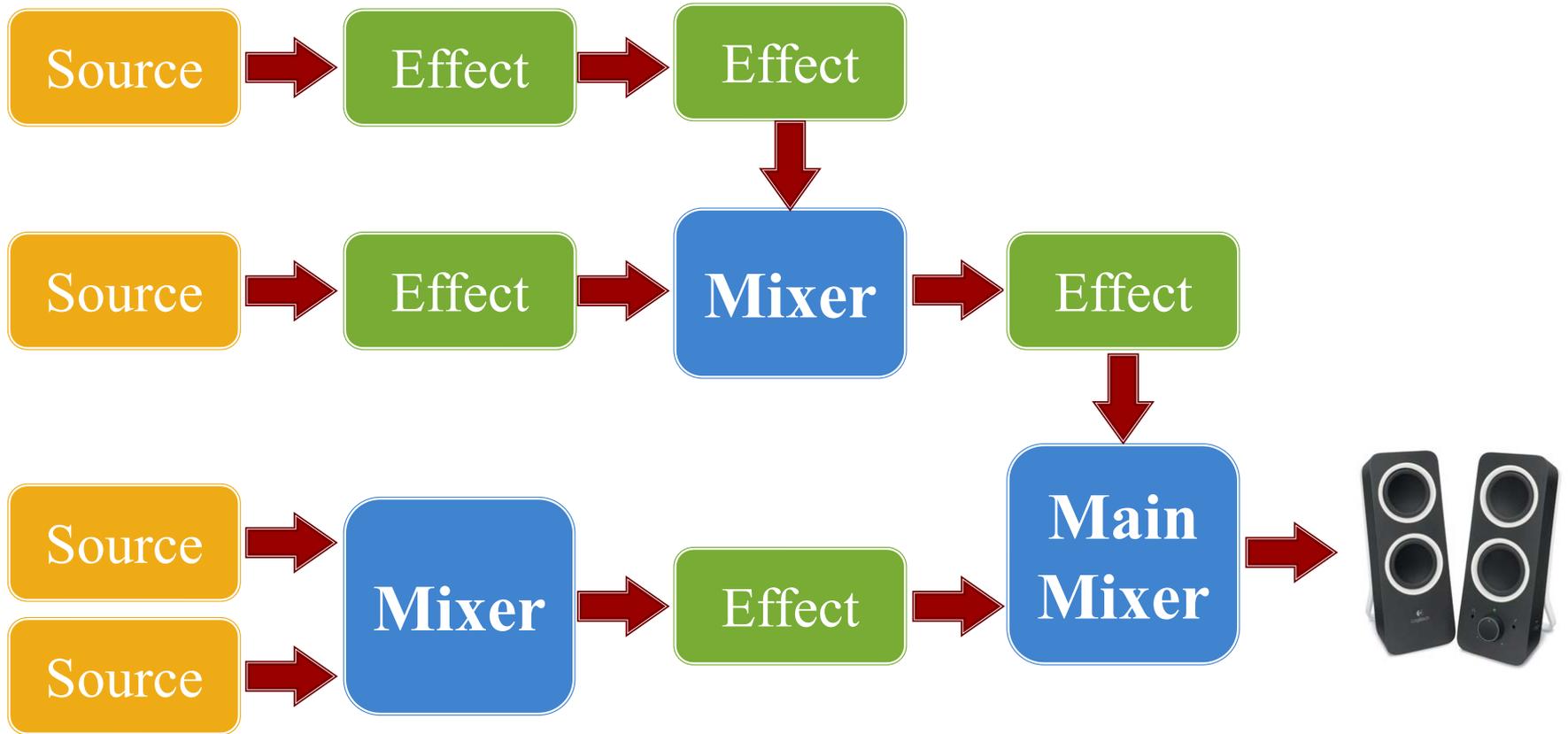


No need
for a key

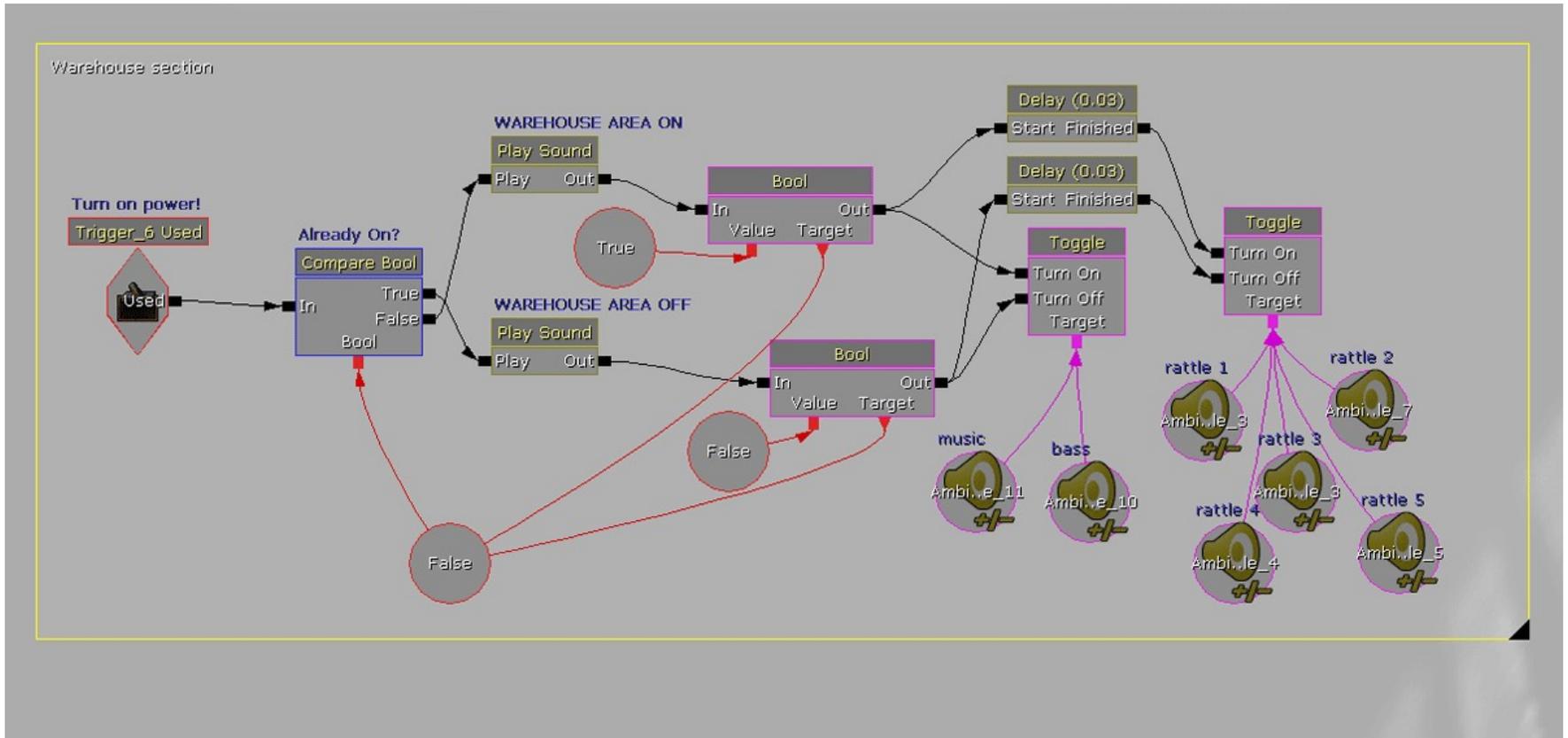
Problem with the Slots Model

- All controls are embedded in the slot
 - **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

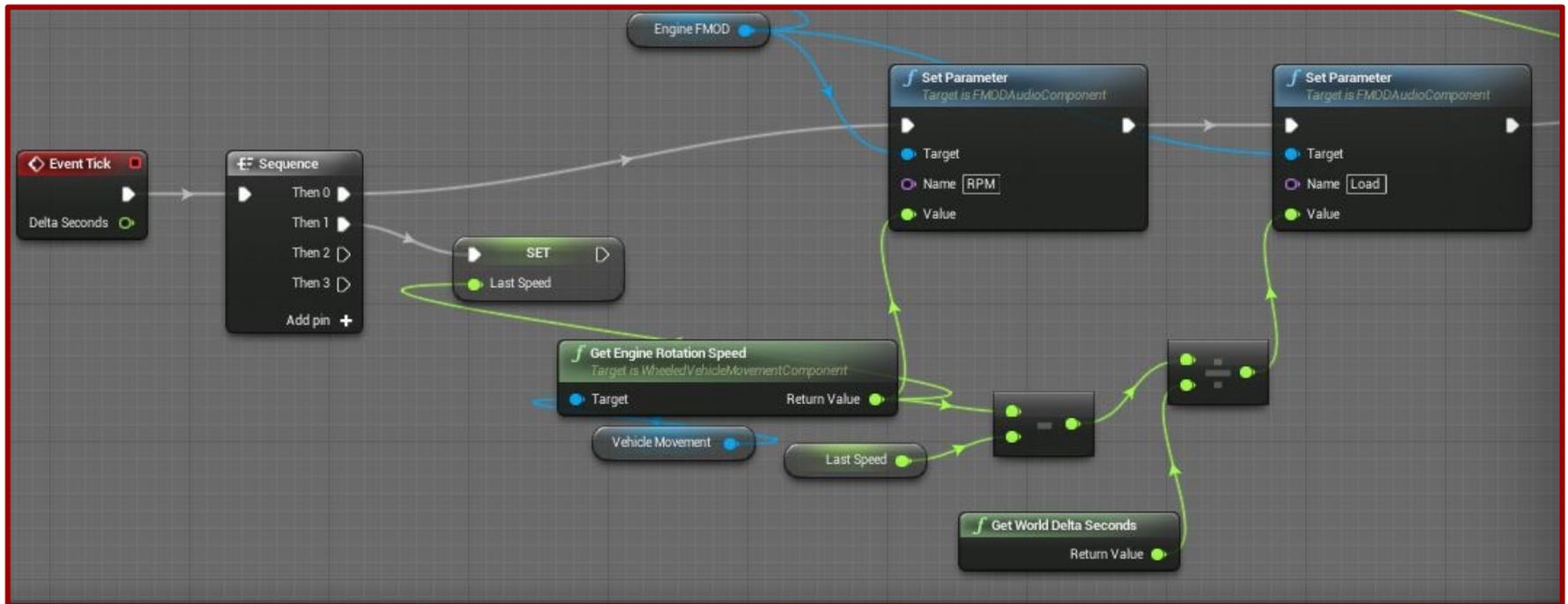
DSP Processing: The Mixer DAG



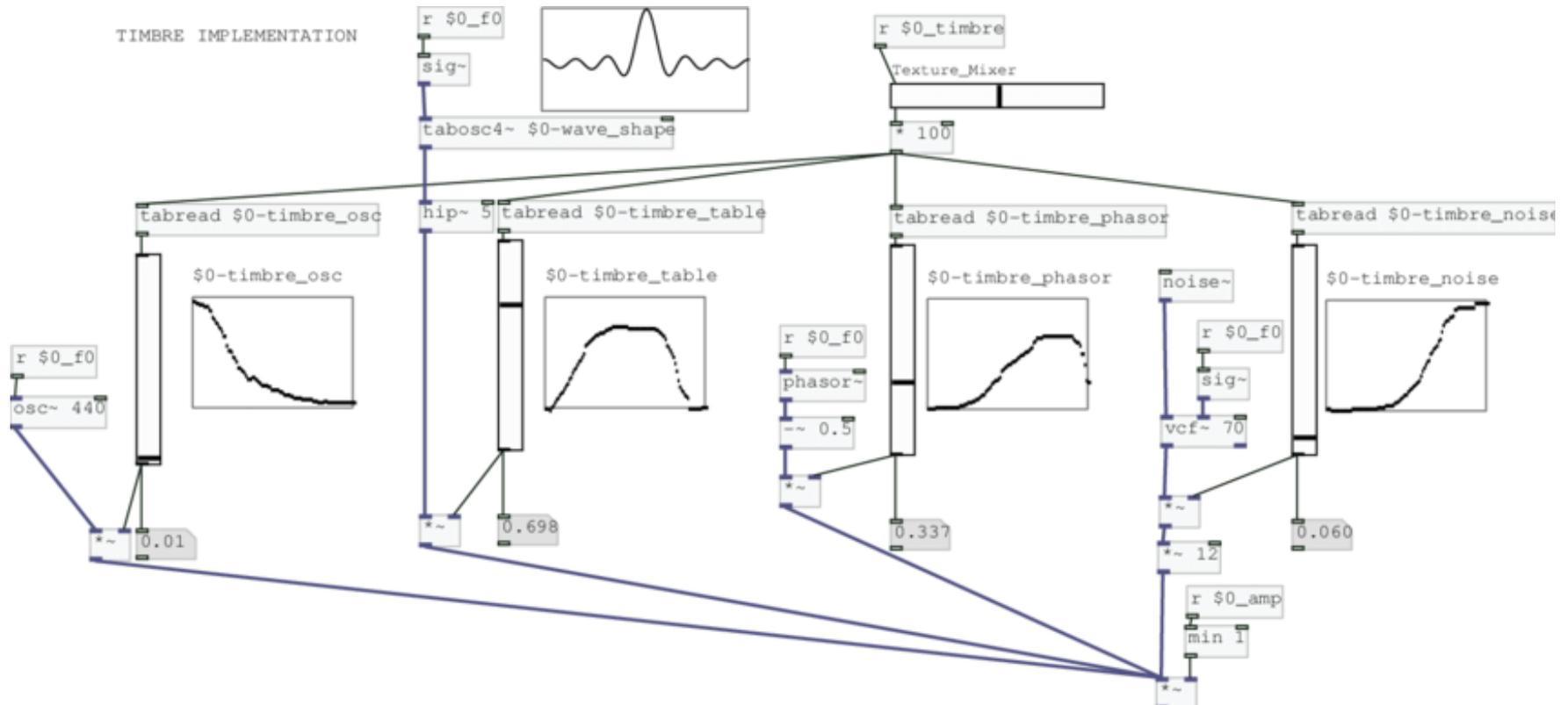
Example: UDK Kismet



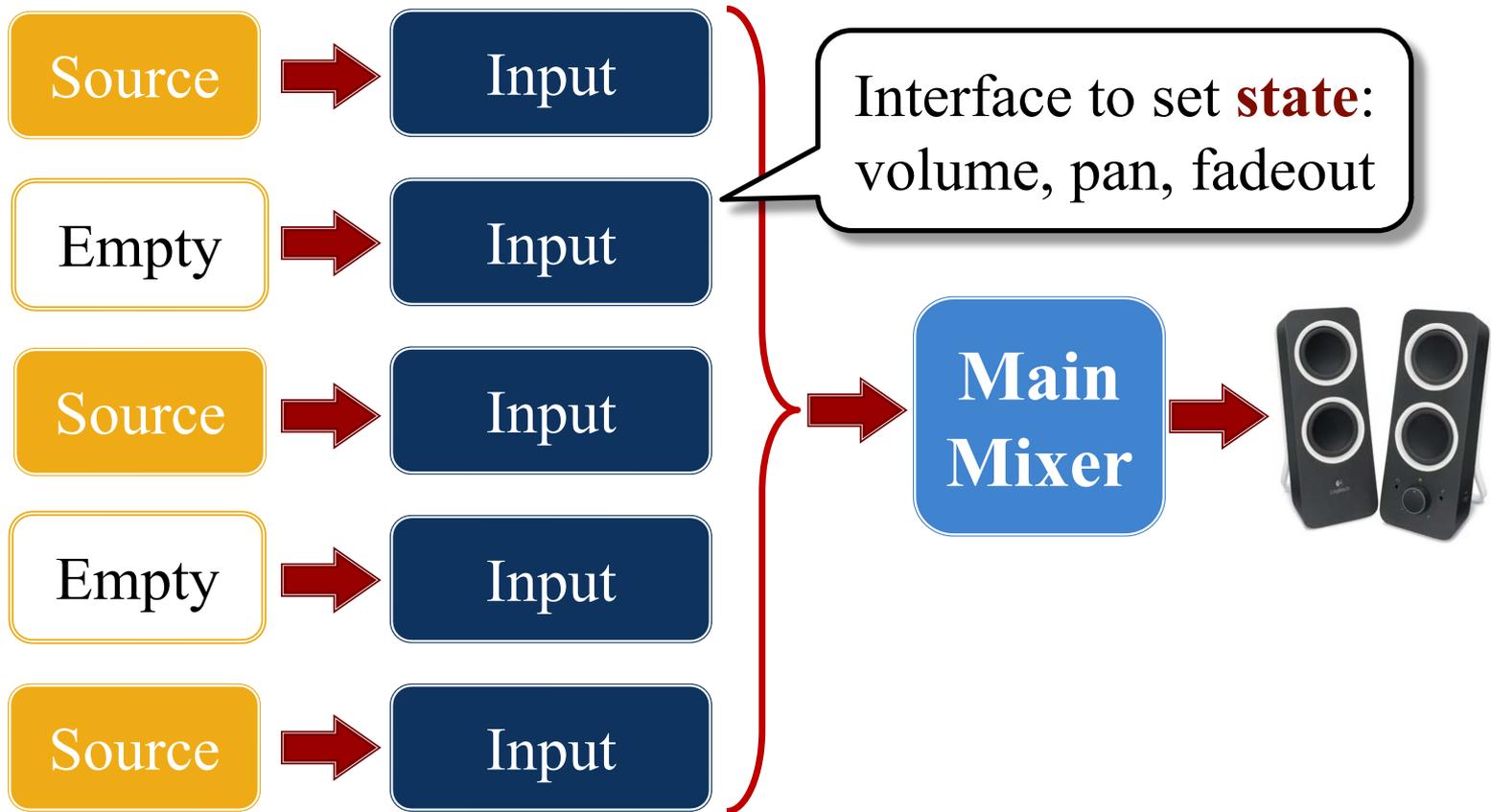
Example: FMOD



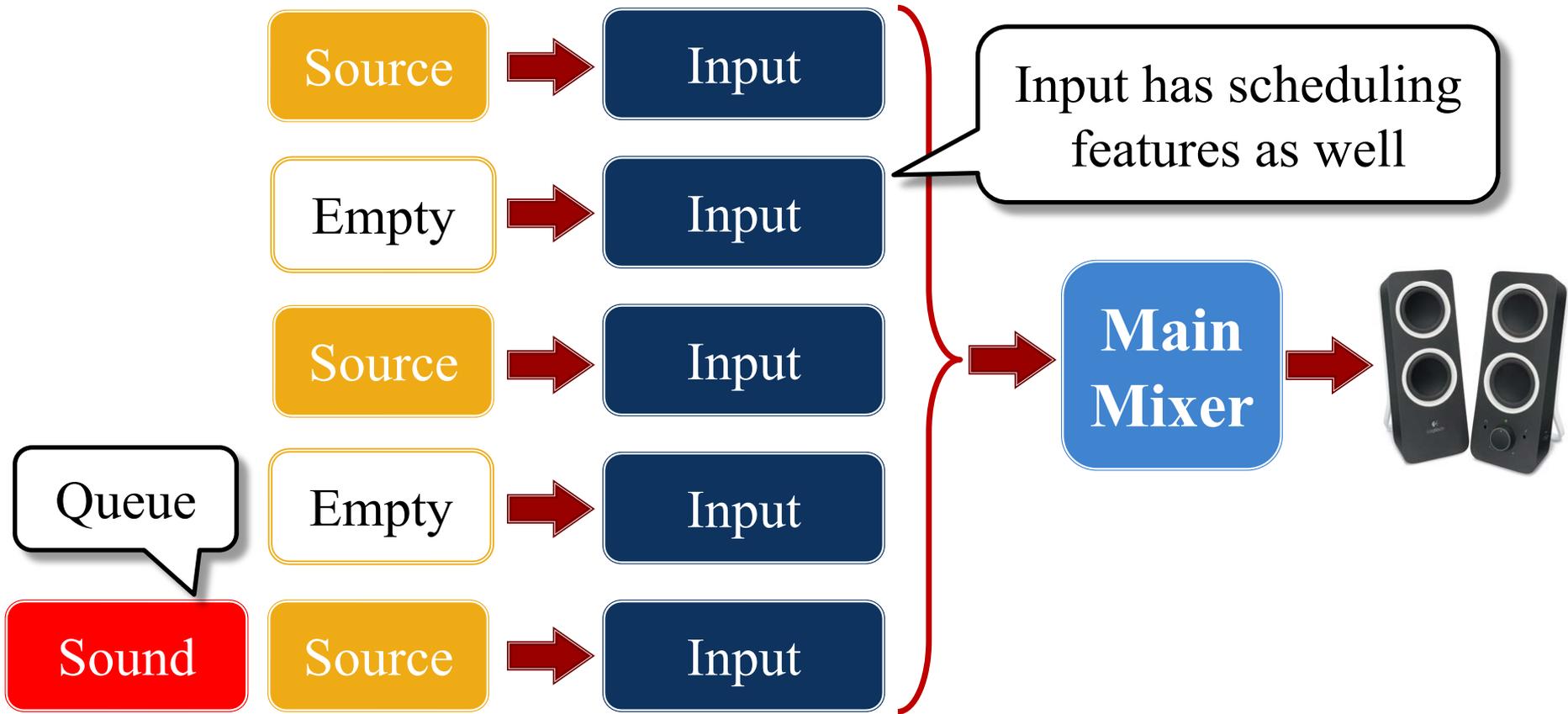
Example: Pure Data



The Slot Model is a Special Case

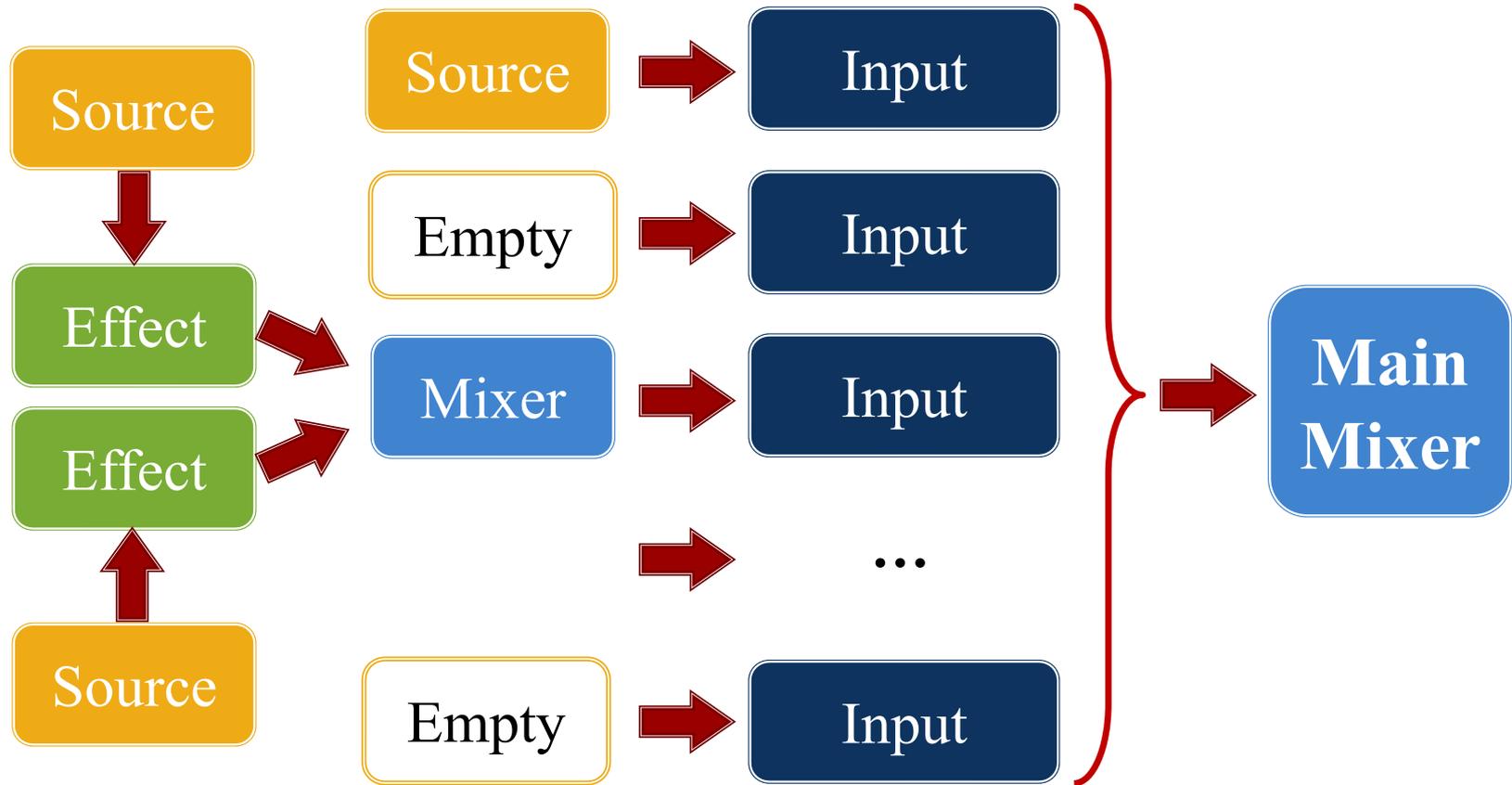


The Slot Model is a Special Case



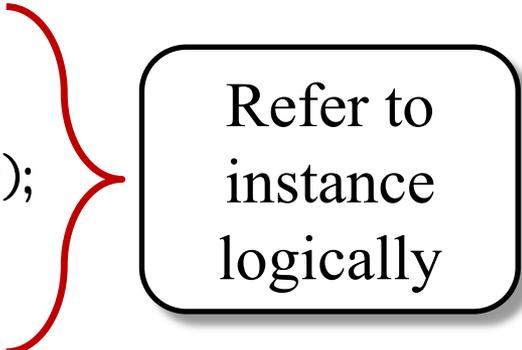
All happens behind scenes of AudioEngine interface.

The Slot Model is a Special Case



Theoretically input should accept any **audio subgraph**

The AudioEngine Revisited

- ```
/**
 * Plays the given sound, and associates it with the specified key.
 *
 * @param key the reference key for the sound effect
 * @param node the audio node to play
 * @param loop whether to loop indefinitely
 * @param volume the sound volume
 */
void play(const string key, const std::shared_ptr<AudioNode>& node);
```
  - ```
void stop(const string key);
```
 - ```
void setVolume(const string key, float volume);
```
  - ```
void getState(const string key);
```
- 
- Refer to instance logically

The AudioEngine Revisited

- ```
/**
 * Plays the given sound, and associates it with the specified key.
 *
 * @param key the reference key for the sound.
 * @param node the audio node.
 * @param volume the volume of the sound.
 * @param state the state of the sound.
 */
void play(const string key, const audio_node & node);

void setVolume(const string key, float volume);

void getState(const string key);
```

Also supported  
by AudioQueue

Refer to  
instance  
logically

# Using AudioNode in AudioEngine

---

- Normal playback is **built on top of it**
  - Uses `sound->createNode()` to get your node
  - So just as fully featured as normal playback
- But the node must implement `completed()`
  - This is optional method for `AudioNode` subclasses
  - The default implementation always returns `false`
  - But that means the sound never finished playing
  - So the scheduler cannot free slot for new sound

# AudioNode Classes in CUGL

---

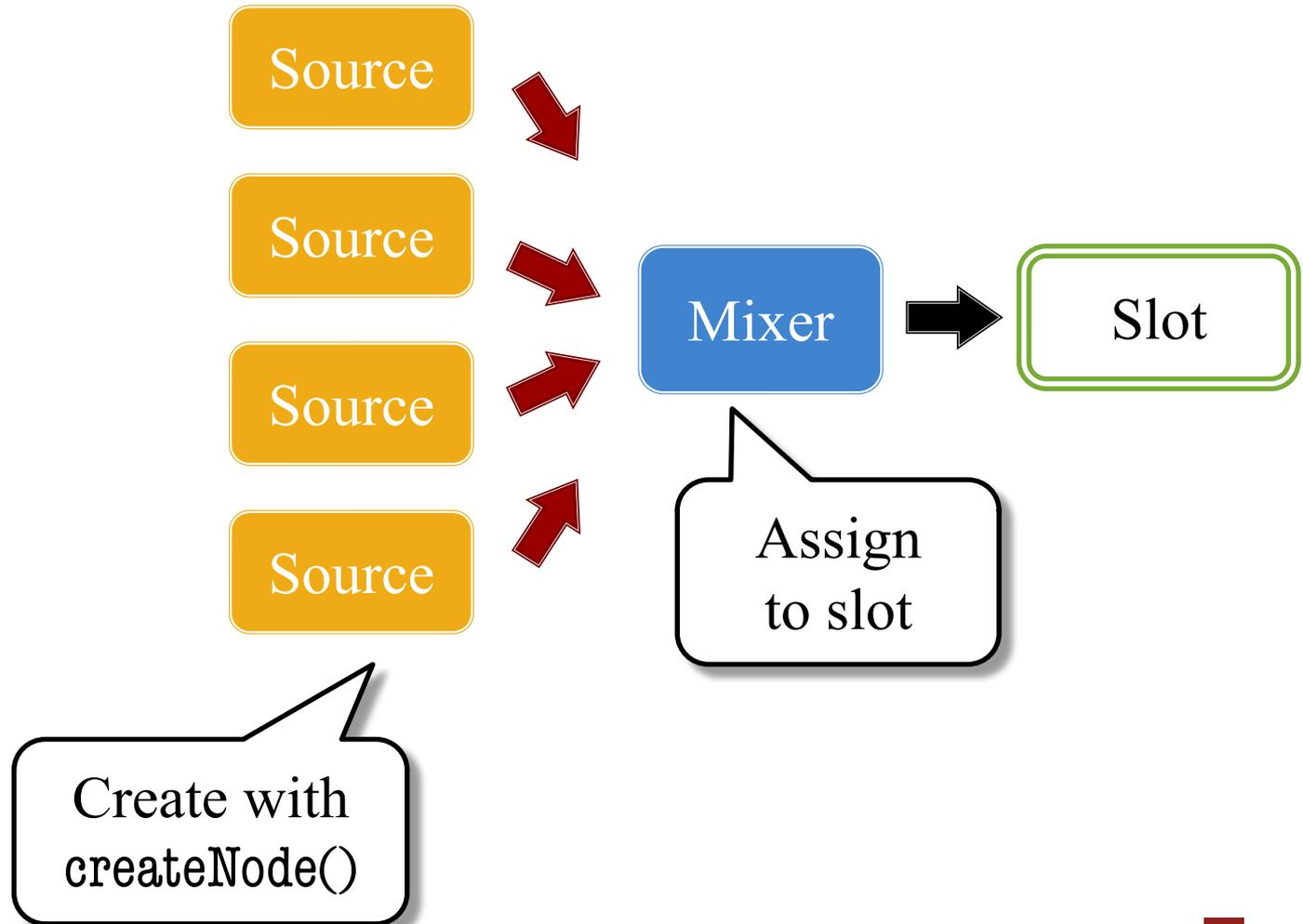
- **AudioPlayer**
  - Single playable instance for a sound asset
- **AudioFader**
  - Fade-in, fade-out and cross-fade effects
- **AudioMixer**
  - Group several **simultaneous** nodes together
- **AudioScheduler**
  - Used to queue up sounds in a **sequence**

# AudioNode Classes in CUGL

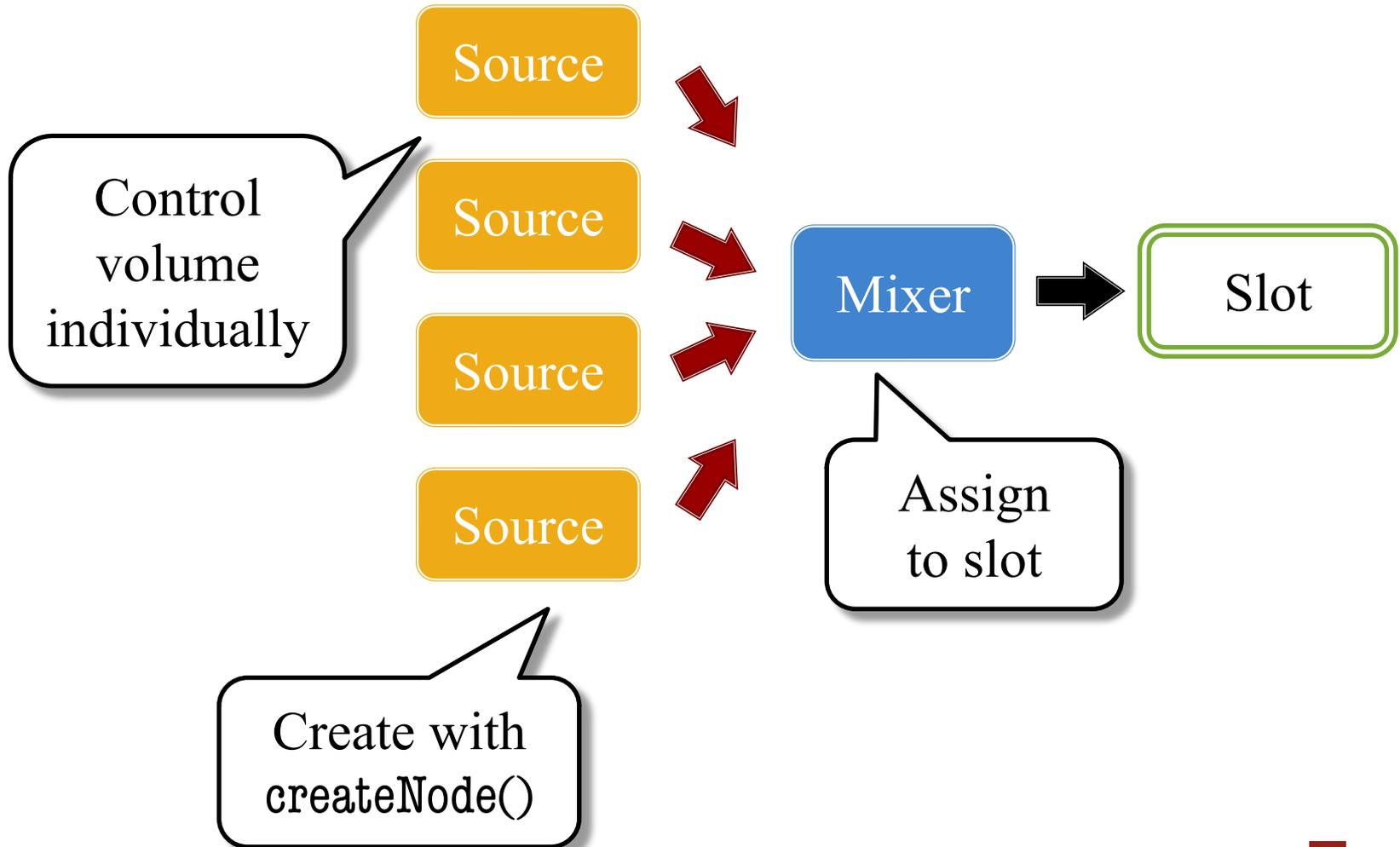
---

- **AudioPanner**
  - Simple stereo channel panning
- **AudioSpinner**
  - Like panner but works on 7.1 sound fields
- **AudioResampler**
  - Converts audio to different sample rate
- **AudioReverb**
  - Creates echo effects

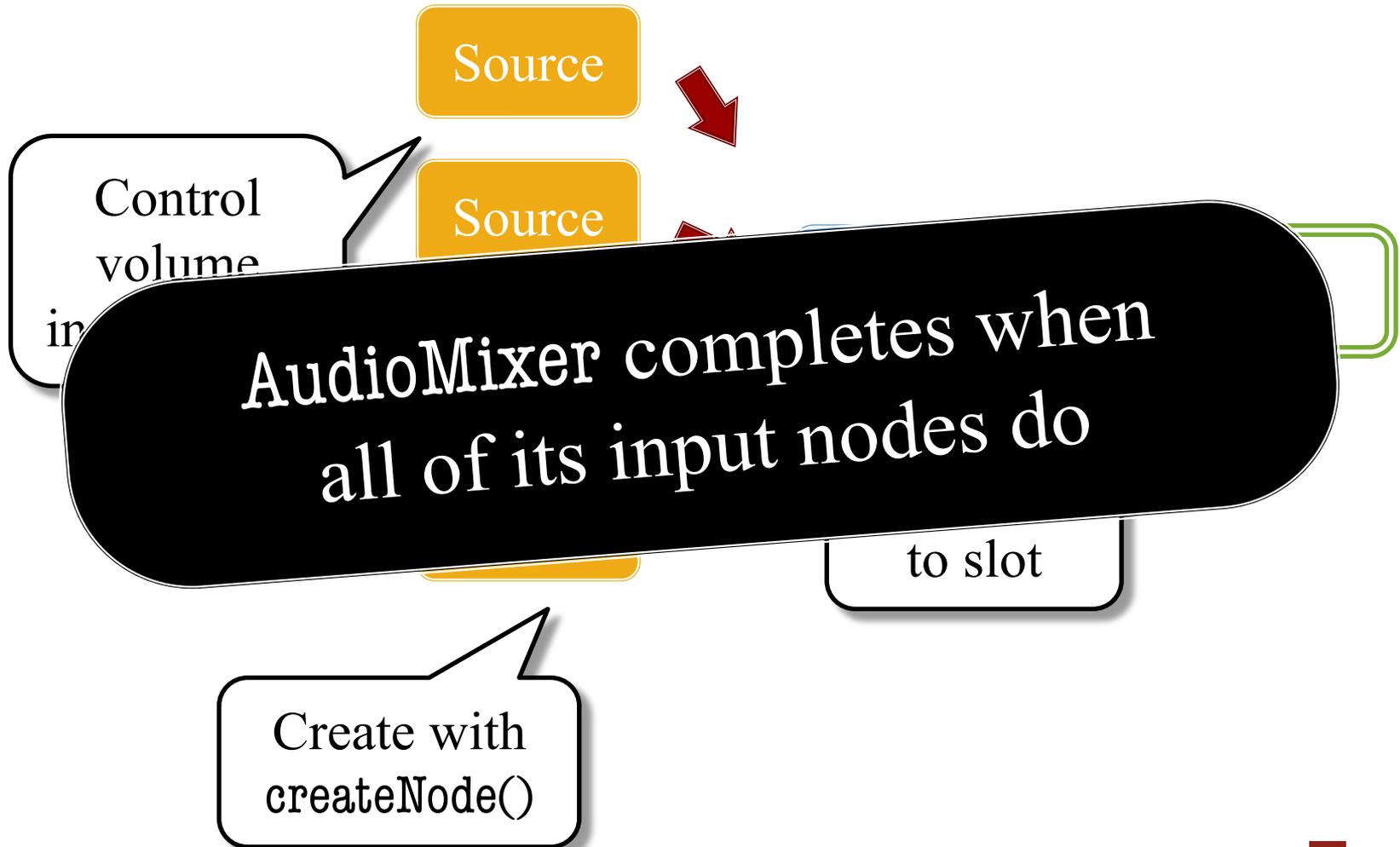
# Application: Vertical Layering



# Application: Vertical Layering



# Application: Vertical Layering



# Two Special AudioNodes

- Class **AudioOutput**

- Terminal node of the graph
- Represents output device
- Can be *named* or *default*
- Defines channels, sample rate

AudioOutput



AudioOutput



- Class **AudioInput**

- Initial node of the graph
- Represents input device
- Can be *named* or *default*
- May or may not match output

AudioOutput



# Two Special AudioNodes

- Class **AudioOutput**
  - Terminal node of the graph
  - Represents output device
  - Can be *named* or *default*
  - Defines channels, sample rate
- Class **AudioInput**
  - Initial node of the graph
  - Represents input device
  - Can be *named* or *default*
  - May or may not match output

AudioInput



AudioInput



AudioInput



# Two Special AudioNodes

- Class **AudioOutput**

- Terminal node of the graph
- Represents output device
- Can be *named*

AudioInput



These are all managed by the AudioDevices singleton

- Class **AudioInput**

- In the graph
- Represents input device
- Can be *named* or *default*
- May or may not match output

AudioInput



# 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
  - CUGL has some early support for all this