# 140

Sonic College 2018
Jakob Schmid

140

Jeppe Carlsen (design, programming)

Niels Fyrst, Andreas Peitersen (visual design)

Jakob Schmid (audio)

Developed as hobby project over 3 years

PSN

STEAM

Wii U

Humble Bundle

XBOX ONE

gog.com

# 140

**IGF award 2013**

Excellence in Audio

- Honorable mention: Technical Excellence

**Spilprisen 2014**

Sound of the Year

**Nordic Game Award 2014**

Artistic Achievement

# 140 Soundtrack

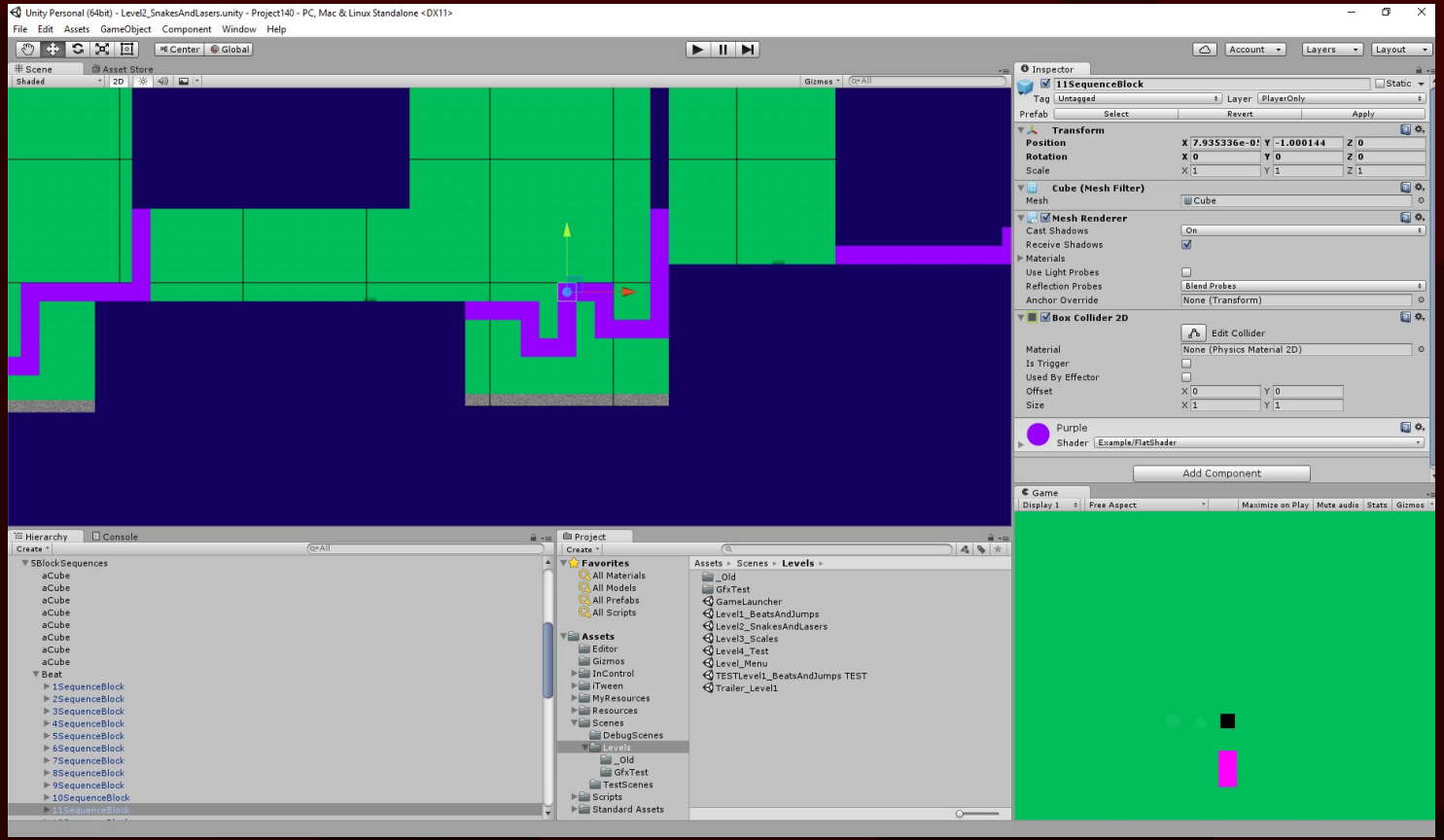
Vinyl

- iam8bit

Digital

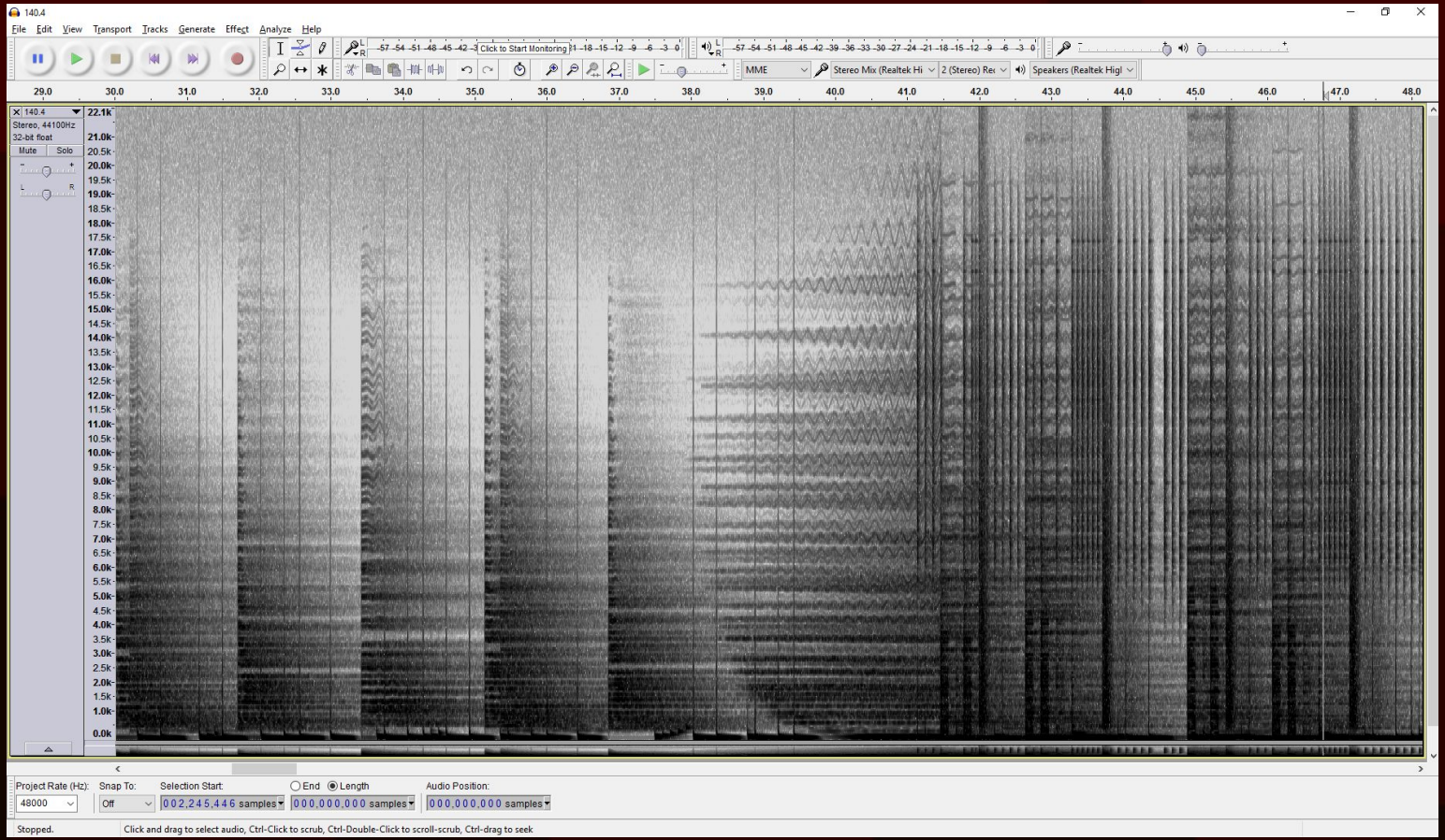
- Steam
- GOG.com
- Spotify
- iTunes
- Amazon

# Developed in Unity 3



# Ableton Live

# Audacity


140.4
File Edit View Transport Tracks Generate Effect Analyze Help
Click to Start Monitoring
MME
Stereo Mix (Realtek Hi
2 (Stereo) Rec
Speakers (Realtek High
140.4
Stereo, 44100Hz
32-bit float
Mute Solo
Project Rate (Hz):
48000
Snap To:
Off
Selection Start:
End Length
Audio Position:
002,245,446 samples
000,000,000 samples
000,000,000 samples
Stopped.
Click and drag to select audio, Ctrl-Click to scrub, Ctrl-Double-Click to scroll-scrub, Ctrl-drag to seek


Audio in 140

# Audio in 140

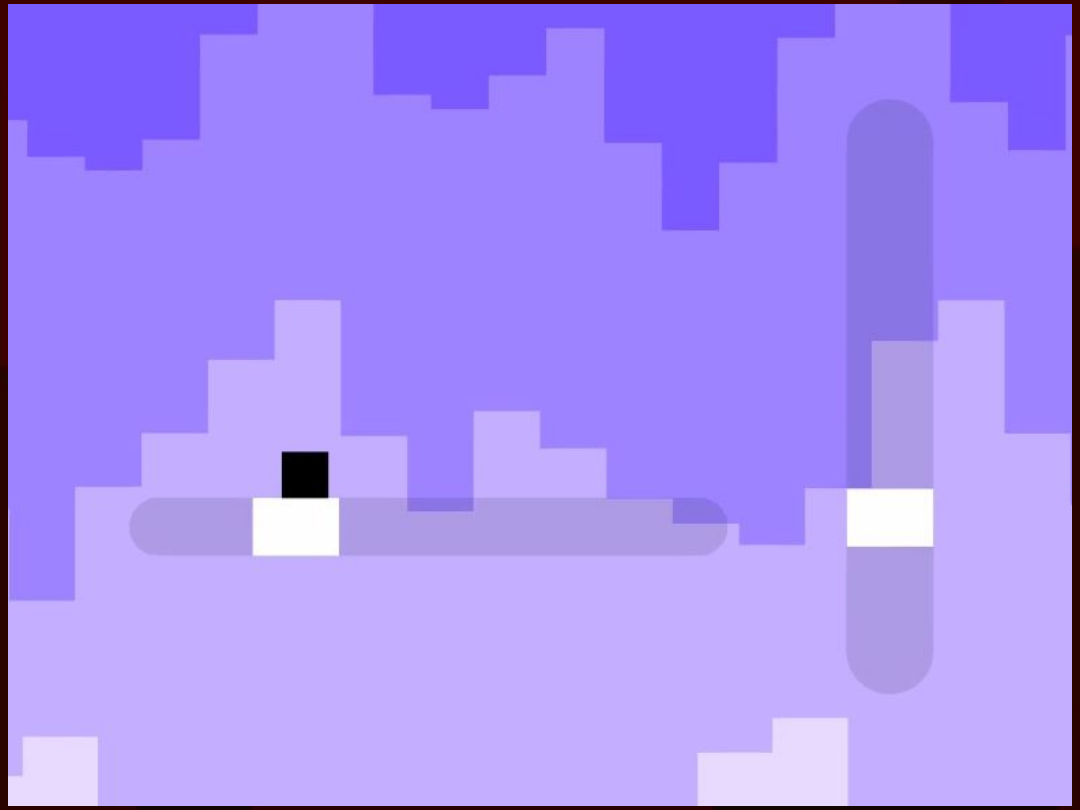
Overview

- Control game from music
- Interactive music
- Music timing in 140
- Fun audio tricks

# 140 demo


Control Game from Music

# Moving a Platform



# Moving a Platform

1. 
wait for 16th note #1

2. 
start moving

3. 
wait for 16th note #8

4. 
start moving

# Basic Approach

- Play music loop
- Use audio time from loop to control game elements (instead of game time)

Unity built-in audio:

```
AudioSource.time                    (seconds)
AudioSource.timeSamples             (samples)
```

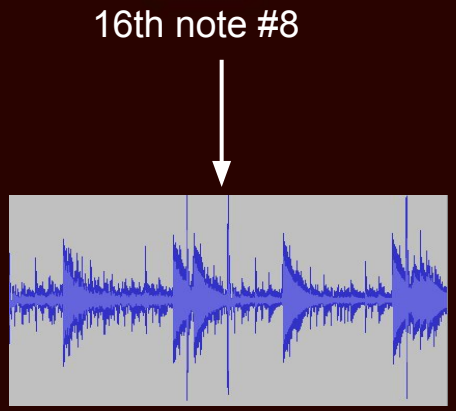
FMOD Unity Integration:

```
EventInstance.getTimelinePosition   (milliseconds)
```

# Music Events

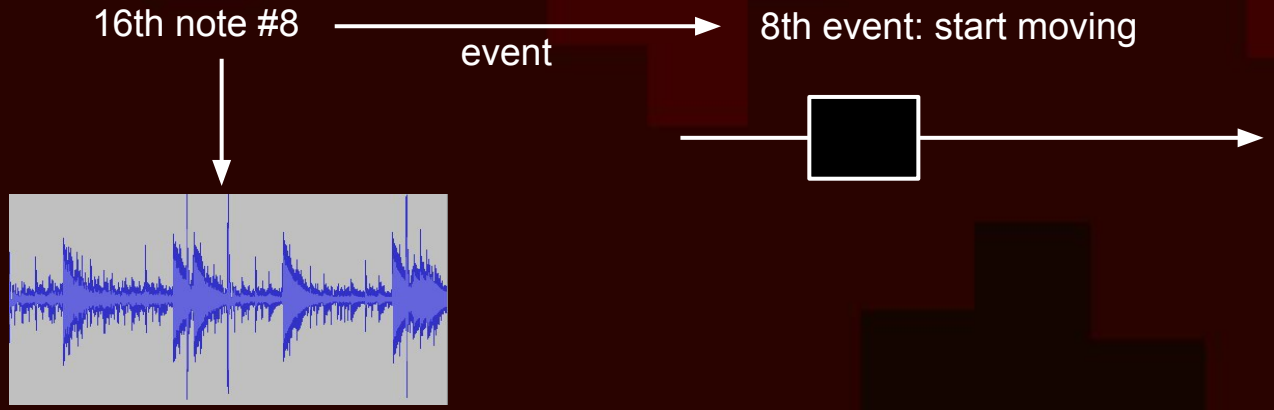
‘Waiting for 16th note #8’ means:

- Get audio time from playing loop
- When next musical beat reached, raise event
- On the 8th event, do something



# Music Events

Game elements listen for events and trigger animation on beats



# Music Events

- When next musical beat reached, raise event

  - And when is that, exactly?

# Useful Calculations

For a given tempo, how long is a note in seconds?

# Tempo

How long is a 140 BPM 16th note in seconds?

# Tempo

How long is a 140 BPM 16th note in seconds?

**140 beat/m**

# Tempo

How long is a 140 BPM 16th note in seconds?

```
140 beat/m * 4 note/beat = 560 note/m
```

# Tempo

How long is a 140 BPM 16th note in seconds?

```
140 beat/m * 4 note/beat = 560 note/m
                         = 560/60 note/s
```

# Tempo

How long is a 140 BPM 16th note in seconds?

```
140 beat/m * 4 note/beat = 560 note/m
                         = 560/60 note/s
```

This means that we have:

```
60/560 s/note
```

# Tempo

How long is a 140 BPM 16th note in seconds?

```
140 beat/m * 4 note/beat = 560 note/m
                         = 560/60 note/s
```

This means that we have:

```
                           60/560 s/note
                        ~= 0.10714 s/note
```

# Moving a Platform

1. 
wait for 16th note #0

2. 
start moving

3. 
wait for 16th note #8

4. 
start moving

# Moving a Platform

1. 

2. 

3. 

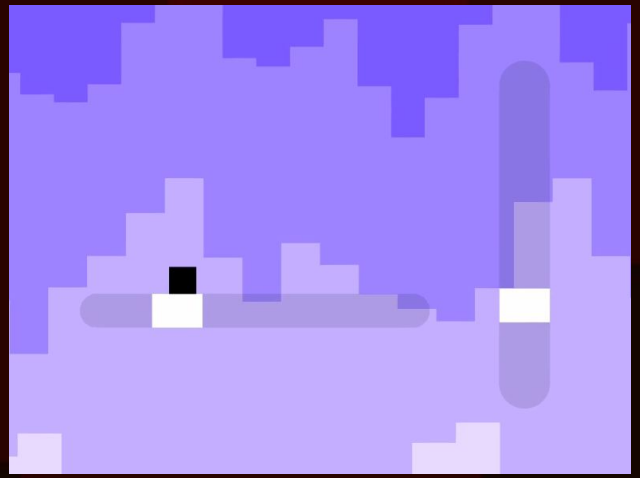
4. 

wait for 16th note #0,
time = 0 s

wait for 16th note #8,
time = 8 * 0.10714 s
= 0.857 s

# Summary

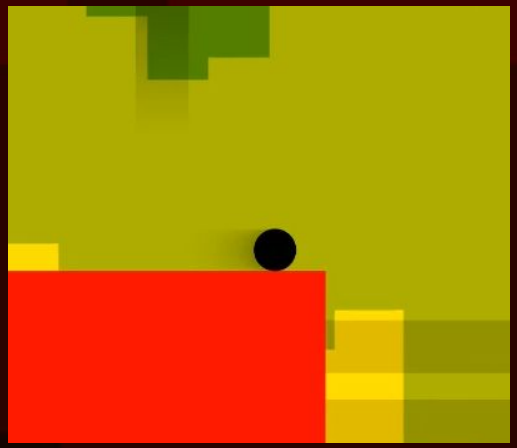
- Game elements wait for music events to control animation
- Music system observes `AudioSource.time` (Unity built-in)
- ... or `EventInstance.getTimelinePosition` (FMOD Unity)
- Tempo can be converted to seconds
- Music events are triggered when a given time has been reached

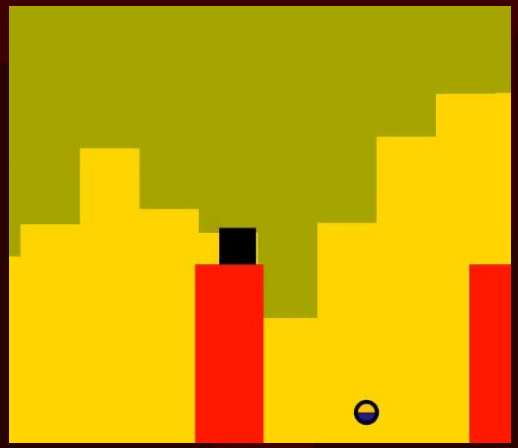


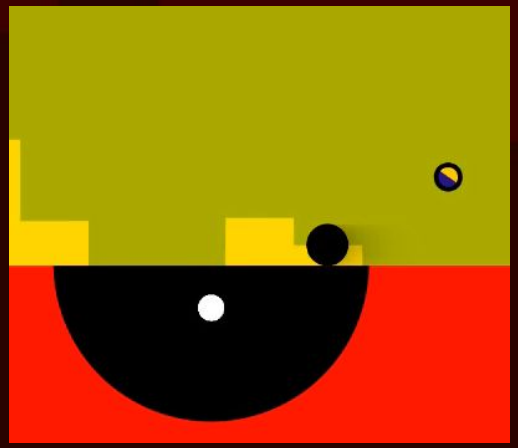

Interactive Music

# Interactive Music Mixing

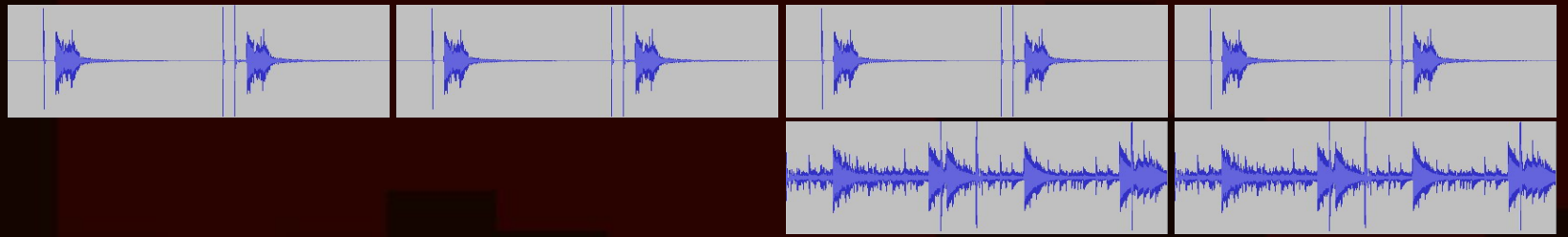
We wanted to mix music interactively in Unity.







time



# The Music Timing Problem

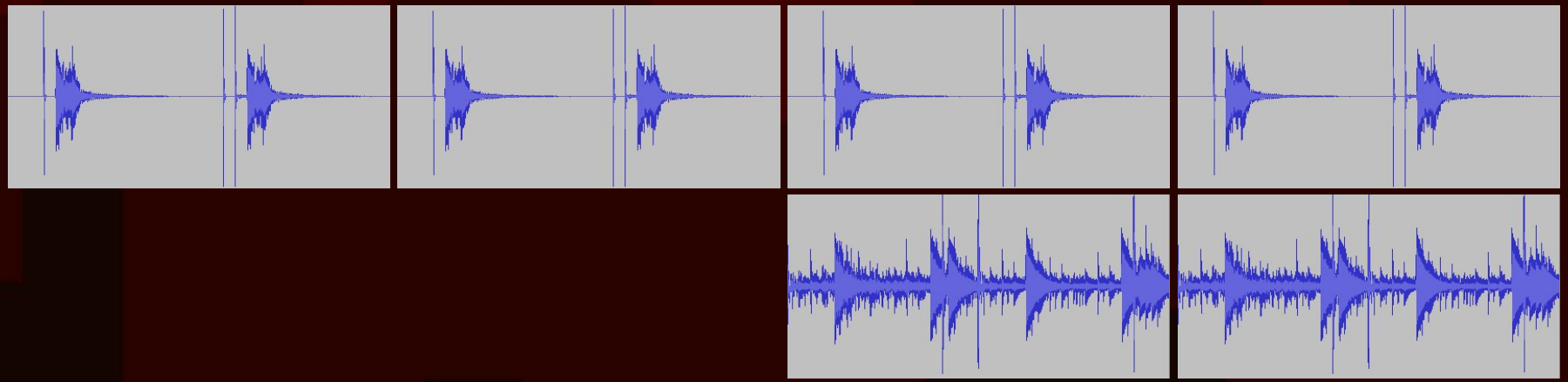
For beat-oriented music, loops should be synchronized with sample accuracy.

- That means a precision of 0.00002 s

# Loop Transition

Goal:

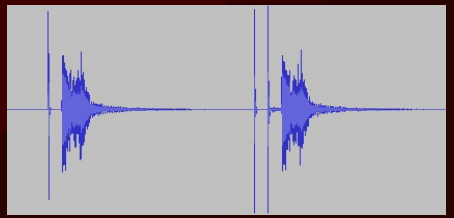
- Start loop A and let it run for a while.
- Then start loop B.
- B should be sample-accurately synchronized with loop A.



# Loop Transition Problem

Start loop A:

```
audioSourceA.Play()
```



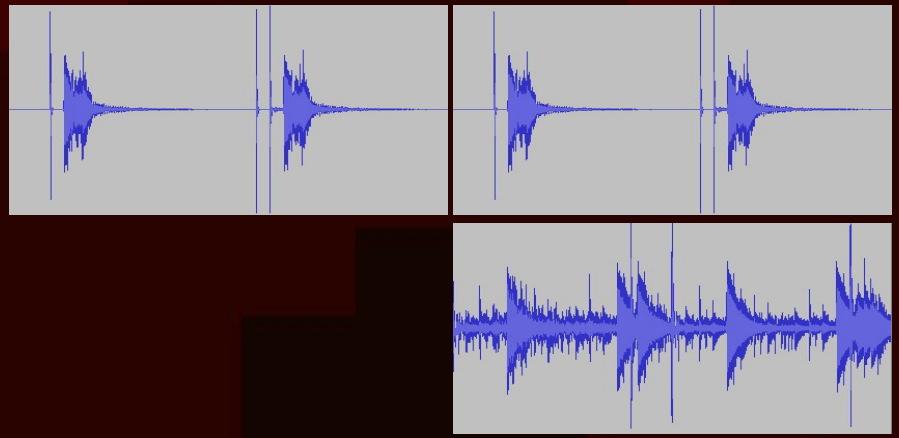
# Loop Transition Problem

Start loop A:

```
audioSourceA.Play()
```

Exactly when A loops, start loop B:

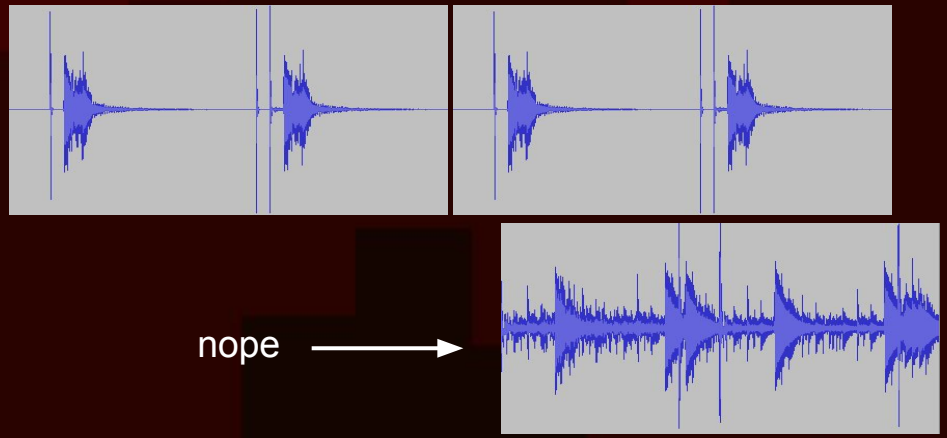
```
Wait for A to loop, then:
  audioSourceB.Play()
```



# Loop Transition Problem

It doesn’t work!

New sound is out of sync.

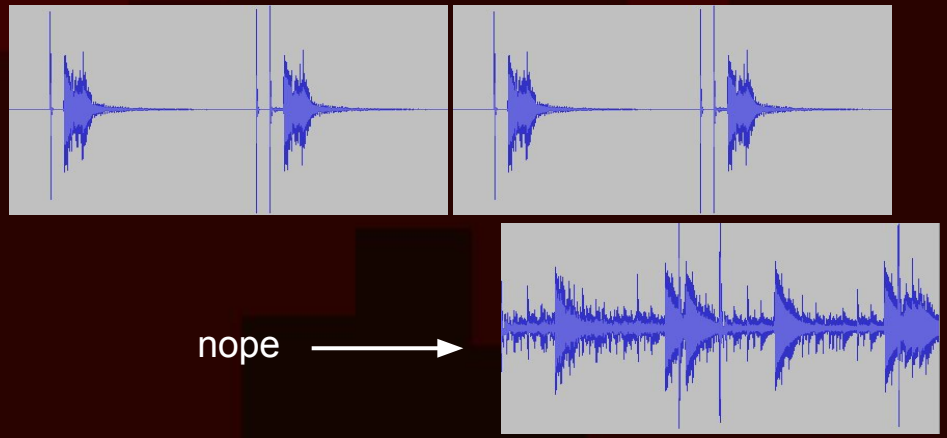


# Loop Transition Problem

It doesn’t work!

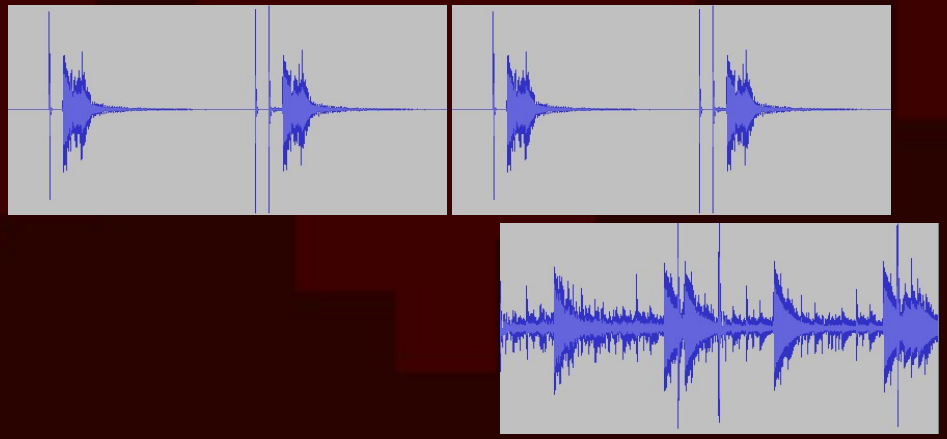
New sound is out of sync.

The problem exists in every sound engine.



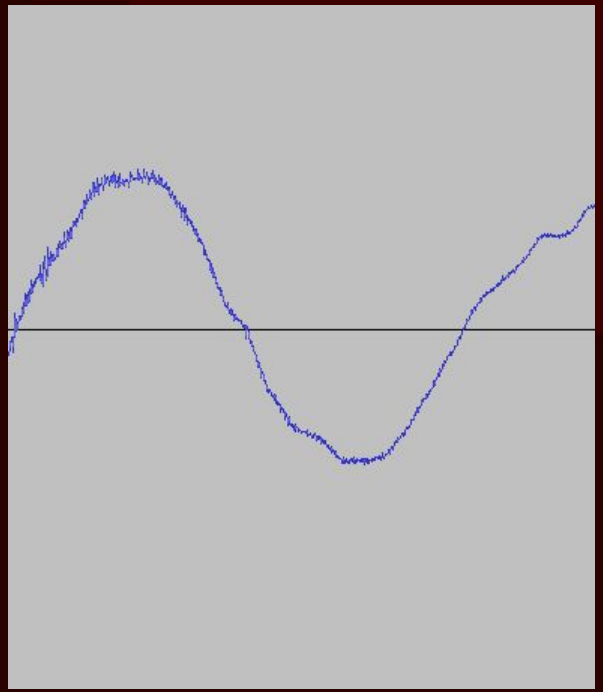
# Loop Transition Problem

Why?



# Audio Rendering

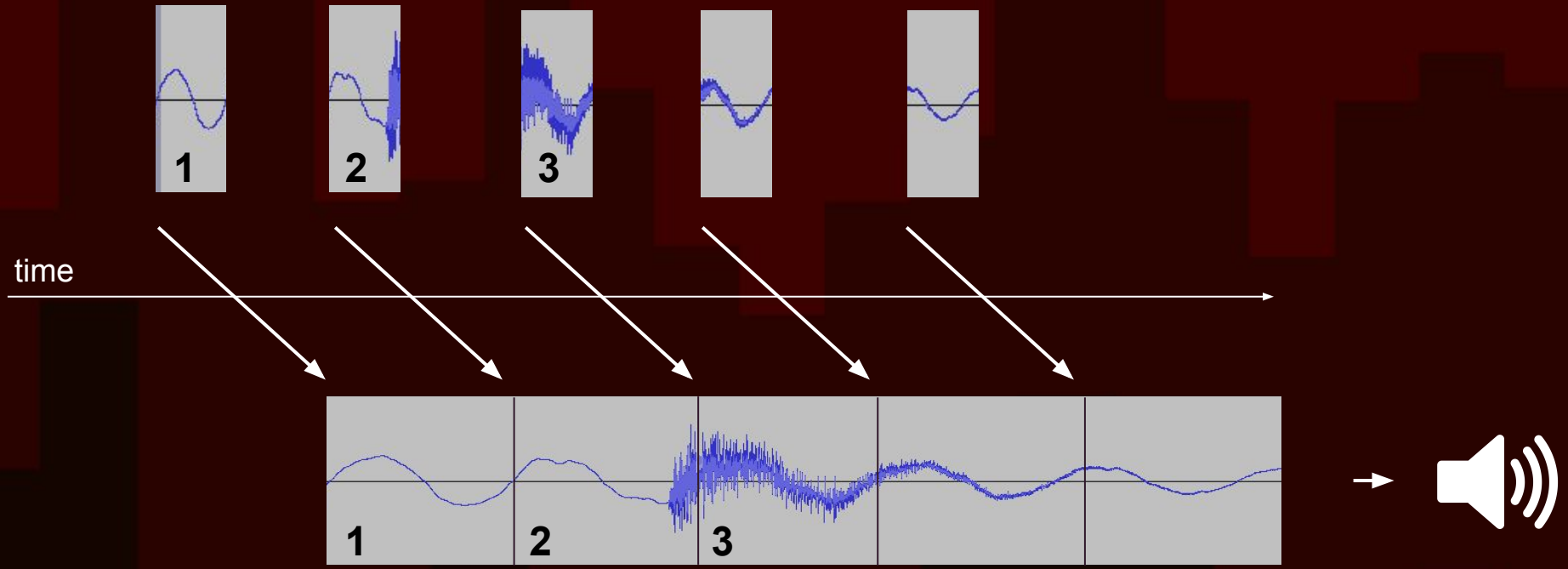
Audio is rendered a fixed number of samples at a time:



1024 sample buffer, 21 ms

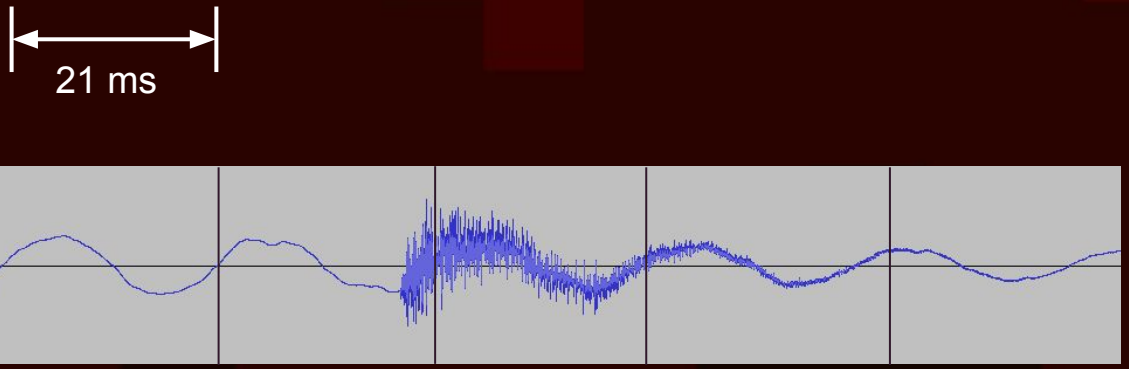
# Audio Rendering

The sound card plays a buffer while the next one is being rendered:



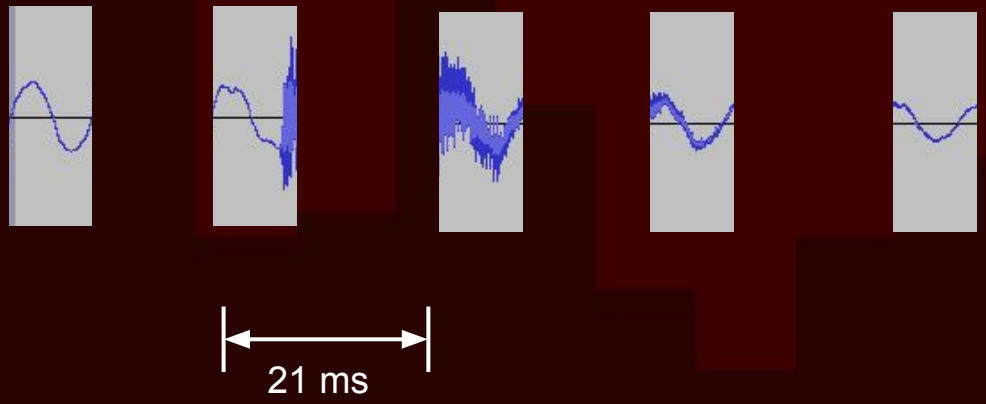
# Audio Rendering

If buffers are e.g. 1024 samples long, we need a new one every 21 ms.



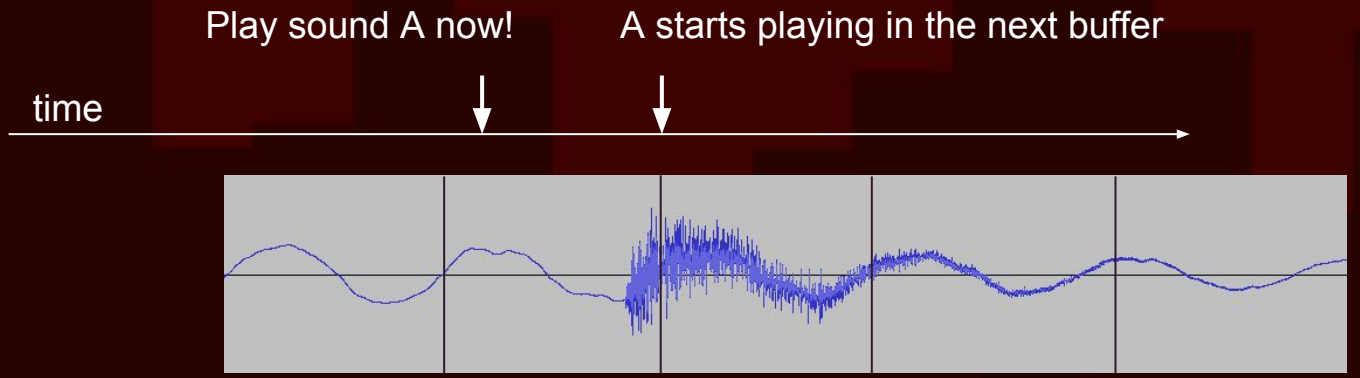
# Audio Rendering

In this case, a new buffer is rendered every 21 ms:



# Audio Rendering

- New sounds won’t start immediately, but earliest in the next audio buffer
- Their start time will also be quantized to buffer start times, e.g. 21 ms



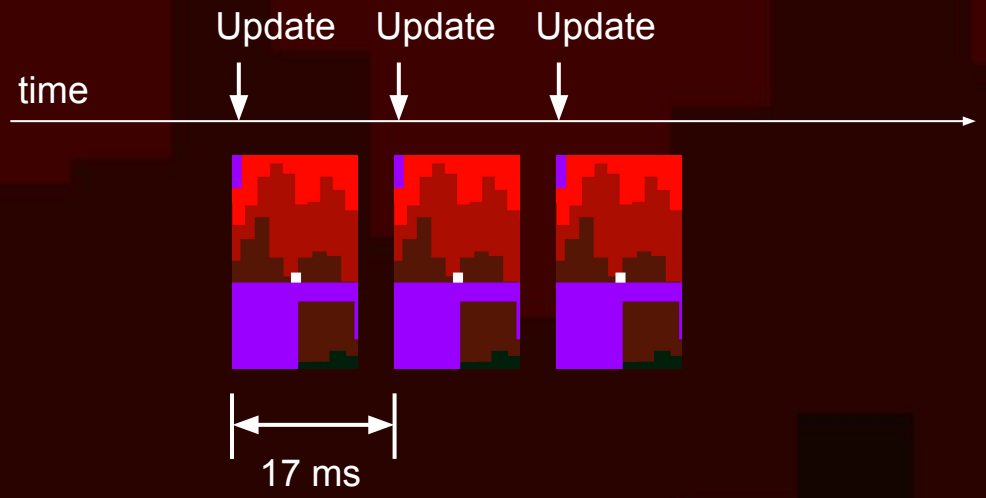
# Audio Rendering

In Unity, our audio code will probably be in an Update method

```
using UnityEngine;

public class MyAwesomeScript : MonoBehaviour
{
    public AudioSource mySound;

    // Use this for initialization
    void Start()
    {
    }

    // Update is called once per frame
    void Update()
    {
        mySound.Play();
    }
}
```

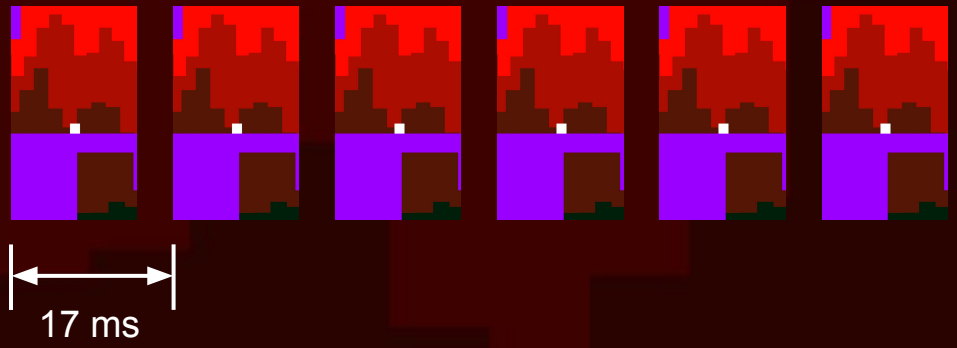
# Audio Rendering

Unity Update methods are called for every video frame
(e.g. 17 ms at 60 FPS)



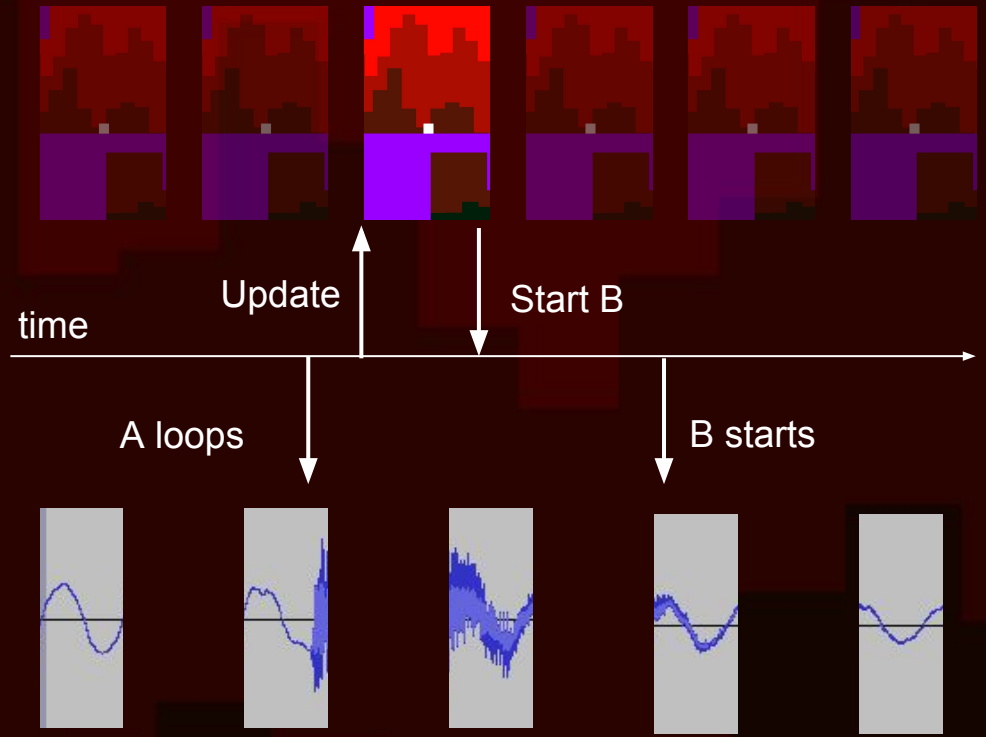
# Audio Rendering

Audio buffers and video frames are not synchronized:



# Audio Rendering

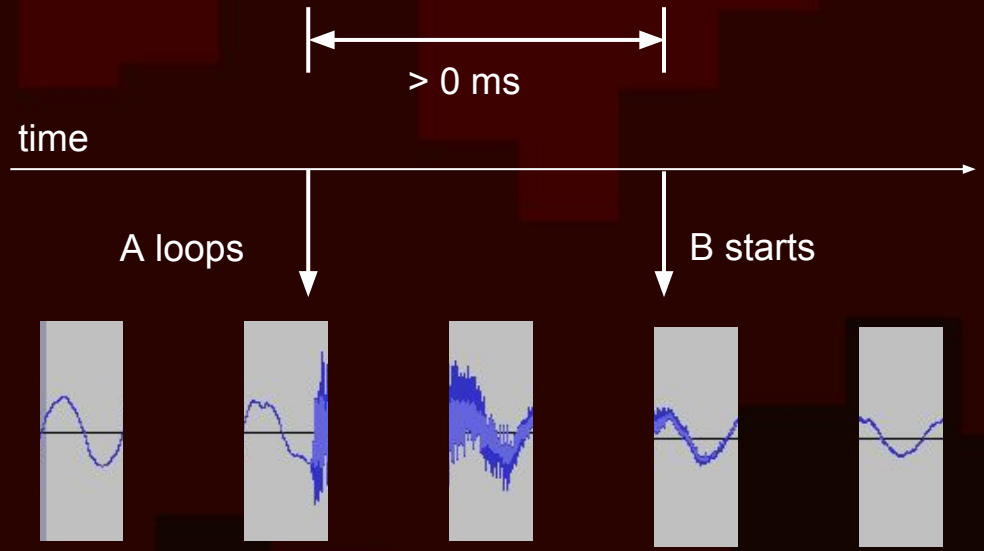
So, if we want to start a sound B exactly when another sound A loops...



# Immediately is Too Late

So, if we want to start a sound B exactly when another sound A loops...

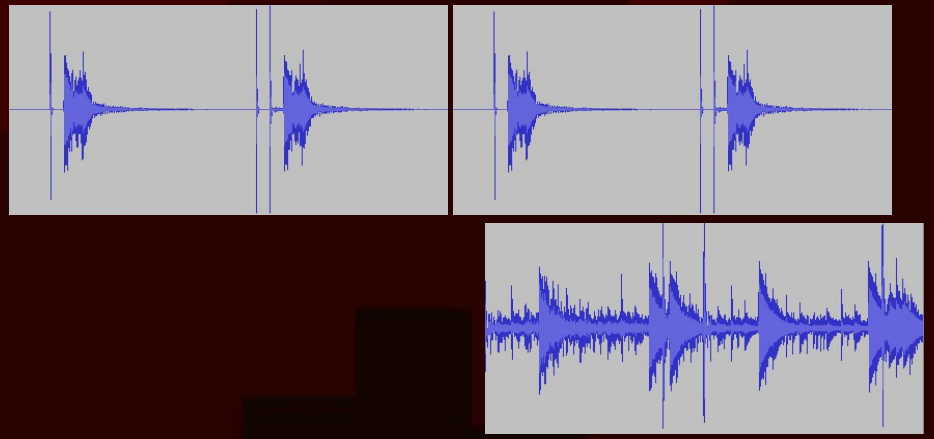
Detecting it in Update and playing B immediately is too late!



# Immediately is Too Late

So, if we want to start a sound B exactly when another sound A loops...

Detecting it in Update and playing B immediately is too late!

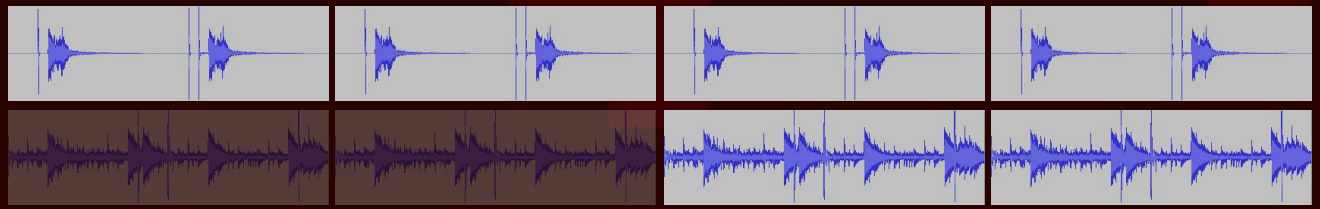


# Interactive Music Solutions

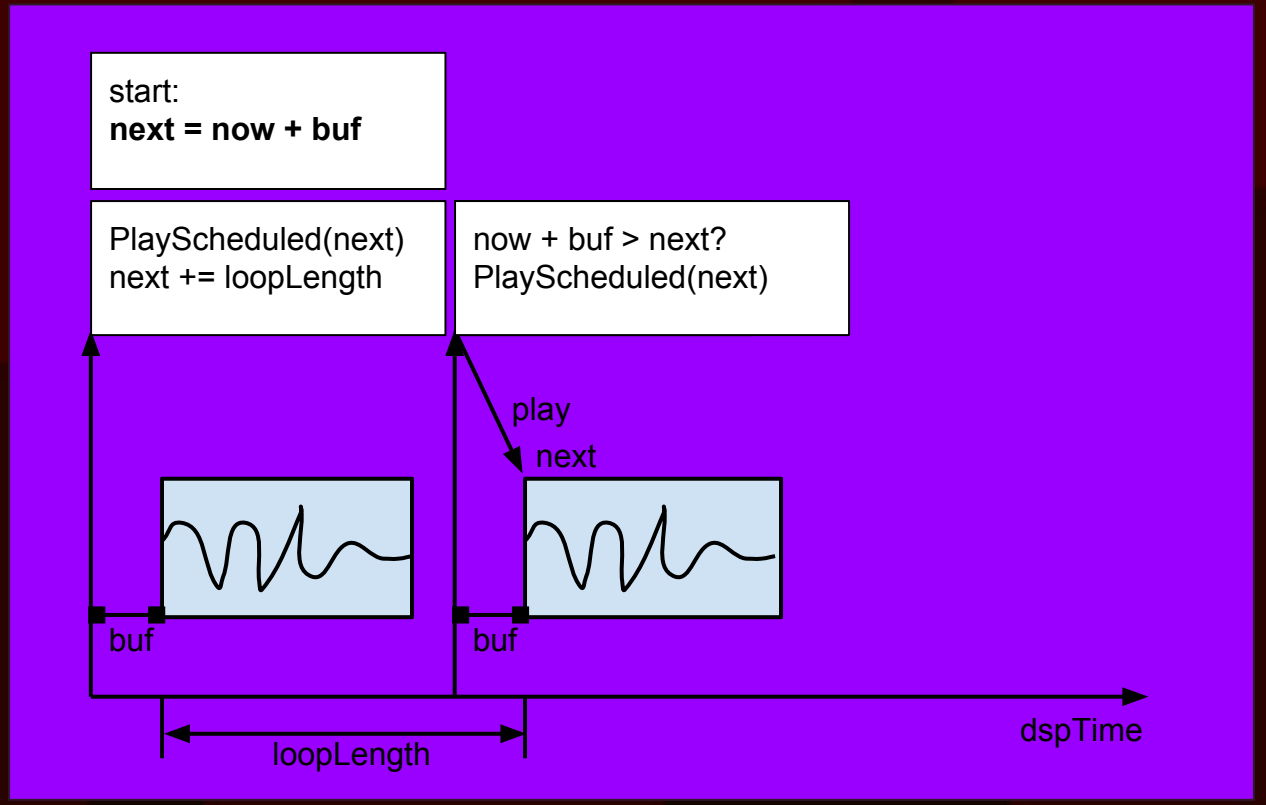
- Synchronized Loops
- PlayScheduled

# Solution A: Synchronized Loops

- All loops should be exactly same length, or integer multiples
- All loops should be started in the same frame, possibly muted
- New loops cannot be started
- Never change pitch



# Solution B: PlayScheduled



# Solution B: PlayScheduled

```
Start:
    buf = 0.1 // as low as possible
    next = AudioSettings.dspTime + buf
Update:
    now = AudioSettings.dspTime
    if(now + buf > next)
        audio.PlayScheduled( next )
        next += loopLength
```

- see http://www.schmid.dk/gallery/play_scheduled/ for C# code

# Solution Comparison

Solution A: Synchronized Loops

- Very simple to implement
- Requires a loop for every single independent musical element
- Loops must be same length or integer multiple
- Pitch cannot be changed

# Solution Comparison

Solution A: Synchronized Loops

- Very simple to implement
- Requires a loop for every single independent musical element
- Loops must be same length or integer multiple
- Pitch cannot be changed

Solution B: PlayScheduled

- Non-trivial implementation
- Flexible: Individual notes can be sequenced
- No length requirement for music sounds
- Pitch can be changed

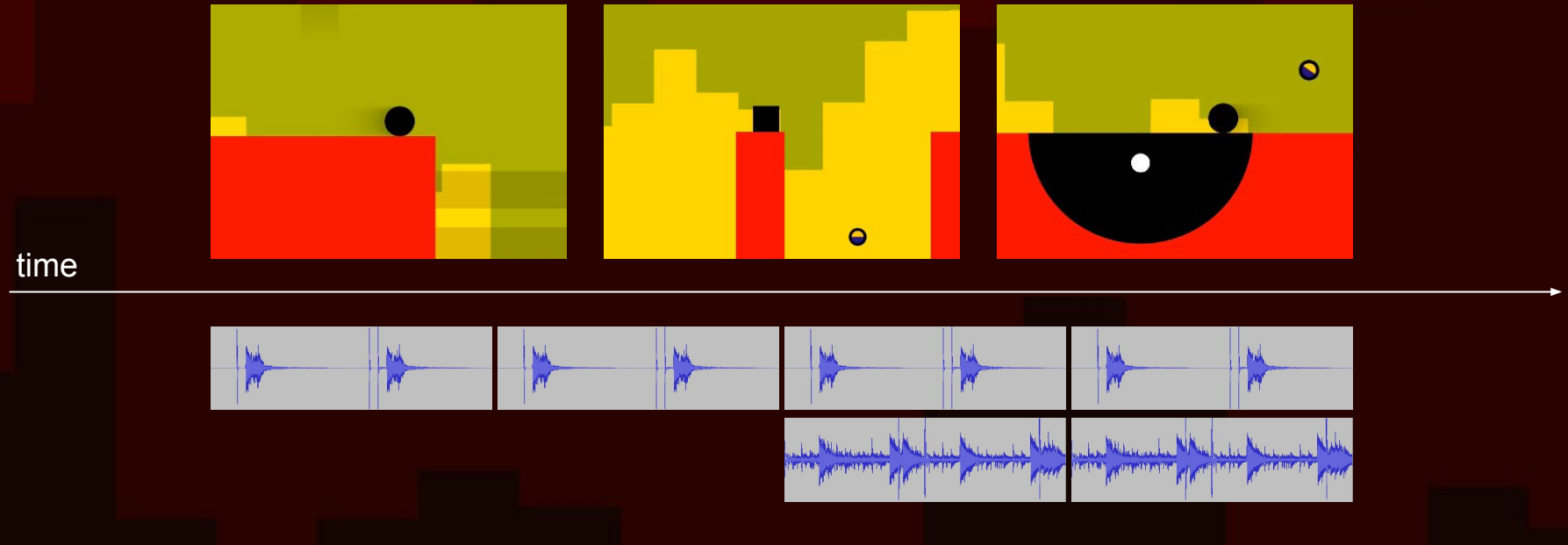
# Summary

- Synchronizing loops with sample accuracy is tricky
- Audio is rendered in buffers, delaying and quantizing sounds
- Unity Update calls correspond to video frames, not audio buffers
- Immediately is too late: detecting loop and reacting in Update results in a delay
- Solution A: Synchronized Loops
- Solution B: PlayScheduled


Music Timing in 140

# Music Timing in 140

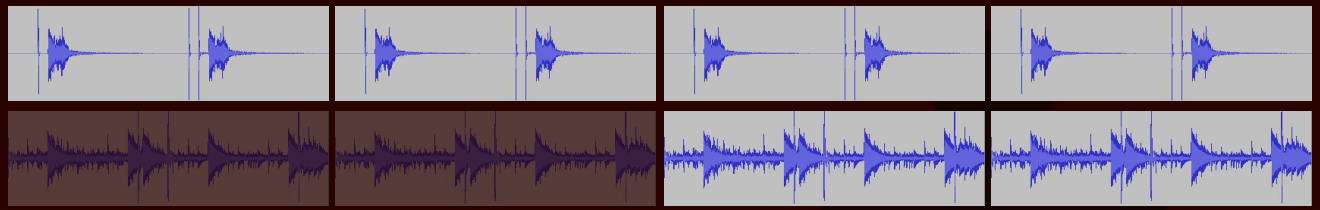
- We wanted the music to be mixed interactively with the gameplay.
- Loops should be sample-accurate.
- We were using Unity 3 at the time, which limited our options (no PlayScheduled)



# Music Requirements

Simple solution with sample-accurate timing:

- All music must be loops of a fixed length, or multiples of that length.
- Start all loops in same frame, possibly muted.



# Music Requirements

Simple solution with sample-accurate timing:

- All music must be loops of a fixed length, or multiples of that length.
- Start all loops in same frame, possibly muted.

During game progression:

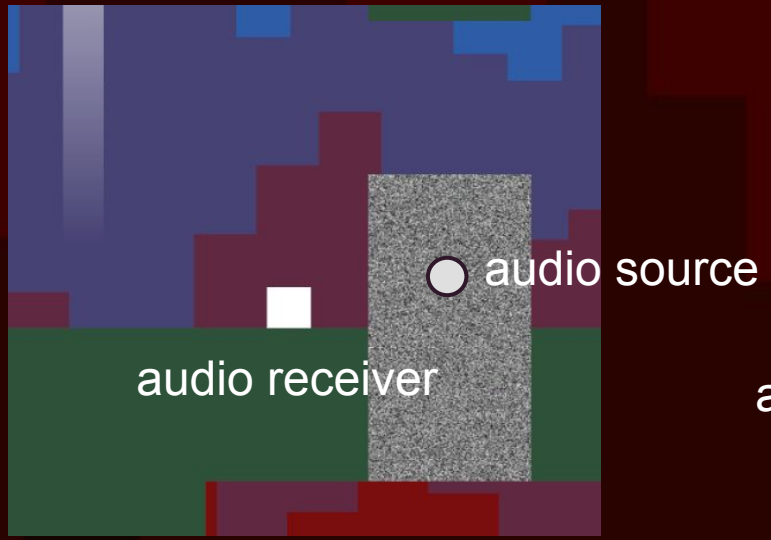
- Control volume/muting and pan.
- Never change pitch unless just before stopping a loop.

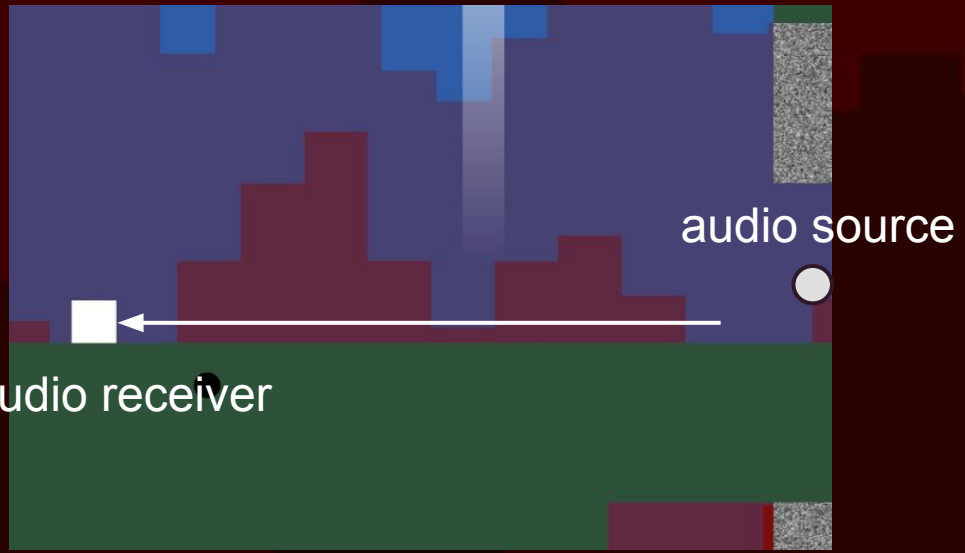
# Localized Music Loops

- Music loops are “physically” placed in level geometry
- Dynamic mixing occurs as player moves around
- Certain areas can gain unique atmosphere based on music

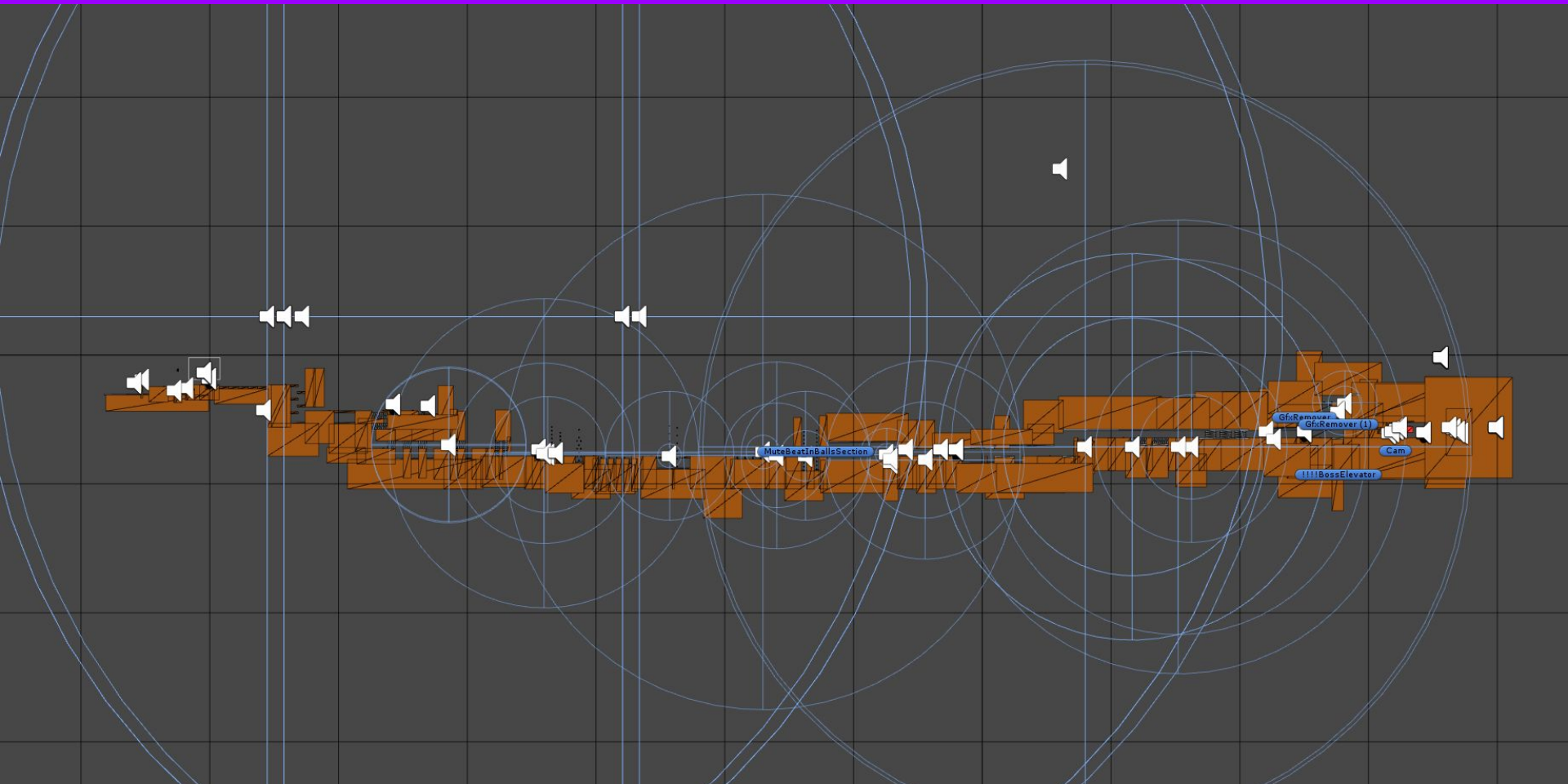
# Localized Music Loops

Simple attenuation and panning for music loops using the built-in audio system



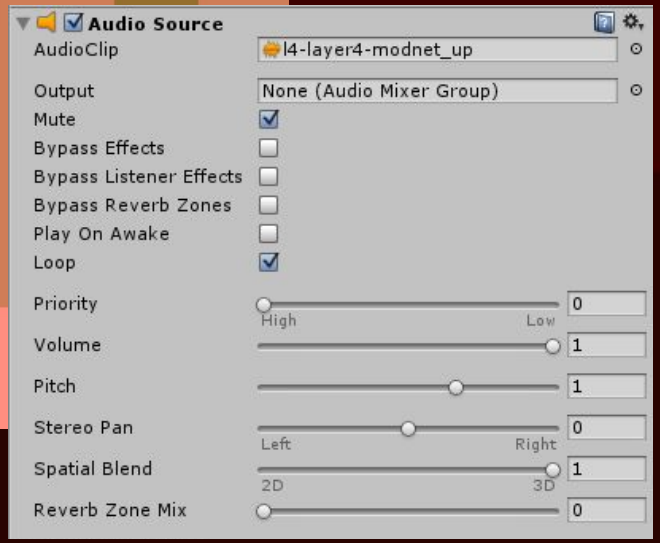



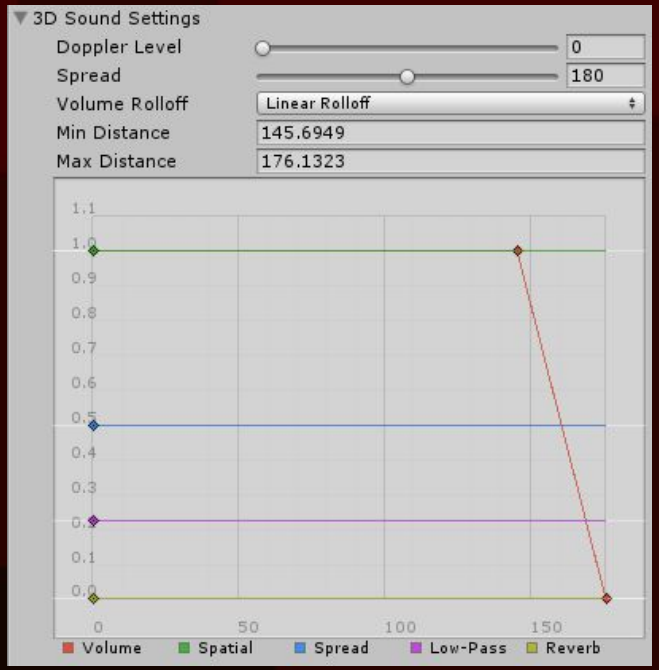
# Localized Music Loops - Level 4



# 140 demo

# Example Audio Loop

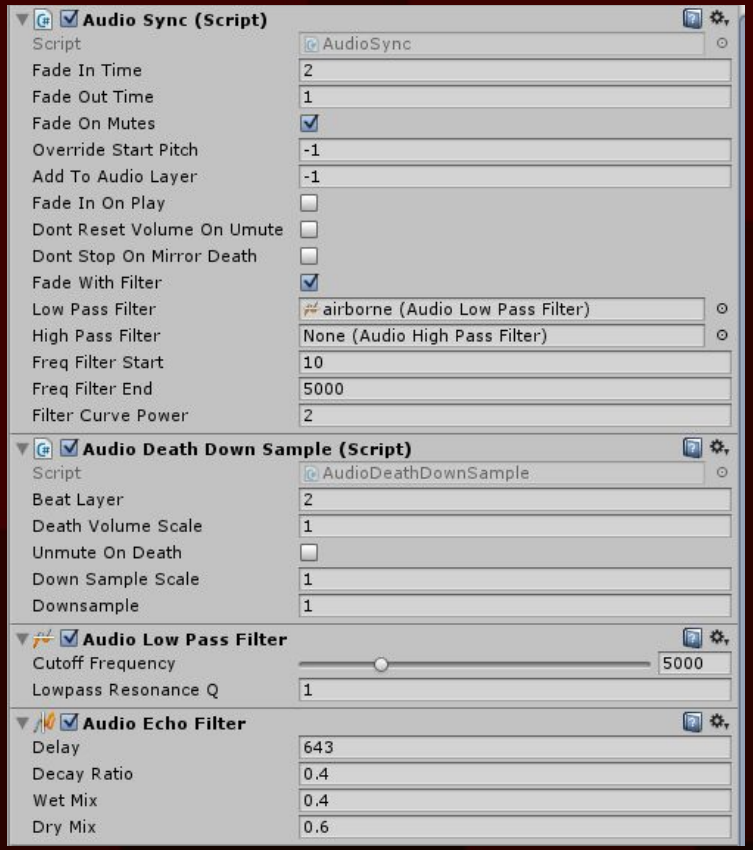


# Example Audio Loop Components

AudioSync component: handles all loops

- Handles fade in/out
- Controls filters
- Adds loop to ‘layer’ (group)



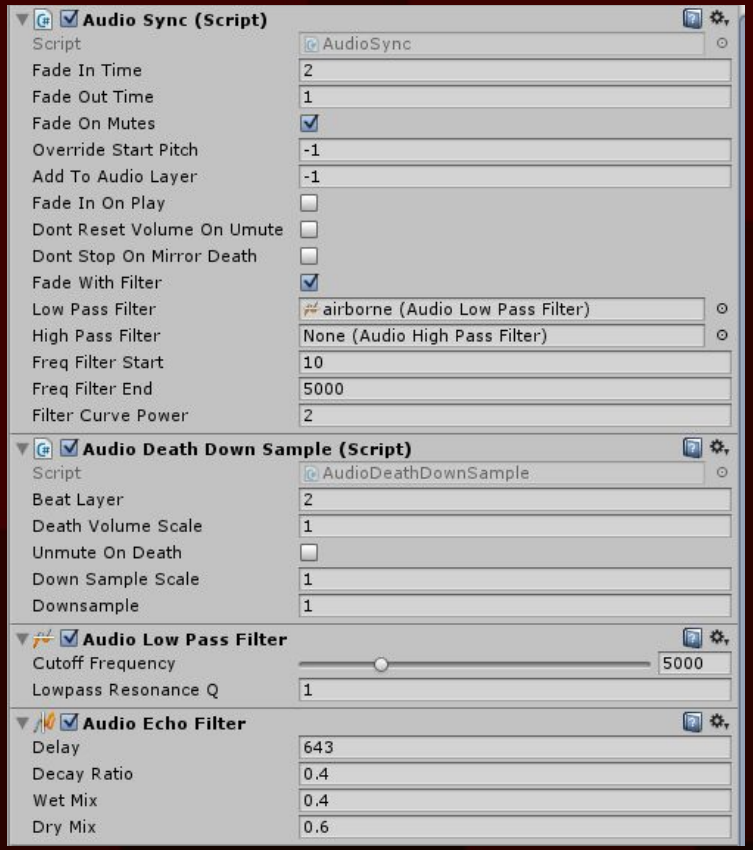
# Example Audio Loop Components

AudioSync component: handles all loops

- Handles fade in/out
- Controls filters
- Adds loop to ‘layer’ (group)

AudioDeathDownSample:

downsample filter (more on this later)



# Example Audio Loop Components

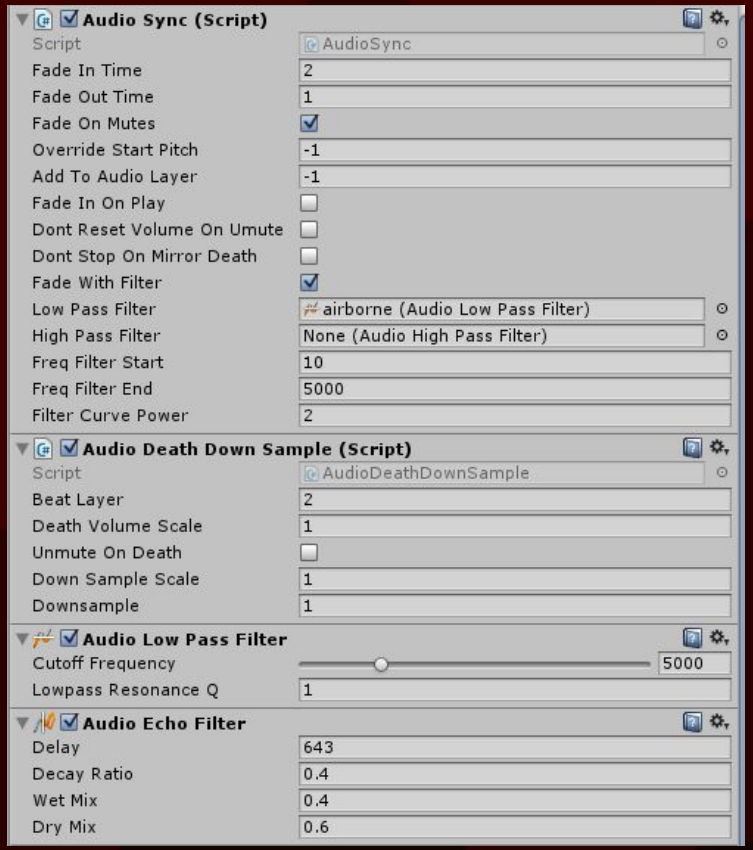
AudioSync component: handles all loops

- Handles fade in/out
- Controls filters
- Adds loop to ‘layer’ (group)

AudioDeathDownSample:

downsample filter

AudioLowPassFilter: built-in Unity LPF



# Example Audio Loop Components

AudioSync component: handles all loops

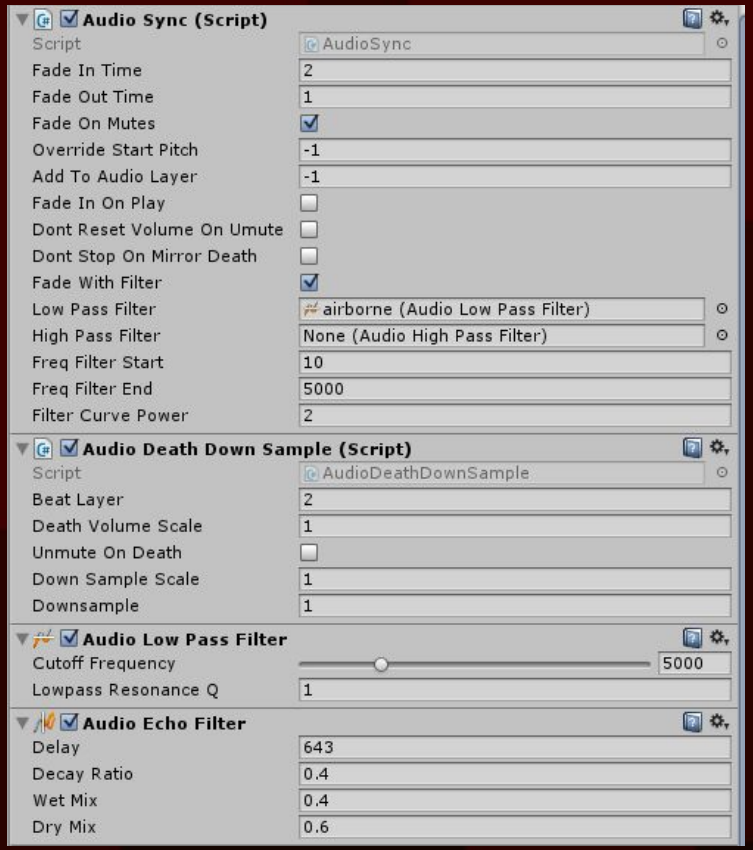
- Handles fade in/out
- Controls filters
- Adds loop to ‘layer’ (group)

AudioDeathDownSample:

downsample filter

AudioLowPassFilter: built-in Unity LPF

AudioEchoFilter: built-in Unity delay

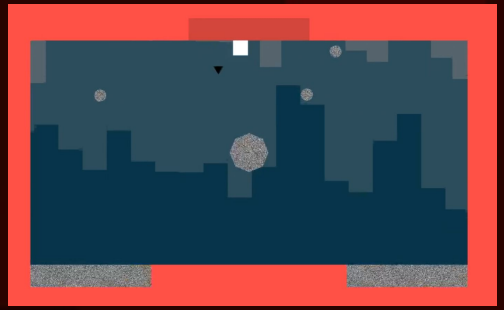


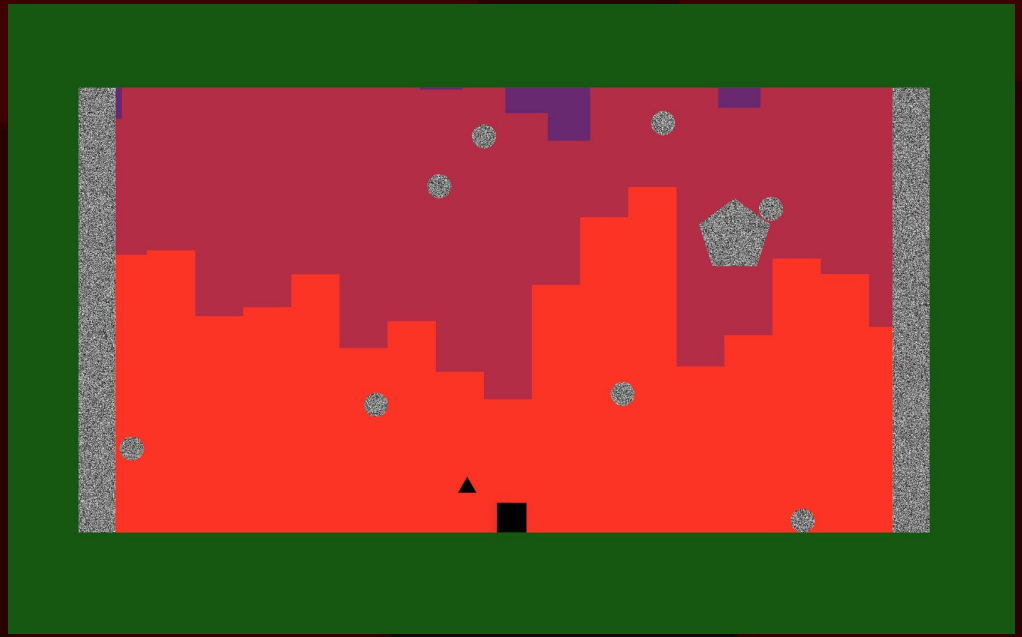
# Masking Loops

Music loops can be masked:

- Fade using filters
- Delay
- Unmuting at specific beats

# Level 4 Boss



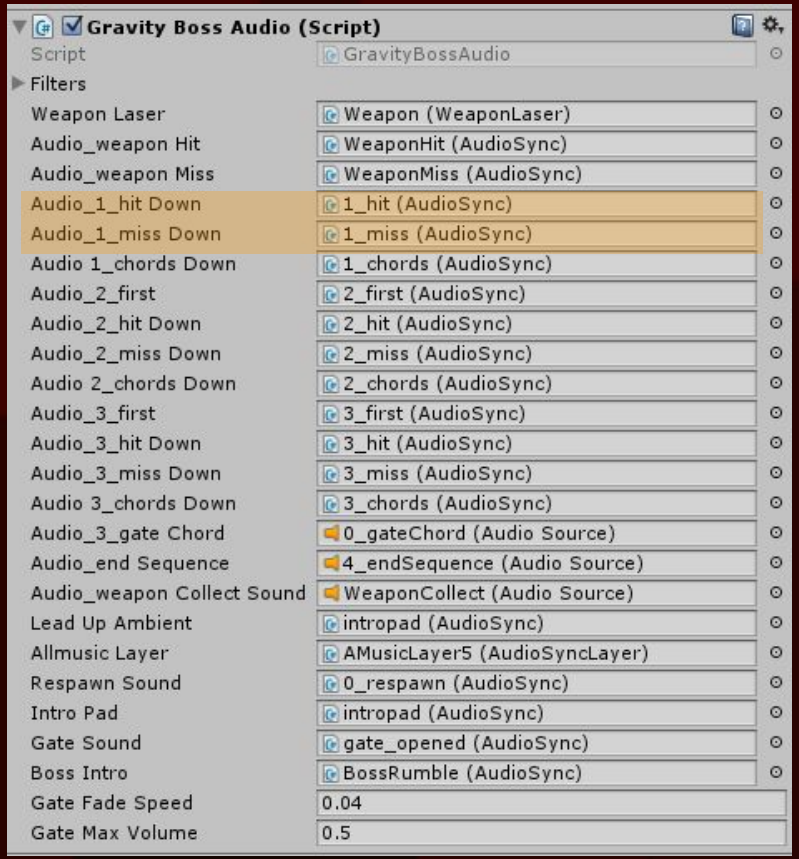


# Level 4 Boss

- Over 20 loops running simultaneously

# Level 4 Boss

- Separate loops for hit and miss
- Muted / unmuted when hit or miss is determined

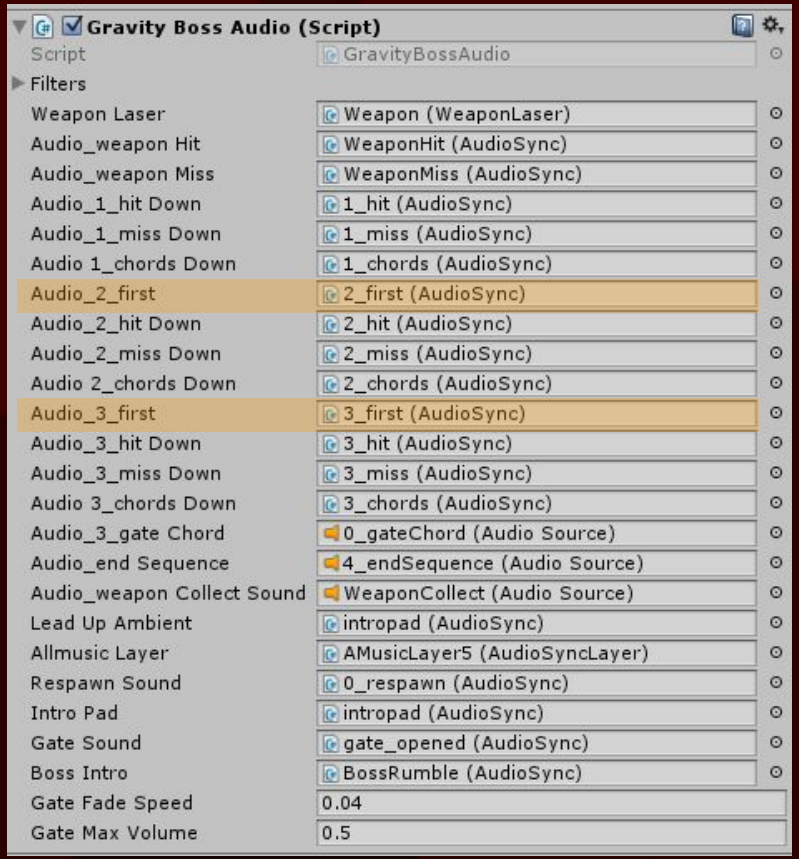


# Level 4 Boss

- Each stage has its own set of loops

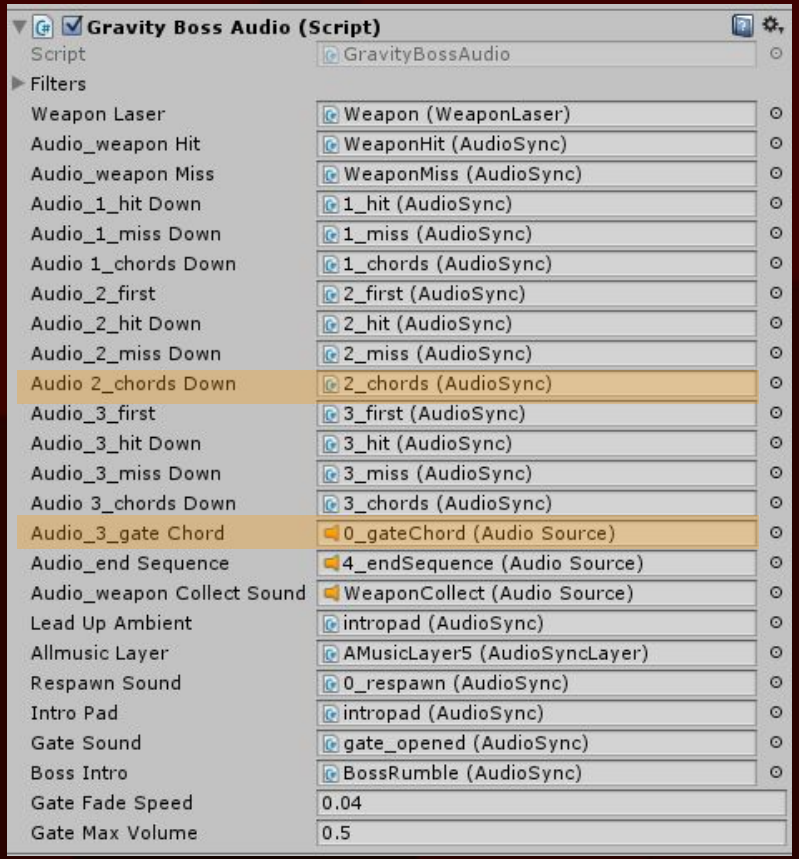
# Level 4 Boss

- Each stage is in a different key
- Loops have long Ableton Reverb tails
- Reverb tails and key change requires special transitional first loop



# Level 4 Boss

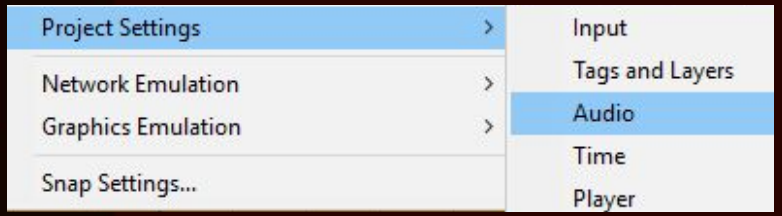
- House chords are faded in using filter between hits to create tension
- Final chord is faded in, anticipating end sequence

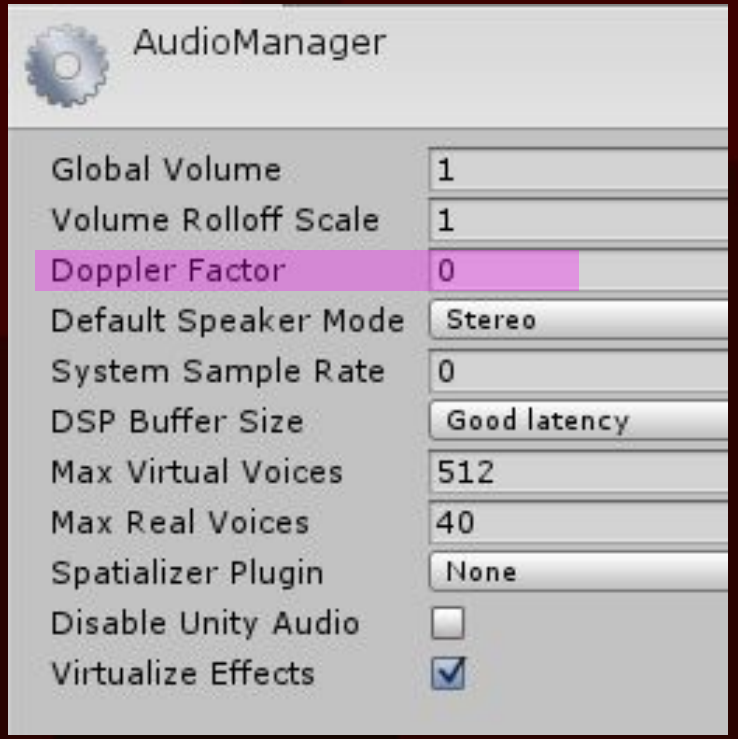


# 140 demo

# Doppler Effect

- Disable Doppler effect to avoid drifting loops!



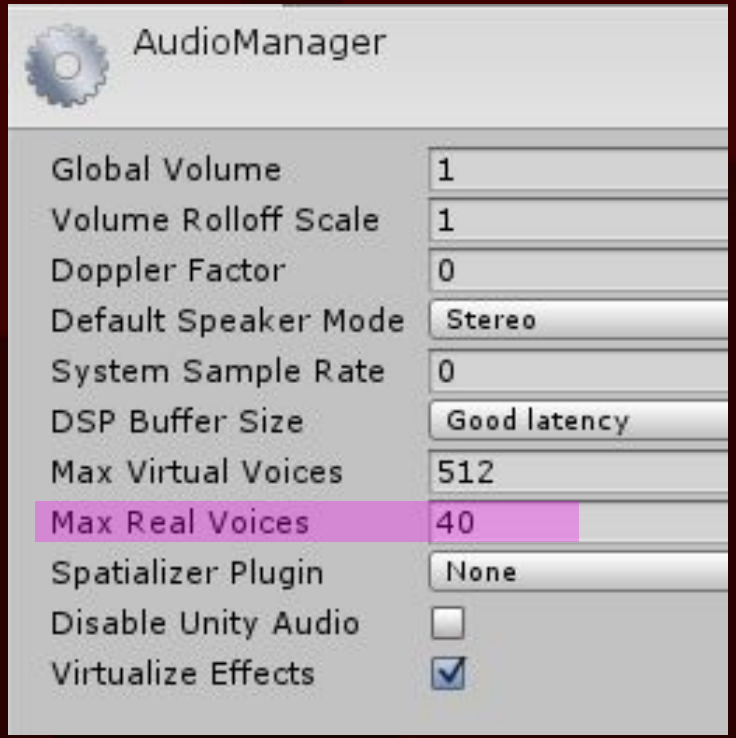



# Max Real Voices

In Unity 3, if more than 32 sounds are playing at once, we lose sample accuracy!

Same limitation in Unity 2017, but the number can be increased.

140 currently uses 40 voices.



# Summary

- In 140, we start all loops at once and control their volume
- Localized music loops: volume and pan using built-in audio system
- Use filters and effects to mask loops
- Level 4 boss music uses muting, fading, and filtering of 20 loops
- Disable Doppler effect
- Check that ‘Max Real Voices’ is enough

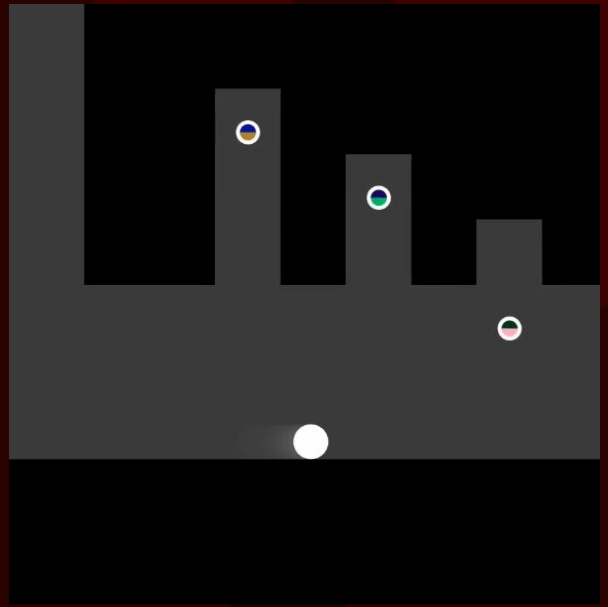
# Fun Audio Tricks

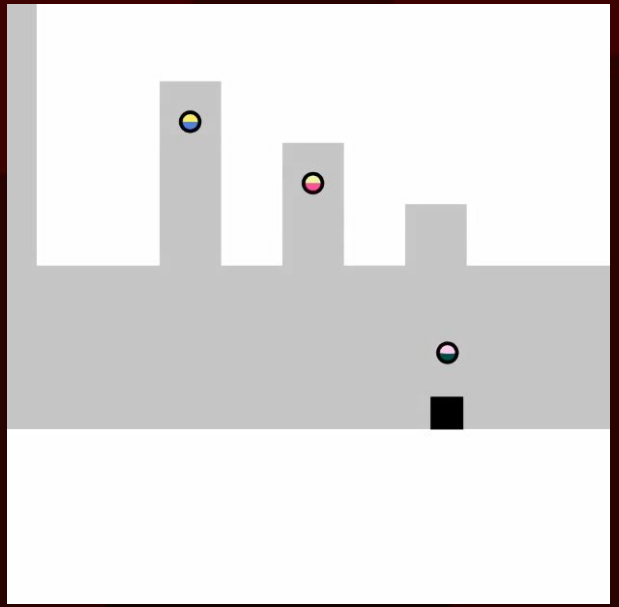
# Fun Audio Tricks

- Modulation
- Cassette tape jam
- Downsampling
- Fake crash

# Menu Modulation

- When picking up a mirror mode key, modulate ambient track from Cm to Dm.
- Track contains no rhythmic elements, so loop synchronization is not an issue.





# From Semitones to Frequency

Modulate ambient track from Cm to Dm:

Frequency of D relative to C (+2 semitones, well-tempered):

$$2^{2/12} \sim 1.12246204830937$$

# From Semitones to Frequency

Modulate ambient track from Cm to Dm:

Frequency of D relative to C (+2 semitones, well-tempered):

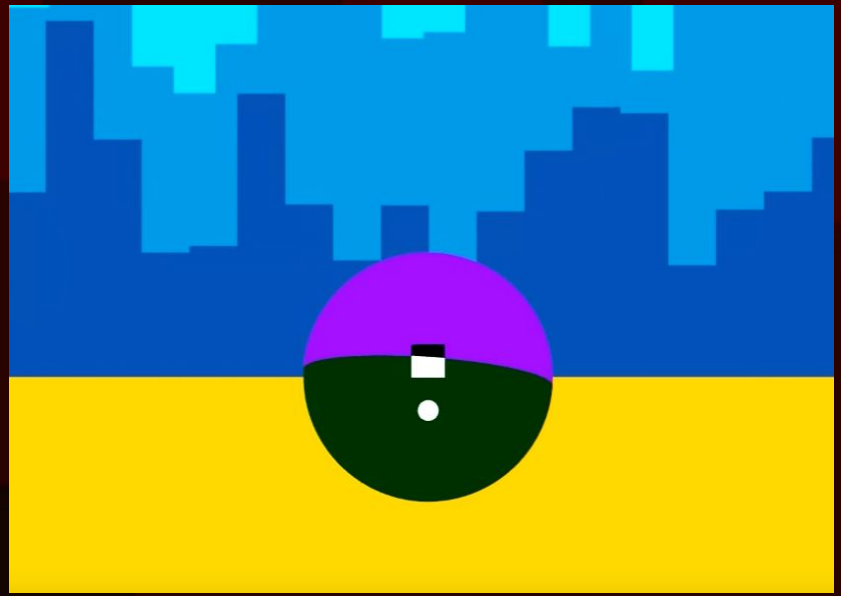
$$2^{2/12} \sim 1.12246204830937$$

Gradual pitch change code, as f goes from 0 to 1:

```
relativePitch = pow(2.0, 2.0 / 12.0)
source.pitch = lerp(1.0, relativePitch, f)
```

# Cassette Tape Jam

When a key is delivered, the playing music is stopped with a cassette tape jam-inspired effect.



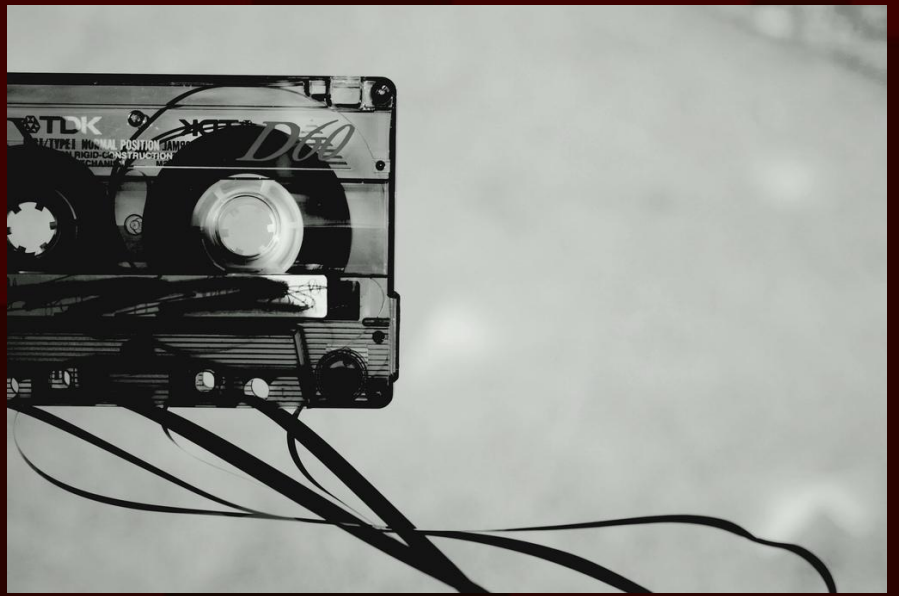



*Image credit: Kristi Bogel*

# Cassette Tape Jam

The tape jam effect is achieved with:

- applying strong vibrato to all music tracks,
- enveloping pitch towards 0 and volume towards silence.

# Cassette Tape Jam Code

The tape jam effect is achieved with:

- applying strong vibrato to all music tracks,
- enveloping pitch towards 0 and volume towards silence.

As f goes from 0 to 1:

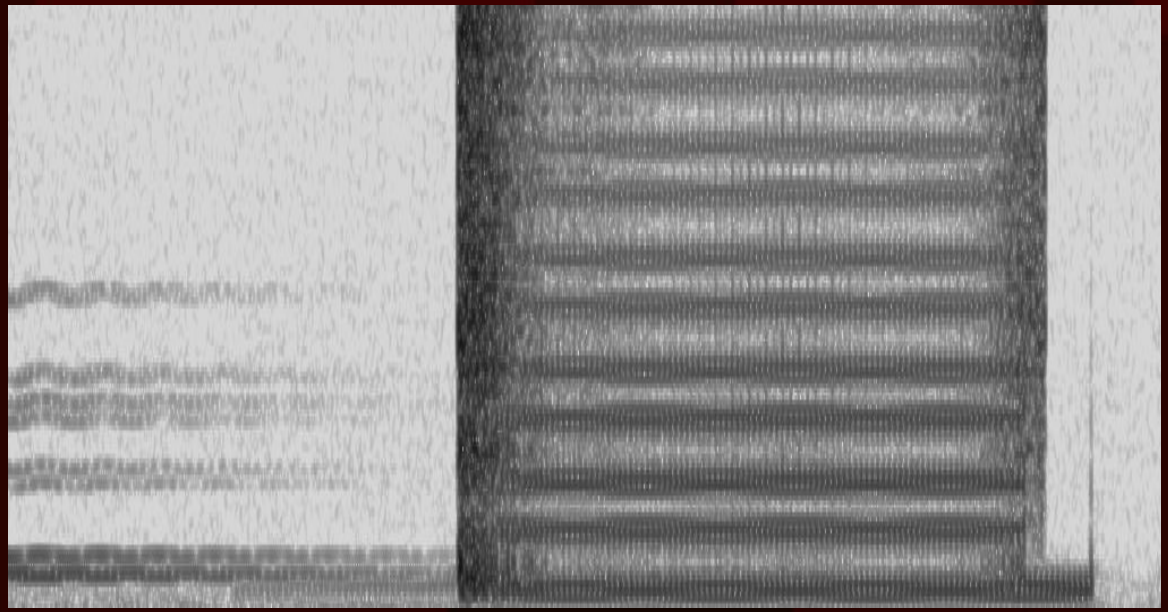
```
source.pitch = (1-f)                          // pitch envelope
             + sin(time * FREQ) * STRENGTH    // vibrato
source.volume = lerp(maxVolume, 0, f)         // amp envelope
```

# Downsampling

When the player dies, the visuals turn black and white, and the audio is brutally distorted.

# Downsampling

- The death audio effect is a simple variable downsampling filter.
- It sounds ugly on purpose.



# Downsampling Filter Code

The simplest variable downsampling filter:

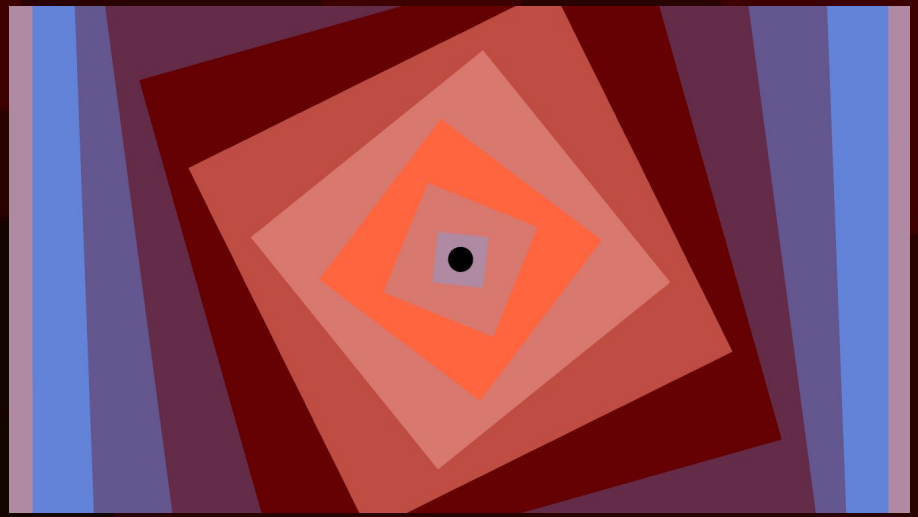
repeat every D’th sample D times.

```
void OnAudioFilterRead(float[] data, int channels) {
  if (D > 1) {                                  // if filter active
    for (int s = 0; s < data.Length; s+=2) {    // for all samples
      data[s]   = data[s / D * D];              // left channel
      data[s+1] = data[s / D * D + 1];          // right channel
    }
}
```

# Fake crash

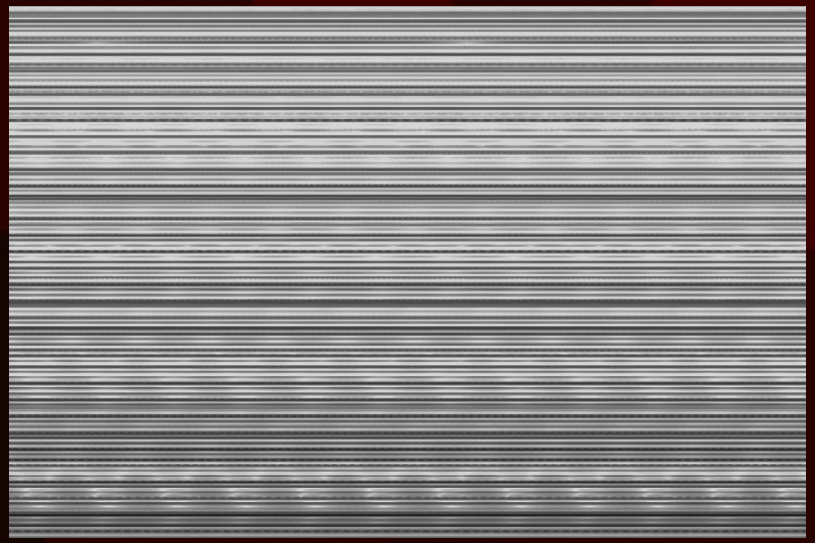
When the final boss is beaten, the game simulates the game crashing.

Or rather, how the game *would* crash if it was running on a SEGA Genesis.



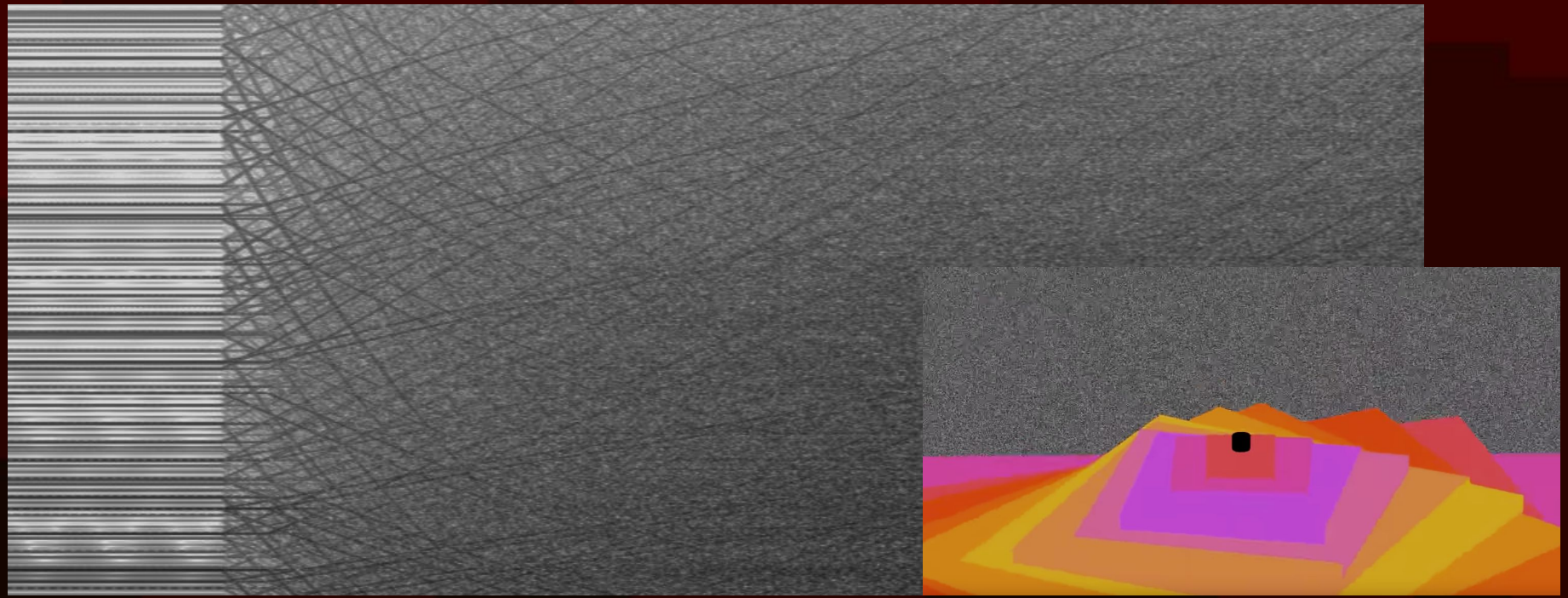
# Fake crash

- The final chord of the boss fight and the screen is unchanged for 9 seconds, leaving at least one YouTuber very nervous.
- Crashes on old oscillator-based systems would have similar behaviour.



# Glissando and 3D Rotation

- The oscillators slowly starts individually wandering towards a final chord.
- The game rotates the view, for the first time exposing a 3D world.



# Summary

- Pitch change on playing track works for ambient music.
- Vibrato and amplitude and pitch envelopes simulate cassette tape jam.
- Downsampling filter is implemented `OnAudioFilterRead` method.
- Game ends with fake crash sound with hanging oscillators.
- Fake crash is resolved with oscillators gliding towards final chord.

# Questions?