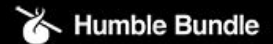


140

Sonic College 2020
Jakob Schmid

Jeppe Carlsen (design, programming)
Niels Fyrst, Andreas Peitersen (visual design)
Jakob Schmid (audio)

Developed as hobby project over 3 years



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



Side A	Side B
140 Title 2:00	140 Part 1 5:07
140 Part 1 0:20	140 Part 2 5:23
140 Part 2 0:20	140 Part 3 5:07
140 Part 3 0:20	140 Part 4 4:48
140 Part 4 0:20	140 Part 5 4:48
140 Part 5 0:20	140 Menu 2:48

Composed and produced by Jakob Schmid
www.schmid.dk / www.carlsongames.com
Created with Ableton Live, Reaktor, Kurz X1 software synthesizer
Vinyl produced by [Lambbit](#) www.lambbit.com 0821-0871
Mastered for vinyl by David Gardner, Supersonic Mastering
The game 140 was created by Jeppe Carlsen, Jakob Schmid, Niels Fyrst, Andreas Arnlid Pedersen
Thanks to Martin Stig Andersen, Peter Ruchardt, Mikkel Svendsen, Mikkel Gjel, SBS Gunner Nyberg, Christian Vogel, Sprog Ruse, Jan W. Sillem, Renee White, Christian Villan
140 is a Carlsen Game. The 140 soundtrack by Jakob Schmid is licensed under a Creative Commons Attribution 4.0 International License. To view a copy of this license, visit www.creativecommons.org/licenses/by/4.0/

Includes:
Digital Soundtrack
Steam Code for Full Game

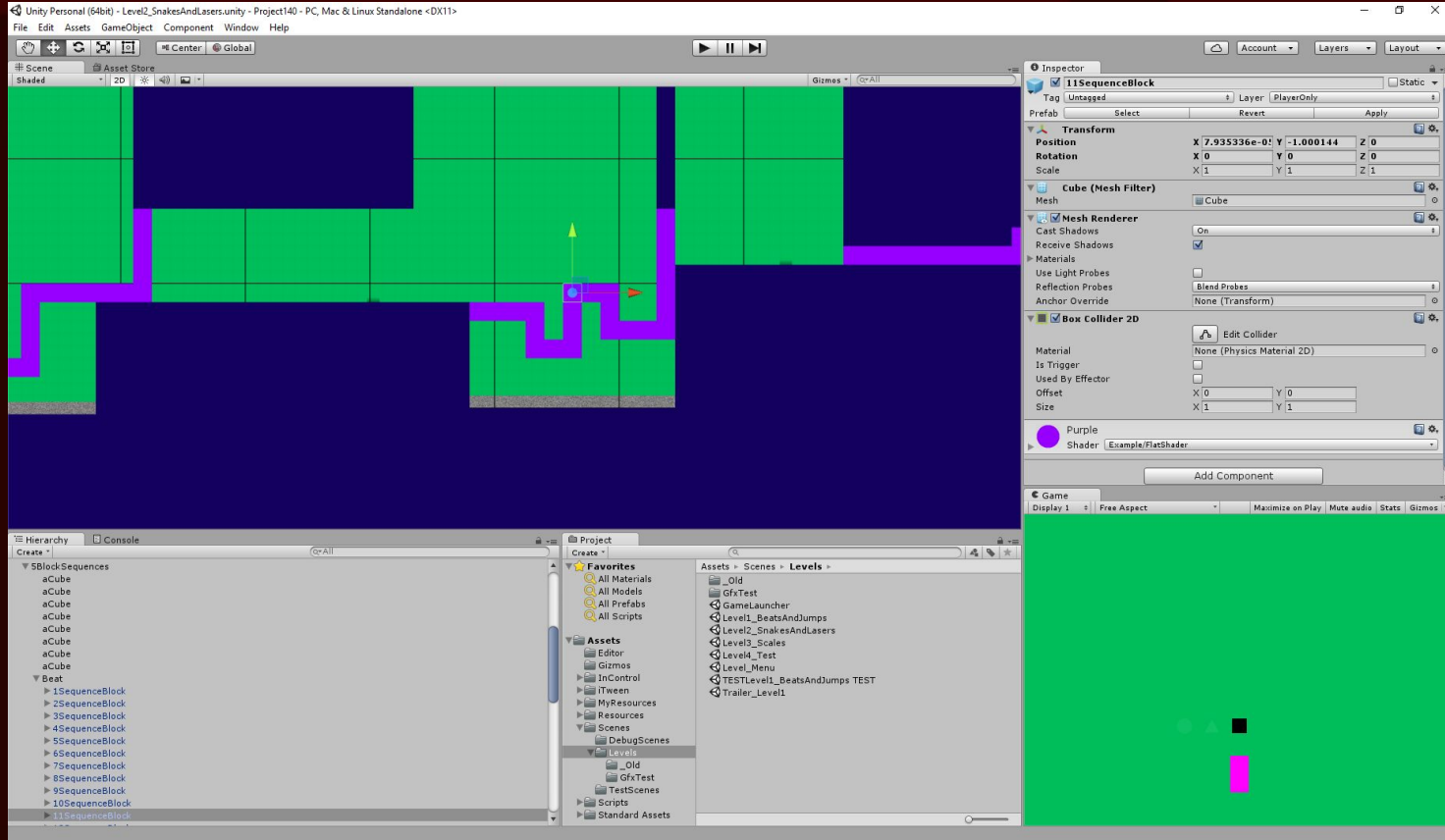
Music by:
Jakob Schmid

Carlsen Games
iam8bit
iam8bit.com

140 Vinyl Soundtrack
Limited Edition of 1400

<https://schmid.dk/games/140/soundtrack/>

Developed in Unity 3



Ableton Live

The screenshot displays the Ableton Live 9 Suite interface, showing a multi-track session view. The top menu bar includes File, Edit, Create, View, Options, and Help. The session view is organized into tracks, each with a name and a color-coded header. The tracks are:

- 1 Drum
- 2 Drum
- clap ve
- kywhl snrl
- laserba
- cuta drums
- real drums
- sub
- 9 level0
- theme
- l2-bass
- l2-swell
- l2-JK-noise
- l2-JK-jump
- l2-JK-build
- DONTUS
- bass
- acid
- acid
- Master

Each track has a MIDI or Audio input section, a monitor section, and a volume/gain section. The MIDI tracks have various MIDI routing options, and the audio tracks have volume and pan controls. The Master track has a volume control and a solo button. The interface also shows a Drum Rack, Instrument Rack, Operator, LFO, Auto Pan, and Saturator sections at the bottom.

The Drum Rack section shows a grid of drums with various samples and triggers. The Instrument Rack section shows a selected instrument (Key) with various parameters and a waveform display. The Operator section shows a signal flow diagram with various processing blocks. The LFO section shows a low-frequency oscillator with various parameters. The Auto Pan section shows a panning automation envelope. The Saturator section shows a saturation effect with various parameters.

Audacity

The screenshot displays the Audacity software interface. At the top, the menu bar includes File, Edit, View, Transport, Tracks, Generate, Effect, Analyze, and Help. Below the menu bar is a toolbar with various icons for playback, editing, and analysis. The main window shows a spectrogram of an audio file, with the frequency spectrum on the vertical axis (ranging from 0.0k to 22.1k Hz) and time on the horizontal axis (ranging from 29.0 to 48.0 seconds). The spectrogram shows a complex waveform with many vertical lines, indicating a high-frequency signal. The bottom status bar shows the Project Rate (48000 Hz), Snap To (Off), Selection Start (002,245,446 samples), End (000,000,000 samples), and Audio Position (000,000,000 samples). The status bar also indicates that the audio is Stopped.

140.4

File Edit View Transport Tracks Generate Effect Analyze Help

Click to Start Monitoring MME Stereo Mix (Realtek Hi 2 (Stereo) Ret Speakers (Realtek High

29.0 30.0 31.0 32.0 33.0 34.0 35.0 36.0 37.0 38.0 39.0 40.0 41.0 42.0 43.0 44.0 45.0 46.0 47.0 48.0

X 140.4 22.1k
Stereo, 44100Hz
32-bit float
Mute Solo

20.0k
19.5k
19.0k
18.5k
18.0k
17.5k
17.0k
16.5k
16.0k
15.5k
15.0k
14.5k
14.0k
13.5k
13.0k
12.5k
12.0k
11.5k
11.0k
10.5k
10.0k
9.5k
9.0k
8.5k
8.0k
7.5k
7.0k
6.5k
6.0k
5.5k
5.0k
4.5k
4.0k
3.5k
3.0k
2.5k
2.0k
1.5k
1.0k
0.0k

Project Rate (Hz): 48000 Snap To: Off Selection Start: 002,245,446 samples End: 000,000,000 samples Audio Position: 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

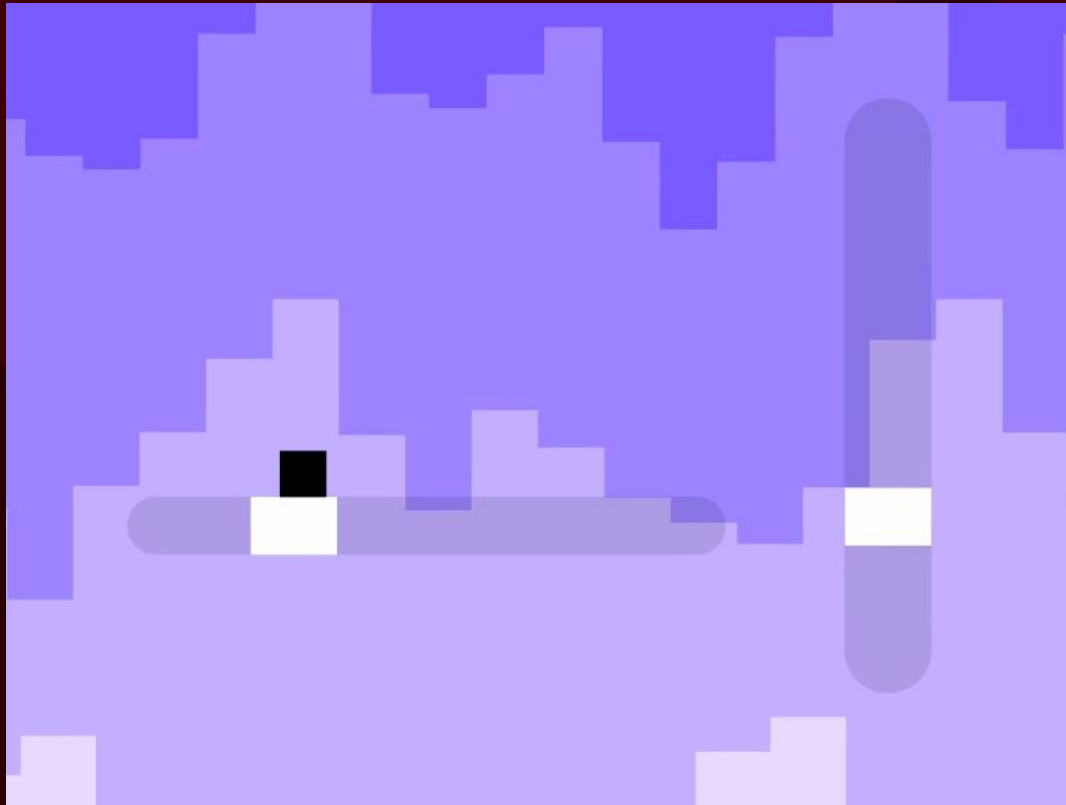
level 1, movers



Control Game from Music



Moving a Platform



Moving a Platform



wait for 16th note #1



start moving



wait for 16th note #8



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:

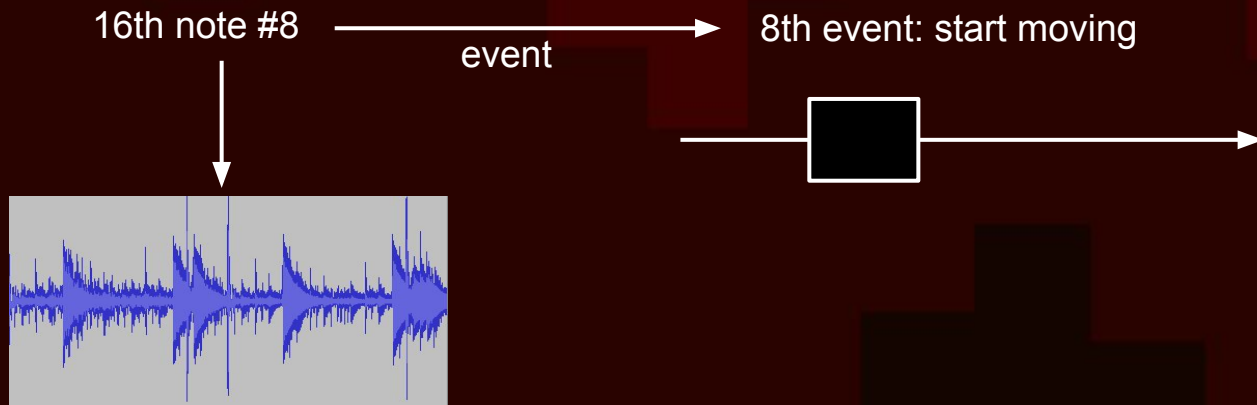
- Get audio time from playing loop
- When next musical beat reached, raise event
- On the 8th event, do something

16th note #8



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?

$$140 \text{ beat/m} * 4 \text{ note/beat} = 560 \text{ note/m}$$



Tempo

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

$$\begin{aligned} 140 \text{ beat/m} * 4 \text{ note/beat} &= 560 \text{ note/m} \\ &= 560/60 \text{ note/s} \end{aligned}$$



Tempo

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

$$\begin{aligned} 140 \text{ beat/m} * 4 \text{ note/beat} &= 560 \text{ note/m} \\ &= 560/60 \text{ note/s} \end{aligned}$$

This means that we have:

$$60/560 \text{ s/note}$$



Tempo

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

$$\begin{aligned} 140 \text{ beat/m} * 4 \text{ note/beat} &= 560 \text{ note/m} \\ &= 560/60 \text{ note/s} \end{aligned}$$

This means that we have:

$$\begin{aligned} &60/560 \text{ s/note} \\ &\approx 0.10714 \text{ s/note} \end{aligned}$$



Moving a Platform



wait for 16th note #0



start moving



wait for 16th note #8



start moving

Moving a Platform



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

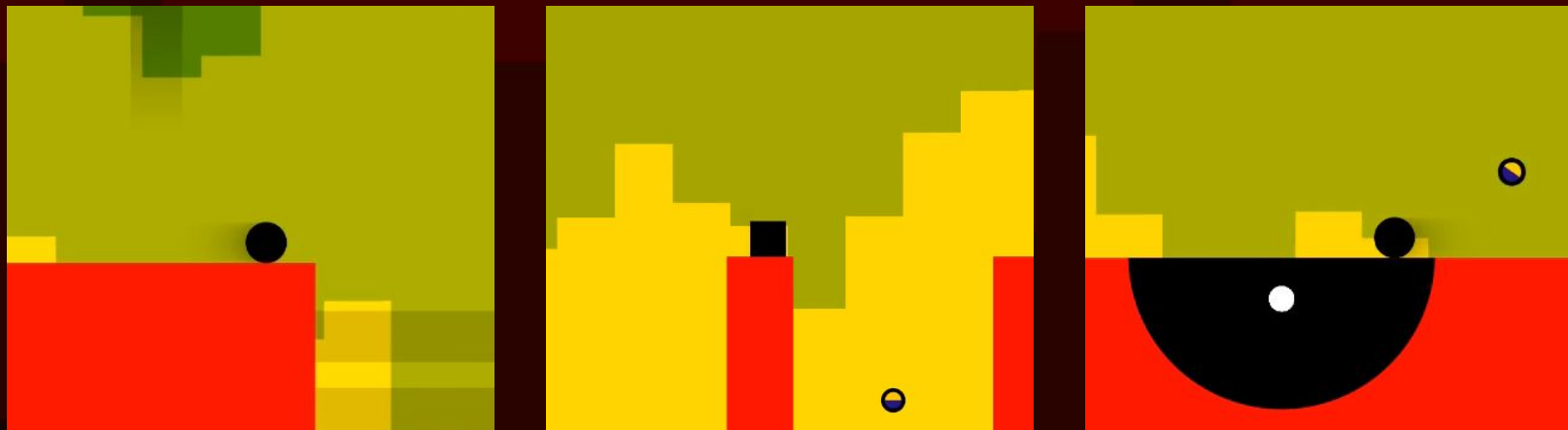


Interactive Music

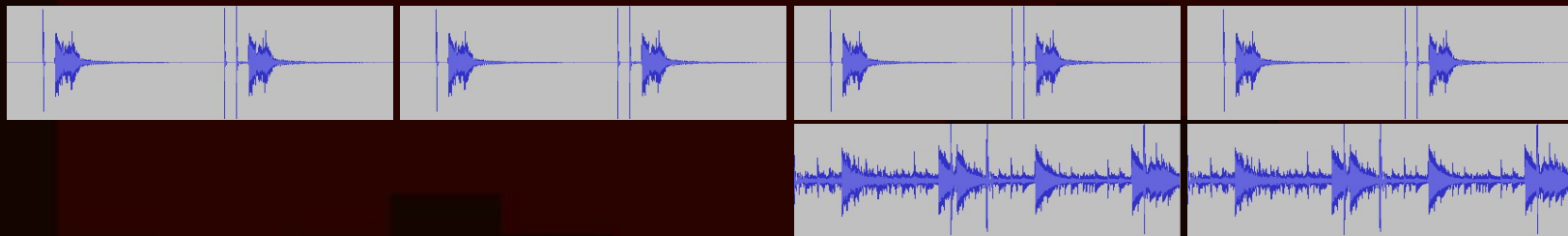
The background features a complex, abstract composition of overlapping shapes. A large, bright red area occupies the upper right portion. A vibrant purple shape is prominent in the lower left and extends upwards. Dark green, jagged shapes are scattered in the lower right. The overall effect is a dynamic, multi-colored field.

Interactive Music Mixing

We wanted to mix music interactively in Unity.



time



The Music Timing Problem

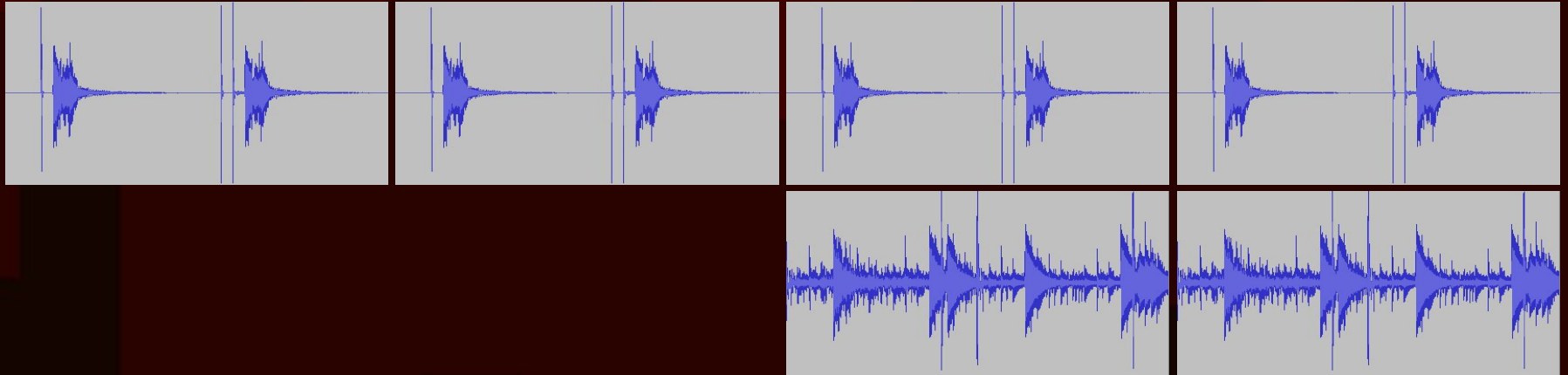
For beat-oriented music, loops should be synchronized with sample accuracy.

- That means a precision of 0.00002 s

Loop Transition

Goal:

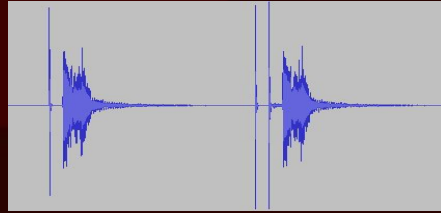
- Start loop A and let it run for a while.
- Then start loop B.
- B should be sample-accurately synchronized with loop A.



Loop Transition Problem

Start loop A:

```
audioSourceA.Play()
```



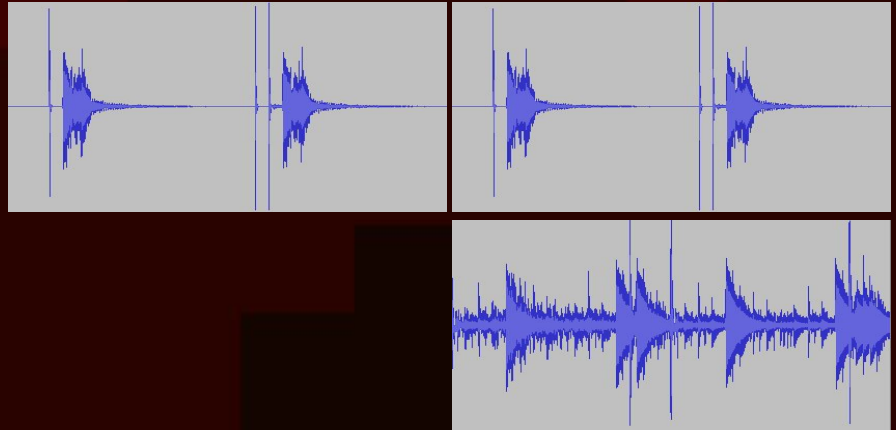
Loop Transition Problem

Start loop A:

```
audioSourceA.Play()
```

Exactly when A loops, start loop B:

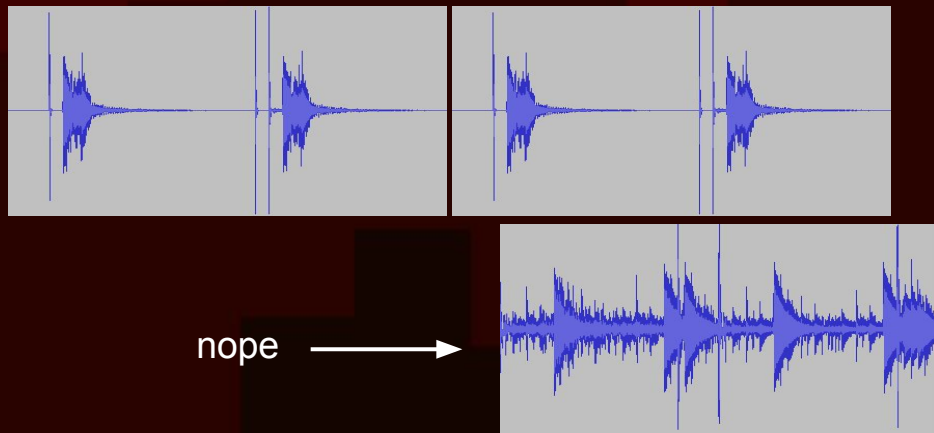
```
Wait for A to loop, then:  
audioSourceB.Play()
```



Loop Transition Problem

It doesn't work!

New sound is out of sync.

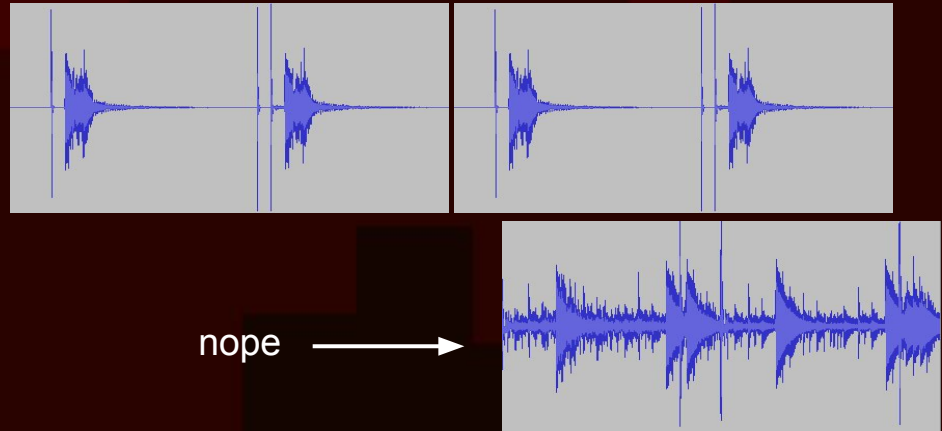


Loop Transition Problem

It doesn't work!

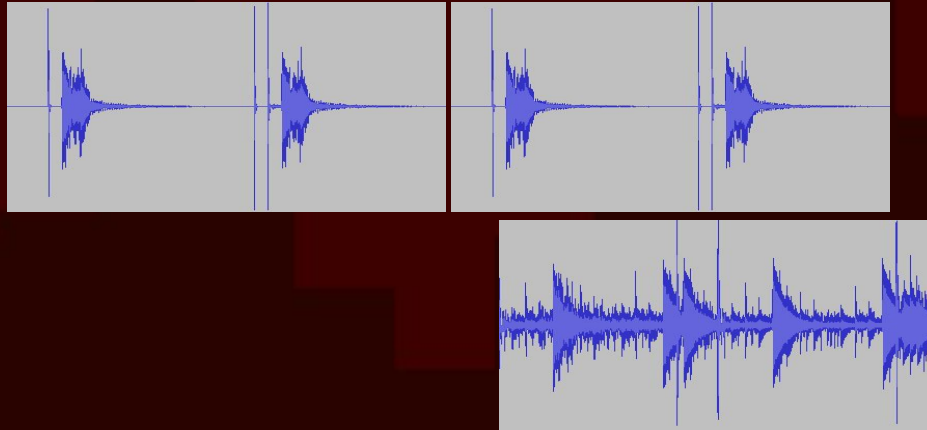
New sound is out of sync.

The problem exists in every sound engine.



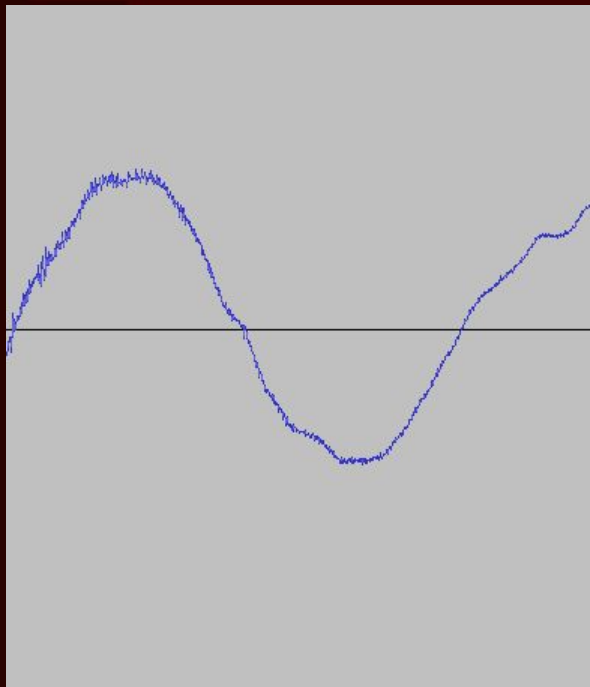
Loop Transition Problem

Why?



Audio Rendering

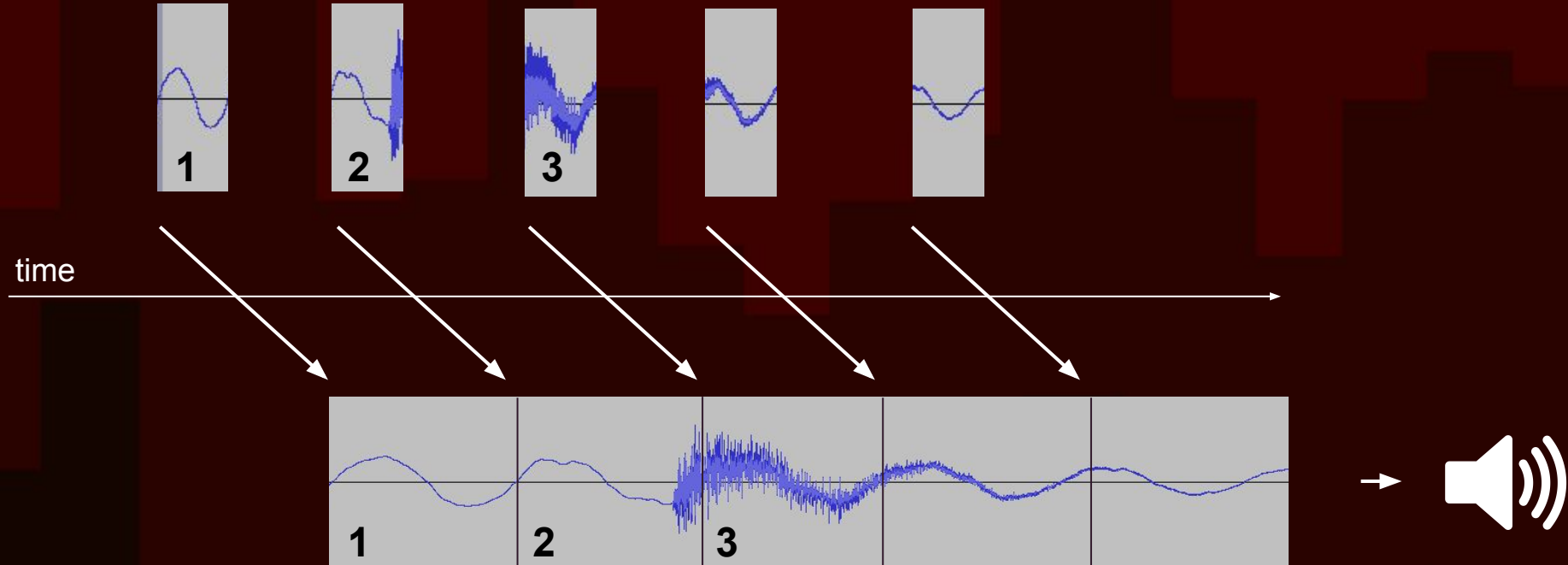
Audio is rendered a fixed number of samples at a time:



← 1024 sample buffer, 21 ms

Audio Rendering

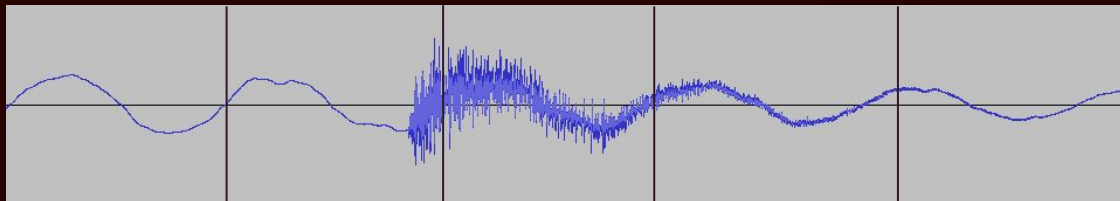
The sound card plays a buffer while the next one is being rendered:



Audio Rendering

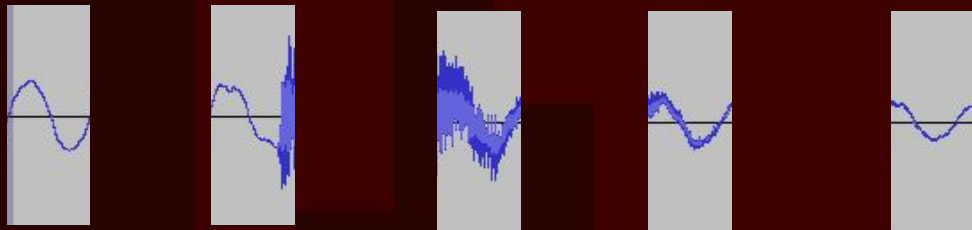
If buffers are e.g. 1024 samples long, we need a new one every 21 ms.

← 21 ms →



Audio Rendering

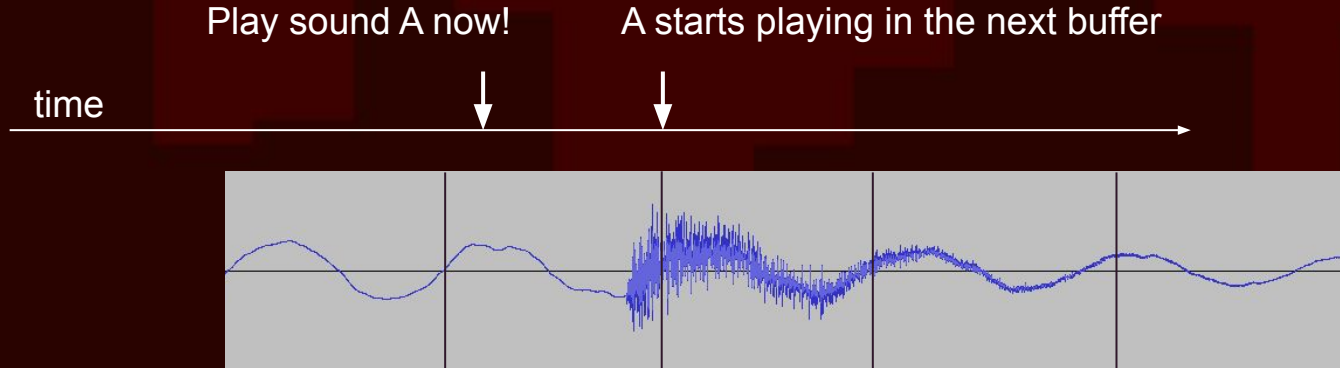
In this case, a new buffer is rendered every 21 ms:



← 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;

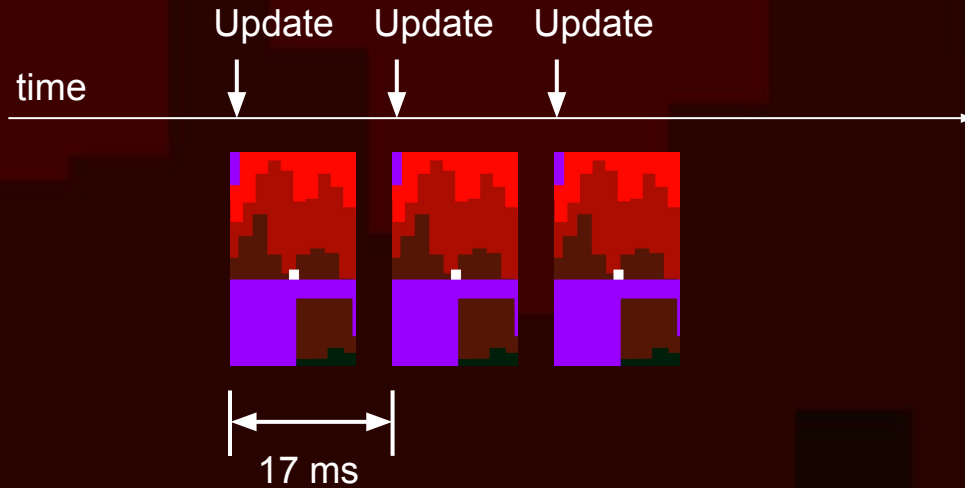
public class MyAwesomeScript : MonoBehaviour
{
    public AudioSource mySound;

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

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

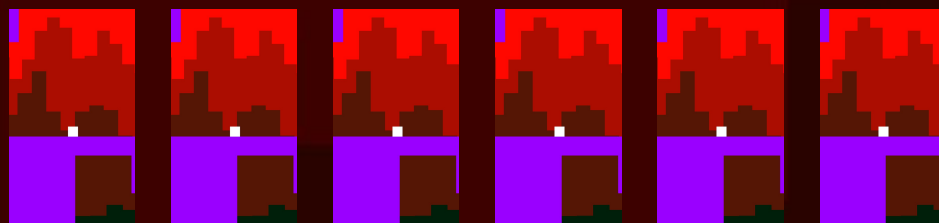
Audio Rendering

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

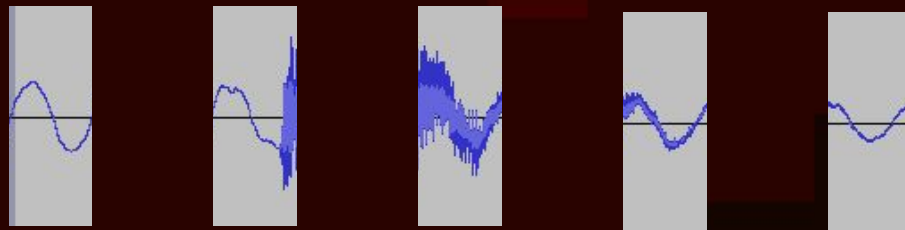


Audio Rendering

Audio buffers and video frames are **not** synchronized:



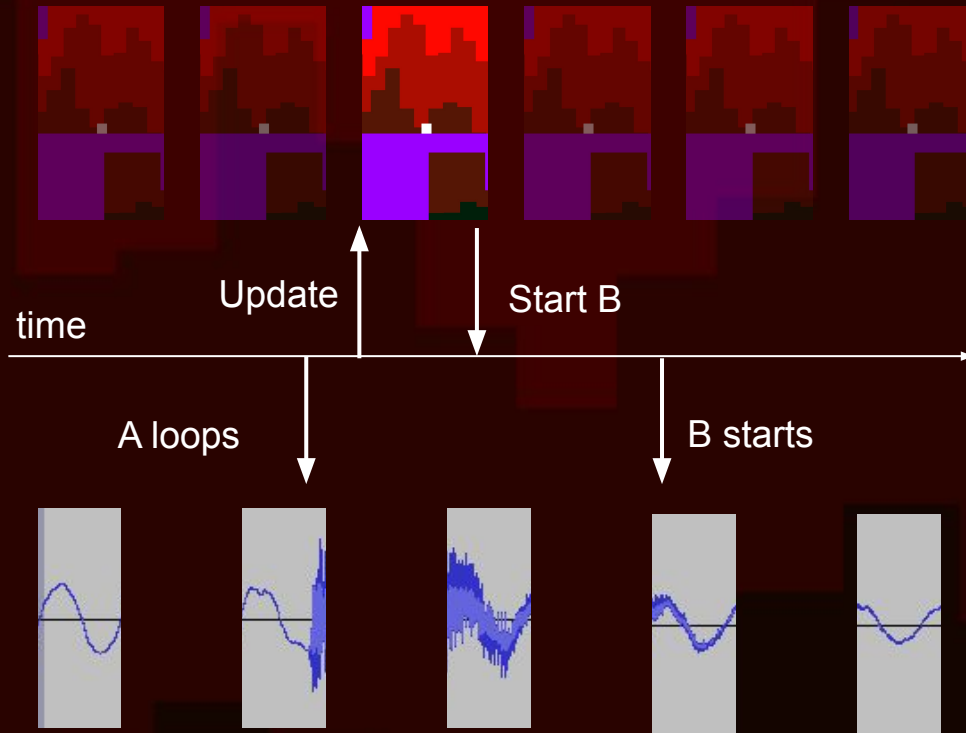
17 ms



21 ms

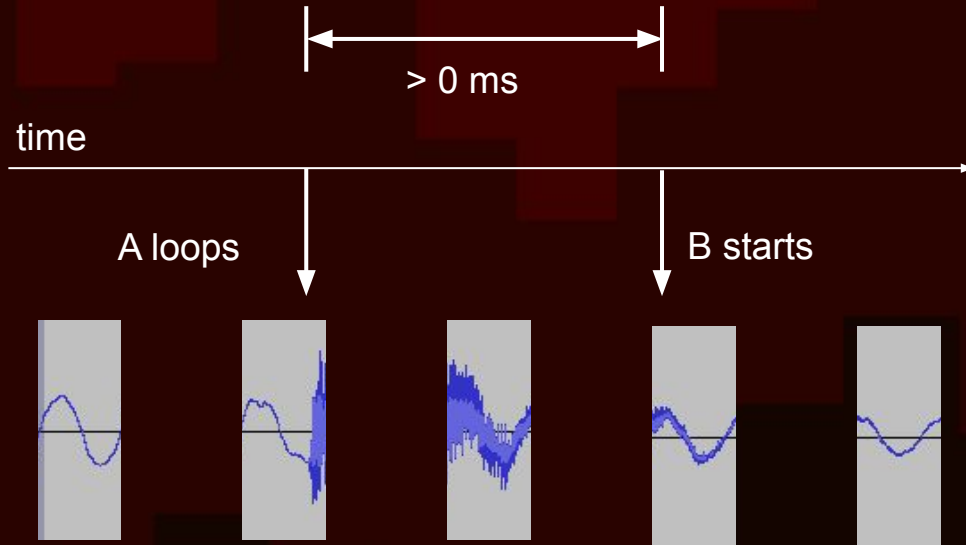
Audio Rendering

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



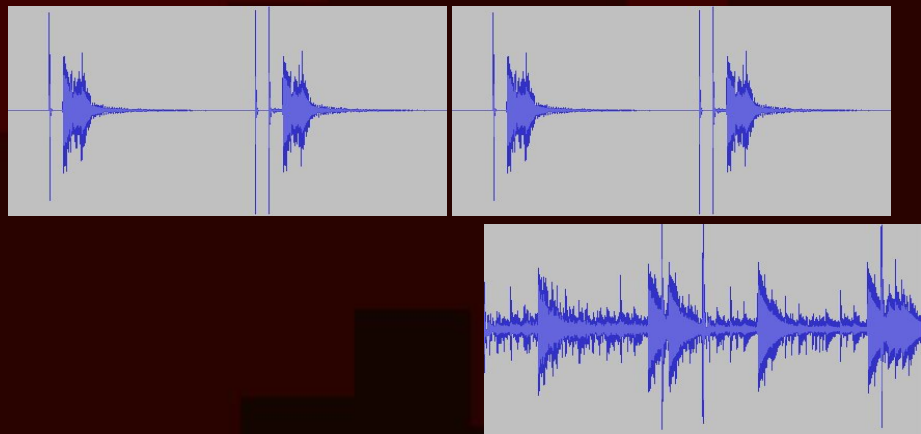
Immediately is Too Late

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!**

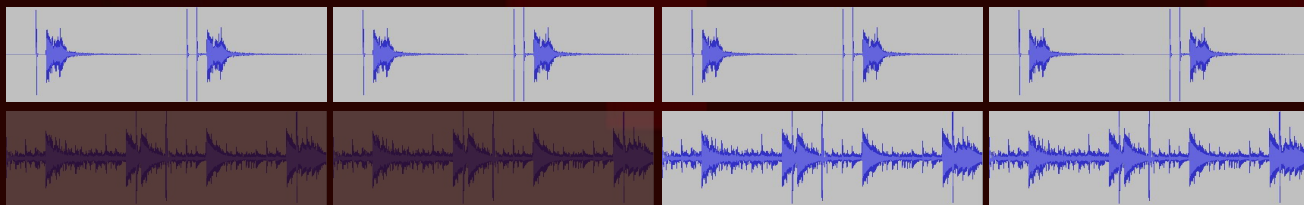


Interactive Music Solutions

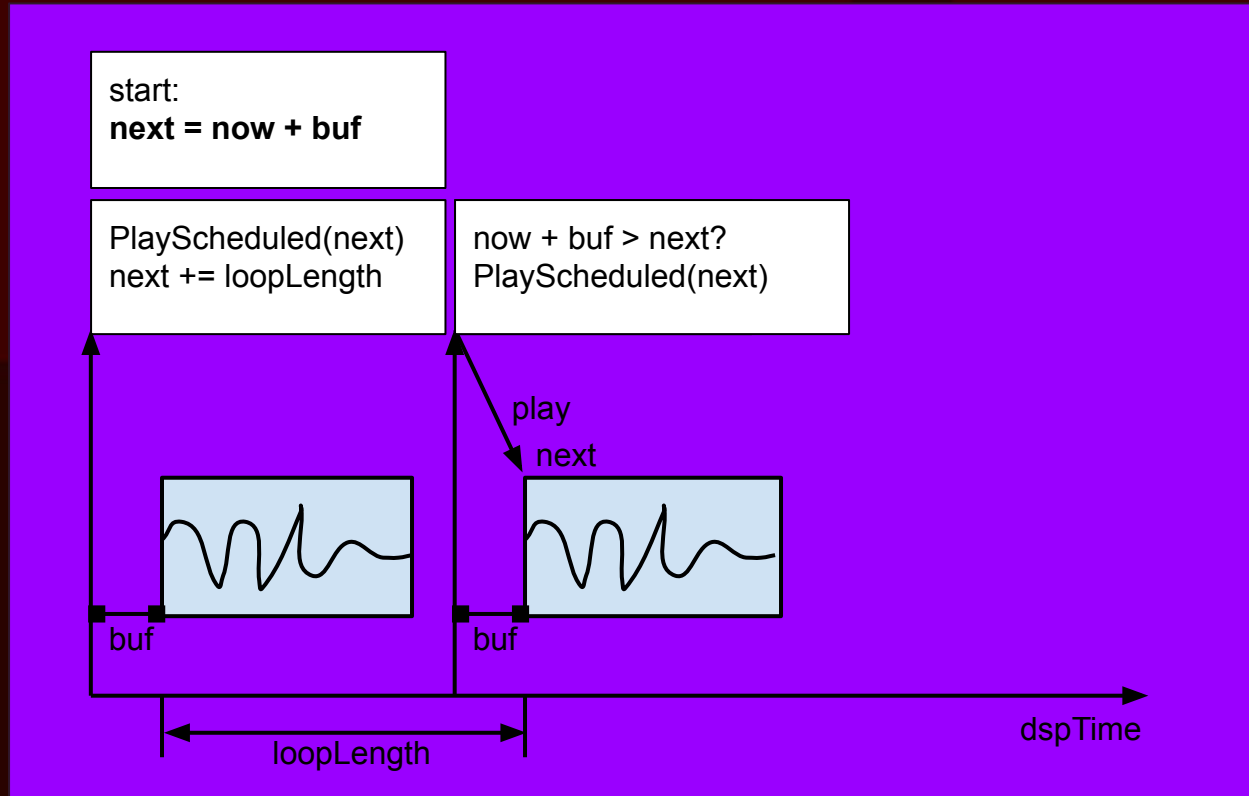
- Synchronized Loops
- PlayScheduled

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

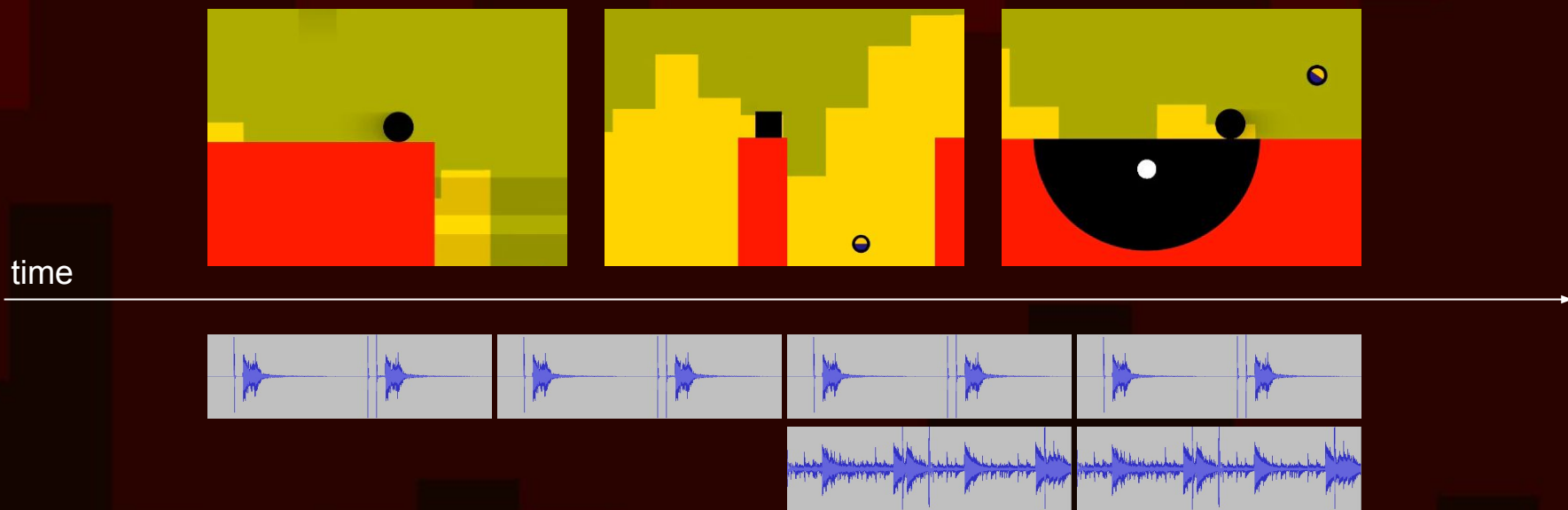
- Synchronizing loops with sample accuracy is tricky
- Audio is rendered in buffers, delaying and quantizing sounds
- Unity Update calls correspond to video frames, not audio buffers
- **Immediately is too late**: detecting loop and reacting in Update results in a delay
- Solution A: Synchronized Loops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



Music Timing in 140

Music Timing in 140

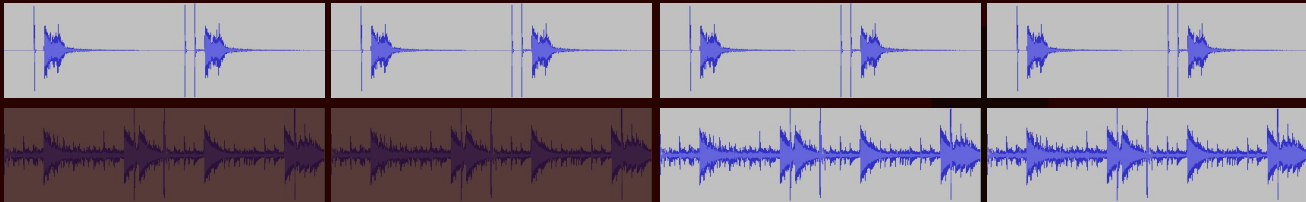
- We wanted the music to be mixed interactively with the gameplay.
- Loops should be sample-accurate.
- We were using Unity 3 at the time, which limited our options (no PlayScheduled)



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:

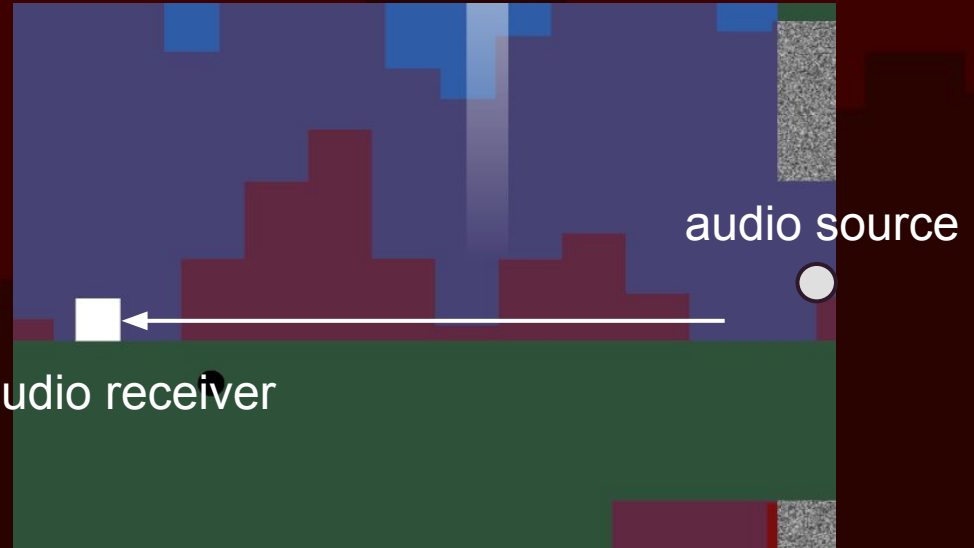
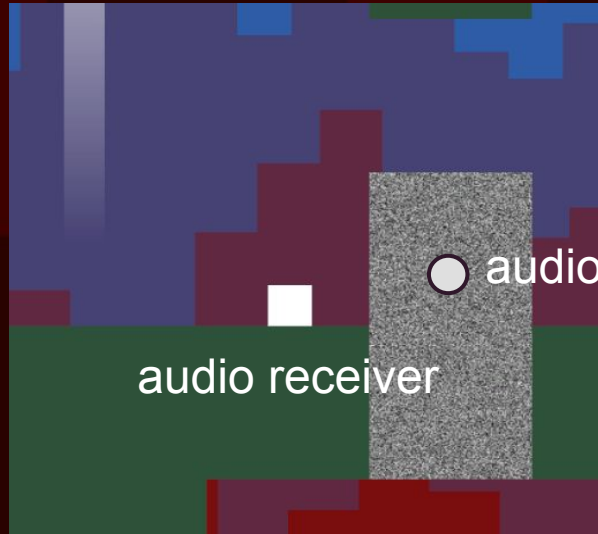
- Control volume/muting and pan.
- Never change pitch unless just before stopping a loop.

Localized Music Loops

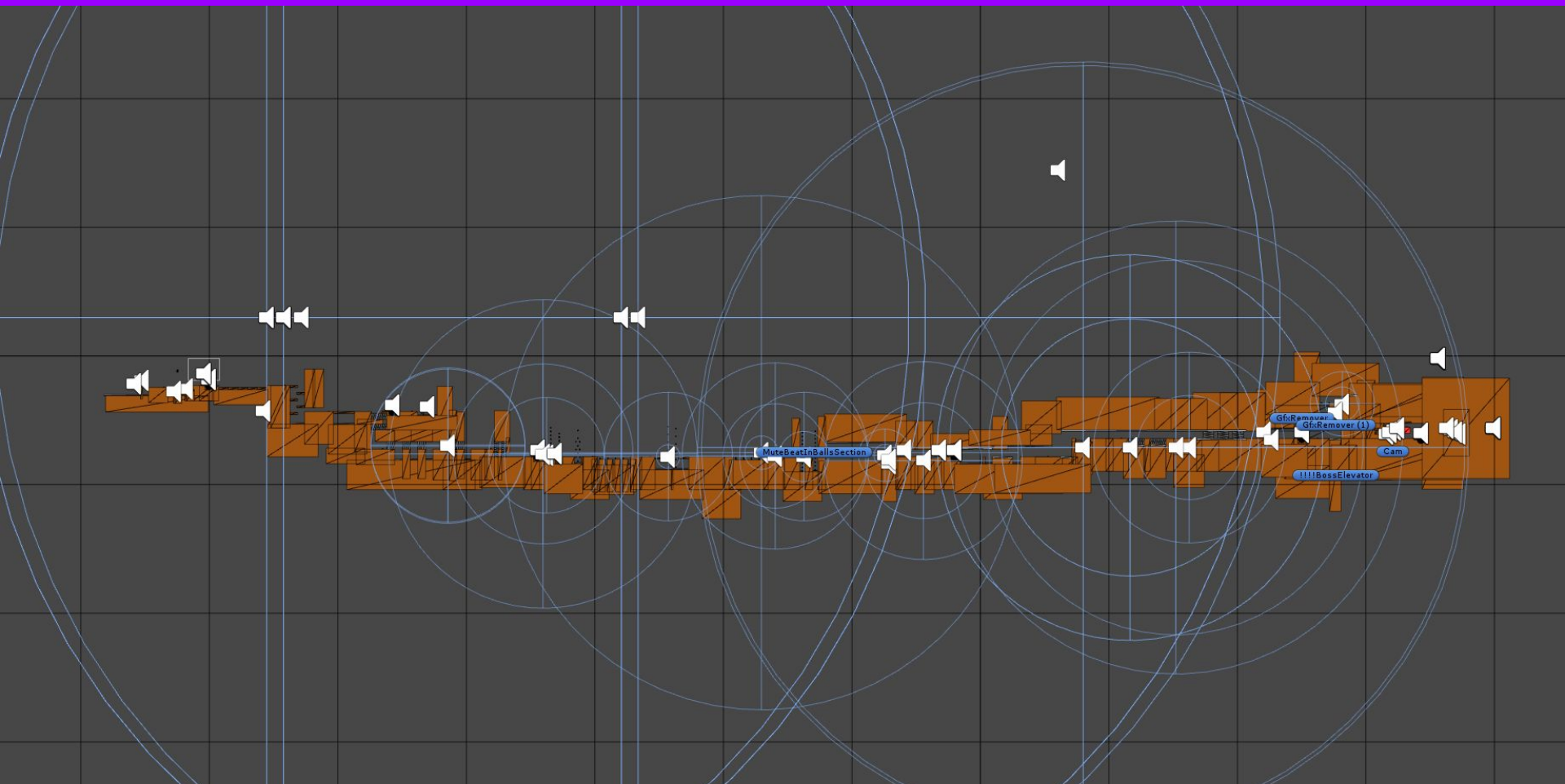
- Music loops are “physically” placed in level geometry
- Dynamic mixing occurs as player moves around
- Certain areas can gain unique atmosphere based on music

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

Audio Source

- AudioClip: l4-layer4-modnet_up
- Output: None (Audio Mixer Group)
- Mute:
- Bypass Effects:
- Bypass Listener Effects:
- Bypass Reverb Zones:
- Play On Awake:
- Loop:
- Priority: 0
- Volume: 1
- Pitch: 1
- Stereo Pan: 0
- Spatial Blend: 1
- Reverb Zone Mix: 0

3D Sound Settings

- Doppler Level: 0
- Spread: 180
- Volume Rolloff: Linear Rolloff
- Min Distance: 145.6949
- Max Distance: 176.1323

Graph showing parameter values over distance (0 to 150+):

Distance	Volume	Spatial	Spread	Low-Pass	Reverb
0	1.0	1.0	0.5	0.5	0.0
145.6949	1.0	1.0	0.5	0.5	0.0
176.1323	0.0	0.0	0.0	0.0	0.0

Example Audio Loop Components

AudioSync component: handles all loops

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

The screenshot displays four audio components in a game engine's inspector:

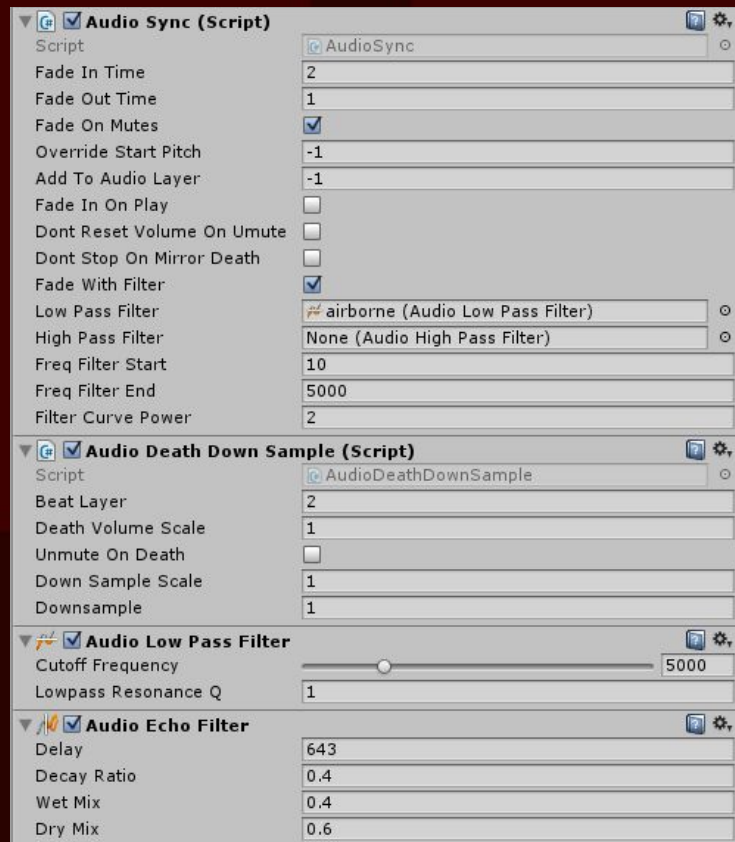
- Audio Sync (Script)**:
 - Script: AudioSync
 - Fade In Time: 2
 - Fade Out Time: 1
 - Fade On Mutes:
 - Override Start Pitch: -1
 - Add To Audio Layer: -1
 - Fade In On Play:
 - Dont Reset Volume On Umute:
 - Dont Stop On Mirror Death:
 - Fade With Filter:
 - Low Pass Filter: airborne (Audio Low Pass Filter)
 - High Pass Filter: None (Audio High Pass Filter)
 - Freq Filter Start: 10
 - Freq Filter End: 5000
 - Filter Curve Power: 2
- Audio Death Down Sample (Script)**:
 - Script: AudioDeathDownSample
 - Beat Layer: 2
 - Death Volume Scale: 1
 - Unmute On Death:
 - Down Sample Scale: 1
 - Downsample: 1
- Audio Low Pass Filter**:
 - Cutoff Frequency: 5000
 - Lowpass Resonance Q: 1
- Audio Echo Filter**:
 - Delay: 643
 - Decay Ratio: 0.4
 - Wet Mix: 0.4
 - Dry Mix: 0.6

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)



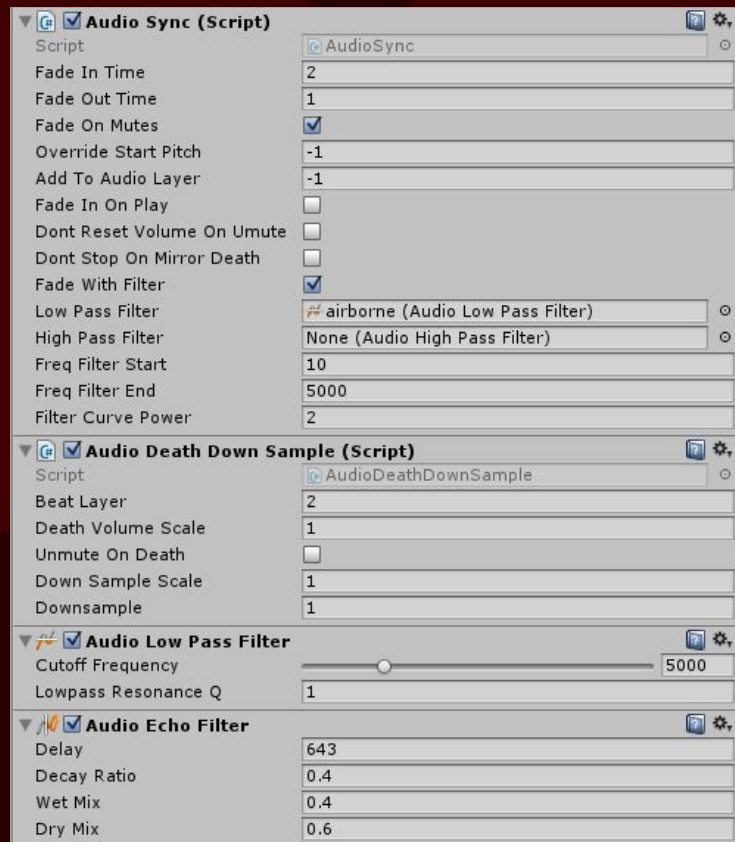
Example Audio Loop Components

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

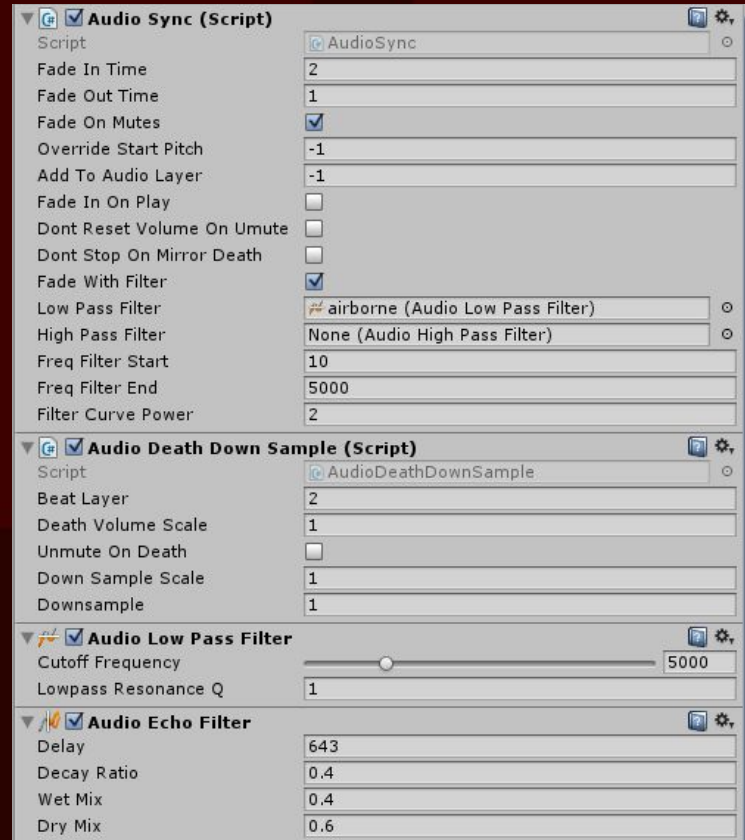
AudioSync component: handles all loops

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

AudioDeathDownSample:
downsample filter

AudioLowPassFilter: built-in Unity LPF

AudioEchoFilter: built-in Unity delay

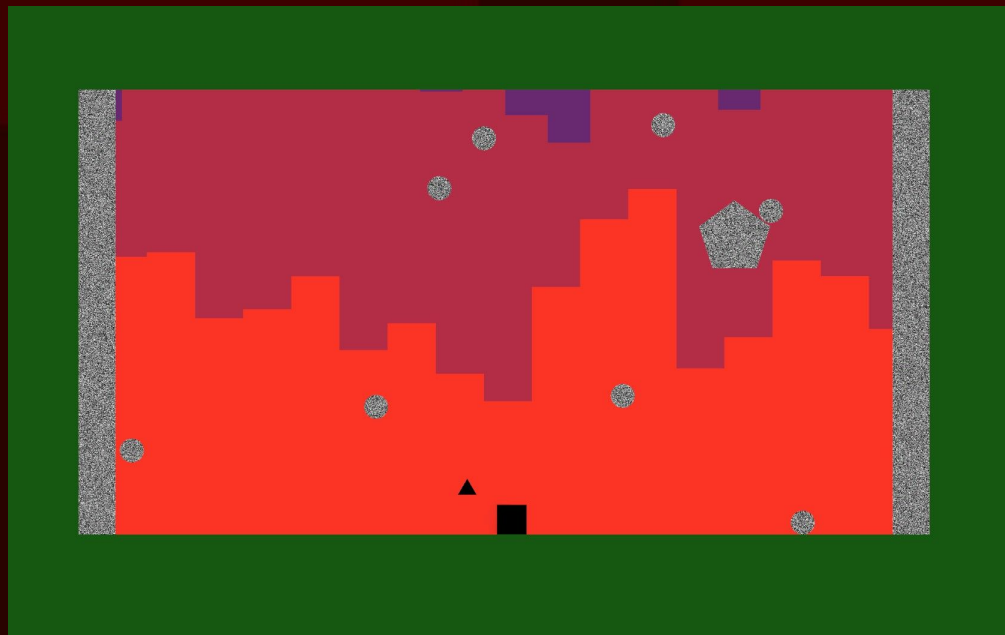
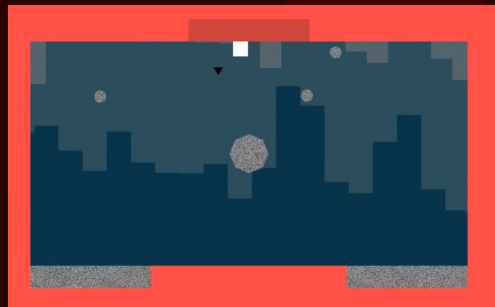


Masking Loops

Music loops can be masked:

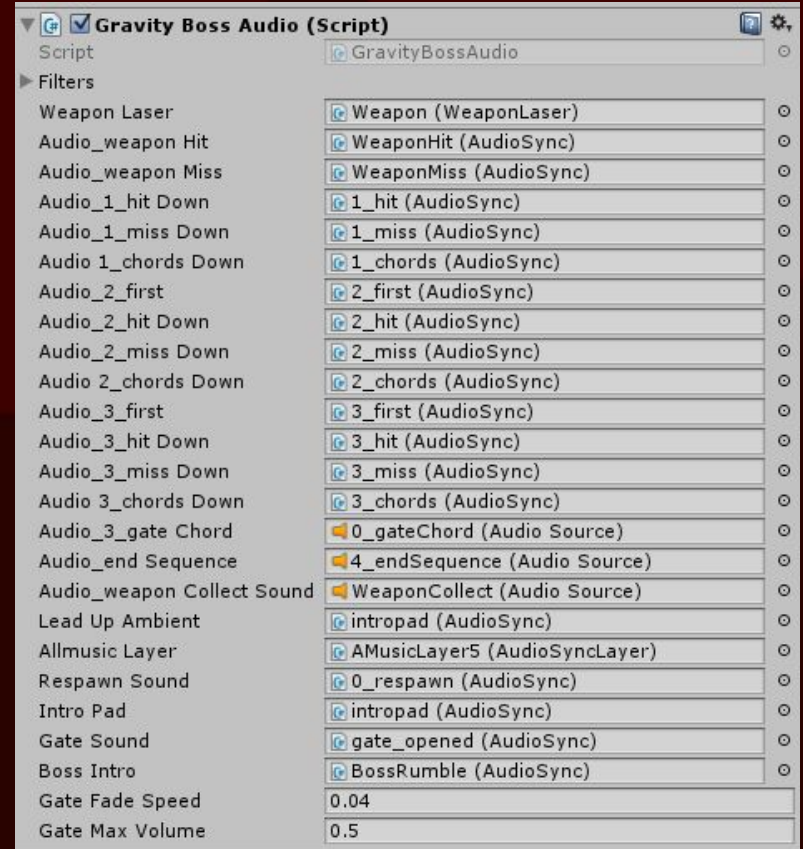
- Fade using filters
- Delay
- Unmuting at specific beats

Level 4 Boss



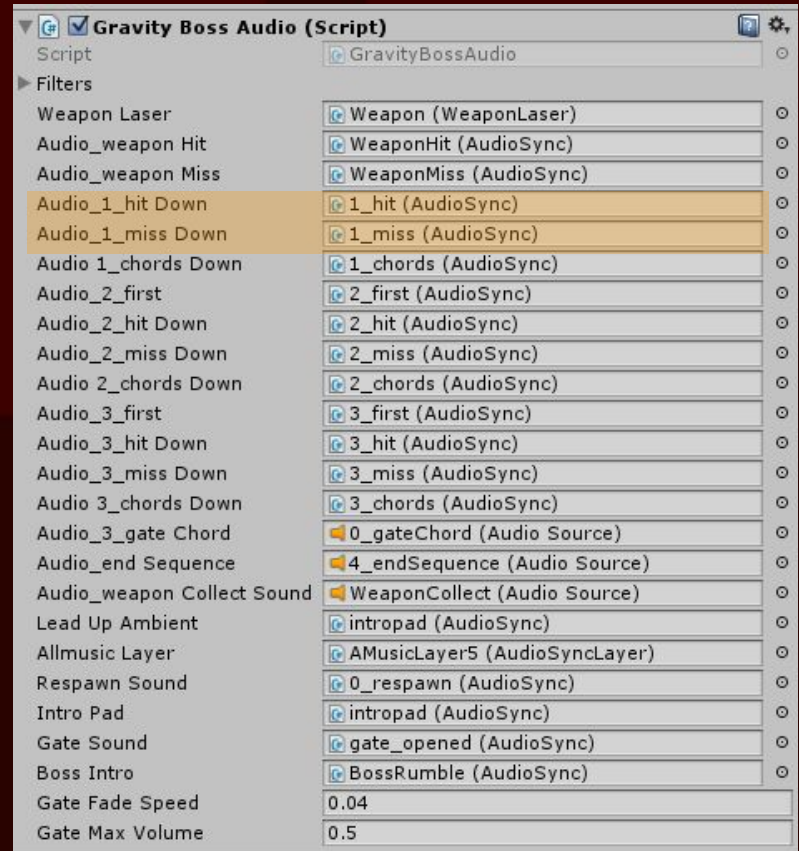
Level 4 Boss

- Over 20 loops running simultaneously



Level 4 Boss

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



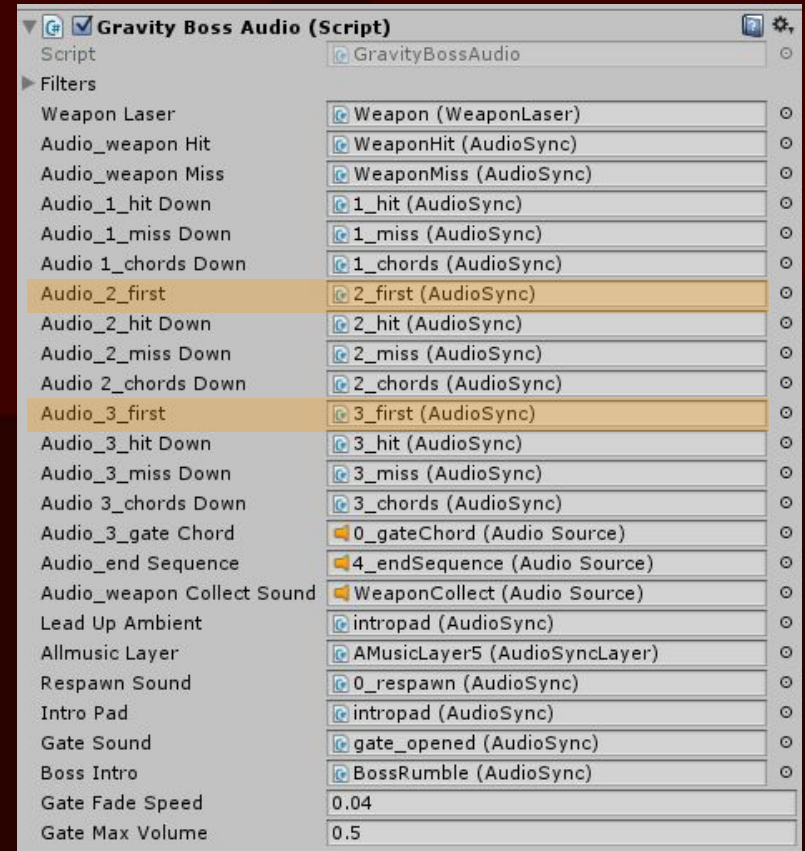
Level 4 Boss

- Each stage has its own set of loops



Level 4 Boss

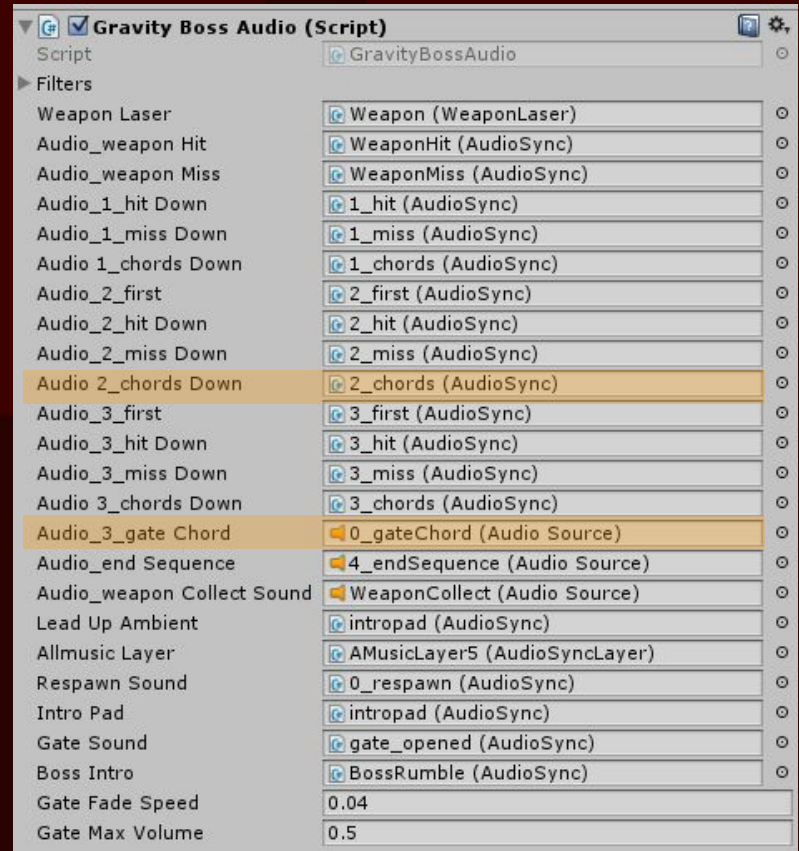
- Each stage is in a different key
- Loops have long Ableton Reverb tails
- Reverb tails and key change requires special transitional first loop



Event	Audio Source
Script	GravityBossAudio
Weapon Laser	Weapon (WeaponLaser)
Audio_weapon Hit	WeaponHit (AudioSync)
Audio_weapon Miss	WeaponMiss (AudioSync)
Audio_1_hit Down	1_hit (AudioSync)
Audio_1_miss Down	1_miss (AudioSync)
Audio_1_chords Down	1_chords (AudioSync)
Audio_2_first	2_first (AudioSync)
Audio_2_hit Down	2_hit (AudioSync)
Audio_2_miss Down	2_miss (AudioSync)
Audio_2_chords Down	2_chords (AudioSync)
Audio_3_first	3_first (AudioSync)
Audio_3_hit Down	3_hit (AudioSync)
Audio_3_miss Down	3_miss (AudioSync)
Audio_3_chords Down	3_chords (AudioSync)
Audio_3_gate Chord	0_gateChord (Audio Source)
Audio_end Sequence	4_endSequence (Audio Source)
Audio_weapon Collect Sound	WeaponCollect (Audio Source)
Lead Up Ambient	intropad (AudioSync)
Allmusic Layer	AMusicLayer5 (AudioSyncLayer)
Respawn Sound	0_respawn (AudioSync)
Intro Pad	intropad (AudioSync)
Gate Sound	gate_opened (AudioSync)
Boss Intro	BossRumble (AudioSync)
Gate Fade Speed	0.04
Gate Max Volume	0.5

Level 4 Boss

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



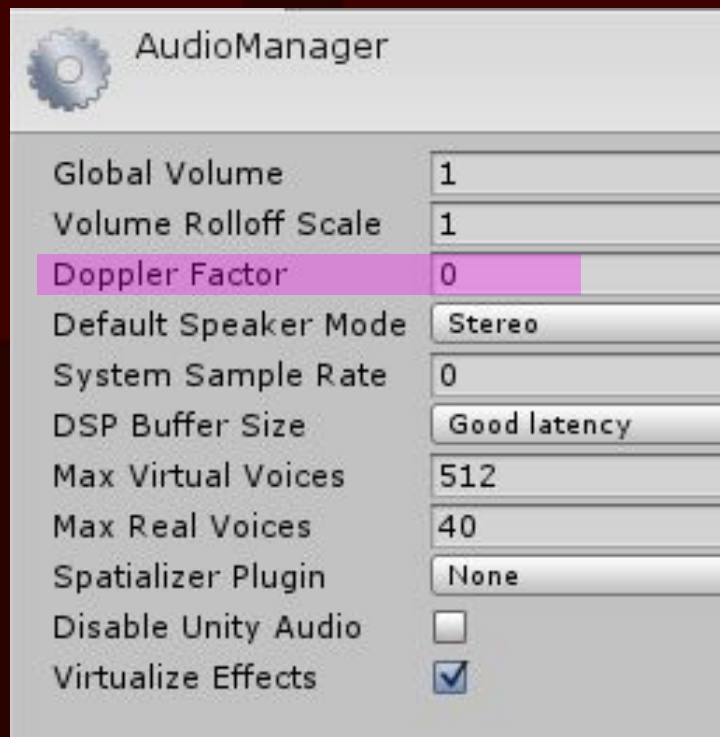
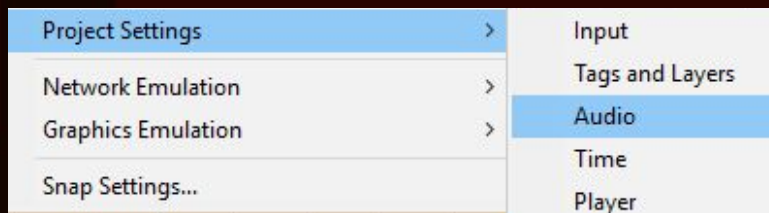
140 demo

level 4 boss



Doppler Effect

- Disable Doppler effect to avoid drifting loops!

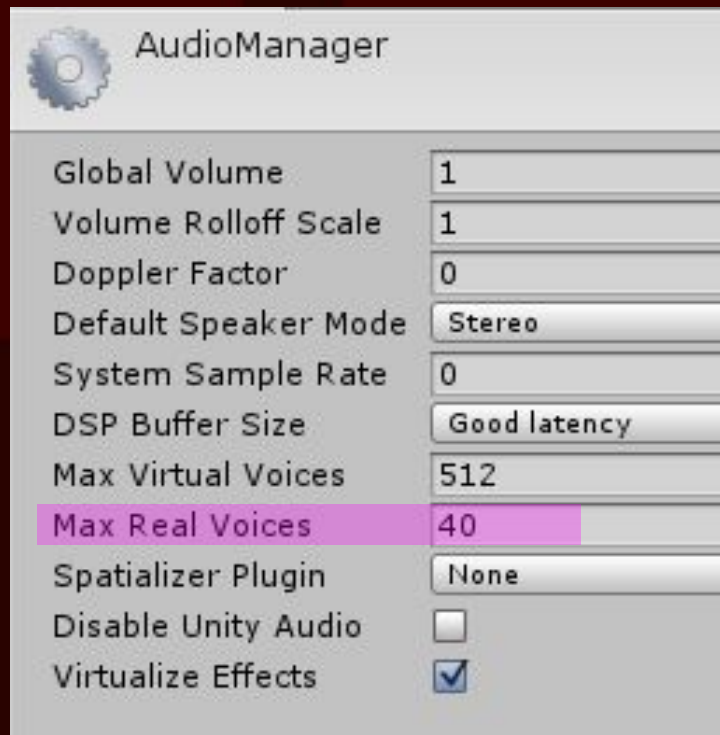


Max Real Voices

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

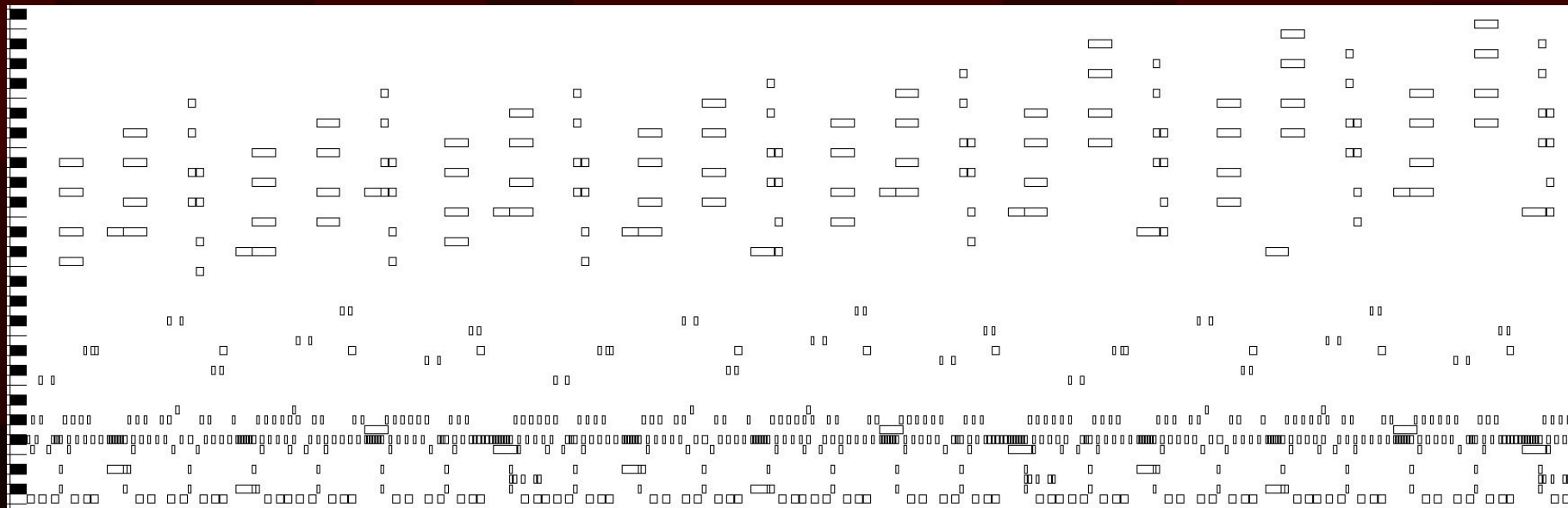


140 Music Production

The image displays a professional digital audio workstation (DAW) interface, likely Ableton Live, used for music production. The main workspace is a piano roll showing a complex arrangement of tracks. The tracks are organized into several sections: **MUSIC** (including various layer and beat tracks), **SOUND EFFECTS** (including key_pickup), and **INSTRUMENTS** (including 28 Instrument Rac, gate, larve vari, aman_brot, larve, and Master). The piano roll shows a dense arrangement of notes and triggers across 32 channels, with various MIDI notes and automation curves visible. The bottom of the interface features several plugin windows: **I4-bass R** (a synthesizer with a filter and envelope), **Audio Effect Rack** (containing a **Waveshaper** plugin), **Saturator** (with a drive and depth control), **Dynamic Tube** (with a compressor-style envelope), and **EQ Eight** (with a frequency response curve). The top of the interface shows transport controls, tempo (140.00), and various utility buttons.

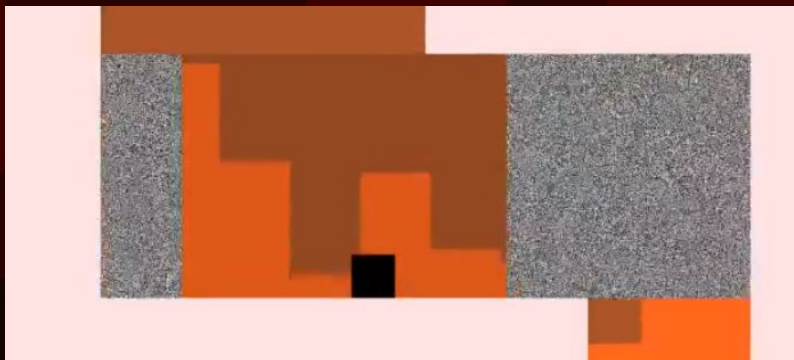
The Puzzle of Music

Making music for a *music game* can be like solving a puzzle

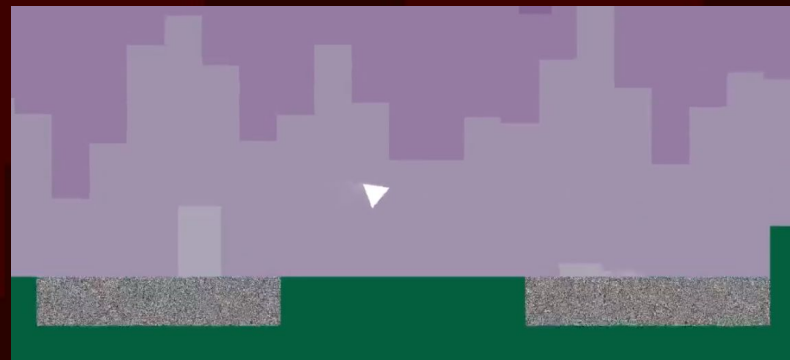


KillBlocks and Toggler

Two example game mechanics

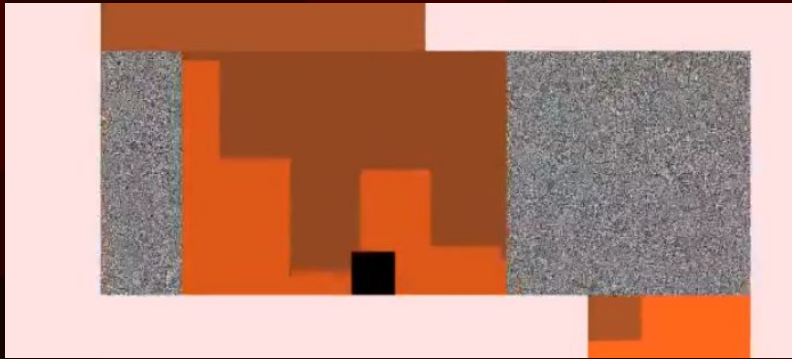


KillBlock



Toggler

KillBlocks

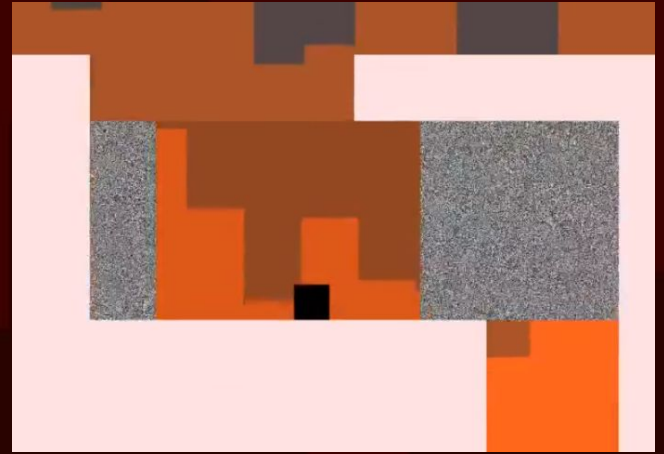


KillBlock



KillBlock Rhythm Pattern

- Communicates to player exactly when certain game areas are either safe or lethal
- Must correspond exactly to game logic timing



time 

> > > > x

> MOVE blocks right

x TOGGLE all blocks

KillBlock Rhythm Pattern

- Music runs at 140 beats/minute
- A 'bar' is 4 beats ~ 1.7 seconds
- Our KillBlock rhythm is exactly 2 bars



time (bars) \longrightarrow

1 . . . : . . . 2 . . . : . . .

> > > > x

KillBlock Sounds

- > MOVE sound is a 'Landlord stab'
- x TOGGLE sound is a 808 cowbell

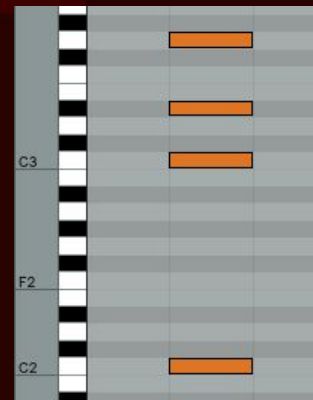


> MOVE Sound

- 'Landlord' stab
- Classic house sample
- Sampled minor chord played on piano-like FM synth
- Origin of sample seems to be 1984 Linndrum demo tape
- Made famous by Landlord's 'I Like It (blow out dub)' (1989)



→ sounds like




Sampled Chords

- Sampled chord is played back at different sample rates
 - Resulting output is the same chord with new base notes
- (foundational for all sampler-based music)



KillBlock Harmony

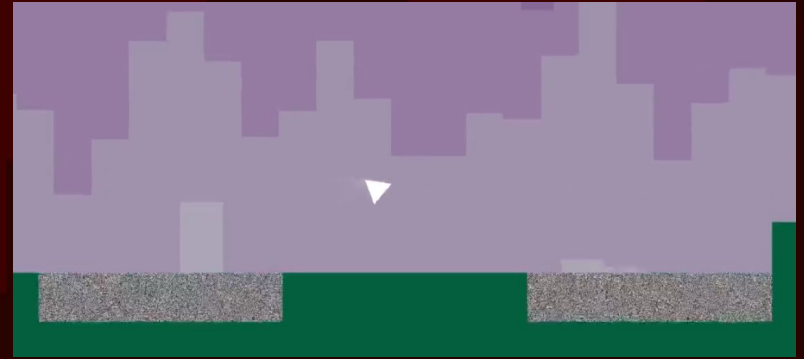
- The result can be heard in the **KillBlock loop** 
- 2-bar rhythmic loop
- 8-bar harmonic loop

bars

1 . : . 2 . : . 3 . : . 4 . : . 5 . : . 6 . : . 7 . : . 8 . : .
Cm D#m F#m C#m Em Gm Dm Fm



KillBlocks and Toggles



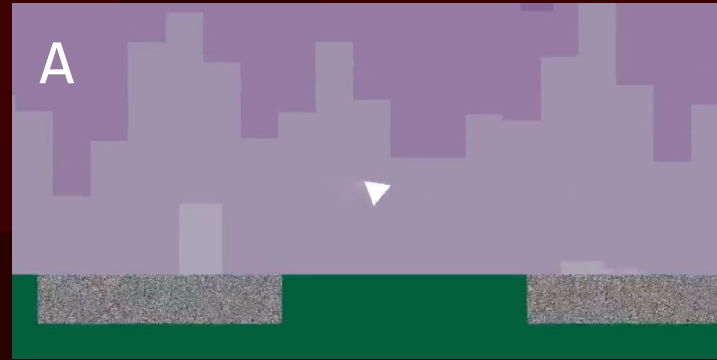
Toggler

Toggler Sound

The toggler loop alternates between two different Operator patches

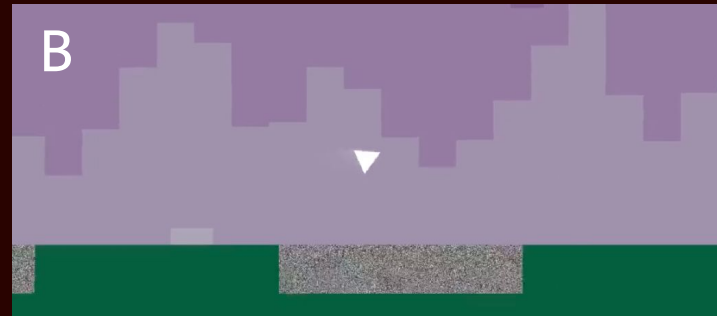
State A Sound

The screenshot shows the State A Operator patch interface. The left panel contains four sliders for Coarse, Fine, Fixed, and Level, with values: Coarse 3, Fine 0, Fixed -12 dB, Level -12 dB; Coarse 2, Fine 0, Fixed -14 dB, Level -14 dB; Coarse 1, Fine 0, Fixed -7.6 dB, Level -7.6 dB; Coarse 1, Fine 0, Fixed -12 dB, Level -12 dB. The middle panel shows the Envelope graph with parameters: Attack 0.00 ms, Decay 1.77 s, Release 1.37 s, Time<Vel 0%, Wave Sin. The right panel shows LFO (Sine, Rate 111.88, Amount 32%), Filter (12, 24, Clean, Freq 18.5 kHz, Res 20%), Pitch Env (0.0%), Spread (0%), Transpose (0 st), Time (29%), Tone (70%), and Volume (-2.3 dB).



State B Sound

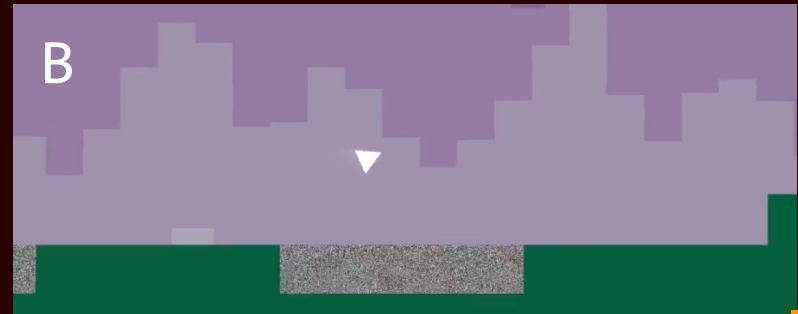
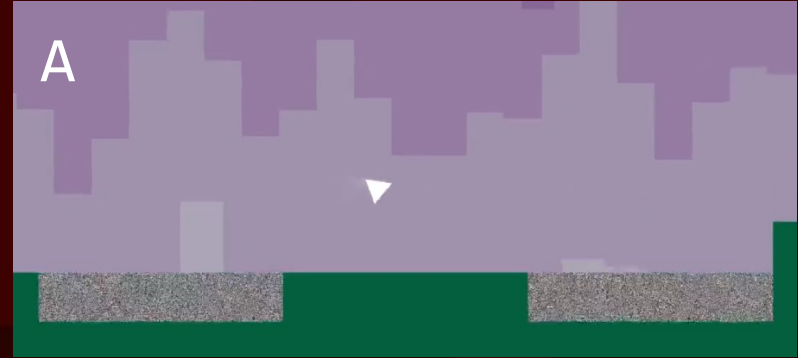
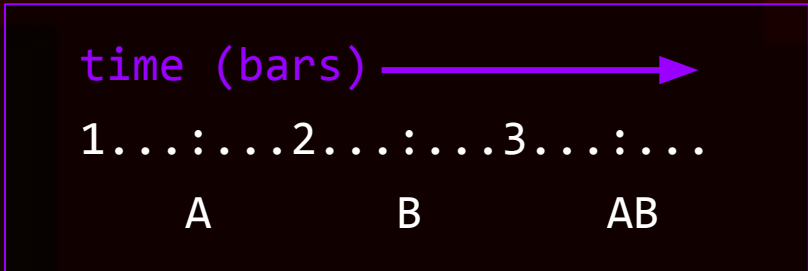
The screenshot shows the State B Operator patch interface. The left panel contains four sliders for Freq, Multi, Fixed, and Level, with values: Freq 186 Hz, Multi 0.01, Fixed -58 dB, Level -58 dB; Coarse 5, Fine 0, Fixed -7.0 dB, Level -7.0 dB; Coarse 1, Fine 0, Fixed -8.1 dB, Level -8.1 dB; Coarse 1, Fine 0, Fixed 0.0 dB, Level 0.0 dB. The middle panel shows the Envelope graph with parameters: Attack 0.00 ms, Decay 600 ms, Release 6.42 s, Time<Vel 0%, Wave Sin. The right panel shows LFO (Sine, Rate 96.76, Amount 52%), Filter (12, 24, OSR, Freq 55.3 Hz, Res 37%), Pitch Env (0.0%), Spread (0%), Transpose (0 st), Time (0%), Tone (70%), and Volume (-13 dB).



Toggler Rhythm Pattern

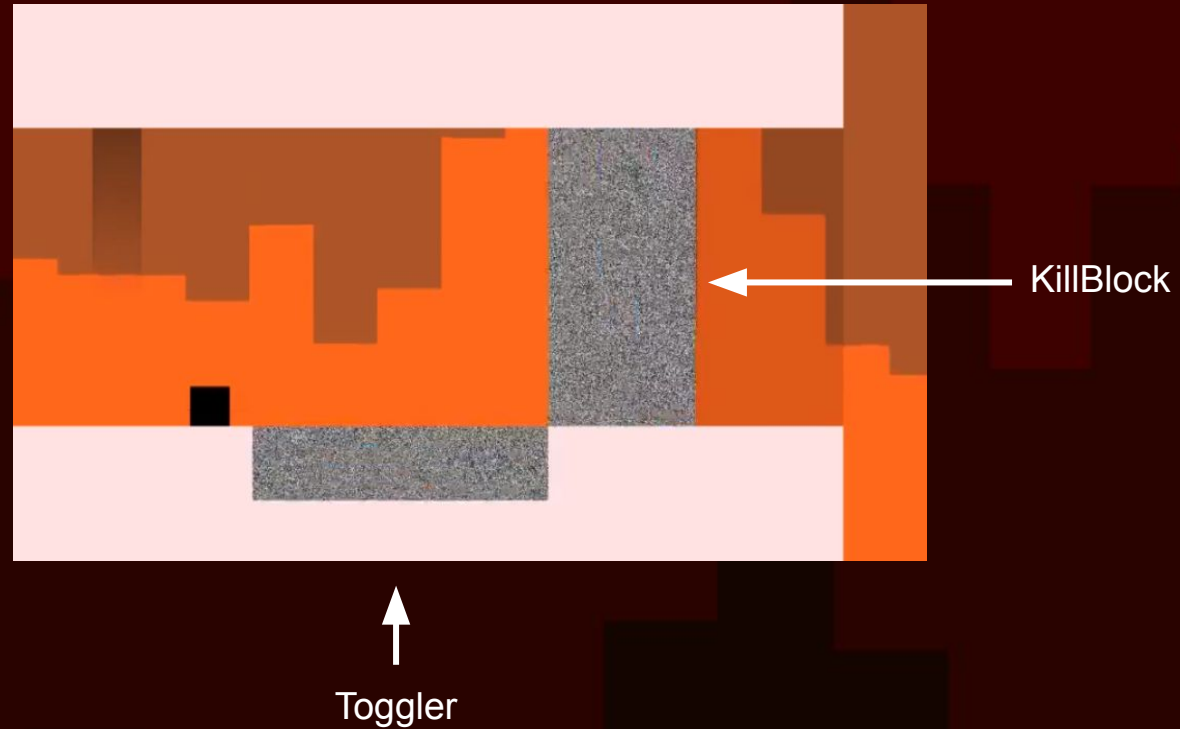
The toggler loop: 

- 3-bar rhythmic loop
- Game logic toggle floors between lethal and non-lethal
- Two states: A and B



Toggler and KillBlock

Both play at once in this jump puzzle!



Toggler and KillBlock Rhythms

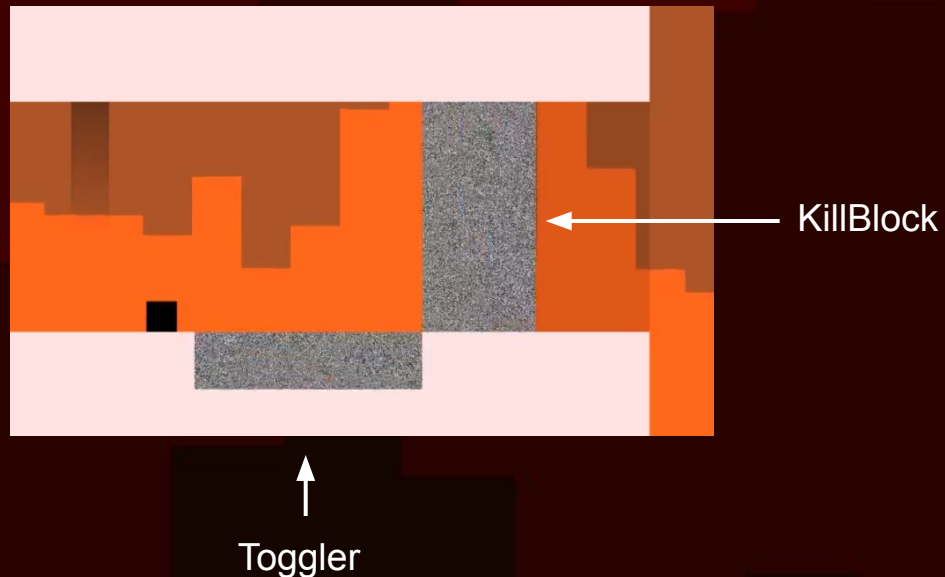
- KillBlock loop is 2 bars
- Toggler loop is 3 bars

KillBlock

```
1 . . . : . . . 2 . . . : . . .  
  > >      > > x
```

Toggler

```
1 . . . : . . . 2 . . . : . . . 3 . . . : . . .  
          A          B          A B
```



Toggler and KillBlock Looped

Loop simultaneously after $2 \times 3 = 6$ bars

KillBlock x 3

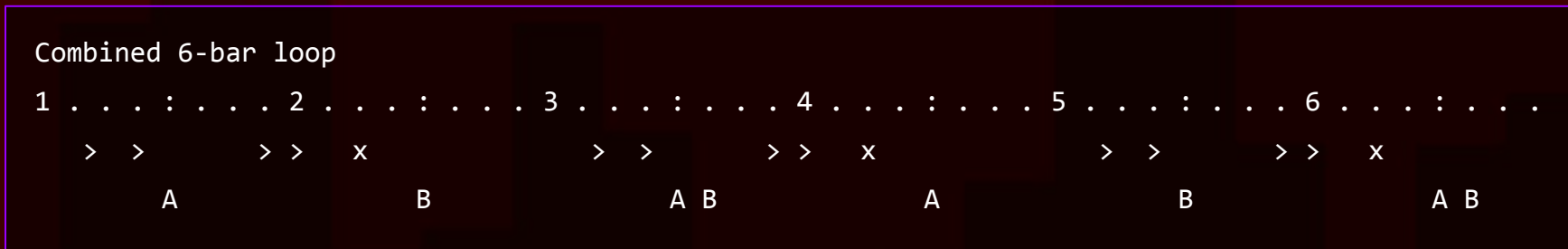
```
1 . . . : . . . 2 . . . : . . . 3 . . . : . . . 4 . . . : . . . 5 . . . : . . . 6 . . . : . . .  
  > >      > > x          | > >      > > x          | > >      > > x  
                    loop                    loop
```

Toggler x 2

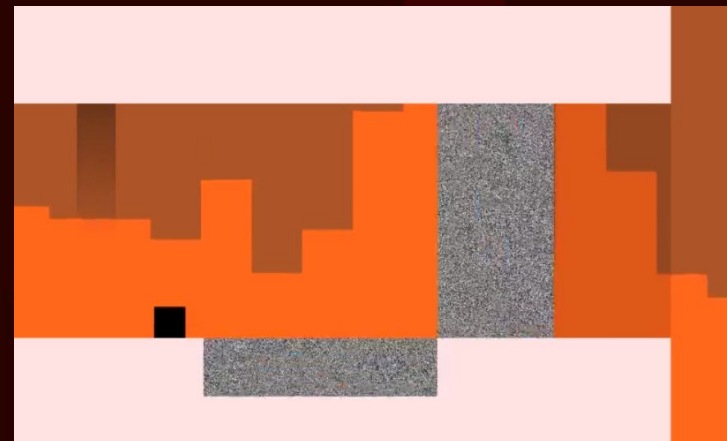
```
1 . . . : . . . 2 . . . : . . . 3 . . . : . . . 4 . . . : . . . 5 . . . : . . . 6 . . . : . . .  
          A          B          A B          |          A          B          A B  
                    loop
```

Toggler and KillBlock Combined

The combined 6-bar loop of **Toggler and KillBlock**: 



This pattern is what the player must grasp
to pass the jump puzzle



Toggler Harmony

- Toggler loop must be in harmony with KillBlock loop
- 8-bar harmonic loop

KillBlock:	Cm	D#m	F#m	C#m	Em	Gm	Dm	Fm
Toggler:	Cm7	D#m7	Bm11	C#m7	Em7	Cm11	Dm7	Fm7

Toggler Full Loop

- 3-bar rhythmic loop
- 8-bar harmonic loop
- Full loop: $3 \times 8 = 24$ bars

1	...	2	...	3	...	4	...	5	...	6	...	7	...	8	...	9	...	10	...	11	...	12	...
A		B		AB		A		B		AB		A		B		AB		A		B		AB	
Cm7		D#m7		Bm11		C#m7		Em7		Cm11		Dm7		Fm7		Cm11		D#m7		F#m7		C#m11	
13	...	14	...	15	...	16	...	17	...	18	...	19	...	20	...	21	...	22	...	23	...	24	...
A		B		AB		A		B		AB		A		B		AB		A		B		AB	
Em7		Gm7		Dm11		Fm7		Cm7		D#m11		F#m7		C#m7		Em11		Gm7		Dm7		Fm11	

Toggler Full Loop

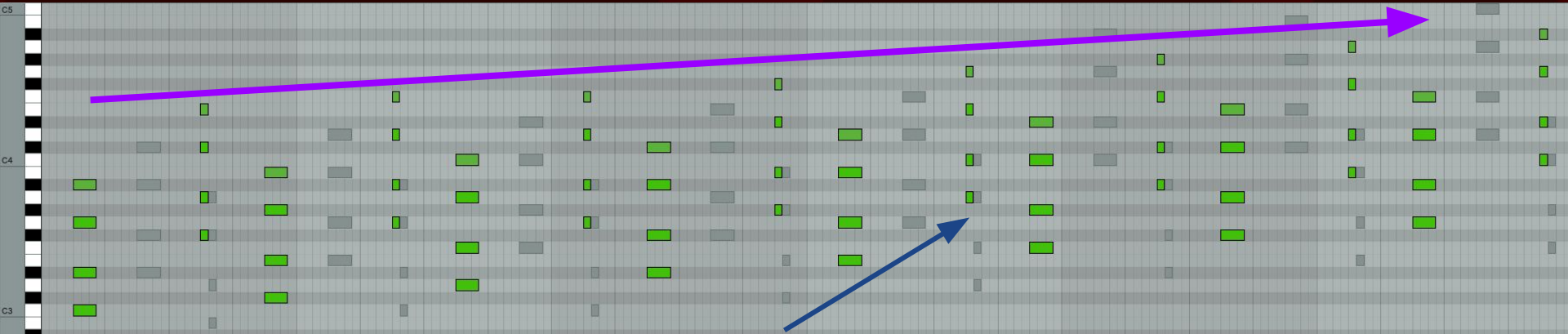
1	2	3	4	5	6	7	8	9	10	11	12	
A	B	AB	A	B	AB	A	B	AB	A	B	AB	
Cm7	D#m7	Bm11	C#m7	Em7	Cm11	Dm7	Fm7	Cm11	D#m7	F#m7	C#m11	
13	14	15	16	17	18	19	20	21	22	23	24	
A	B	AB	A	B	AB	A	B	AB	A	B	AB	
Em7	Gm7	Dm11	Fm7	Cm7	D#m11	F#m7	C#m7	Em11	Gm7	Dm7	Fm11	



Rising Pattern

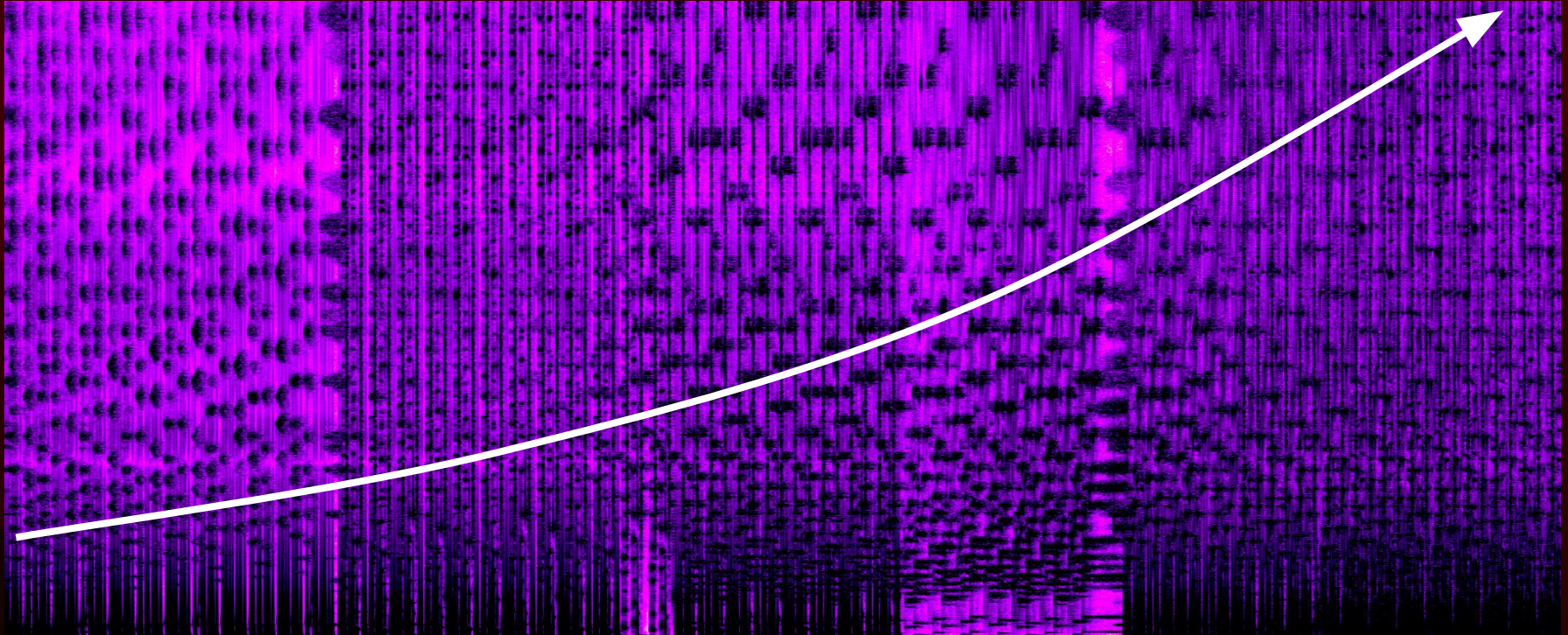
Level 4 is composed to emulate frequency continuously rising:

- Uses chord inversions to create 4-chord rising sequences
- Chord notes generally ascend over full 24-bar loop



Rising Pattern

Spectral analysis of soundtrack version shows rising frequency pattern



Fun Audio Tricks

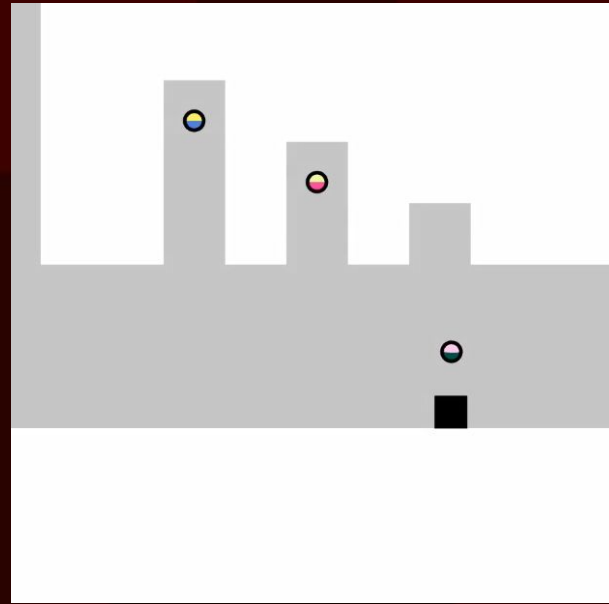
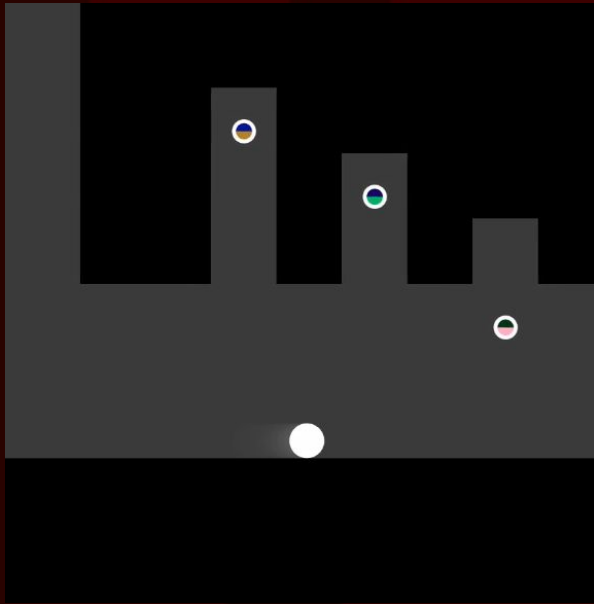
The background is a pixelated, low-resolution scene. The top half is a bright red sky. The bottom half is a brown ground. A purple path starts from the bottom left, goes right, then up, then right again, leading to a white door. There are some white and light purple pixels scattered around the door area.

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:

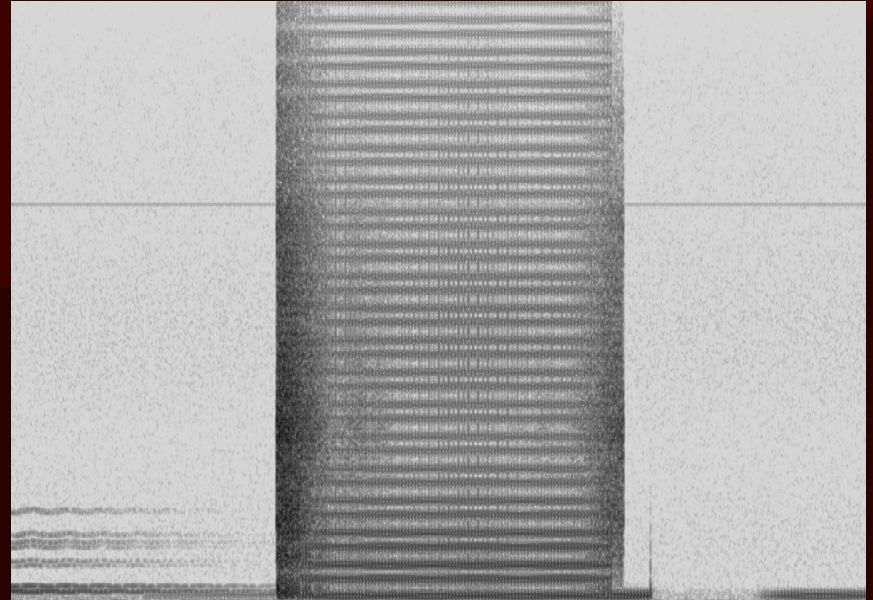
- applying strong vibrato to all music tracks,
- enveloping pitch towards 0 and volume towards silence.

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
```

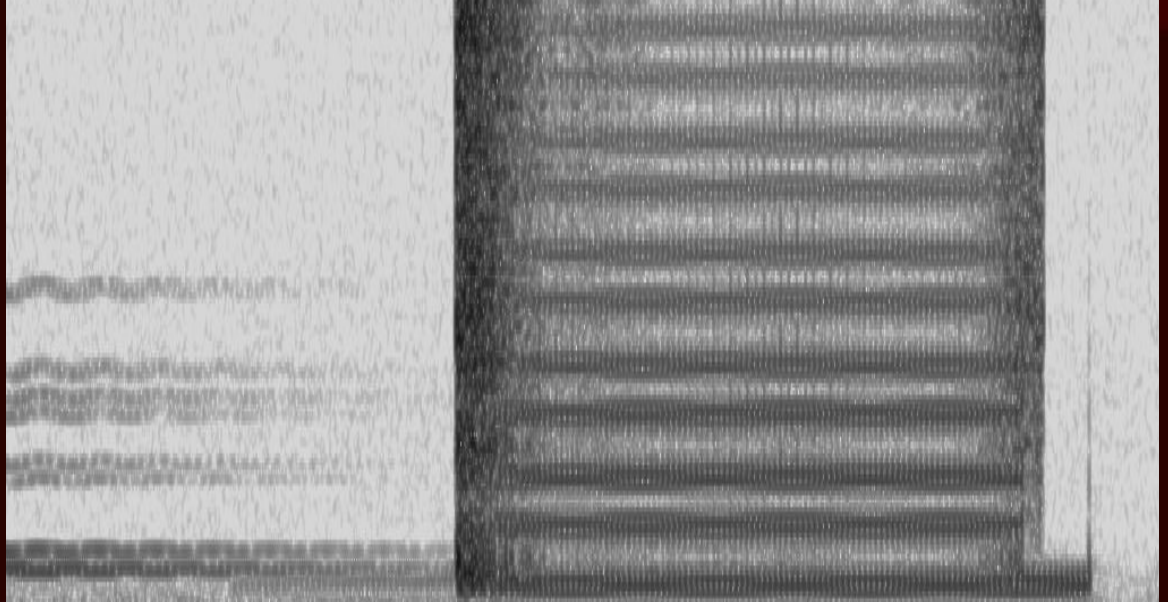
Downsampling

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.



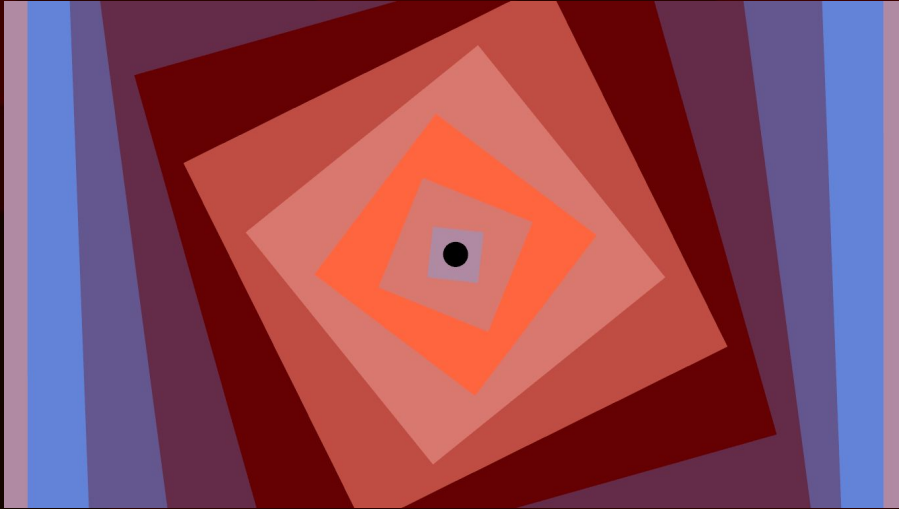
Downsampling Filter Code

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  
        }  
    }  
}
```

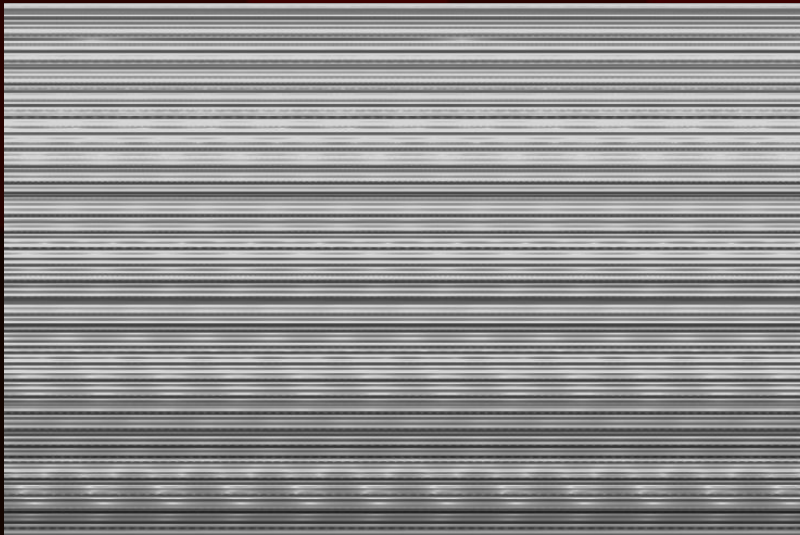
Fake crash

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.



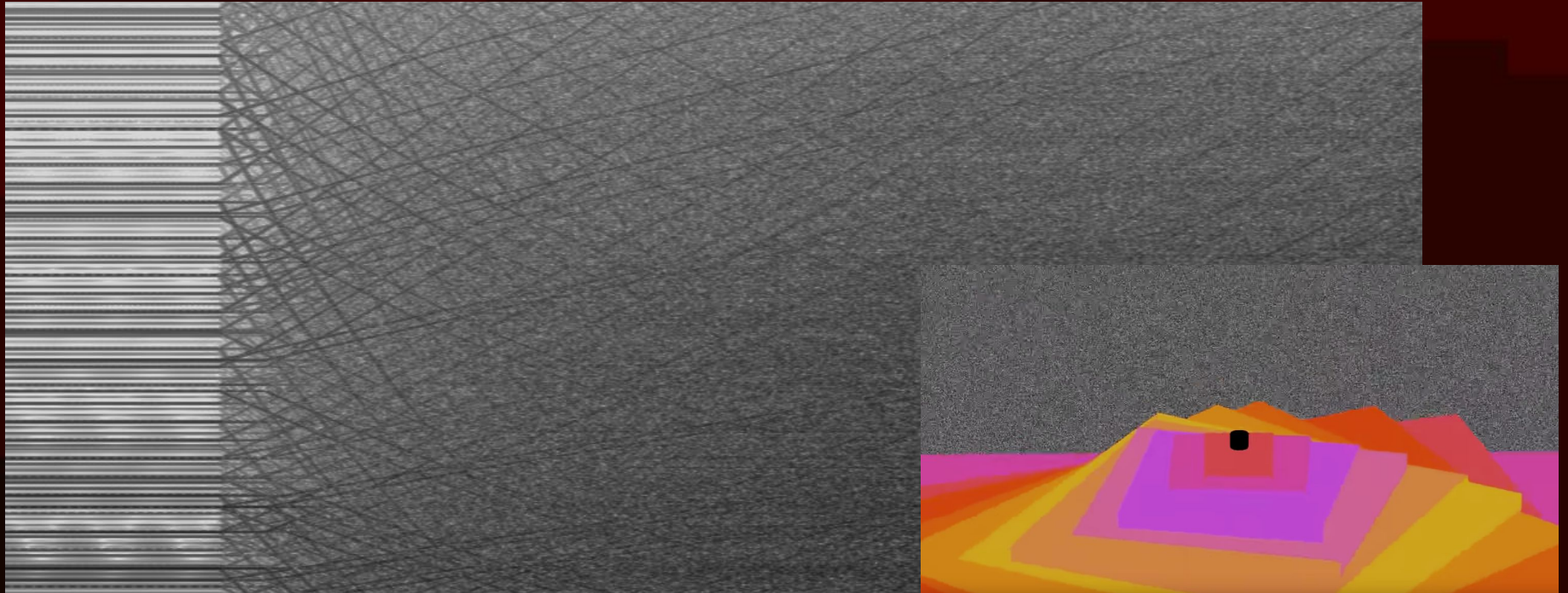
Fake crash

- 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?



