Game Audio Programming in C++

Guy Somberg

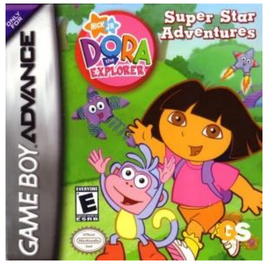
Echtra Games, Inc.

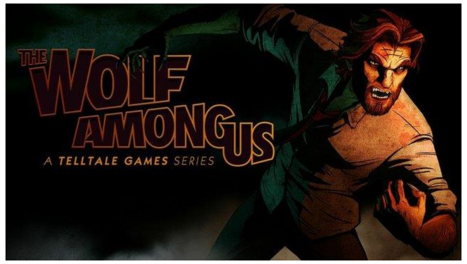
Who Am I?

- In Games Since 2002
- Owned the audio engine at (nearly) every company

Who Am I?

- In Games Since 2002
- …and shipped lots of games

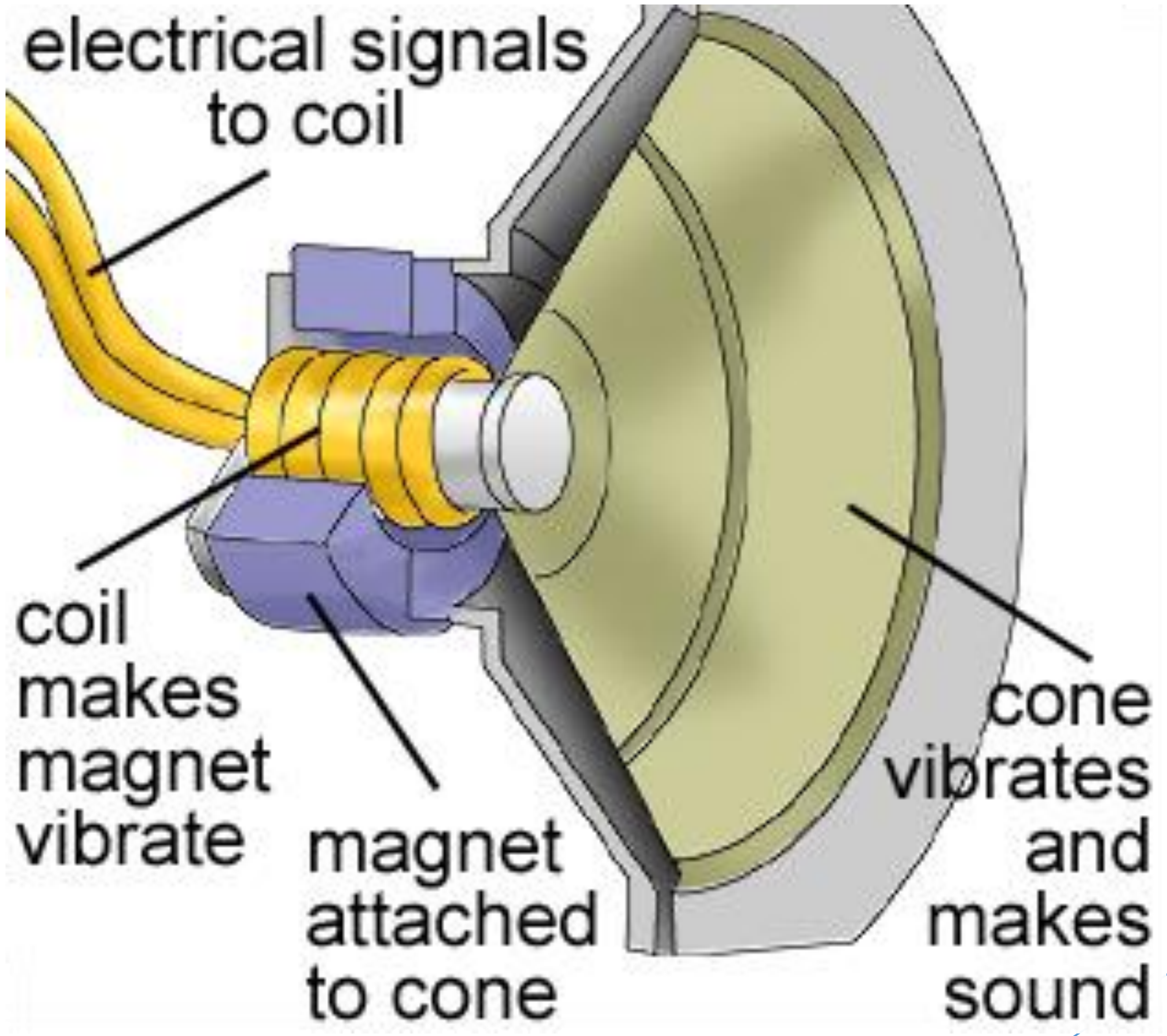




Order of Operations

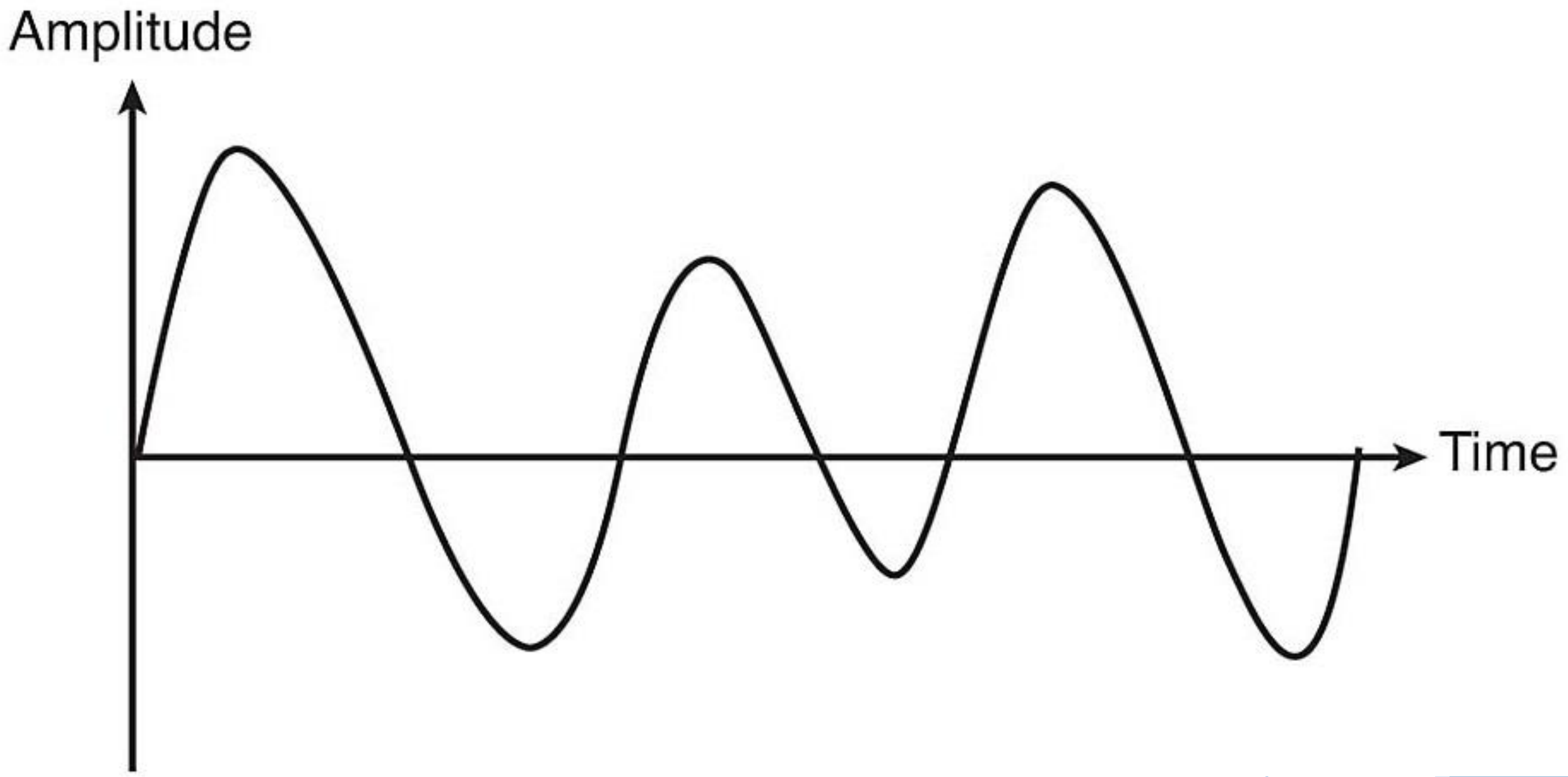
- Audio Fundamentals
  - An “as if” model of what’s actually happening
- Game Audio Programming
  - The current state of the art
- Toward a standard C++ audio library

Audio Fundamentals: Speakers


electrical signals to coil
coil makes magnet vibrate
magnet attached to cone
cone vibrates and makes sound

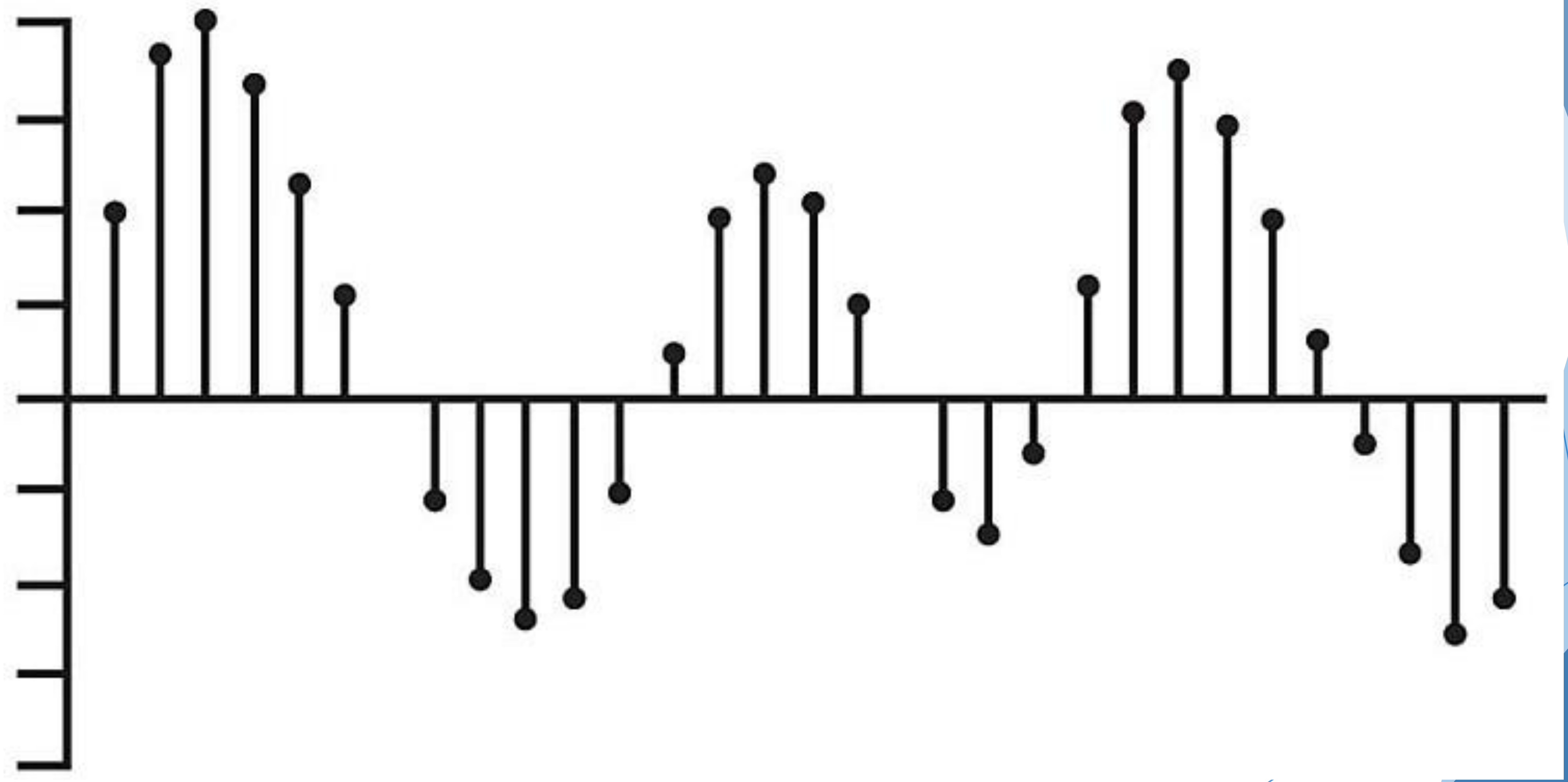
http://explorephysics.edublogs.org/2010/09/15/speakers-and-physics/

Audio Fundamentals: Waveforms



http://www.networkworld.com/subnets/cisco/chapters/1587052695/graphics/04fig01.jpg

Audio Fundamentals: Pulse Code Modulation



Great, So We Can Play One Sound

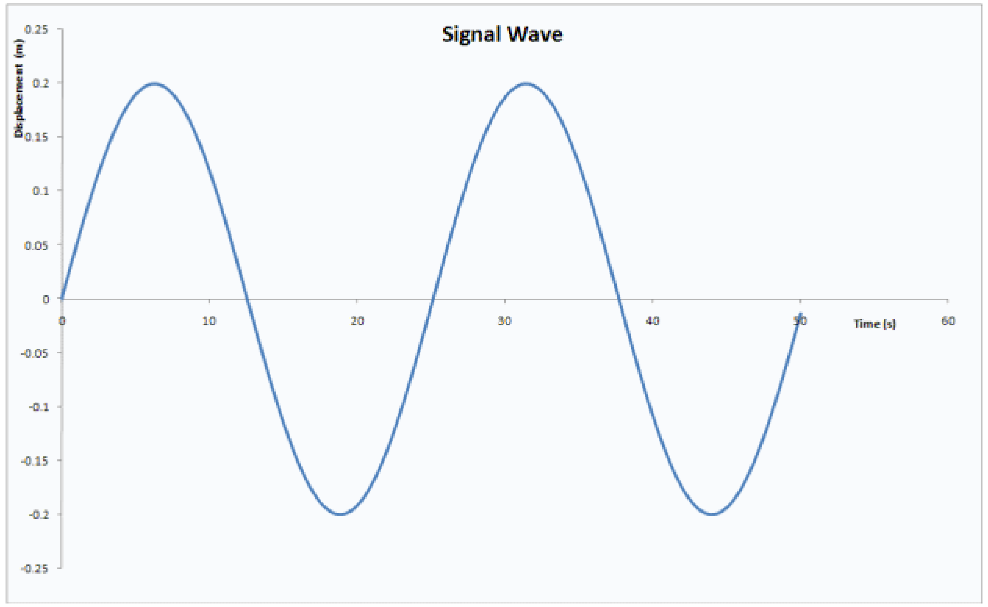
Great, So We Can Play One Sound

- Big deal

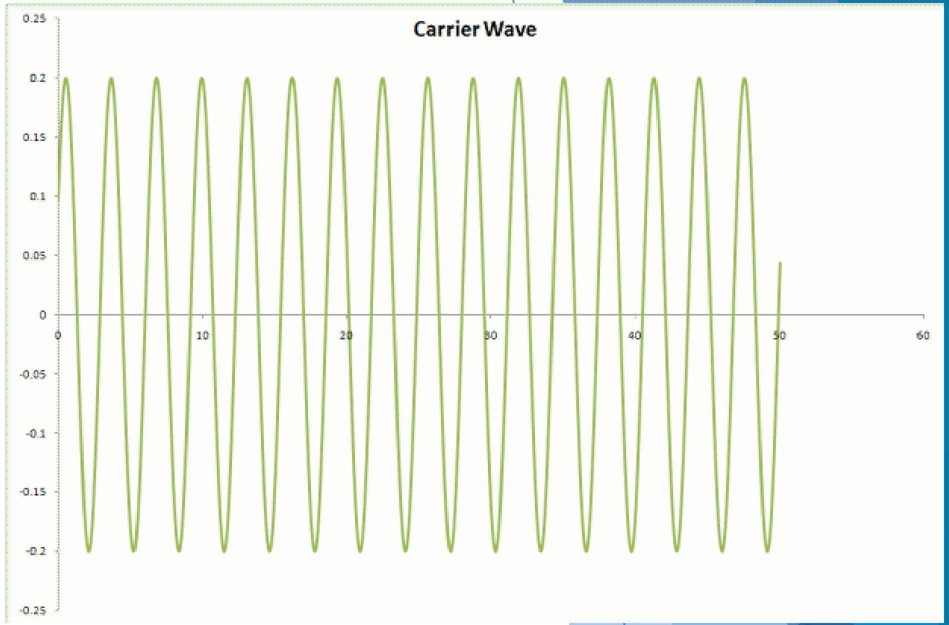
Great, So We Can Play One Sound

- Big deal
- Let’s play more than one

Playing Two Sounds



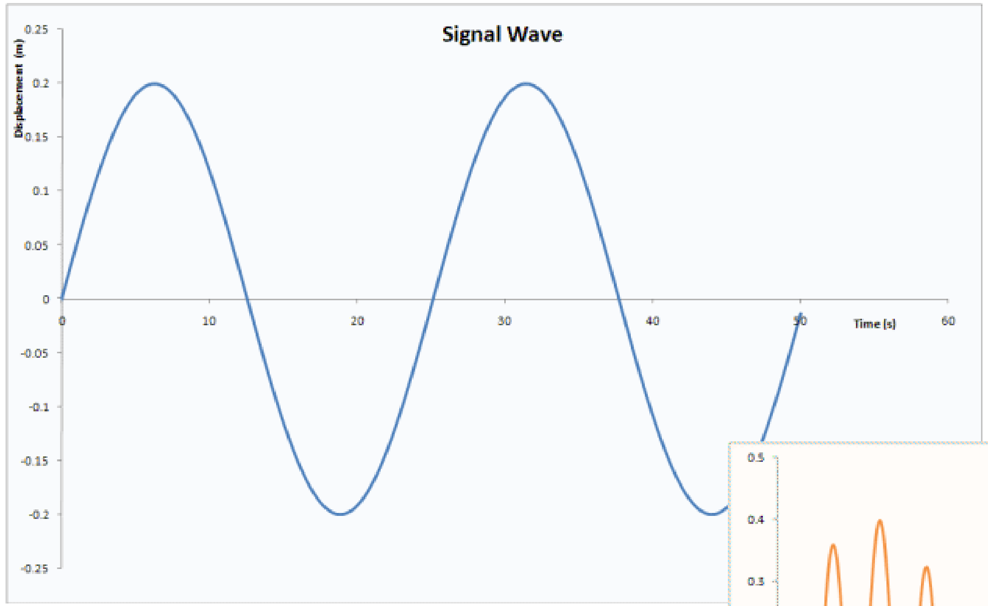
+



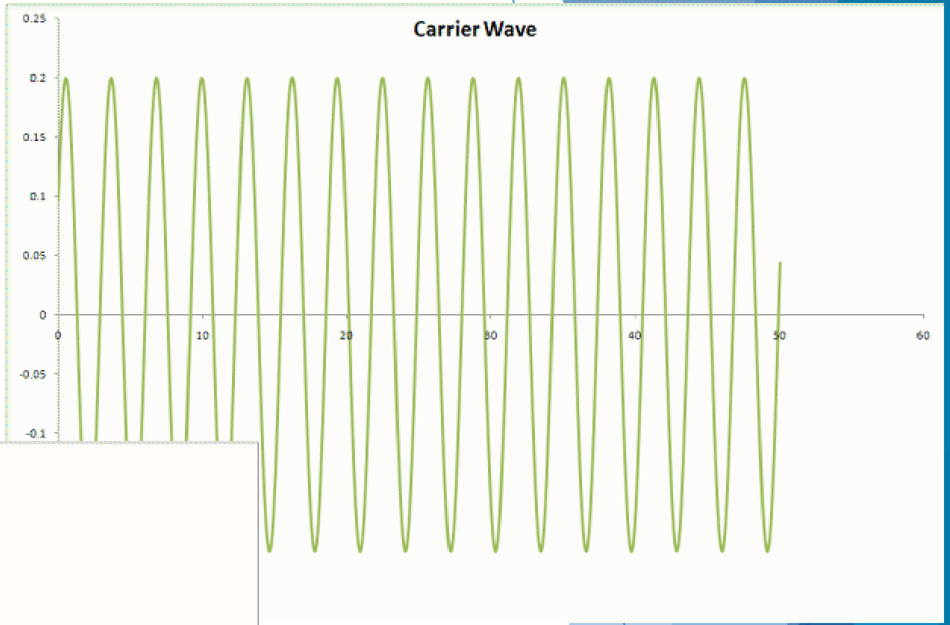
? =

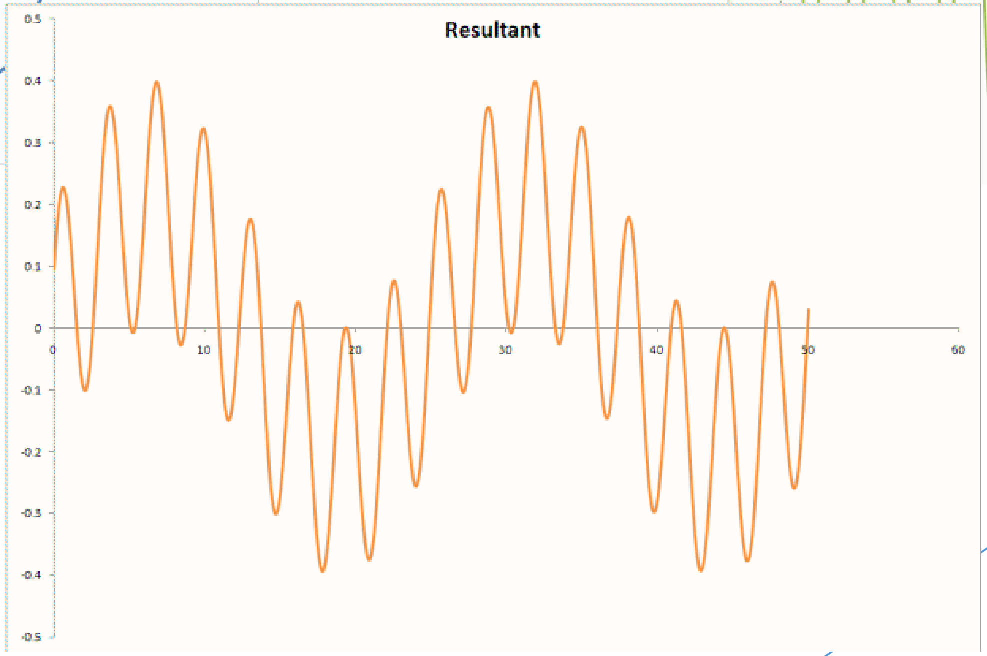
http://www.antonine-education.co.uk/Salters/MDM/Sending.htm

Playing Two Sounds



+





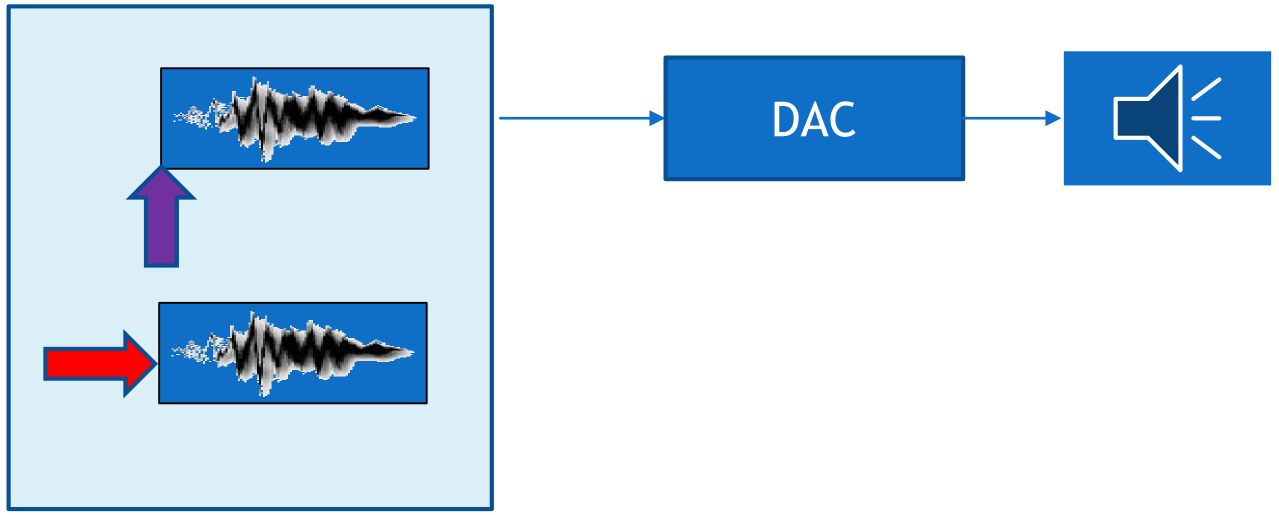
Playing Two Sounds

$$Output = A + B$$

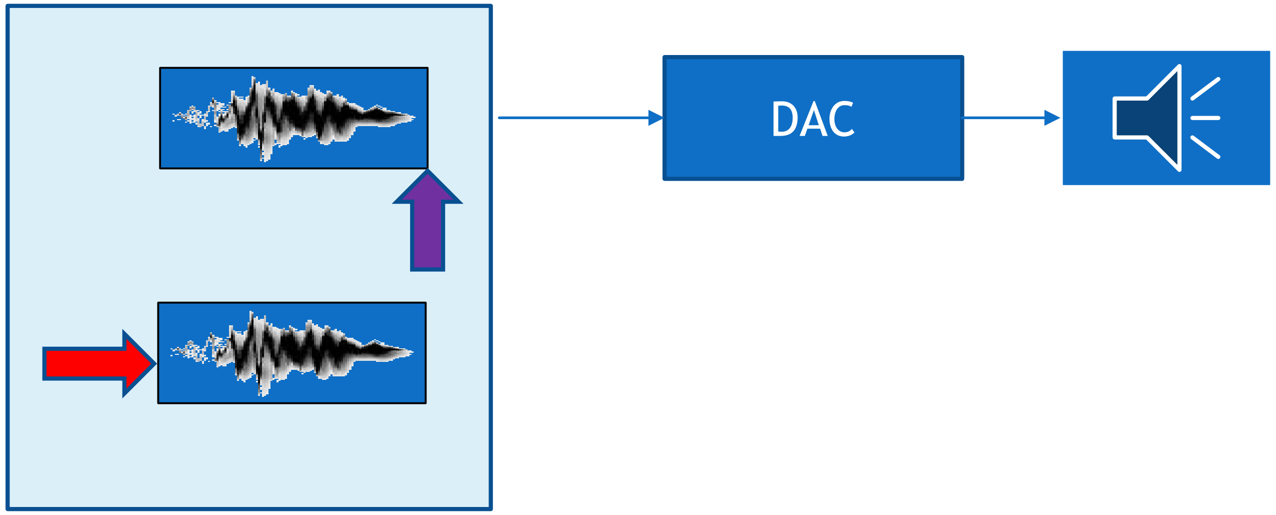
Playing Multiple Sounds

$$Output = \sum_{i} Signal_i$$

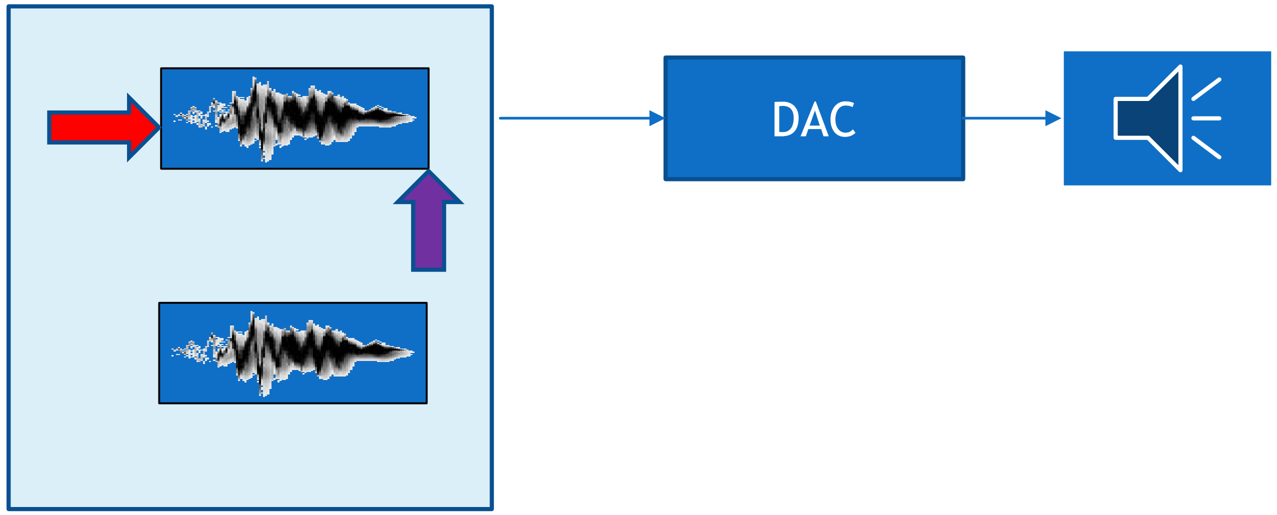
An “As-If” Audio Device



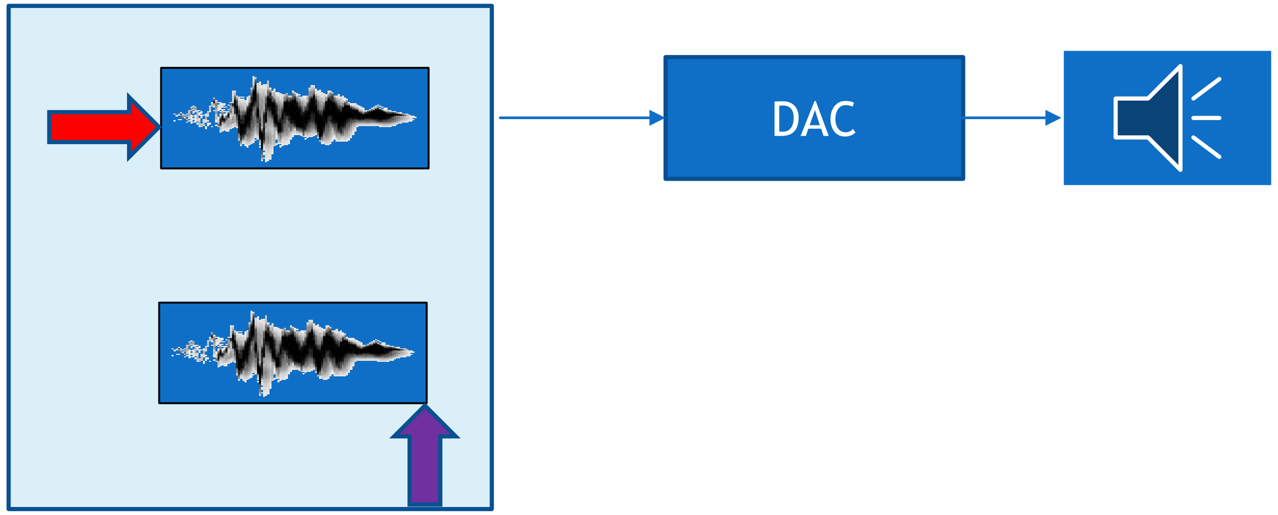
An “As-If” Audio Device



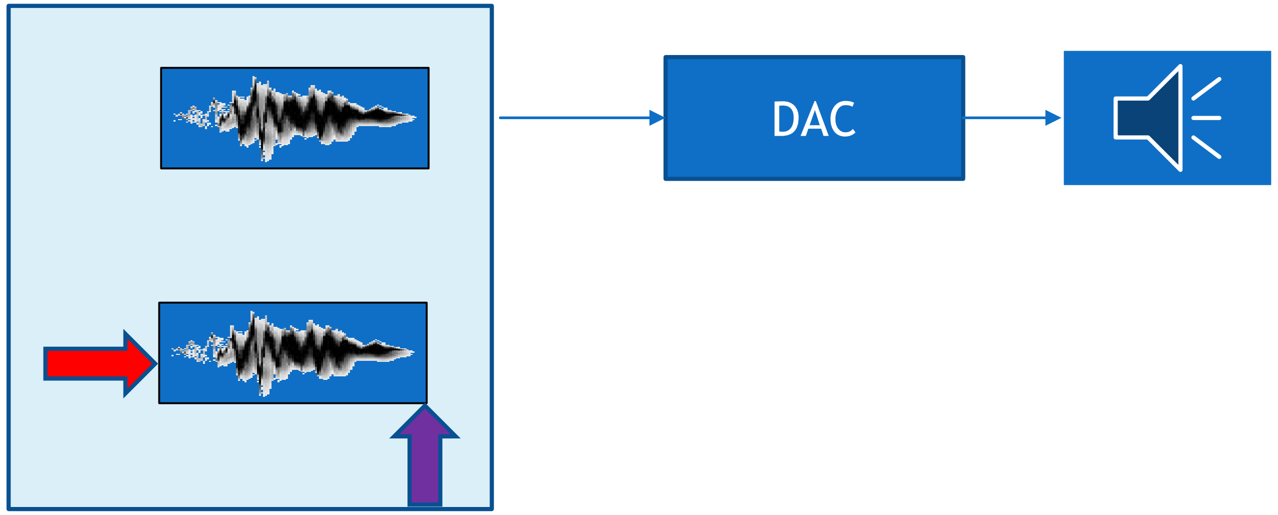
An “As-If” Audio Device



An "As-If" Audio Device

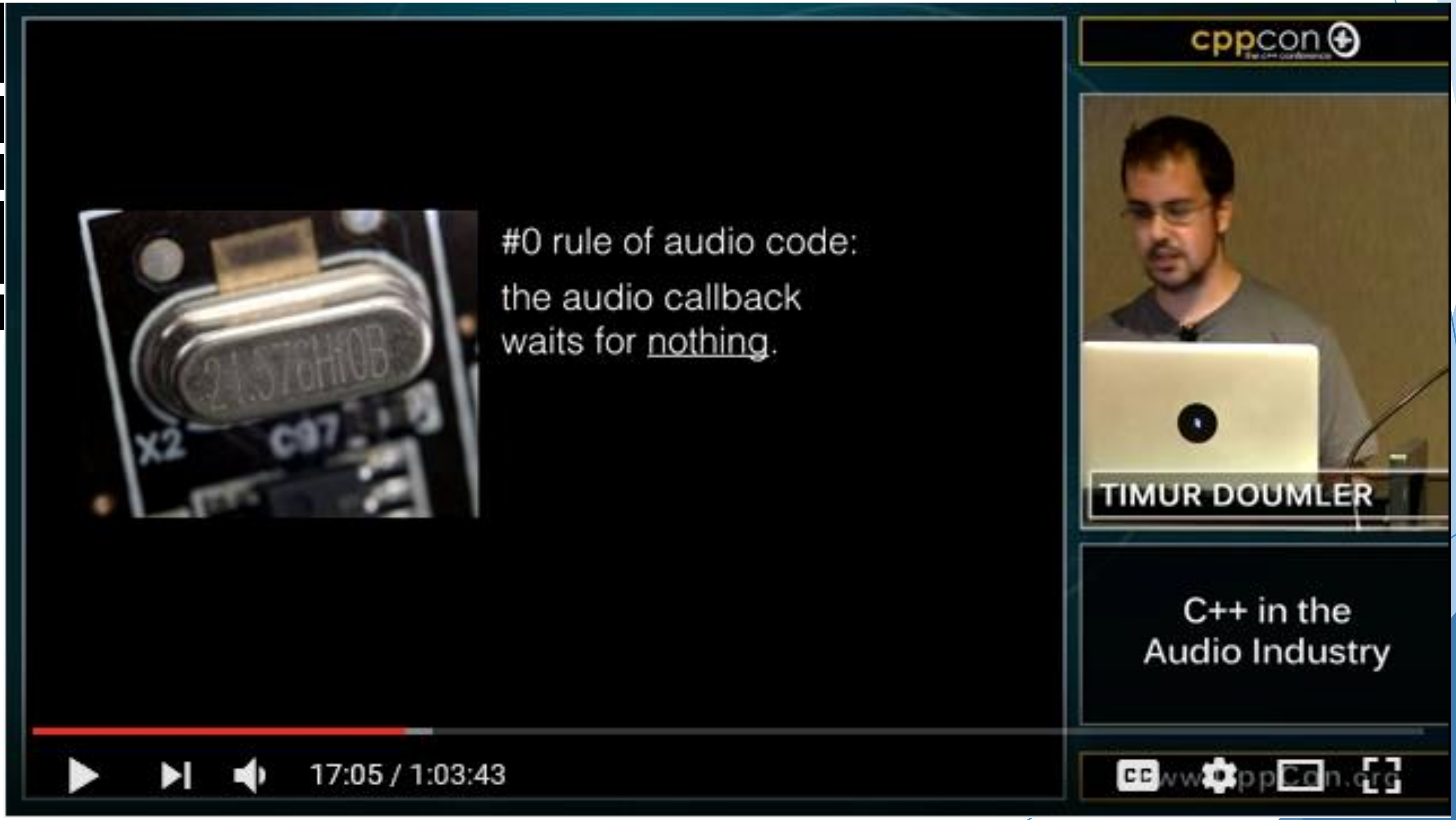


An “As-If” Audio Device



CppCon 2015

https://www.youtube.com/watch?v=boPEO2auJj4&t=3s



#0 rule of audio code:

the audio callback waits for nothing.

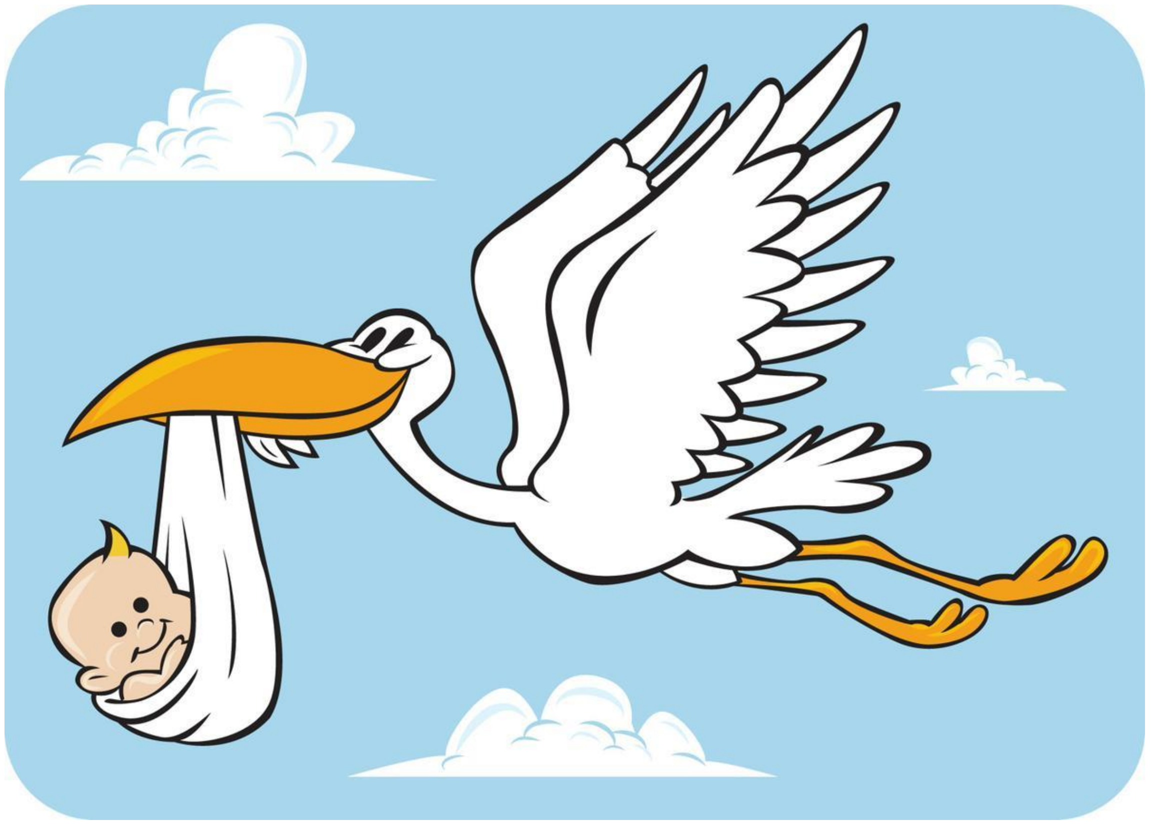
Mixer Thread Must be Real-time Safe

- We have to guarantee that
  - The function will return in time < buffer length
  - Will finish processing the whole buffer
  - Output contains valid audio data
  - No errors/exceptions

Mixer Thread Must be Real-time Safe

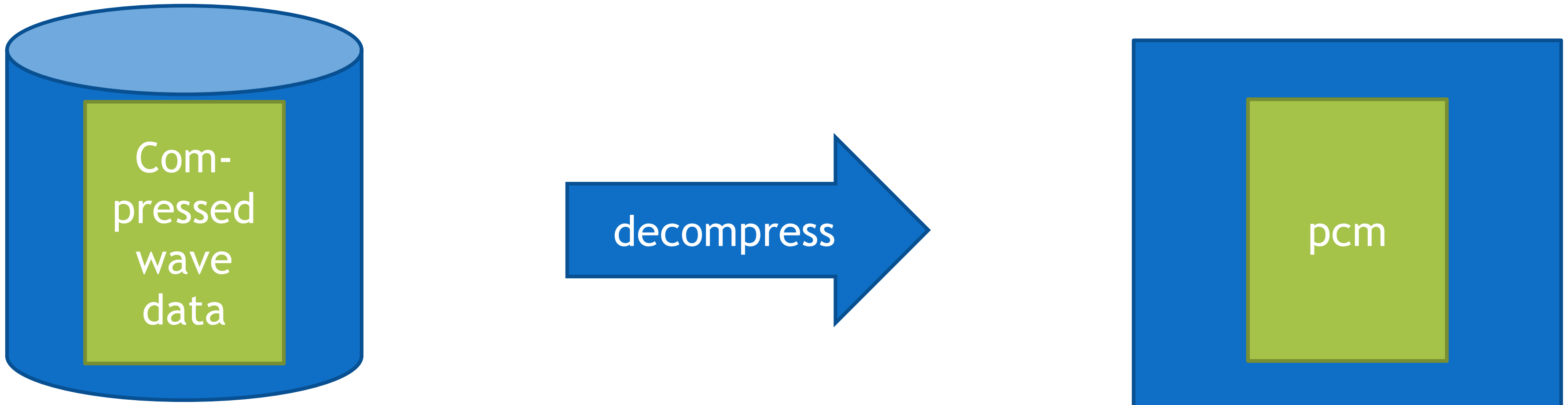
- We have to guarantee that
  - The function will return in time < buffer length
  - Will finish processing the whole buffer
  - Output contains valid audio data
  - No errors/exceptions
- Therefore:
  - Mixer thread must run at high OS priority
  - Never block the mixer thread
    - Lock-free = good. Wait-free = optimal.
  - No memory allocations/deallocations
  - No I/O (console, IPC, disk, network, etc.)

Where Audio Data Comes From

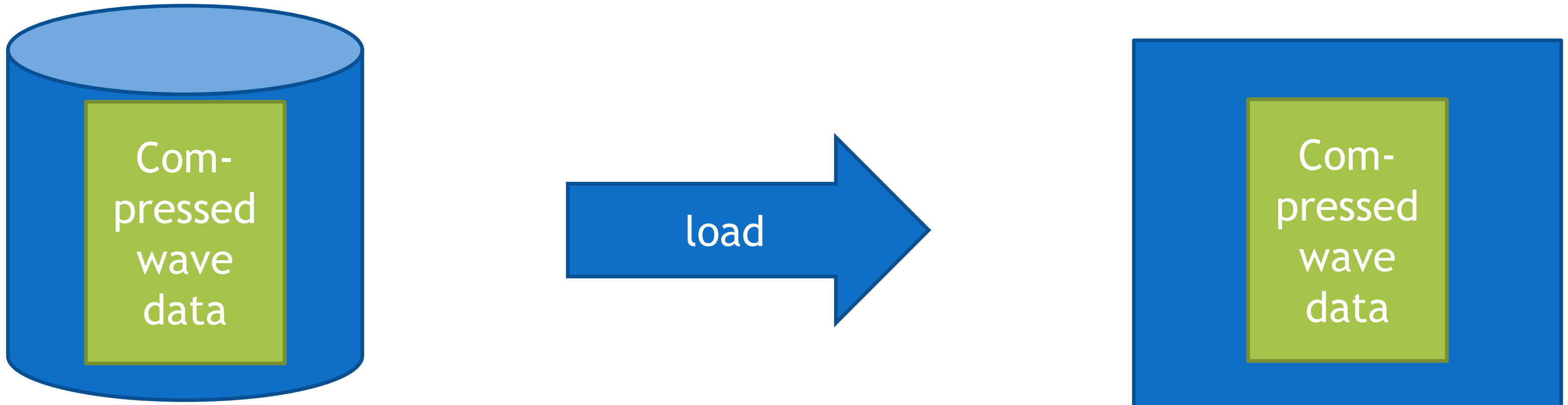


https://pix-media.priceonomics-media.com/blog/699/stork_baby.jpg

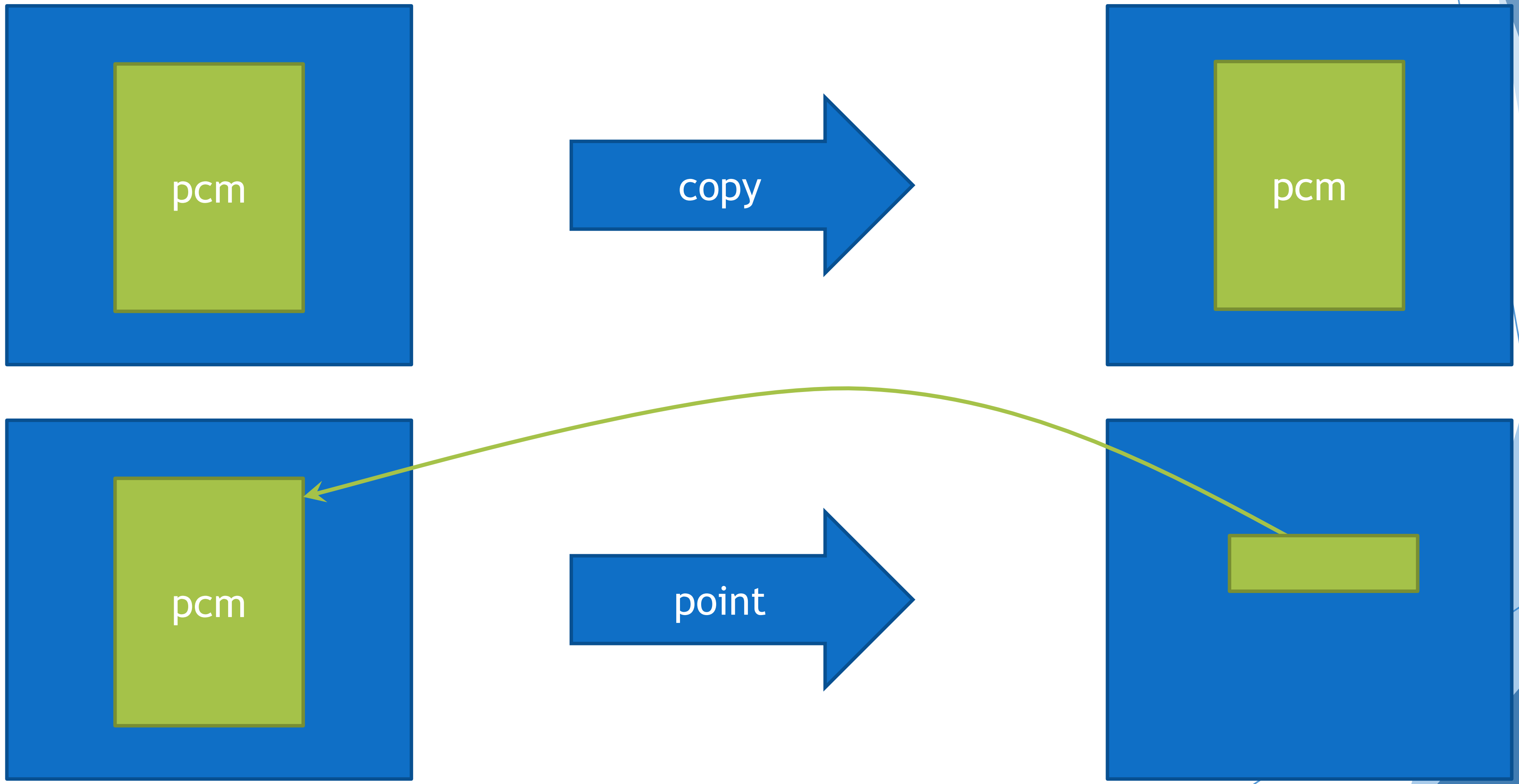
Option 1: File Decompressed to Memory (Sample)



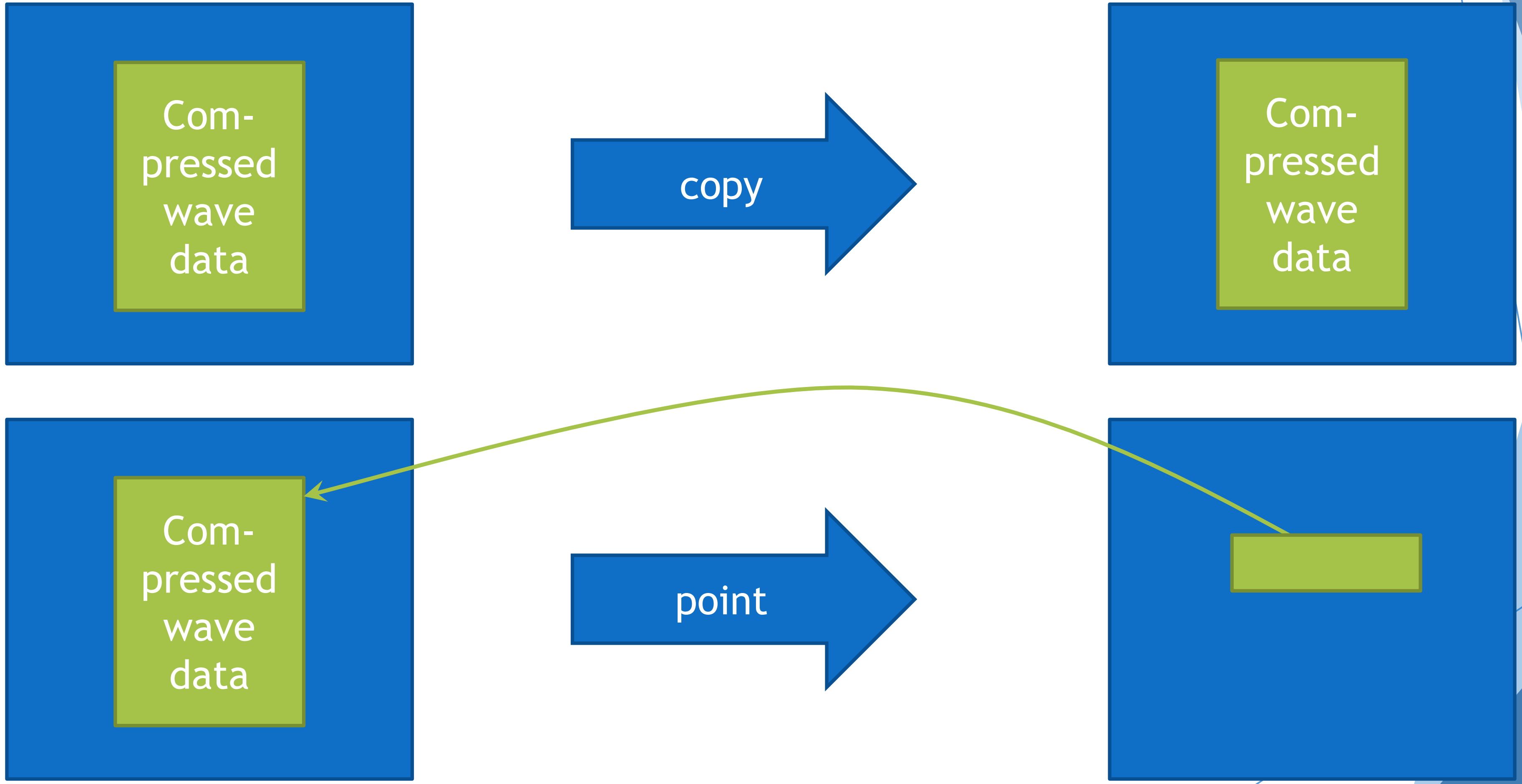
Option 2: Compressed Sample



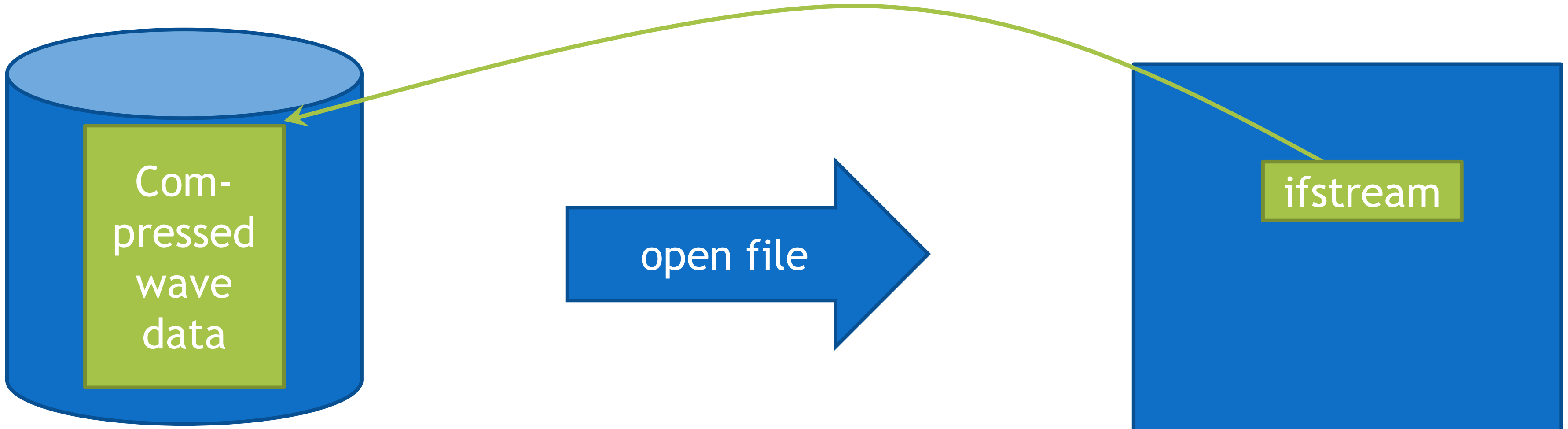
Option 3: Memory Buffer



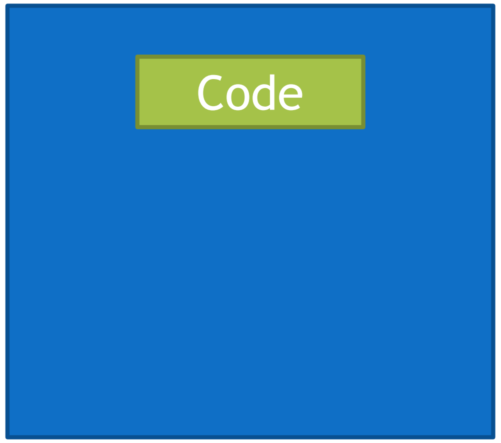
Option 3b: Compressed Memory Buffer



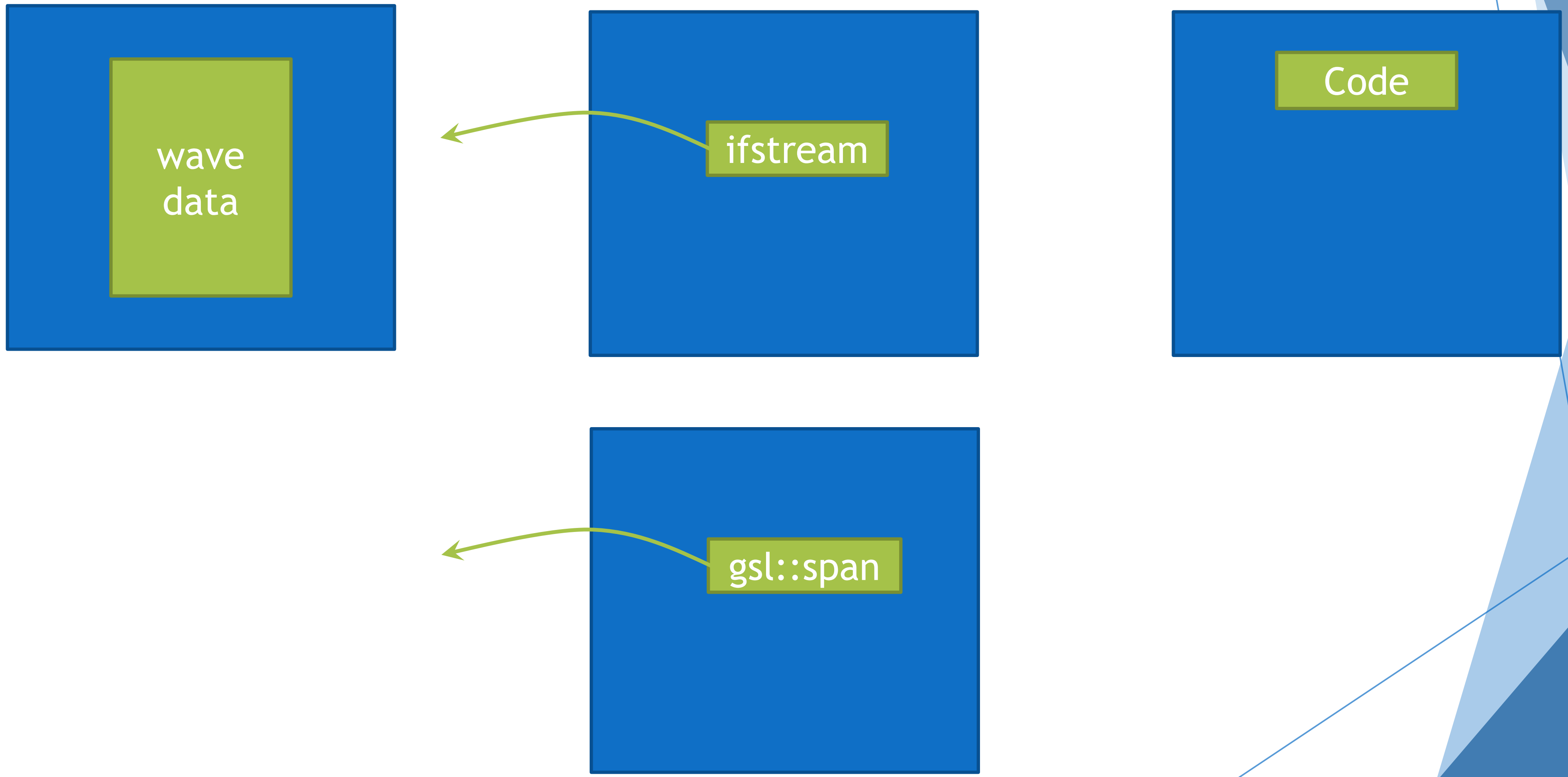
Option 4: Stream



Option 5: Synth



Actually…



Something like

```
std::pair<
  std::variant<
    std::vector<std::byte>,
    gsl::span<std::byte>,
    std::ifstream,
    std::function<…>>,
  std::audio::compression_type>
```

Spot Check: Where Are We?

- Audio data comes in
- Multiple sounds are played back
- Output to sound card

Spot Check: Where Are We?

- Audio data comes in
- Multiple sounds are played back
- Output to sound card

- What now?

Now…

- Resampling/Clipping/ Limiting
- 3D Panning and Attenuation
- Submixes
- Effects
- Reverb
- Dialog/Subtitles
- Randomization/Modulation
- Parameter automation
- Game hookup
- LFOs
- Mixing tech
  - (e.g. snapshots, VCAs…)
- Platform-specific requirements
  - (e.g. controller speakers)
- Obstruction/Occlusion/ Exclusion
- Background sounds/Ambience
- Music
- Audio tools
- …

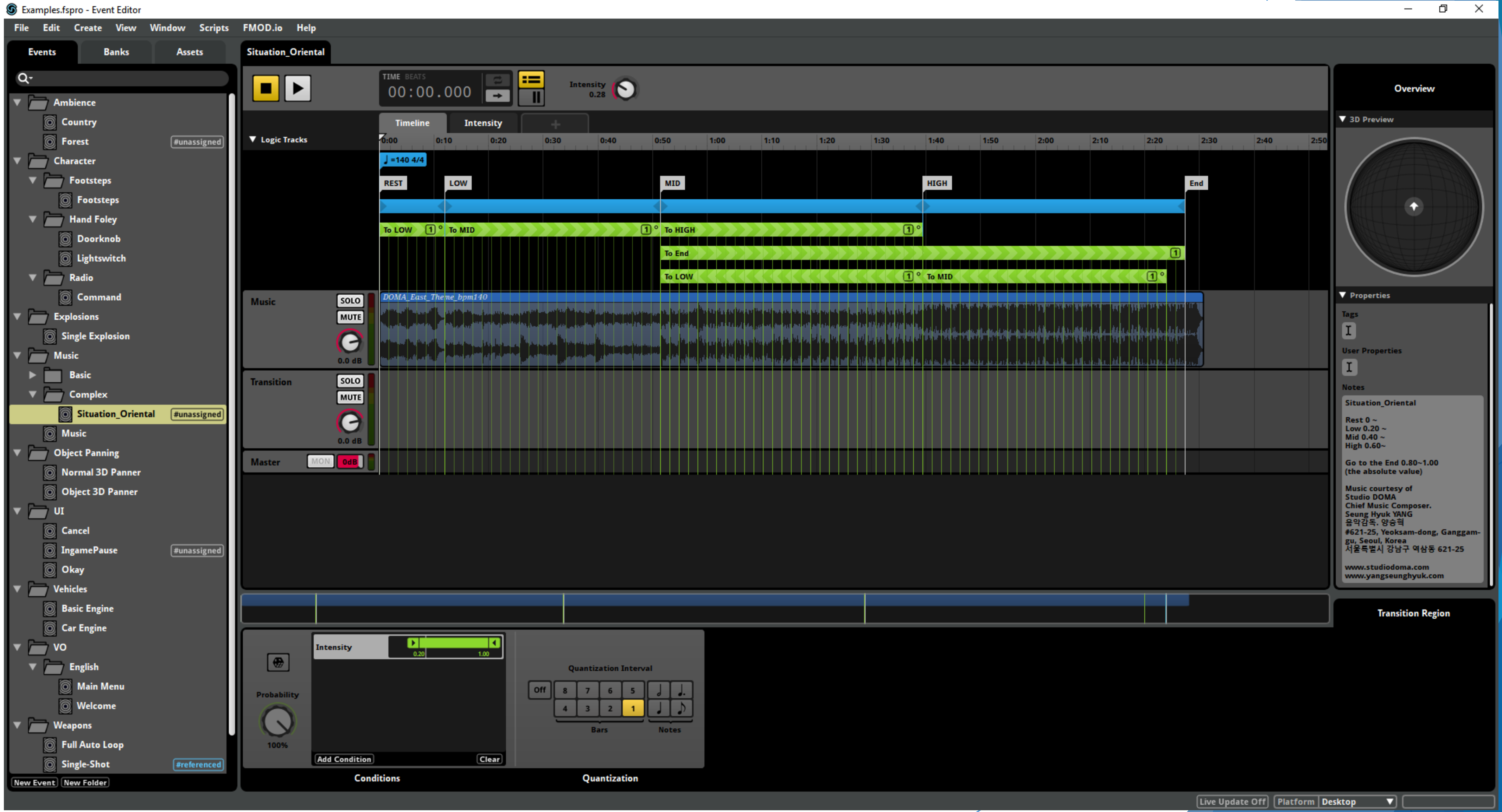
Current State of the Art

- OS APIs
  - WASAPI, ASIO, CoreAudio, PulseAudio, OSS, ALSA, OpenSL, etc.

Current State of the Art

- OS APIs
  - WASAPI, ASIO, CoreAudio, PulseAudio, OSS, ALSA, OpenSL, etc.
- But middleware is king:
  - FMOD Studio
  - Audiokinetic Wwise
  - CRI ADX2

FMOD Studio



Situation_Oriental

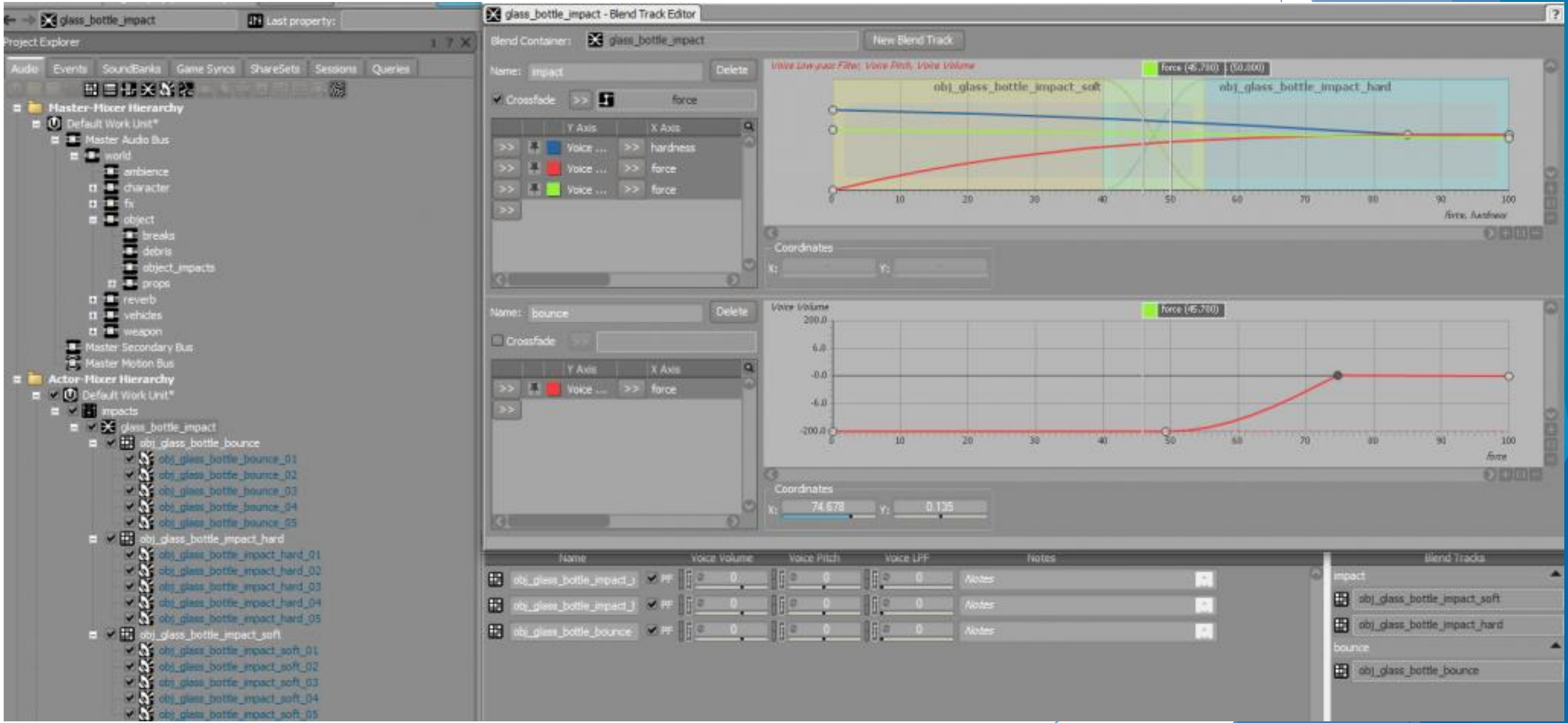
Rest 0 ~
Low 0.20 ~
Mid 0.40 ~
High 0.60~

Go to the End 0.80~1.00
(the absolute value)

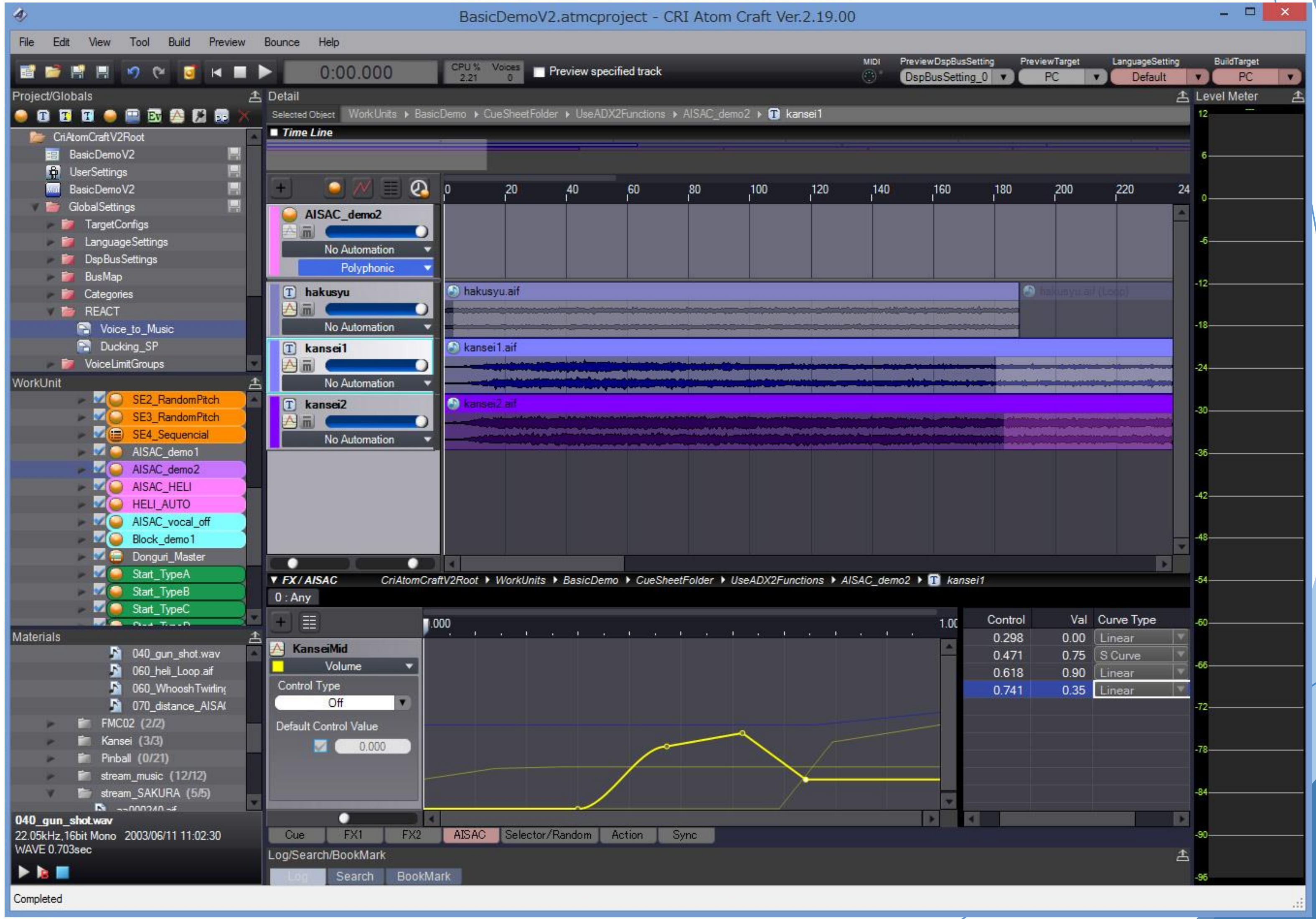
Music courtesy of
Studio DOMA
Chief Music Composer.
Seung Hyuk YANG
음악감독. 양승혁
#621-25, Yeoksam-dong, Ganggam-gu, Seoul, Korea
서울특별시 강남구 역삼동 621-25

www.studiodoma.com
www.yangseunghyuk.com

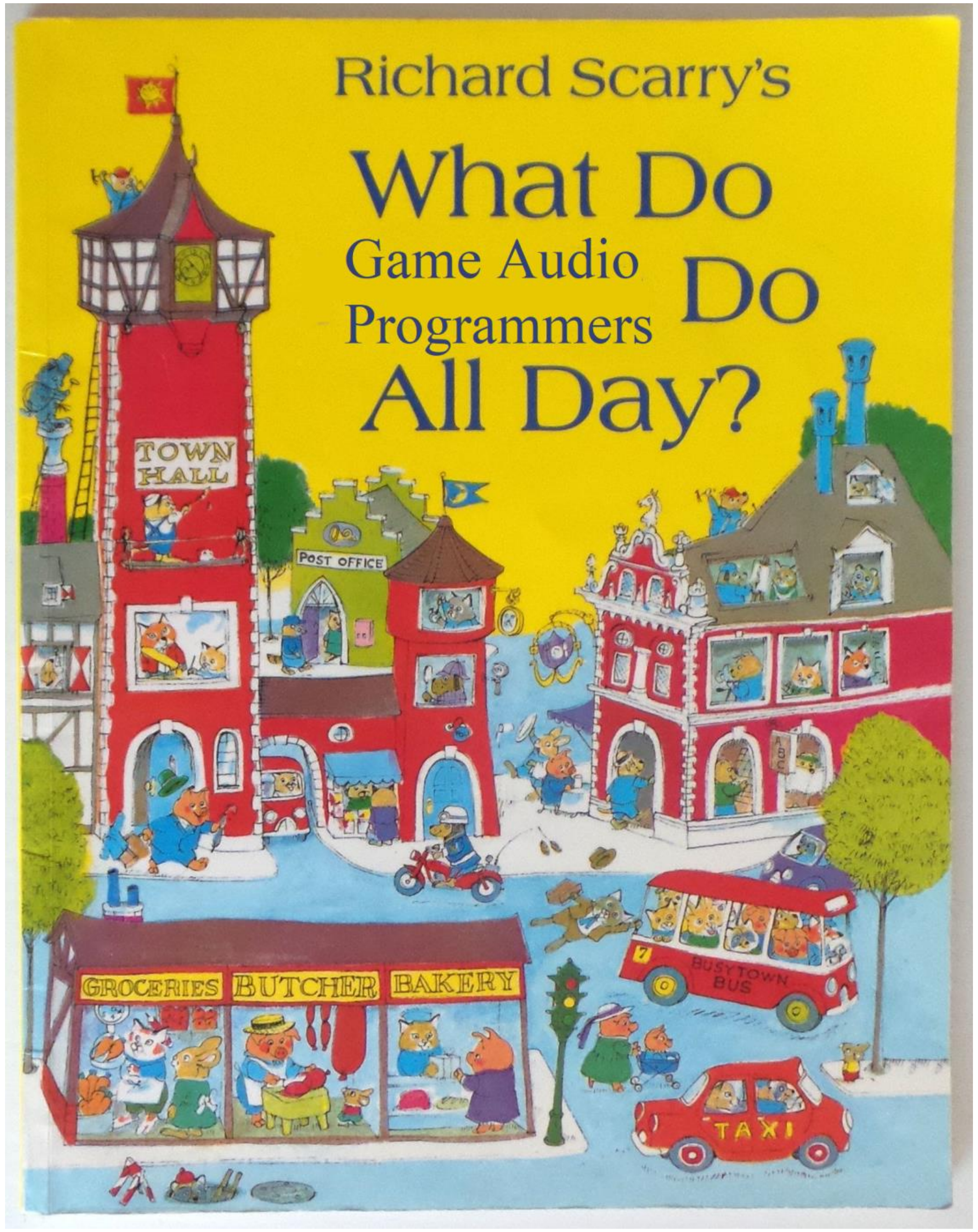
AudioKinetic Wwise


glass_bottle_impact
Last property:
Project Explorer
Audio
Events
SoundBanks
Game Syncs
ShareSets
Sessions
Queries
Master-Mixer Hierarchy
Default Work Unit*
Master Audio Bus
world
ambience
character
fx
object
breaks
debris
object_impacts
props
reverb
vehicles
weapon
Master Secondary Bus
Master Motion Bus
Actor-Mixer Hierarchy
Default Work Unit*
impacts
glass_bottle_impact
obj_glass_bottle_bounce
obj_glass_bottle_bounce_01
obj_glass_bottle_bounce_02
obj_glass_bottle_bounce_03
obj_glass_bottle_bounce_04
obj_glass_bottle_bounce_05
obj_glass_bottle_impact_hard
obj_glass_bottle_impact_hard_01
obj_glass_bottle_impact_hard_02
obj_glass_bottle_impact_hard_03
obj_glass_bottle_impact_hard_04
obj_glass_bottle_impact_hard_05
obj_glass_bottle_impact_soft
obj_glass_bottle_impact_soft_01
obj_glass_bottle_impact_soft_02
obj_glass_bottle_impact_soft_03
obj_glass_bottle_impact_soft_04
obj_glass_bottle_impact_soft_05
glass_bottle_impact - Blend Track Editor
Blend Container:
glass_bottle_impact
New Blend Track
Name:
impact
Delete
Crossfade
force
Y Axis
X Axis
Voice ...
hardness
force
Voice Low-pass Filter, Voice Pitch, Voice Volume
force (45.700)
(50.000)
obj_glass_bottle_impact_soft
obj_glass_bottle_impact_hard
force, hardness
Coordinates
X:
Y:
bounce
Voice Volume
200.0
6.0
-0.0
-6.0
-200.0
0
10
20
30
40
50
60
70
80
90
100
force
74.678
0.135
Name
Voice Volume
Voice Pitch
Voice LPF
Notes
Blend Tracks
obj_glass_bottle_impact_s
obj_glass_bottle_impact_h
obj_glass_bottle_bounce
PF
0
impact
obj_glass_bottle_impact_soft
obj_glass_bottle_impact_hard
bounce
obj_glass_bottle_bounce

CRI ADX2



Control	Val	Curve Type
0.298	0.00	Linear
0.471	0.75	S Curve
0.618	0.90	Linear
0.741	0.35	Linear


Richard Scarry's
What Do
Game Audio
Programmers
Do
All Day?
TOWN HALL
POST OFFICE
GROCERIES
BUTCHER
BAKERY
BUSYTOWN BUS
TAXI

Different Kinds of Audio Programmer

- Technical Sound Designer
  - Sound designer who can jump into the code to create hooks and implement features if necessary.
- Audio Engine Programmer
  - Implements/maintains audio engine logic and tools
  - Interacts with middleware/game code, and works closely with sound designers
- DSP Programmer
  - Implements custom effects and mixing techniques
- …

Different Kinds of Audio Programmer

- Technical Sound Designer
  - Sound designer who can jump into the code to create hooks and implement features if necessary.
- Audio Engine Programmer
  - Implements/maintains audio engine logic and tools
  - Interacts with middleware/game code, and works closely with sound designers
- DSP Programmer
  - Implements custom effects and mixing techniques
- …

Me!

Kinds of Things We Do

- Set up game hooks
- Figure out why sounds aren't playing
- Design complex events
- Implement custom effects
- Maintain audio engine logic
- Write integration tools
- Fix inscrutable bugs
- Implement DAW features in realtime
- Write auditing tools
- Automate content creation
- Content packaging rules
- Unlock sound designers creativity

How We Use C++

```
template<typename Fxn, typename... Ts>
using MemberFunctionReturn = typename std::result_of<Fxn&&(FFMODPlayingEvent&&, Ts&&...)>::type;

template<typename Fxn, typename... Ts>
static MemberFunctionReturn<Fxn, Ts...> ExecutePlayingEventFunction(
  int PlayingEventId, Fxn&& Function, Ts&&... ts)
{
  auto PlayEventShared = GetPlayingEvent(PlayingEventId);
  auto* PlayingEvent = PlayEventShared.Get();
  if (PlayingEvent != nullptr) {
    return (PlayingEvent->*Function)(std::forward<Ts>(ts)...);
  }

  return MemberFunctionReturn<Fxn, Ts...>();
}
```

How We Use C++

```
template<typename Fxn, typename... Ts>
using MemberFunctionReturn = typename std::result_of<Fxn&&(FFMODPlayingEvent&&, Ts&&...)>::type;

template<typename Fxn, typename... Ts>
static MemberFunctionReturn<Fxn, Ts...> ExecutePlayingEventFunction(
  int PlayingEventId, Fxn&& Function, Ts&&... ts)
{
  auto PlayEventShared = GetPlayingEvent(PlayingEventId);
  auto* PlayingEvent = PlayEventShared.Get();
  if (PlayingEvent != nullptr) {
    return (PlayingEvent->*Function)(std::forward<Ts>(ts)...);
  }

  return MemberFunctionReturn<Fxn, Ts...>();
}
```

How We Use C++

```
void UModularSynthComponent::SetAttackTime(float AttackTimeMsec)
{
    SynthCommand([this, AttackTimeMsec]()
    {
        EpicSynth1.SetEnvAttackTime(AttackTimeMsec);
    });
}
```

Let’s Bootstrap!

- We’ll build a game audio engine right now
- Using FMOD Studio low-level API

Minimal Sound Playback

```cpp
#include "fmod.hpp"

int main() {
  FMOD::System* pSystem = nullptr;
  FMOD::System_Create(&pSystem);
  pSystem->init(128, FMOD_INIT_NORMAL, nullptr);

  FMOD::Sound* pSound = nullptr;
  pSystem->createSound(R"(c:\windows\media\tada.wav)", FMOD_DEFAULT, nullptr, &pSound);

  FMOD::Channel* pChannel = nullptr;
  pSystem->playSound(pSound, nullptr, false, &pChannel);

  bool bIsPlaying = true;
  while (bIsPlaying) {
    pChannel->isPlaying(&bIsPlaying);
    pSystem->update();
  }

  return 0;
}
```

Let’s Build an Audio Engine (v1)

- 99% Light Speed

```
class AudioEngine
{
public:
    static void Init();
    static void Update();
    static void Shutdown();

    void LoadSound(const string& strSoundName, bool b3d=true, bool bLooping=false, bool bStream=false);
    void UnLoadSound(const string& strSoundName);
    void Set3dListenerAndOrientation(const Vector3& vPosition, const Vector3& vLook, const Vector3& vUp);
    int PlaySound(const string& strSoundName, const Vector3& vPos=Vector3{0,0,0}, float fVolumedB = 0.0f);
    void StopChannel(int nChannelId);
    void PauseChannel(int nChannelId);
    void StopAllChannels();
    void SetChannel3dPosition(int nChannelId, const Vector3& vPosition);
    void SetChannelVolume(int nChannelId, float fVolumedB);
    bool IsPlaying(int nChannelId) const;
    // Add more functions as you need...
};
```

```cpp
class AudioEngine
{
public:
    static void Init();
    static void Update();
    static void Shutdown();

    void LoadSound(const string& strSoundName, bool b3d=true, bool bLooping=false, bool bStream=false);
    void UnLoadSound(const string& strSoundName);
    void Set3dListenerAndOrientation(const Vector3& vPosition, const Vector3& vLook, const Vector3& vUp);
    int PlaySound(const string& strSoundName, const Vector3& vPos=Vector3{0,0,0}, float fVolumedB = 0.0f);
    void StopChannel(int nChannelId);
    void PauseChannel(int nChannelId);
    void StopAllChannels();
    void SetChannel3dPosition(int nChannelId, const Vector3& vPosition);
    void SetChannelVolume(int nChannelId, float fVolumedB);
    bool IsPlaying(int nChannelId) const;
    // Add more functions as you need...
};
```

```
class AudioEngine
{
public:
    static void Init();
    static void Update();
    static void Shutdown();

    void LoadSound(const string& strSoundName, bool b3d=true, bool bLooping=false, bool bStream=false);
    void UnLoadSound(const string& strSoundName);
    void Set3dListenerAndOrientation(const Vector3& vPosition, const Vector3& vLook, const Vector3& vUp);
    int PlaySound(const string& strSoundName, const Vector3& vPos=Vector3{0,0,0}, float fVolumedB = 0.0f);
    void StopChannel(int nChannelId);
    void PauseChannel(int nChannelId);
    void StopAllChannels();
    void SetChannel3dPosition(int nChannelId, const Vector3& vPosition);
    void SetChannelVolume(int nChannelId, float fVolumedB);
    bool IsPlaying(int nChannelId) const;
    // Add more functions as you need...
};
```

```
struct Implementation
{
    Implementation();
    ~Implementation();

    void Update();

    FMOD::System* mpSystem;

    int mnNextChannelId;

    typedef map<string, FMOD::Sound*> SoundMap;
    typedef map<int, FMOD::Channel*> ChannelMap;
    SoundMap mSounds;
    ChannelMap mChannels;
};
```

```
Implementation* sgpImplementation = nullptr;

void AudioEngine::Init()
{
    sgpImplementation = new Implementation;
}

void AudioEngine::Update()
{
    sgpImplementation->Update();
}

void AudioEngine::Shutdown()
{
    delete sgpImplementation;
}
```

```
struct Implementation
{
    Implementation();
    ~Implementation();

    void Update();

    FMOD::System* mpSystem;

    int mnNextChannelId;

    typedef map<string, FMOD::Sound*> SoundMap;
    typedef map<int, FMOD::Channel*> ChannelMap;
    SoundMap mSounds;
    ChannelMap mChannels;
};
```

```
Implementation* sgpImplementation = nullptr;

void AudioEngine::Init()
{
    sgpImplementation = new Implementation;
}

void AudioEngine::Update()
{
    sgpImplementation->Update();
}

void AudioEngine::Shutdown()
{
    delete sgpImplementation;
}
```

```
struct Implementation
{
    Implementation();
    ~Implementation();

    void Update();

    FMOD::System* mpSystem;

    int mnNextChannelId;

    typedef map<string, FMOD::Sound*> SoundMap;
    typedef map<int, FMOD::Channel*> ChannelMap;
    SoundMap mSounds;
    ChannelMap mChannels;
};
```

```
Implementation* sgpImplementation = nullptr;

void AudioEngine::Init()
{
    sgpImplementation = new Implementation;
}

void AudioEngine::Update()
{
    sgpImplementation->Update();
}

void AudioEngine::Shutdown()
{
    delete sgpImplementation;
}
```

```cpp
void Implementation::Update()
{
    vector<ChannelMap::iterator> pStoppedChannels;
    for(auto it = mChannels.begin(), itEnd = mChannels.end(); it != itEnd; ++it)
    {
        bool bIsPlaying = false;
        it->second->isPlaying(&bIsPlaying);
        if(!bIsPlaying)
        {
            pStoppedChannels.push_back(it);
        }
    }
    for(auto& it : pStoppedChannels)
    {
        mChannels.erase(it);
    }
    mpSystem->update();
}
```

```
void Implementation::Update()
{
    vector<ChannelMap::iterator> pStoppedChannels;
    for(auto it = mChannels.begin(), itEnd = mChannels.end(); it != itEnd; ++it)
    {
        bool bIsPlaying = false;
        it->second->isPlaying(&bIsPlaying);
        if(!bIsPlaying)
        {
            pStoppedChannels.push_back(it);
        }
    }
    for(auto& it : pStoppedChannels)
    {
        mChannels.erase(it);
    }
    mpSystem->update();
}
```

```
void Implementation::Update()
{
    vector<ChannelMap::iterator> pStoppedChannels;
    for(auto it = mChannels.begin(), itEnd = mChannels.end(); it != itEnd; ++it)
    {
        bool bIsPlaying = false;
        it->second->isPlaying(&bIsPlaying);
        if(!bIsPlaying)
        {
            pStoppedChannels.push_back(it);
        }
    }
    for(auto& it : pStoppedChannels)
    {
        mChannels.erase(it);
    }
    mpSystem->update();
}
```

```
void Implementation::Update()
{
    vector<ChannelMap::iterator> pStoppedChannels;
    for(auto it = mChannels.begin(), itEnd = mChannels.end(); it != itEnd; ++it)
    {
        bool bIsPlaying = false;
        it->second->isPlaying(&bIsPlaying);
        if(!bIsPlaying)
        {
            pStoppedChannels.push_back(it);
        }
    }
    for(auto& it : pStoppedChannels)
    {
        mChannels.erase(it);
    }
    mpSystem->update();
}
```

```cpp
void AudioEngine::LoadSound(const std::string& strSoundName, bool b3d, bool bLooping, bool bStream)
{
    auto tFoundIt = sgpImplementation->mSounds.find(strSoundName);
    if(tFoundIt != sgpImplementation->mSounds.end())
        return;
    FMOD_MODE eMode = FMOD_DEFAULT;
    eMode |= b3d ? FMOD_3D : FMOD_2D;
    eMode |= bLooping ? FMOD_LOOP_NORMAL : FMOD_LOOP_OFF;
    eMode |= bStream ? FMOD_CREATESTREAM : FMOD_CREATECOMPRESSEDSAMPLE;
    FMOD::Sound* pSound = nullptr;
    sgpImplementation->mpSystem->createSound(strSoundName.c_str(), eMode, nullptr, &pSound);
    if(pSound)
    {
        sgpImplementation->mSounds[strSoundName] = pSound;
    }
}

void AudioEngine::UnLoadSound(const std::string& strSoundName)
{
    auto tFoundIt = sgpImplementation->mSounds.find(strSoundName);
    if(tFoundIt == sgpImplementation->mSounds.end())
        return;
    tFoundIt->second->release();
    sgpImplementation->mSounds.erase(tFoundIt);
}
```

```
void AudioEngine::LoadSound(const std::string& strSoundName, bool b3d, bool bLooping, bool bStream)
{
    auto tFoundIt = sgpImplementation->mSounds.find(strSoundName);
    if(tFoundIt != sgpImplementation->mSounds.end())                        1
        return;
    FMOD_MODE eMode = FMOD_DEFAULT;
    eMode |= b3d ? FMOD_3D : FMOD_2D;
    eMode |= bLooping ? FMOD_LOOP_NORMAL : FMOD_LOOP_OFF;
    eMode |= bStream ? FMOD_CREATESTREAM : FMOD_CREATECOMPRESSEDSAMPLE;
    FMOD::Sound* pSound = nullptr;
    sgpImplementation->mpSystem->createSound(strSoundName.c_str(), eMode, nullptr, &pSound);
    if(pSound)
    {
        sgpImplementation->mSounds[strSoundName] = pSound;
    }
}

void AudioEngine::UnLoadSound(const std::string& strSoundName)
{
    auto tFoundIt = sgpImplementation->mSounds.find(strSoundName);
    if(tFoundIt == sgpImplementation->mSounds.end())                           1
        return;
    tFoundIt->second->release();
    sgpImplementation->mSounds.erase(tFoundIt);
}
```

```
void AudioEngine::LoadSound(const std::string& strSoundName, bool b3d, bool bLooping, bool bStream)
{
    auto tFoundIt = sgpImplementation->mSounds.find(strSoundName);
    if(tFoundIt != sgpImplementation->mSounds.end())
        return;
    FMOD_MODE eMode = FMOD_DEFAULT;
    eMode |= b3d ? FMOD_3D : FMOD_2D;
    eMode |= bLooping ? FMOD_LOOP_NORMAL : FMOD_LOOP_OFF;
    eMode |= bStream ? FMOD_CREATESTREAM : FMOD_CREATECOMPRESSEDSAMPLE;
    FMOD::Sound* pSound = nullptr;
    sgpImplementation->mpSystem->createSound(strSoundName.c_str(), eMode, nullptr, &pSound);2
    if(pSound)
    {
        sgpImplementation->mSounds[strSoundName] = pSound;
    }
}

void AudioEngine::UnLoadSound(const std::string& strSoundName)
{
    auto tFoundIt = sgpImplementation->mSounds.find(strSoundName);
    if(tFoundIt == sgpImplementation->mSounds.end())
        return;
    tFoundIt->second->release();      2
    sgpImplementation->mSounds.erase(tFoundIt);
}
```

```
void AudioEngine::LoadSound(const std::string& strSoundName, bool b3d, bool bLooping, bool bStream)
{
    auto tFoundIt = sgpImplementation->mSounds.find(strSoundName);
    if(tFoundIt != sgpImplementation->mSounds.end())
        return;
    FMOD_MODE eMode = FMOD_DEFAULT;
    eMode |= b3d ? FMOD_3D : FMOD_2D;
    eMode |= bLooping ? FMOD_LOOP_NORMAL : FMOD_LOOP_OFF;
    eMode |= bStream ? FMOD_CREATESTREAM : FMOD_CREATECOMPRESSEDSAMPLE;
    FMOD::Sound* pSound = nullptr;
    sgpImplementation->mpSystem->createSound(strSoundName.c_str(), eMode, nullptr, &pSound);
    if(pSound)
    {
        sgpImplementation->mSounds[strSoundName] = pSound;           3
    }
}

void AudioEngine::UnLoadSound(const std::string& strSoundName)
{
    auto tFoundIt = sgpImplementation->mSounds.find(strSoundName);
    if(tFoundIt == sgpImplementation->mSounds.end())
        return;
    tFoundIt->second->release();
    sgpImplementation->mSounds.erase(tFoundIt); 3
}
```

```
int AudioEngine::PlaySound(const std::string& strSoundName, const Vector3& vPosition, float fVolumedB)
{
    int nChannelId = sgpImplementation->mnNextChannelId++;
    auto tFoundIt = sgpImplementation->mSounds.find(strSoundName);
    if(tFoundIt == sgpImplementation->mSounds.end())
    {
        LoadSound(strSoundName);
        tFoundIt = sgpImplementation->mSounds.find(strSoundName);
        if(tFoundIt == sgpImplementation->mSounds.end())
        {
            return nChannelId;
        }
    }
    FMOD::Channel* pChannel = nullptr;
    sgpImplementation->mpSystem->playSound(tFoundIt->second, nullptr, true, &pChannel);
    if(pChannel)
    {
        FMOD_VECTOR position = VectorToFmod(vPosition);
        pChannel->set3DAttributes(&position, nullptr);
        pChannel->setVolume(dBToVolume(fVolumedB));
        pChannel->setPaused(false);
        sgpImplementation->mChannels[nChannelId] = pChannel;
    }
    return nChannelId;
}
```

```
int AudioEngine::PlaySound(const std::string& strSoundName, const Vector3& vPosition, float fVolumedB)
{
    int nChannelId = sgpImplementation->mnNextChannelId++;
    auto tFoundIt = sgpImplementation->mSounds.find(strSoundName);
    if(tFoundIt == sgpImplementation->mSounds.end())
    {
        LoadSound(strSoundName);
        tFoundIt = sgpImplementation->mSounds.find(strSoundName);
        if(tFoundIt == sgpImplementation->mSounds.end())
        {
            return nChannelId;
        }
    }
    FMOD::Channel* pChannel = nullptr;
    sgpImplementation->mpSystem->playSound(tFoundIt->second, nullptr, true, &pChannel);
    if(pChannel)
    {
        FMOD_VECTOR position = VectorToFmod(vPosition);
        pChannel->set3DAttributes(&position, nullptr);
        pChannel->setVolume(dBToVolume(fVolumedB));
        pChannel->setPaused(false);
        sgpImplementation->mChannels[nChannelId] = pChannel;
    }
    return nChannelId;
}
```

```
int AudioEngine::PlaySound(const std::string& strSoundName, const Vector3& vPosition, float fVolumedB)
{
    int nChannelId = sgpImplementation->mnNextChannelId++;
    auto tFoundIt = sgpImplementation->mSounds.find(strSoundName);
    if(tFoundIt == sgpImplementation->mSounds.end())
    {
        LoadSound(strSoundName);
        tFoundIt = sgpImplementation->mSounds.find(strSoundName);
        if(tFoundIt == sgpImplementation->mSounds.end())
        {
            return nChannelId;
        }
    }
    FMOD::Channel* pChannel = nullptr;
    sgpImplementation->mpSystem->playSound(tFoundIt->second, nullptr, true, &pChannel);
    if(pChannel)
    {
        FMOD_VECTOR position = VectorToFmod(vPosition);
        pChannel->set3DAttributes(&position, nullptr);
        pChannel->setVolume(dBToVolume(fVolumedB));
        pChannel->setPaused(false);
        sgpImplementation->mChannels[nChannelId] = pChannel;
    }
    return nChannelId;
}
```

```
int AudioEngine::PlaySound(const std::string& strSoundName, const Vector3& vPosition, float fVolumedB)
{
    int nChannelId = sgpImplementation->mnNextChannelId++;
    auto tFoundIt = sgpImplementation->mSounds.find(strSoundName);
    if(tFoundIt == sgpImplementation->mSounds.end())
    {
        LoadSound(strSoundName);
        tFoundIt = sgpImplementation->mSounds.find(strSoundName);
        if(tFoundIt == sgpImplementation->mSounds.end())
        {
            return nChannelId;
        }
    }
    FMOD::Channel* pChannel = nullptr;
    sgpImplementation->mpSystem->playSound(tFoundIt->second, nullptr, true, &pChannel);
    if(pChannel)
    {
        FMOD_VECTOR position = VectorToFmod(vPosition);
        pChannel->set3DAttributes(&position, nullptr);
        pChannel->setVolume(dBToVolume(fVolumedB));
        pChannel->setPaused(false);
        sgpImplementation->mChannels[nChannelId] = pChannel;
    }
    return nChannelId;
}
```

```
int AudioEngine::PlaySound(const std::string& strSoundName, const Vector3& vPosition, float fVolumedB)
{
    int nChannelId = sgpImplementation->mnNextChannelId++;
    auto tFoundIt = sgpImplementation->mSounds.find(strSoundName);
    if(tFoundIt == sgpImplementation->mSounds.end())
    {
        LoadSound(strSoundName);
        tFoundIt = sgpImplementation->mSounds.find(strSoundName);
        if(tFoundIt == sgpImplementation->mSounds.end())
        {
            return nChannelId;
        }
    }
    FMOD::Channel* pChannel = nullptr;
    sgpImplementation->mpSystem->playSound(tFoundIt->second, nullptr, true, &pChannel);
    if(pChannel)
    {
        FMOD_VECTOR position = VectorToFmod(vPosition);
        pChannel->set3DAttributes(&position, nullptr);
        pChannel->setVolume(dBToVolume(fVolumedB));
        pChannel->setPaused(false);
        sgpImplementation->mChannels[nChannelId] = pChannel;
    }
    return nChannelId;
}
```

```cpp
void AudioEngine::SetChannelXXX(int nChannelId, Blah xxxValue)
{
    auto tFoundIt = sgpImplementation->mChannels.find(nChannelId);
    if(tFoundIt == sgpImplementation->mChannels.end())
        return;

    tFoundIt->second->setXXX(xxxValue);
}

// For example
void AudioEngine::SetChannelVolume(int nChannelId, float fVolumedB)
{
    auto tFoundIt = sgpImplementation->mChannels.find(nChannelId);
    if(tFoundIt == sgpImplementation->mChannels.end())
        return;

    tFoundIt->second->setVolume(dBToVolume(fVolumedB));
}
```

```
void AudioEngine::SetChannelXXX(int nChannelId, Blah xxxValue)
{
    auto tFoundIt = sgpImplementation->mChannels.find(nChannelId);
    if(tFoundIt == sgpImplementation->mChannels.end())
        return;

    tFoundIt->second->setXXX(xxxValue);
}
```

Boring

```
// For example
void AudioEngine::SetChannelVolume(int nChannelId, float fVolumedB)
{
    auto tFoundIt = sgpImplementation->mChannels.find(nChannelId);
    if(tFoundIt == sgpImplementation->mChannels.end())
        return;

    tFoundIt->second->setVolume(dBToVolume(fVolumedB));
}
```

Summary

- Line count: ~250 LOC
- Features:
  - Sound playback in 3D
  - Volume control
  - Jukebox functions

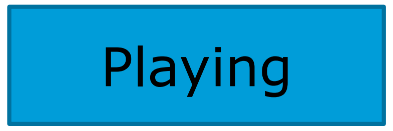
Summary

- Line count: ~250 LOC
- Features:
  - Sound playback in 3D
  - Volume control
  - Jukebox functions
- But...
  - Adding new features is hard

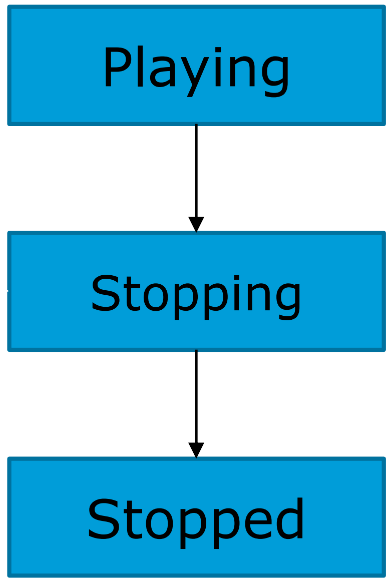
Adding Features

- To add features, we need to reorganize into a state machine
- Exemplar features:
  - Fadeouts
  - Async Loads
  - Virtual Sounds*

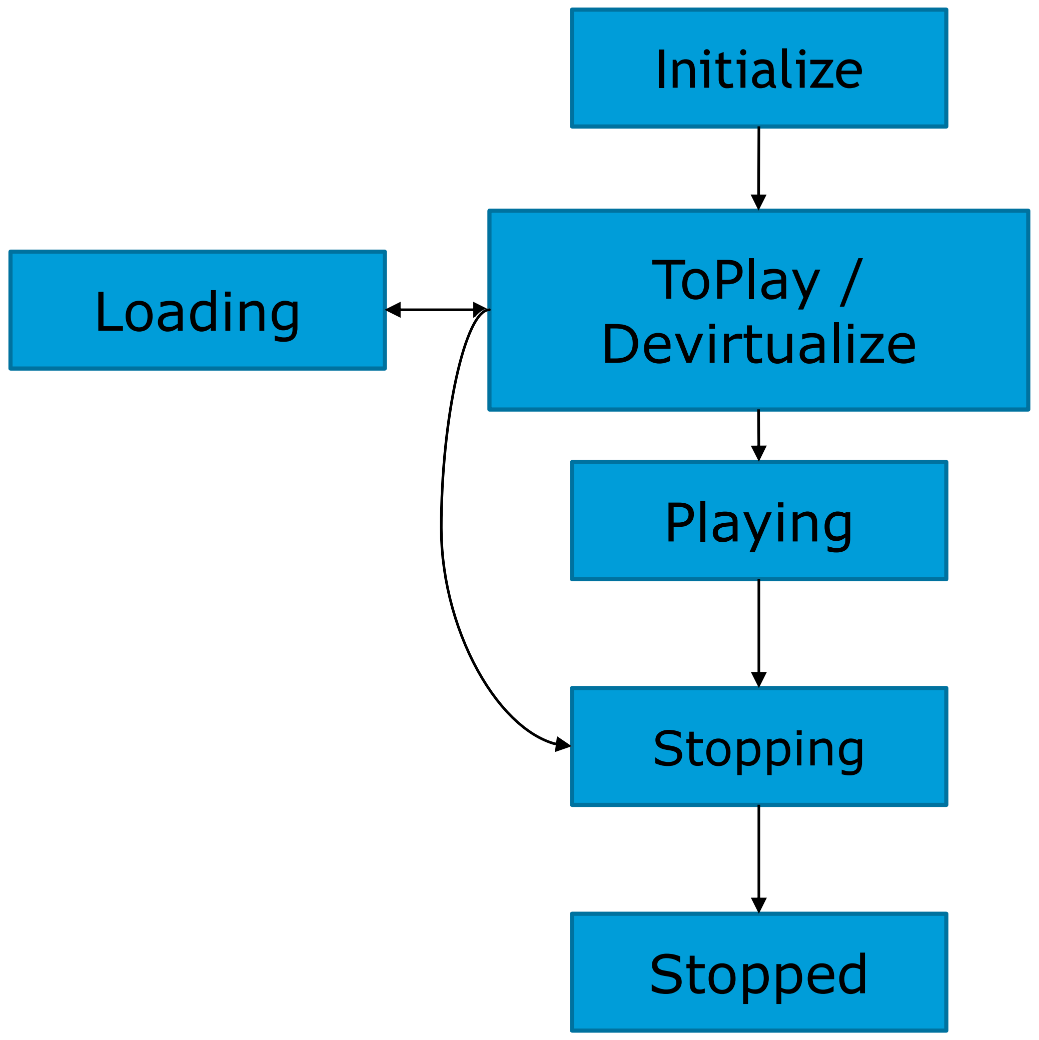
Our State Machine



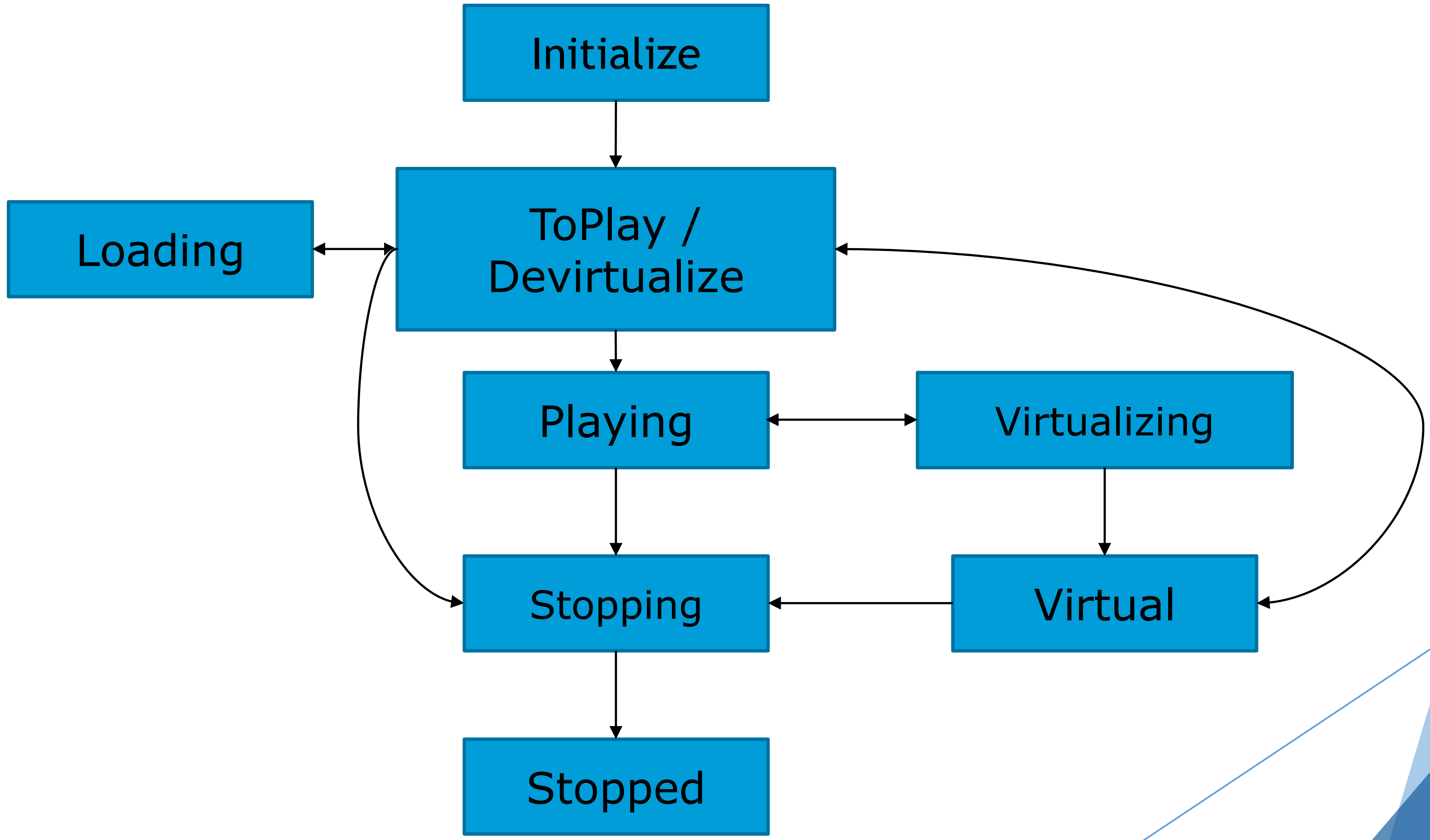
Our State Machine



Our State Machine

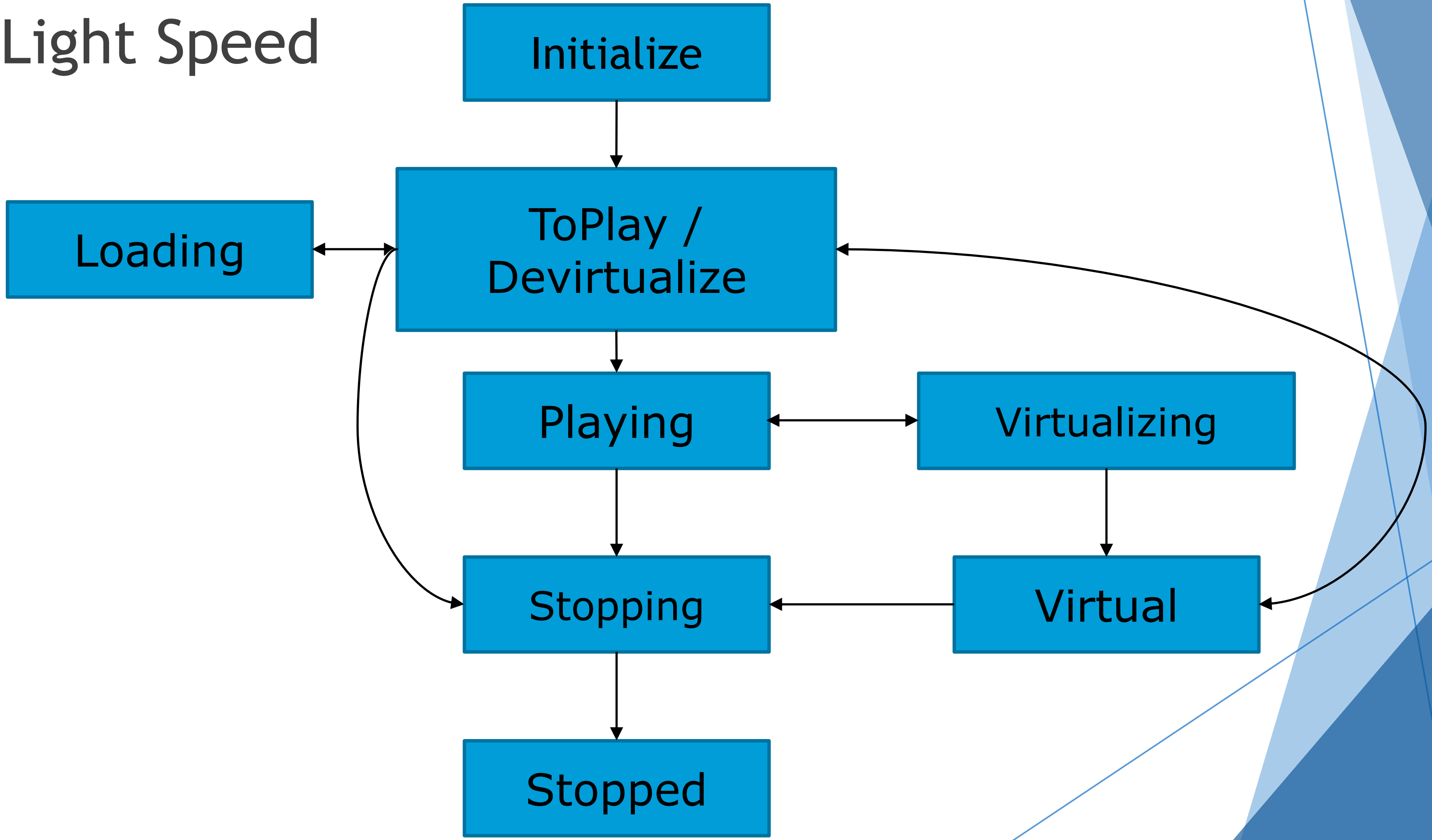


Our State Machine



Let’s Build an Audio Engine (v2)

- 99.9% Light Speed



```
class AudioEngine
{
public:
    static void Init();
    static void Update(float fTimeDeltaSeconds);
    static void Shutdown();

    struct SoundDefinition
    {
        std::string mSoundName;
        float fDefaultVolumedB;
        float fMinDistance;
        float fMaxDistance;
        bool bIs3d;
        bool bIsLooping;
        bool bIsStreaming;
    };

    int RegisterSound(const SoundDefinition& tSoundDefinition, bool bLoad = true);
    void UnregisterSound(int nSoundId);
    void LoadSound(int nSoundId);
    void UnLoadSound(int nSoundId);
    void Set3dListenerAndOrientation(const Vector3& vPosition, const Vector3& vLook, const Vector3& vUp);
    int PlaySound(int nSoundId, const Vector3& vPosition = Vector3{ 0, 0, 0 }, float fVolumedB = 0.0f);
    void StopChannel(int nChannelId, float fFadeTimeSeconds = 0.0f);
    void StopAllChannels();
    void SetChannel3dPosition(int nChannelId, const Vector3& vPosition);
    void SetChannelVolume(int nChannelId, float fVolumedB);
    bool IsPlaying(int nChannelId) const;
};
```

```
class AudioEngine
{
public:
    static void Init();
    static void Update(float fTimeDeltaSeconds);
    static void Shutdown();

    struct SoundDefinition
    {
        std::string mSoundName;
        float fDefaultVolumedB;
        float fMinDistance;
        float fMaxDistance;
        bool bIs3d;
        bool bIsLooping;
        bool bIsStreaming;
    };

    int RegisterSound(const SoundDefinition& tSoundDefinition, bool bLoad = true);
    void UnregisterSound(int nSoundId);
    void LoadSound(int nSoundId);
    void UnLoadSound(int nSoundId);
    void Set3dListenerAndOrientation(const Vector3& vPosition, const Vector3& vLook, const Vector3& vUp);
    int PlaySound(int nSoundId, const Vector3& vPosition = Vector3{ 0, 0, 0 }, float fVolumedB = 0.0f);
    void StopChannel(int nChannelId, float fFadeTimeSeconds = 0.0f);
    void StopAllChannels();
    void SetChannel3dPosition(int nChannelId, const Vector3& vPosition);
    void SetChannelVolume(int nChannelId, float fVolumedB);
    bool IsPlaying(int nChannelId) const;
};
```

```
class AudioEngine
{
public:
    static void Init();
    static void Update(float fTimeDeltaSeconds);
    static void Shutdown();

    struct SoundDefinition
    {
        std::string mSoundName;
        float fDefaultVolumedB;
        float fMinDistance;
        float fMaxDistance;
        bool bIs3d;
        bool bIsLooping;
        bool bIsStreaming;
    };

    int RegisterSound(const SoundDefinition& tSoundDefinition, bool bLoad = true);
    void UnregisterSound(int nSoundId);
    void LoadSound(int nSoundId);
    void UnLoadSound(int nSoundId);
    void Set3dListenerAndOrientation(const Vector3& vPosition, const Vector3& vLook, const Vector3& vUp);
    int PlaySound(int nSoundId, const Vector3& vPosition = Vector3{ 0, 0, 0 }, float fVolumedB = 0.0f);
    void StopChannel(int nChannelId, float fFadeTimeSeconds = 0.0f);
    void StopAllChannels();
    void SetChannel3dPosition(int nChannelId, const Vector3& vPosition);
    void SetChannelVolume(int nChannelId, float fVolumedB);
    bool IsPlaying(int nChannelId) const;
};
```

Fake!

```
class AudioEngine
{
public:
    static void Init();
    static void Update(float fTimeDeltaSeconds);
    static void Shutdown();

    struct SoundDefinition
    {
        std::string mSoundName;
        float fDefaultVolumedB;
        float fMinDistance;
        float fMaxDistance;
        bool bIs3d;
        bool bIsLooping;
        bool bIsStreaming;
    };

    int RegisterSound(const SoundDefinition& tSoundDefinition, bool bLoad = true);
    void UnregisterSound(int nSoundId);
    void LoadSound(int nSoundId);
    void UnLoadSound(int nSoundId);
    void Set3dListenerAndOrientation(const Vector3& vPosition, const Vector3& vLook, const Vector3& vUp);
    int PlaySound(int nSoundId, const Vector3& vPosition = Vector3{ 0, 0, 0 }, float fVolumedB = 0.0f);
    void StopChannel(int nChannelId, float fFadeTimeSeconds = 0.0f);
    void StopAllChannels();
    void SetChannel3dPosition(int nChannelId, const Vector3& vPosition);
    void SetChannelVolume(int nChannelId, float fVolumedB);
    bool IsPlaying(int nChannelId) const;
};
```

Fake!

But…shippable

```
class AudioEngine
{
public:
    static void Init();
    static void Update(float fTimeDeltaSeconds);
    static void Shutdown();

    struct SoundDefinition
    {
        std::string mSoundName;
        float fDefaultVolumedB;
        float fMinDistance;
        float fMaxDistance;
        bool bIs3d;
        bool bIsLooping;
        bool bIsStreaming;
    };

    int RegisterSound(const SoundDefinition& tSoundDefinition, bool bLoad = true);
    void UnregisterSound(int nSoundId);
    void LoadSound(int nSoundId);
    void UnLoadSound(int nSoundId);
    void Set3dListenerAndOrientation(const Vector3& vPosition, const Vector3& vLook, const Vector3& vUp);
    int PlaySound(int nSoundId, const Vector3& vPosition = Vector3{ 0, 0, 0 }, float fVolumedB = 0.0f);
    void StopChannel(int nChannelId, float fFadeTimeSeconds = 0.0f);
    void StopAllChannels();
    void SetChannel3dPosition(int nChannelId, const Vector3& vPosition);
    void SetChannelVolume(int nChannelId, float fVolumedB);
    bool IsPlaying(int nChannelId) const;
};
```

Fake!

But…shippable

Also, boring

```
class AudioEngine
{
public:
    static void Init();
    static void Update(float fTimeDeltaSeconds);
    static void Shutdown();

    struct SoundDefinition
    {
        std::string mSoundName;
        float fDefaultVolumedB;
        float fMinDistance;
        float fMaxDistance;
        bool bIs3d;
        bool bIsLooping;
        bool bIsStreaming;
    };

    int RegisterSound(const SoundDefinition& tSoundDefinition, bool bLoad = true);
    void UnregisterSound(int nSoundId);
    void LoadSound(int nSoundId);
    void UnLoadSound(int nSoundId);
    void Set3dListenerAndOrientation(const Vector3& vPosition, const Vector3& vLook, const Vector3& vUp);
    int PlaySound(int nSoundId, const Vector3& vPosition = Vector3{ 0, 0, 0 }, float fVolumedB = 0.0f);
    void StopChannel(int nChannelId, float fFadeTimeSeconds = 0.0f);
    void StopAllChannels();
    void SetChannel3dPosition(int nChannelId, const Vector3& vPosition);
    void SetChannelVolume(int nChannelId, float fVolumedB);
    bool IsPlaying(int nChannelId) const;
};
```

The same

```cpp
class AudioEngine
{
public:
    static void Init();
    static void Update(float fTimeDeltaSeconds);
    static void Shutdown();

    struct SoundDefinition
    {
        std::string mSoundName;
        float fDefaultVolumedB;
        float fMinDistance;
        float fMaxDistance;
        bool bIs3d;
        bool bIsLooping;
        bool bIsStreaming;
    };

    int RegisterSound(const SoundDefinition& tSoundDefinition, bool bLoad = true);
    void UnregisterSound(int nSoundId);
    void LoadSound(int nSoundId);
    void UnLoadSound(int nSoundId);
    void Set3dListenerAndOrientation(const Vector3& vPosition, const Vector3& vLook, const Vector3& vUp);
    int PlaySound(int nSoundId, const Vector3& vPosition = Vector3{ 0, 0, 0 }, float fVolumedB = 0.0f);
    void StopChannel(int nChannelId, float fFadeTimeSeconds = 0.0f);
    void StopAllChannels();
    void SetChannel3dPosition(int nChannelId, const Vector3& vPosition);
    void SetChannelVolume(int nChannelId, float fVolumedB);
    bool IsPlaying(int nChannelId) const;
};
```

Except

```
class AudioEngine
{
public:
    static void Init();
    static void Update(float fTimeDeltaSeconds);
    static void Shutdown();

    struct SoundDefinition
    {
        std::string mSoundName;
        float fDefaultVolumedB;
        float fMinDistance;
        float fMaxDistance;
        bool bIs3d;
        bool bIsLooping;
        bool bIsStreaming;
    };

    int RegisterSound(const SoundDefinition& tSoundDefinition, bool bLoad = true);
    void UnregisterSound(int nSoundId);
    void LoadSound(int nSoundId);
    void UnLoadSound(int nSoundId);
    void Set3dListenerAndOrientation(const Vector3& vPosition, const Vector3& vLook, const Vector3& vUp);
    int PlaySound(int nSoundId, const Vector3& vPosition = Vector3{ 0, 0, 0 }, float fVolumedB = 0.0f);
    void StopChannel(int nChannelId, float fFadeTimeSeconds = 0.0f);
    void StopAllChannels();
    void SetChannel3dPosition(int nChannelId, const Vector3& vPosition);
    void SetChannelVolume(int nChannelId, float fVolumedB);
    bool IsPlaying(int nChannelId) const;
};
```

Also

Except

Okay, I lied

```
void Implementation::LoadSound(int nSoundId)
{
    if(SoundIsLoaded(nSoundId))
        return;

    auto tFoundIt = mSounds.find(nSoundId);
    if(tFoundIt != mSounds.end())
        return;

    FMOD_MODE eMode = FMOD_NONBLOCKING;
    eMode |= tFoundIt->second->mSoundDefinition.bIs3d ? (FMOD_3D | FMOD_3D_INVERSETAPEREDROLLOFF) : FMOD_2D;
    eMode |= tFoundIt->second->mSoundDefinition.bIsLooping ? FMOD_LOOP_NORMAL : FMOD_LOOP_OFF;
    eMode |= tFoundIt->second->mSoundDefinition.bIsStreaming ? FMOD_CREATESTREAM : FMOD_CREATECOMPRESSEDSAMPLE;

    mpSystem->createSound(tFoundIt->second->mSoundDefinition.mSoundName.c_str(), eMode, nullptr,
                          &tFoundIt->second->mpSound);

    if(tFoundIt->second->mpSound)
    {
        tFoundIt->second->mpSound->set3DMinMaxDistance(tFoundIt->second->mSoundDefinition.fMinDistance,
                                                       tFoundIt->second->mSoundDefinition.fMaxDistance);
    }
}
```

Okay, I lied

```
void Implementation::LoadSound(int nSoundId)
{
    if(SoundIsLoaded(nSoundId))
        return;

    auto tFoundIt = mSounds.find(nSoundId);
    if(tFoundIt != mSounds.end())
        return;

    FMOD_MODE eMode = FMOD_NONBLOCKING;
    eMode |= tFoundIt->second->mSoundDefinition.bIs3d ? (FMOD_3D | FMOD_3D_INVERSETAPEREDROLLOFF) : FMOD_2D;
    eMode |= tFoundIt->second->mSoundDefinition.bIsLooping ? FMOD_LOOP_NORMAL : FMOD_LOOP_OFF;
    eMode |= tFoundIt->second->mSoundDefinition.bIsStreaming ? FMOD_CREATESTREAM : FMOD_CREATECOMPRESSEDSAMPLE;

    mpSystem->createSound(tFoundIt->second->mSoundDefinition.mSoundName.c_str(), eMode, nullptr,
                          &tFoundIt->second->mpSound);

    if(tFoundIt->second->mpSound)
    {
        tFoundIt->second->mpSound->set3DMinMaxDistance(tFoundIt->second->mSoundDefinition.fMinDistance,
                                                       tFoundIt->second->mSoundDefinition.fMaxDistance);
    }
}
```

```cpp
struct Channel
{
    Channel(Implementation& tImplementation, int nSoundId, const AudioEngine::SoundDefinition& tSoundDefinition,
            const Vector3& vPosition, float fVolumedB);

    enum class State
    { INITIALIZE, TOPLAY, LOADING, PLAYING, STOPPING, STOPPED, VIRTUALIZING, VIRTUAL, DEVIRTUALIZE, };

    Implementation& mImplementation;
    FMOD::Channel* mpChannel = nullptr;
    int mSoundId;
    Vector3 mvPosition;
    float mfVolumedB = 0.0f;
    float mfSoundVolume = 0.0f;
    State meState = State::INITIALIZE;
    bool mbStopRequsted = false;
    AudioFader mStopFader;
    AudioFader mVirtualizeFader;

    void Update(float fTimeDeltaSeconds);
    void UpdateChannelParameters();
    bool ShouldBeVirtual(bool bAllowOneShotVirtuals) const;
    bool IsPlaying() const;
    float GetVolumedB() const;
};
```

```
struct Channel
{
    Channel(Implementation& tImplementation, int nSoundId, const AudioEngine::SoundDefinition& tSoundDefinition,
            const Vector3& vPosition, float fVolumedB);

    enum class State
    { INITIALIZE, TOPLAY, LOADING, PLAYING, STOPPING, STOPPED, VIRTUALIZING, VIRTUAL, DEVIRTUALIZE, };

    Implementation& mImplementation;
    FMOD::Channel* mpChannel = nullptr;
    int mSoundId;
    Vector3 mvPosition;
    float mfVolumedB = 0.0f;
    float mfSoundVolume = 0.0f;
    State meState = State::INITIALIZE;
    bool mbStopRequsted = false;
    AudioFader mStopFader;
    AudioFader mVirtualizeFader;

    void Update(float fTimeDeltaSeconds);
    void UpdateChannelParameters();
    bool ShouldBeVirtual(bool bAllowOneShotVirtuals) const;
    bool IsPlaying() const;
    float GetVolumedB() const;
};
```

```cpp
struct Channel
{
    Channel(Implementation& tImplementation, int nSoundId, const AudioEngine::SoundDefinition& tSoundDefinition,
            const Vector3& vPosition, float fVolumedB);

    enum class State
    { INITIALIZE, TOPLAY, LOADING, PLAYING, STOPPING, STOPPED, VIRTUALIZING, VIRTUAL, DEVIRTUALIZE, };

    Implementation& mImplementation;
    FMOD::Channel* mpChannel = nullptr;
    int mSoundId;
    Vector3 mvPosition;
    float mfVolumedB = 0.0f;
    float mfSoundVolume = 0.0f;
    State meState = State::INITIALIZE;
    bool mbStopRequsted = false;
    AudioFader mStopFader;
    AudioFader mVirtualizeFader;

    void Update(float fTimeDeltaSeconds);
    void UpdateChannelParameters();
    bool ShouldBeVirtual(bool bAllowOneShotVirtuals) const;
    bool IsPlaying() const;
    float GetVolumedB() const;
};
```

```
struct Channel
{
    Channel(Implementation& tImplementation, int nSoundId, const AudioEngine::SoundDefinition& tSoundDefinition,
            const Vector3& vPosition, float fVolumedB);

    enum class State
    { INITIALIZE, TOPLAY, LOADING, PLAYING, STOPPING, STOPPED, VIRTUALIZING, VIRTUAL, DEVIRTUALIZE, };

    Implementation& mImplementation;
    FMOD::Channel* mpChannel = nullptr;
    int mSoundId;
    Vector3 mvPosition;
    float mfVolumedB = 0.0f;
    float mfSoundVolume = 0.0f;
    State meState = State::INITIALIZE;
    bool mbStopRequsted = false;
    AudioFader mStopFader;
    AudioFader mVirtualizeFader;

    void Update(float fTimeDeltaSeconds);
    void UpdateChannelParameters();
    bool ShouldBeVirtual(bool bAllowOneShotVirtuals) const;
    bool IsPlaying() const;
    float GetVolumedB() const;
};
```

```cpp
int AudioEngine::PlaySound(int nSoundId, const Vector3& vPosition, float fVolumedB)
{
    int nChannelId = sgpImplementation->mnNextChannelId++;
    auto tSoundIt = sgpImplementation->mSounds.find(nSoundId);
    if(tSoundIt == sgpImplementation->mSounds.end())
        return nChannelId;

    sgpImplementation->mChannels[nChannelId] = make_unique<Implementation::Channel>(*sgpImplementation, nSoundId,
                                                          tSoundIt->second->mSoundDefinition, vPosition, fVolumedB);
    return nChannelId;
}

void AudioEngine::StopChannel(int nChannelId, float fFadeTimeSeconds)
{
    auto tFoundIt = sgpImplementation->mChannels.find(nChannelId);
    if(tFoundIt == sgpImplementation->mChannels.end())
        return;

    if(fFadeTimeSeconds <= 0.0f)
    {
        tFoundIt->second->mpChannel->stop();
    }
    else
    {
        tFoundIt->second->mbStopRequsted = true;
        tFoundIt->second->mStopFader.StartFade(SILENCE_dB, fFadeTimeSeconds);
    }
}
```

```
int AudioEngine::PlaySound(int nSoundId, const Vector3& vPosition, float fVolumedB)
{
    int nChannelId = sgpImplementation->mnNextChannelId++;
    auto tSoundIt = sgpImplementation->mSounds.find(nSoundId);
    if(tSoundIt == sgpImplementation->mSounds.end())
        return nChannelId;

    sgpImplementation->mChannels[nChannelId] = make_unique<Implementation::Channel>(*sgpImplementation, nSoundId,
                                                            tSoundIt->second->mSoundDefinition, vPosition, fVolumedB);
    return nChannelId;
}

void AudioEngine::StopChannel(int nChannelId, float fFadeTimeSeconds)
{
    auto tFoundIt = sgpImplementation->mChannels.find(nChannelId);
    if(tFoundIt == sgpImplementation->mChannels.end())
        return;

    if(fFadeTimeSeconds <= 0.0f)
    {
        tFoundIt->second->mpChannel->stop();
    }
    else
    {
        tFoundIt->second->mbStopRequsted = true;
        tFoundIt->second->mStopFader.StartFade(SILENCE_dB, fFadeTimeSeconds);
    }
}
```

```
int AudioEngine::PlaySound(int nSoundId, const Vector3& vPosition, float fVolumedB)
{
    int nChannelId = sgpImplementation->mnNextChannelId++;
    auto tSoundIt = sgpImplementation->mSounds.find(nSoundId);
    if(tSoundIt == sgpImplementation->mSounds.end())
        return nChannelId;

    sgpImplementation->mChannels[nChannelId] = make_unique<Implementation::Channel>(*sgpImplementation, nSoundId,
                                                           tSoundIt->second->mSoundDefinition, vPosition, fVolumedB);
    return nChannelId;
}

void AudioEngine::StopChannel(int nChannelId, float fFadeTimeSeconds)
{
    auto tFoundIt = sgpImplementation->mChannels.find(nChannelId);
    if(tFoundIt == sgpImplementation->mChannels.end())
        return;

    if(fFadeTimeSeconds <= 0.0f)
    {
        tFoundIt->second->mpChannel->stop();
    }
    else
    {
        tFoundIt->second->mbStopRequsted = true;
        tFoundIt->second->mStopFader.StartFade(SILENCE_dB, fFadeTimeSeconds);
    }
}
```

```
int AudioEngine::PlaySound(int nSoundId, const Vector3& vPosition, float fVolumedB)
{
    int nChannelId = sgpImplementation->mnNextChannelId++;
    auto tSoundIt = sgpImplementation->mSounds.find(nSoundId);
    if(tSoundIt == sgpImplementation->mSounds.end())
        return nChannelId;

    sgpImplementation->mChannels[nChannelId] = make_unique<Implementation::Channel>(*sgpImplementation, nSoundId,
                                                        tSoundIt->second->mSoundDefinition, vPosition, fVolumedB);
    return nChannelId;
}

void AudioEngine::StopChannel(int nChannelId, float fFadeTimeSeconds)
{
    auto tFoundIt = sgpImplementation->mChannels.find(nChannelId);
    if(tFoundIt == sgpImplementation->mChannels.end())
        return;

    if(fFadeTimeSeconds <= 0.0f)
    {
        tFoundIt->second->mpChannel->stop();
    }
    else
    {
        tFoundIt->second->mbStopRequsted = true;
        tFoundIt->second->mStopFader.StartFade(SILENCE_dB, fFadeTimeSeconds);
    }
}
```

```
void AudioEngine::SetChannelXXX(int nChannelId, Blah xxxValue)
{
    auto tFoundIt = sgpImplementation->mChannels.find(nChannelId);
    if(tFoundIt == sgpImplementation->mChannels.end())
        return;

    tFoundIt->second->XXX = xxxValue;
}

// For example
void AudioEngine::SetChannelVolume(int nChannelId, float fVolumedB)
{
    auto tFoundIt = sgpImplementation->mChannels.find(nChannelId);
    if(tFoundIt == sgpImplementation->mChannels.end())
        return;

    tFoundIt->second->mfVolumedB = fVolumedB;
}
```

```
void AudioEngine::SetChannelXXX(int nChannelId, Blah xxxValue)
{
    auto tFoundIt = sgpImplementation->mChannels.find(nChannelId);
    if(tFoundIt == sgpImplementation->mChannels.end())
        return;

    tFoundIt->second->XXX = xxxValue;
}
```

Boring

```
// For example
void AudioEngine::SetChannelVolume(int nChannelId, float fVolumedB)
{
    auto tFoundIt = sgpImplementation->mChannels.find(nChannelId);
    if(tFoundIt == sgpImplementation->mChannels.end())
        return;

    tFoundIt->second->mfVolumedB = fVolumedB;
}
```

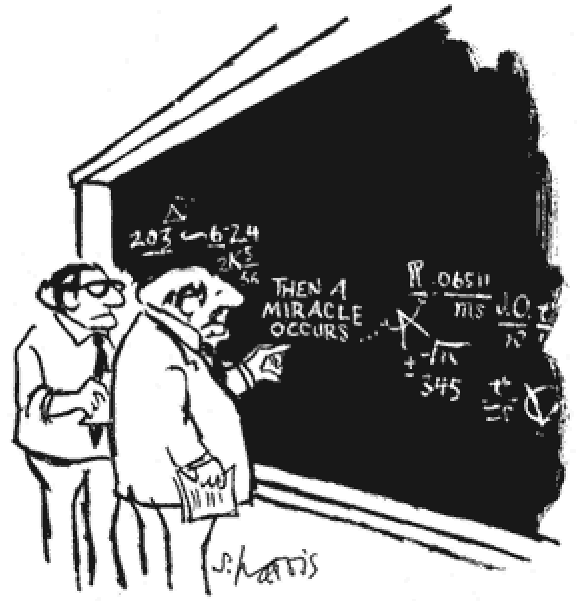
```
void AudioEngine::SetChannelXXX(int nChannelId, Blah xxxValue)
{
    auto tFoundIt = sgpImplementation->mChannels.find(nChannelId);
    if(tFoundIt == sgpImplementation->mChannels.end())
        return;

    tFoundIt->second->XXX = xxxValue;
}

// For example
void AudioEngine::SetChannelVolume(int nChannelId, float fVolumedB)
{
    auto tFoundIt = sgpImplementation->mChannels.find(nChannelId);
    if(tFoundIt == sgpImplementation->mChannels.end())
        return;

    tFoundIt->second->mfVolumedB = fVolumedB;
}
```

```
void Implementation::Update(float fTimeDeltaSeconds)
{
    vector<ChannelMap::iterator> pStoppedChannels;
    for(auto it = mChannels.begin(), itEnd = mChannels.end(); it != itEnd; ++it)
    {
        it->second->Update(fTimeDeltaSeconds);
        if(it->second->meState == Channel::State::STOPPED)
        {
            pStoppedChannels.push_back(it);
        }
    }

    for(auto& it : pStoppedChannels)
    {
        mChannels.erase(it);
    }

    mpSystem->update();
}
```

```
void Implementation::Update(float fTimeDeltaSeconds)
{
    vector<ChannelMap::iterator> pStoppedChannels;
    for(auto it = mChannels.begin(), itEnd = mChannels.end(); it != itEnd; ++it)
    {
        it->second->Update(fTimeDeltaSeconds);
        if(it->second->meState == Channel::State::STOPPED)
        {
            pStoppedChannels.push_back(it);
        }
    }

    for(auto& it : pStoppedChannels)
    {
        mChannels.erase(it);
    }

    mpSystem->update();
}
```

```
void Implementation::Channel::Update(float fTimeDeltaSeconds)
{
    switch(meState)
    {








    }
}
```

```
void Implementation::Channel::Update(float fTimeDeltaSeconds)
{
    switch(meState)
    {
```



"I THINK YOU SHOULD BE MORE EXPLICIT HERE IN STEP TWO."

```
    }
}
```

```
case Implementation::Channel::State::INITIALIZE:
    [[fallthrough]];
case Implementation::Channel::State::DEVIRTUALIZE:
case Implementation::Channel::State::TOPLAY:
{
  if(mbStopRequsted) {
    meState = State::STOPPING;
    return;
  }

  if(ShouldBeVirtual(true)) {
    if(IsOneShot()) {
      meState = State::STOPPING;
    else {
      meState = State::VIRTUAL;
    }
    return;
  }

  if(!mImplementation.SoundIsLoaded(mSoundId)) {
    mImplementation.LoadSound(mSoundId);
    meState = State::LOADING;
    return;
  }
```

```
  mpChannel = nullptr;

  auto tSoundIt = mImplementation.mSounds.find(mSoundId);
  if(tSoundIt != mImplementation.mSounds.end())
    mImplementation.mpSystem->playSound(
        tSoundIt->second->mpSound,
        nullptr, true, &mpChannel);

  if(mpChannel) {
    if(meState == State::DEVIRTUALIZE)
      mVirtualizeFader.StartFade(SILENCE_dB, 0.0f,
                                           VIRTUALIZE_FADE_TIME);

    meState = State::PLAYING;

    FMOD_VECTOR position = VectorToFmod(mvPosition);
    mpChannel->set3DAttributes(&position, nullptr);
    mpChannel->setVolume(dBToVolume(GetVolumedB()));
    mpChannel->setPaused(false);
  }
  else
  {
    meState = State::STOPPING;
  }
}
break;
```

```
case Implementation::Channel::State::INITIALIZE:
    [[fallthrough]];
case Implementation::Channel::State::DEVIRTUALIZE:
case Implementation::Channel::State::TOPLAY:
{
  if(mbStopRequsted) {
    meState = State::STOPPING;
    return;
  }

  if(ShouldBeVirtual(true)) {
    if(IsOneShot()) {
      meState = State::STOPPING;
    else {
      meState = State::VIRTUAL;
    }
    return;
  }

  if(!mImplementation.SoundIsLoaded(mSoundId)) {
    mImplementation.LoadSound(mSoundId);
    meState = State::LOADING;
    return;
  }
```

```
  mpChannel = nullptr;

  auto tSoundIt = mImplementation.mSounds.find(mSoundId);
  if(tSoundIt != mImplementation.mSounds.end())
    mImplementation.mpSystem->playSound(
        tSoundIt->second->mpSound,
        nullptr, true, &mpChannel);

  if(mpChannel) {
    if(meState == State::DEVIRTUALIZE)
      mVirtualizeFader.StartFade(SILENCE_dB, 0.0f,
                                           VIRTUALIZE_FADE_TIME);

    meState = State::PLAYING;

    FMOD_VECTOR position = VectorToFmod(mvPosition);
    mpChannel->set3DAttributes(&position, nullptr);
    mpChannel->setVolume(dBToVolume(GetVolumedB()));
    mpChannel->setPaused(false);
  }
  else
  {
    meState = State::STOPPING;
  }
}
break;
```

```
case Implementation::Channel::State::INITIALIZE:
    [[fallthrough]];
case Implementation::Channel::State::DEVIRTUALIZE:
case Implementation::Channel::State::TOPLAY:
{
  if(mbStopRequsted) {
    meState = State::STOPPING;
    return;
  }

  if(ShouldBeVirtual(true)) {
    if(IsOneShot()) {
      meState = State::STOPPING;
    else {
      meState = State::VIRTUAL;
    }
    return;
  }

  if(!mImplementation.SoundIsLoaded(mSoundId)) {
    mImplementation.LoadSound(mSoundId);
    meState = State::LOADING;
    return;
  }
```

```
  mpChannel = nullptr;

  auto tSoundIt = mImplementation.mSounds.find(mSoundId);
  if(tSoundIt != mImplementation.mSounds.end())
    mImplementation.mpSystem->playSound(
        tSoundIt->second->mpSound,
        nullptr, true, &mpChannel);

  if(mpChannel) {
    if(meState == State::DEVIRTUALIZE)
      mVirtualizeFader.StartFade(SILENCE_dB, 0.0f,
                                           VIRTUALIZE_FADE_TIME);

    meState = State::PLAYING;

    FMOD_VECTOR position = VectorToFmod(mvPosition);
    mpChannel->set3DAttributes(&position, nullptr);
    mpChannel->setVolume(dBToVolume(GetVolumedB()));
    mpChannel->setPaused(false);
  }
  else
  {
    meState = State::STOPPING;
  }
}
break;
```

```cpp
case Implementation::Channel::State::INITIALIZE:
    [[fallthrough]];
case Implementation::Channel::State::DEVIRTUALIZE:
case Implementation::Channel::State::TOPLAY:
{
  if(mbStopRequsted) {
    meState = State::STOPPING;
    return;
  }

  if(ShouldBeVirtual(true)) {
    if(IsOneShot()) {
      meState = State::STOPPING;
    else {
      meState = State::VIRTUAL;
    }
    return;
  }

  if(!mImplementation.SoundIsLoaded(mSoundId)) {
    mImplementation.LoadSound(mSoundId);
    meState = State::LOADING;
    return;
  }
```

```cpp
  mpChannel = nullptr;

  auto tSoundIt = mImplementation.mSounds.find(mSoundId);
  if(tSoundIt != mImplementation.mSounds.end())
    mImplementation.mpSystem->playSound(
        tSoundIt->second->mpSound,
        nullptr, true, &mpChannel);

  if(mpChannel) {
    if(meState == State::DEVIRTUALIZE)
      mVirtualizeFader.StartFade(SILENCE_dB, 0.0f,
                                           VIRTUALIZE_FADE_TIME);

    meState = State::PLAYING;

    FMOD_VECTOR position = VectorToFmod(mvPosition);
    mpChannel->set3DAttributes(&position, nullptr);
    mpChannel->setVolume(dBToVolume(GetVolumedB()));
    mpChannel->setPaused(false);
  }
  else
  {
    meState = State::STOPPING;
  }
}
break;
```

```
case Implementation::Channel::State::INITIALIZE:
    [[fallthrough]];
case Implementation::Channel::State::DEVIRTUALIZE:
case Implementation::Channel::State::TOPLAY:
{
  if(mbStopRequsted) {
    meState = State::STOPPING;
    return;
  }

  if(ShouldBeVirtual(true)) {
    if(IsOneShot()) {
      meState = State::STOPPING;
    else {
      meState = State::VIRTUAL;
    }
    return;
  }

  if(!mImplementation.SoundIsLoaded(mSoundId)) {
    mImplementation.LoadSound(mSoundId);
    meState = State::LOADING;
    return;
  }
```

```
  mpChannel = nullptr;

  auto tSoundIt = mImplementation.mSounds.find(mSoundId);
  if(tSoundIt != mImplementation.mSounds.end())
    mImplementation.mpSystem->playSound(
        tSoundIt->second->mpSound,
        nullptr, true, &mpChannel);

  if(mpChannel) {
    if(meState == State::DEVIRTUALIZE)
      mVirtualizeFader.StartFade(SILENCE_dB, 0.0f,
                                 VIRTUALIZE_FADE_TIME);

    meState = State::PLAYING;

    FMOD_VECTOR position = VectorToFmod(mvPosition);
    mpChannel->set3DAttributes(&position, nullptr);
    mpChannel->setVolume(dBToVolume(GetVolumedB()));
    mpChannel->setPaused(false);
  }
  else
  {
    meState = State::STOPPING;
  }
}
break;
```

```cpp
case Implementation::Channel::State::INITIALIZE:
    [[fallthrough]];
case Implementation::Channel::State::DEVIRTUALIZE:
case Implementation::Channel::State::TOPLAY:
{
  if(mbStopRequsted) {
    meState = State::STOPPING;
    return;
  }

  if(ShouldBeVirtual(true)) {
    if(IsOneShot()) {
      meState = State::STOPPING;
    else {
      meState = State::VIRTUAL;
    }
    return;
  }

  if(!mImplementation.SoundIsLoaded(mSoundId)) {
    mImplementation.LoadSound(mSoundId);
    meState = State::LOADING;
    return;
  }
```

```cpp
  mpChannel = nullptr;

  auto tSoundIt = mImplementation.mSounds.find(mSoundId);
  if(tSoundIt != mImplementation.mSounds.end())
    mImplementation.mpSystem->playSound(
        tSoundIt->second->mpSound,
        nullptr, true, &mpChannel);

  if(mpChannel) {
    if(meState == State::DEVIRTUALIZE)
      mVirtualizeFader.StartFade(SILENCE_dB, 0.0f,
                                             VIRTUALIZE_FADE_TIME);

    meState = State::PLAYING;

    FMOD_VECTOR position = VectorToFmod(mvPosition);
    mpChannel->set3DAttributes(&position, nullptr);
    mpChannel->setVolume(dBToVolume(GetVolumedB()));
    mpChannel->setPaused(false);
  }
  else
  {
    meState = State::STOPPING;
  }
}
break;
```

```
case Implementation::Channel::State::INITIALIZE:
  [[fallthrough]];
case Implementation::Channel::State::DEVIRTUALIZE:
case Implementation::Channel::State::TOPLAY:
{
 if(mbStopRequsted) {
  meState = State::STOPPING;
  return;
 }

 if(ShouldBeVirtual(true)) {
  if(IsOneShot()) {
   meState = State::STOPPING;
  else {
   meState = State::VIRTUAL;
  }
  return;
 }

 if(!mImplementation.SoundIsLoaded(mSoundId)) {
  mImplementation.LoadSound(mSoundId);
  meState = State::LOADING;
  return;
 }
 mpChannel = nullptr;

 auto tSoundIt = mImplementation.mSounds.find(mSoundId);
 if(tSoundIt != mImplementation.mSounds.end())
  mImplementation.mpSystem->playSound(
    tSoundIt->second->mpSound,
    nullptr, true, &mpChannel);

 if(mpChannel) {
  if(meState == State::DEVIRTUALIZE)
   mVirtualizeFader.StartFade(SILENCE_dB, 0.0f,
                 VIRTUALIZE_FADE_TIME);

  meState = State::PLAYING;

  FMOD_VECTOR position = VectorToFmod(mvPosition);
  mpChannel->set3DAttributes(&position, nullptr);
  mpChannel->setVolume(dBToVolume(GetVolumedB()));
  mpChannel->setPaused(false);
 }
 else
 {
  meState = State::STOPPING;
 }
}
break;
```

```
case Implementation::Channel::State::LOADING:
    if(mImplementation.SoundIsLoaded(mSoundId))
    {
        meState = State::TOPLAY;
    }
    break;

case Implementation::Channel::State::PLAYING:
    mVirtualizeFader.Update(fTimeDeltaSeconds);
    UpdateChannelParameters();

    if(!IsPlaying() || mbStopRequsted)
    {
        meState = State::STOPPING;
        return;
    }

    if(ShouldBeVirtual(false))
    {
        mVirtualizeFader.StartFade(SILENCE_dB, VIRTUALIZE_FADE_TIME);
        meState = State::VIRTUALIZING;
    }
    break;
```

```
case Implementation::Channel::State::STOPPING:
    mStopFader.Update(fTimeDeltaSeconds);
    UpdateChannelParameters();
    if(mStopFader.IsFinished())
    {
        mpChannel->stop();
    }
    if(!IsPlaying())
    {
        meState = State::STOPPED;
        return;
    }
    break;

case Implementation::Channel::State::STOPPED: break;
case Implementation::Channel::State::LOADING:
    if(mImplementation.SoundIsLoaded(mSoundId))
    {
        meState = State::TOPLAY;
    }
    break;

case Implementation::Channel::State::PLAYING:
    mVirtualizeFader.Update(fTimeDeltaSeconds);
    UpdateChannelParameters();

    if(!IsPlaying() || mbStopRequsted)
    {
        meState = State::STOPPING;
        return;
    }

    if(ShouldBeVirtual(false))
    {
        mVirtualizeFader.StartFade(SILENCE_dB, VIRTUALIZE_FADE_TIME);
        meState = State::VIRTUALIZING;
    }
    break;
```

```
case Implementation::Channel::State::STOPPING:
    mStopFader.Update(fTimeDeltaSeconds);
    UpdateChannelParameters();
    if(mStopFader.IsFinished())
    {
        mpChannel->stop();
    }
    if(!IsPlaying())
    {
        meState = State::STOPPED;
        return;
    }
    break;

case Implementation::Channel::State::STOPPED: break;
case Implementation::Channel::State::LOADING:
    if(mImplementation.SoundIsLoaded(mSoundId))
    {
        meState = State::TOPLAY;
    }
    break;

case Implementation::Channel::State::PLAYING:
    mVirtualizeFader.Update(fTimeDeltaSeconds);
    UpdateChannelParameters();

    if(!IsPlaying() || mbStopRequsted)
    {
        meState = State::STOPPING;
        return;
    }

    if(ShouldBeVirtual(false))
    {
        mVirtualizeFader.StartFade(SILENCE_dB, VIRTUALIZE_FADE_TIME);
        meState = State::VIRTUALIZING;
    }
    break;
```

```
case Implementation::Channel::State::STOPPING:
    mStopFader.Update(fTimeDeltaSeconds);
    UpdateChannelParameters();
    if(mStopFader.IsFinished())
    {
        mpChannel->stop();
    }
    if(!IsPlaying())
    {
        meState = State::STOPPED;
        return;
    }
    break;

case Implementation::Channel::State::STOPPED: break;
case Implementation::Channel::State::LOADING:
    if(mImplementation.SoundIsLoaded(mSoundId))
    {
        meState = State::TOPLAY;
    }
    break;

case Implementation::Channel::State::PLAYING:
    mVirtualizeFader.Update(fTimeDeltaSeconds);
    UpdateChannelParameters();

    if(!IsPlaying() || mbStopRequsted)
    {
        meState = State::STOPPING;
        return;
    }

    if(ShouldBeVirtual(false))
    {
        mVirtualizeFader.StartFade(SILENCE_dB, VIRTUALIZE_FADE_TIME);
        meState = State::VIRTUALIZING;
    }
    break;
```

```
case Implementation::Channel::State::STOPPING:
    mStopFader.Update(fTimeDeltaSeconds);
    UpdateChannelParameters();
    if(mStopFader.IsFinished())
    {
        mpChannel->stop();
    }
    if(!IsPlaying())
    {
        meState = State::STOPPED;
        return;
    }
    break;

case Implementation::Channel::State::STOPPED: break;
case Implementation::Channel::State::LOADING:
    if(mImplementation.SoundIsLoaded(mSoundId))
    {
        meState = State::TOPLAY;
    }
    break;

case Implementation::Channel::State::PLAYING:
    mVirtualizeFader.Update(fTimeDeltaSeconds);
    UpdateChannelParameters();

    if(!IsPlaying() || mbStopRequsted)
    {
        meState = State::STOPPING;
        return;
    }

    if(ShouldBeVirtual(false))
    {
        mVirtualizeFader.StartFade(SILENCE_dB, VIRTUALIZE_FADE_TIME);
        meState = State::VIRTUALIZING;
    }
    break;
```

```
case Implementation::Channel::State::STOPPING:
    mStopFader.Update(fTimeDeltaSeconds);
    UpdateChannelParameters();
    if(mStopFader.IsFinished())
    {
        mpChannel->stop();
    }
    if(!IsPlaying())
    {
        meState = State::STOPPED;
        return;
    }
    break;

case Implementation::Channel::State::STOPPED: break;
case Implementation::Channel::State::LOADING:
    if(mImplementation.SoundIsLoaded(mSoundId))
    {
        meState = State::TOPLAY;
    }
    break;

case Implementation::Channel::State::PLAYING:
    mVirtualizeFader.Update(fTimeDeltaSeconds);
    UpdateChannelParameters();

    if(!IsPlaying() || mbStopRequsted)
    {
        meState = State::STOPPING;
        return;
    }

    if(ShouldBeVirtual(false))
    {
        mVirtualizeFader.StartFade(SILENCE_dB, VIRTUALIZE_FADE_TIME);
        meState = State::VIRTUALIZING;
    }
    break;
```

```
case Implementation::Channel::State::STOPPING:
    mStopFader.Update(fTimeDeltaSeconds);
    UpdateChannelParameters();
    if(mStopFader.IsFinished())
    {
        mpChannel->stop();
    }
    if(!IsPlaying())
    {
        meState = State::STOPPED;
        return;
    }
    break;

case Implementation::Channel::State::STOPPED: break;
case Implementation::Channel::State::LOADING:
    if(mImplementation.SoundIsLoaded(mSoundId))
    {
        meState = State::TOPLAY;
    }
    break;

case Implementation::Channel::State::PLAYING:
    mVirtualizeFader.Update(fTimeDeltaSeconds);
    UpdateChannelParameters();

    if(!IsPlaying() || mbStopRequsted)
    {
        meState = State::STOPPING;
        return;
    }

    if(ShouldBeVirtual(false))
    {
        mVirtualizeFader.StartFade(SILENCE_dB, VIRTUALIZE_FADE_TIME);
        meState = State::VIRTUALIZING;
    }
    break;
```

```
case Implementation::Channel::State::STOPPING:
    mStopFader.Update(fTimeDeltaSeconds);
    UpdateChannelParameters();
    if(mStopFader.IsFinished())
    {
        mpChannel->stop();
    }
    if(!IsPlaying())
    {
        meState = State::STOPPED;
        return;
    }
    break;

case Implementation::Channel::State::STOPPED: break;
case Implementation::Channel::State::LOADING:
    if(mImplementation.SoundIsLoaded(mSoundId))
    {
        meState = State::TOPLAY;
    }
    break;

case Implementation::Channel::State::PLAYING:
    mVirtualizeFader.Update(fTimeDeltaSeconds);
    UpdateChannelParameters();

    if(!IsPlaying() || mbStopRequsted)
    {
        meState = State::STOPPING;
        return;
    }

    if(ShouldBeVirtual(false))
    {
        mVirtualizeFader.StartFade(SILENCE_dB, VIRTUALIZE_FADE_TIME);
        meState = State::VIRTUALIZING;
    }
    break;
```

```
case Implementation::Channel::State::STOPPING:
    mStopFader.Update(fTimeDeltaSeconds);
    UpdateChannelParameters();
    if(mStopFader.IsFinished())
    {
        mpChannel->stop();
    }
    if(!IsPlaying())
    {
        meState = State::STOPPED;
        return;
    }
    break;

case Implementation::Channel::State::STOPPED: break;
case Implementation::Channel::State::VIRTUALIZING:
    mVirtualizeFader.Update(fTimeDeltaSeconds);
    UpdateChannelParameters();
    if(!ShouldBeVirtual(false))
    {
        mVirtualizeFader.StartFade(0.0f, VIRTUALIZE_FADE_TIME);
        meState = State::PLAYING;
        break;
    }
    if(mVirtualizeFader.IsFinished())
    {
        mpChannel->stop();
        meState = State::VIRTUAL;
    }
    break;

case Implementation::Channel::State::VIRTUAL:
    if(mbStopRequsted)
    {
        meState = State::STOPPING;
    }
    else if(!ShouldBeVirtual(false))
    {
        meState = State::DEVIRTUALIZE;
    }
    break;
```

Summary

- Line count: ~600 LOC
  - Almost all state machine logic
- Features:
  - Sound playback in 3D
  - Volume control
  - Jukebox functions
  - Async file I/O
  - Virtualization
  - Fadeouts
  - Hooks for more features

std::audio?

std::audio?

- The standard can’t replace FMOD/Wwise/ADX2

std::audio?

- The standard can’t replace FMOD/Wwise/ADX2
    - It shouldn’t!

std::audio?

- The standard can’t replace FMOD/Wwise/ADX2
  - It shouldn’t!
- But maybe std::audio can provide a standard way to communicate with the audio device

What the Standard Says about Audio

What the Standard Says about Audio

Toward a Standard C++ Audio Library

Toward a Standard C++ Audio Library

- Why?

Toward a Standard C++ Audio Library

- Why?
  - P0669R0 “Why We Should Standardize 2D Graphics for C++”

Toward a Standard C++ Audio Library

- Why?
  - P0669R0 “Why We Should Standardize 2D Graphics for C++”
- “Game devs won’t use it”

Toward a Standard C++ Audio Library

- Why?
  - P0669R0 “Why We Should Standardize 2D Graphics for C++”
- “Game devs won’t use it”
  - Some games will, and games are not the only customers.

Toward a Standard C++ Audio Library

- Why?
  - P0669R0 “Why We Should Standardize 2D Graphics for C++”
- “Game devs won’t use it”
  - Some games will, and games are not the only customers.
- “Widely-used libraries already solve this”

Toward a Standard C++ Audio Library

- Why?
  - P0669R0 “Why We Should Standardize 2D Graphics for C++”
- “Game devs won’t use it”
  - Some games will, and games are not the only customers.
- “Widely-used libraries already solve this”
  - Exactly! The standard is supposed to standardize existing practice.

Abstractions

- device
- voice
- source
  - buffer
  - file_stream
  - synth
- submix
- effect/effect_instance

Device

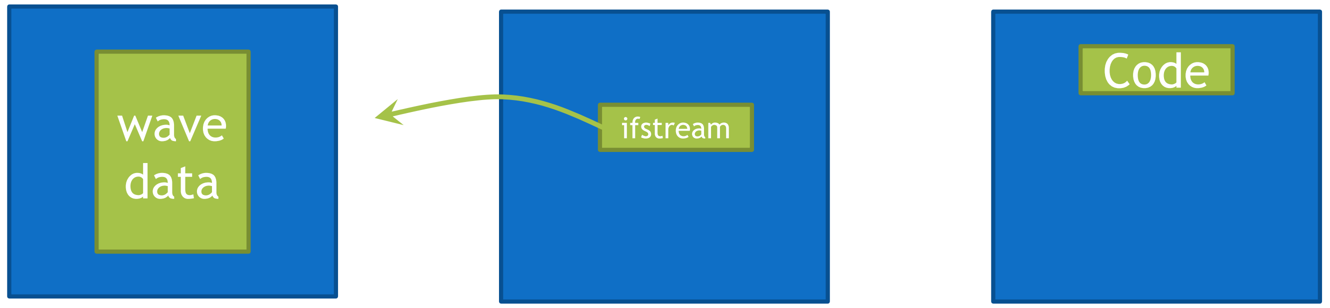
- Outputs wave data to a sound driver
- Most PCs have more than one!
  - Stereo output
  - Optical output
  - Bluetooth headphones
- Null driver for computers with no audio out

Voice

- Currently-playing sound
- Functions to get/set:
  - Volume
  - Pitch
  - Left/Right panning
  - Mute/Pause
  - Playback seek position
  - Etc.

Source

- Abstract base class
- Three built-in implementations:
  - buffer: An in-memory buffer of audio data
  - file_stream: A pointer to a file that is streamed into a buffer
  - synth: A sound synthesizer; abstract base class



Effect/EffectInstance

- An object that can apply an effect to playing audio. E.g.
  - Low-Pass Filter (or High-Pass or Band-Pass)
  - Equalizer
  - Reverb
  - Delay
- effect: Abstract base class
- effect_instance: Applies an effect to a voice or a submix

Submix

- Takes voices and submixes as inputs
- Mixed result as output
- Can apply effects

My Favorite C++ 10*-liner

```cpp
#include <audio>
#include <thread>
#include <chrono>

using namespace std::experimental::audio;
using namespace std::literals::chrono_literals;

int main() {
  device audio_device;
  auto tada = load_from_disk(R"(C:\Windows\Media\tada.wav)");
  auto voice = audio_device.play_sound(tada);
  while (voice->is_playing()) {
    std::this_thread::sleep_for(100ms);
  }
  return 0;
}
```

```
class LowPassFilter : public effect
{
  void process(float* buffer_in, float* buffer_out,
               size_t length_samples, int num_channels) override
  {
    const float RC = 1.0f / (1000.0f * 2 * 3.14f);
    const float dt = 1.0f / 48000.0f;
    const float alpha = dt / (RC + dt);
    for (int i = 0; i < num_channels; i++) {
      buffer_out[i] = buffer_in[i];
    }
    for (int i=num_channels; i<(length_samples*num_channels); i+=num_channels) {
      for (int j = 0; j < num_channels; j++) {
        int current = i + j;
        int previous = i + j - num_channels;
        buffer_out[current] = buffer_out[previous] +
                              (alpha*(buffer_in[current] - buffer_out[previous]));
      }
    }
  }
};
```

https://www.quora.com/Whats-the-C-coding-for-a-low-pass-filter

```
class LowPassFilter : public effect
{
  void process(float* buffer_in, float* buffer_out,
               size_t length_samples, int num_channels) override
  {
    const float RC = 1.0f / (1000.0f * 2 * 3.14f);
    const float dt = 1.0f / 48000.0f;
    const float alpha = dt / (RC + dt);
    for (int i = 0; i < num_channels; i++) {
      buffer_out[i] = buffer_in[i];
    }
    for (int i=num_channels; i<(length_samples*num_channels); i+=num_channels) {
      for (int j = 0; j < num_channels; j++) {
        int current = i + j;
        int previous = i + j - num_channels;
        buffer_out[current] = buffer_out[previous] +
                              (alpha*(buffer_in[current] - buffer_out[previous]));
      }
    }
  }
};
```

https://www.quora.com/Whats-the-C-coding-for-a-low-pass-filter

```
class LowPassFilter : public effect
{
  void process(float* buffer_in, float* buffer_out,
               size_t length_samples, int num_channels) override
  {
    const float RC = 1.0f / (1000.0f * 2 * 3.14f);
    const float dt = 1.0f / 48000.0f;
    const float alpha = dt / (RC + dt);
    for (int i = 0; i < num_channels; i++) {
      buffer_out[i] = buffer_in[i];
    }
    for (int i=num_channels; i<(length_samples*num_channels); i+=num_channels) {
      for (int j = 0; j < num_channels; j++) {
        int current = i + j;
        int previous = i + j - num_channels;
        buffer_out[current] = buffer_out[previous] +
                            (alpha*(buffer_in[current] - buffer_out[previous]));
      }
    }
  }
};
```

https://www.quora.com/Whats-the-C-coding-for-a-low-pass-filter

```
class LowPassFilter : public effect
{
  void process(float* buffer_in, float* buffer_out,
               size_t length_samples, int num_channels) override
  {
    const float RC = 1.0f / (1000.0f * 2 * 3.14f);
    const float dt = 1.0f / 48000.0f;
    const float alpha = dt / (RC + dt);
    for (int i = 0; i < num_channels; i++) {
      buffer_out[i] = buffer_in[i];
    }
    for (int i=num_channels; i<(length_samples*num_channels); i+=num_channels) {
      for (int j = 0; j < num_channels; j++) {
        int current = i + j;
        int previous = i + j - num_channels;
        buffer_out[current] = buffer_out[previous] +
                    (alpha*(buffer_in[current] - buffer_out[previous]));
      }
    }
  }
};
```

https://www.quora.com/Whats-the-C-coding-for-a-low-pass-filter

Adding a Low-Pass Filter

```
int main() {
  device audio_device;
  auto tada = load_from_disk(R"(C:\Windows\Media\tada.wav)");
  auto voice = audio_device.play_sound(tada);
  while (voice->is_playing()) {
    std::this_thread::sleep_for(100ms);
  }
  return 0;
}
```

Adding a Low-Pass Filter

```
int main() {
  device audio_device;
  auto tada = load_from_disk(R"(C:\Windows\Media\tada.wav)");
  auto voice = audio_device.play_sound(tada);
  voice->add_effect<LowPassFilter>();
  while (voice->is_playing()) {
    std::this_thread::sleep_for(100ms);
  }
  return 0;
}
```

Setting up Submixes

```
auto master = audio_device.create_submix();

auto sfx = audio_device.create_submix();
auto music = audio_device.create_submix();
auto ambience = audio_device.create_submix();
auto vox = audio_device.create_submix();

sfx->assign_to_submix(*master);
music->assign_to_submix(*master);
ambience->assign_to_submix(*master);
vox->assign_to_submix(*master);

sfx->set_volume(0.0625f);
ambience->add_effect<LowPassFilter>();
```

Setting up Submixes

```
auto master = audio_device.create_submix();

auto sfx = audio_device.create_submix();
auto music = audio_device.create_submix();
auto ambience = audio_device.create_submix();
auto vox = audio_device.create_submix();

sfx->assign_to_submix(*master);
music->assign_to_submix(*master);
ambience->assign_to_submix(*master);
vox->assign_to_submix(*master);

sfx->set_volume(0.0625f);
ambience->add_effect<LowPassFilter>();
```

Setting up Submixes

```
auto master = audio_device.create_submix();

auto sfx = audio_device.create_submix();
auto music = audio_device.create_submix();
auto ambience = audio_device.create_submix();
auto vox = audio_device.create_submix();

sfx->assign_to_submix(*master);
music->assign_to_submix(*master);
ambience->assign_to_submix(*master);
vox->assign_to_submix(*master);

sfx->set_volume(0.0625f);
ambience->add_effect<LowPassFilter>();
```

Setting up Submixes

```
auto master = audio_device.create_submix();

auto sfx = audio_device.create_submix();
auto music = audio_device.create_submix();
auto ambience = audio_device.create_submix();
auto vox = audio_device.create_submix();

sfx->assign_to_submix(*master);
music->assign_to_submix(*master);
ambience->assign_to_submix(*master);
vox->assign_to_submix(*master);

sfx->set_volume(0.0625f);
ambience->add_effect<LowPassFilter>();
```

Setting up Submixes

```
auto master = audio_device.create_submix();

auto sfx = audio_device.create_submix();
auto music = audio_device.create_submix();
auto ambience = audio_device.create_submix();
auto vox = audio_device.create_submix();

sfx->assign_to_submix(*master);
music->assign_to_submix(*master);
ambience->assign_to_submix(*master);
vox->assign_to_submix(*master);

sfx->set_volume(0.0625f);
ambience->add_effect<LowPassFilter>();
```

Playing Through a Submix

```
int main() {
  device audio_device;
  auto tada = load_from_disk(R"(C:\Windows\Media\tada.wav)");
  auto voice = audio_device.play_sound(tada);
  while (voice->is_playing()) {
    std::this_thread::sleep_for(100ms);
  }
  return 0;
}
```

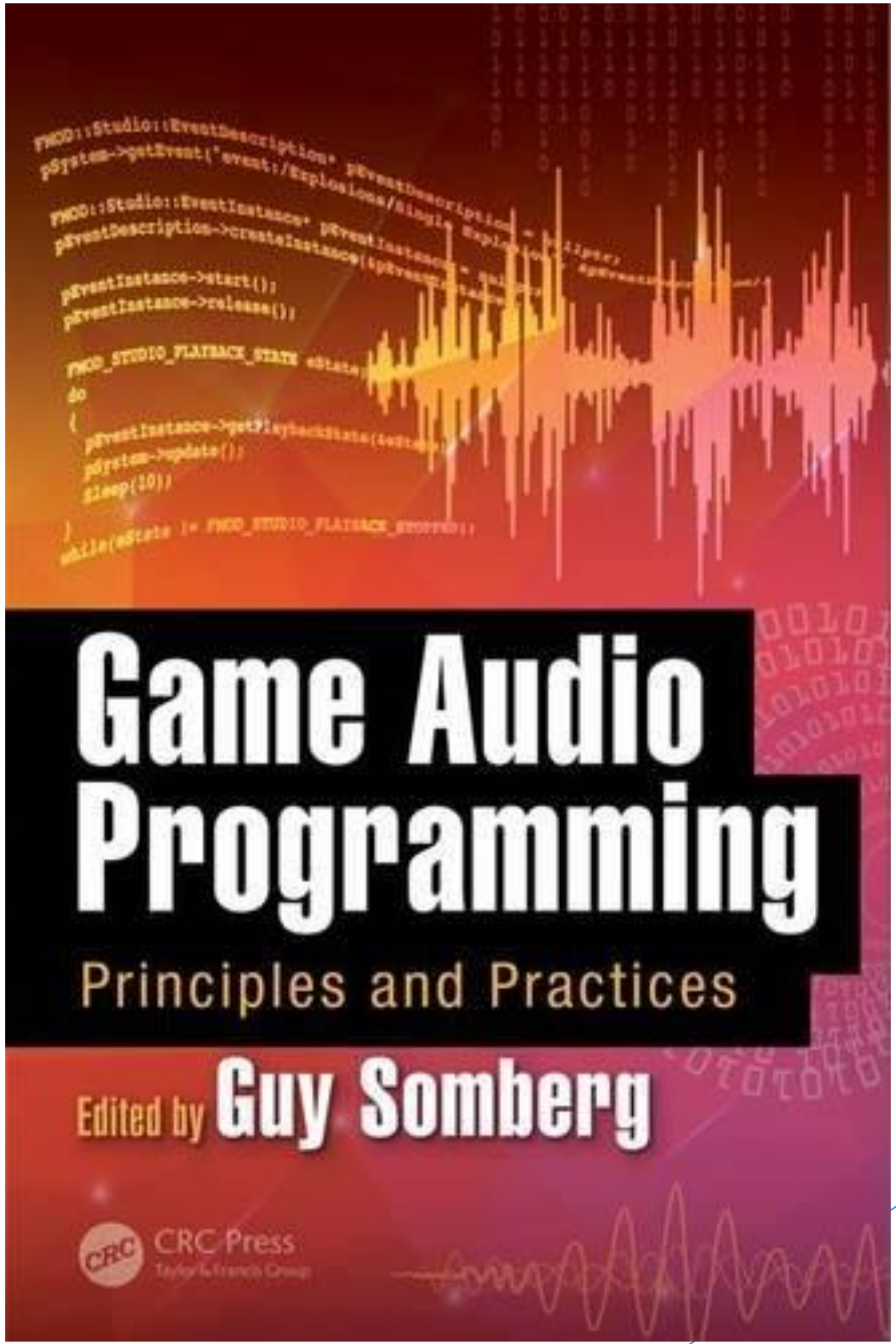
Playing Through a Submix

```
int main() {
  device audio_device;
  auto tada = load_from_disk(R"(C:\Windows\Media\tada.wav)");
  auto voice = audio_device.play_sound(tada);
  voice->assign_to_submix(*sfx);
  while (voice->is_playing()) {
    std::this_thread::sleep_for(100ms);
  }
  return 0;
}
```

Playing Through a Submix

```
int main() {
  device audio_device;
  auto tada = load_from_disk(R"(C:\Windows\Media\tada.wav)");
  auto voice = audio_device.play_sound(tada);
  voice->assign_to_submix(*ambience);
  while (voice->is_playing()) {
    std::this_thread::sleep_for(100ms);
  }
  return 0;
}
```

Shameless Plug



Questions

- Comments
- Compliments
- Complaints

- guy@gameaudioprogrammer.com