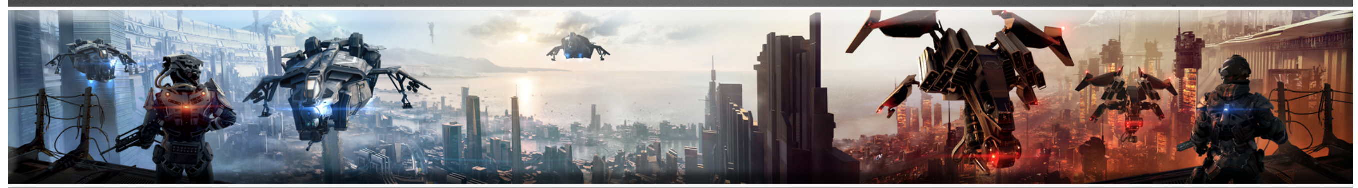
The Next-Gen Dynamic Sound System of Killzone Shadow Fall



GDC 14

GAME DEVELOPERS CONFERENCE
SAN FRANCISCO, CA
MARCH 17–21, 2014
EXPO DATES: MARCH 19–21
2014

Take Away

Take Away

- Promote experimentation

Take Away

- Promote experimentation
- Own your tools to allow it to happen

Take Away

- Promote experimentation
- Own your tools to allow it to happen
- Reduce “Idea to Game” time as much as possible

Take Away

Take Away

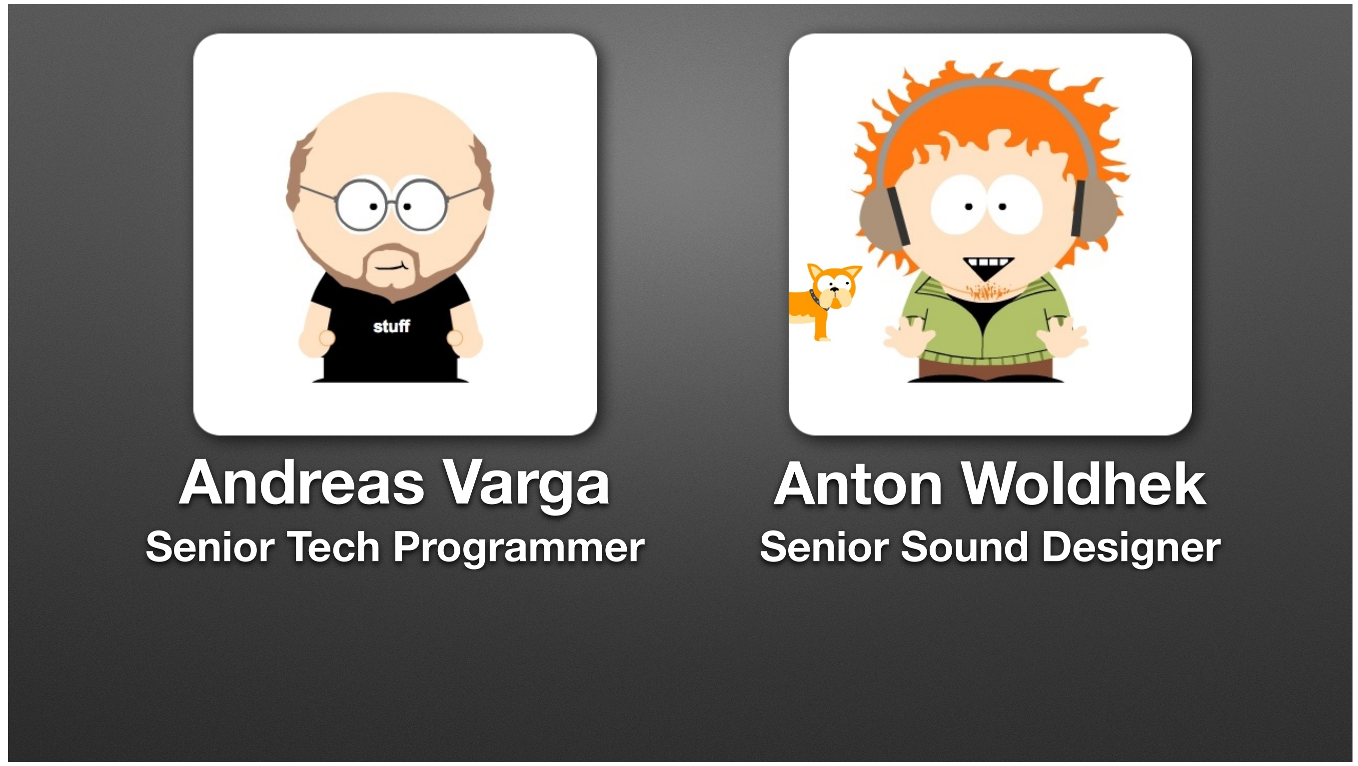
- If a programmer is in the creative loop, creativity suffers

Take Away

- If a programmer is in the creative loop, creativity suffers
- Don't be afraid to give designers a visual programming tool

Take Away

- If a programmer is in the creative loop, creativity suffers
- Don't be afraid to give designers a visual programming tool
- Generating code from assets is easy and powerful



Who are Andreas & Anton? Why would you listen to them? Well, we worked on sound for these games…

Who are Andreas & Anton? Why would you listen to them? Well, we worked on sound for these games…

We work at Guerrilla, which is one of Sony's 1st party studios. Here's a view of our sound studios


Killzone 2
Killzone 3

After Killzone 3, we started working on **Killzone Shadow Fall**, which was a **launch title** for the **PS4**. This was planned as a next-gen title from the beginning.

Here's a short video showing how the game looks and sounds.

Here's a short video showing how the game looks and sounds.

Andreas: But what exactly does next-gen mean?
We thought next gen sound (initially) wouldn't be about fancy new DSP but about faster iteration and less optimization. To allow the creation of more flexible sounds that truly can become a part of the game design. We also wanted to make sure that the sound team works with the same basic work flow as the rest of the company.
So our motivation was to build a **cutting edge** sound system and tool set, but **don't build a new synth**.

What is Next-Gen Audio?

Anton

What is Next-Gen Audio?

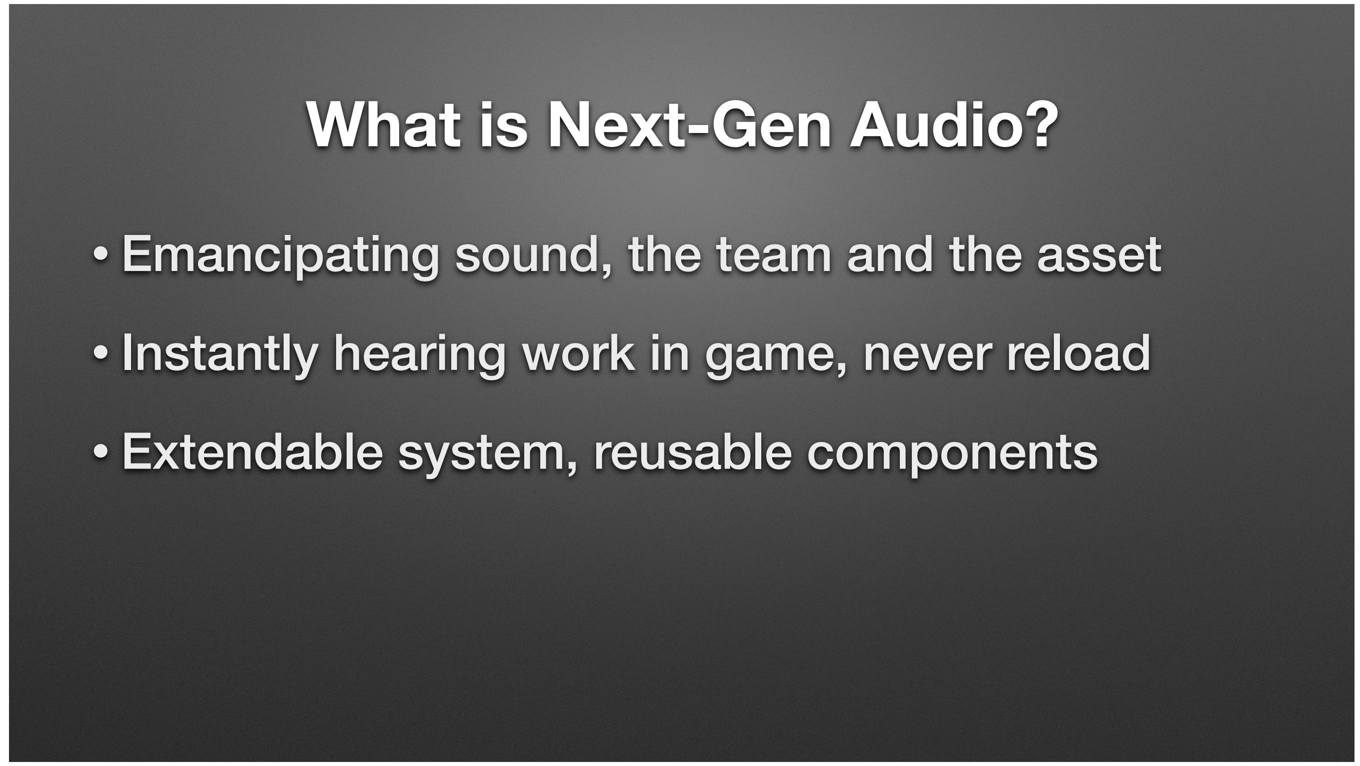
- Emancipating sound, the team and the asset

Anton

What is Next-Gen Audio?

- Emancipating sound, the team and the asset
- Instantly hearing work in game, never reload

Anton


What is Next-Gen Audio?
• Emancipating sound, the team and the asset
• Instantly hearing work in game, never reload
• Extendable system, reusable components

Anton

What is Next-Gen Audio?

- Emancipating sound, the team and the asset
- Instantly hearing work in game, never reload
- Extendable system, reusable components
- High run-time performance

Anton

What is Next-Gen Audio?

- Emancipating sound, the team and the asset
- Instantly hearing work in game, never reload
- Extendable system, reusable components
- High run-time performance
- Don’t worry about the synth

Anton

Why Emancipate?

Anton

Why Emancipate?

- Audio is separate from rest

Anton

Why Emancipate?

- Audio is separate from rest
- Physical: sound proof rooms

Anton

Why Emancipate?

- Audio is separate from rest
- Physical: sound proof rooms
- Mental: closed off sound engine and tools

Anton

Why Emancipate?

- Audio is separate from rest
- Physical: sound proof rooms
- Mental: closed off sound engine and tools
- Should actively pursue integration

Anton

Why Integrate?

Anton

Why Integrate?

- Gain a community of end users that:

Anton

Why Integrate?

- Gain a community of end users that:
  - Can help with complex setup

Anton

Why Integrate?

- Gain a community of end users that:
    - Can help with complex setup
    - Teach you new tricks

Anton

Why Integrate?

- Gain a community of end users that:
  - Can help with complex setup
  - Teach you new tricks
  - Find bugs

Anton

Why Integrate?

- Gain a community of end users that:
  - Can help with complex setup
  - Teach you new tricks
  - Find bugs
  - Benefit mutually from improvements

Anton

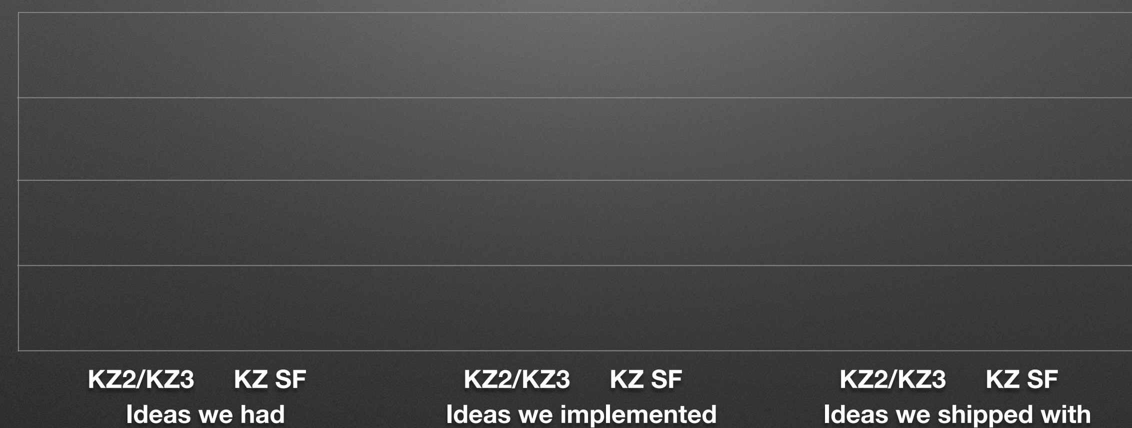
Motivation for Change

Motivation for Change

Previous generation very risk averse, failing was not possible

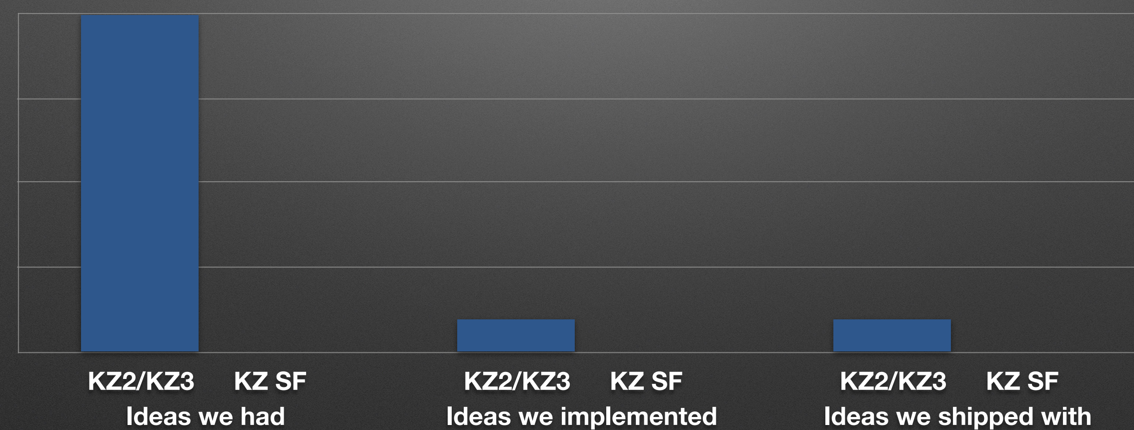
Motivation for Change

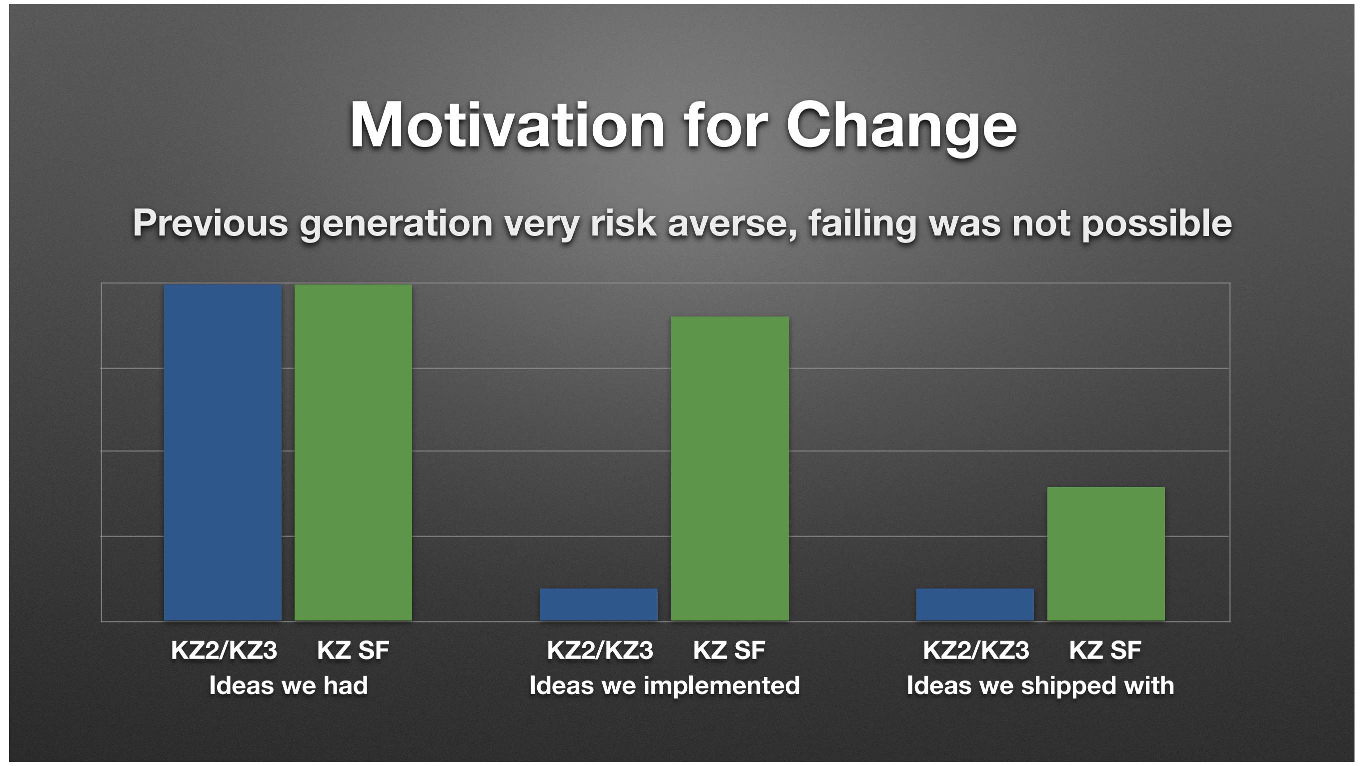
Previous generation very risk averse, failing was not possible



Motivation for Change

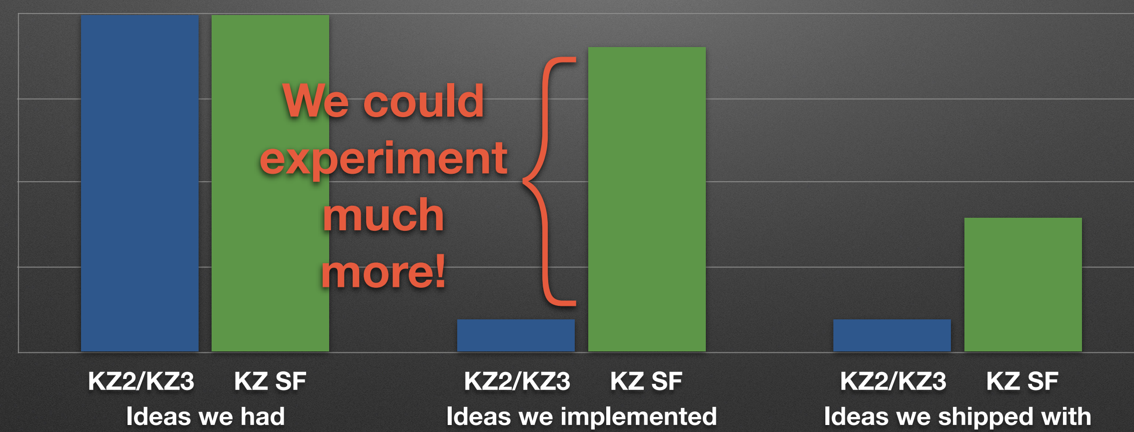
Previous generation very risk averse, failing was not possible




Motivation for Change
Previous generation very risk averse, failing was not possible
KZ2/KZ3
KZ SF
Ideas we had
KZ2/KZ3
KZ SF
Ideas we implemented
KZ2/KZ3
KZ SF
Ideas we shipped with

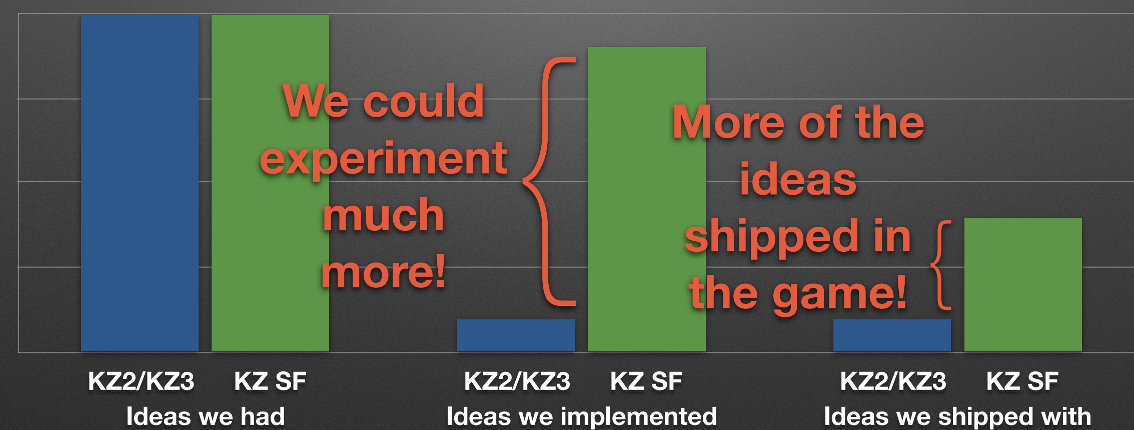
Motivation for Change

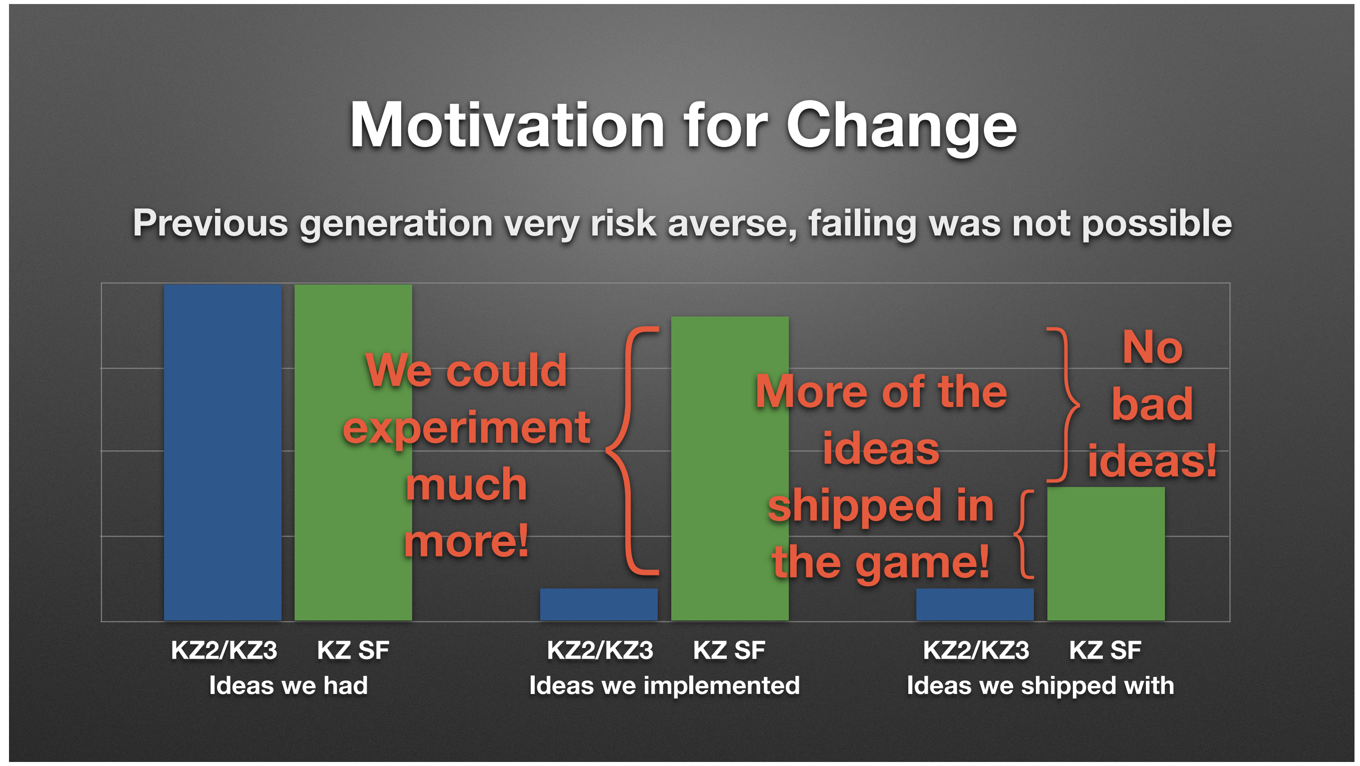
Previous generation very risk averse, failing was not possible



Motivation for Change

Previous generation very risk averse, failing was not possible



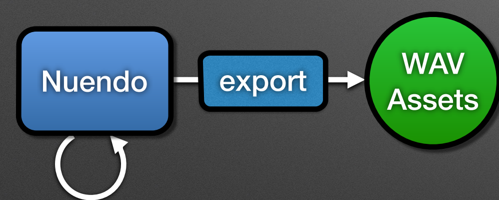

Motivation for Change
Previous generation very risk averse, failing was not possible
We could experiment much more!
More of the ideas shipped in the game!
No bad ideas!
KZ2/KZ3
KZ SF
Ideas we had
KZ2/KZ3
KZ SF
Ideas we implemented
KZ2/KZ3
KZ SF
Ideas we shipped with

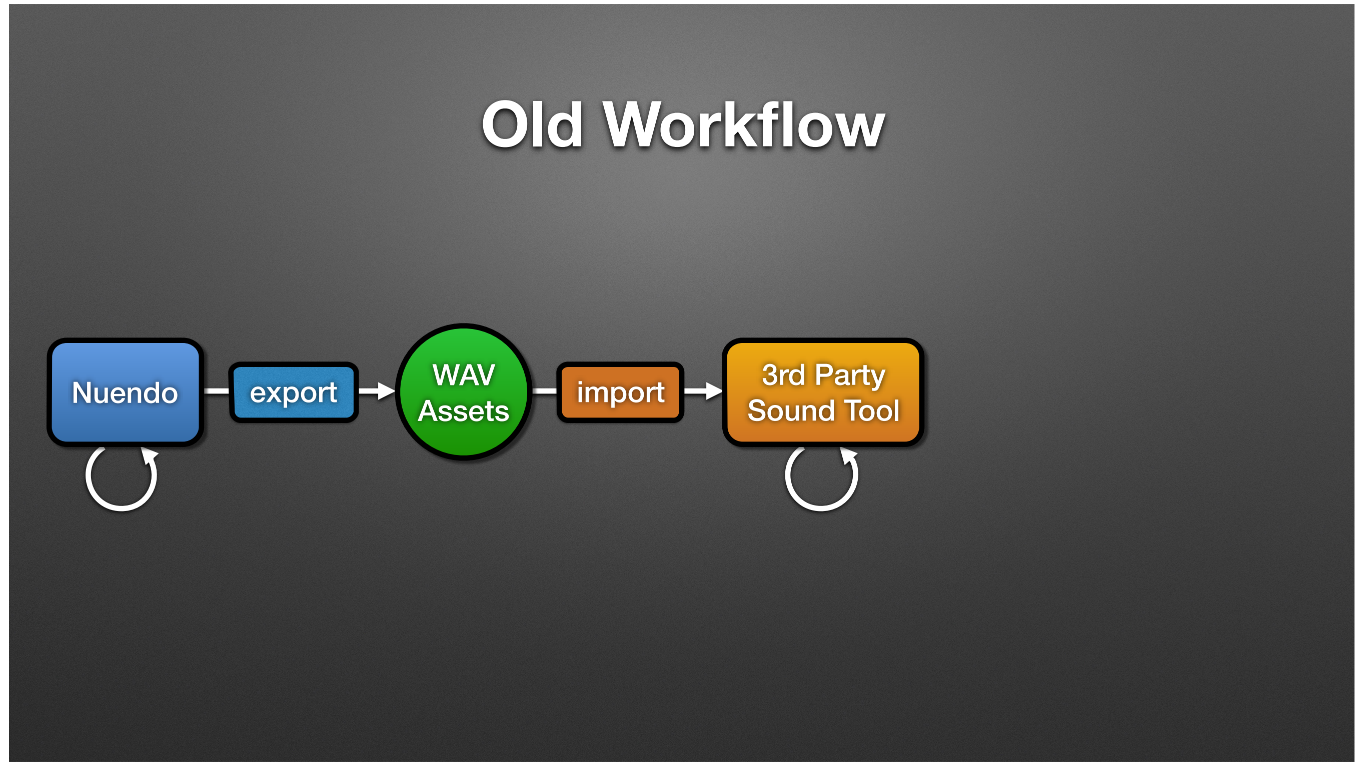
Old Workflow

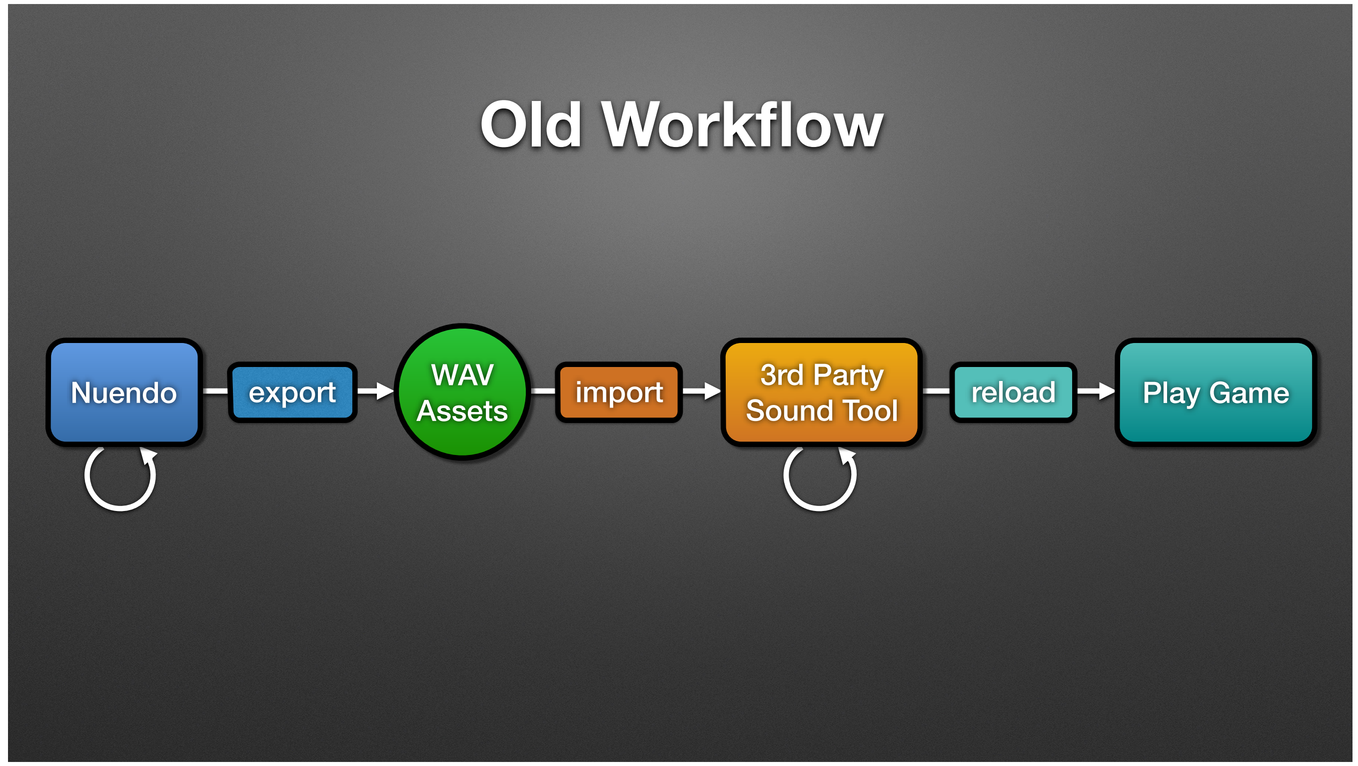
Nuendo

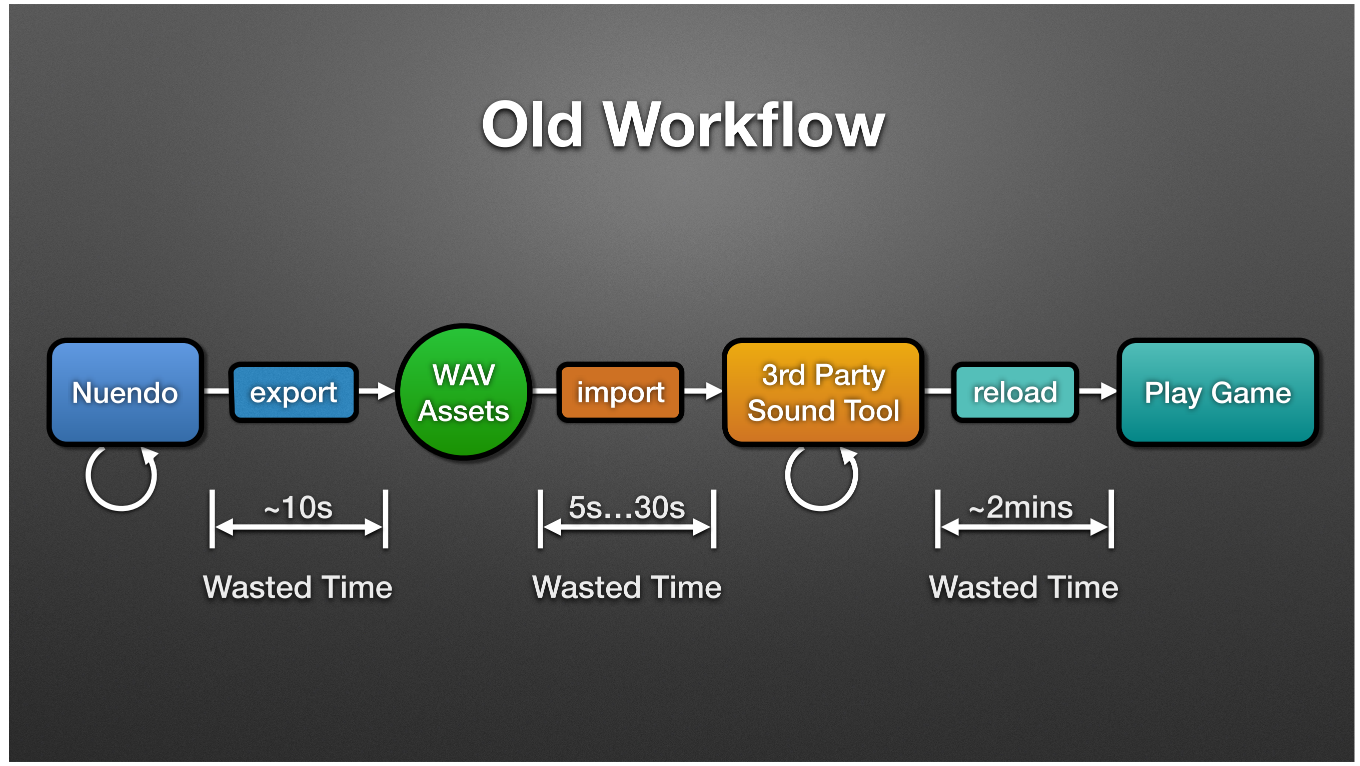
Old Workflow

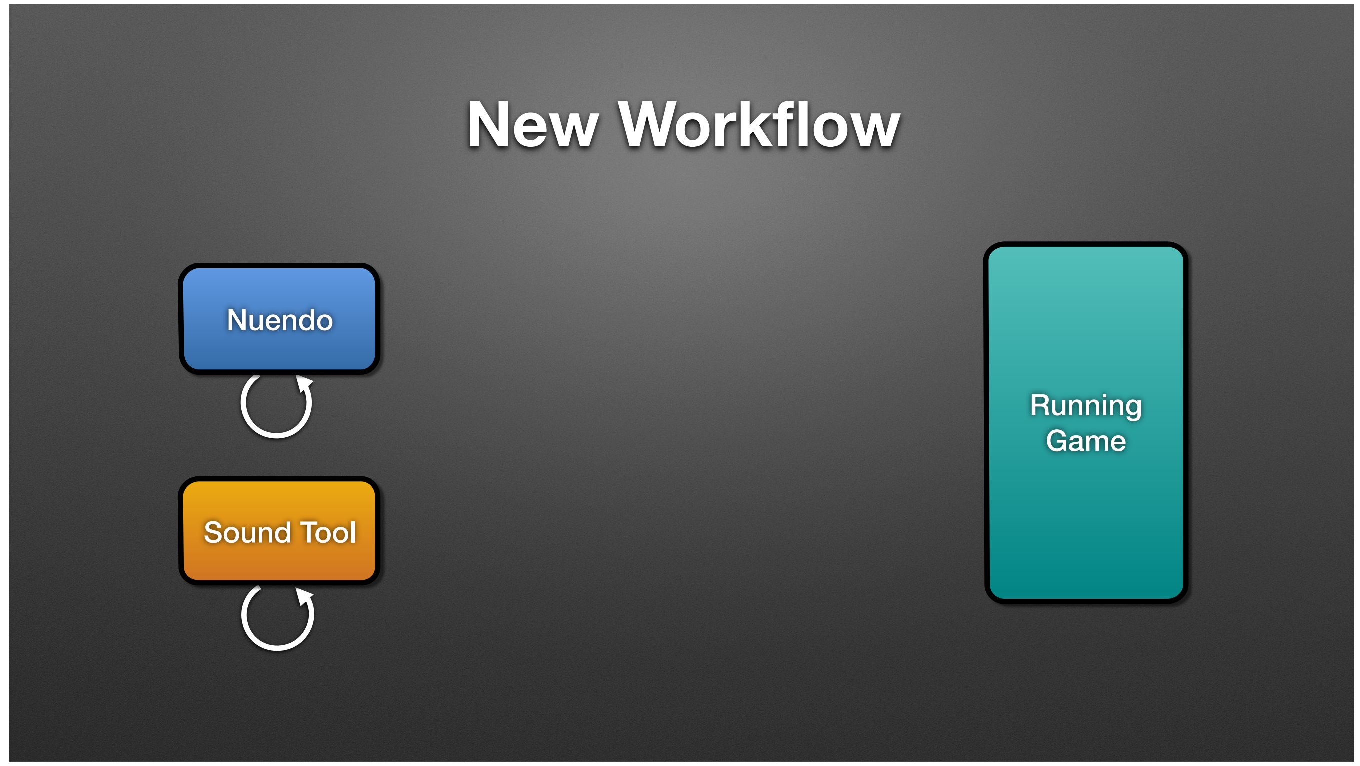
Old Workflow

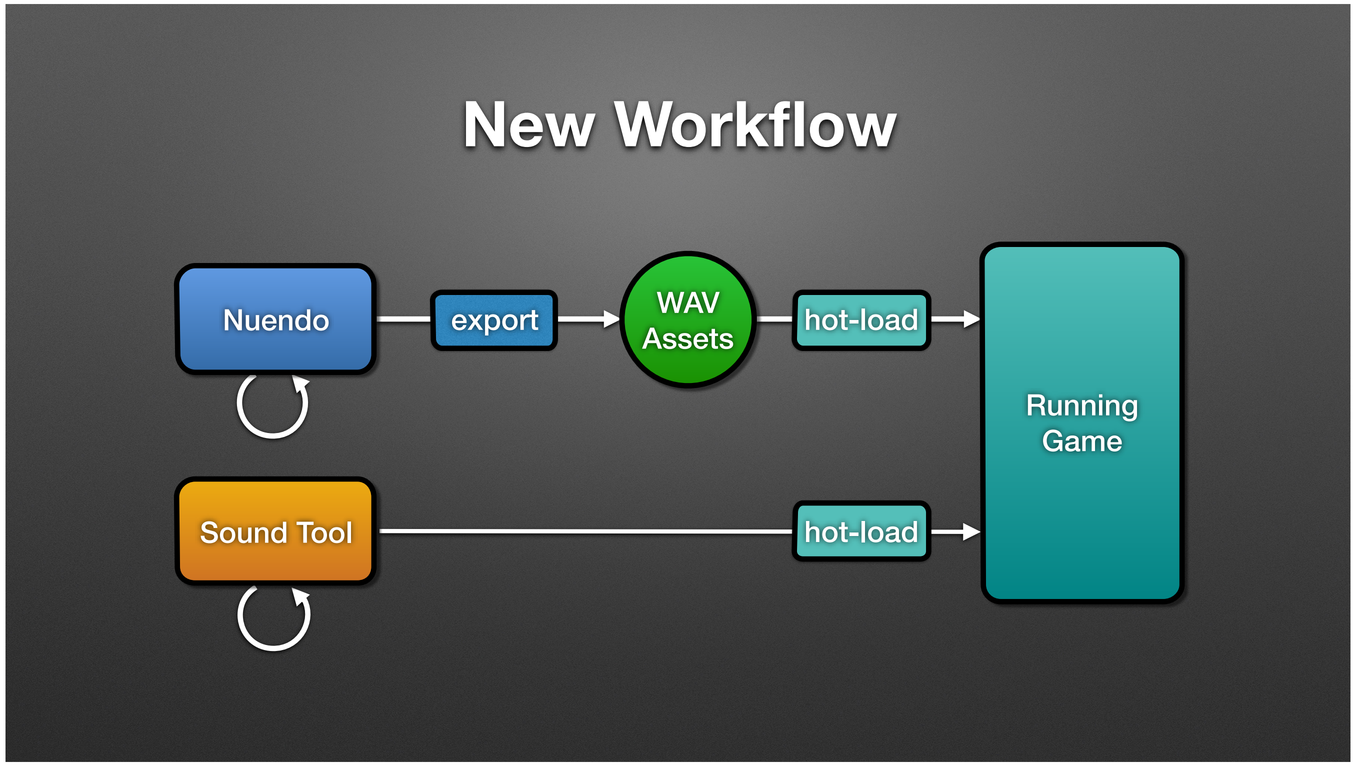


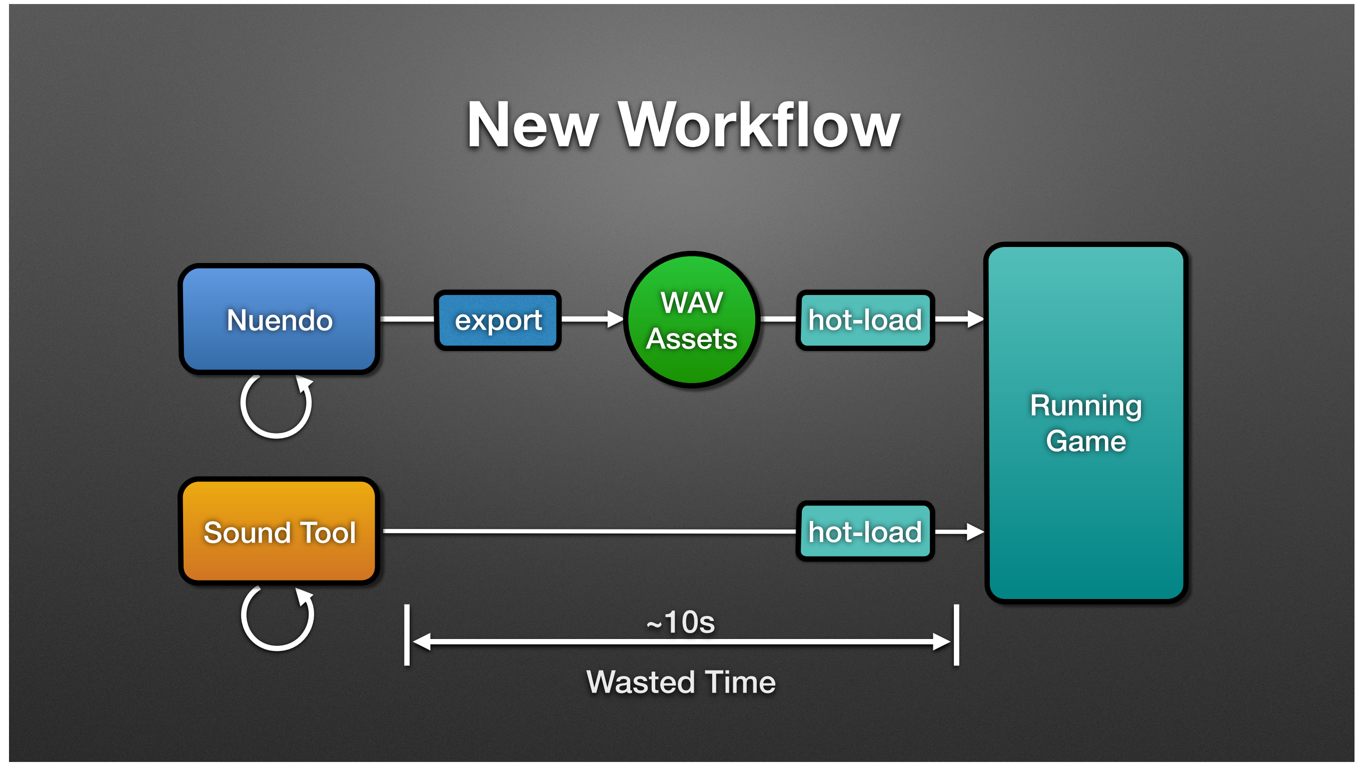

Old Workflow
Nuendo
export
WAV Assets
import
3rd Party Sound Tool


Old Workflow
Nuendo
export
WAV Assets
import
3rd Party Sound Tool
reload
Play Game


Old Workflow
Nuendo
export
WAV Assets
import
3rd Party Sound Tool
reload
Play Game
~10s
Wasted Time
5s…30s
Wasted Time
~2mins
Wasted Time


New Workflow
Nuendo
Sound Tool
Running Game


New Workflow
Nuendo
export
WAV Assets
hot-load
Running Game
Sound Tool
hot-load


New Workflow
Nuendo
export
WAV Assets
hot-load
Running Game
Sound Tool
hot-load
~10s
Wasted Time

Video of iterating on a single sound by re-exporting its wav files from Nuendo and syncing the changes with the running game.

Video of iterating on a single sound by re-exporting its wav files from Nuendo and syncing the changes with the running game.

No More Auditioning Tool

No More Auditioning Tool

- Game is the inspiration

No More Auditioning Tool

- Game is the inspiration
- Get sound into the game quickly

No More Auditioning Tool

- Game is the inspiration
- Get sound into the game quickly
- No interruptions

No More Auditioning Tool

- Game is the inspiration
- Get sound into the game quickly
- No interruptions
- Why audition anywhere else?

How to Emancipate?

Andreas

How to Emancipate?

- Make sound engine and tools in-house

Andreas

How to Emancipate?

- Make sound engine and tools in-house
- Allows deep integration

Andreas

How to Emancipate?

- Make sound engine and tools in-house
- Allows deep integration
- Immediate changes, based on designer feedback

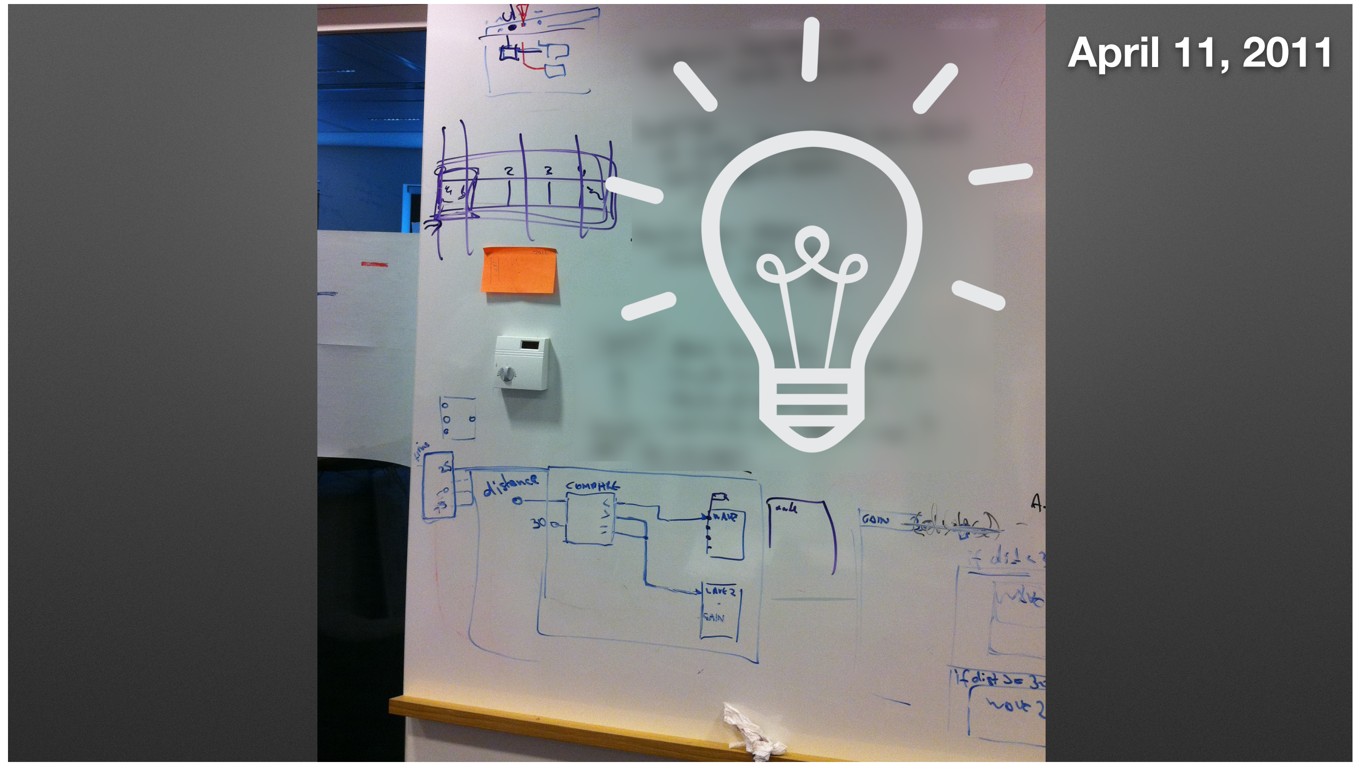
Andreas

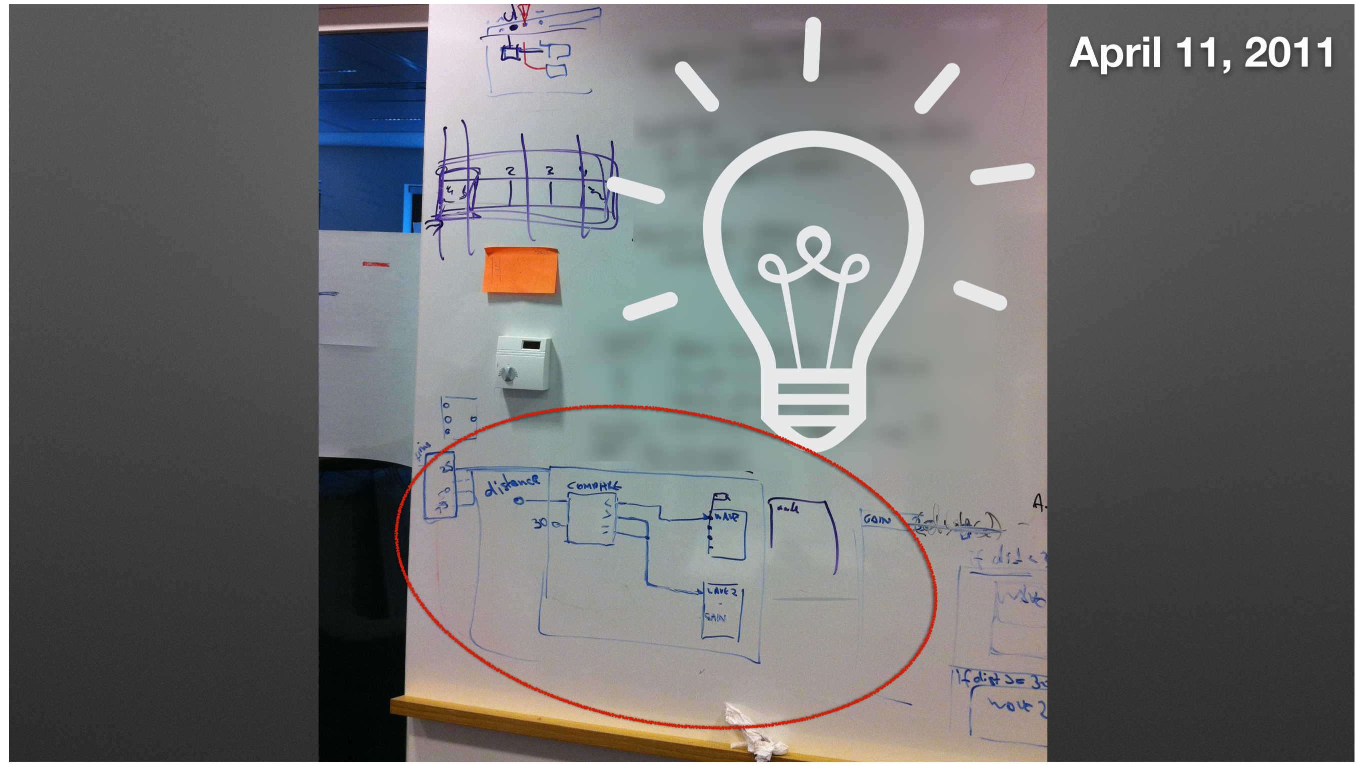
How to Emancipate?

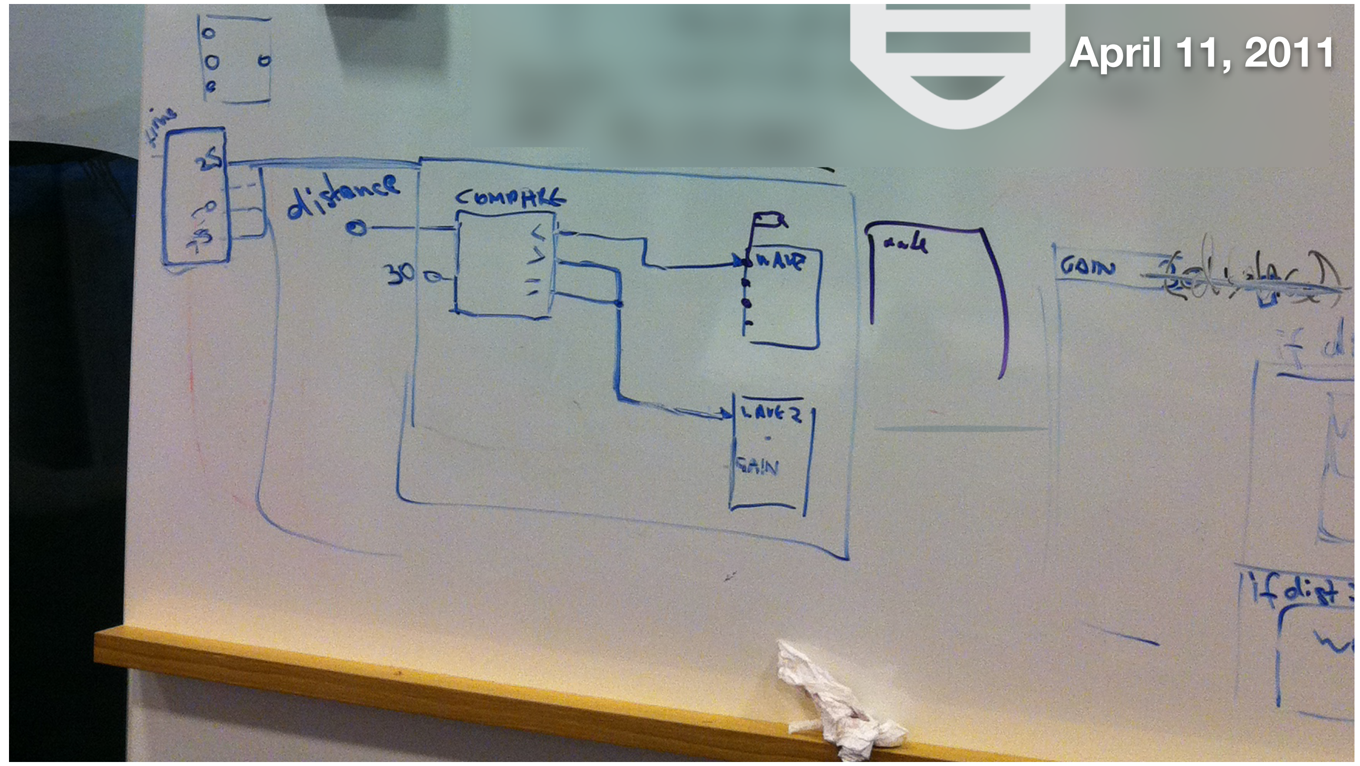
- Make sound engine and tools in-house
- Allows deep integration
- Immediate changes, based on designer feedback
- No arbitrary technical limitations

Andreas

Extendable, Reusable
and
High Performance?

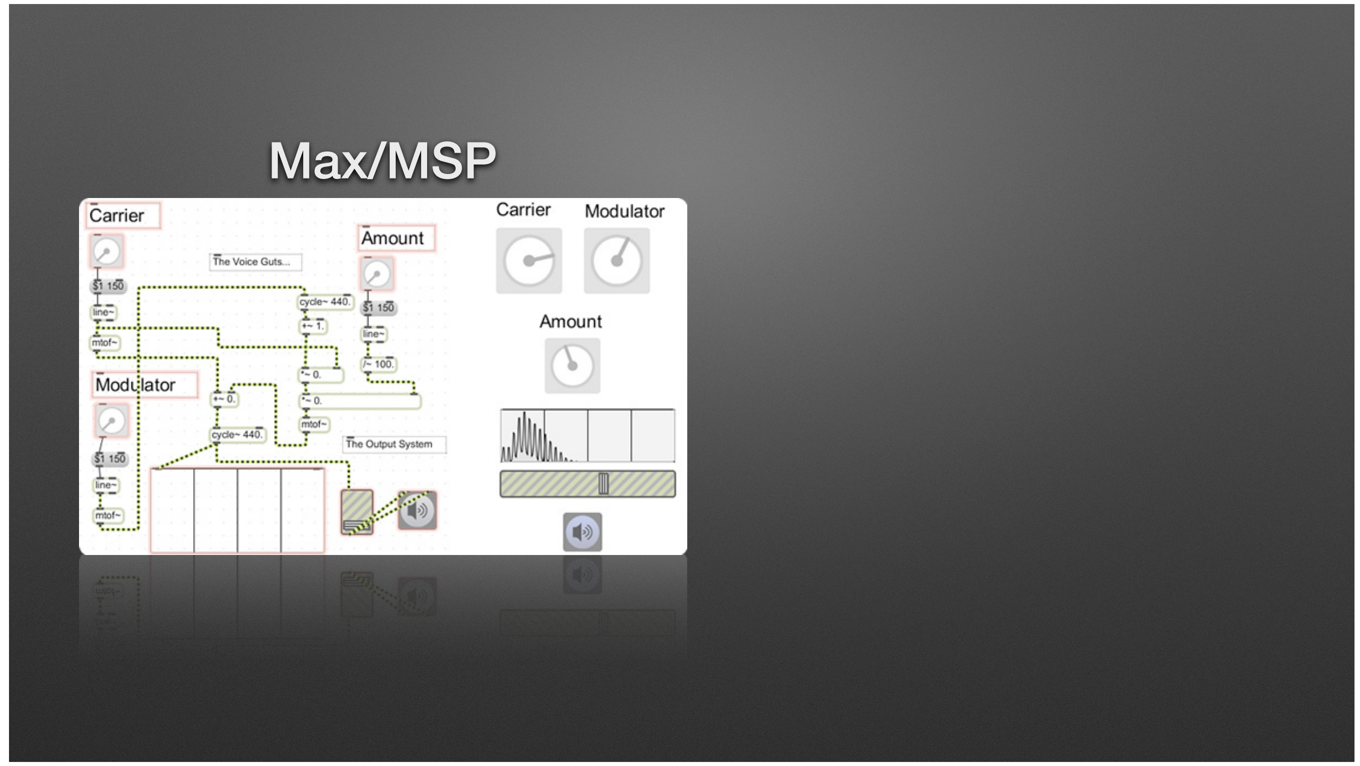

April 11, 2011


April 11, 2011


April 11, 2011
distance
30
GAIN

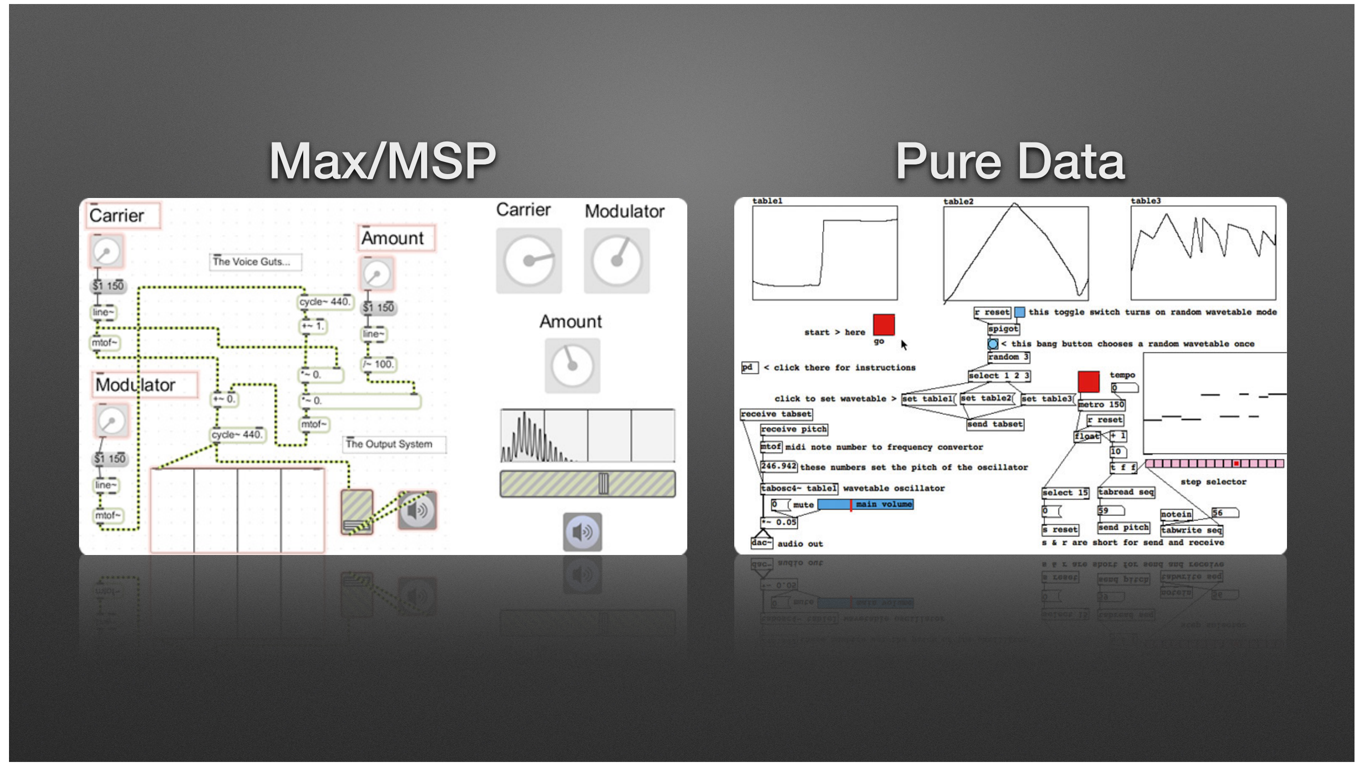
Anton: Sound designers are **already used to data flow** environments such as **Max/MSP** and **puredata**.

Andreas: But while that's one **good way of thinking** about our system, it's **also a bit different** from them. Our system doesn't work at on the **sample level or audio rate**, i.e. we're **not** using our graphs **for synthesis**, but just for **playback of** already existing **waveform data**.



Anton: Sound designers are **already used to data flow** environments such as **Max/MSP** and **puredata**.

Andreas: But while that's one **good way of thinking** about our system, it's **also a bit different** from them. Our system doesn't work at on the **sample level or audio rate**, i.e. we're **not** using our graphs **for synthesis**, but just for **playback of** already existing **waveform data**.



Anton: Sound designers are **already used to data flow** environments such as **Max/MSP** and **puredata**.

Andreas: But while that's one **good way of thinking** about our system, it's **also a bit different** from them. Our system doesn't work at on the **sample level or audio rate**, i.e. we're **not** using our graphs **for synthesis**, but just for **playback of** already existing **waveform data**.

⚠️ Technical Details Ahead! ⚠️

⚠️ Technical Details Ahead! ⚠️

- We will show code

⚠️ Technical Details Ahead! ⚠️

- We will show code
- But you don't have to be afraid

⚠️ Technical Details Ahead! ⚠️

- We will show code
- But you don't have to be afraid
- Sound designers will never have to look at it

Graph Sound System

Graph Sound System

- Generate native code from data flow

Graph Sound System

- Generate native code from data flow
- Execute resulting program every 5.333ms (187Hz)

Graph Sound System

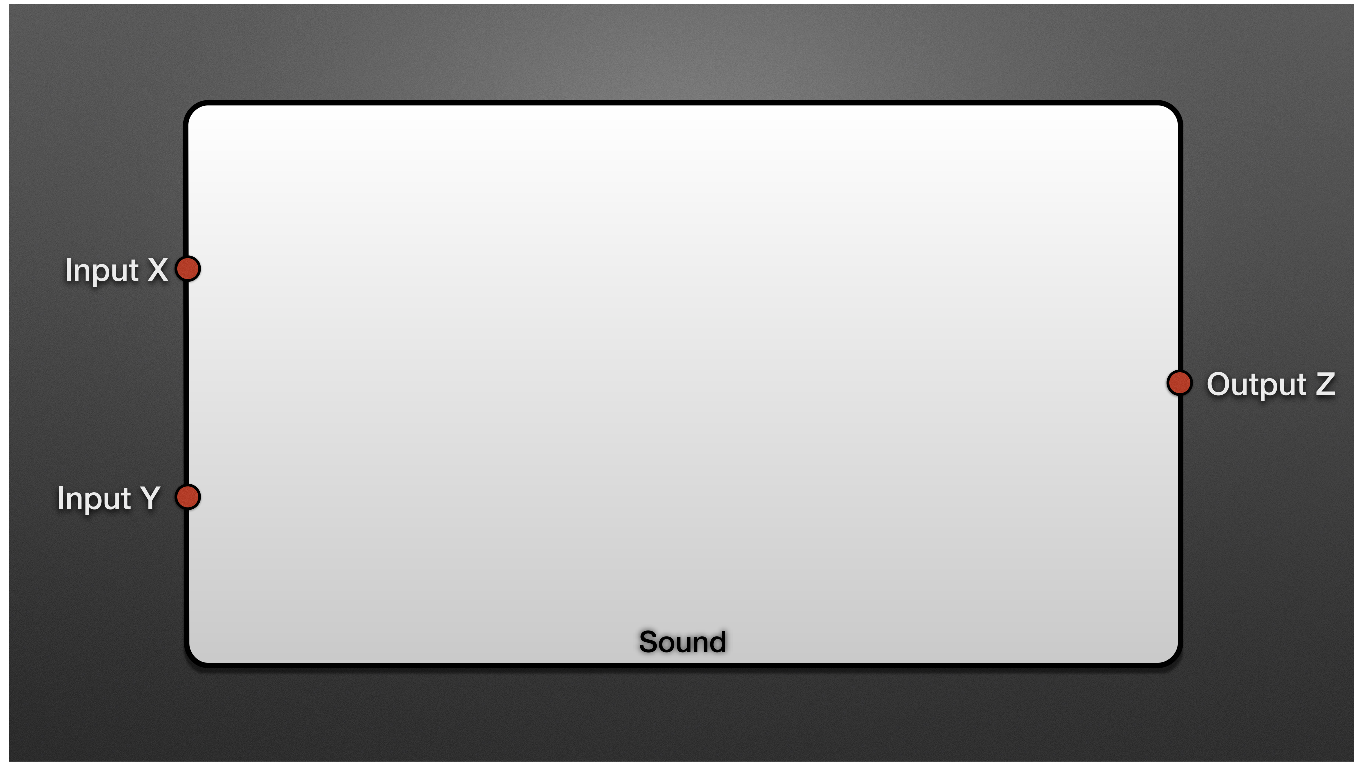
- Generate native code from data flow
- Execute resulting program every 5.333ms (187Hz)
- Used to play samples and modulate parameters

Graph Sound System

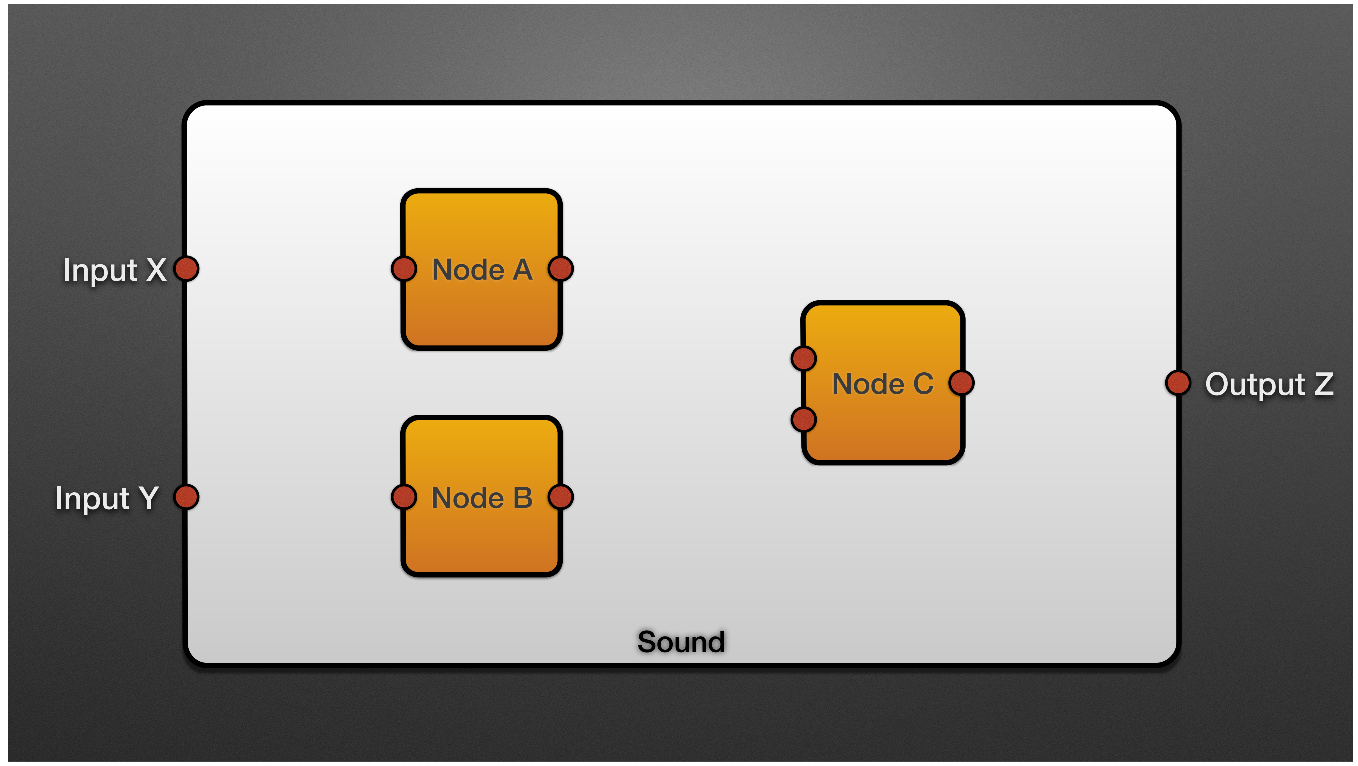
- Generate native code from data flow
- Execute resulting program every 5.333ms (187Hz)
- Used to play samples and modulate parameters
- NOT running actual synthesis

Graph Sound System

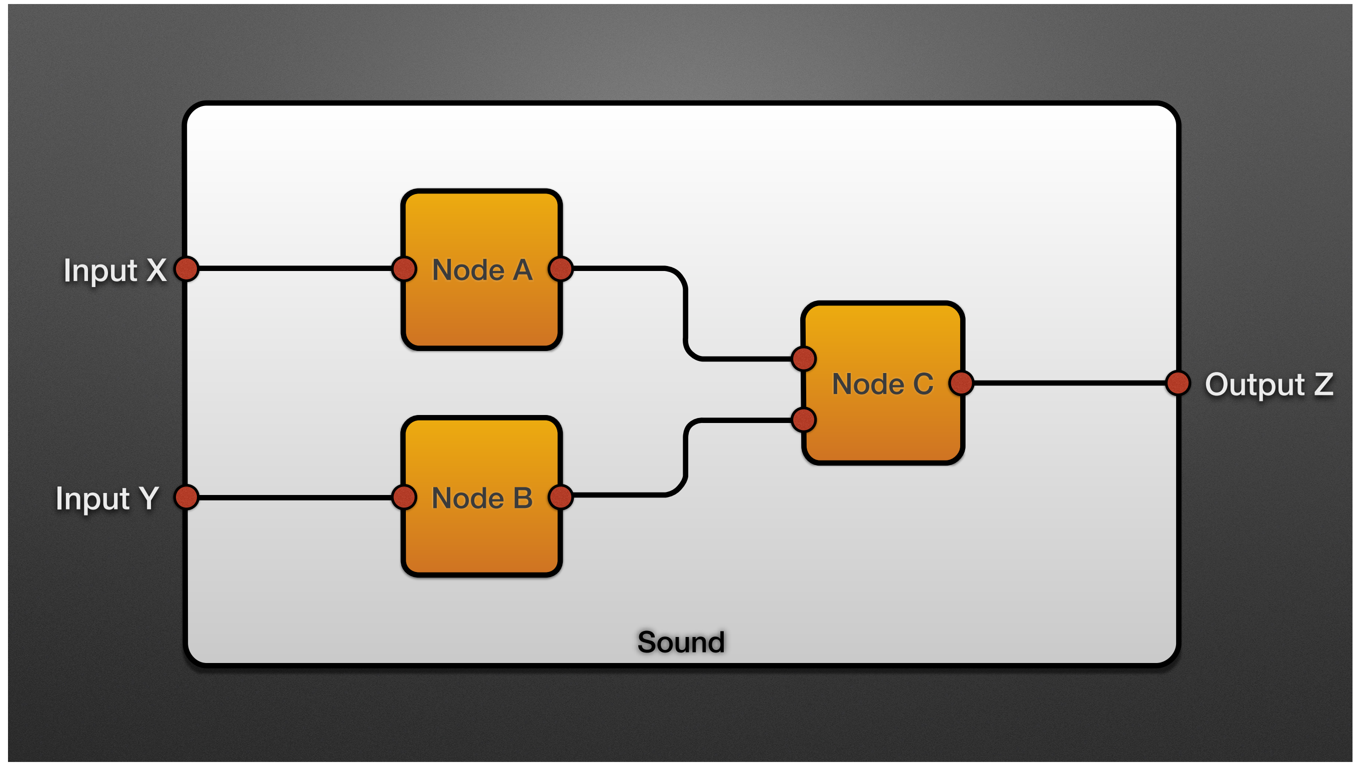
- Generate native code from data flow
- Execute resulting program every 5.333ms (187Hz)
- Used to play samples and modulate parameters
- NOT running actual synthesis
- Dynamic behavior



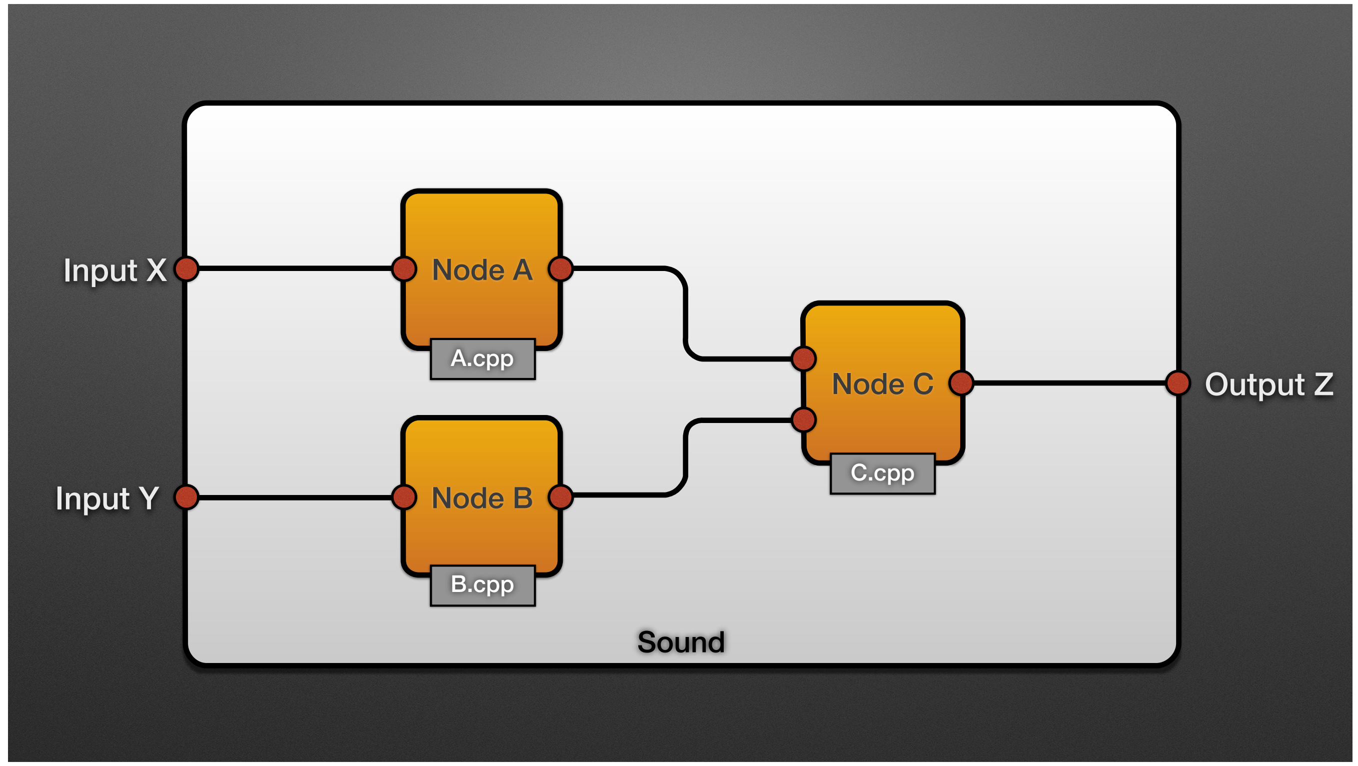
The graphs we build have certain **inputs** and **outputs** and are made up of individual **nodes** and **connections**. Note that each graph can also be used as a node, which is very important.
The functionality of each node is defined in a **C++ file**, which typically just contains **one function**.
When we need to translate this graph into C++ code, we take those **node definition files** and paste them into a C++ file for the whole graph.
We then **generate the code** according to **how** the **nodes** are **connected**, by calling the individual node functions in the right order.



The graphs we build have certain **inputs** and **outputs** and are made up of individual **nodes** and **connections**. Note that each graph can also be used as a node, which is very important.
The functionality of each node is defined in a **C++ file**, which typically just contains **one function**.
When we need to translate this graph into C++ code, we take those **node definition files** and paste them into a C++ file for the whole graph.
We then **generate the code** according to **how** the **nodes** are **connected**, by calling the individual node functions in the right order.

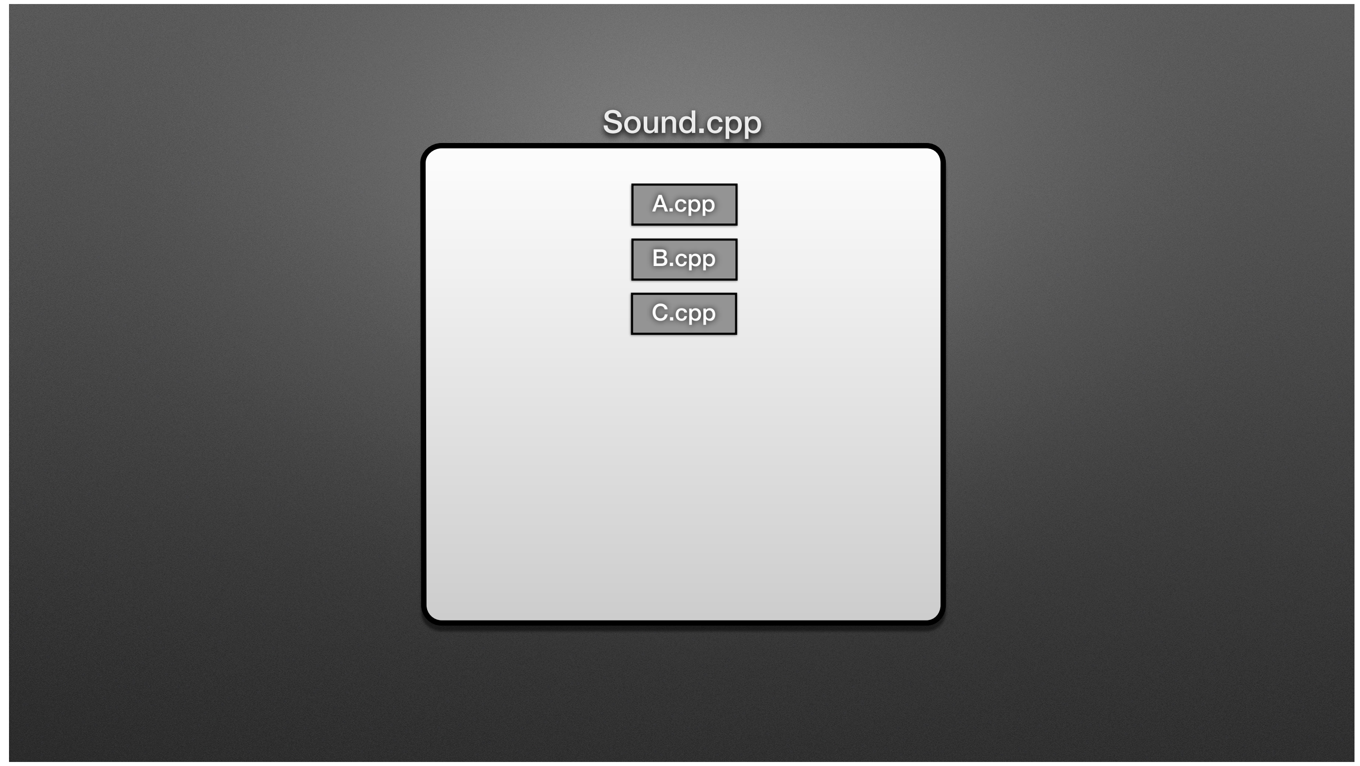


The graphs we build have certain **inputs** and **outputs** and are made up of individual **nodes** and **connections**. Note that each graph can also be used as a node, which is very important.
The functionality of each node is defined in a **C++ file**, which typically just contains **one function**.
When we need to translate this graph into C++ code, we take those **node definition files** and paste them into a C++ file for the whole graph.
We then **generate the code** according to **how** the **nodes** are **connected**, by calling the individual node functions in the right order.

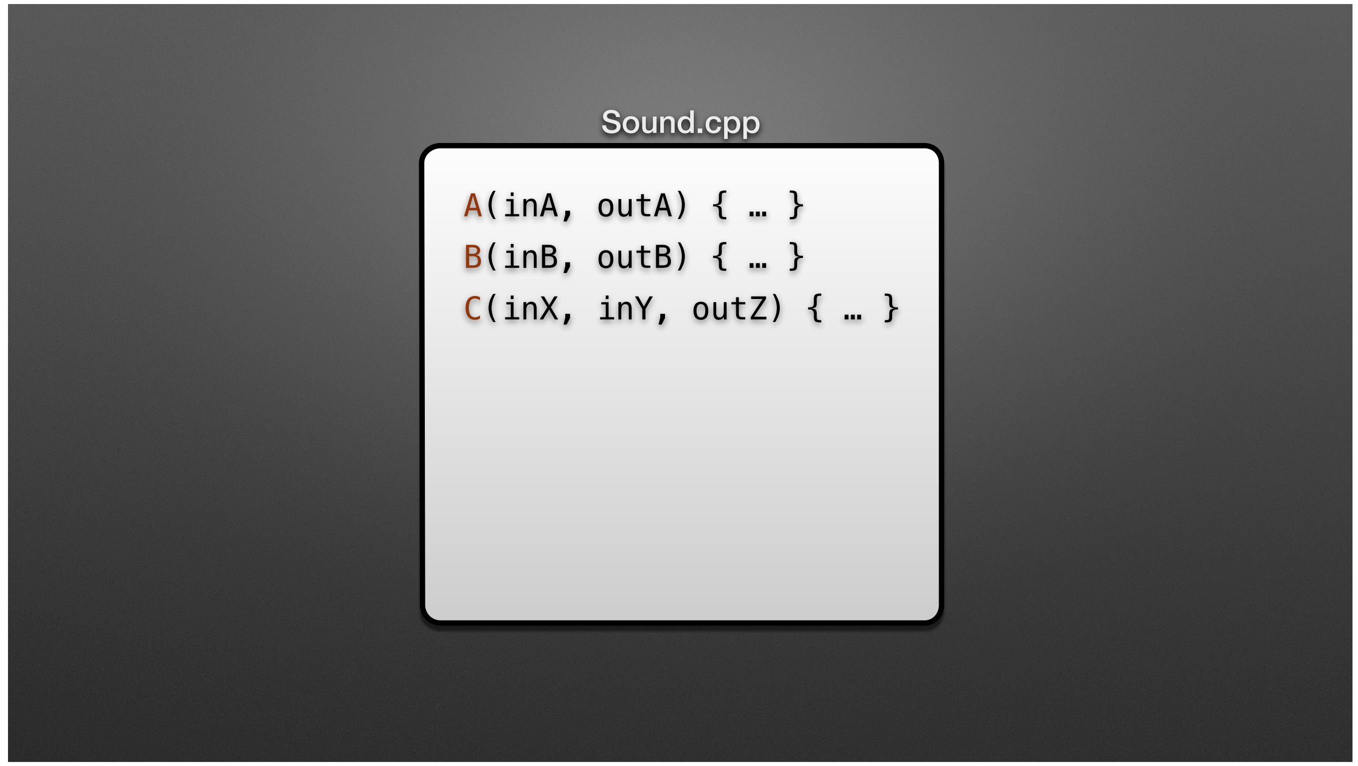


The graphs we build have certain **inputs** and **outputs** and are made up of individual **nodes** and **connections**. Note that each graph can also be used as a node, which is very important.
The functionality of each node is defined in a **C++ file**, which typically just contains **one function**.
When we need to translate this graph into C++ code, we take those **node definition files** and paste them into a C++ file for the whole graph.
We then **generate the code** according to **how** the **nodes** are **connected**, by calling the individual node functions in the right order.

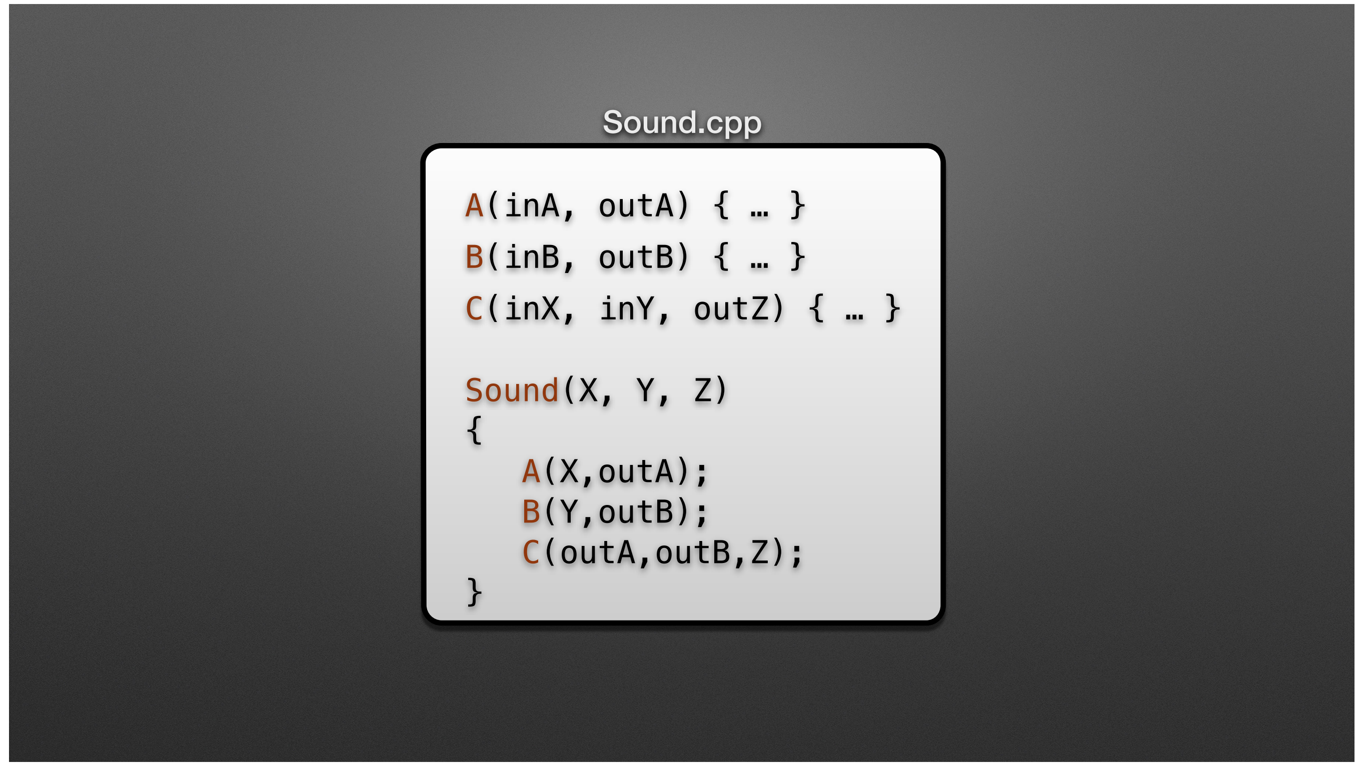
The graphs we build have certain **inputs** and **outputs** and are made up of individual **nodes** and **connections**. Note that each graph can also be used as a node, which is very important.
The functionality of each node is defined in a **C++ file**, which typically just contains **one function**.
When we need to translate this graph into C++ code, we take those **node definition files** and paste them into a C++ file for the whole graph.
We then **generate the code** according to **how** the **nodes** are **connected**, by calling the individual node functions in the right order.



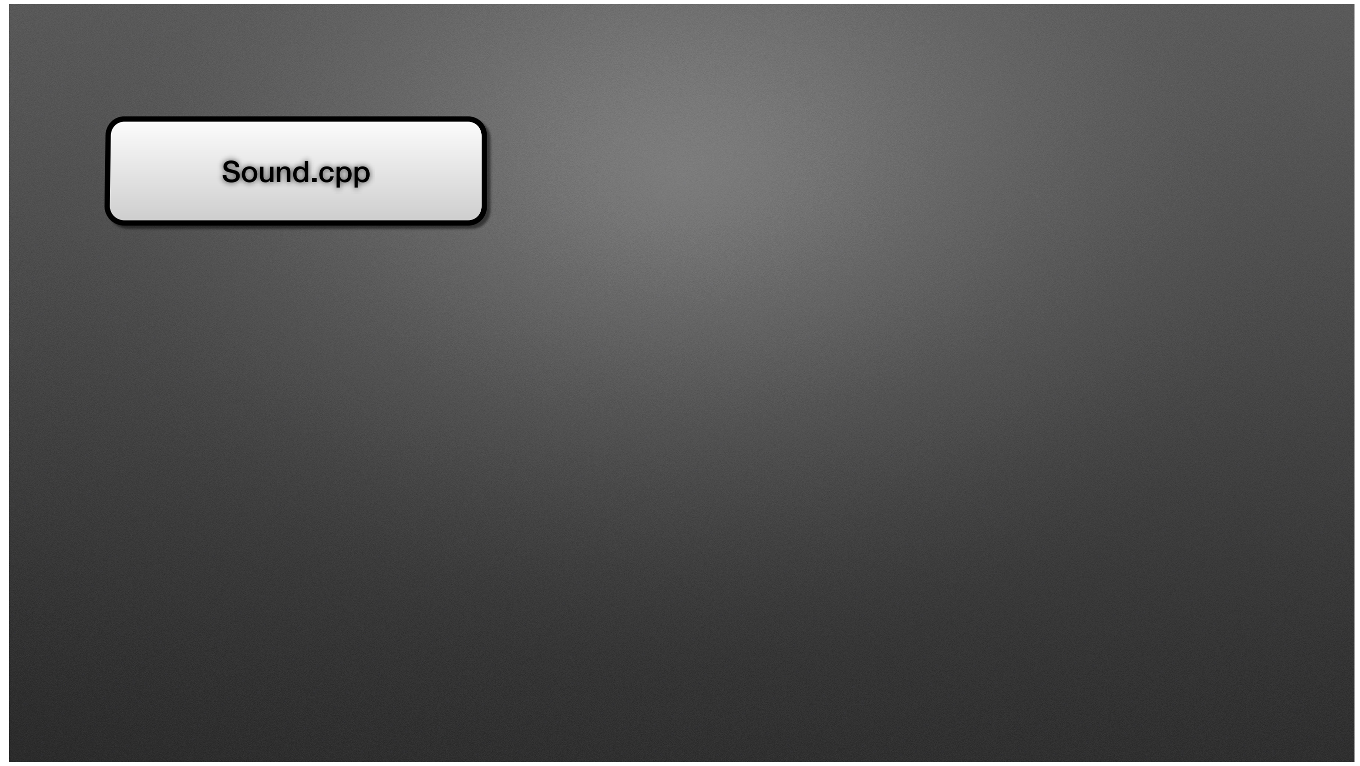
The graphs we build have certain **inputs** and **outputs** and are made up of individual **nodes** and **connections**. Note that each graph can also be used as a node, which is very important.
The functionality of each node is defined in a **C++ file**, which typically just contains **one function**.
When we need to translate this graph into C++ code, we take those **node definition files** and paste them into a C++ file for the whole graph.
We then **generate the code** according to **how** the **nodes** are **connected**, by calling the individual node functions in the right order.



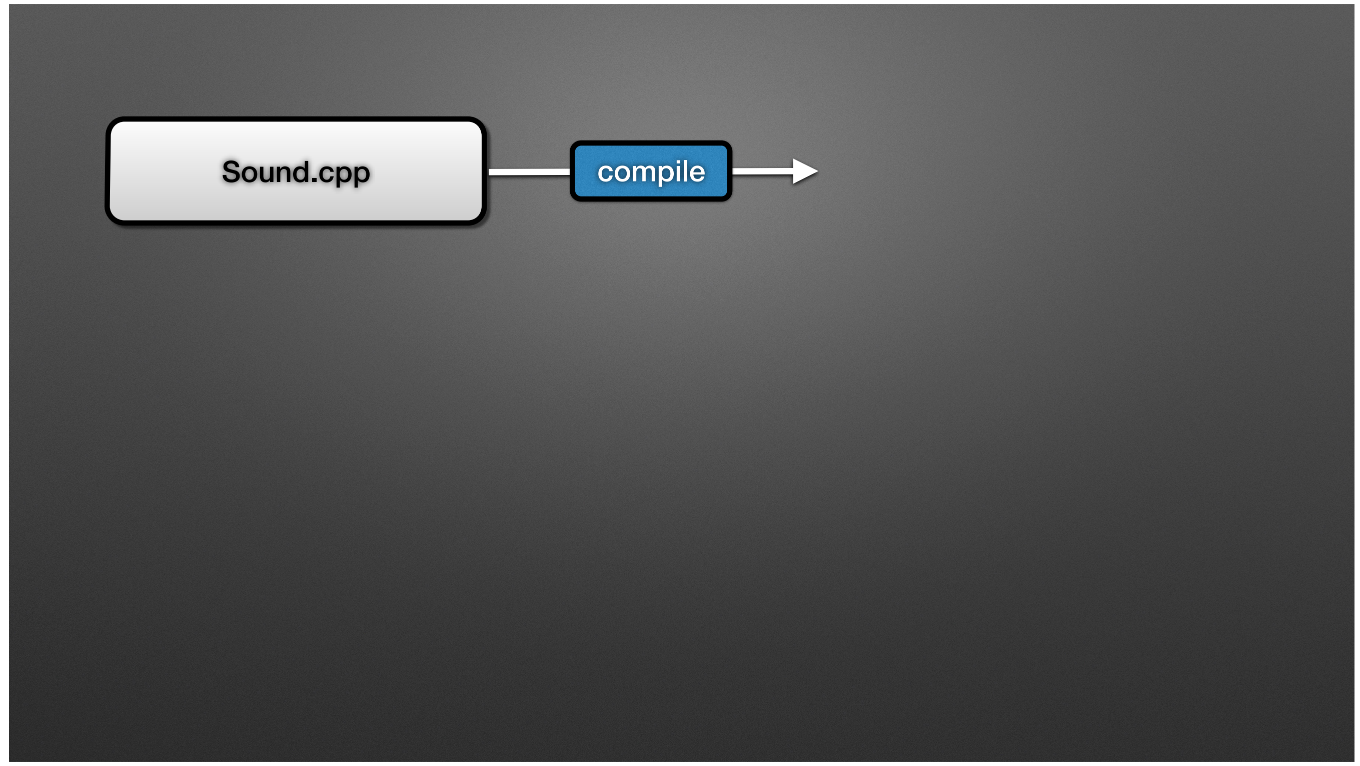
The graphs we build have certain **inputs** and **outputs** and are made up of individual **nodes** and **connections**. Note that each graph can also be used as a node, which is very important.
The functionality of each node is defined in a **C++ file**, which typically just contains **one function**.
When we need to translate this graph into C++ code, we take those **node definition files** and paste them into a C++ file for the whole graph.
We then **generate the code** according to **how** the **nodes** are **connected**, by calling the individual node functions in the right order.



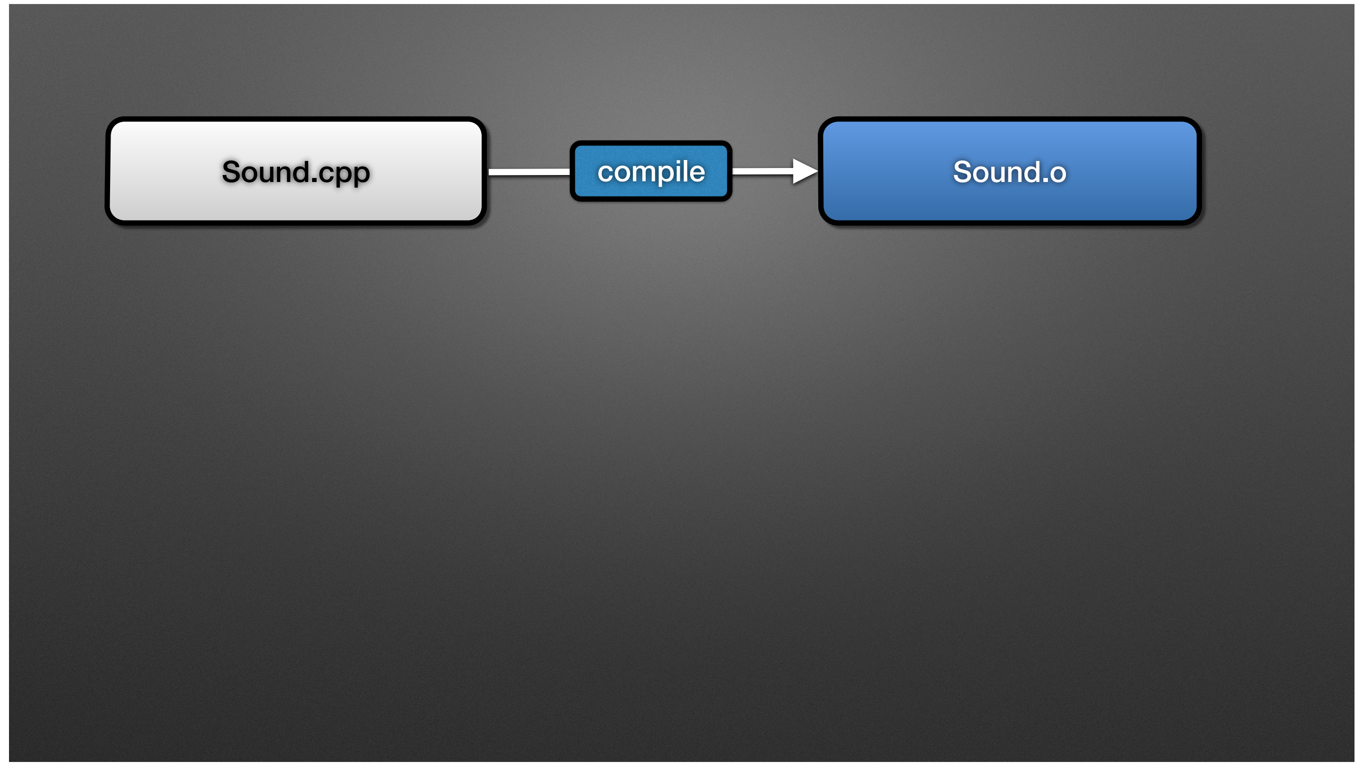
The graphs we build have certain **inputs** and **outputs** and are made up of individual **nodes** and **connections**. Note that each graph can also be used as a node, which is very important.
The functionality of each node is defined in a **C++ file**, which typically just contains **one function**.
When we need to translate this graph into C++ code, we take those **node definition files** and paste them into a C++ file for the whole graph.
We then **generate the code** according to **how** the **nodes** are **connected**, by calling the individual node functions in the right order.



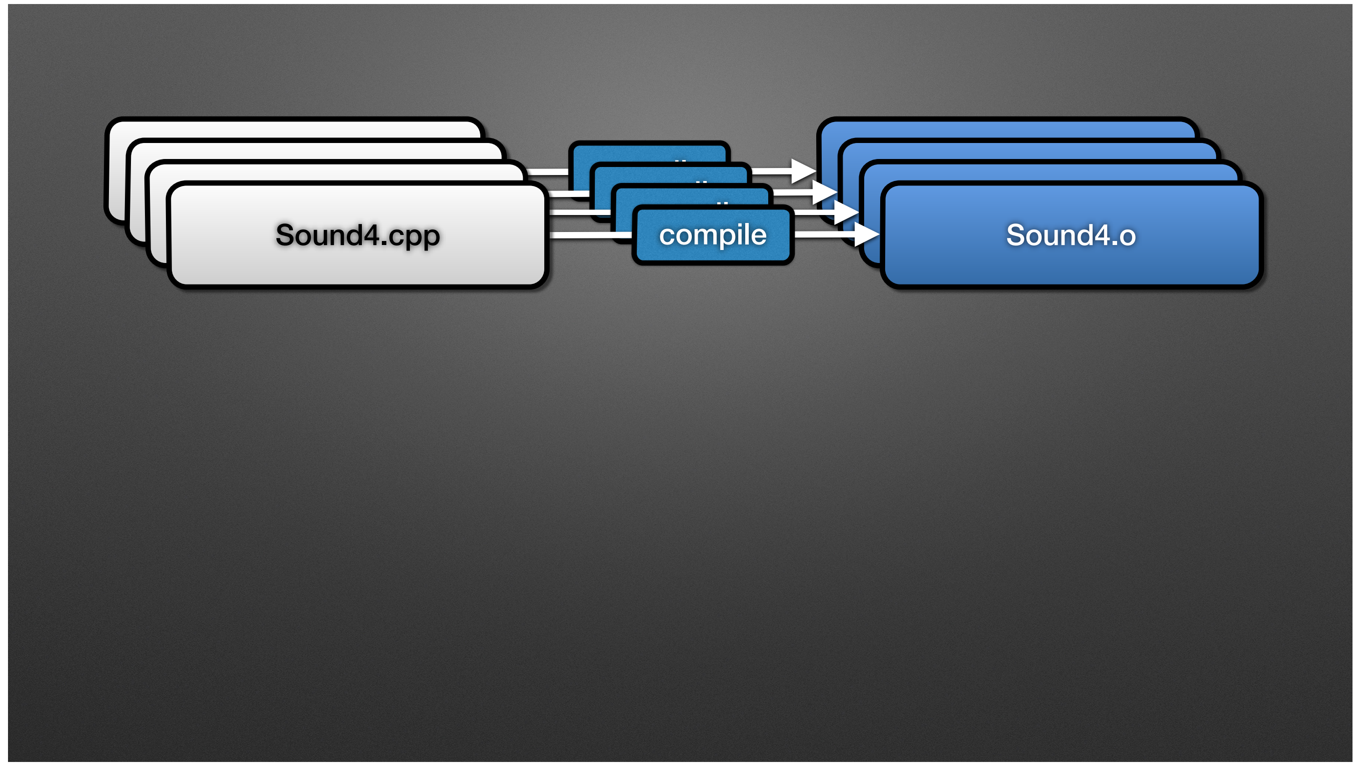
So as part of our **content pipeline** the C++ code that was generated for the graph is then **compiled** into an object file, and together with other object files gets **linked into a dynamic library**, which are loaded as PRX files on the PS4 at runtime. This allows us to execute the code in the game.
Our initial attempt was slightly different tough: We’ve first experimented with LLVM using byte code and the LLVM JIT compiler at runtime, which was nice for the initial testing and development. But soon after that we switched to a more traditional offline compilation step, using clang and dynamic libraries.



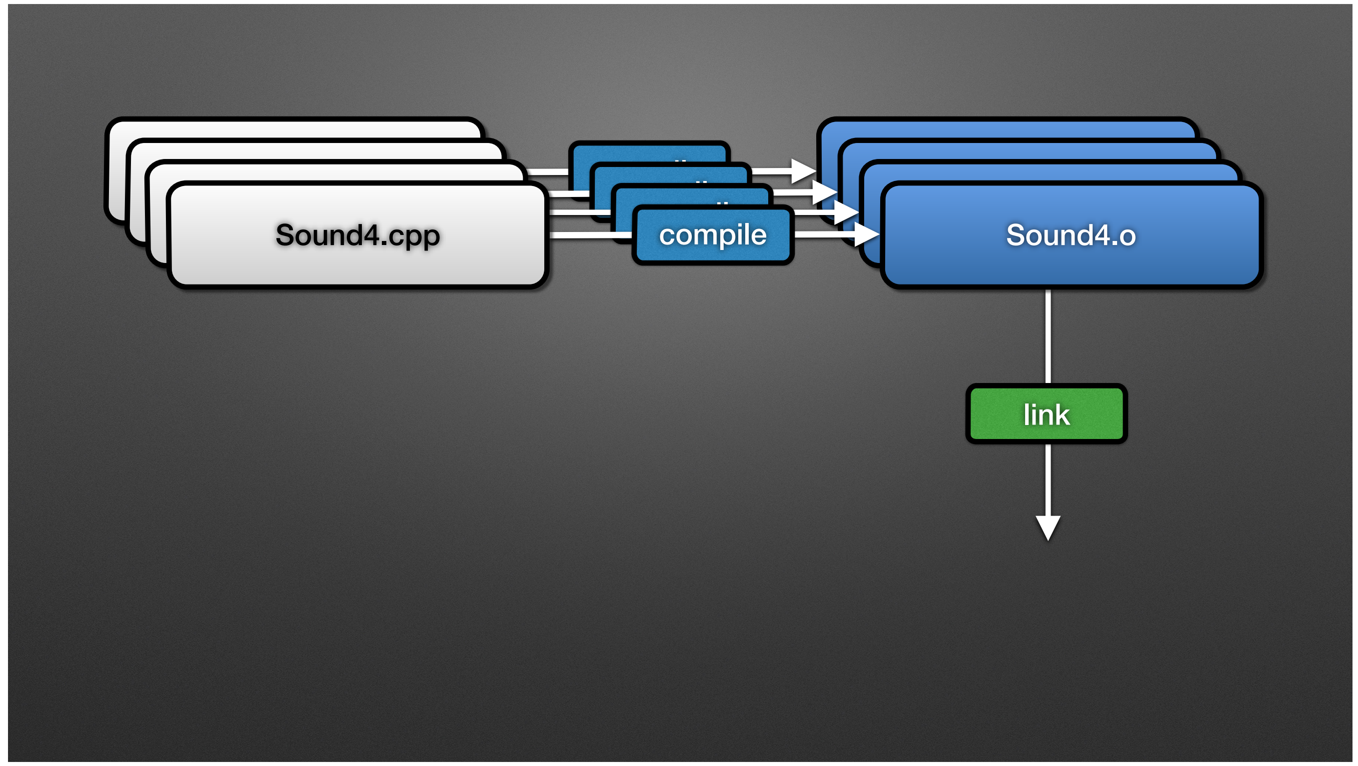
So as part of our **content pipeline** the C++ code that was generated for the graph is then **compiled** into an object file, and together with other object files gets **linked into a dynamic library**, which are loaded as PRX files on the PS4 at runtime. This allows us to execute the code in the game.
Our initial attempt was slightly different tough: We've first experimented with LLVM using byte code and the LLVM JIT compiler at runtime, which was nice for the initial testing and development. But soon after that we switched to a more traditional offline compilation step, using clang and dynamic libraries.



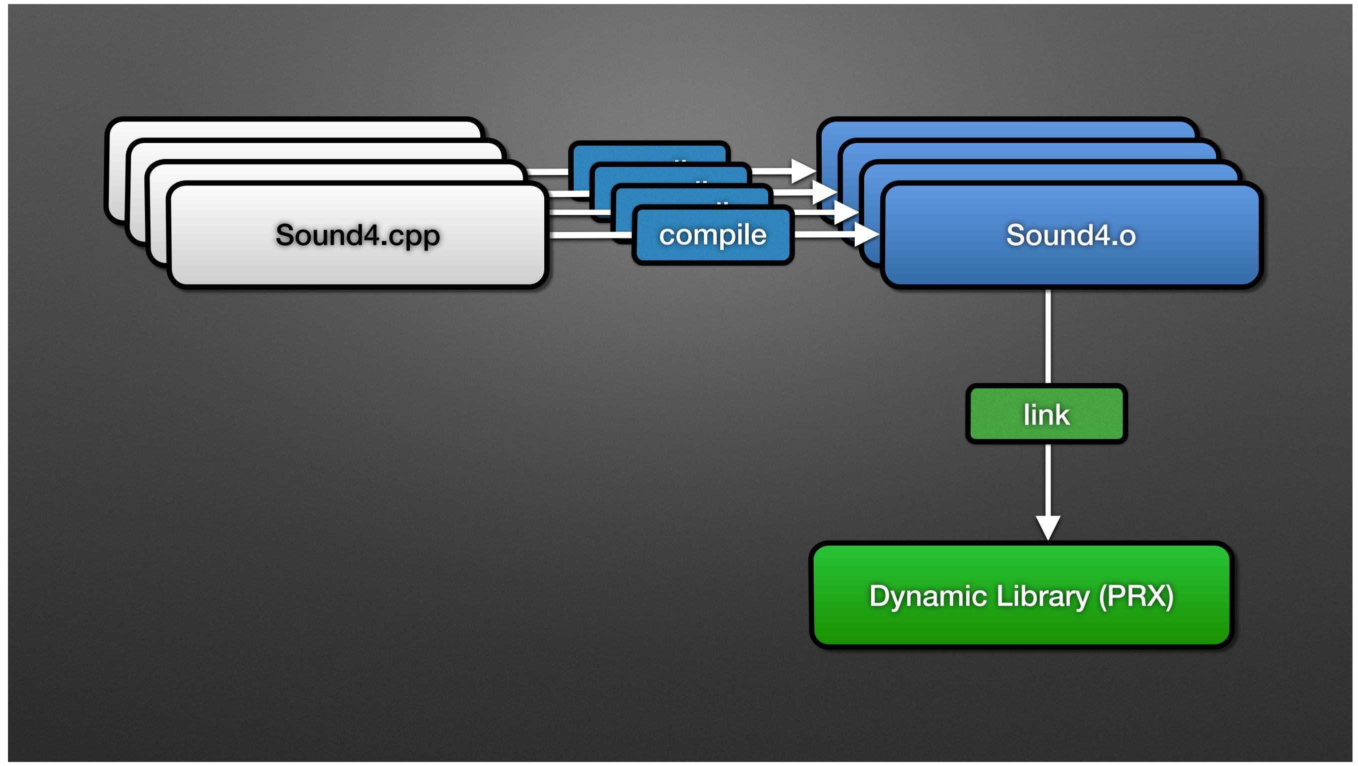
So as part of our **content pipeline** the C++ code that was generated for the graph is then **compiled** into an object file, and together with other object files gets **linked into a dynamic library**, which are loaded as PRX files on the PS4 at runtime. This allows us to execute the code in the game.
Our initial attempt was slightly different tough: We've first experimented with LLVM using byte code and the LLVM JIT compiler at runtime, which was nice for the initial testing and development. But soon after that we switched to a more traditional offline compilation step, using clang and dynamic libraries.



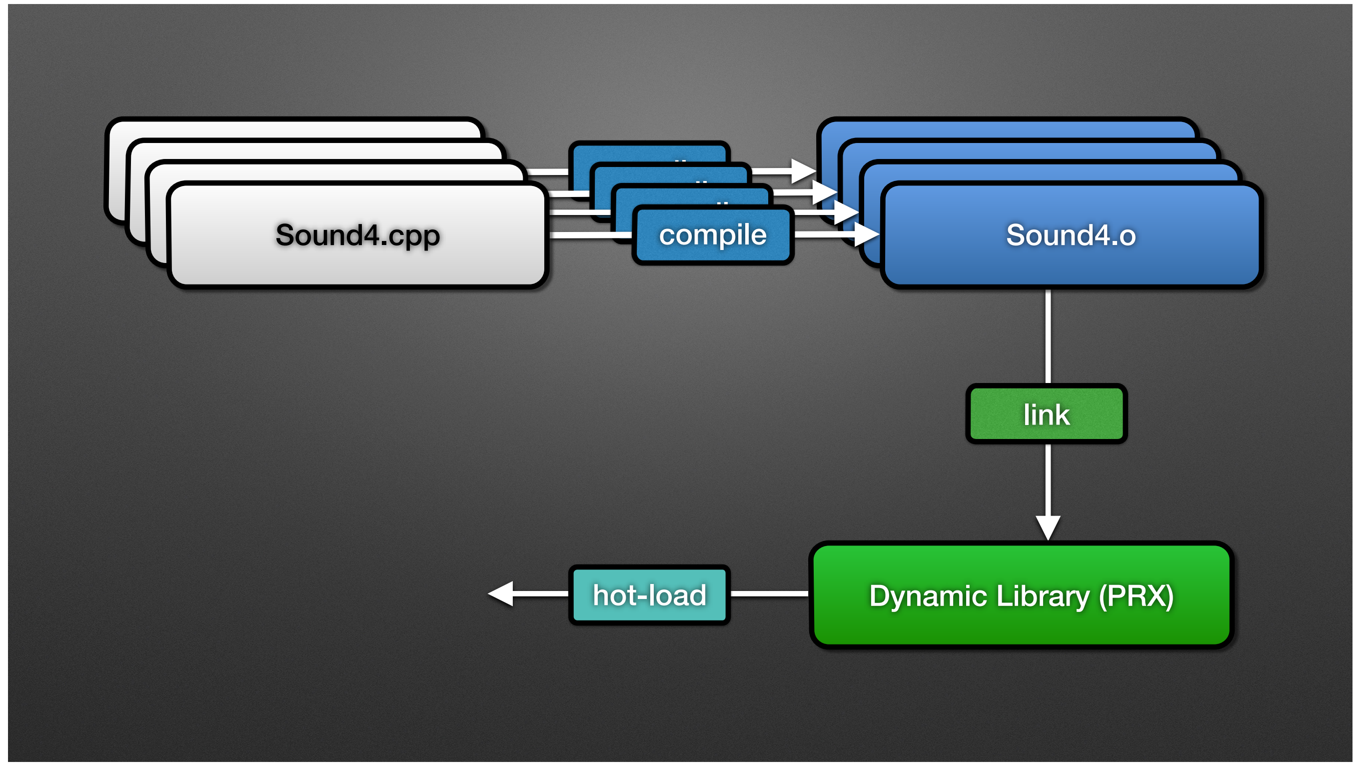
So as part of our **content pipeline** the C++ code that was generated for the graph is then **compiled** into an object file, and together with other object files gets **linked into a dynamic library**, which are loaded as PRX files on the PS4 at runtime. This allows us to execute the code in the game.
Our initial attempt was slightly different tough: We've first experimented with LLVM using byte code and the LLVM JIT compiler at runtime, which was nice for the initial testing and development. But soon after that we switched to a more traditional offline compilation step, using clang and dynamic libraries.



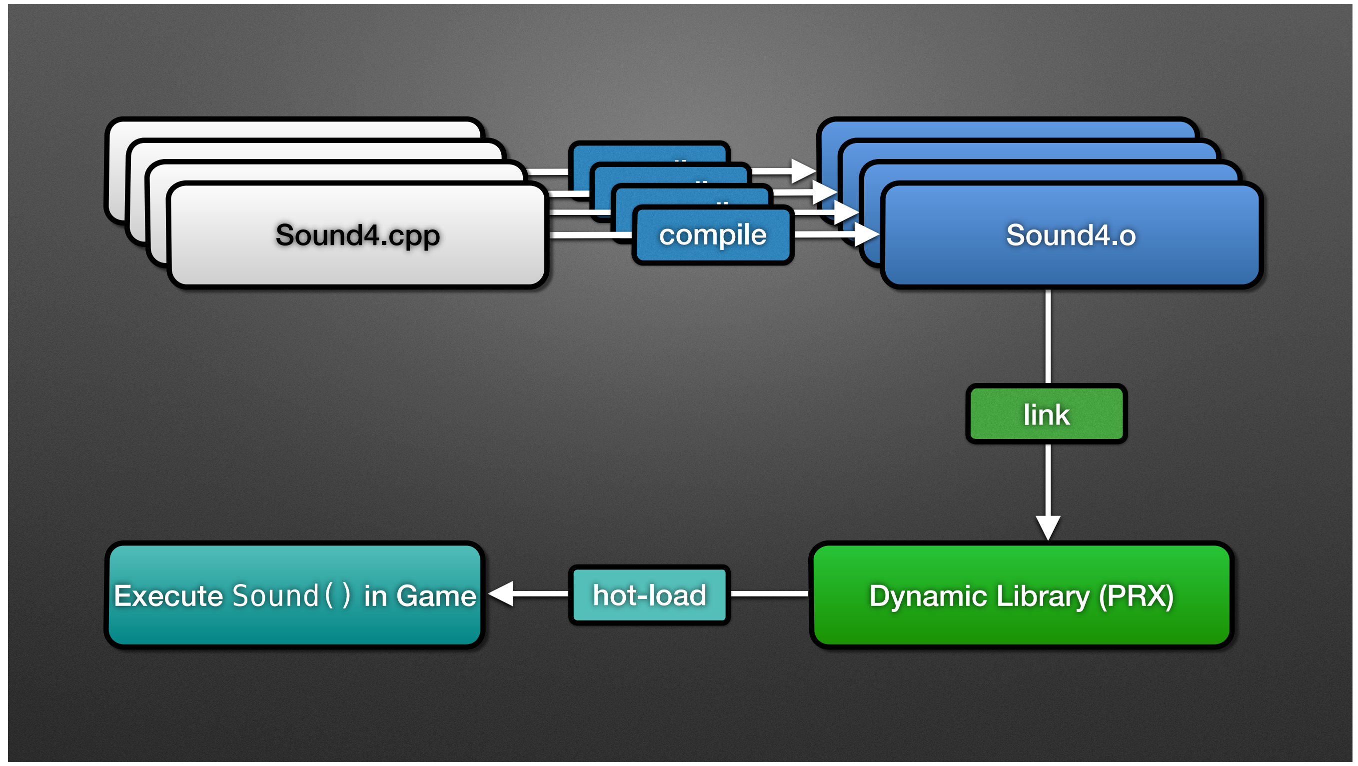
So as part of our **content pipeline** the C++ code that was generated for the graph is then **compiled** into an object file, and together with other object files gets **linked into a dynamic library**, which are loaded as PRX files on the PS4 at runtime. This allows us to execute the code in the game.
Our initial attempt was slightly different tough: We've first experimented with LLVM using byte code and the LLVM JIT compiler at runtime, which was nice for the initial testing and development. But soon after that we switched to a more traditional offline compilation step, using clang and dynamic libraries.



So as part of our **content pipeline** the C++ code that was generated for the graph is then **compiled** into an object file, and together with other object files gets **linked into a dynamic library**, which are loaded as PRX files on the PS4 at runtime. This allows us to execute the code in the game.
Our initial attempt was slightly different tough: We've first experimented with LLVM using byte code and the LLVM JIT compiler at runtime, which was nice for the initial testing and development. But soon after that we switched to a more traditional offline compilation step, using clang and dynamic libraries.



So as part of our **content pipeline** the C++ code that was generated for the graph is then **compiled** into an object file, and together with other object files gets **linked into a dynamic library**, which are loaded as PRX files on the PS4 at runtime. This allows us to execute the code in the game.
Our initial attempt was slightly different tough: We've first experimented with LLVM using byte code and the LLVM JIT compiler at runtime, which was nice for the initial testing and development. But soon after that we switched to a more traditional offline compilation step, using clang and dynamic libraries.

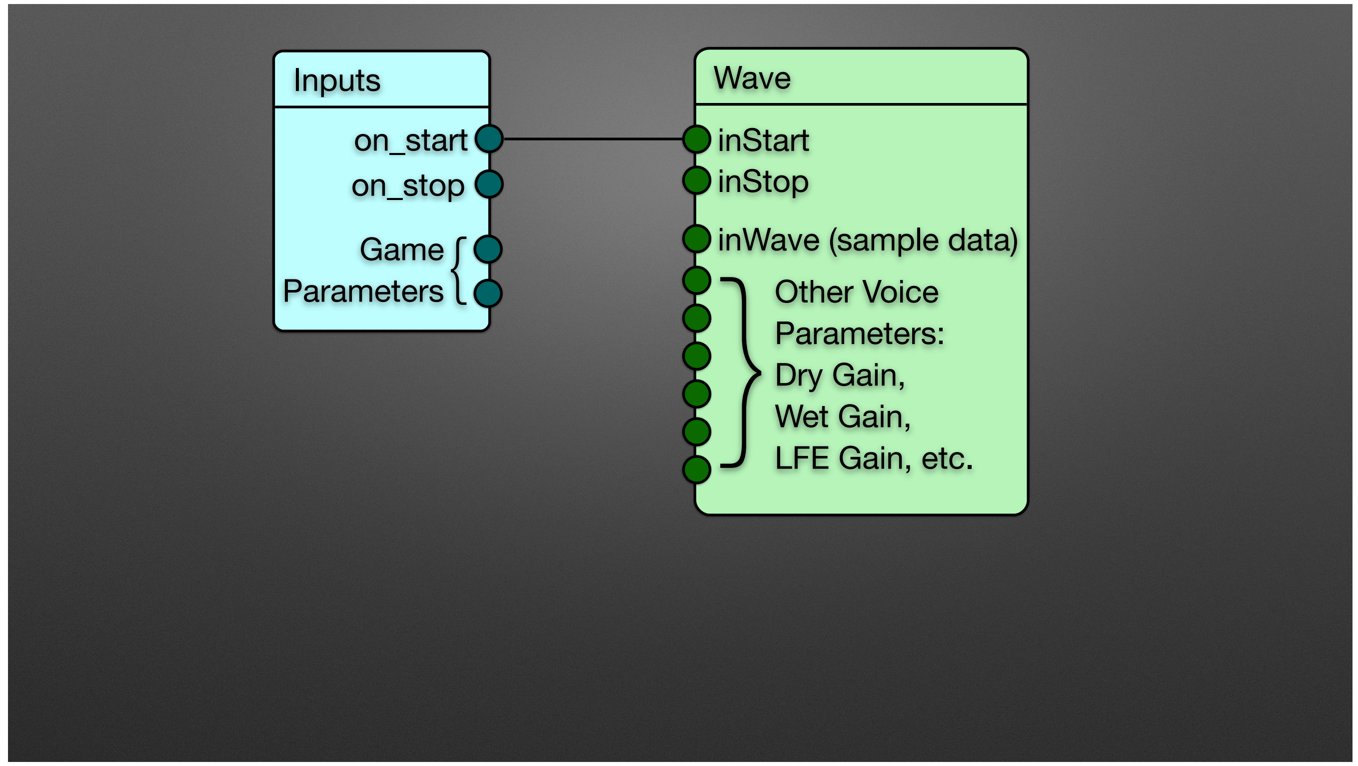


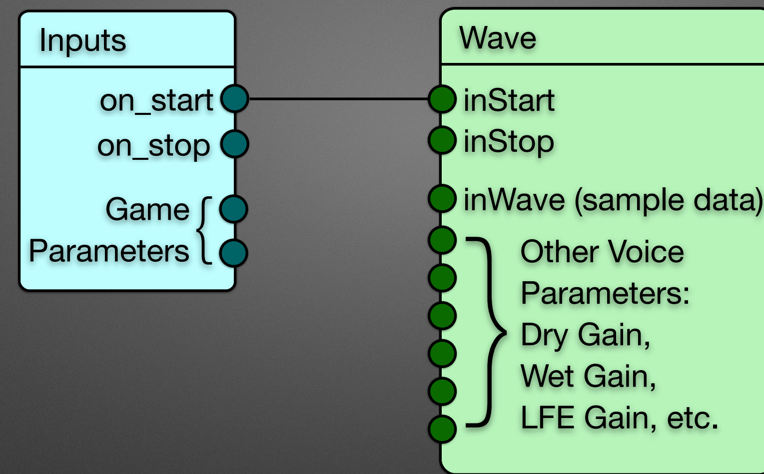
So as part of our **content pipeline** the C++ code that was generated for the graph is then **compiled** into an object file, and together with other object files gets **linked into a dynamic library**, which are loaded as PRX files on the PS4 at runtime. This allows us to execute the code in the game.

Our initial attempt was slightly different tough: We've first experimented with LLVM using byte code and the LLVM JIT compiler at runtime, which was nice for the initial testing and development. But soon after that we switched to a more traditional offline compilation step, using clang and dynamic libraries.

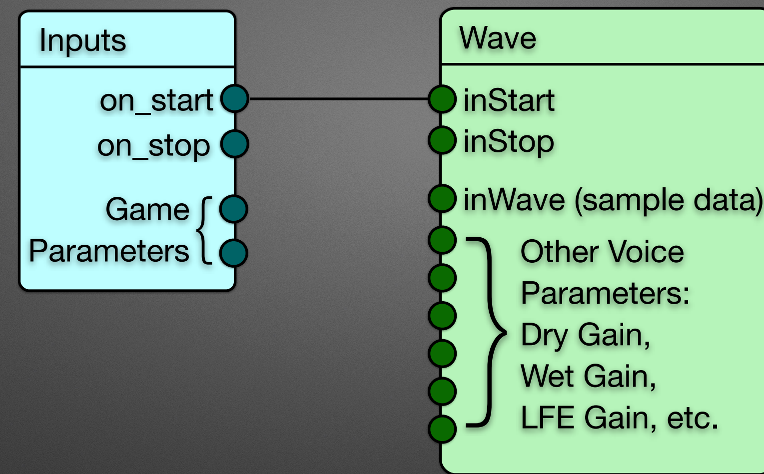
The most simple graph sound example we can come up with

Lets get to know the system by building a very basic sound

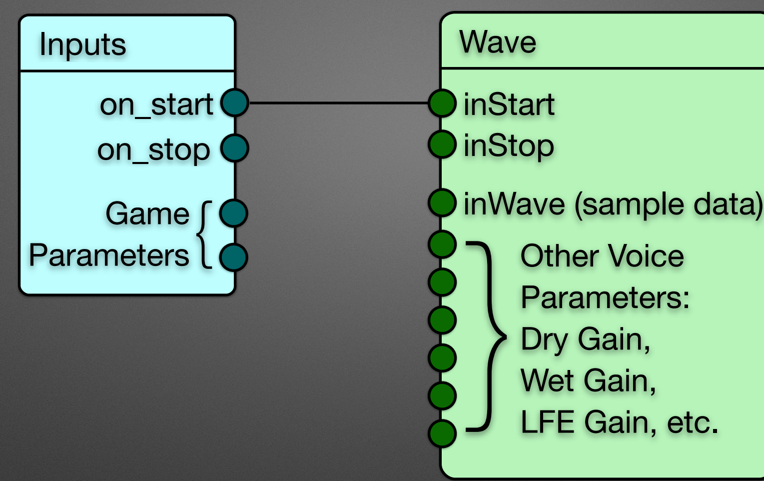

Inputs
on_start
on_stop
Game
Parameters
Wave
inStart
inStop
inWave (sample data)
Other Voice
Parameters:
Dry Gain,
Wet Gain,
LFE Gain, etc.



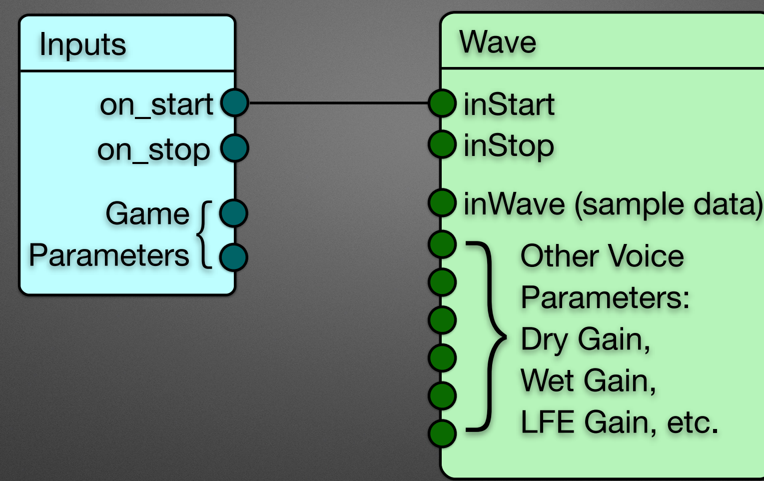
- **Wave** node is used to play sample data



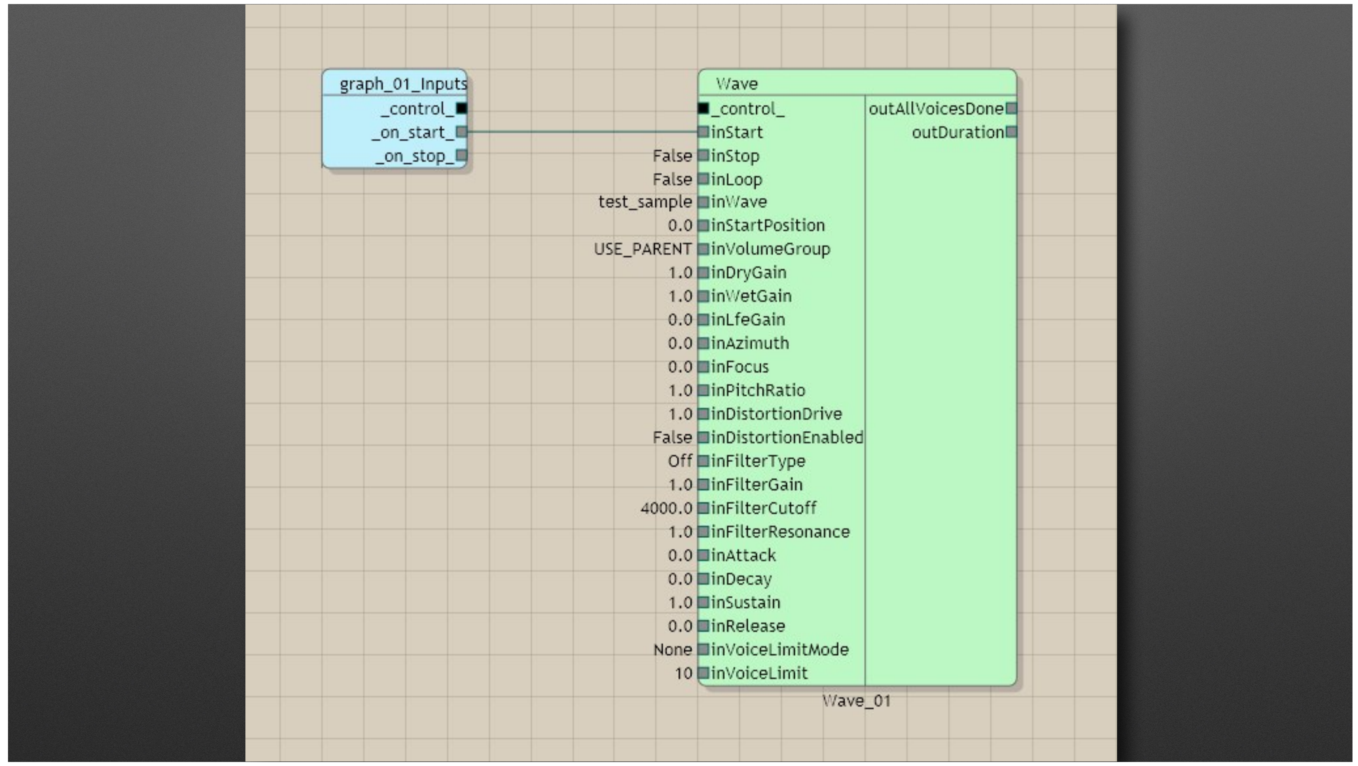
- **Wave** node is used to play sample data
- The **on_start** input is activated when the sound starts

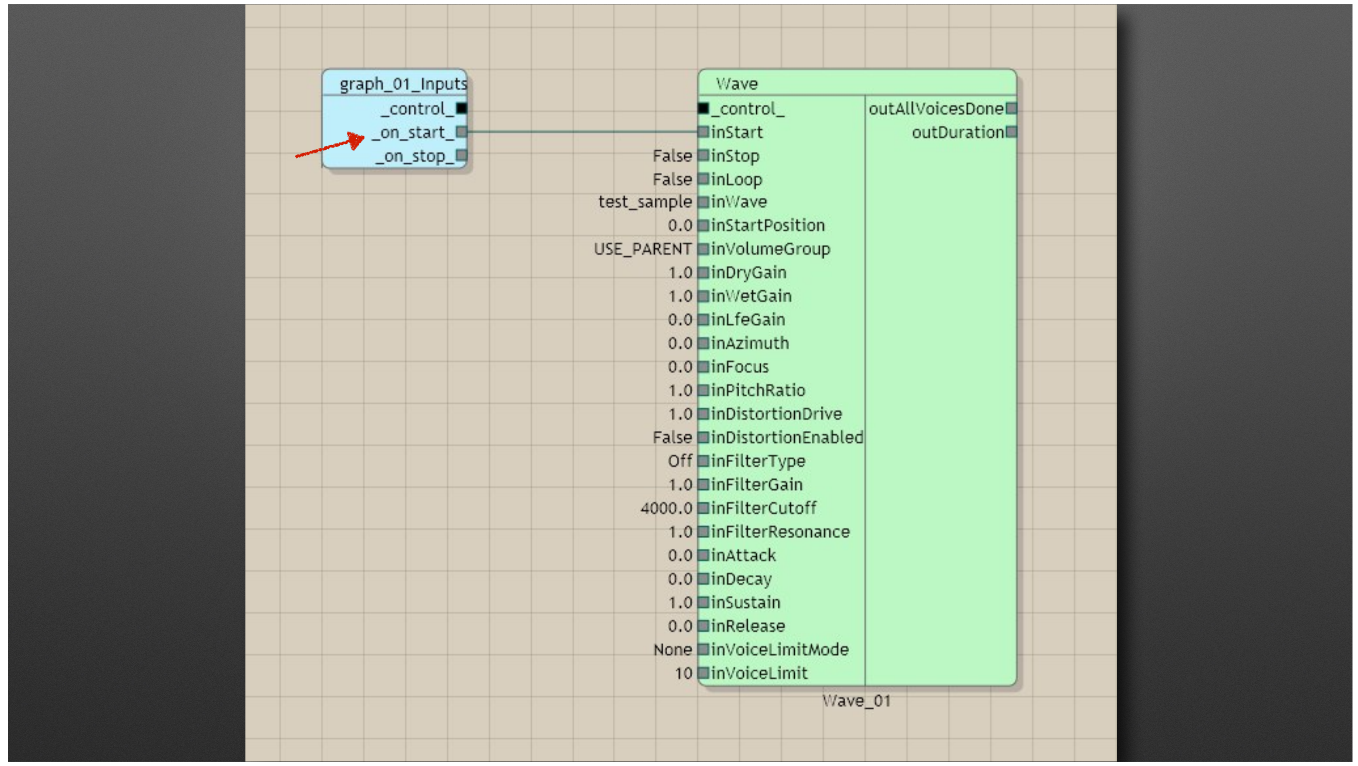


- **Wave** node is used to play sample data
- The **on_start** input is activated when the sound starts
- When **inStart** is active, a voice is created and played

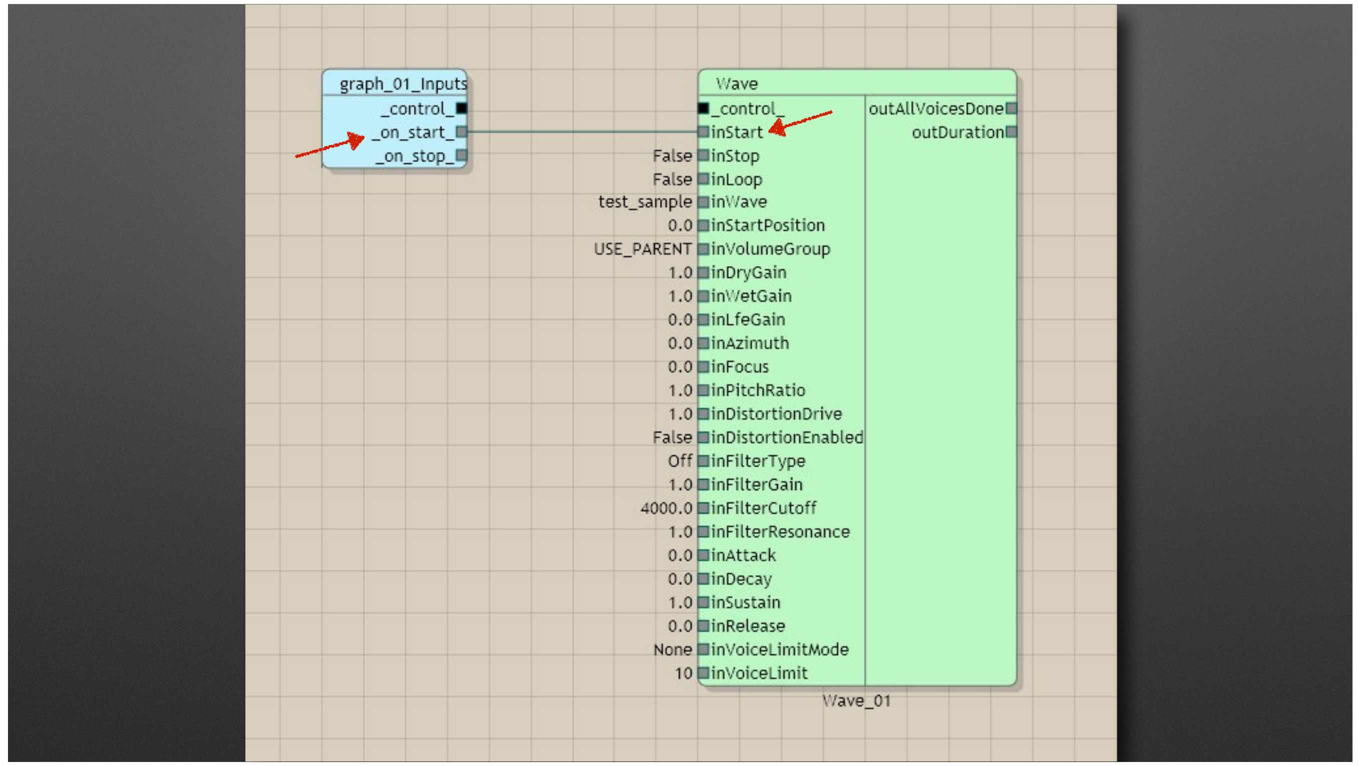


- **Wave** node is used to play sample data
- The **on_start** input is activated when the sound starts
- When **inStart** is active, a voice is created and played
- So this graph plays one sample when the sound starts

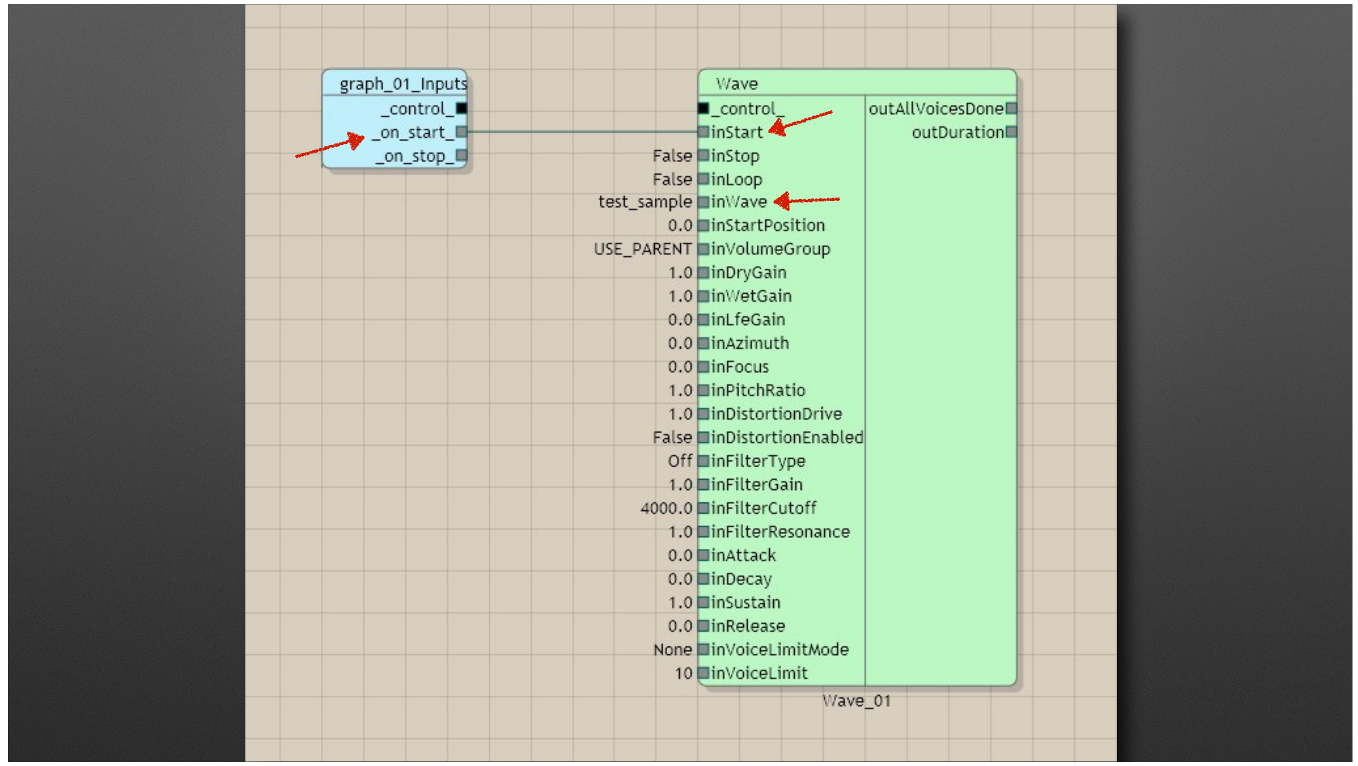

graph_01_Inputs
control
_on_start_
_on_stop_
Wave
control
inStart
False inStop
False inLoop
test_sample inWave
0.0 inStartPosition
USE_PARENT inVolumeGroup
1.0 inDryGain
1.0 inWetGain
0.0 inLfeGain
0.0 inAzimuth
0.0 inFocus
1.0 inPitchRatio
1.0 inDistortionDrive
False inDistortionEnabled
Off inFilterType
1.0 inFilterGain
4000.0 inFilterCutoff
1.0 inFilterResonance
0.0 inAttack
0.0 inDecay
1.0 inSustain
0.0 inRelease
None inVoiceLimitMode
10 inVoiceLimit
outAllVoicesDone
outDuration
Wave_01

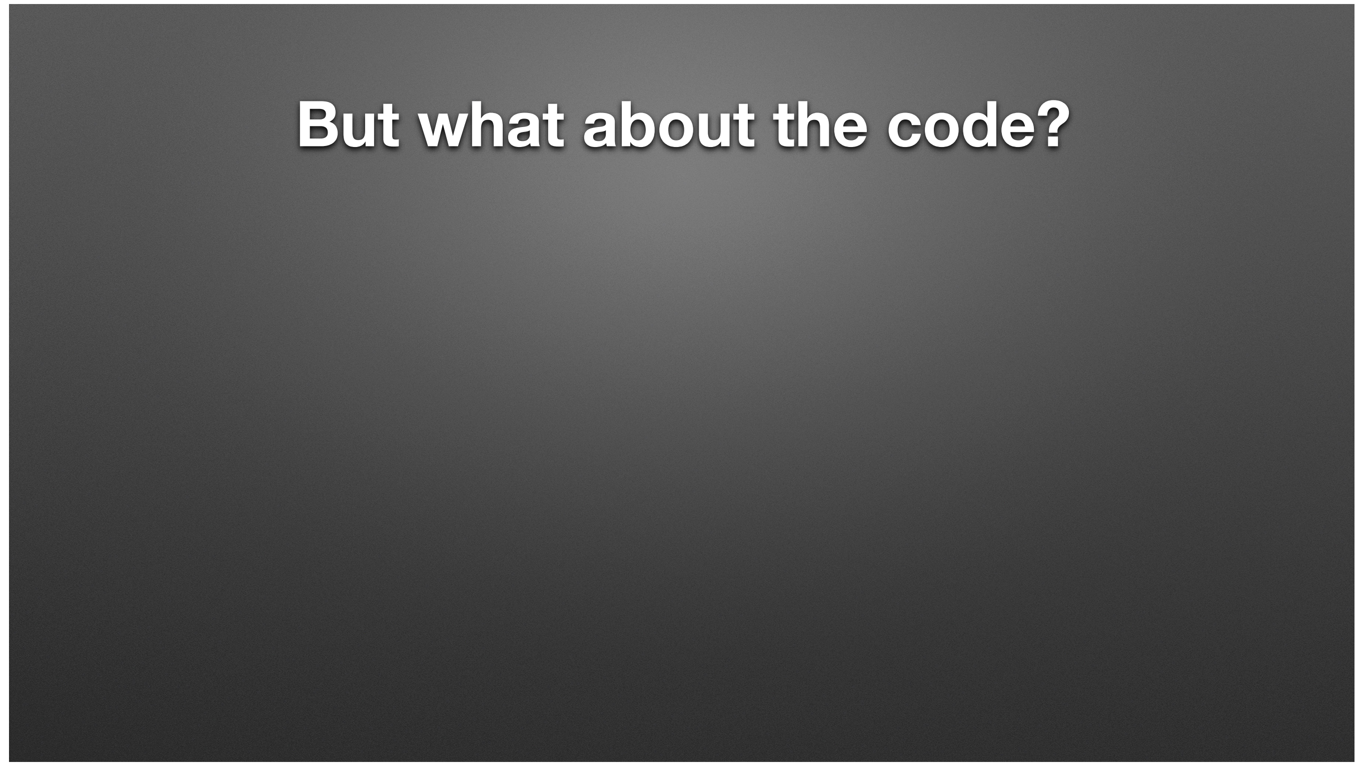


graph_01_Inputs
control
_on_start_
_on_stop_
Wave
control
inStart
False inStop
False inLoop
test_sample inWave
0.0 inStartPosition
USE_PARENT inVolumeGroup
1.0 inDryGain
1.0 inWetGain
0.0 inLfeGain
0.0 inAzimuth
0.0 inFocus
1.0 inPitchRatio
1.0 inDistortionDrive
False inDistortionEnabled
Off inFilterType
1.0 inFilterGain
4000.0 inFilterCutoff
1.0 inFilterResonance
0.0 inAttack
0.0 inDecay
1.0 inSustain
0.0 inRelease
None inVoiceLimitMode
10 inVoiceLimit
outAllVoicesDone
outDuration
Wave_01



graph_01_Inputs
control
_on_start_
_on_stop_
Wave
control
inStart
False inStop
False inLoop
test_sample inWave
0.0 inStartPosition
USE_PARENT inVolumeGroup
1.0 inDryGain
1.0 inWetGain
0.0 inLfeGain
0.0 inAzimuth
0.0 inFocus
1.0 inPitchRatio
1.0 inDistortionDrive
False inDistortionEnabled
Off inFilterType
1.0 inFilterGain
4000.0 inFilterCutoff
1.0 inFilterResonance
0.0 inAttack
0.0 inDecay
1.0 inSustain
0.0 inRelease
None inVoiceLimitMode
10 inVoiceLimit
outAllVoicesDone
outDuration
Wave_01


graph_01_Inputs
control
_on_start_
_on_stop_
Wave
control
inStart
False inStop
False inLoop
test_sample inWave
0.0 inStartPosition
USE_PARENT inVolumeGroup
1.0 inDryGain
1.0 inWetGain
0.0 inLfeGain
0.0 inAzimuth
0.0 inFocus
1.0 inPitchRatio
1.0 inDistortionDrive
False inDistortionEnabled
Off inFilterType
1.0 inFilterGain
4000.0 inFilterCutoff
1.0 inFilterResonance
0.0 inAttack
0.0 inDecay
1.0 inSustain
0.0 inRelease
None inVoiceLimitMode
10 inVoiceLimit
outAllVoicesDone
outDuration
Wave_01



Andreas: This is the **result of the code generator**, it's fairly readable.

You can see the **Wave node function call** and how the **on_start input** of the graph is passed as **a parameter**. The **sample** is a constant **resource attached to the graph** and can be retrieved with this function, which is implemented in the engine code.

Clang's optimizer creates really **tight** assembly **code** from this. It's super **fast**, so we can **run it at a high rate**, currently once per synth frame, i.e. **every 5.3 ms**.

As you can see, **every input of the graph**, becomes a **parameter of the graph function**, and each **node input** is also a **parameter of the node function**.

But what about the code?

```
void Sound(bool _on_start_, bool _on_stop_)
{
   bool outAllVoicesDone = false;
   float outDuration = 0;

   Wave(_on_start_, false, false,
        WaveDataPointer, VoiceParameters, ...,
        &outAllVoicesDone, &outDuration);
}
```

Andreas: This is the **result of the code generator**, it's fairly readable.

You can see the **Wave node function call** and how the **on_start input** of the graph is passed as **a parameter**. The **sample** is a constant **resource attached to the graph** and can be retrieved with this function, which is implemented in the engine code.

Clang's optimizer creates really **tight** assembly **code** from this. It's super **fast**, so we can **run it at a high rate**, currently once per synth frame, i.e. **every 5.3 ms**.

As you can see, **every input of the graph**, becomes a **parameter of the graph function**, and each **node input** is also a **parameter of the node function**.

But what about the code?

```
void Sound(bool _on_start_, bool _on_stop_)
{
   bool outAllVoicesDone = false;
   float outDuration = 0;

   Wave(_on_start_, false, false,
        WaveDataPointer, VoiceParameters, ...,
        &outAllVoicesDone, &outDuration);
}
```

Andreas: This is the **result of the code generator**, it's fairly readable.

You can see the **Wave node function call** and how the **on_start input** of the graph is passed as **a parameter**. The **sample** is a constant **resource attached to the graph** and can be retrieved with this function, which is implemented in the engine code.

Clang's optimizer creates really **tight** assembly **code** from this. It's super **fast**, so we can **run it at a high rate**, currently once per synth frame, i.e. **every 5.3 ms**.

As you can see, **every input of the graph**, becomes a **parameter of the graph function**, and each **node input** is also a **parameter of the node function**.

But what about the code?

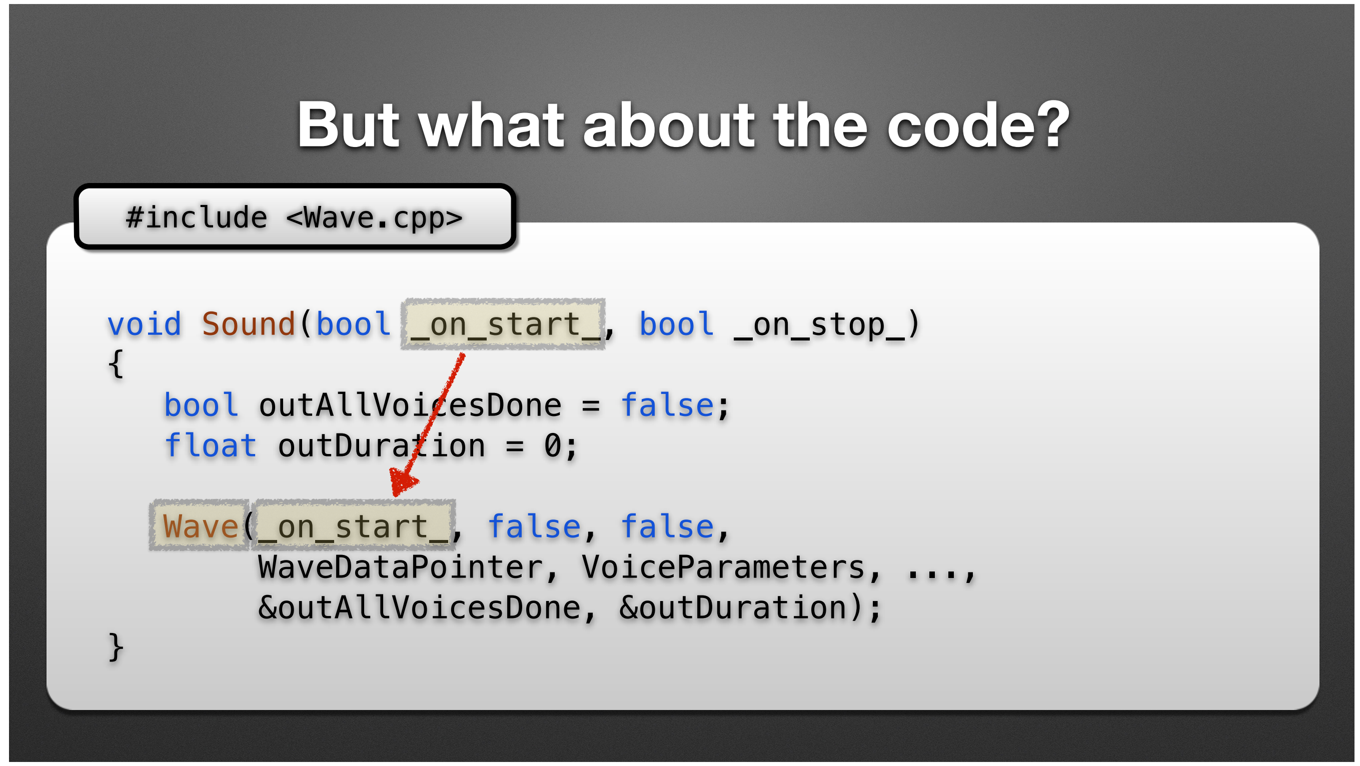
```
void Sound(bool _on_start_, bool _on_stop_)
{
   bool outAllVoicesDone = false;
   float outDuration = 0;

   Wave(_on_start_, false, false,
        WaveDataPointer, VoiceParameters, ...,
        &outAllVoicesDone, &outDuration);
}
```

Andreas: This is the **result of the code generator**, it's fairly readable.

You can see the **Wave node function call** and how the **on_start input** of the graph is passed as **a parameter**. The **sample** is a constant **resource attached to the graph** and can be retrieved with this function, which is implemented in the engine code.

Clang's optimizer creates really **tight** assembly **code** from this. It's super **fast**, so we can **run it at a high rate**, currently once per synth frame, i.e. **every 5.3 ms**.

As you can see, **every input of the graph**, becomes a **parameter of the graph function**, and each **node input** is also a **parameter of the node function**.

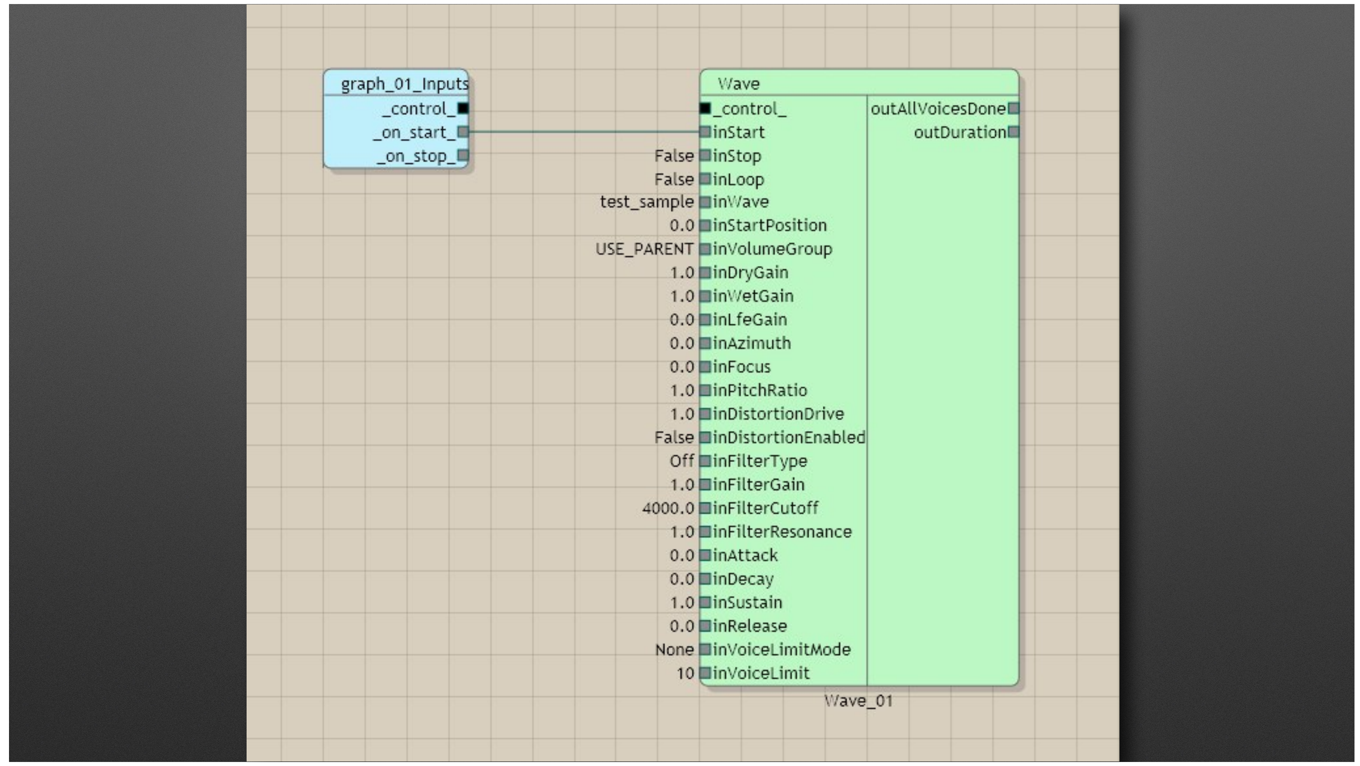


Andreas: This is the **result of the code generator**, it's fairly readable.

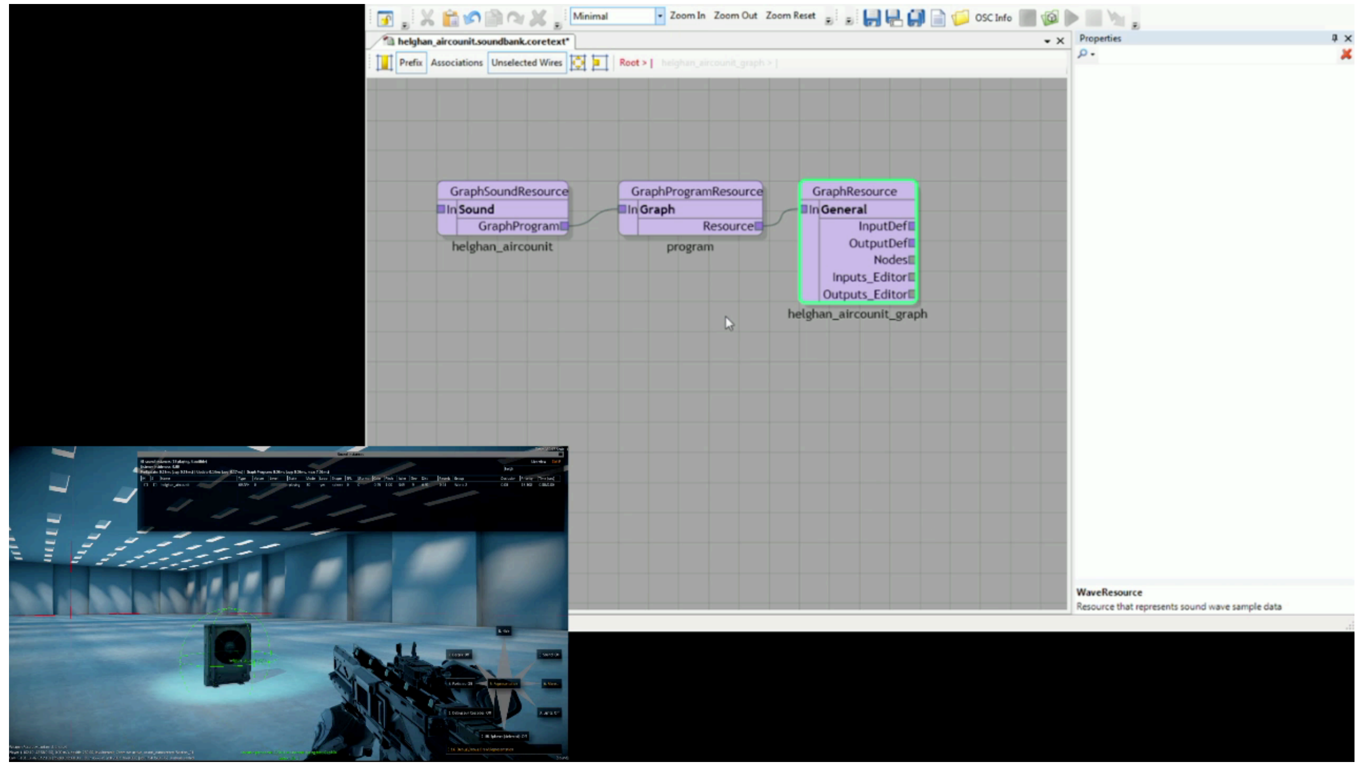
You can see the **Wave node function call** and how the **on_start input** of the graph is passed as **a parameter**. The **sample** is a constant **resource attached to the graph** and can be retrieved with this function, which is implemented in the engine code.

Clang's optimizer creates really **tight** assembly **code** from this. It's super **fast**, so we can **run it at a high rate**, currently once per synth frame, i.e. **every 5.3 ms**.

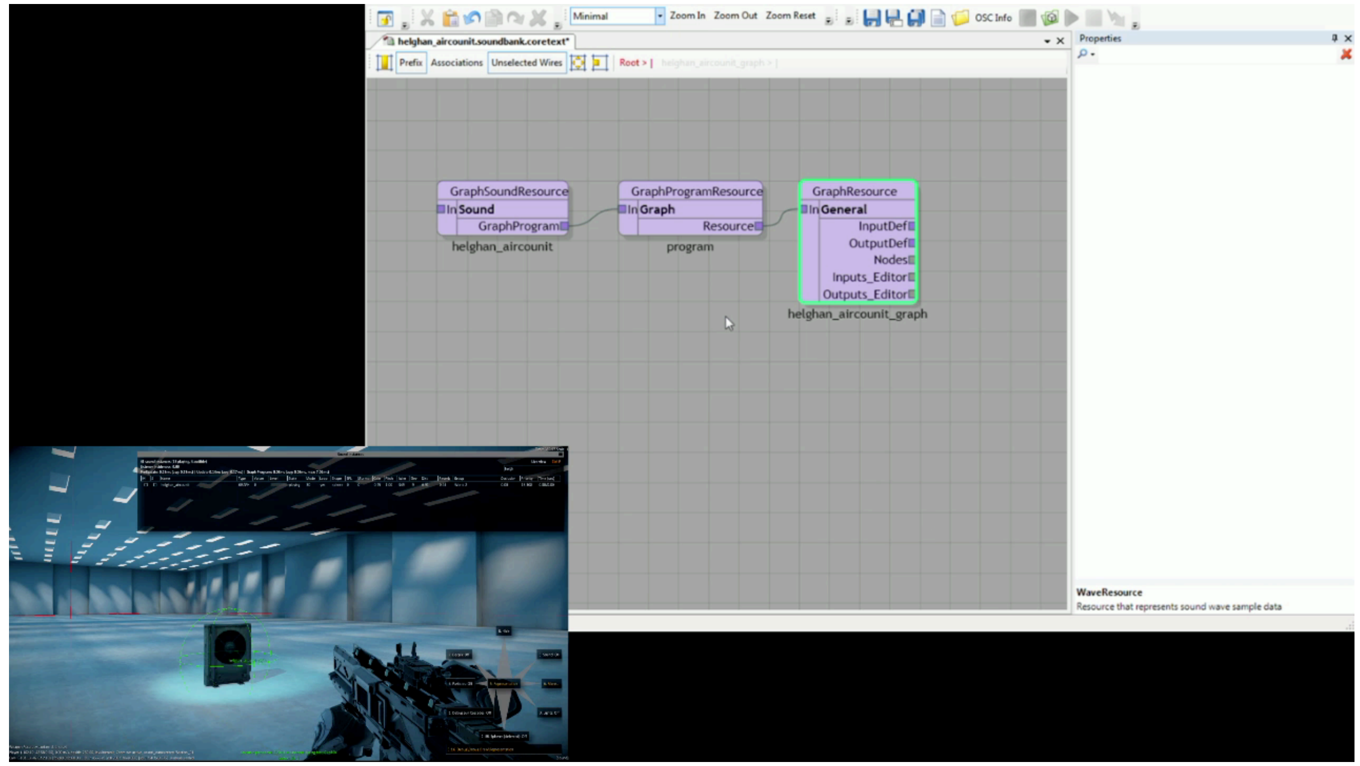
As you can see, **every input of the graph**, becomes a **parameter of the graph function**, and each **node input** is also a **parameter of the node function**.



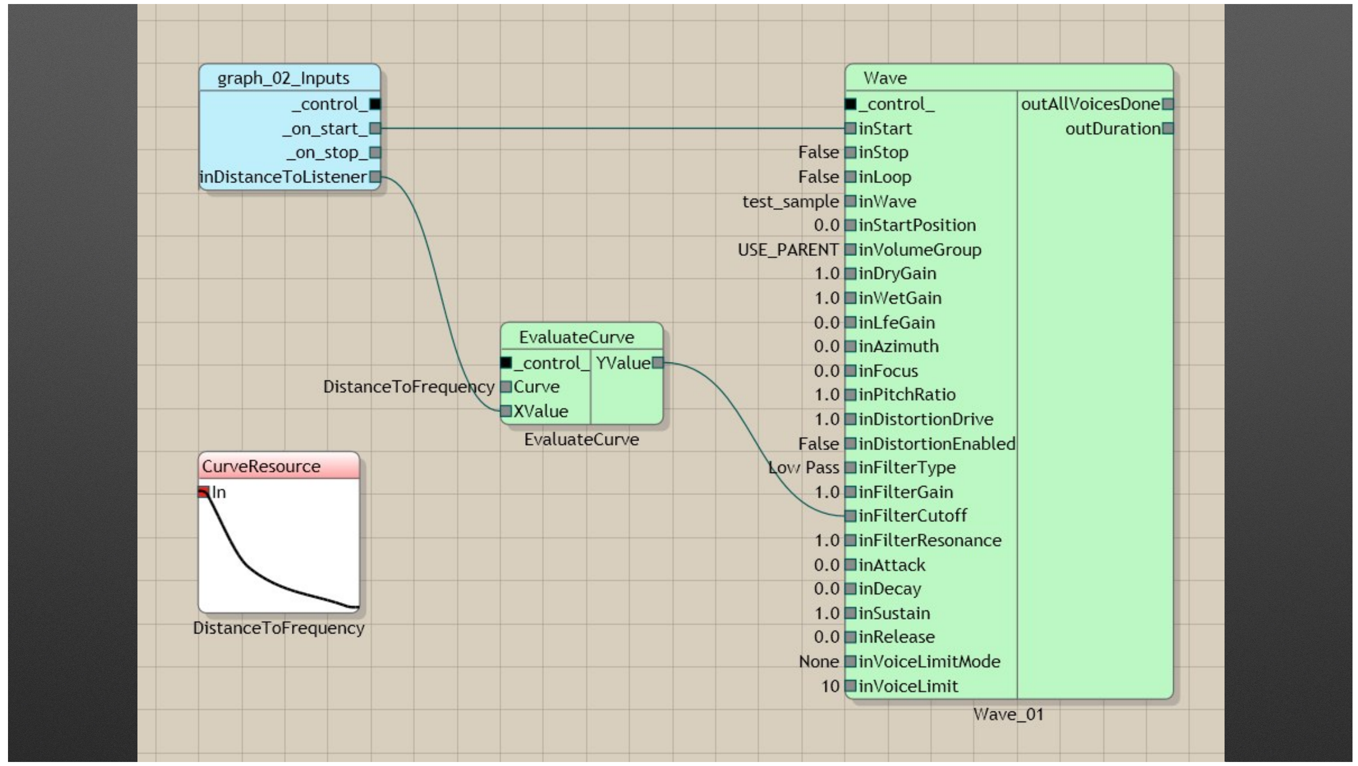
But the nice thing is, that the **code is not immediately visible**. It's there, and as a **programmer** you **can use it to debug** what's going on, but a **sound designer** would **only see the graph** representation.



Anton: Now **let's inject the sound into the game** and **play it**, for testing purposes.
Video showing how a sound is created from scratch and injected into the running game.

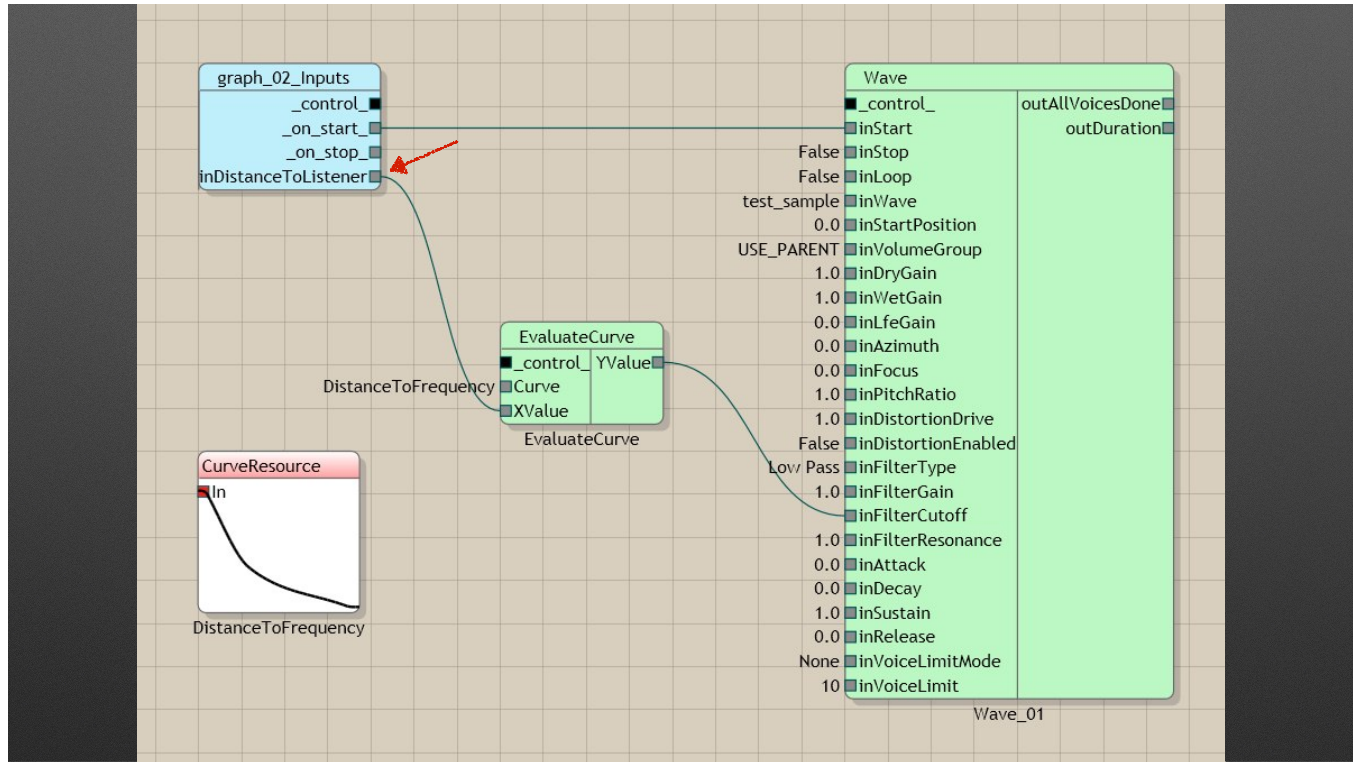


Anton: Now **let's inject the sound into the game** and **play it**, for testing purposes.
Video showing how a sound is created from scratch and injected into the running game.



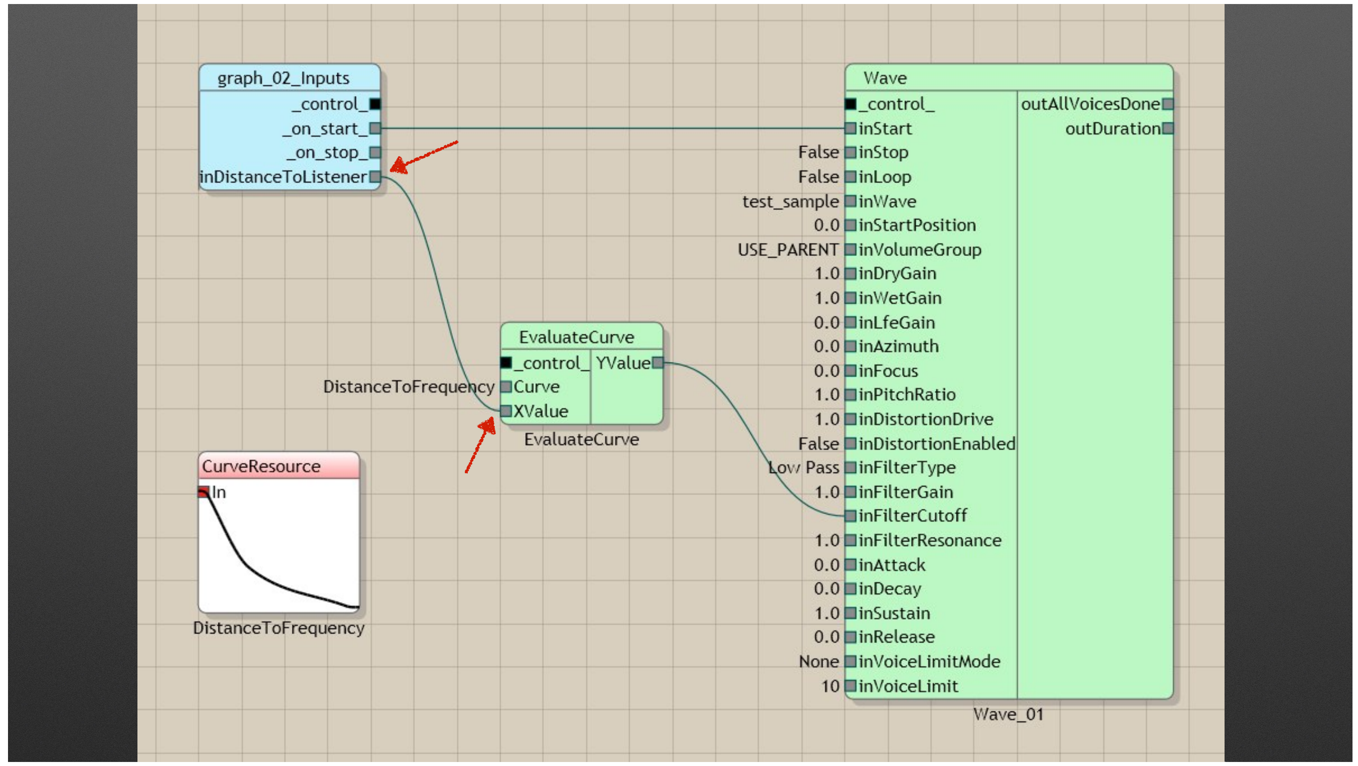
Andreas: Now lets **add** some more **dynamic behaviour to the sound**. As you can see in the blue inputs node, I've **added an input** called **"inDistanceToListener"** which is filled in by the engine with the **distance in meters between** this **sound and** the **listener**. I use this distance as the **X value** of a **curve lookup** node, and use the **Y value** as the **cutoff frequency** of a **low pass filter** of the wave node. The rest is the same as in the previous example.

Btw: As you can see, the gain inputs are all linear values, from 0 to 1. This is not very sound designer friendly, but it makes it easier to do certain calculations. In case you prefer dB full-scale, then we have a very simple node you can put in front to convert from dB to linear values.



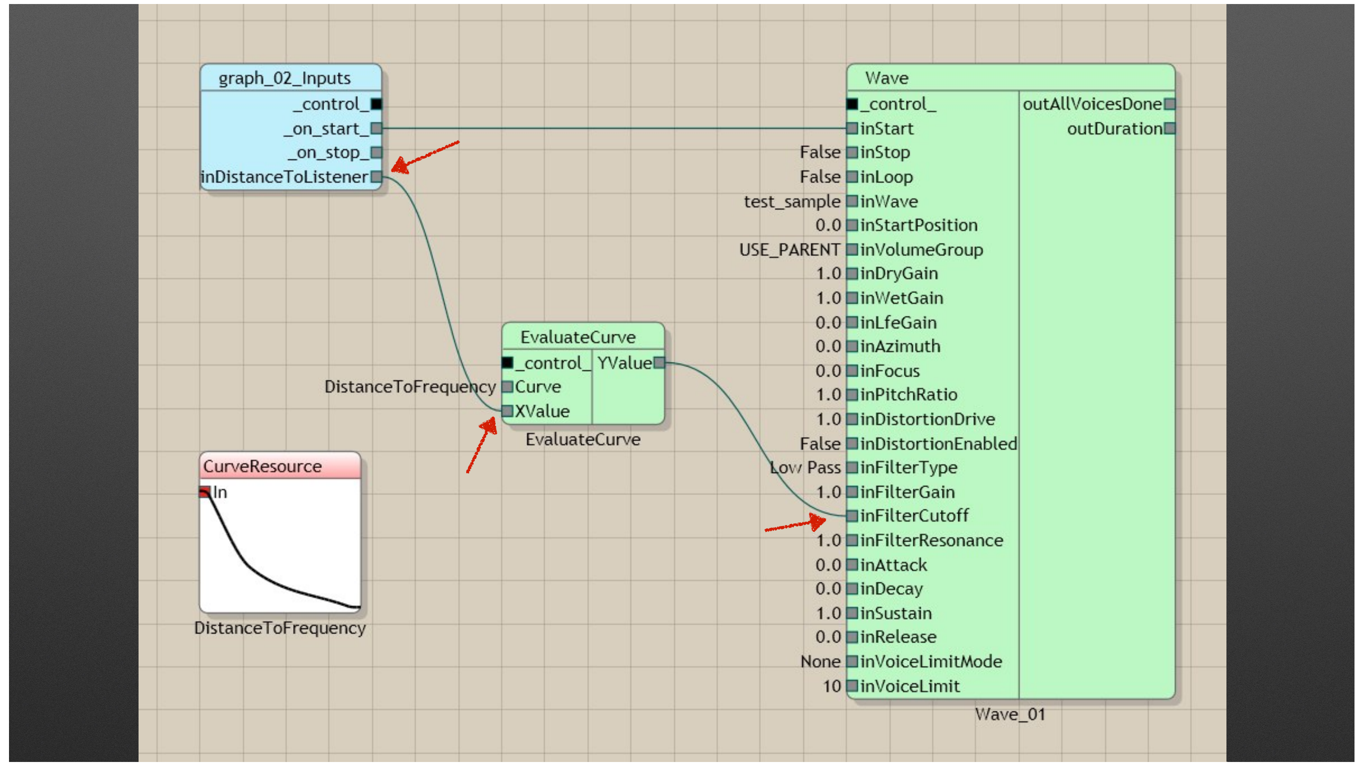
Andreas: Now lets **add** some more **dynamic behaviour to the sound**. As you can see in the blue inputs node, I've **added an input** called **"inDistanceToListener"** which is filled in by the engine with the **distance in meters between** this **sound and** the **listener**. I use this distance as the **X value** of a **curve lookup** node, and use the **Y value** as the **cutoff frequency** of a **low pass filter** of the wave node. The rest is the same as in the previous example.

Btw: As you can see, the gain inputs are all linear values, from 0 to 1. This is not very sound designer friendly, but it makes it easier to do certain calculations. In case you prefer dB full-scale, then we have a very simple node you can put in front to convert from dB to linear values.



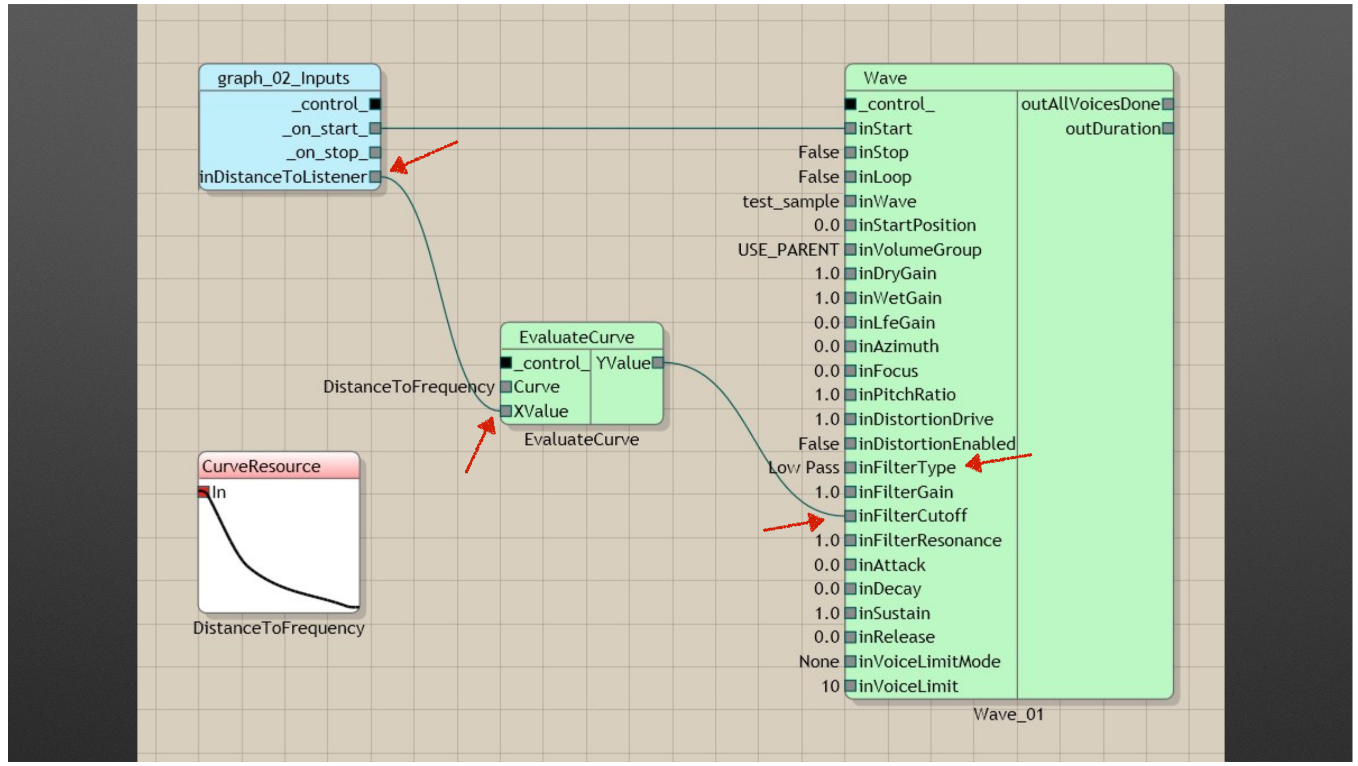
Andreas: Now lets **add** some more **dynamic behaviour to the sound**. As you can see in the blue inputs node, I've **added an input** called **"inDistanceToListener"** which is filled in by the engine with the **distance in meters between** this **sound and** the **listener**. I use this distance as the **X value** of a **curve lookup** node, and use the **Y value** as the **cutoff frequency** of a **low pass filter** of the wave node. The rest is the same as in the previous example.

Btw: As you can see, the gain inputs are all linear values, from 0 to 1. This is not very sound designer friendly, but it makes it easier to do certain calculations. In case you prefer dB full-scale, then we have a very simple node you can put in front to convert from dB to linear values.



Andreas: Now lets **add** some more **dynamic behaviour to the sound**. As you can see in the blue inputs node, I've **added an input** called **"inDistanceToListener"** which is filled in by the engine with the **distance in meters between** this **sound and** the **listener**. I use this distance as the **X value** of a **curve lookup** node, and use the **Y value** as the **cutoff frequency** of a **low pass filter** of the wave node. The rest is the same as in the previous example.

Btw: As you can see, the gain inputs are all linear values, from 0 to 1. This is not very sound designer friendly, but it makes it easier to do certain calculations. In case you prefer dB full-scale, then we have a very simple node you can put in front to convert from dB to linear values.



Andreas: Now lets **add** some more **dynamic behaviour to the sound**. As you can see in the blue inputs node, I've **added an input** called **"inDistanceToListener"** which is filled in by the engine with the **distance in meters between** this **sound and** the **listener**. I use this distance as the **X value** of a **curve lookup** node, and use the **Y value** as the **cutoff frequency** of a **low pass filter** of the wave node. The rest is the same as in the previous example.

Btw: As you can see, the gain inputs are all linear values, from 0 to 1. This is not very sound designer friendly, but it makes it easier to do certain calculations. In case you prefer dB full-scale, then we have a very simple node you can put in front to convert from dB to linear values.

```
void Sound(bool _on_start_, bool _on_stop_,
           float inDistanceToListener)
{
   float outYValue = 0;
   EvaluateCurve(CurveDataPointer, inDistanceToListener,
                 &outYValue);

   bool outAllVoicesDone = false;
   float outDuration = 0;
   Wave(_on_start_, false, false,
        WaveDataPointer, VoiceParameters, outYValue,
        &outAllVoicesDone, &outDuration);
}
```

It's getting **a bit more complicated** now, but it's still easy enough to see what's going on. The **inDistanceToListener input of the graph** becomes a new **graph function parameter**. The **Evaluate node** is now called, and since the **Wave node depends on** the **output of the Evaluate node**, the Wave node **has to be called after** the Evaluate node. The **code generator uses the dependencies** to define the **order** in which it needs to call each node's function.

```
void Sound(bool _on_start_, bool _on_stop_,
           float inDistanceToListener)
{
   float outYValue = 0;
   EvaluateCurve(CurveDataPointer, inDistanceToListener,
                 &outYValue);

   bool outAllVoicesDone = false;
   float outDuration = 0;
   Wave(_on_start_, false, false,
        WaveDataPointer, VoiceParameters, outYValue,
        &outAllVoicesDone, &outDuration);
}
```

It's getting **a bit more complicated** now, but it's still easy enough to see what's going on. The **inDistanceToListener input of the graph** becomes a new **graph function parameter**. The **Evaluate node** is now called, and since the **Wave node depends on** the **output of the Evaluate node**, the Wave node **has to be called after** the Evaluate node. The **code generator uses the dependencies** to define the **order** in which it needs to call each node's function.

```
void Sound(bool _on_start_, bool _on_stop_,
           float inDistanceToListener)
{
   float outYValue = 0;
   EvaluateCurve(CurveDataPointer, inDistanceToListener,
                 &outYValue);

   bool outAllVoicesDone = false;
   float outDuration = 0;
   Wave(_on_start_, false, false,
        WaveDataPointer, VoiceParameters, outYValue,
        &outAllVoicesDone, &outDuration);
}
```

It's getting **a bit more complicated** now, but it's still easy enough to see what's going on. The **inDistanceToListener input of the graph** becomes a new **graph function parameter**. The **Evaluate node** is now called, and since the **Wave node depends on** the **output of the Evaluate node**, the Wave node **has to be called after** the Evaluate node. The **code generator uses the dependencies** to define the **order** in which it needs to call each node's function.

```
void Sound(bool _on_start_, bool _on_stop_,
           float inDistanceToListener)
{
   float outYValue = 0;
   EvaluateCurve(CurveDataPointer, inDistanceToListener,
                 &outYValue);

   bool outAllVoicesDone = false;
   float outDuration = 0;
   Wave(_on_start_, false, false,
        WaveDataPointer, VoiceParameters, outYValue,
        &outAllVoicesDone, &outDuration);
}
```

It's getting **a bit more complicated** now, but it's still easy enough to see what's going on. The **inDistanceToListener input of the graph** becomes a new **graph function parameter**. The **Evaluate node** is now called, and since the **Wave node depends on** the **output of the Evaluate node**, the Wave node **has to be called after** the Evaluate node. The **code generator uses the dependencies** to define the **order** in which it needs to call each node's function.

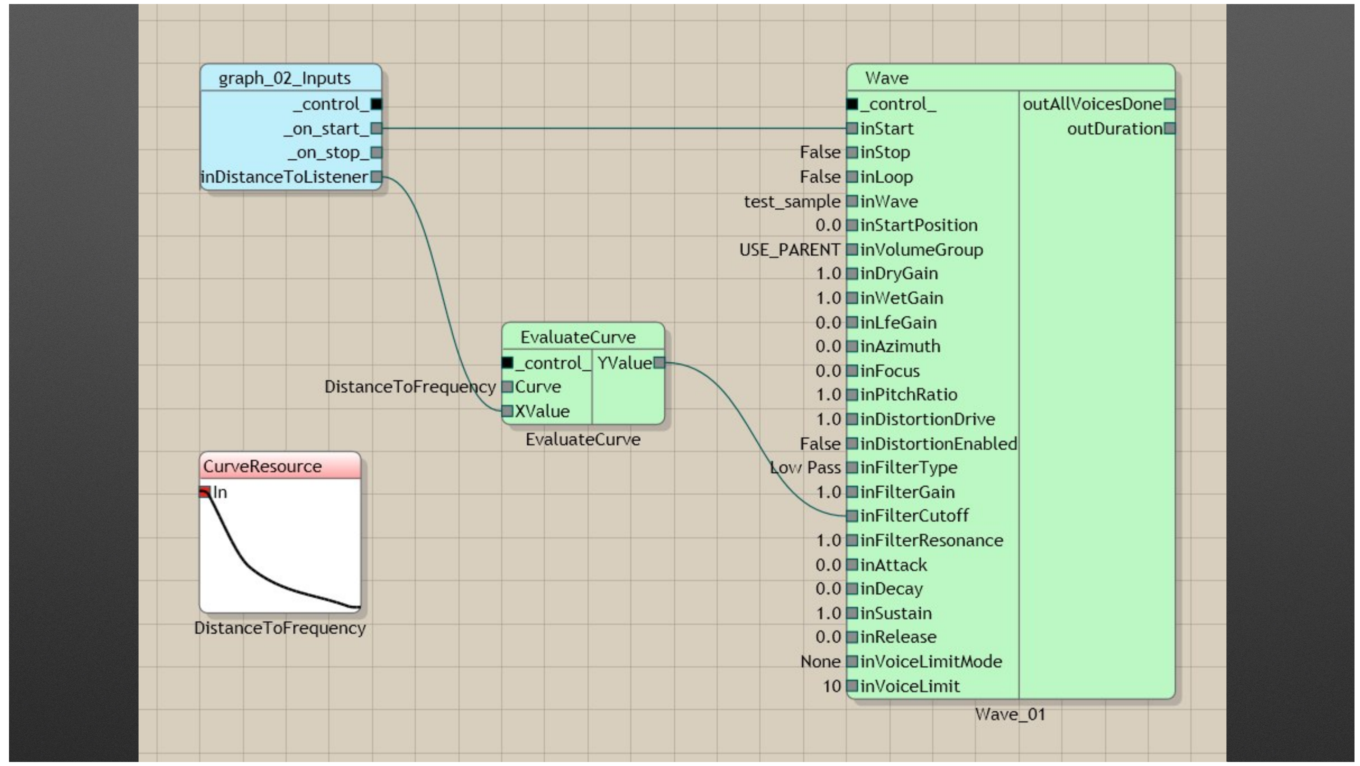
```
void Sound(bool _on_start_, bool _on_stop_,
           float inDistanceToListener)
{
   float outYValue = 0;
   EvaluateCurve(CurveDataPointer, inDistanceToListener,
                 &outYValue);

   bool outAllVoicesDone = false;
   float outDuration = 0;
   Wave(_on_start_, false, false,
        WaveDataPointer, VoiceParameters, outYValue,
        &outAllVoicesDone, &outDuration);
}
```

It's getting **a bit more complicated** now, but it's still easy enough to see what's going on. The **inDistanceToListener input of the graph** becomes a new **graph function parameter**. The **Evaluate node** is now called, and since the **Wave node depends on** the **output of the Evaluate node**, the Wave node **has to be called after** the Evaluate node. The **code generator uses the dependencies** to define the **order** in which it needs to call each node's function.

```
#include <EvaluateCurve.cpp>
#include <Wave.cpp>
```

```
void Sound(bool _on_start_, bool _on_stop_,
           float inDistanceToListener)
{
   float outYValue = 0;
   EvaluateCurve(CurveDataPointer, inDistanceToListener,
                 &outYValue);

   bool outAllVoicesDone = false;
   float outDuration = 0;
   Wave(_on_start_, false, false,
        WaveDataPointer, VoiceParameters, outYValue,
        &outAllVoicesDone, &outDuration);
}
```

It's getting **a bit more complicated** now, but it's still easy enough to see what's going on. The **inDistanceToListener input of the graph** becomes a new **graph function parameter**. The **Evaluate node** is now called, and since the **Wave node depends on** the **output of the Evaluate node**, the Wave node **has to be called after** the Evaluate node. The **code generator uses the dependencies** to define the **order** in which it needs to call each node's function.

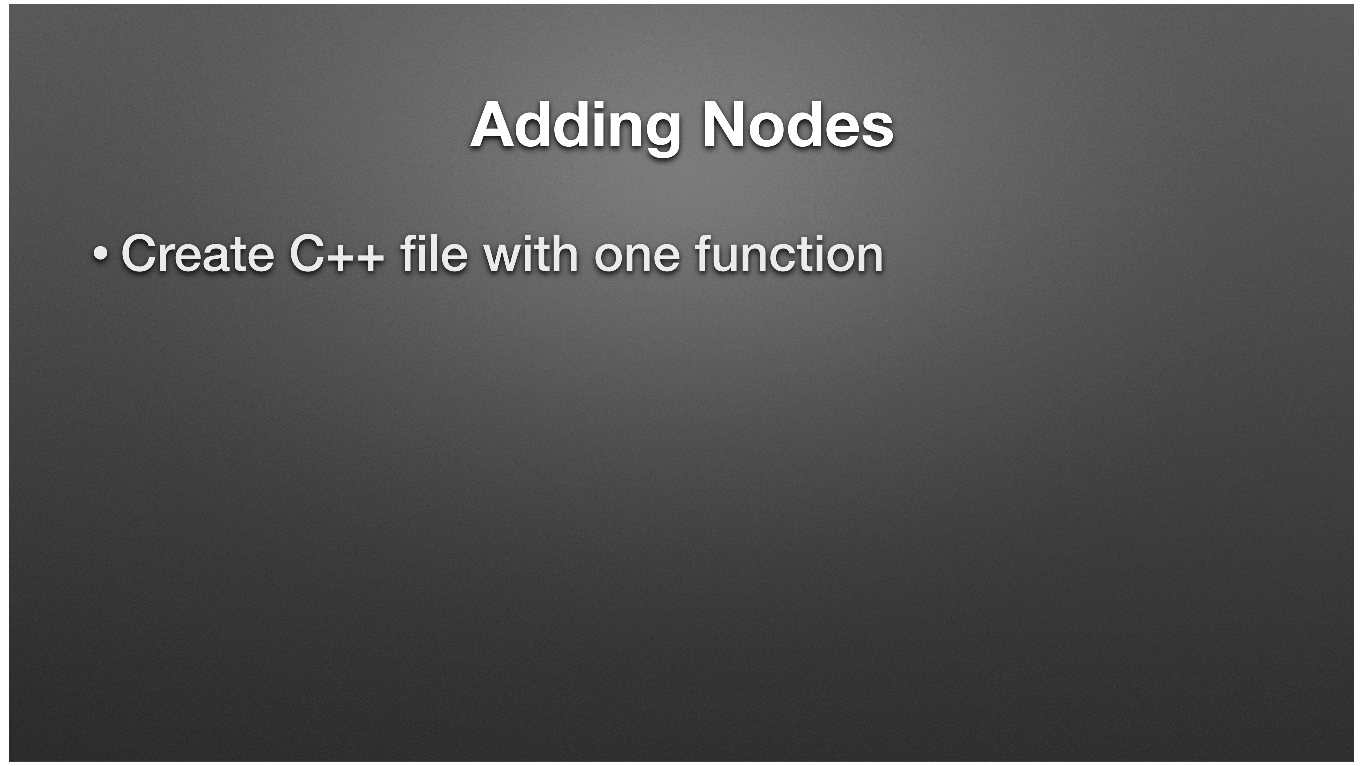

graph_02_Inputs
control
_on_start_
_on_stop_
inDistanceToListener
EvaluateCurve
control
YValue
DistanceToFrequency
Curve
XValue
EvaluateCurve
CurveResource
In
DistanceToFrequency
Wave
control
inStart
False inStop
False inLoop
test_sample inWave
0.0 inStartPosition
USE_PARENT inVolumeGroup
1.0 inDryGain
1.0 inWetGain
0.0 inLfeGain
0.0 inAzimuth
0.0 inFocus
1.0 inPitchRatio
1.0 inDistortionDrive
False inDistortionEnabled
Low Pass inFilterType
1.0 inFilterGain
inFilterCutoff
1.0 inFilterResonance
0.0 inAttack
0.0 inDecay
1.0 inSustain
0.0 inRelease
None inVoiceLimitMode
10 inVoiceLimit
outAllVoicesDone
outDuration
Wave_01

Video showing effect of distance-based low pass filtering example

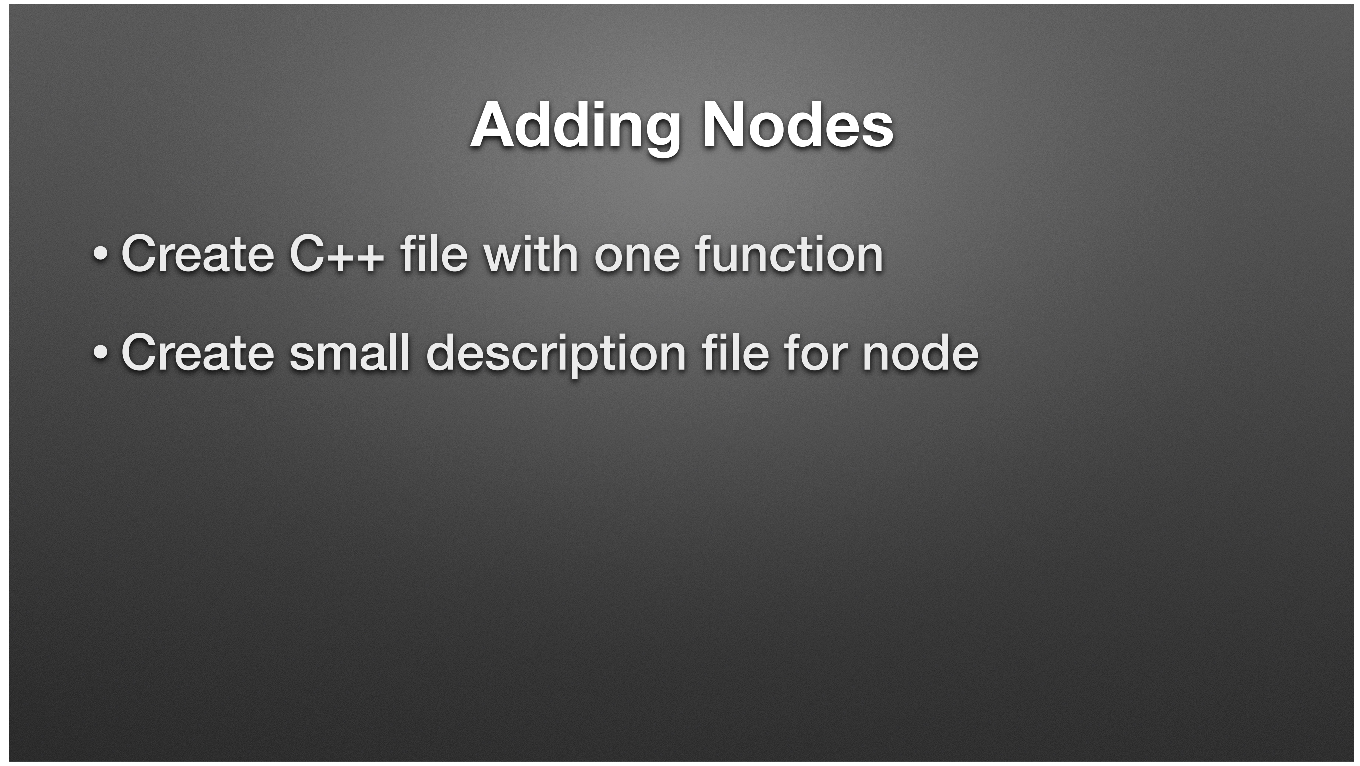
Video showing effect of distance-based low pass filtering example

Adding Nodes

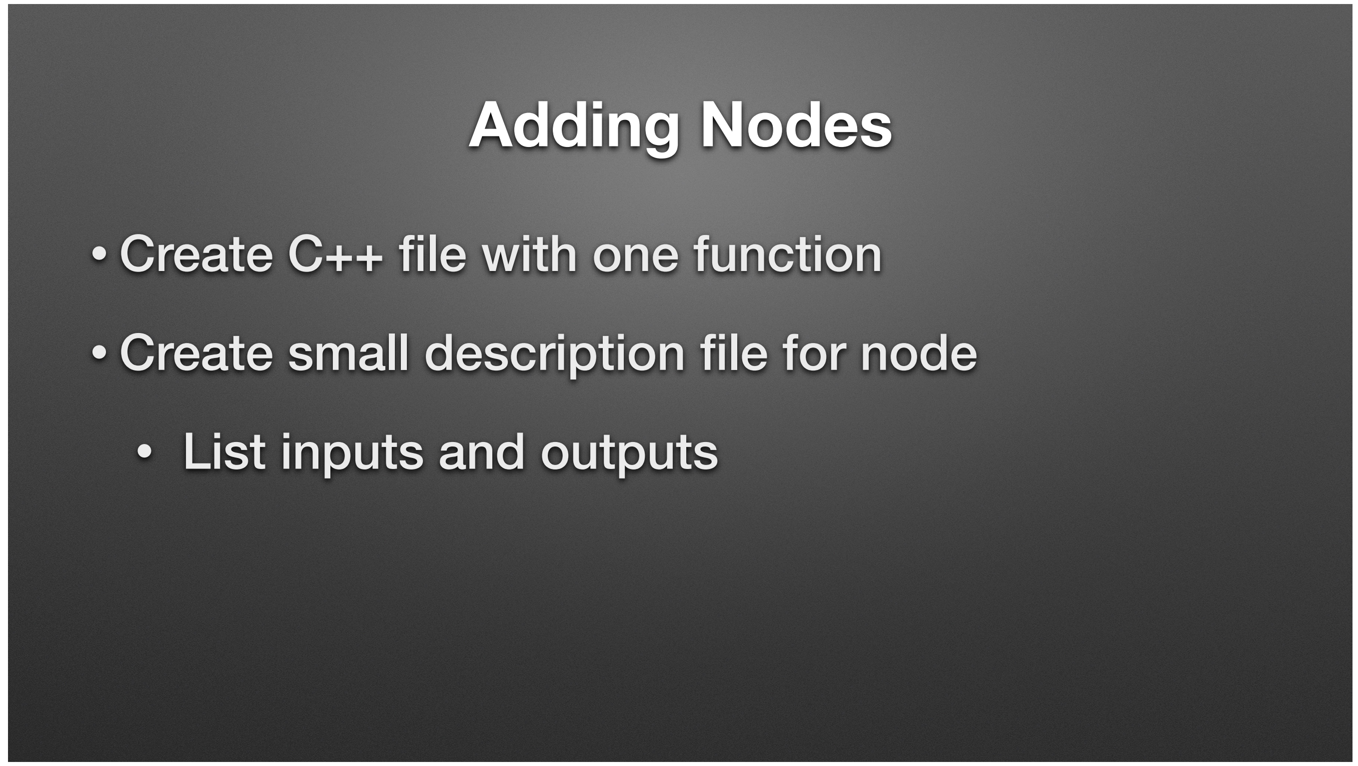
I guess by now you can see how this system allowed us to build powerful dynamic sounds, just limited by the set of nodes we have available.
But why is it truly extendable?
That's because it's really **easy to create a new node** in this system. Basically all you need to do is **create an asset** describing the **inputs and outputs of the node** (can be done in our editor) and a **C++ file** containing the implementation.
Our **code generator will collect** the little snippets of **C++ code** for **each node** and create a combined function for each graph. **No external libraries** are used, even for **math** we **call functions in the game engine**. The generated graph programs are **not linked with anything**, so very small and light weight.
This also means that the **game or** the **editor don't need to be recompiled** to add a simple node. **Nodes and graphs are game assets**, which get turned into **code** automatically by the **content pipeline**.



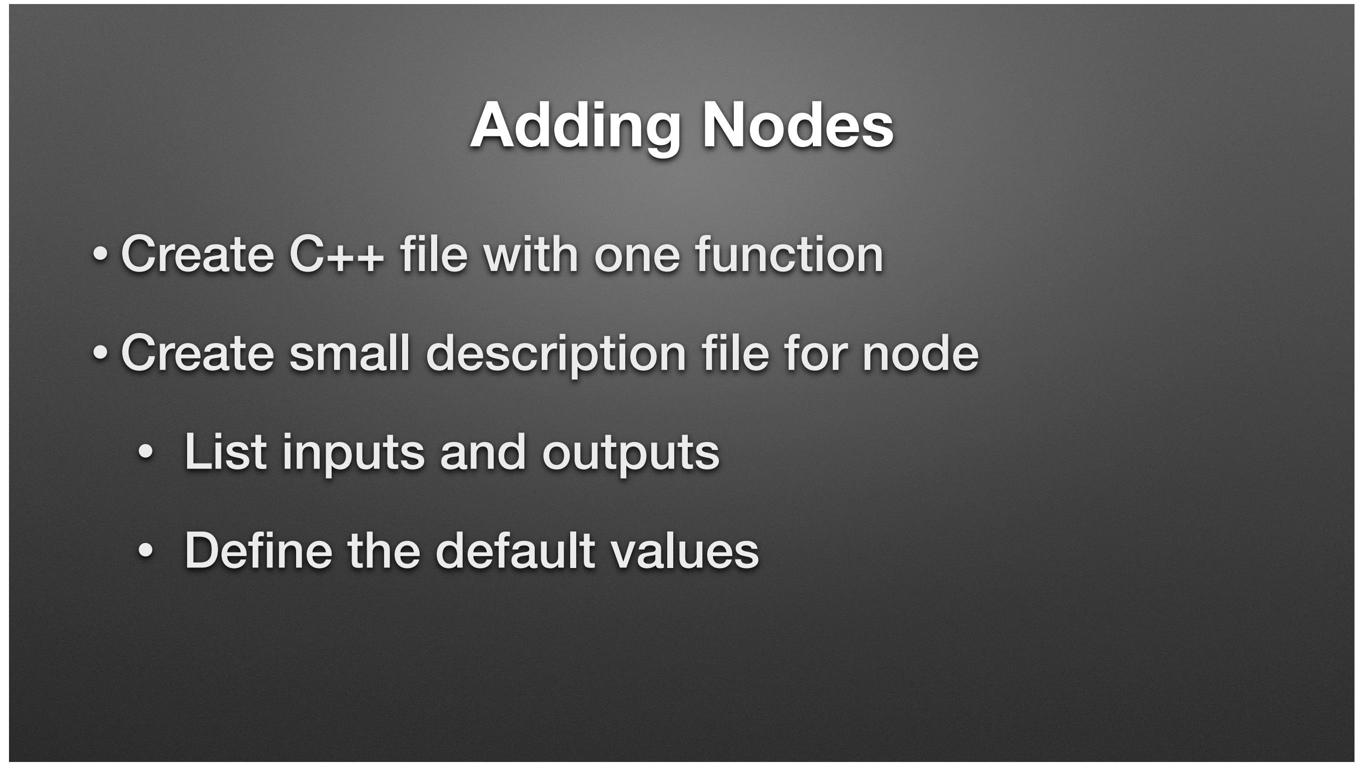
I guess by now you can see how this system allowed us to build powerful dynamic sounds, just limited by the set of nodes we have available.
But why is it truly extendable?
That's because it's really **easy to create a new node** in this system. Basically all you need to do is **create an asset** describing the **inputs and outputs of the node** (can be done in our editor) and a **C++ file** containing the implementation.
Our **code generator will collect** the little snippets of **C++ code** for **each node** and create a combined function for each graph. **No external libraries** are used, even for **math** we **call functions in the game engine**. The generated graph programs are **not linked with anything**, so very small and light weight.
This also means that the **game or** the **editor don't need to be recompiled** to add a simple node. **Nodes and graphs are game assets**, which get turned into **code** automatically by the **content pipeline**.



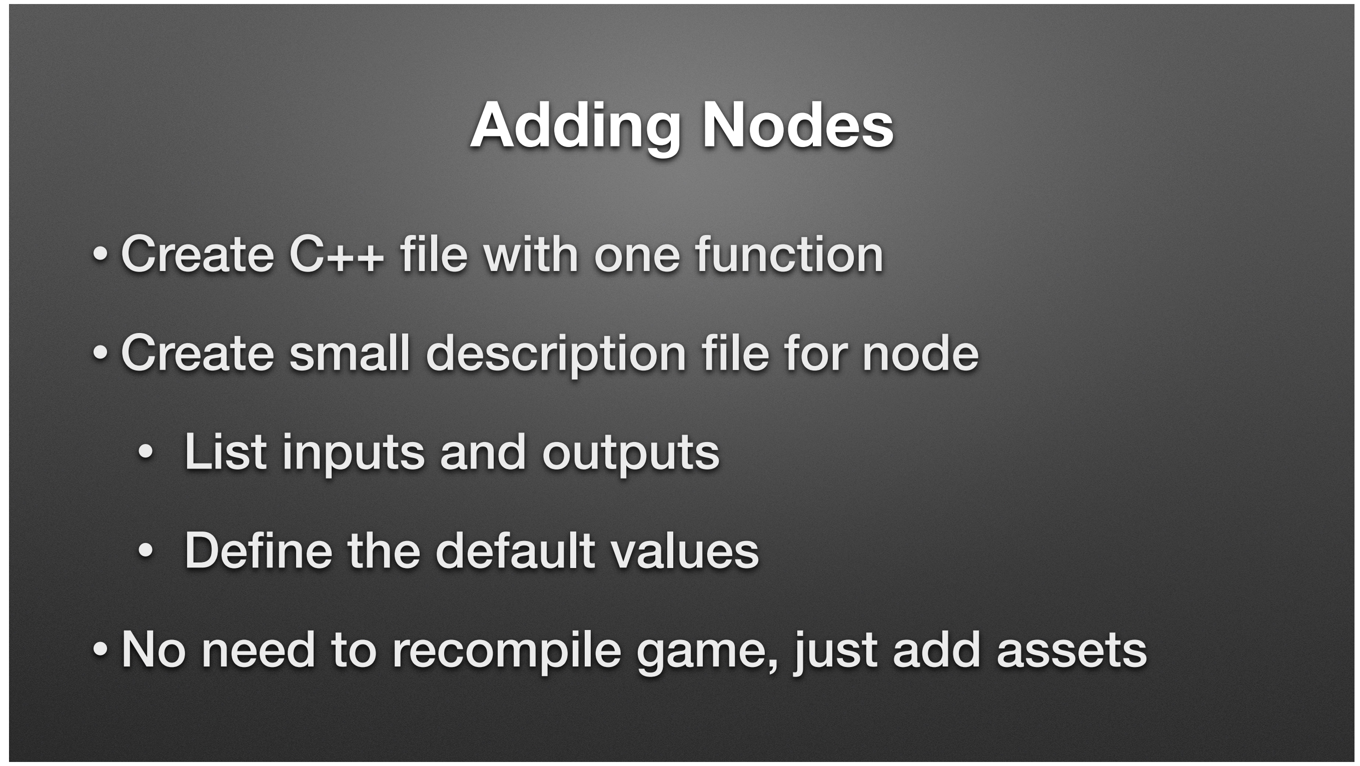
I guess by now you can see how this system allowed us to build powerful dynamic sounds, just limited by the set of nodes we have available.
But why is it truly extendable?
That's because it's really **easy to create a new node** in this system. Basically all you need to do is **create an asset** describing the **inputs and outputs of the node** (can be done in our editor) and a **C++ file** containing the implementation.
Our **code generator will collect** the little snippets of **C++ code** for **each node** and create a combined function for each graph. **No external libraries** are used, even for **math** we **call functions in the game engine**. The generated graph programs are **not linked with anything**, so very small and light weight.
This also means that the **game or** the **editor don't need to be recompiled** to add a simple node. **Nodes and graphs are game assets**, which get turned into **code** automatically by the **content pipeline**.



I guess by now you can see how this system allowed us to build powerful dynamic sounds, just limited by the set of nodes we have available.
But why is it truly extendable?
That's because it's really **easy to create a new node** in this system. Basically all you need to do is **create an asset** describing the **inputs and outputs of the node** (can be done in our editor) and a **C++ file** containing the implementation.
Our **code generator will collect** the little snippets of **C++ code** for **each node** and create a combined function for each graph. **No external libraries** are used, even for **math** we **call functions in the game engine**. The generated graph programs are **not linked with anything**, so very small and light weight.
This also means that the **game or** the **editor don't need to be recompiled** to add a simple node. **Nodes and graphs are game assets**, which get turned into **code** automatically by the **content pipeline**.



I guess by now you can see how this system allowed us to build powerful dynamic sounds, just limited by the set of nodes we have available.
But why is it truly extendable?
That's because it's really **easy to create a new node** in this system. Basically all you need to do is **create an asset** describing the **inputs and outputs of the node** (can be done in our editor) and a **C++ file** containing the implementation.
Our **code generator will collect** the little snippets of **C++ code** for **each node** and create a combined function for each graph. **No external libraries** are used, even for **math** we **call functions in the game engine**. The generated graph programs are **not linked with anything**, so very small and light weight.
This also means that the **game or** the **editor don't need to be recompiled** to add a simple node. **Nodes and graphs are game assets**, which get turned into **code** automatically by the **content pipeline**.



I guess by now you can see how this system allowed us to build powerful dynamic sounds, just limited by the set of nodes we have available.
But why is it truly extendable?
That's because it's really **easy to create a new node** in this system. Basically all you need to do is **create an asset** describing the **inputs and outputs of the node** (can be done in our editor) and a **C++ file** containing the implementation.
Our **code generator will collect** the little snippets of **C++ code** for **each node** and create a combined function for each graph. **No external libraries** are used, even for **math** we **call functions in the game engine**. The generated graph programs are **not linked with anything**, so very small and light weight.
This also means that the **game or** the **editor don't need to be recompiled** to add a simple node. **Nodes and graphs are game assets**, which get turned into **code** automatically by the **content pipeline**.

Andreas: Let's look at an **example node for a mathematical problem**, the obvious ones like add or multiply are trivial, but here's a useful one, **converting from musical semitones to a pitch ratio** value. So for instance inputting a value of **12 semitones**, would result in a **pitch ratio of 2**, i.e. an **octave**.

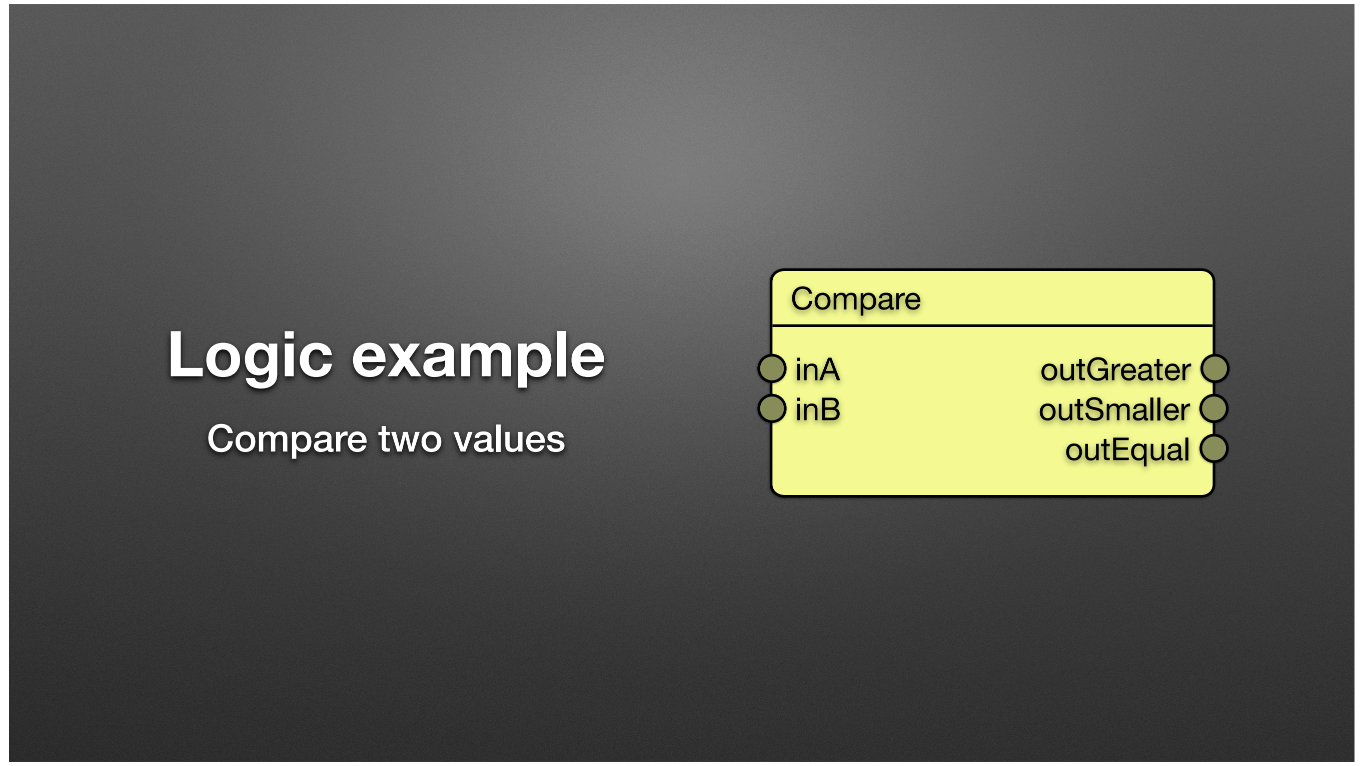
```
void SemitonesToPitchRatio(const float inSemitones,
                           float* outPitchRatio)
{
   *outPitchRatio = Math::Pow(2.0f, inSemitones/12.0f);
}
```

SemiTonesToPitchRatio.cpp

This shows how **easy it is to create a new node**. The only **additional information** that needs to be created is a **meta data file** that describes the **input** and **output** values and their **default values**.

Math example

Convert semitones to pitch ratio



Here's another **simple example** for **comparison of values**. Typically you have different parts that depend on the comparison result, so **it's useful to have** "smaller" and "greater than" **available at the same time**. Also saves us from using additional nodes.

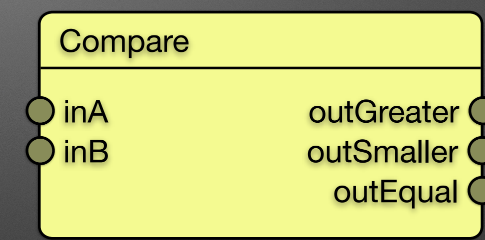
```cpp
void Compare(const float inA, const float inB,
             bool* outSmaller, bool* outGreater,
             bool* outEqual)
{
   *outSmaller      = inA < inB;
   *outGreater      = inA > inB;
   *outEqual        = inA == inB;
}
```

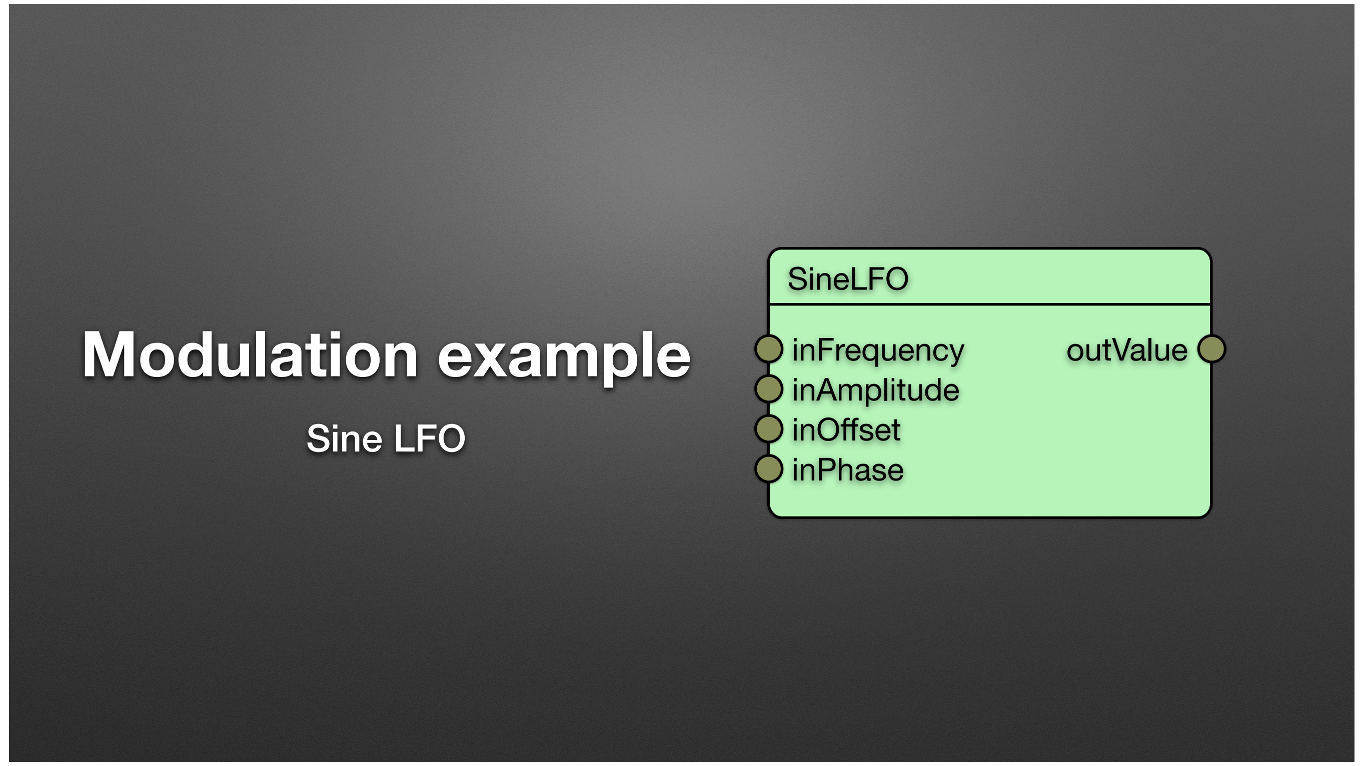
Compare.cpp

The implementation is just as simple. All the **comparison results are stored in output parameters**.

Logic example

Compare two values





This is the **sine LFO node**, it will output a sine wave at a given **frequency and amplitude**. This is a **slightly more complicated** node.

```
void SineLFO(const float inFrequency,
             const float inAmplitude,
             const float inOffset, const float inPhase,
             float* outValue, float* phase)
{
   *outValue = inOffset +
               inAmplitude * Math::Sin((*phase + inPhase) * M_TWO_PI);

   float new_phase = *phase + inFrequency * GraphSound::GetTimeStep();

   if (new_phase > 1.0f)
      new_phase -= 1.0f;

   *phase = new_phase;
}
```

SineLFO.cpp

Let's look at how this is implemented. It's using **two functions** that are implemented in our **engine**, one for **calculating the sine** and one for **getting the time step**. If the game time is scaled (for example if we're running the game in slow motion), then the sound time-step will also be adjusted. However it's also possible to have sounds that are not affected by this.

Normally the **graph execution is state less**, that means that **every time it runs**, it does **exactly the same**, because it **doesn't store any information**, so it **can't depend on previous runs**. This node however is an example of something that **requires state information**, and it's **possible** to do that **on a per node basis**. So here you can see that it uses a special **"phase" parameter,** which is **stored** separately **for each instance of the node**, and is passed in by reference as a pointer. The state system allows us to **store any arbitrary set of data**, for example a **full C++ object**, or just a **simple value** like in this example.

The **Wave node** also uses this to **store** the information about the **voices** it has created.

```cpp
void SineLFO(const float inFrequency,
             const float inAmplitude,
             const float inOffset, const float inPhase,
             float* outValue, float* phase)
{
   *outValue = inOffset +
               inAmplitude * Math::Sin((*phase + inPhase) * M_TWO_PI);

   float new_phase = *phase + inFrequency * GraphSound::GetTimeStep();

   if (new_phase > 1.0f)
      new_phase -= 1.0f;

   *phase = new_phase;
}
```

SineLFO.cpp

Let's look at how this is implemented. It's using **two functions** that are implemented in our **engine**, one for **calculating the sine** and one for **getting the time step**. If the game time is scaled (for example if we're running the game in slow motion), then the sound time-step will also be adjusted. However it's also possible to have sounds that are not affected by this.

Normally the **graph execution is state less**, that means that **every time it runs**, it does **exactly the same**, because it **doesn't store any information**, so it **can't depend on previous runs**. This node however is an example of something that **requires state information**, and it's **possible** to do that **on a per node basis**. So here you can see that it uses a special **"phase" parameter,** which is **stored** separately **for each instance of the node**, and is passed in by reference as a pointer. The state system allows us to **store any arbitrary set of data**, for example a **full C++ object**, or just a **simple value** like in this example.

The **Wave node** also uses this to **store** the information about the **voices** it has created.

```cpp
void SineLFO(const float inFrequency,
             const float inAmplitude,
             const float inOffset, const float inPhase,
             float* outValue, float* phase)
{
   *outValue = inOffset +
               inAmplitude * Math::Sin((*phase + inPhase) * M_TWO_PI);

   float new_phase = *phase + inFrequency * GraphSound::GetTimeStep();

   if (new_phase > 1.0f)
      new_phase -= 1.0f;

   *phase = new_phase;
}
```

SineLFO.cpp

Let's look at how this is implemented. It's using **two functions** that are implemented in our **engine**, one for **calculating the sine** and one for **getting the time step**. If the game time is scaled (for example if we're running the game in slow motion), then the sound time-step will also be adjusted. However it's also possible to have sounds that are not affected by this.

Normally the **graph execution is state less**, that means that **every time it runs**, it does **exactly the same**, because it **doesn't store any information**, so it **can't depend on previous runs**. This node however is an example of something that **requires state information**, and it's **possible** to do that **on a per node basis**. So here you can see that it uses a special **"phase" parameter,** which is **stored** separately **for each instance of the node**, and is passed in by reference as a pointer. The state system allows us to **store any arbitrary set of data**, for example a **full C++ object**, or just a **simple value** like in this example.

The **Wave node** also uses this to **store** the information about the **voices** it has created.

```
void SineLFO(const float inFrequency,
             const float inAmplitude,
             const float inOffset, const float inPhase,
             float* outValue, float* phase)
{
   *outValue = inOffset +
               inAmplitude * Math::Sin((*phase + inPhase) * M_TWO_PI);

   float new_phase = *phase + inFrequency * GraphSound::GetTimeStep();

   if (new_phase > 1.0f)
      new_phase -= 1.0f;

   *phase = new_phase;
}
```

SineLFO.cpp

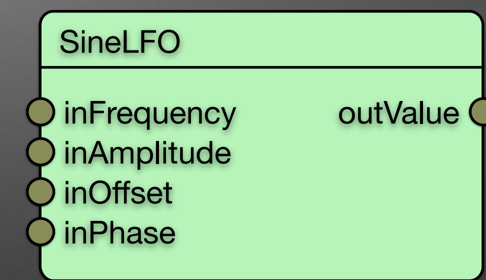
Let's look at how this is implemented. It's using **two functions** that are implemented in our **engine**, one for **calculating the sine** and one for **getting the time step**. If the game time is scaled (for example if we're running the game in slow motion), then the sound time-step will also be adjusted. However it's also possible to have sounds that are not affected by this.

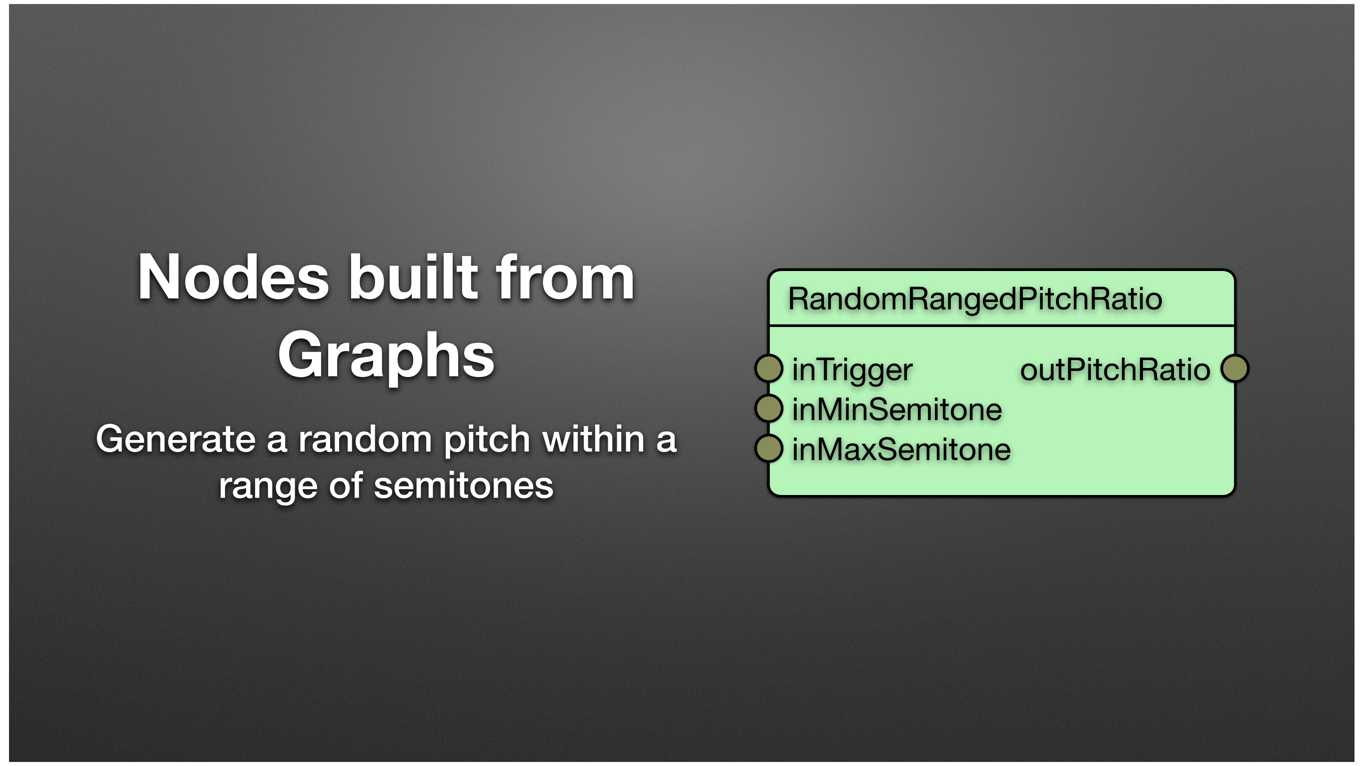
Normally the **graph execution is state less**, that means that **every time it runs**, it does **exactly the same**, because it **doesn't store any information**, so it **can't depend on previous runs**. This node however is an example of something that **requires state information**, and it's **possible** to do that **on a per node basis**. So here you can see that it uses a special **"phase" parameter,** which is **stored** separately **for each instance of the node**, and is passed in by reference as a pointer. The state system allows us to **store any arbitrary set of data**, for example a **full C++ object**, or just a **simple value** like in this example.

The **Wave node** also uses this to **store** the information about the **voices** it has created.

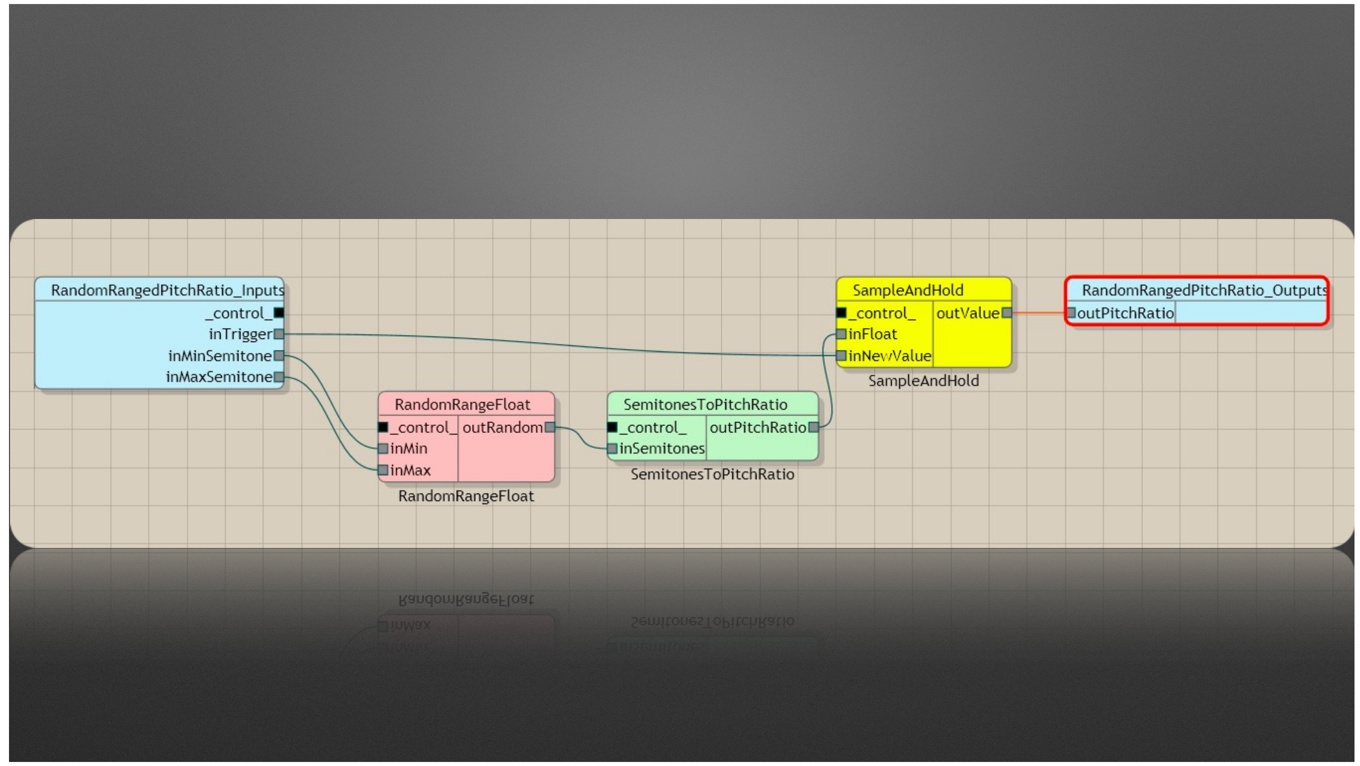
Modulation example

Sine LFO

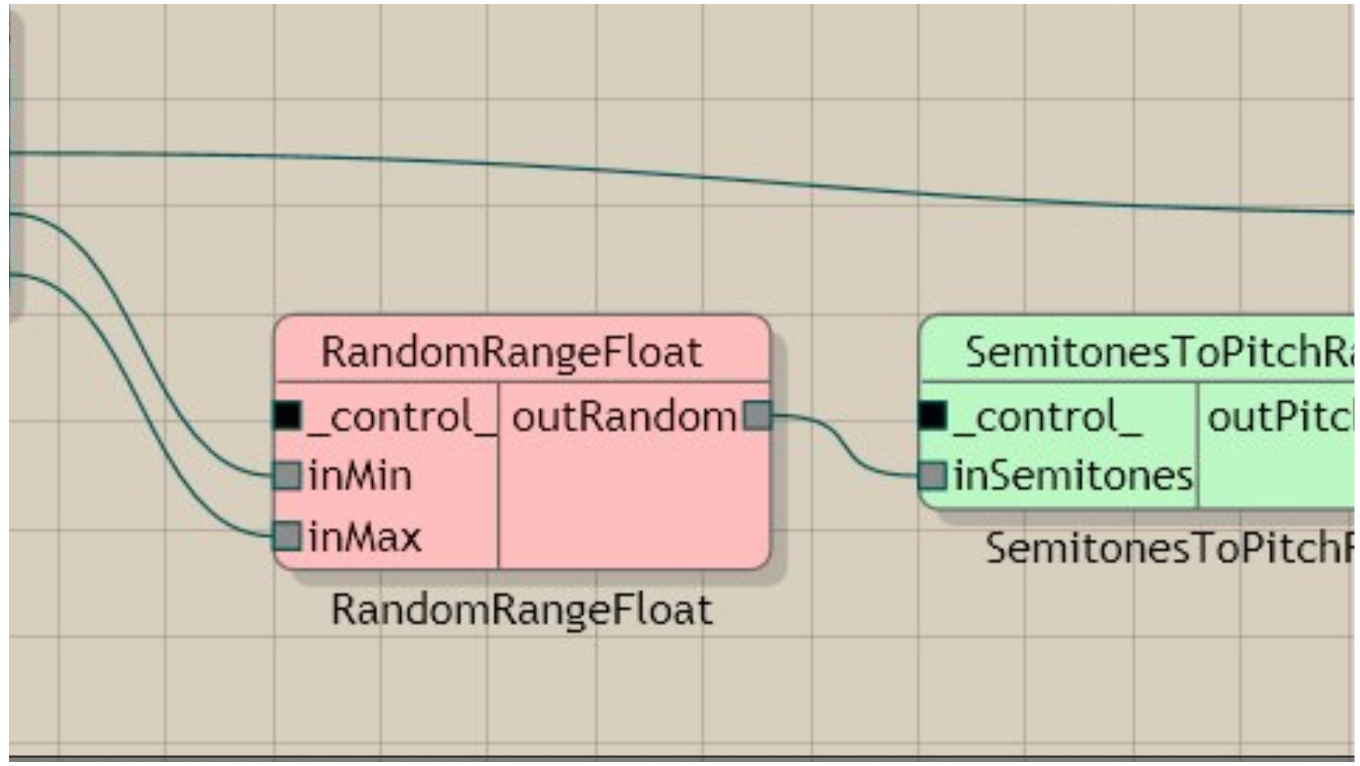




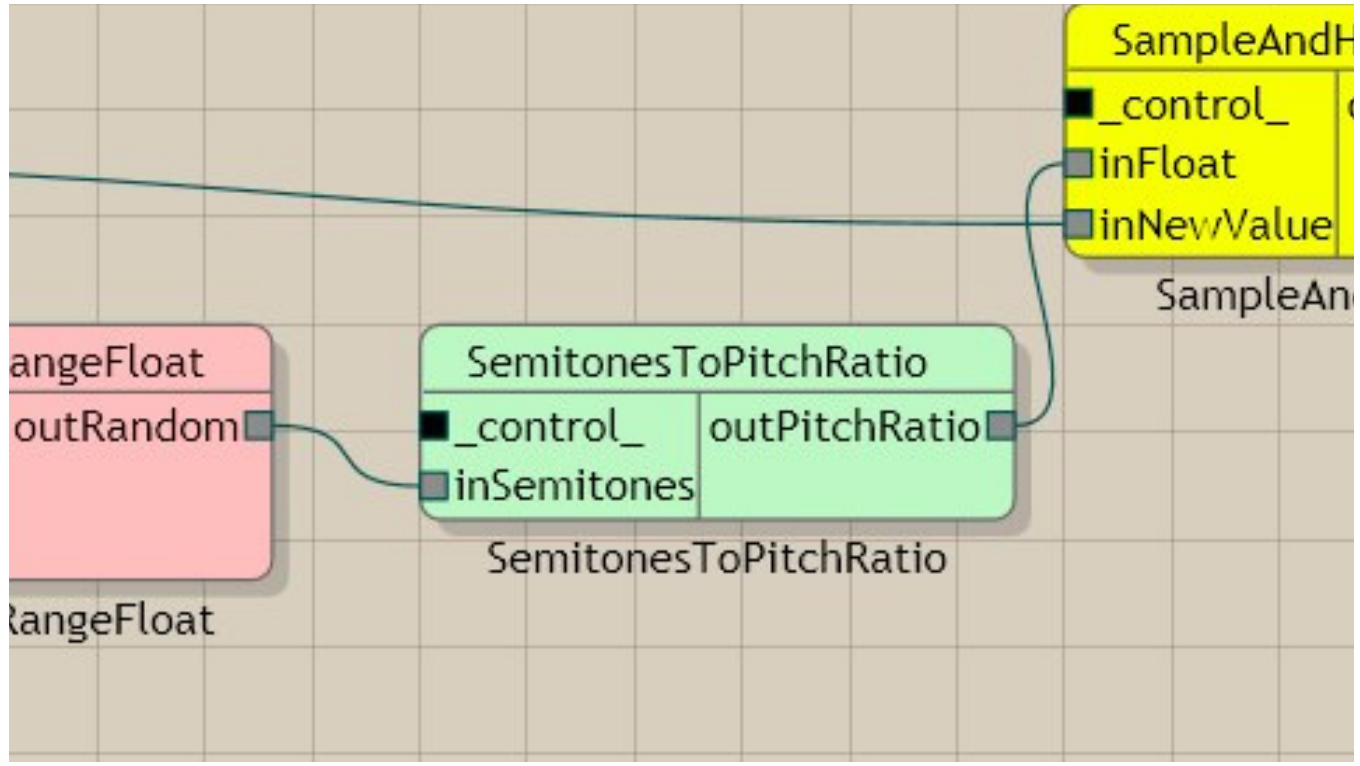
Certain **math or logic** processing that starts to happen often in sounds **can be abstracted into their own nodes**, **without** having to know any **programming**. Here's an example of a node, built from an actual graph.



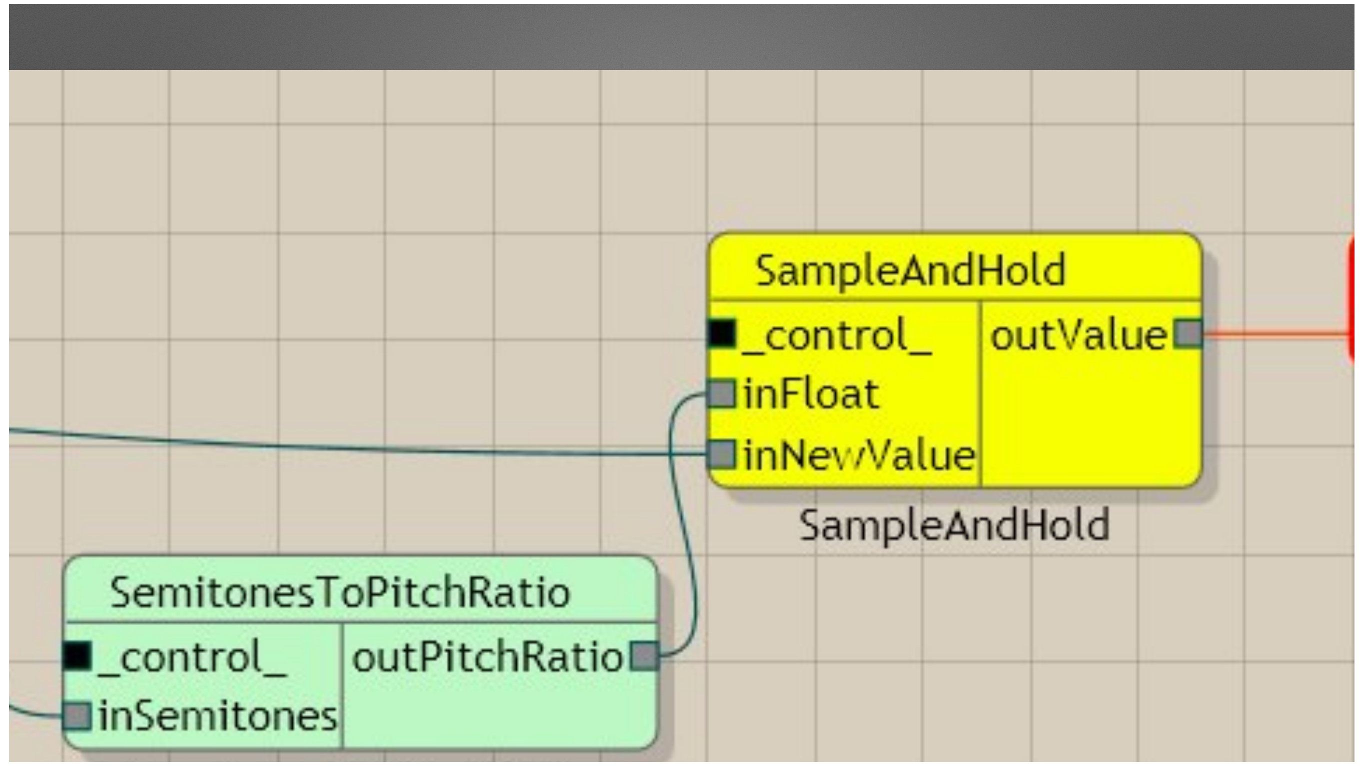
As you can see here, it uses a combination of the **random range node**, to get a **random semitone value**, which it then **converts to a pitch ratio**.
It also uses the **sample&hold node** to make sure the **value doesn't change**, as **normally** the random value would be **recalculated at every update** of the sound, i.e. once every 5.333ms.
Creating a node like this is basically as simple as **selecting a group of nodes**, then right-clicking and **selecting "Collapse nodes"**. The editor will do the rest.



As you can see here, it uses a combination of the **random range node**, to get a **random semitone value**, which it then **converts to a pitch ratio**.
It also uses the **sample&hold node** to make sure the **value doesn't change**, as **normally** the random value would be **recalculated at every update** of the sound, i.e. once every 5.333ms.
Creating a node like this is basically as simple as **selecting a group of nodes**, then right-clicking and **selecting "Collapse nodes"**. The editor will do the rest.



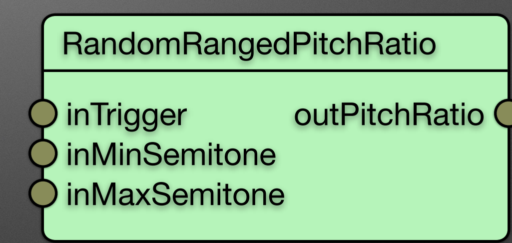
As you can see here, it uses a combination of the **random range node**, to get a **random semitone value**, which it then **converts to a pitch ratio**.
It also uses the **sample&hold node** to make sure the **value doesn't change**, as **normally** the random value would be **recalculated at every update** of the sound, i.e. once every 5.333ms.
Creating a node like this is basically as simple as **selecting a group of nodes**, then right-clicking and **selecting "Collapse nodes"**. The editor will do the rest.



As you can see here, it uses a combination of the **random range node**, to get a **random semitone value**, which it then **converts to a pitch ratio**.
It also uses the **sample&hold node** to make sure the **value doesn't change**, as **normally** the random value would be **recalculated at every update** of the sound, i.e. once every 5.333ms.
Creating a node like this is basically as simple as **selecting a group of nodes**, then right-clicking and **selecting "Collapse nodes"**. The editor will do the rest.

Nodes built from Graphs

Generate a random pitch within a range of semitones



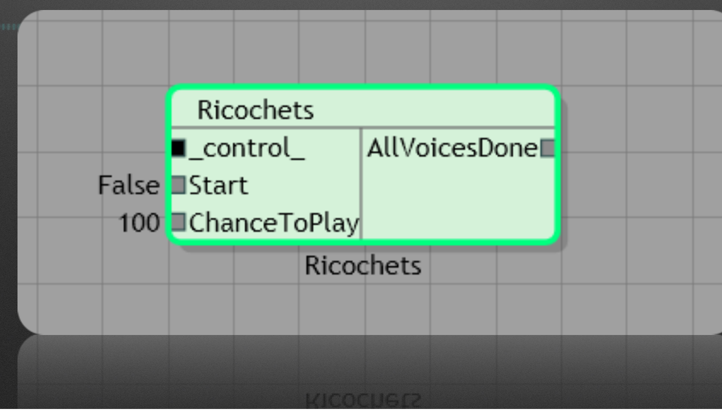
When you notice repeated work…

When you notice repeated work…

…abstract it!

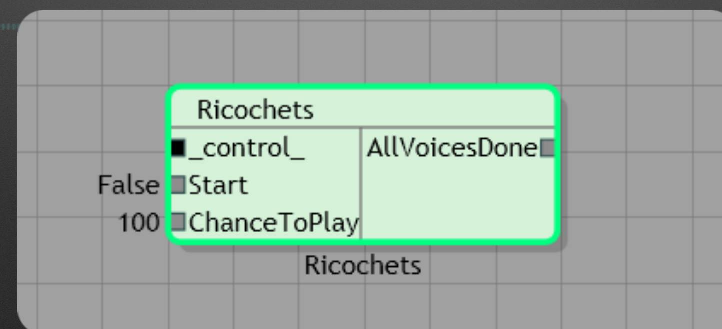
When you notice repeated work…

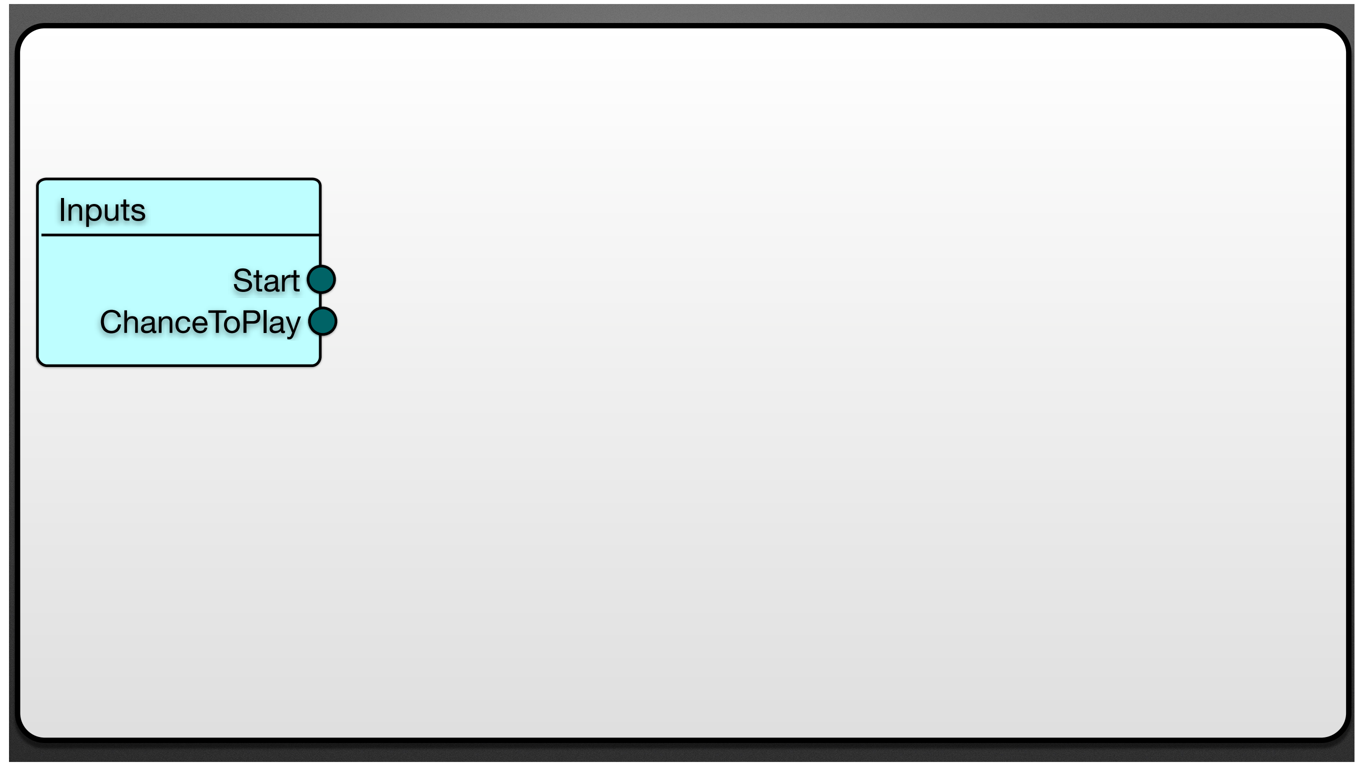
…abstract it!

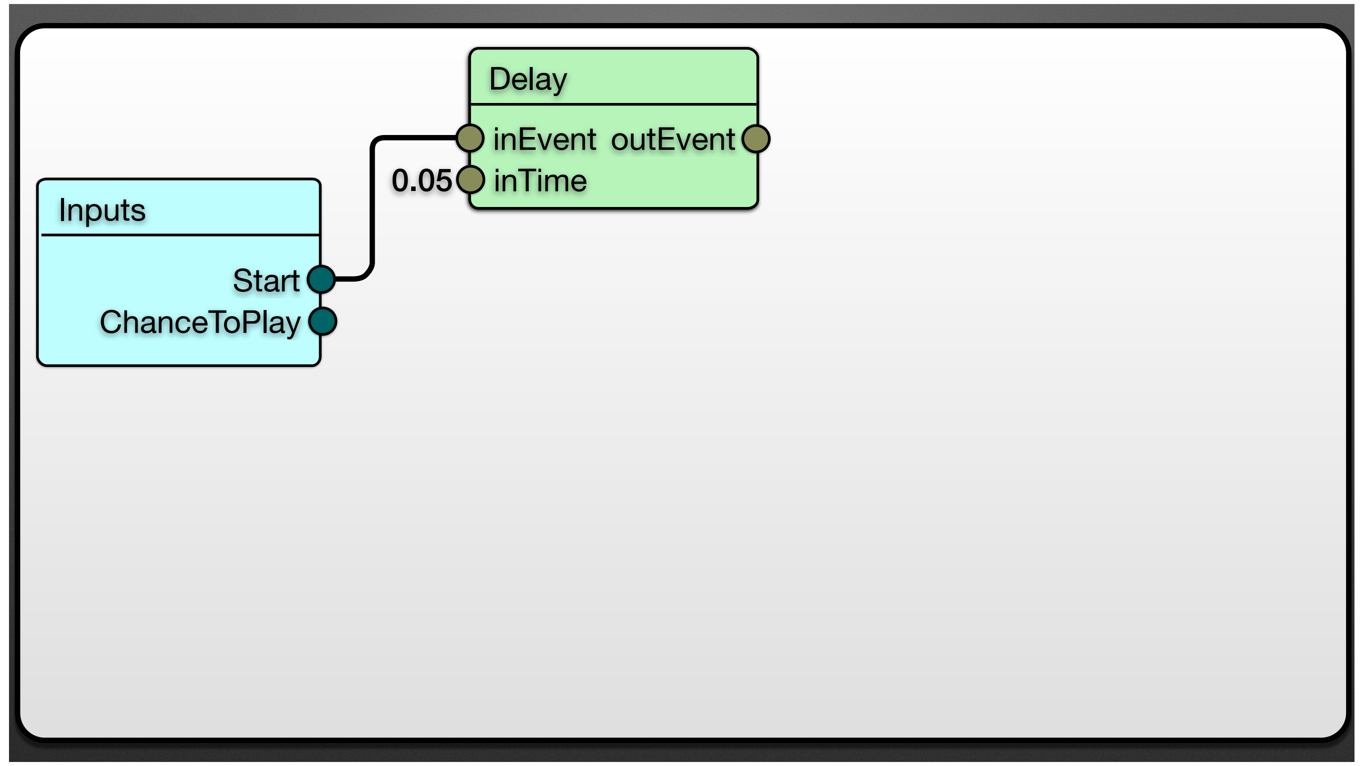


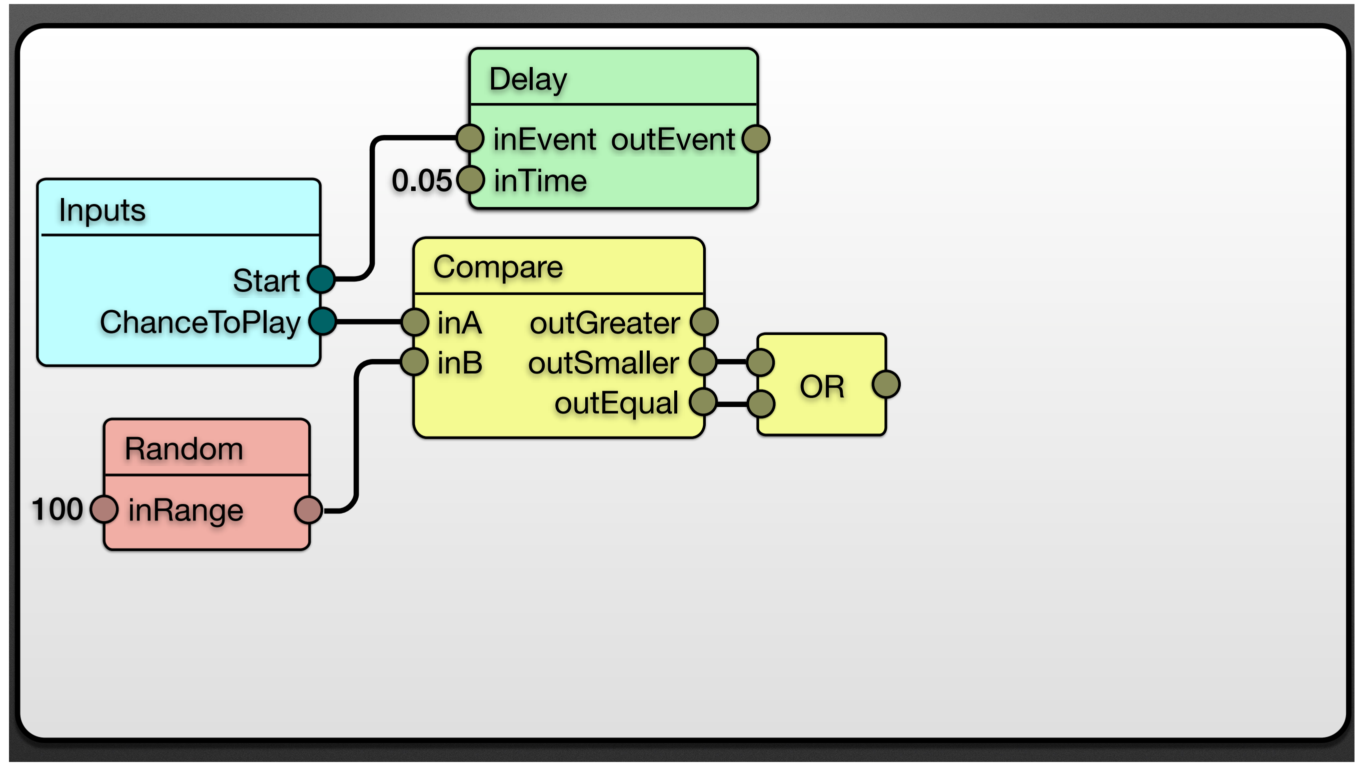
When you notice repeated work…

…abstract it!

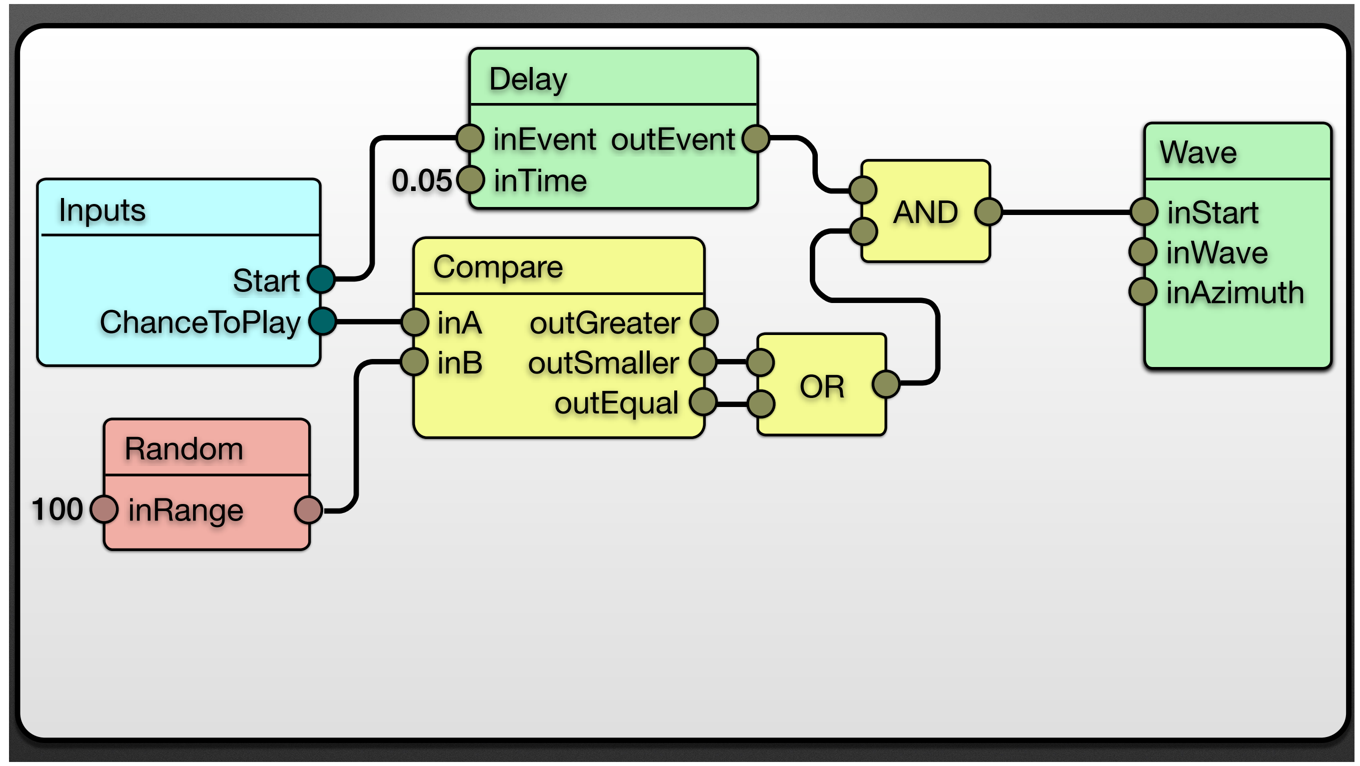



Inputs
Start
ChanceToPlay

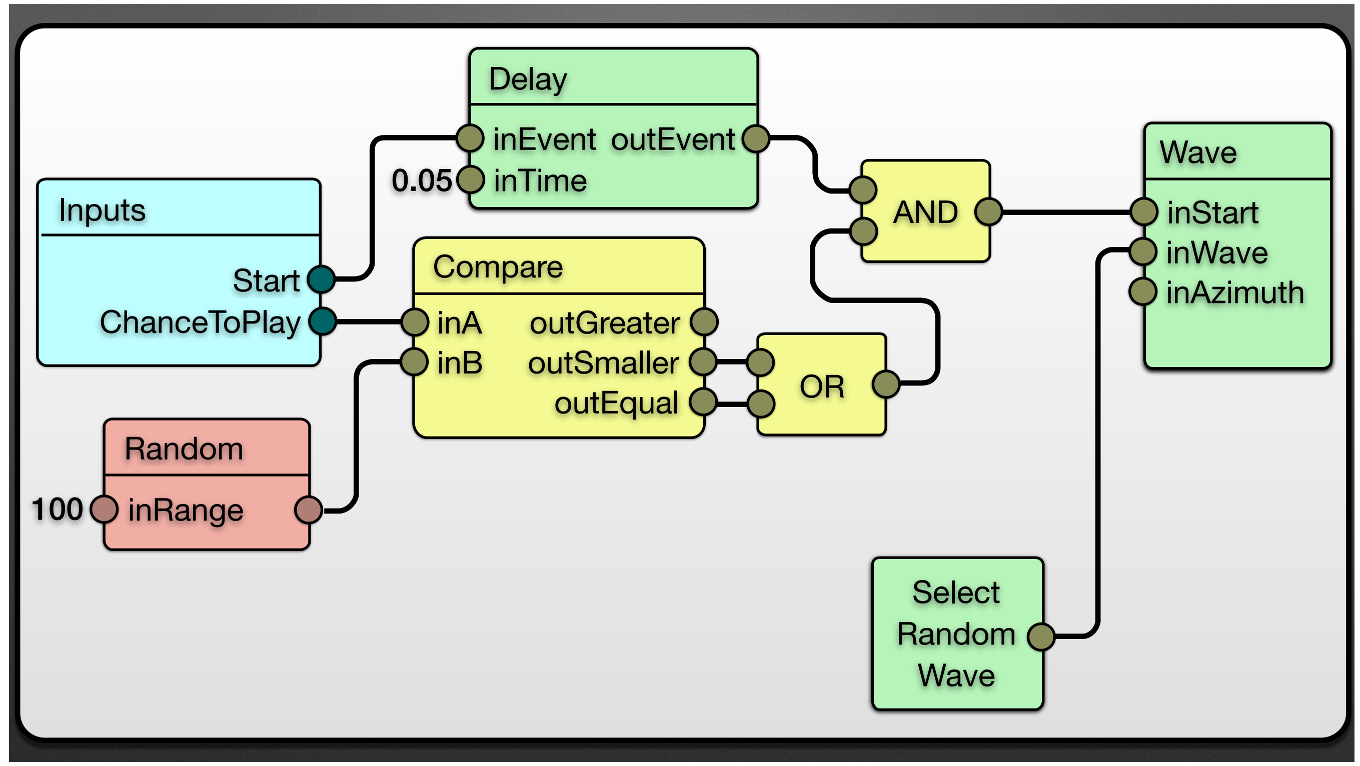

Delay
inEvent
outEvent
0.05
inTime
Inputs
Start
ChanceToPlay



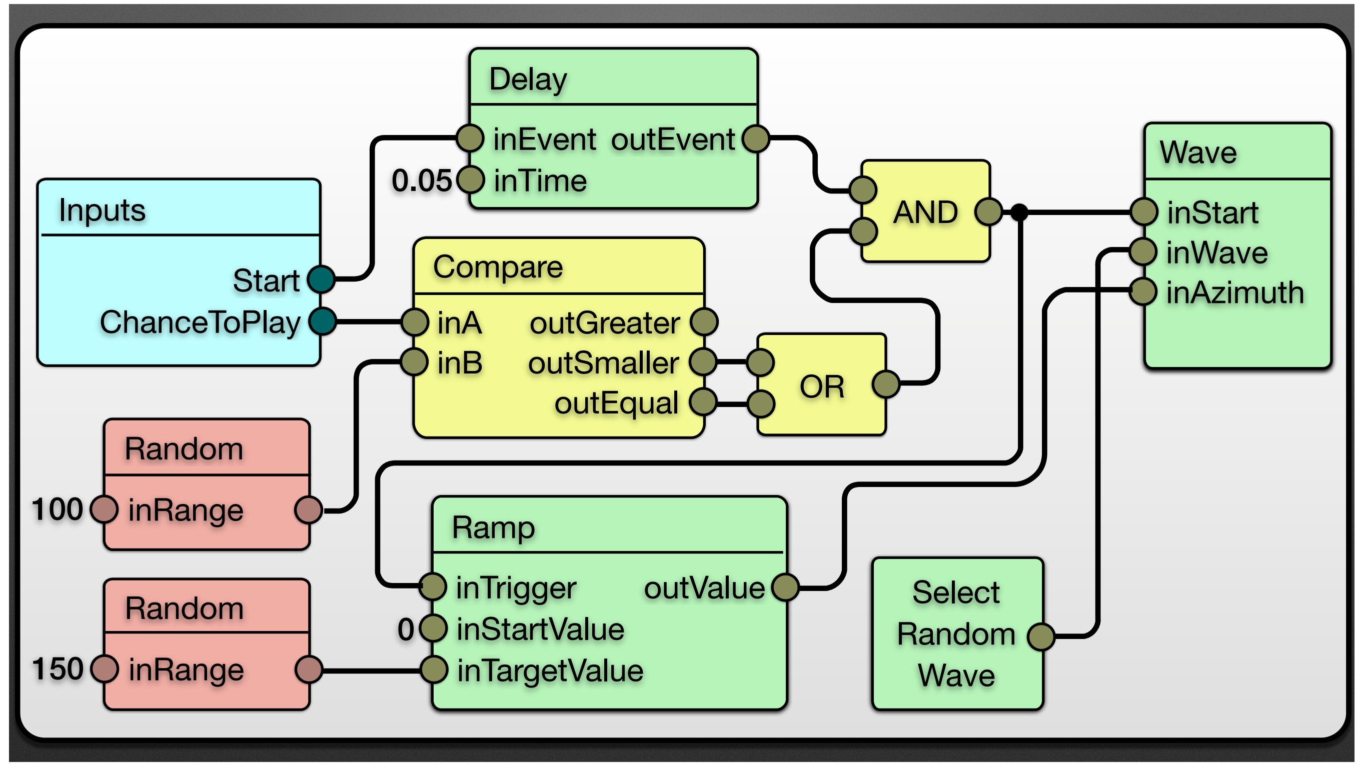
Delay
inEvent
outEvent
0.05
inTime
Inputs
Start
ChanceToPlay
Compare
inA
outGreater
inB
outSmaller
outEqual
OR
Random
100
inRange



Delay
inEvent
outEvent
0.05
inTime
Inputs
Start
ChanceToPlay
Compare
inA
outGreater
inB
outSmaller
outEqual
AND
OR
Wave
inStart
inWave
inAzimuth
Random
100
inRange



Delay
inEvent
outEvent
0.05
inTime
Inputs
Start
ChanceToPlay
Compare
inA
inB
outGreater
outSmaller
outEqual
AND
OR
Wave
inStart
inWave
inAzimuth
Random
100
inRange
Select
Random
Wave


Delay
inEvent
outEvent
0.05
inTime
Inputs
Start
ChanceToPlay
Compare
inA
inB
outGreater
outSmaller
outEqual
AND
OR
Wave
inStart
inWave
inAzimuth
Random
100
inRange
Random
150
inRange
Ramp
inTrigger
outValue
0
inStartValue
inTargetValue
Select
Random
Wave

Experimenting Should Be Safe and Fun

Experimenting Should Be Safe and Fun

- If your idea cannot fail, you’re not going to go to uncharted territory.

Experimenting Should Be Safe and Fun

- If your idea cannot fail, you’re not going to go to uncharted territory.
- We tried a lot

Experimenting Should Be Safe and Fun

- If your idea cannot fail, you're not going to go to uncharted territory.
- We tried a lot
- Some of it worked, some didn't

Experimenting Should Be Safe and Fun

- If your idea cannot fail, you're not going to go to uncharted territory.
- We tried a lot
- Some of it worked, some didn't
- Little or no code support - all in audio design

Examples

Fire Rate Variation

Fire Rate Variation

- Experiment: How to make battle sound more interesting

Fire Rate Variation

- Experiment: How to make battle sound more interesting

Fire Rate Variation

- Experiment: How to make battle sound more interesting

Fire Rate Variation

- Experiment: How to make battle sound more interesting
- Idea: Slight variation in fire rate

Fire Rate Variation

- Experiment: How to make battle sound more interesting
- Idea: Slight variation in fire rate

What the Mind Hears, …

What the Mind Hears, …

What the Mind Hears, …

- Experiment: Birds should react when you fire gun

What the Mind Hears, …

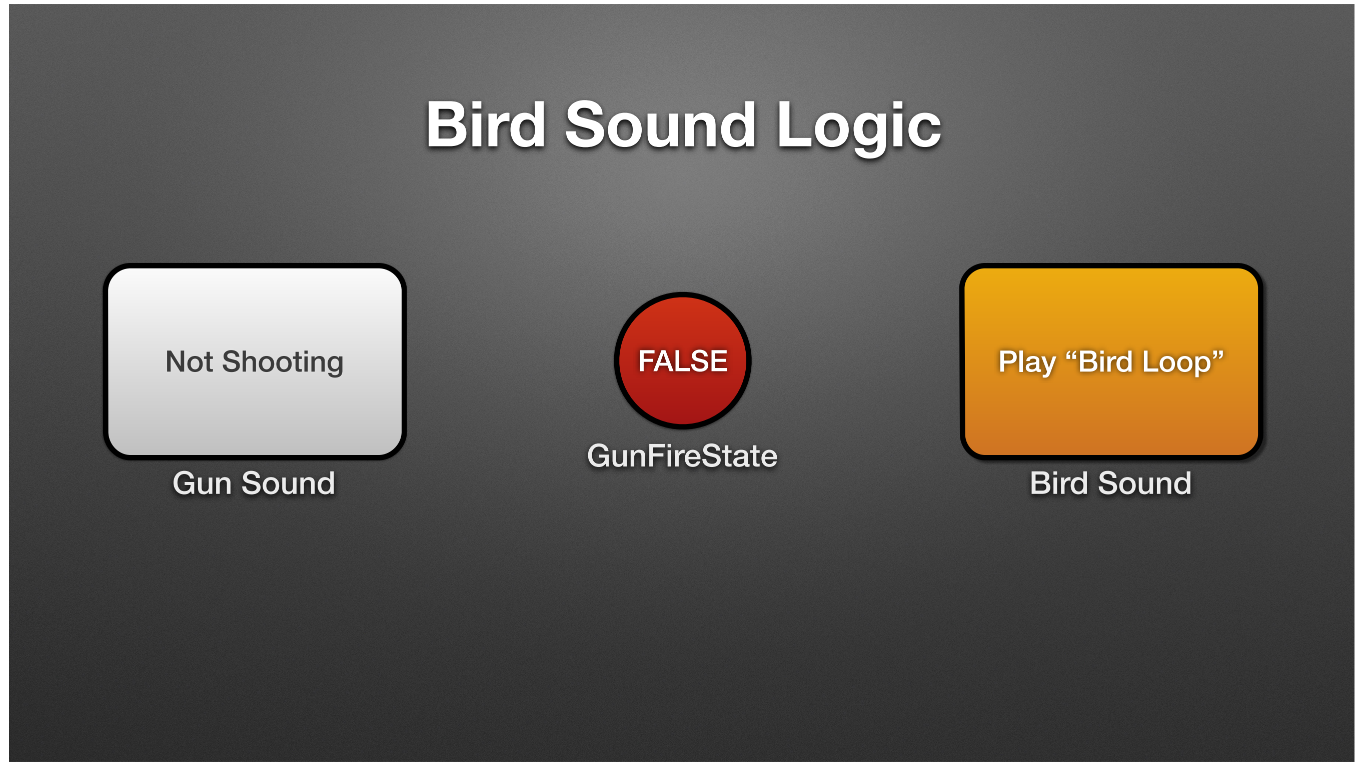
- Experiment: Birds should react when you fire gun
- Birds exist as sound only

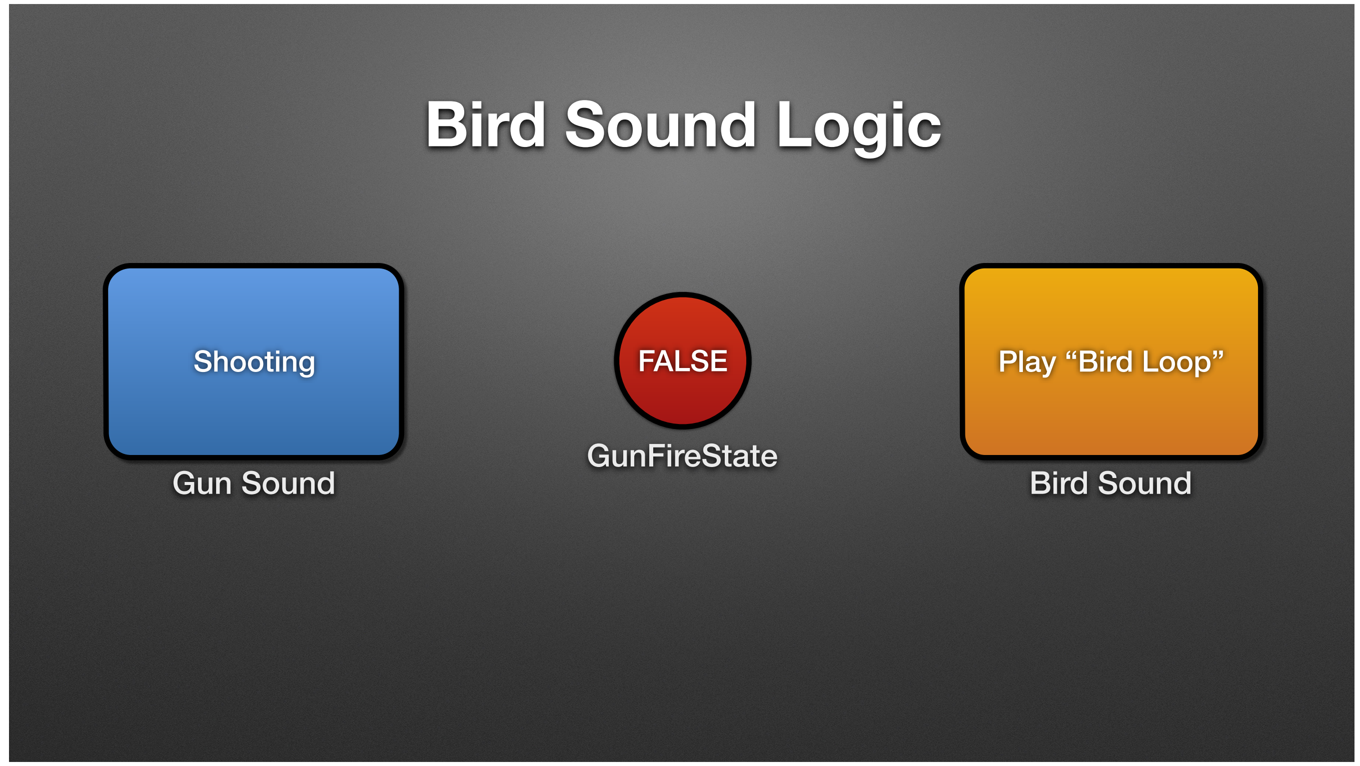
What the Mind Hears, …

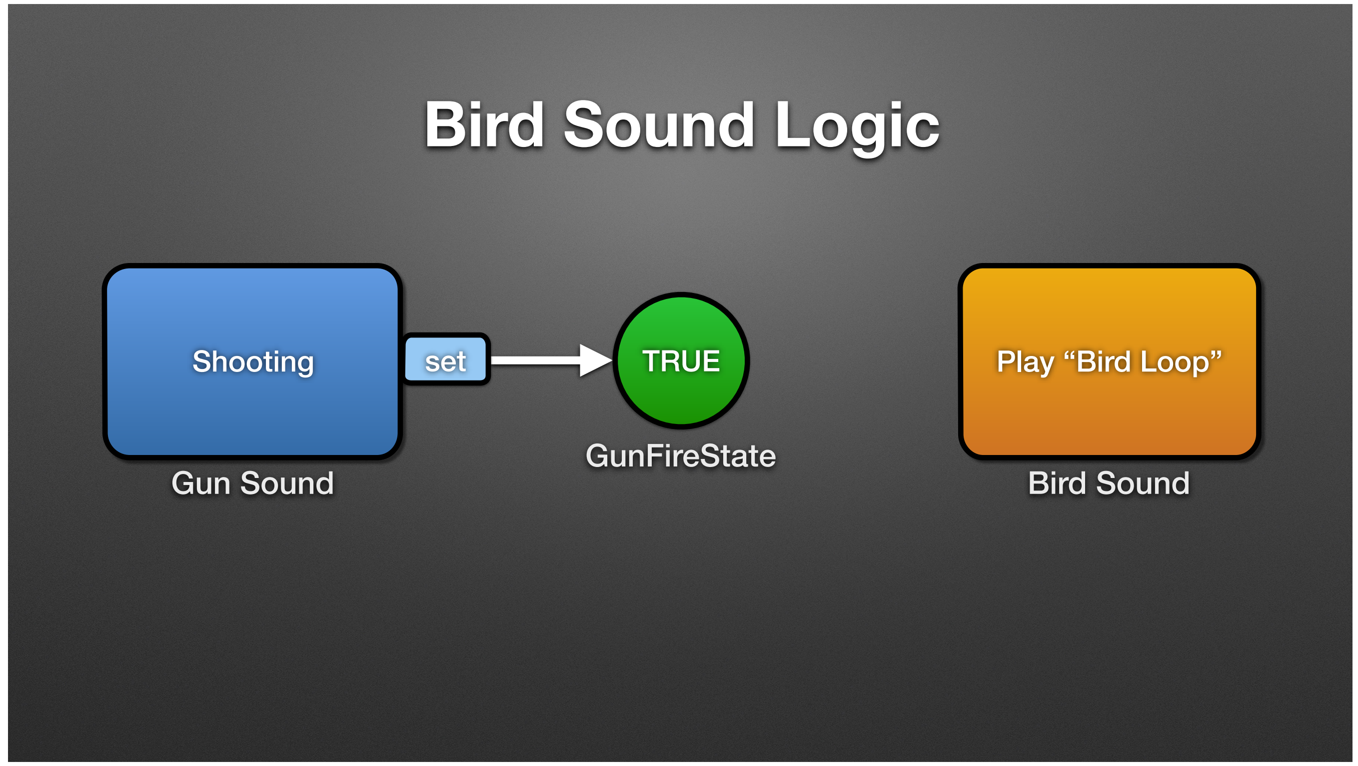
- Experiment: Birds should react when you fire gun
- Birds exist as sound only
- Use sound-to-sound messaging system

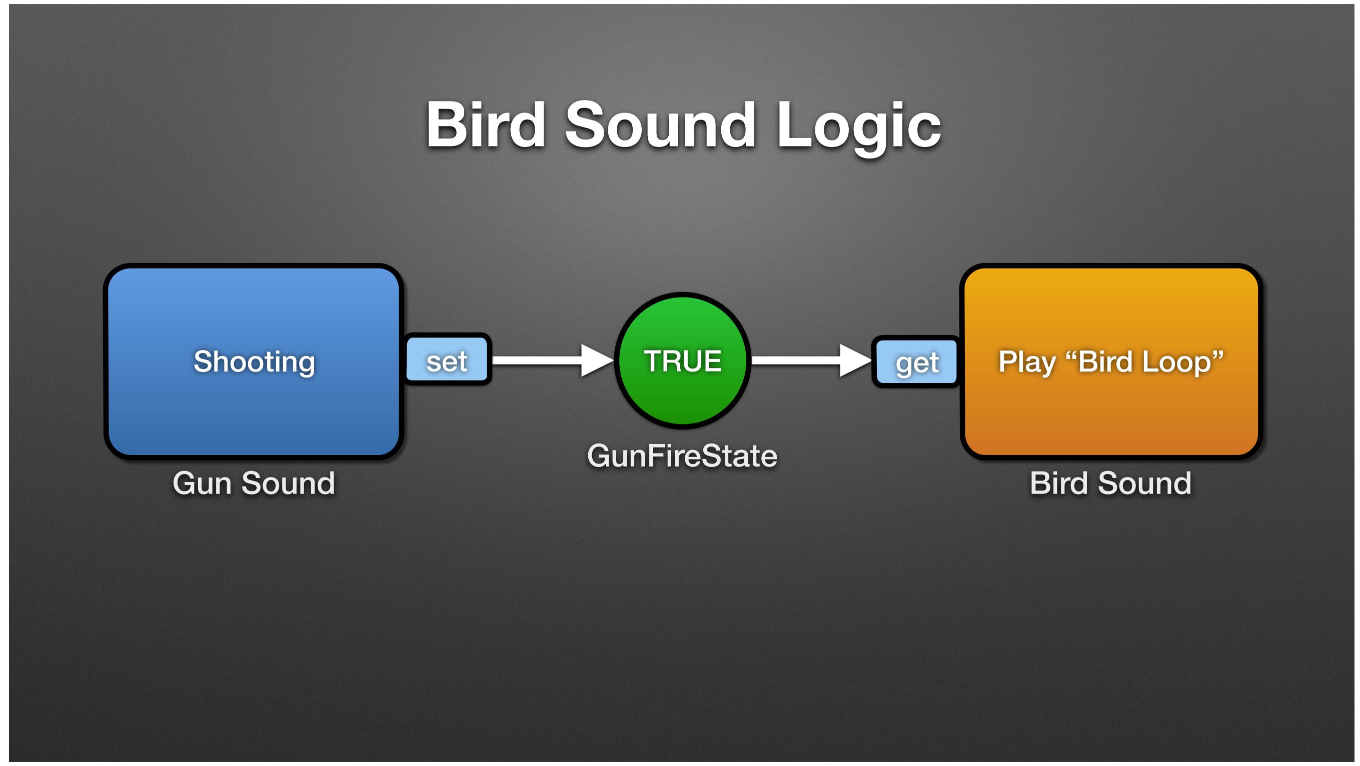
What the Mind Hears, …

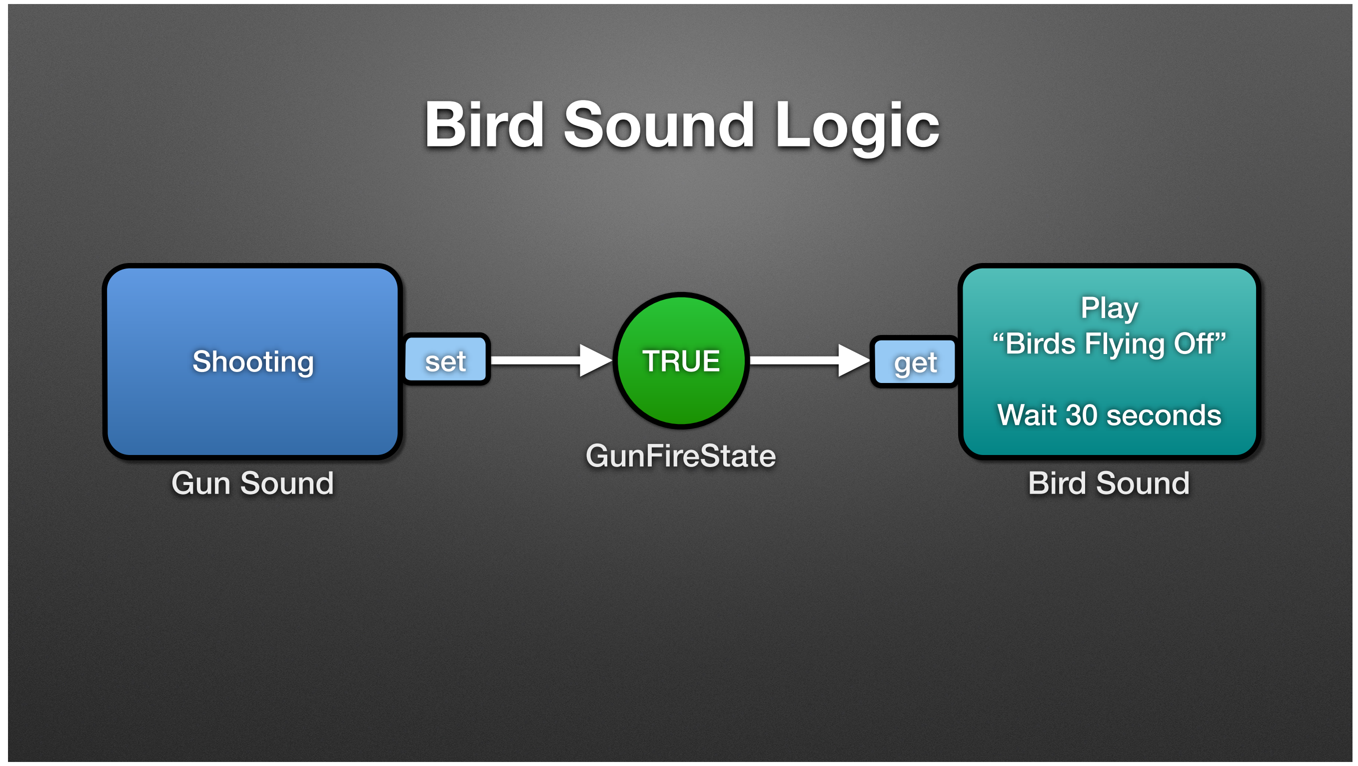
- Experiment: Birds should react when you fire gun
- Birds exist as sound only
- Use sound-to-sound messaging system
- Gun sounds notify bird sounds of shot

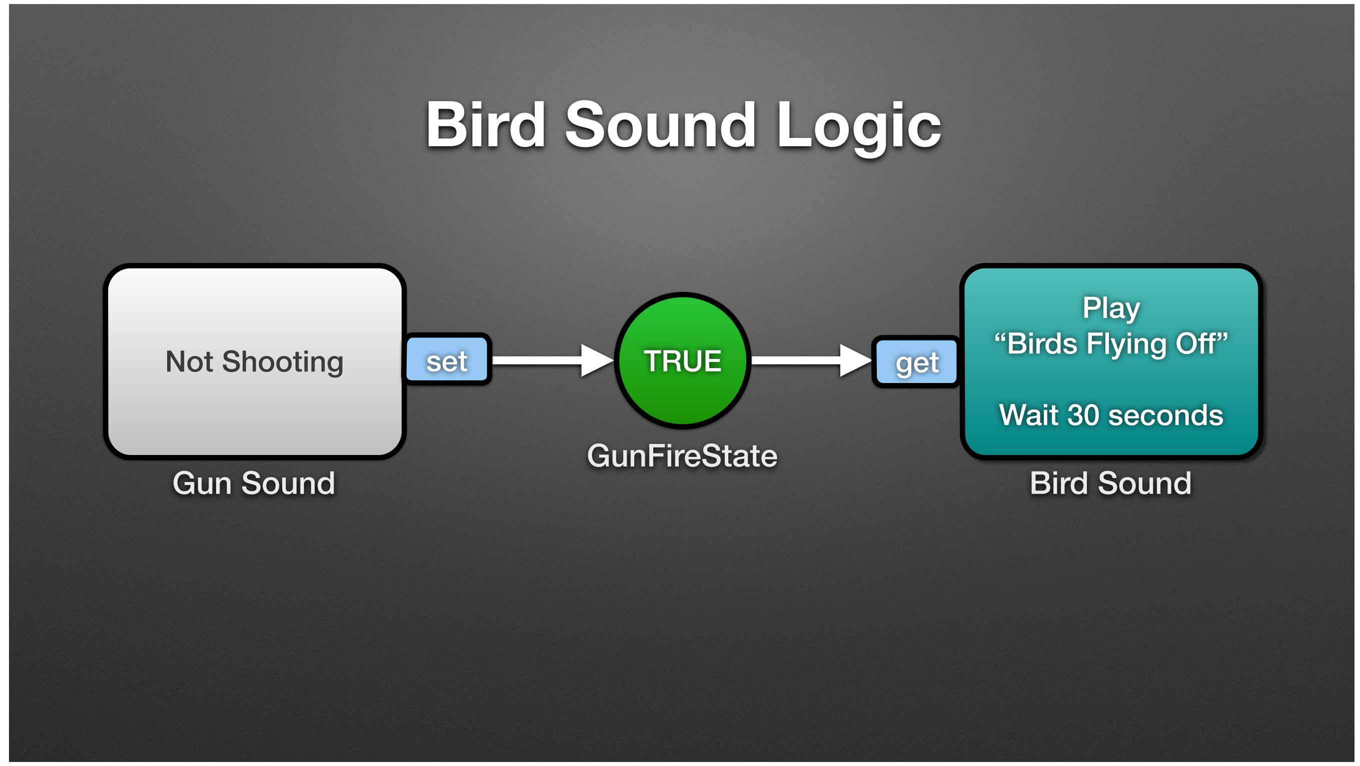

Bird Sound Logic
Not Shooting
Gun Sound
FALSE
GunFireState
Play “Bird Loop”
Bird Sound

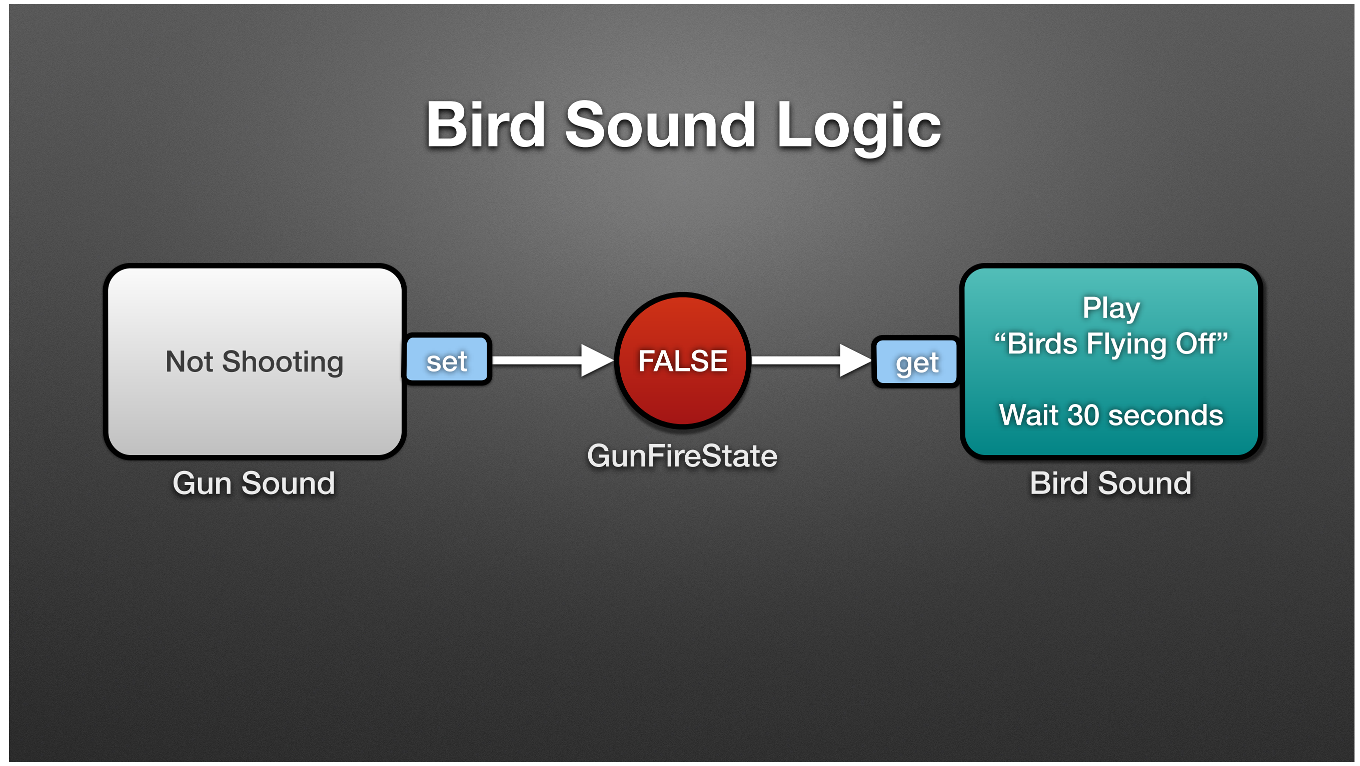

Bird Sound Logic
Shooting
Gun Sound
FALSE
GunFireState
Play “Bird Loop”
Bird Sound

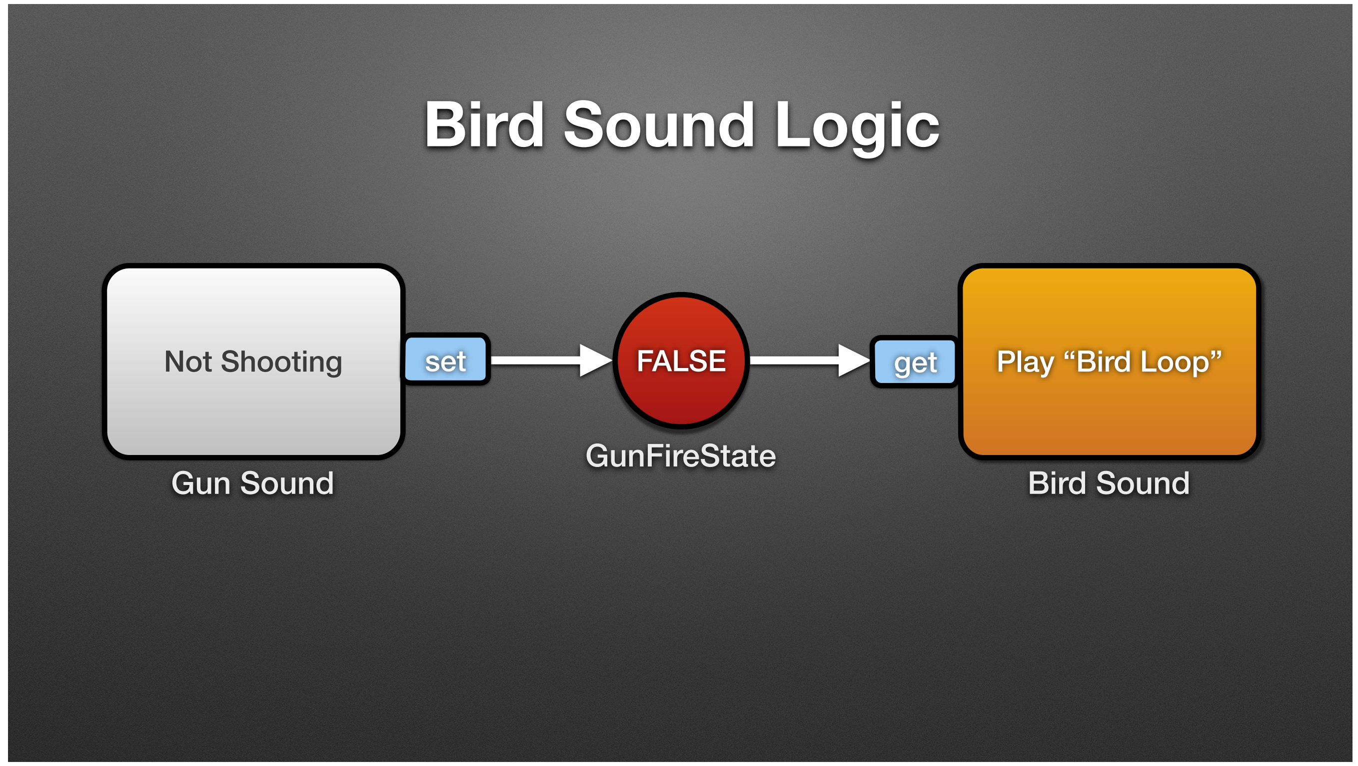

Bird Sound Logic
Shooting
set
TRUE
GunFireState
Play “Bird Loop”
Gun Sound
Bird Sound


Bird Sound Logic
Shooting
set
TRUE
get
Play “Bird Loop”
Gun Sound
GunFireState
Bird Sound

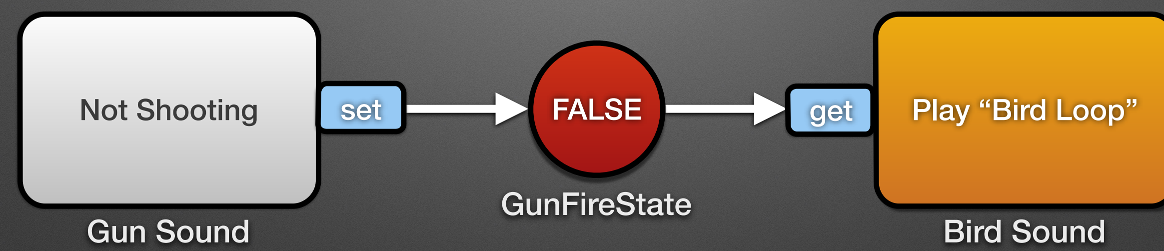

Bird Sound Logic
Shooting
set
TRUE
get
Play
“Birds Flying Off”
Wait 30 seconds
Gun Sound
GunFireState
Bird Sound


Bird Sound Logic
Not Shooting
set
TRUE
get
Play
“Birds Flying Off”
Wait 30 seconds
Gun Sound
GunFireState
Bird Sound


Bird Sound Logic
Not Shooting
set
FALSE
get
Play
“Birds Flying Off”
Wait 30 seconds
Gun Sound
GunFireState
Bird Sound


Bird Sound Logic
Not Shooting
set
FALSE
get
Play “Bird Loop”
Gun Sound
GunFireState
Bird Sound

Bird Sound Logic



Any sound can send a message which can be received by any other sound

Material Dependent Environmental Reactions (MADDER)

Material Dependent Environmental Reactions (MADDER)

- Inspiration:
  Guns have this explosive force in the world when they're fired

Material Dependent Environmental Reactions (MADDER)

- Inspiration:
  Guns have this explosive force in the world when they're fired
- Things that are near start to rattle

See http://www.youtube.com/watch?v=nMkiDguYtGw

See http://www.youtube.com/watch?v=nMkiDguYtGw

See http://www.youtube.com/watch?v=nMkiDguYtGw

MADDER

MADDER

- Inputs required:

MADDER

- Inputs required:
  - Distance to closest surface

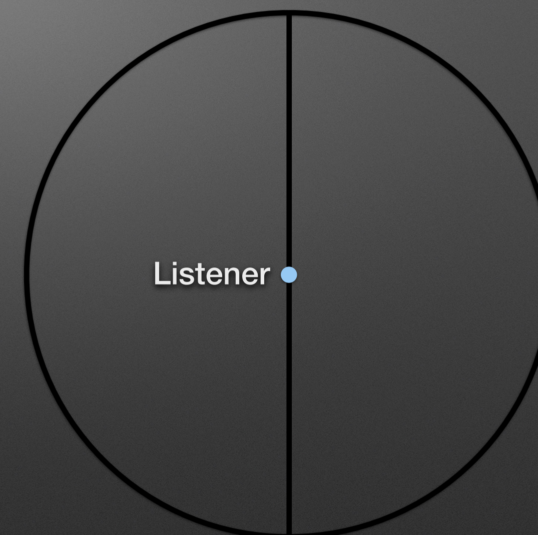
MADDER

- Inputs required:
  - Distance to closest surface
  - Direction of closest surface

MADDER

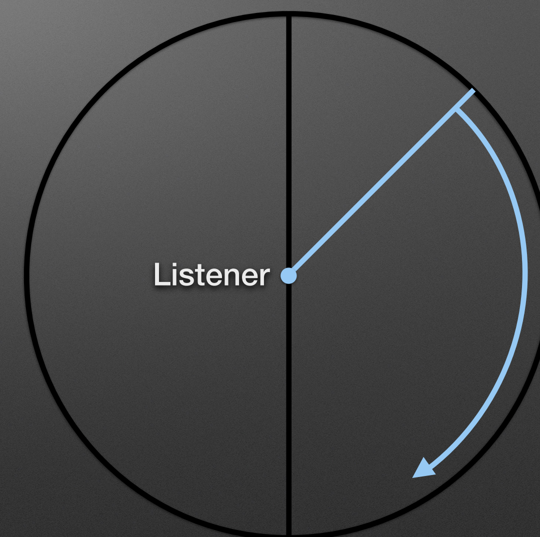
- Inputs required:
  - Distance to closest surface
  - Direction of closest surface
  - Material of closest surface

MADDER raycast



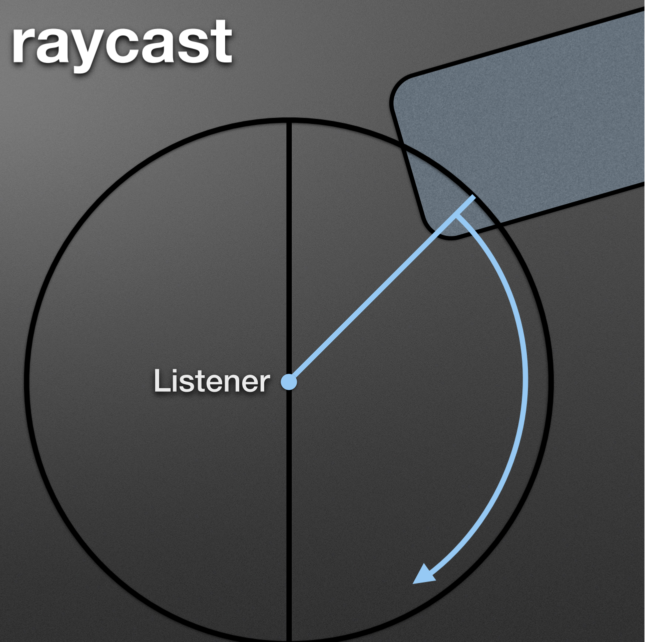
MADDER raycast

- Sweeping raycast around listener



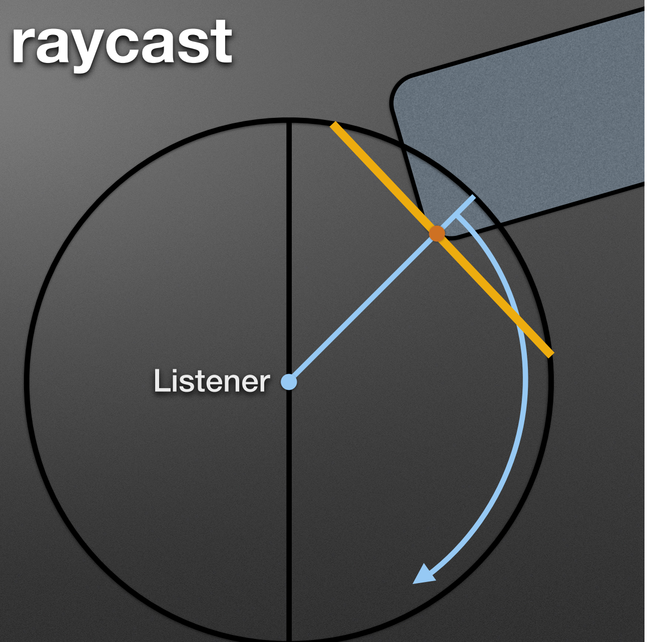
MADDER raycast

- Sweeping raycast around listener



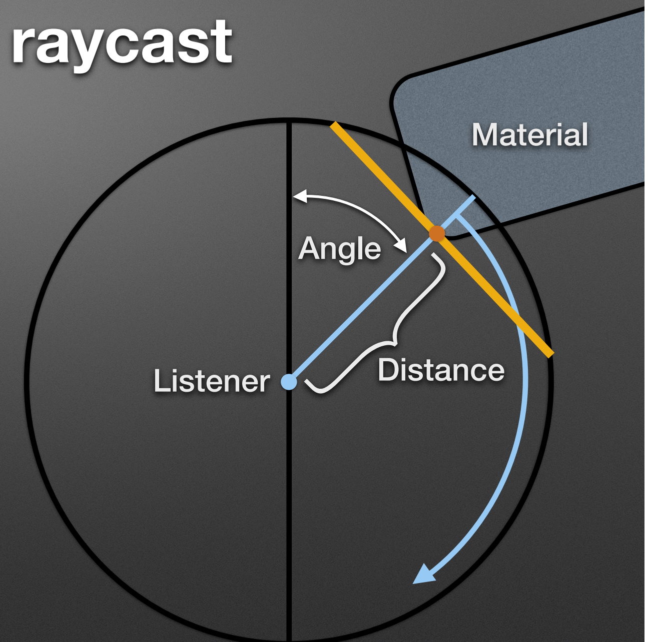
MADDER raycast

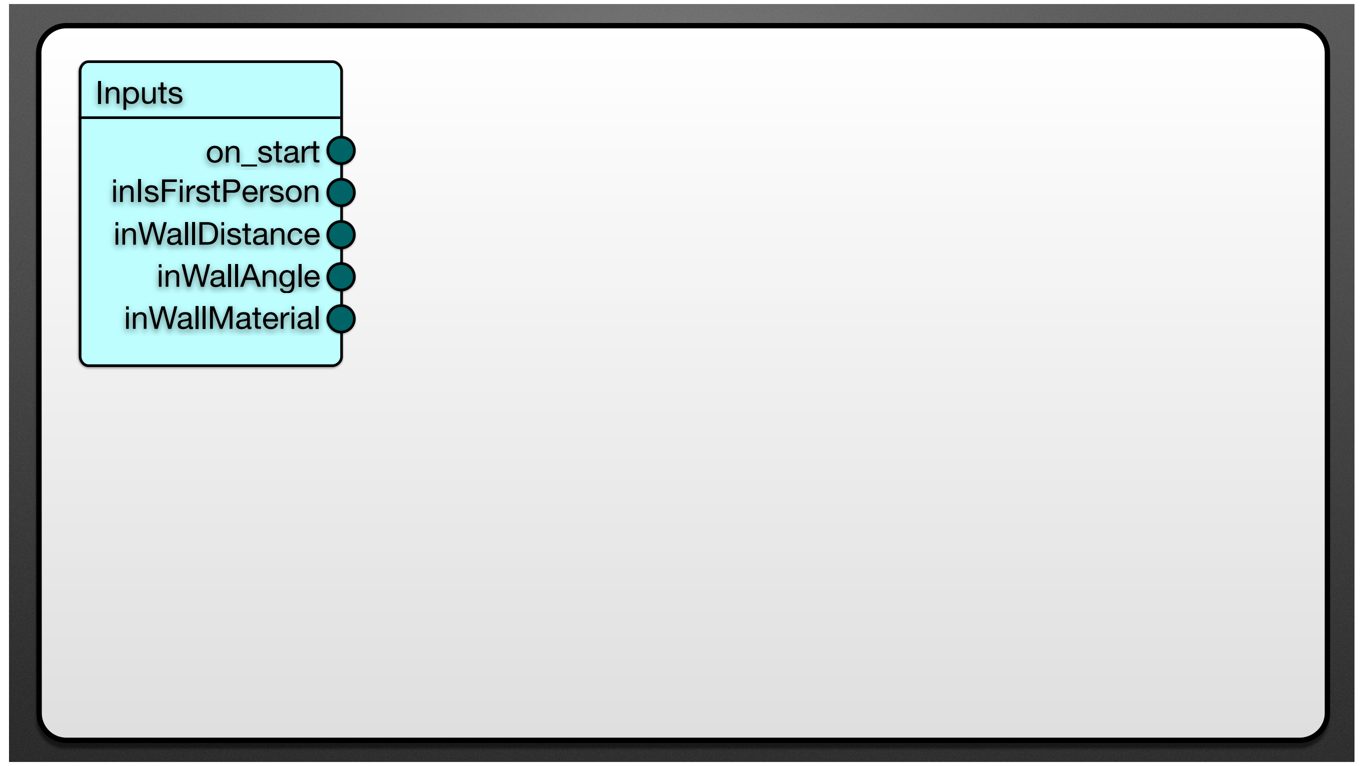
- Sweeping raycast around listener
- After one revolution, closest hit point is used

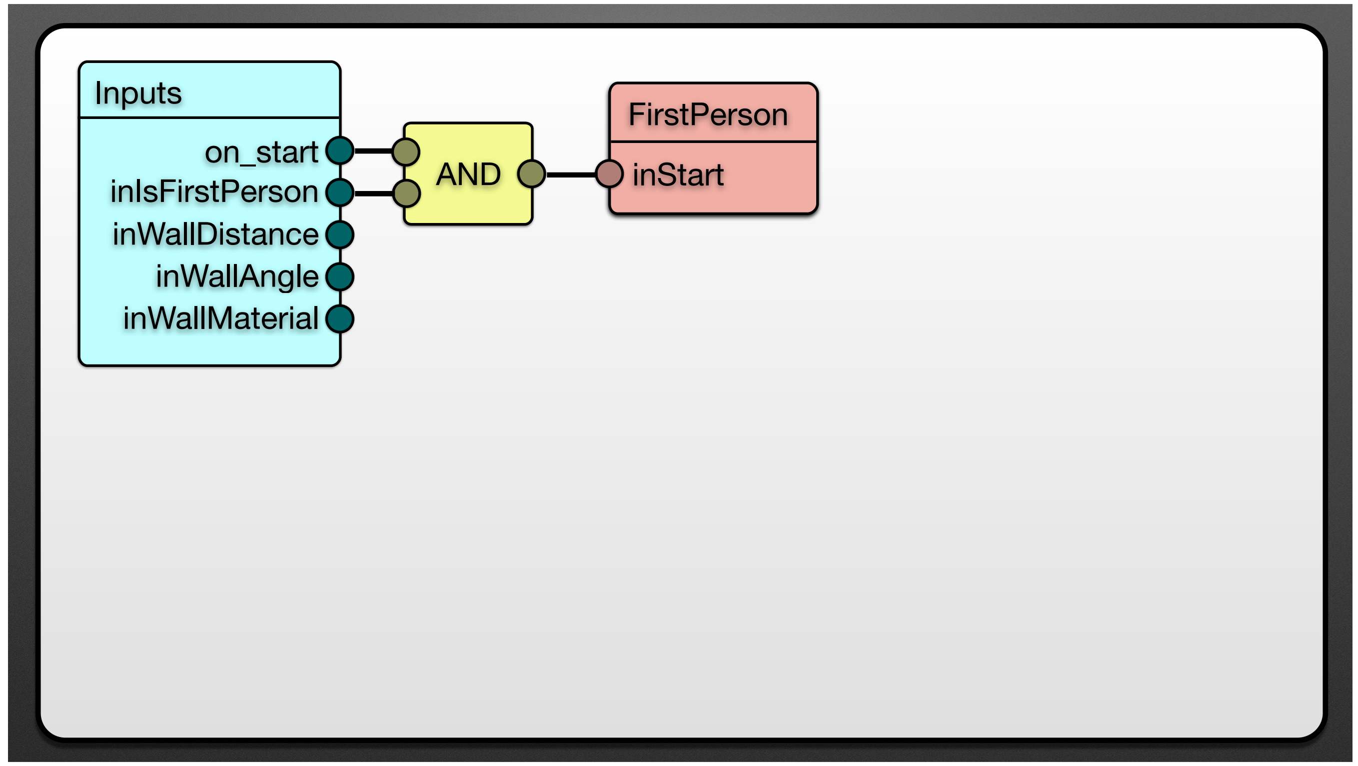


MADDER raycast

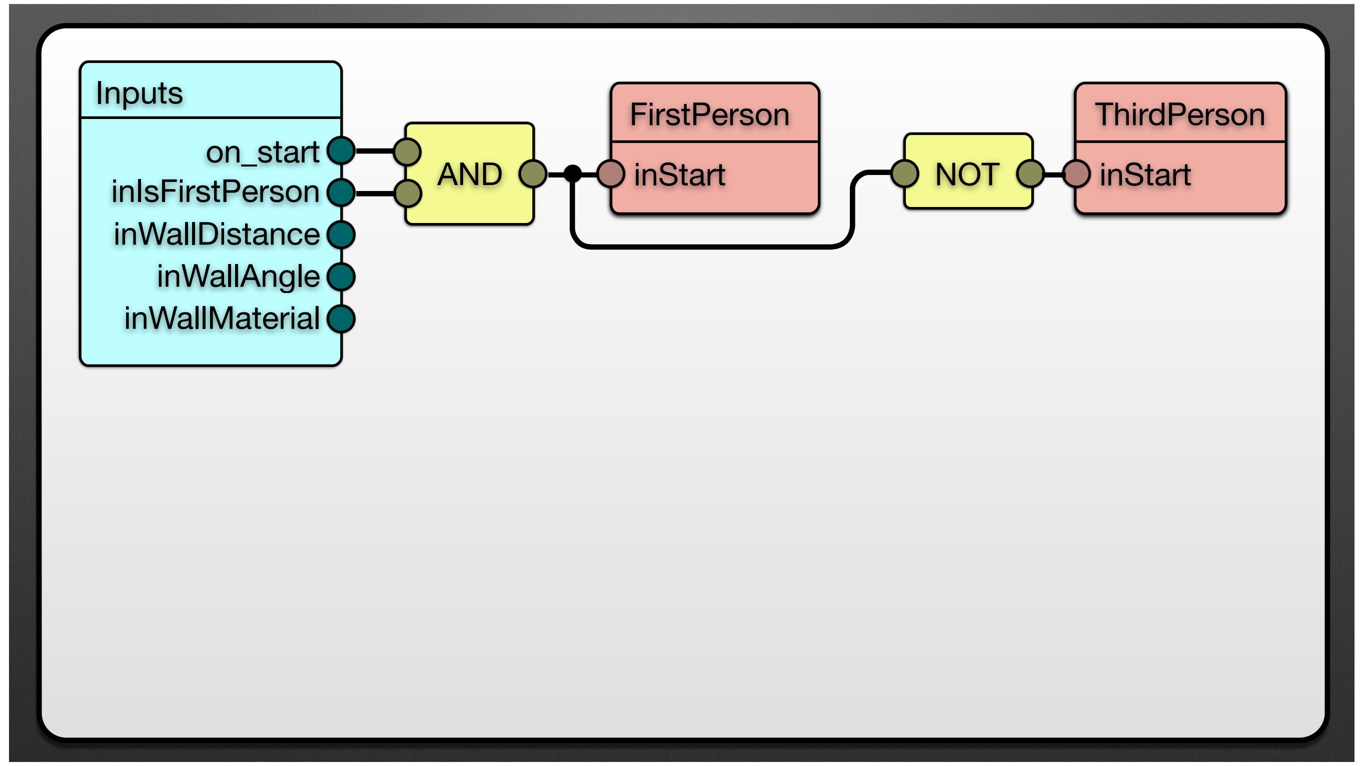
- Sweeping raycast around listener
- After one revolution, closest hit point is used
- Distance, angle and material is passed to sound

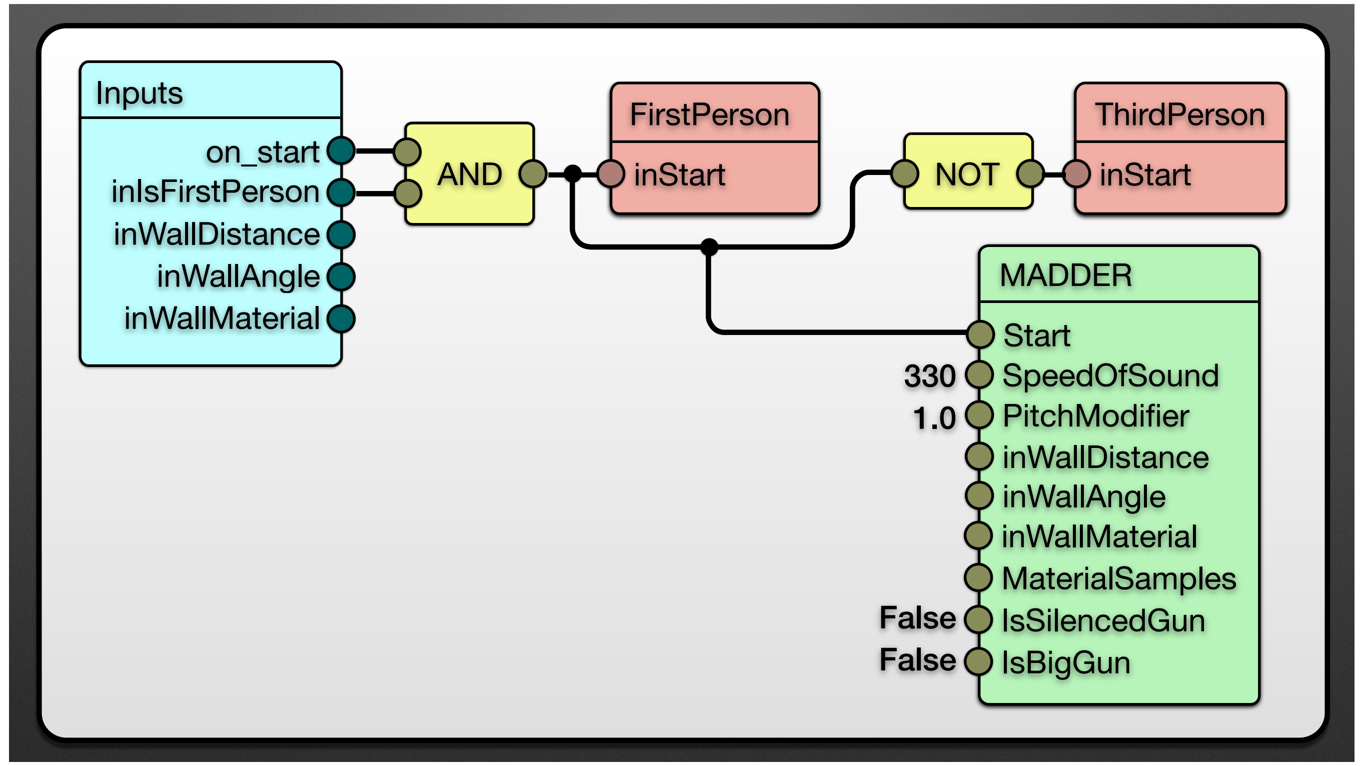



Inputs
on_start
inIsFirstPerson
inWallDistance
inWallAngle
inWallMaterial

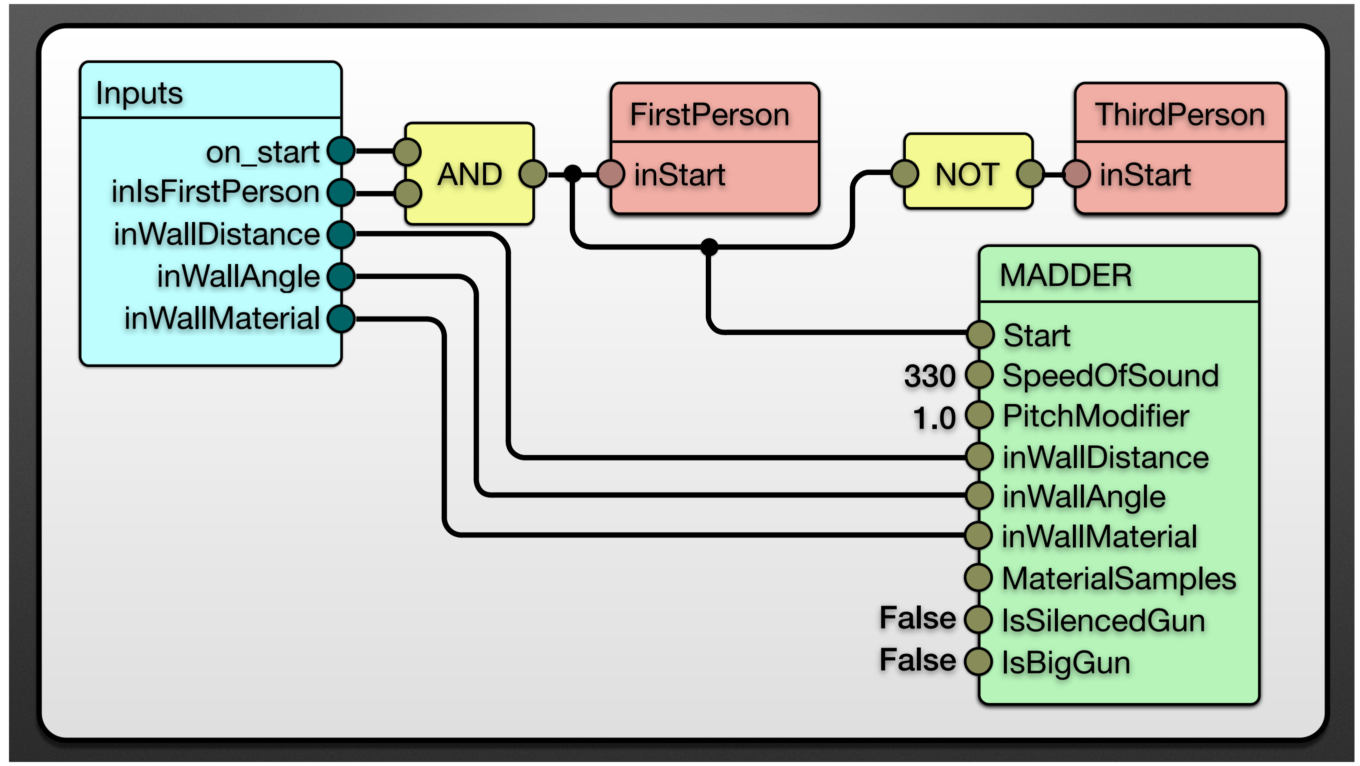


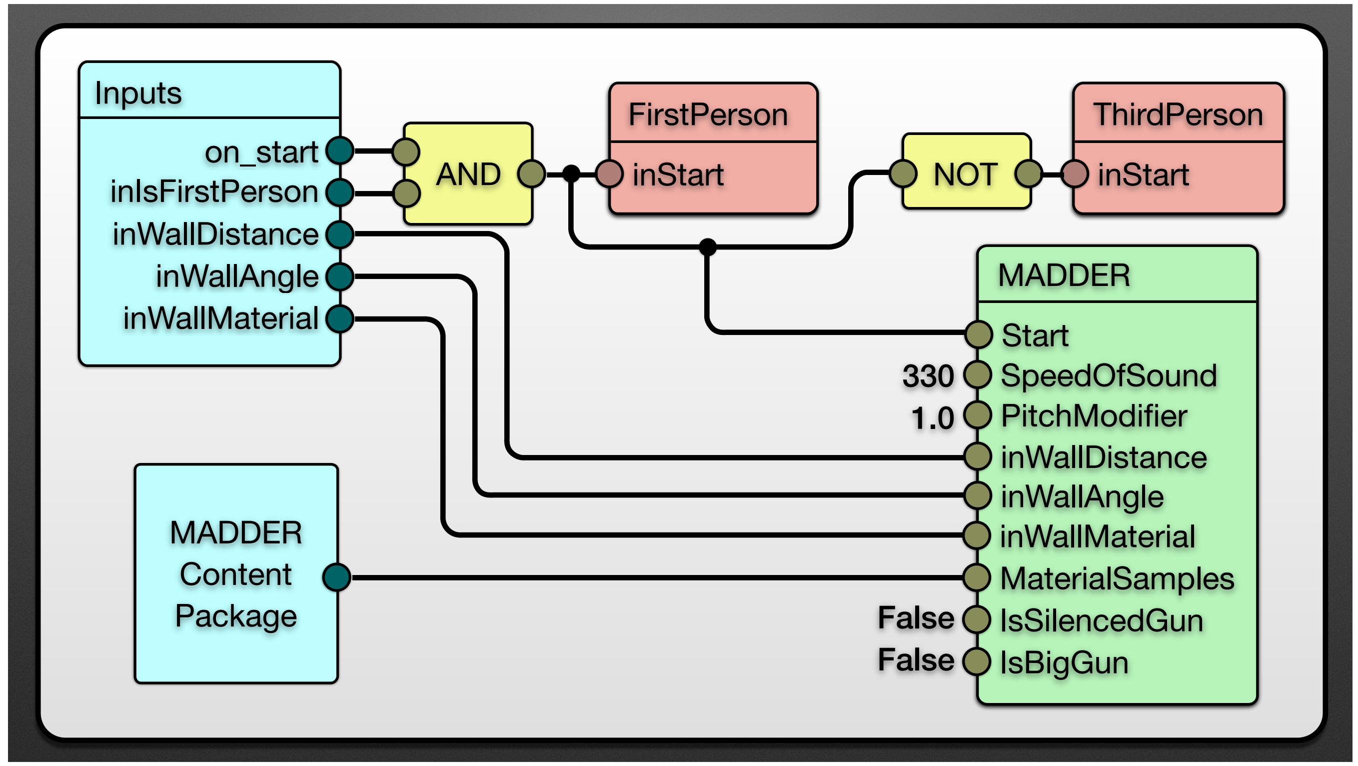
Inputs
on_start
inIsFirstPerson
inWallDistance
inWallAngle
inWallMaterial
AND
FirstPerson
inStart


Inputs
on_start
inIsFirstPerson
inWallDistance
inWallAngle
inWallMaterial
AND
FirstPerson
inStart
NOT
ThirdPerson
inStart

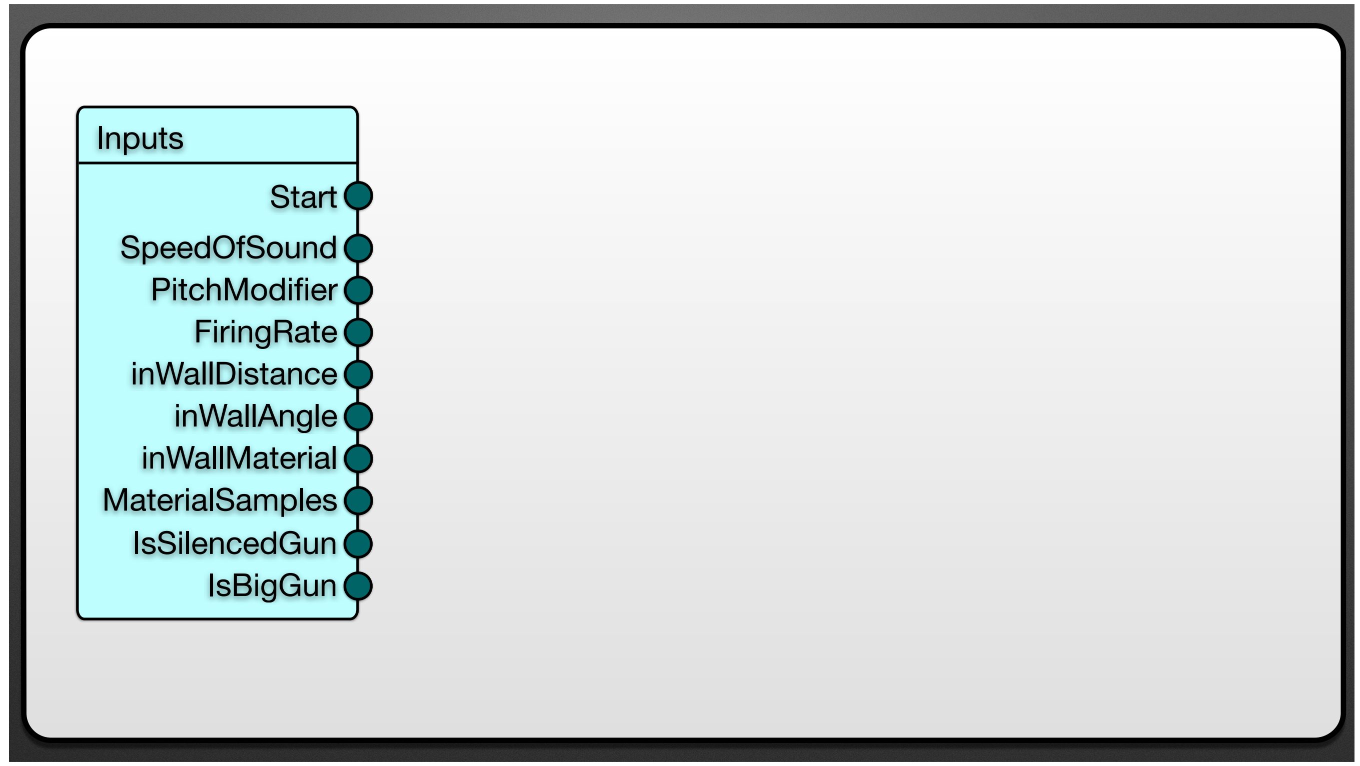


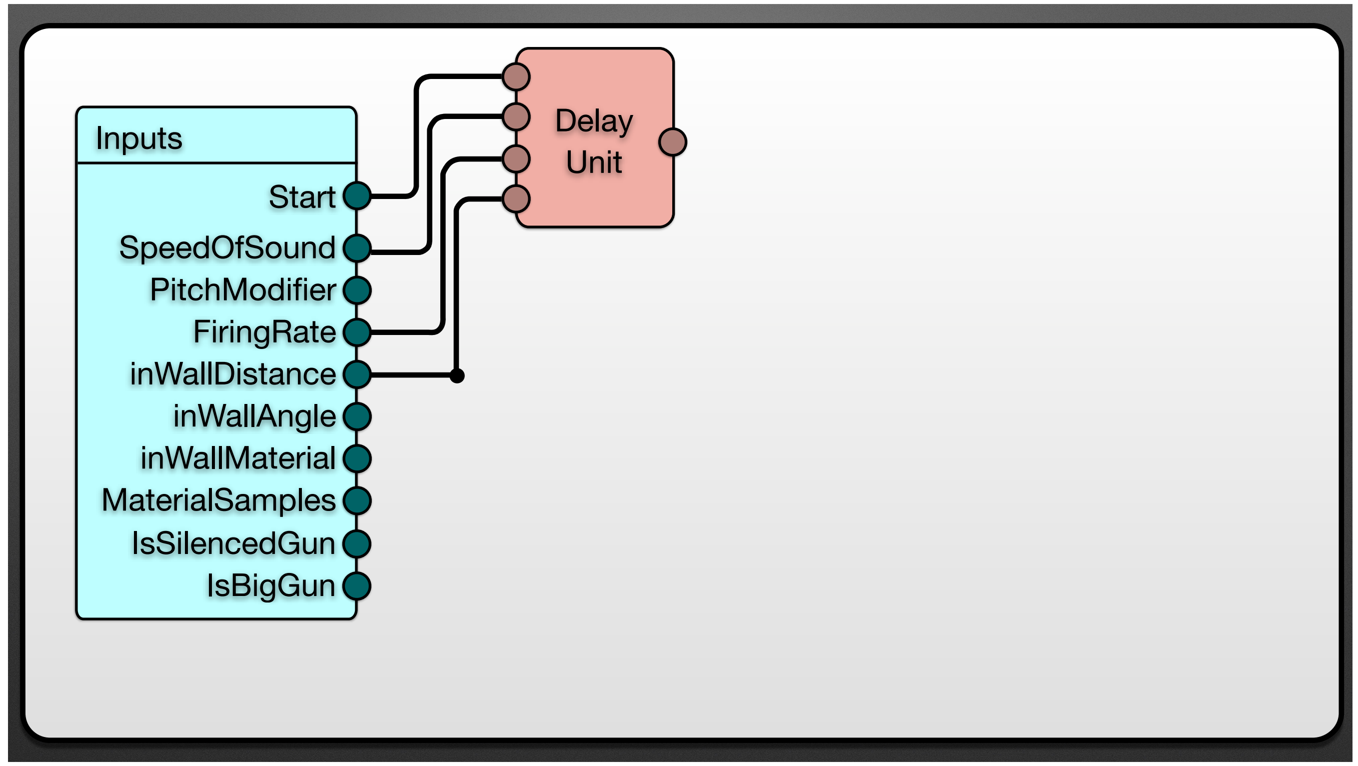
Inputs
on_start
inIsFirstPerson
inWallDistance
inWallAngle
inWallMaterial
AND
FirstPerson
inStart
NOT
ThirdPerson
inStart
MADDER
Start
330
SpeedOfSound
1.0
PitchModifier
inWallDistance
inWallAngle
inWallMaterial
MaterialSamples
False
IsSilencedGun
False
IsBigGun

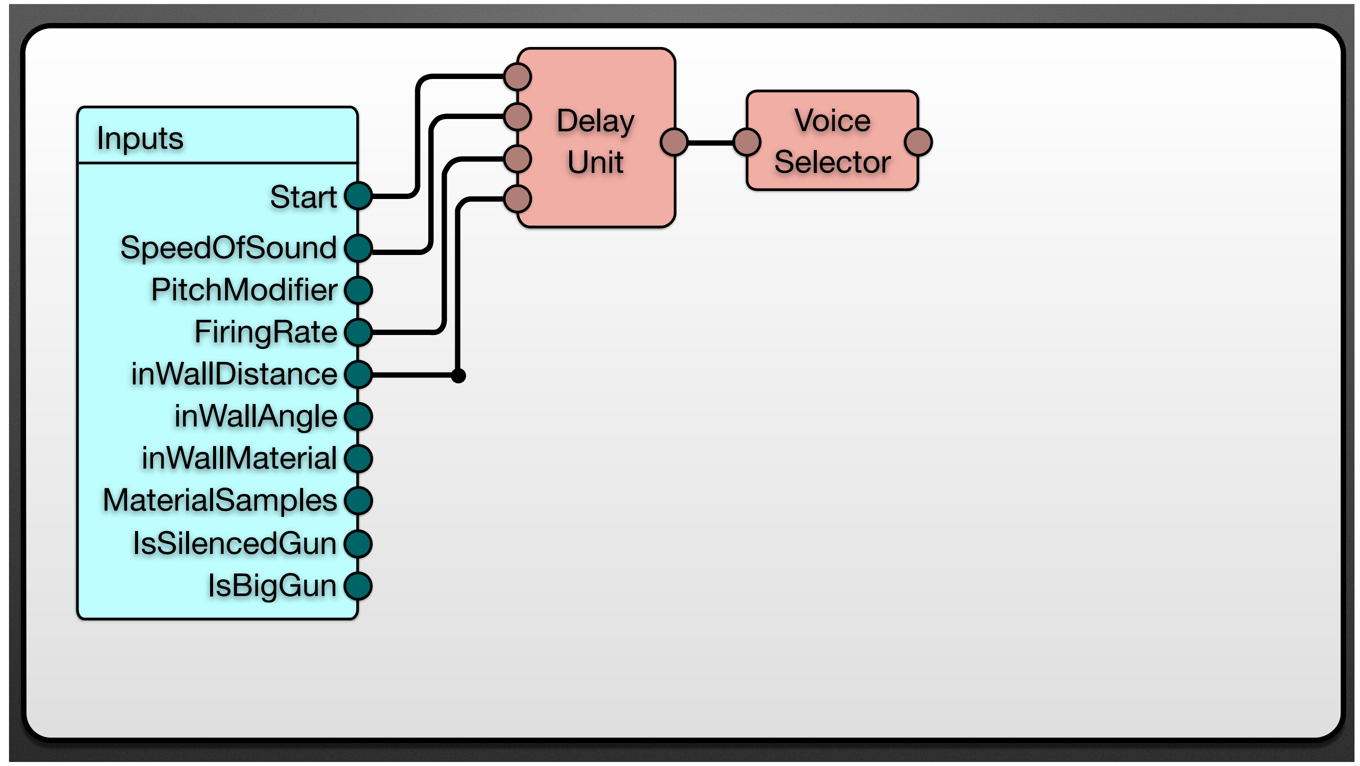

Inputs
on_start
inIsFirstPerson
inWallDistance
inWallAngle
inWallMaterial
AND
FirstPerson
inStart
NOT
ThirdPerson
inStart
MADDER
Start
330
SpeedOfSound
1.0
PitchModifier
inWallDistance
inWallAngle
inWallMaterial
MaterialSamples
False
IsSilencedGun
False
IsBigGun



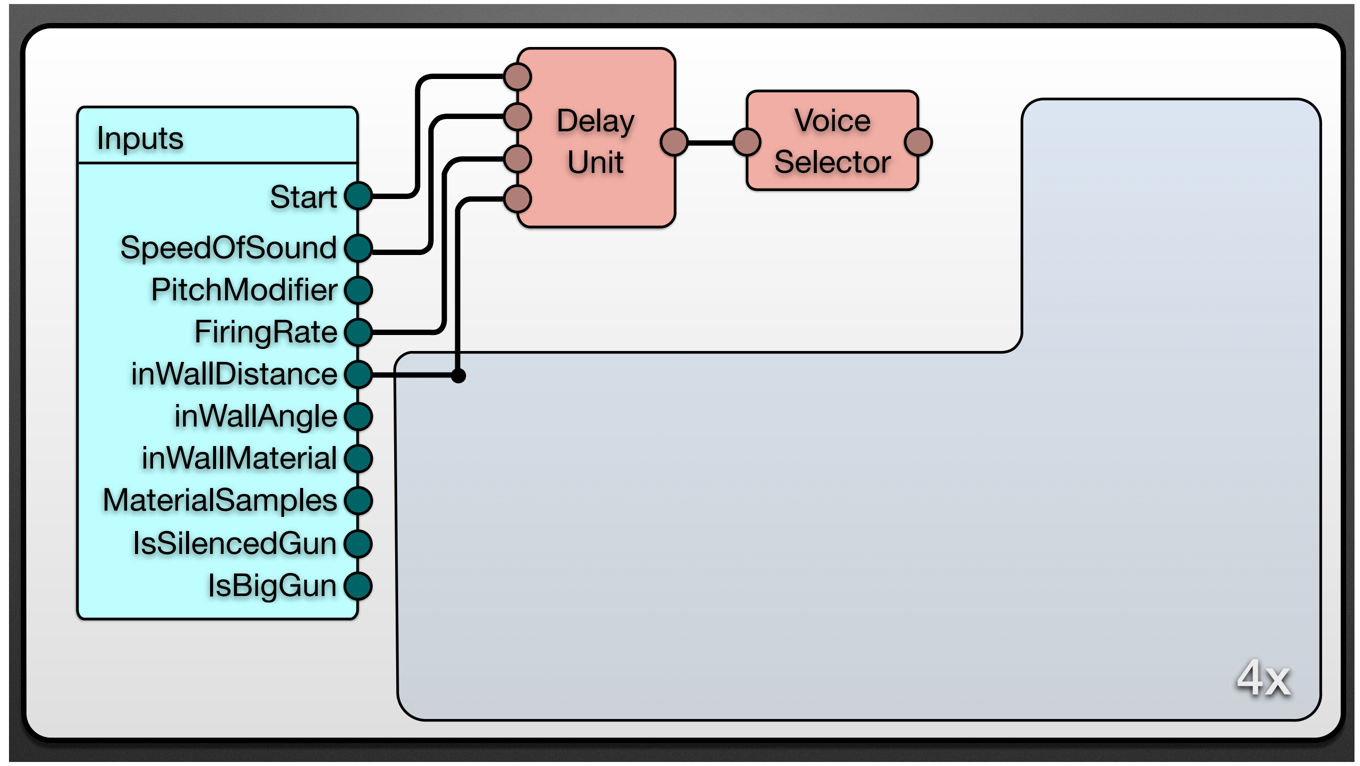
Inputs
on_start
inIsFirstPerson
inWallDistance
inWallAngle
inWallMaterial
AND
FirstPerson
inStart
NOT
ThirdPerson
inStart
MADDER
Start
330
SpeedOfSound
1.0
PitchModifier
inWallDistance
inWallAngle
inWallMaterial
MaterialSamples
False
IsSilencedGun
False
IsBigGun
MADDER
Content
Package


Inputs
Start
SpeedOfSound
PitchModifier
FiringRate
inWallDistance
inWallAngle
inWallMaterial
MaterialSamples
IsSilencedGun
IsBigGun

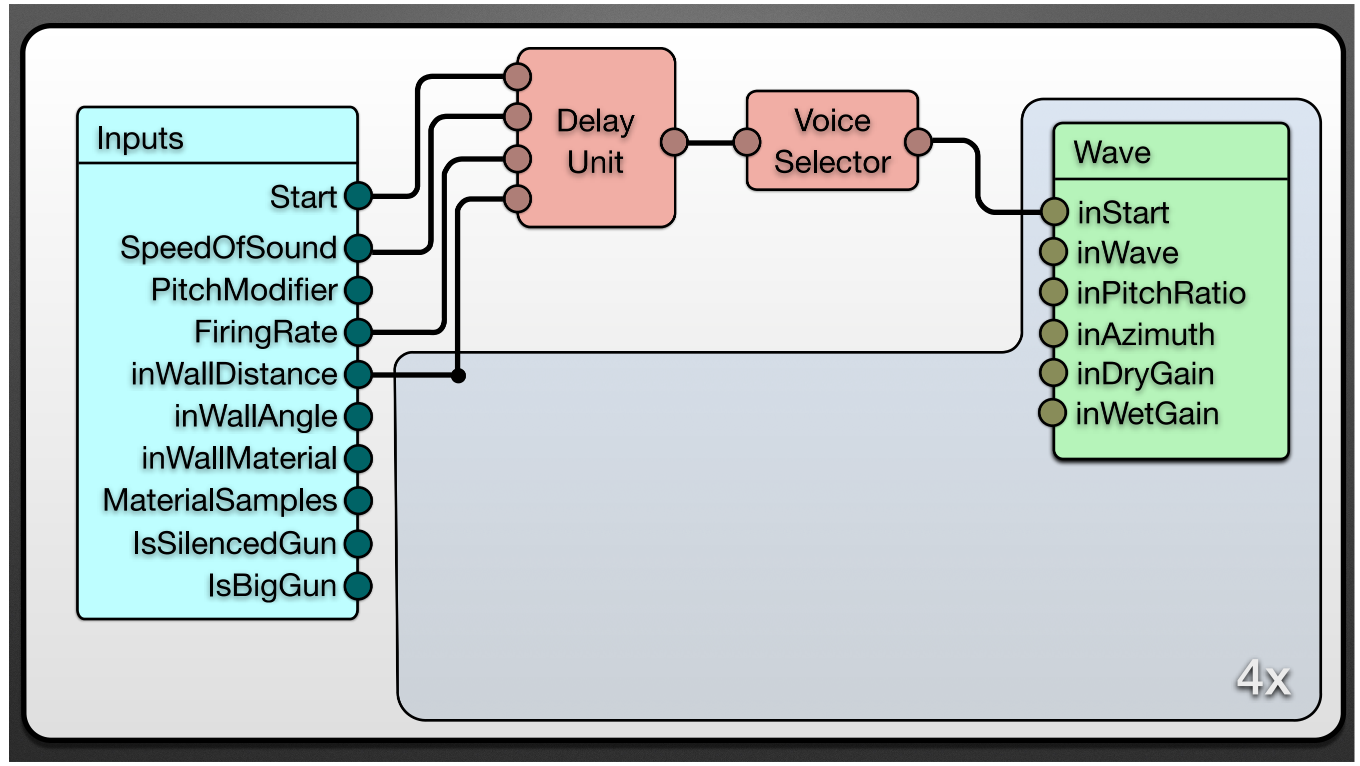

Delay
Unit
Inputs
Start
SpeedOfSound
PitchModifier
FiringRate
inWallDistance
inWallAngle
inWallMaterial
MaterialSamples
IsSilencedGun
IsBigGun



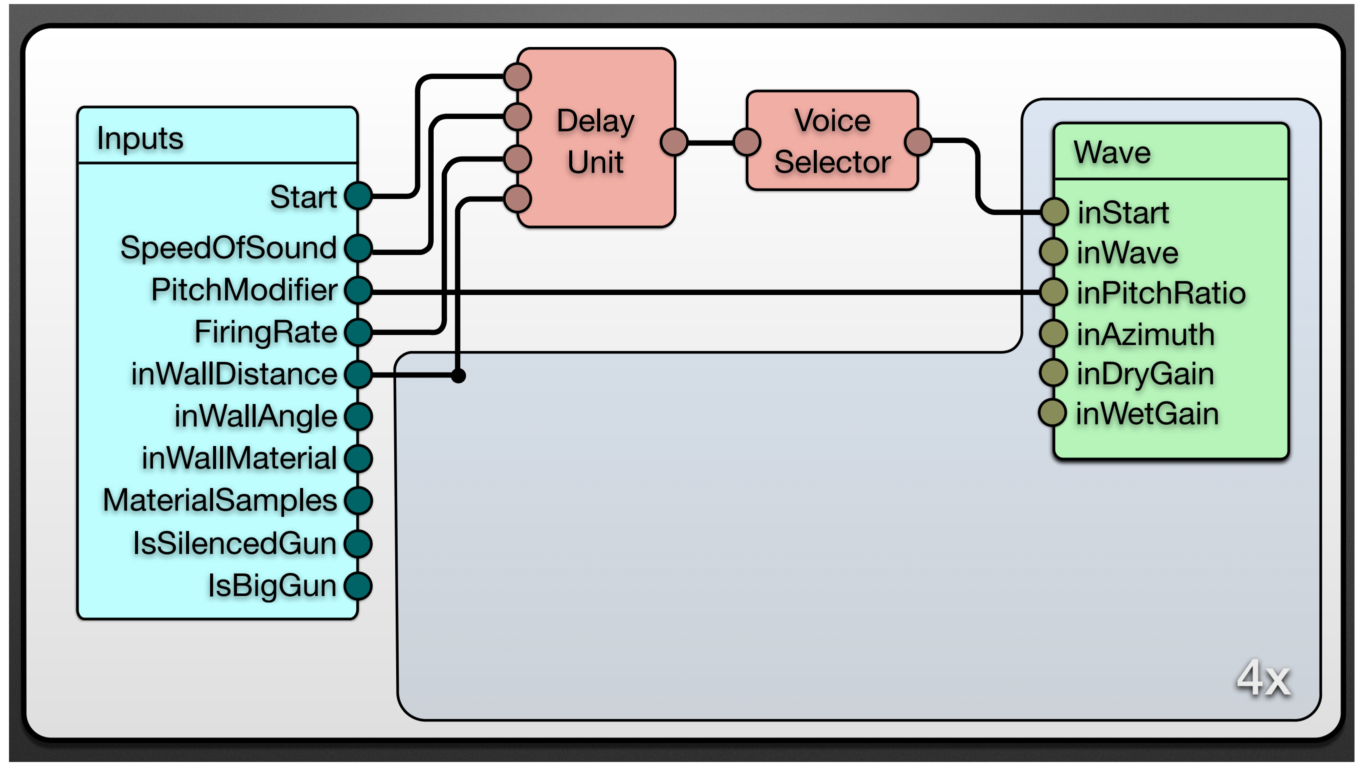
Inputs
Start
SpeedOfSound
PitchModifier
FiringRate
inWallDistance
inWallAngle
inWallMaterial
MaterialSamples
IsSilencedGun
IsBigGun
Delay Unit
Voice Selector



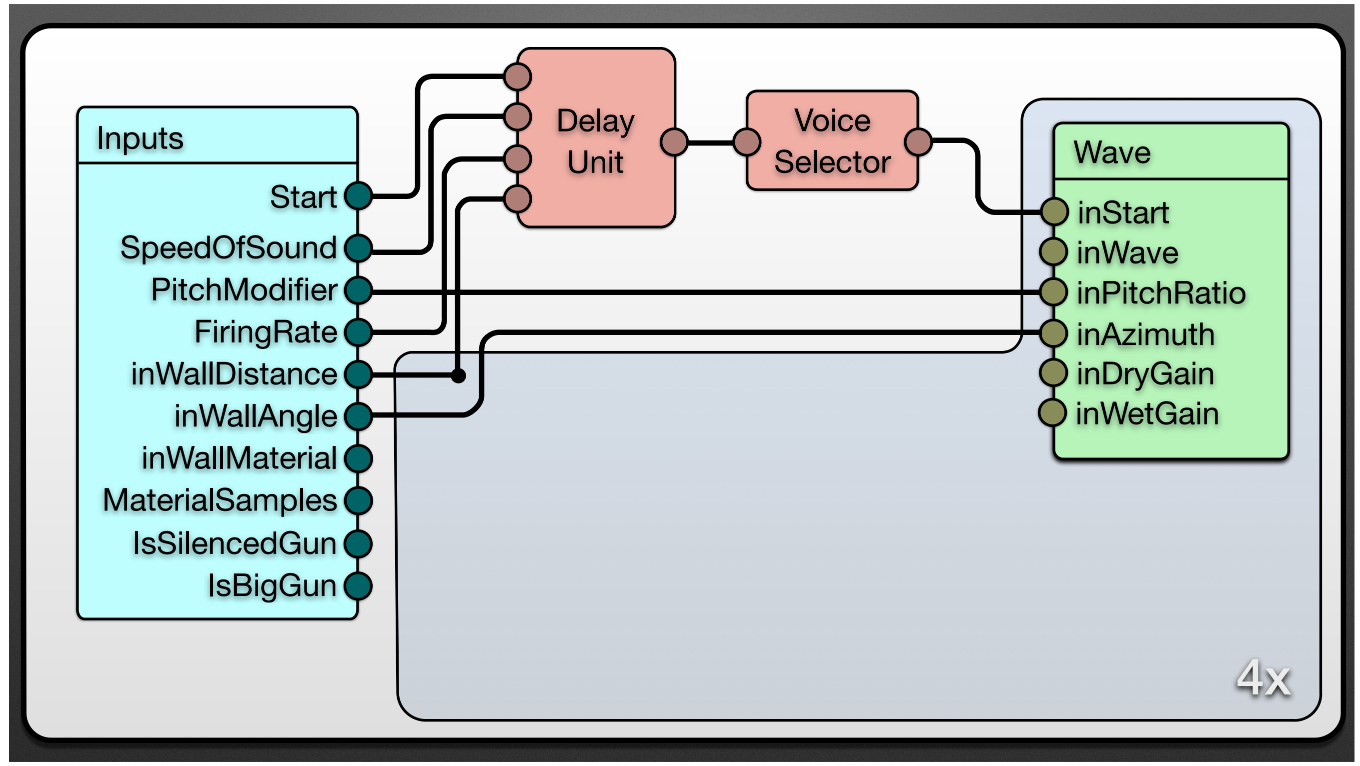
Inputs
Start
SpeedOfSound
PitchModifier
FiringRate
inWallDistance
inWallAngle
inWallMaterial
MaterialSamples
IsSilencedGun
IsBigGun
Delay Unit
Voice Selector
4x



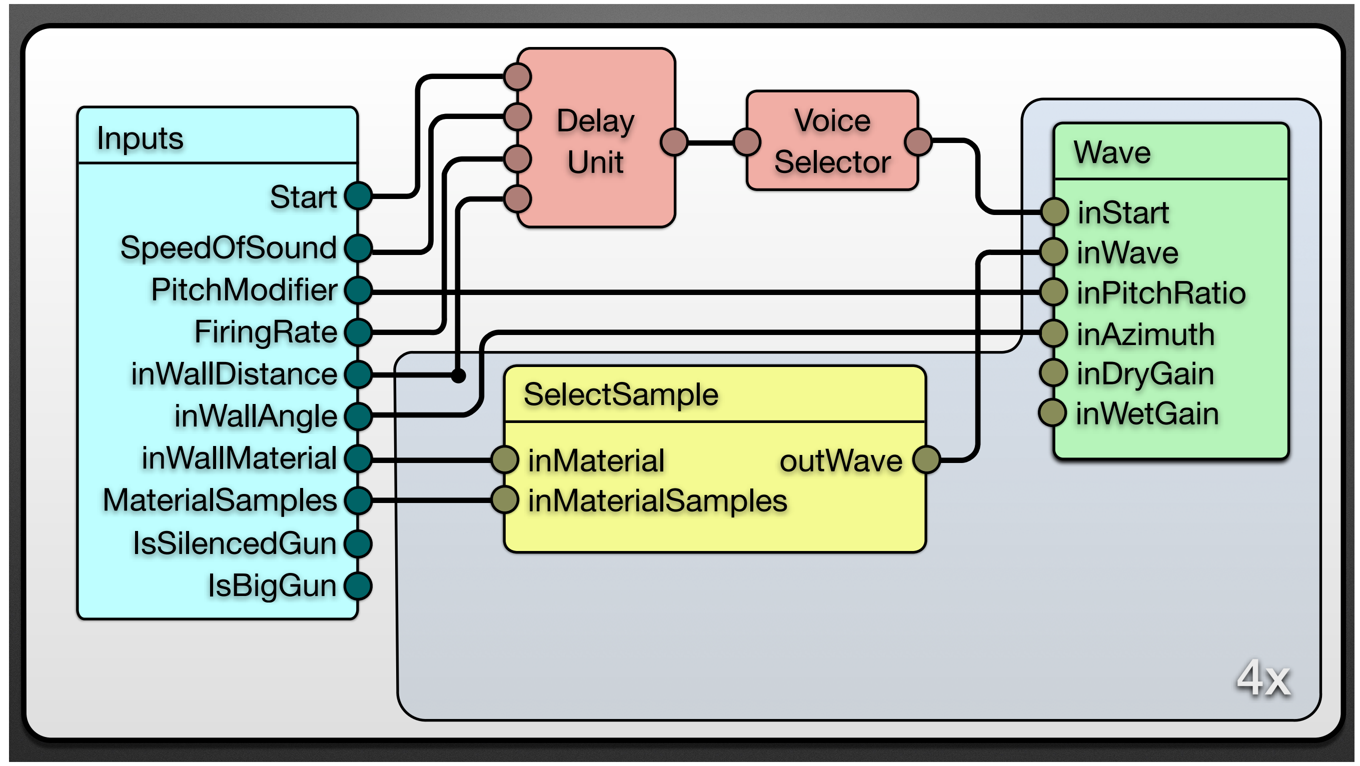
Inputs
Start
SpeedOfSound
PitchModifier
FiringRate
inWallDistance
inWallAngle
inWallMaterial
MaterialSamples
IsSilencedGun
IsBigGun
Delay Unit
Voice Selector
Wave
inStart
inWave
inPitchRatio
inAzimuth
inDryGain
inWetGain
4x



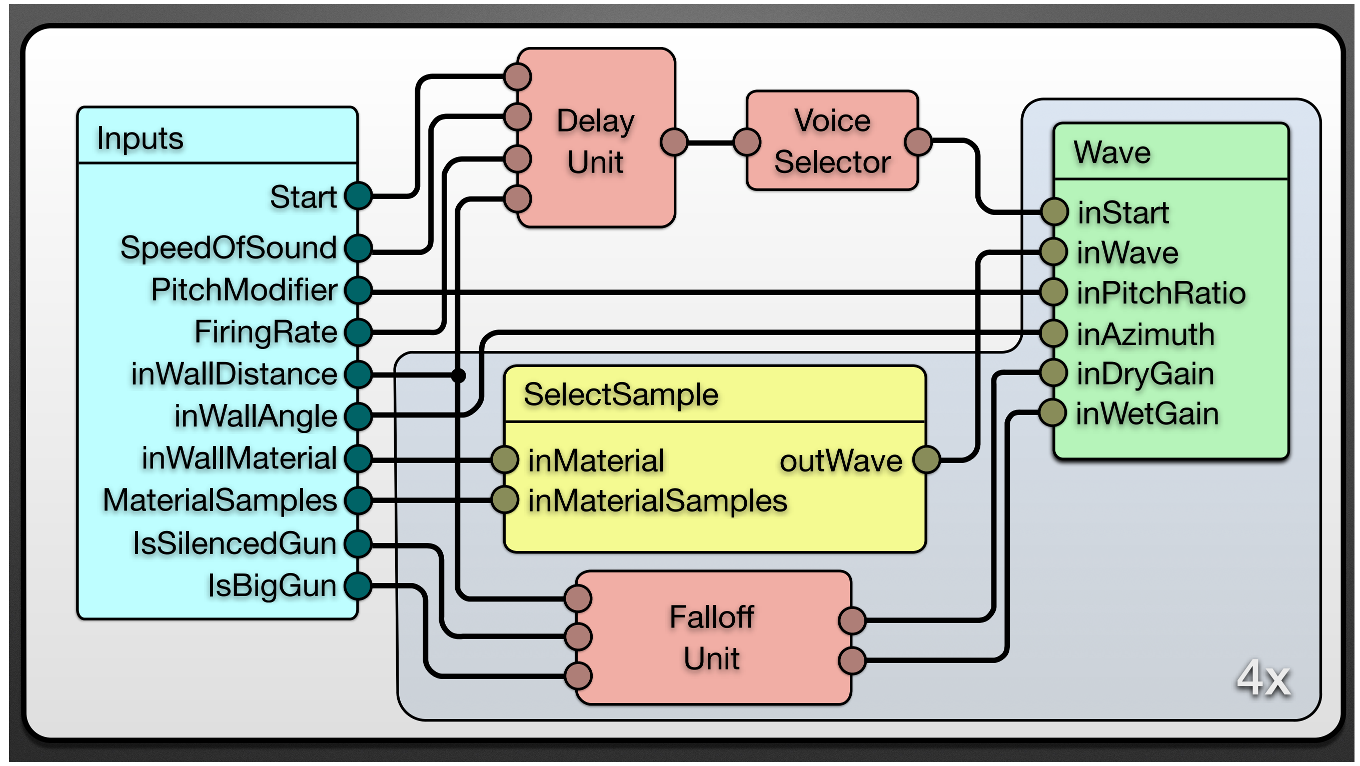
Inputs
Start
SpeedOfSound
PitchModifier
FiringRate
inWallDistance
inWallAngle
inWallMaterial
MaterialSamples
IsSilencedGun
IsBigGun
Delay Unit
Voice Selector
Wave
inStart
inWave
inPitchRatio
inAzimuth
inDryGain
inWetGain
4x



Inputs
Start
SpeedOfSound
PitchModifier
FiringRate
inWallDistance
inWallAngle
inWallMaterial
MaterialSamples
IsSilencedGun
IsBigGun
Delay Unit
Voice Selector
Wave
inStart
inWave
inPitchRatio
inAzimuth
inDryGain
inWetGain
4x



Inputs
Start
SpeedOfSound
PitchModifier
FiringRate
inWallDistance
inWallAngle
inWallMaterial
MaterialSamples
IsSilencedGun
IsBigGun
Delay
Unit
Voice
Selector
Wave
inStart
inWave
inPitchRatio
inAzimuth
inDryGain
inWetGain
SelectSample
inMaterial
outWave
inMaterialSamples
4x



Inputs
Start
SpeedOfSound
PitchModifier
FiringRate
inWallDistance
inWallAngle
inWallMaterial
MaterialSamples
IsSilencedGun
IsBigGun
Delay Unit
Voice Selector
Wave
inStart
inWave
inPitchRatio
inAzimuth
inDryGain
inWetGain
SelectSample
inMaterial
outWave
inMaterialSamples
Falloff Unit
4x

Video showing MADDER effect for 4 different materials

Video showing MADDER effect for 4 different materials

Video showing initial MADDER prototype with 4 different materials in scene.

Video showing initial MADDER prototype with 4 different materials in scene.

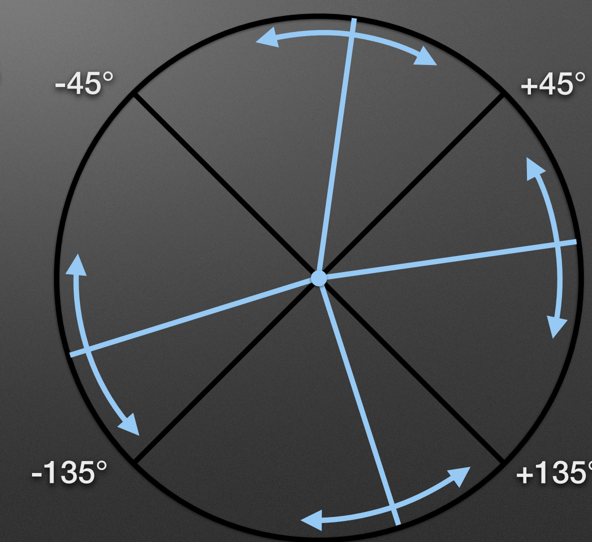
MADDER Four-angle raycast

MADDER Four-angle raycast

- Divide in four quadrants around listener

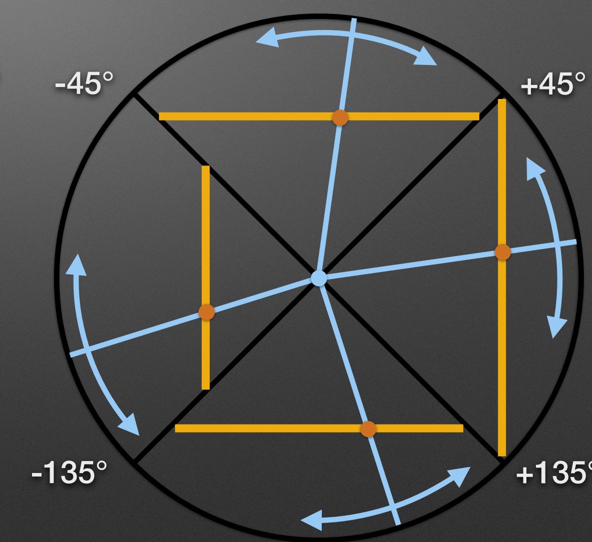
MADDER Four-angle raycast

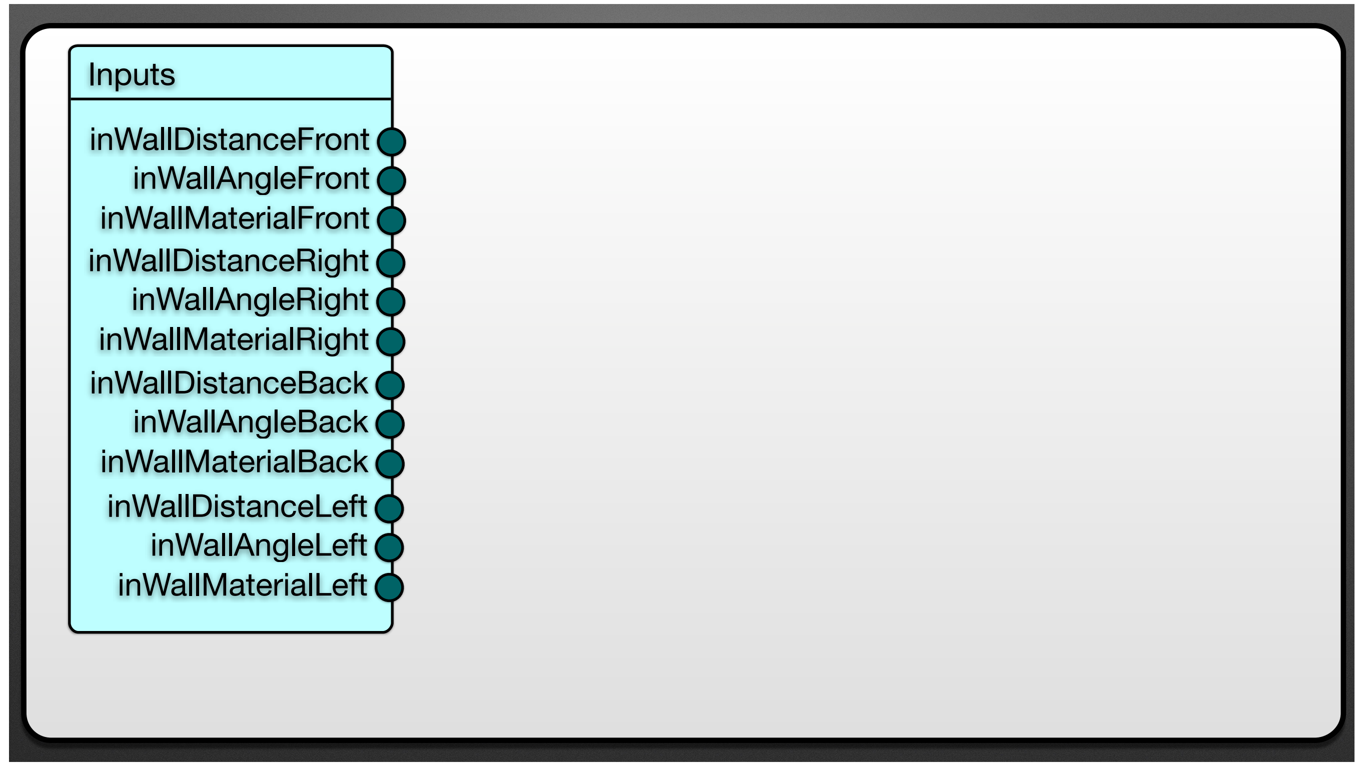
- Divide in four quadrants around listener
- One sweeping raycast in each quadrant

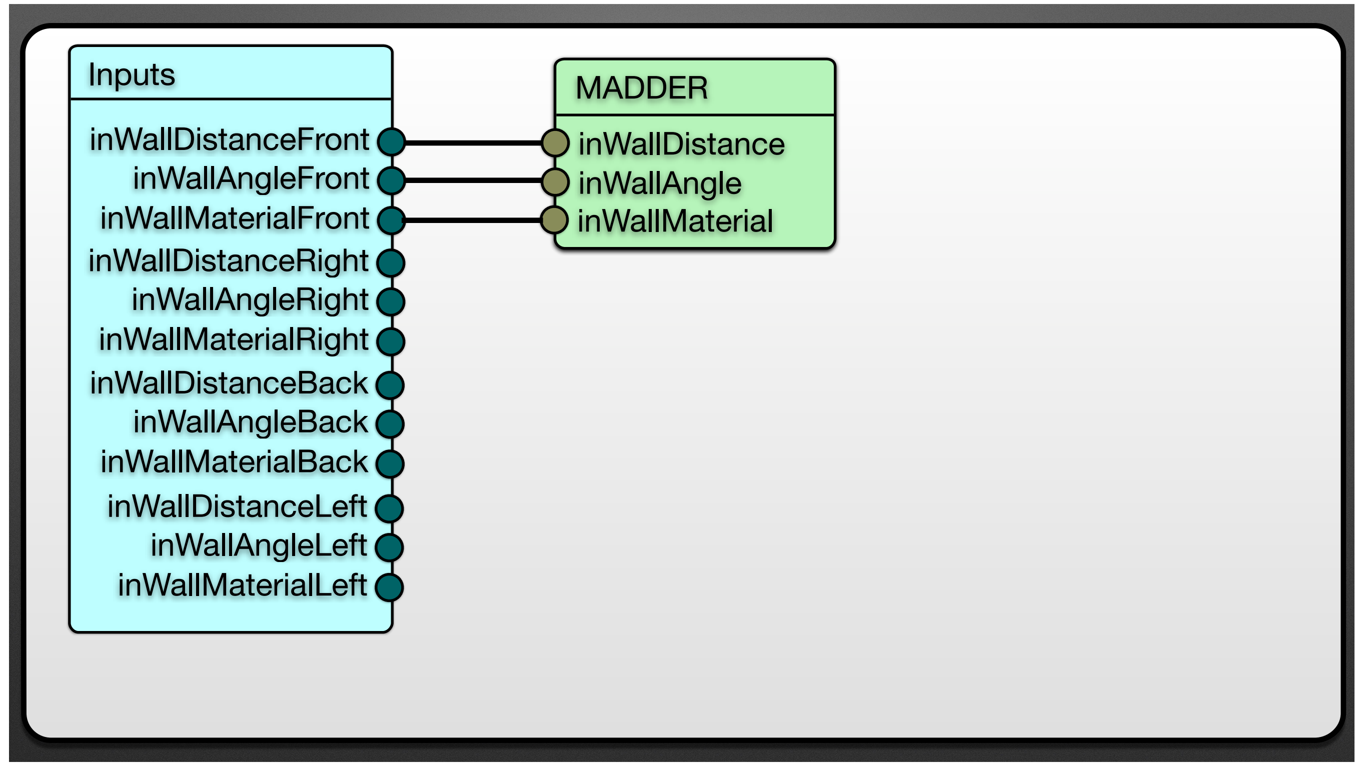


MADDER Four-angle raycast

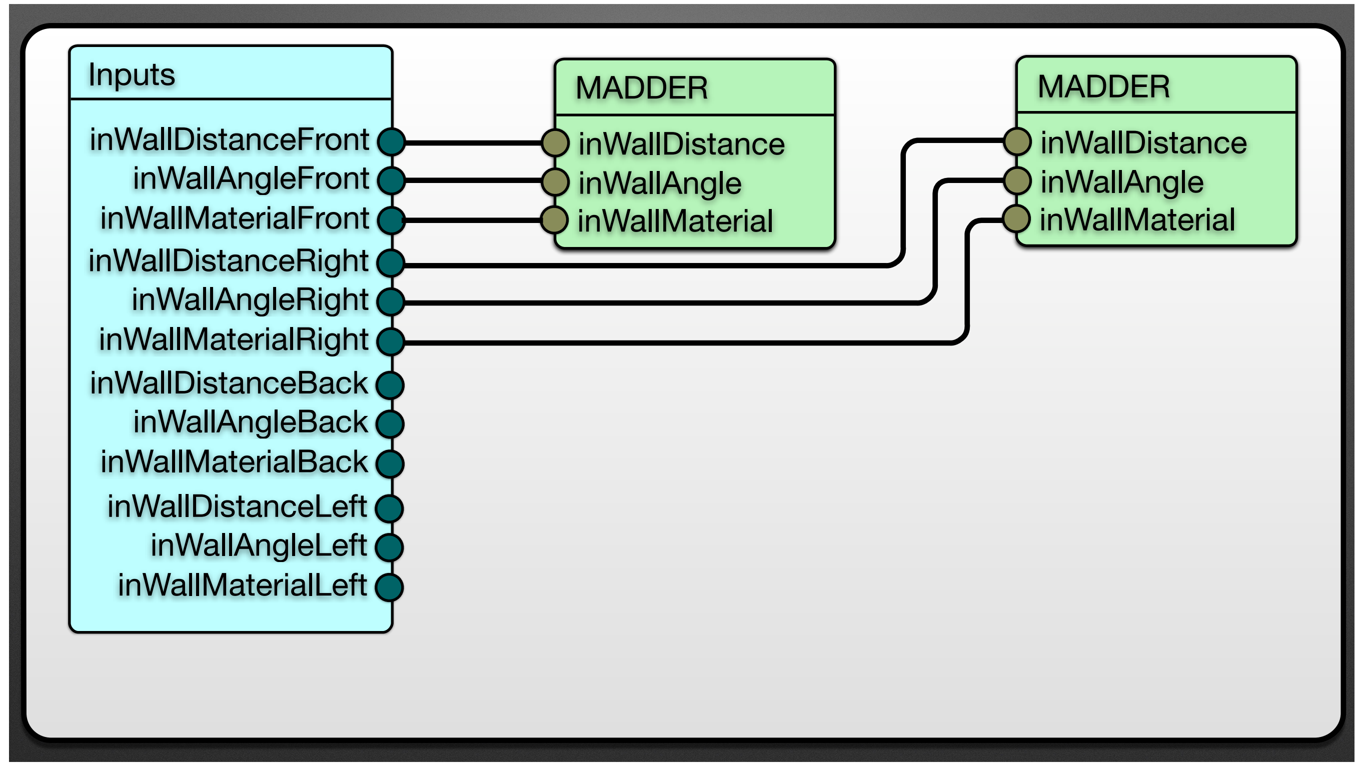
- Divide in four quadrants around listener
- One sweeping raycast in each quadrant
- Closest hit point in each quadrant is taken

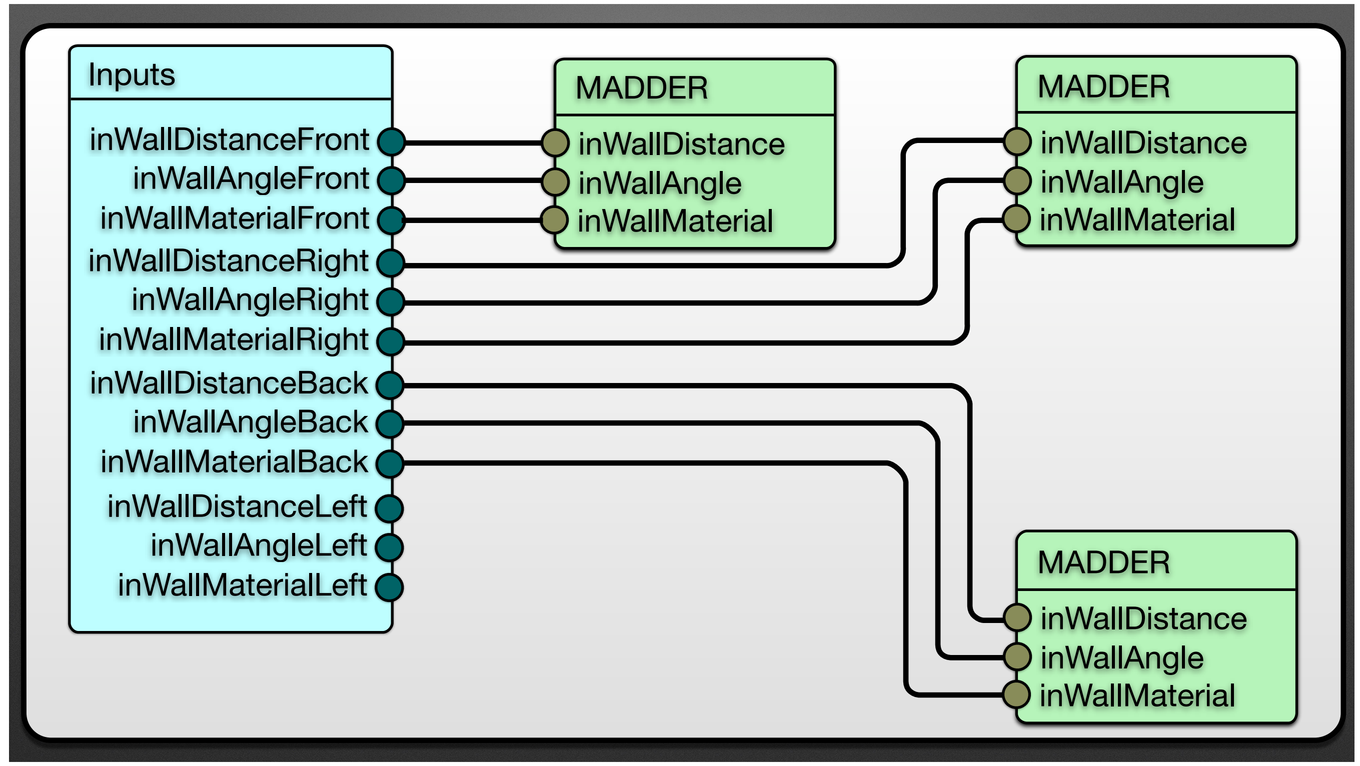


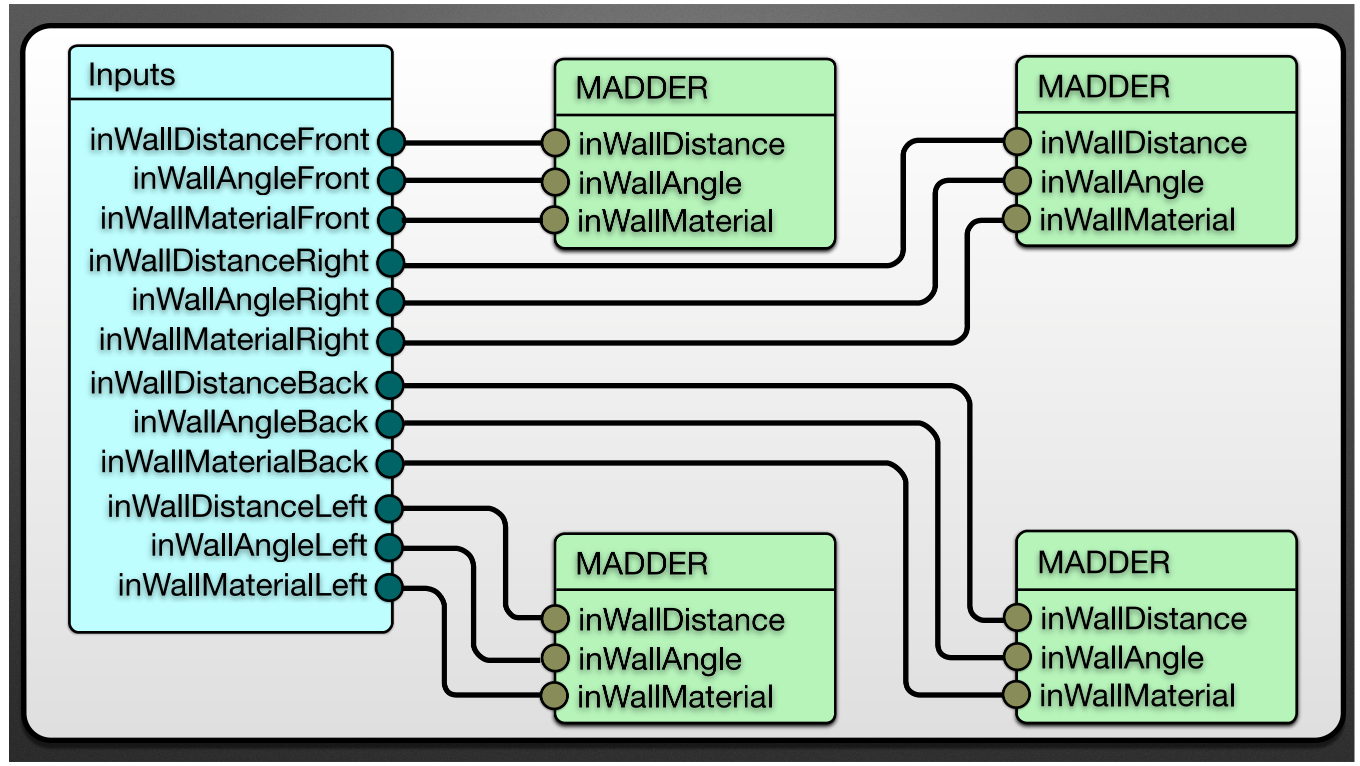

Inputs
inWallDistanceFront
inWallAngleFront
inWallMaterialFront
inWallDistanceRight
inWallAngleRight
inWallMaterialRight
inWallDistanceBack
inWallAngleBack
inWallMaterialBack
inWallDistanceLeft
inWallAngleLeft
inWallMaterialLeft



Inputs
inWallDistanceFront
inWallAngleFront
inWallMaterialFront
inWallDistanceRight
inWallAngleRight
inWallMaterialRight
inWallDistanceBack
inWallAngleBack
inWallMaterialBack
inWallDistanceLeft
inWallAngleLeft
inWallMaterialLeft
MADDER
inWallDistance
inWallAngle
inWallMaterial


Inputs
inWallDistanceFront
inWallAngleFront
inWallMaterialFront
inWallDistanceRight
inWallAngleRight
inWallMaterialRight
inWallDistanceBack
inWallAngleBack
inWallMaterialBack
inWallDistanceLeft
inWallAngleLeft
inWallMaterialLeft
MADDER
inWallDistance
inWallAngle
inWallMaterial
MADDER
inWallDistance
inWallAngle
inWallMaterial


Inputs
inWallDistanceFront
inWallAngleFront
inWallMaterialFront
inWallDistanceRight
inWallAngleRight
inWallMaterialRight
inWallDistanceBack
inWallAngleBack
inWallMaterialBack
inWallDistanceLeft
inWallAngleLeft
inWallMaterialLeft
MADDER
inWallDistance
inWallAngle
inWallMaterial
MADDER
inWallDistance
inWallAngle
inWallMaterial
MADDER
inWallDistance
inWallAngle
inWallMaterial



Inputs
inWallDistanceFront
inWallAngleFront
inWallMaterialFront
inWallDistanceRight
inWallAngleRight
inWallMaterialRight
inWallDistanceBack
inWallAngleBack
inWallMaterialBack
inWallDistanceLeft
inWallAngleLeft
inWallMaterialLeft
MADDER
inWallDistance
inWallAngle
inWallMaterial
MADDER
inWallDistance
inWallAngle
inWallMaterial
MADDER
inWallDistance
inWallAngle
inWallMaterial
MADDER
inWallDistance
inWallAngle
inWallMaterial

Video showing final 4-angle MADDER with 4 materials placed in the scene

Video showing final 4-angle MADDER with 4 materials placed in the scene

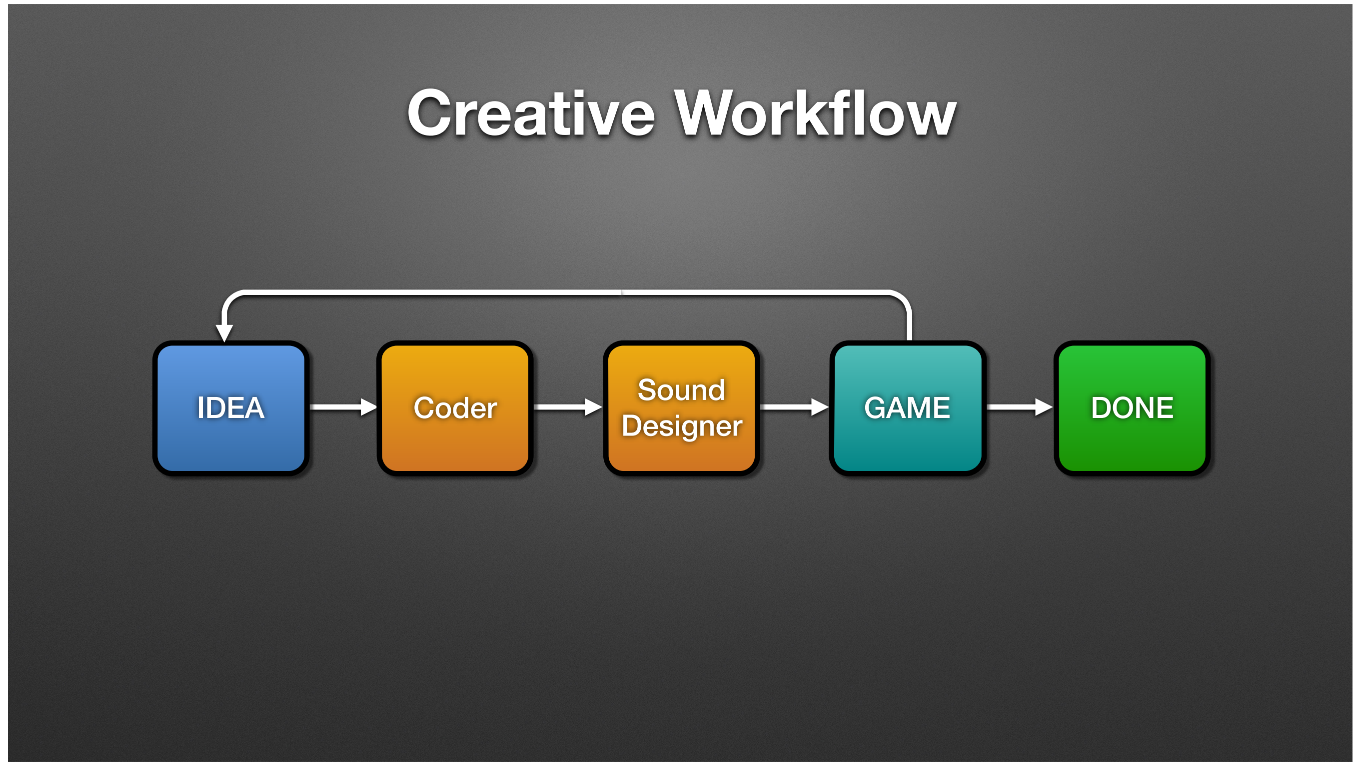
Capture from final game with MADDER disabled

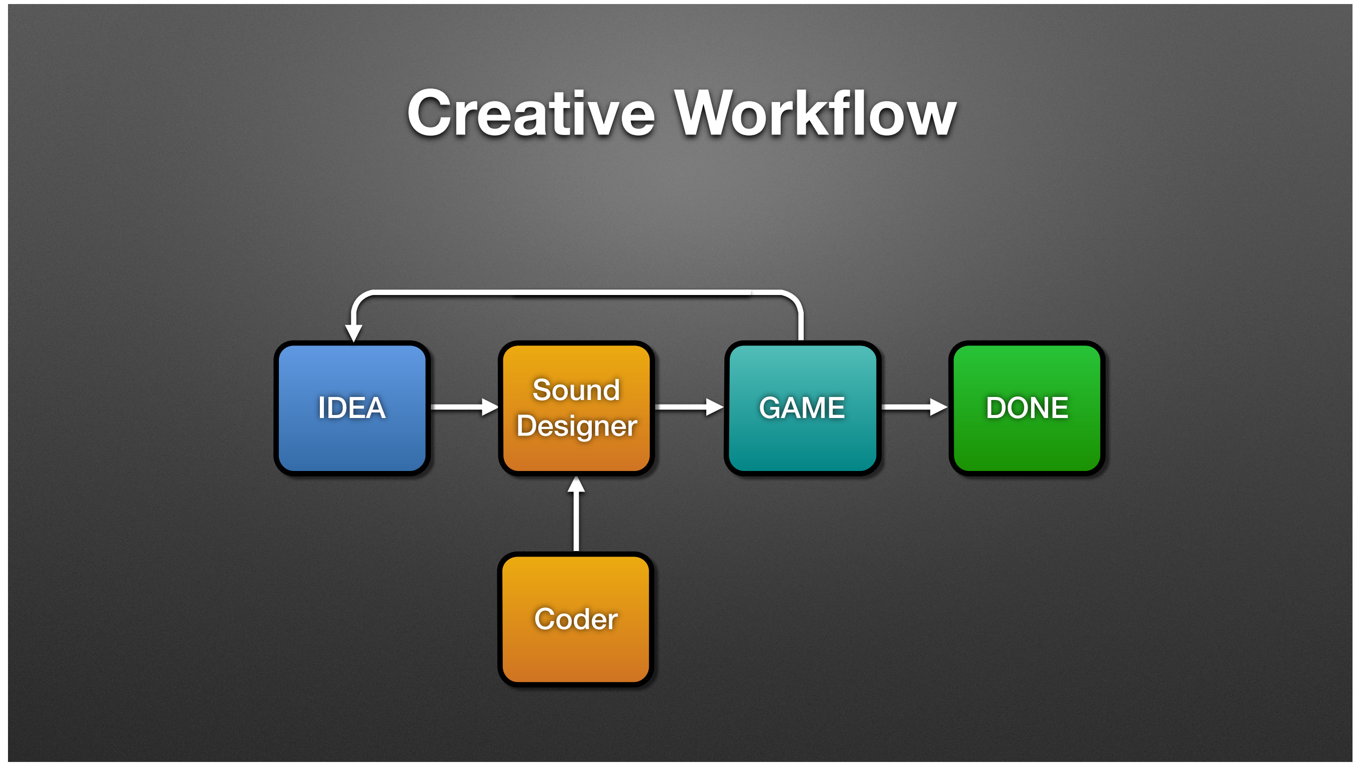
Capture from final game with MADDER disabled

Capture from final game with MADDER enabled

Capture from final game with MADDER enabled

Creative Workflow


Creative Workflow
IDEA
Coder
Sound Designer
GAME
DONE


Creative Workflow
IDEA
Sound Designer
GAME
DONE
Coder

Gun Tails

Gun Tails

- Experiment with existing data

Gun Tails

- Experiment with existing data
  - Wall raycast

Gun Tails

- Experiment with existing data
  - Wall raycast
  - Inside/Outside

Gun Tails

- Experiment with existing data
  - Wall raycast
  - Inside/Outside
  - Rounds Fired

Gun Tails

- Experiment with existing data
  - Wall raycast
  - Inside/Outside
  - Rounds Fired

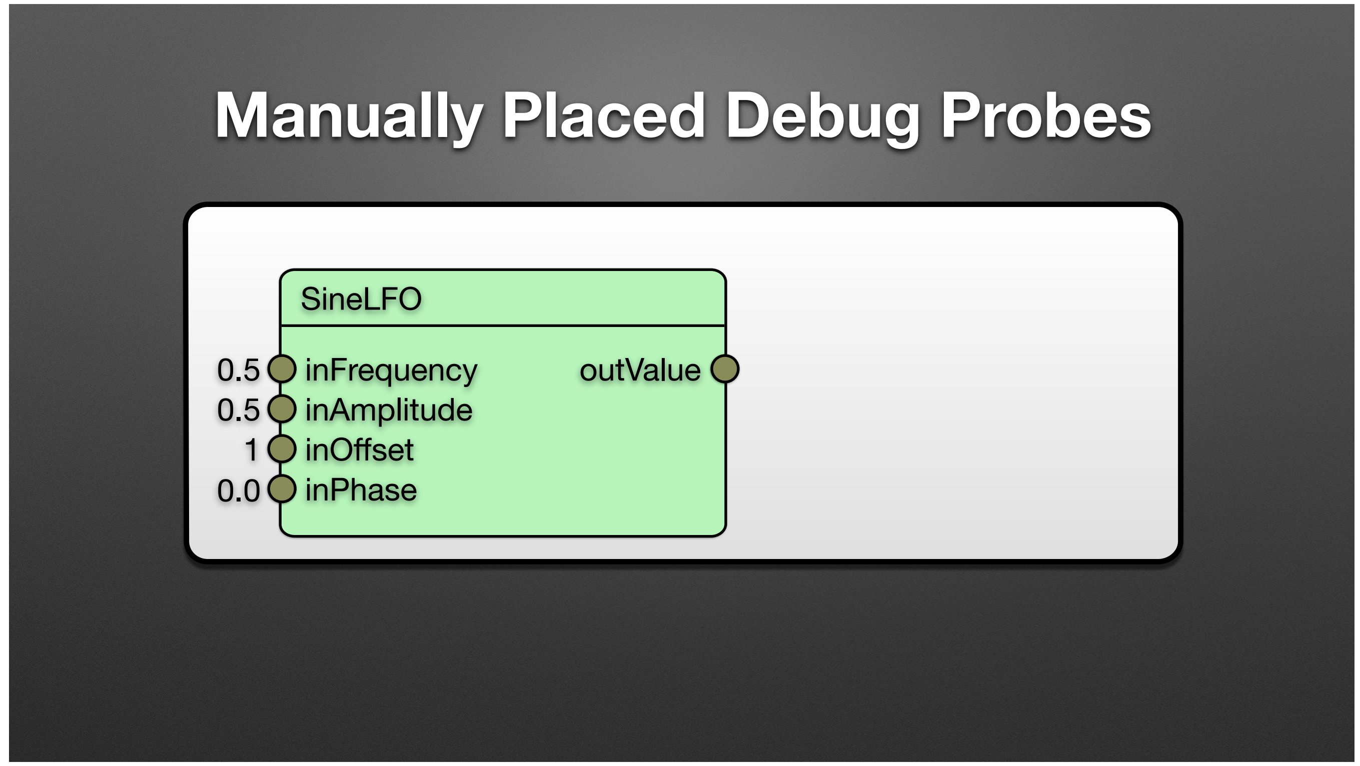
Gun Tails

- Experiment with existing data
  - Wall raycast
  - Inside/Outside
  - Rounds Fired

Gun Tails

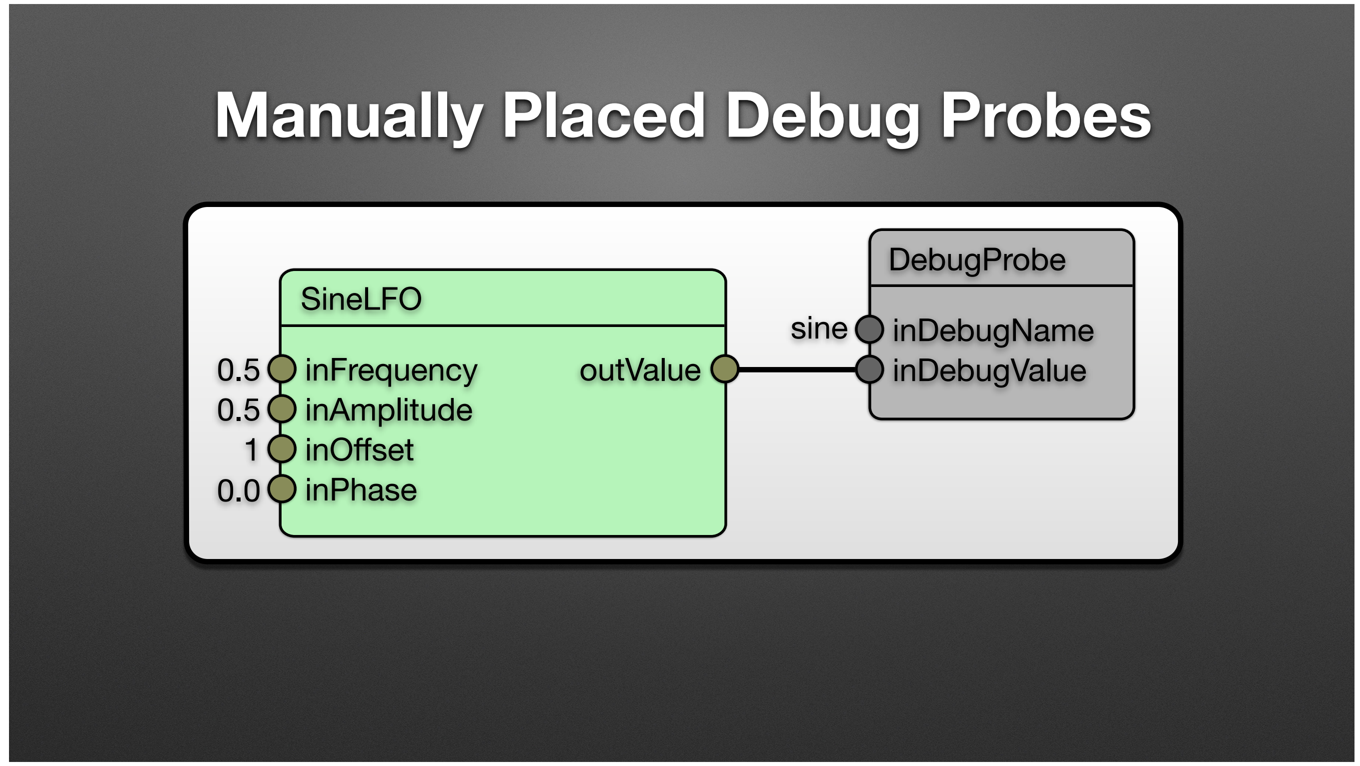
- Experiment with existing data
  - Wall raycast
  - Inside/Outside
  - Rounds Fired

What kind of **debugging support** did we have? Obviously a **sound designer is creating** much more **complex logic** now, and that means **bugs will creep in**. We need to be able to **find problems** such as incorrect behaviour **quickly**.



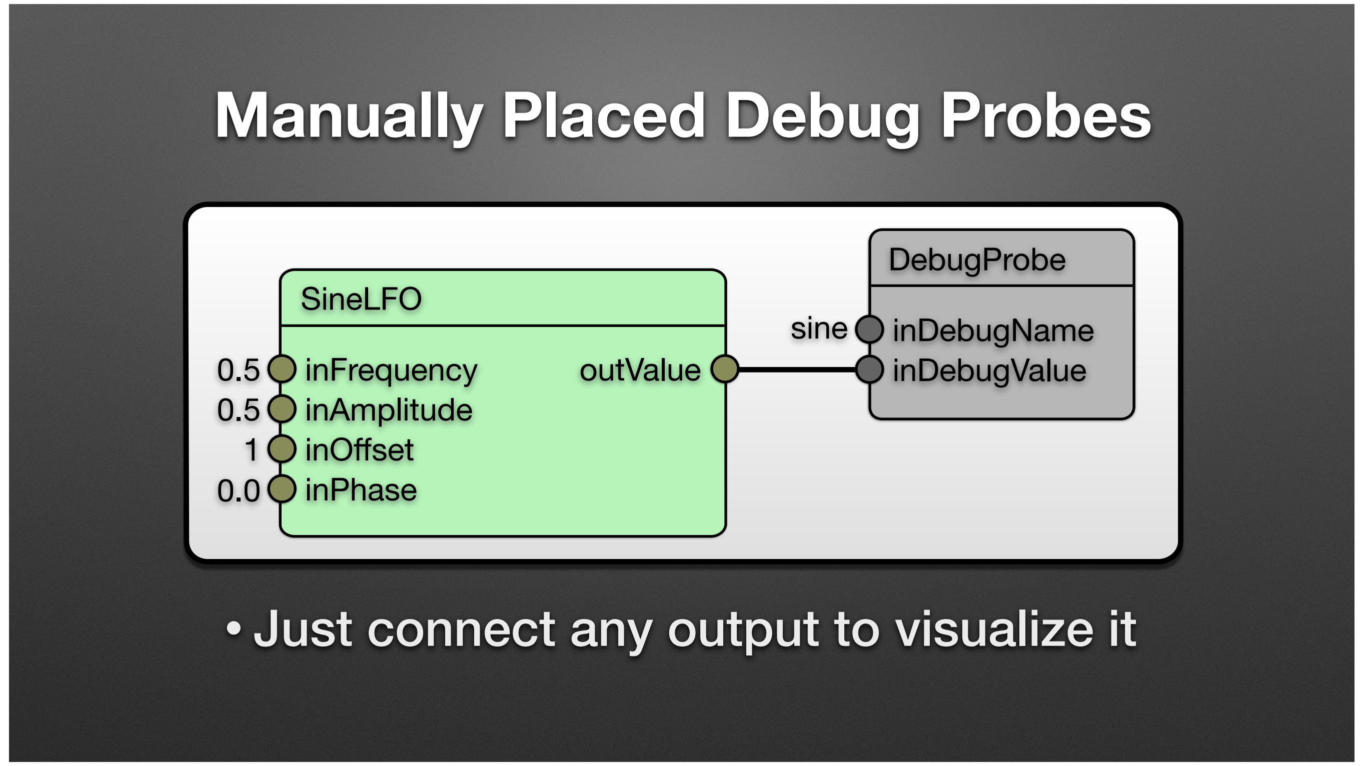
Anton: These are **placed by designers** into their sounds, to **query individual outputs** of nodes. The values are shown as a **curve on screen**, useful for debugging a single value and how it changes over time. **Not very useful for quickly changing things**, such as boolean events. Since the sound **graphs execute multiple times per game frame**, but the **debug** visualisation **is only drawn at the game frame rate**, a **quick change can be missed**.
In the final version of the game, these nodes are completely ignored.

Screenshot!



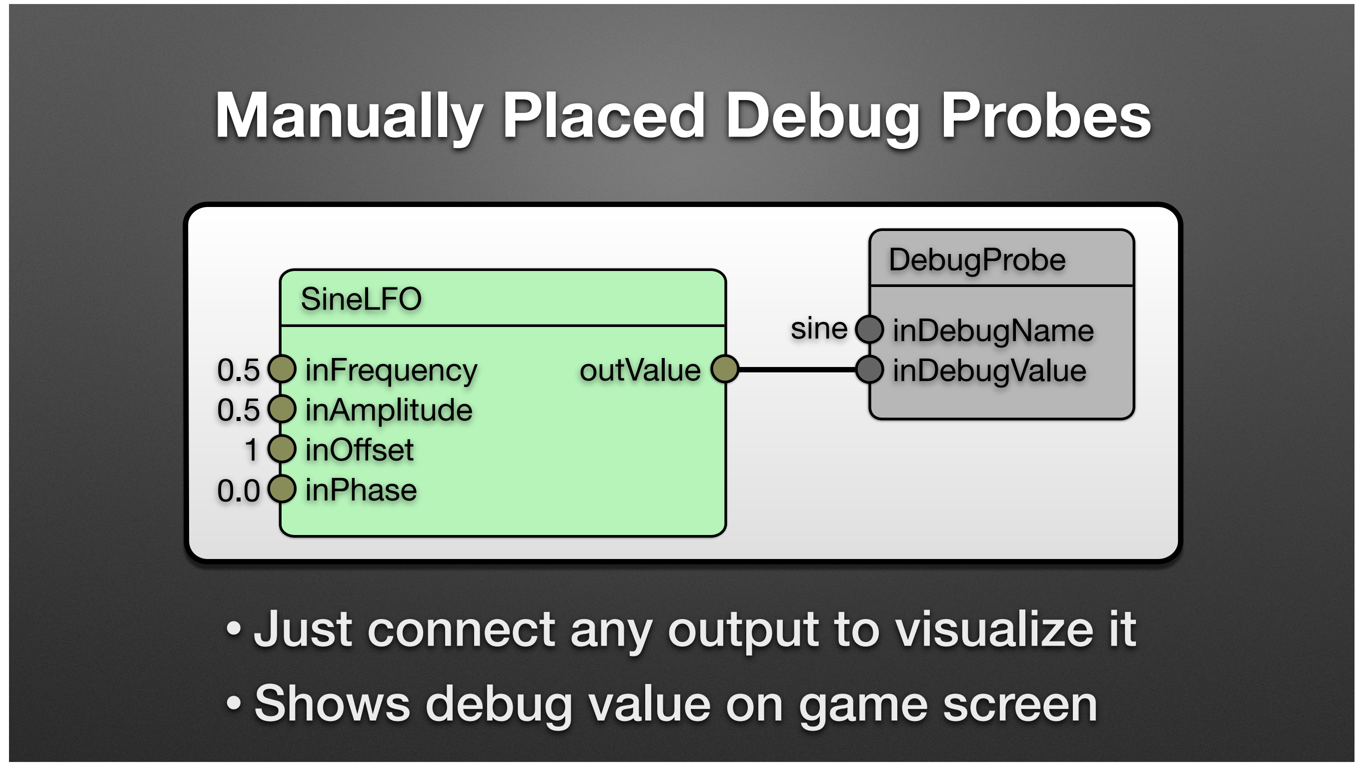
Anton: These are **placed by designers** into their sounds, to **query individual outputs** of nodes. The values are shown as a **curve on screen**, useful for debugging a single value and how it changes over time. **Not very useful for quickly changing things**, such as boolean events. Since the sound **graphs execute multiple times per game frame**, but the **debug** visualisation **is only drawn at the game frame rate**, a **quick change can be missed**.
In the final version of the game, these nodes are completely ignored.

Screenshot!



Anton: These are **placed by designers** into their sounds, to **query individual outputs** of nodes. The values are shown as a **curve on screen**, useful for debugging a single value and how it changes over time. **Not very useful for quickly changing things**, such as boolean events. Since the sound **graphs execute multiple times per game frame**, but the **debug** visualisation **is only drawn at the game frame rate**, a **quick change can be missed**.
In the final version of the game, these nodes are completely ignored.

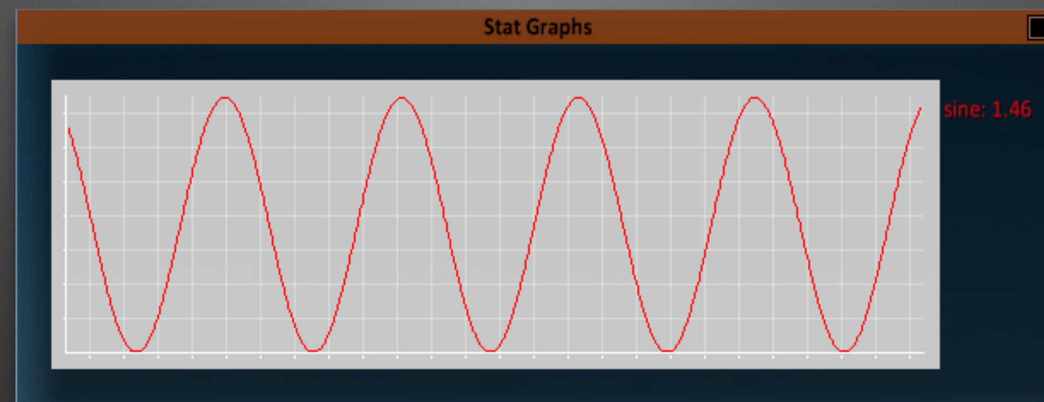
Screenshot!



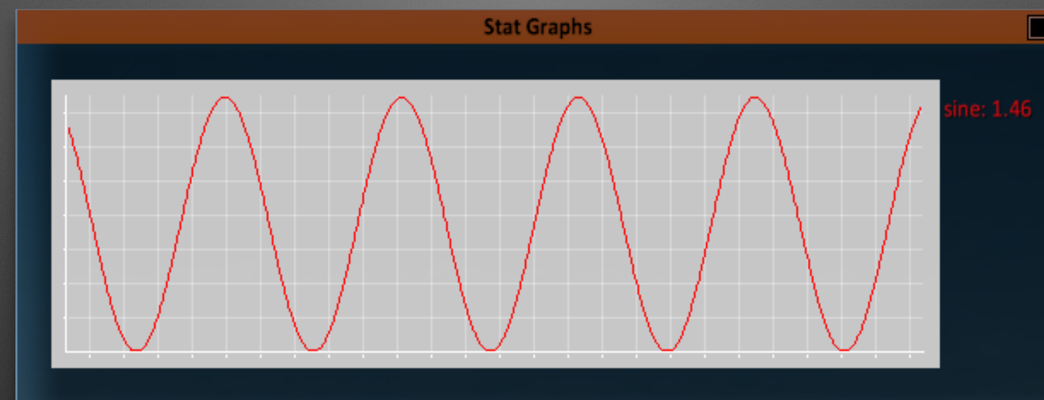
Anton: These are **placed by designers** into their sounds, to **query individual outputs** of nodes. The values are shown as a **curve on screen**, useful for debugging a single value and how it changes over time. **Not very useful for quickly changing things**, such as boolean events. Since the sound **graphs execute multiple times per game frame**, but the **debug** visualisation **is only drawn at the game frame rate**, a **quick change can be missed**.
In the final version of the game, these nodes are completely ignored.

Screenshot!

Manually Placed Debug Probes



Manually Placed Debug Probes



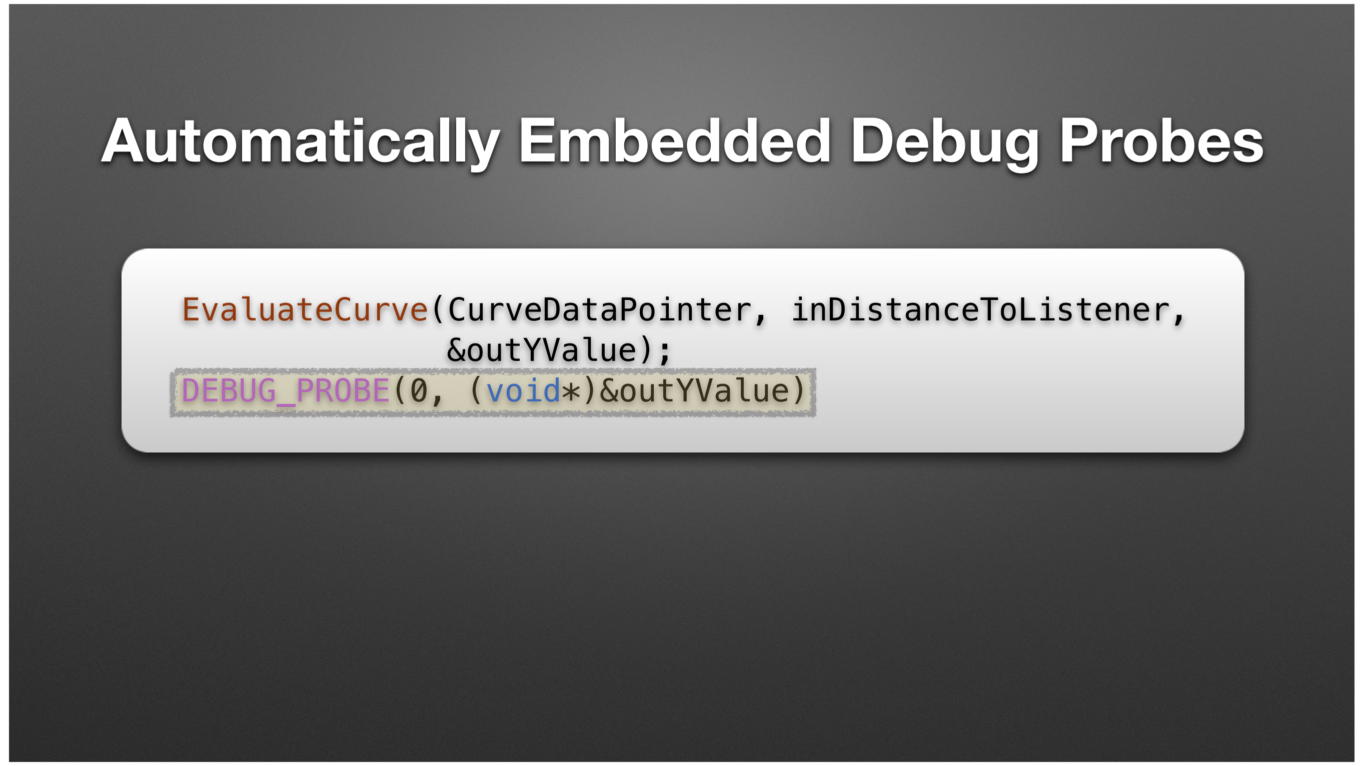
Can miss quick changes because game frame rate is lower than sound update rate!

Automatically Embedded Debug Probes

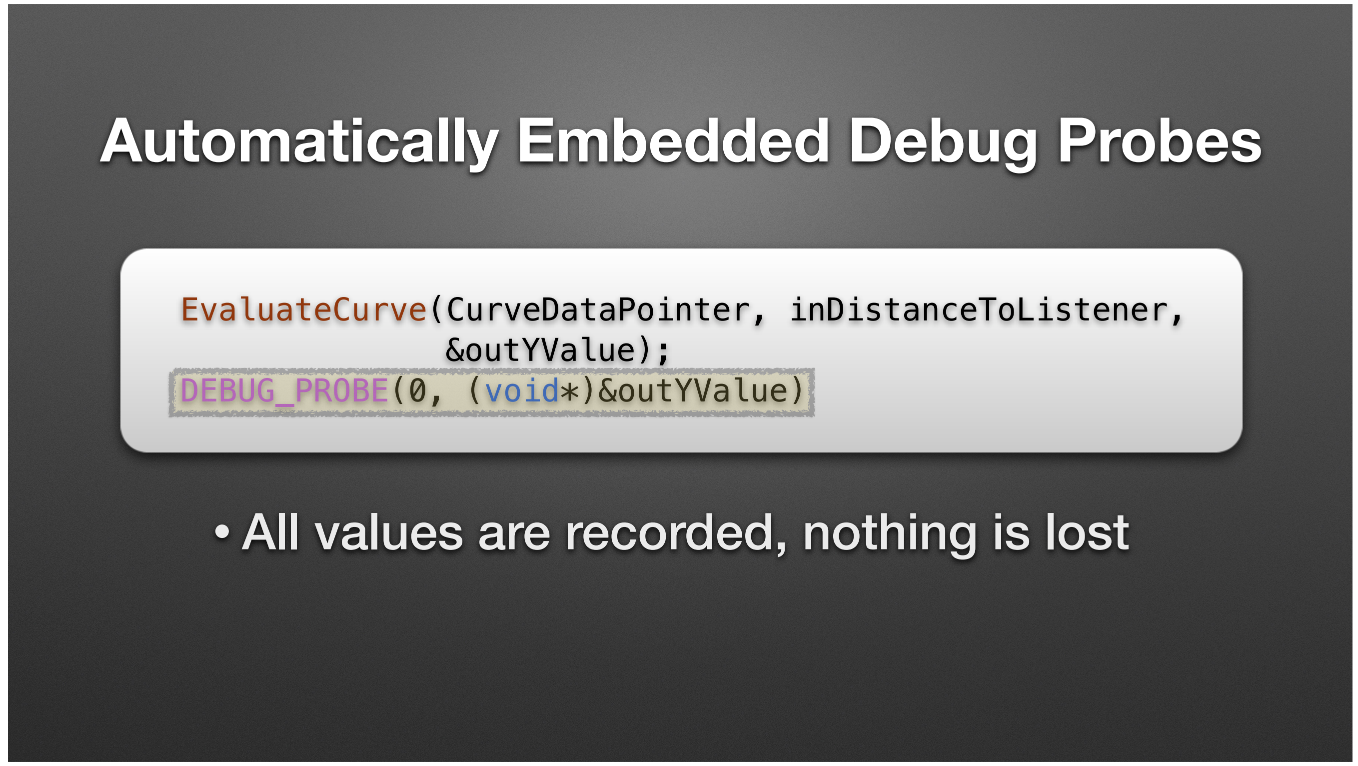
```
EvaluateCurve(CurveDataPointer, inDistanceToListener,
              &outYValue);
DEBUG_PROBE(0, (void*)&outYValue)
```

Andreas: Each graph is generated in **two versions**, one **without debugging** code (for final game) and one **with automatically generated debug probe code** for every node. We emit these **DEBUG_PROBE macros** into the graph functions, so we can easily **disable the debug support at compile time**.

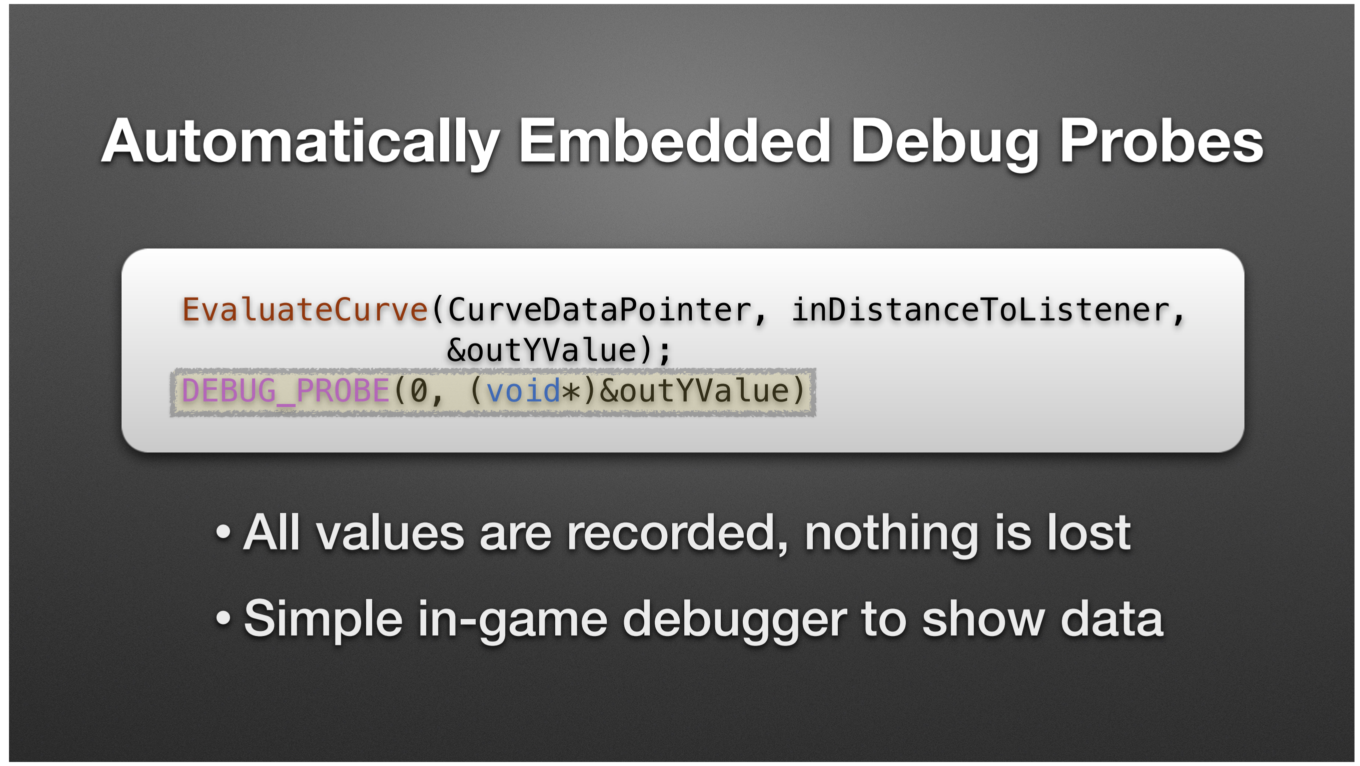
When **executing** a graph **with debugging** enabled, the **code** in the macro **collects the values of the inputs and outputs of every node** (and of the graph itself) and **passes them to the engine**, which stores the information.



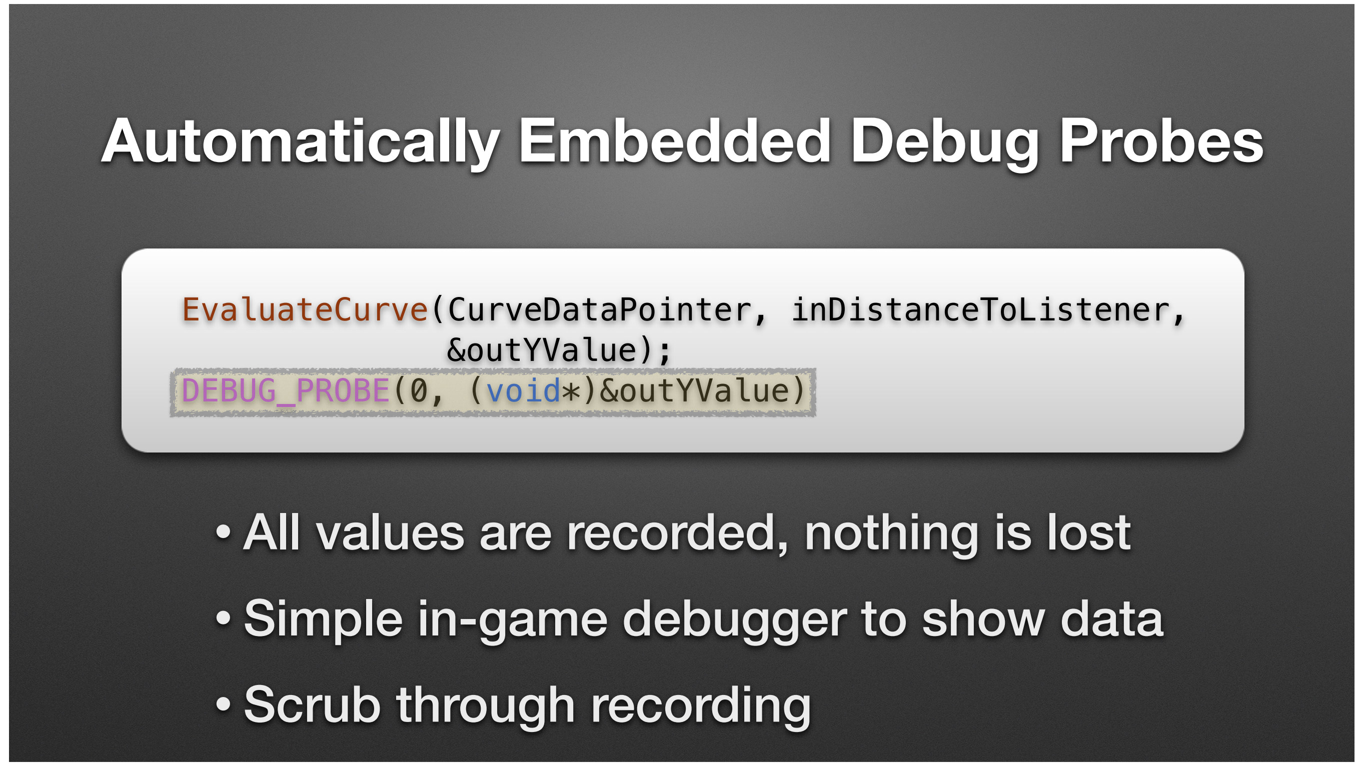
Andreas: Each graph is generated in **two versions**, one **without debugging** code (for final game) and one **with automatically generated debug probe code** for every node. We emit these **DEBUG_PROBE macros** into the graph functions, so we can easily **disable the debug support at compile time**.
When **executing** a graph **with debugging** enabled, the **code** in the macro **collects the values of the inputs and outputs of every node** (and of the graph itself) and **passes them to the engine**, which stores the information.



Andreas: Each graph is generated in **two versions**, one **without debugging** code (for final game) and one **with automatically generated debug probe code** for every node. We emit these **DEBUG_PROBE macros** into the graph functions, so we can easily **disable the debug support at compile time**.
When **executing** a graph **with debugging** enabled, the **code** in the macro **collects the values of the inputs and outputs of every node** (and of the graph itself) and **passes them to the engine**, which stores the information.



Andreas: Each graph is generated in **two versions**, one **without debugging** code (for final game) and one **with automatically generated debug probe code** for every node. We emit these **DEBUG_PROBE macros** into the graph functions, so we can easily **disable the debug support at compile time**.
When **executing** a graph **with debugging** enabled, the **code** in the macro **collects the values of the inputs and outputs of every node** (and of the graph itself) and **passes them to the engine**, which stores the information.

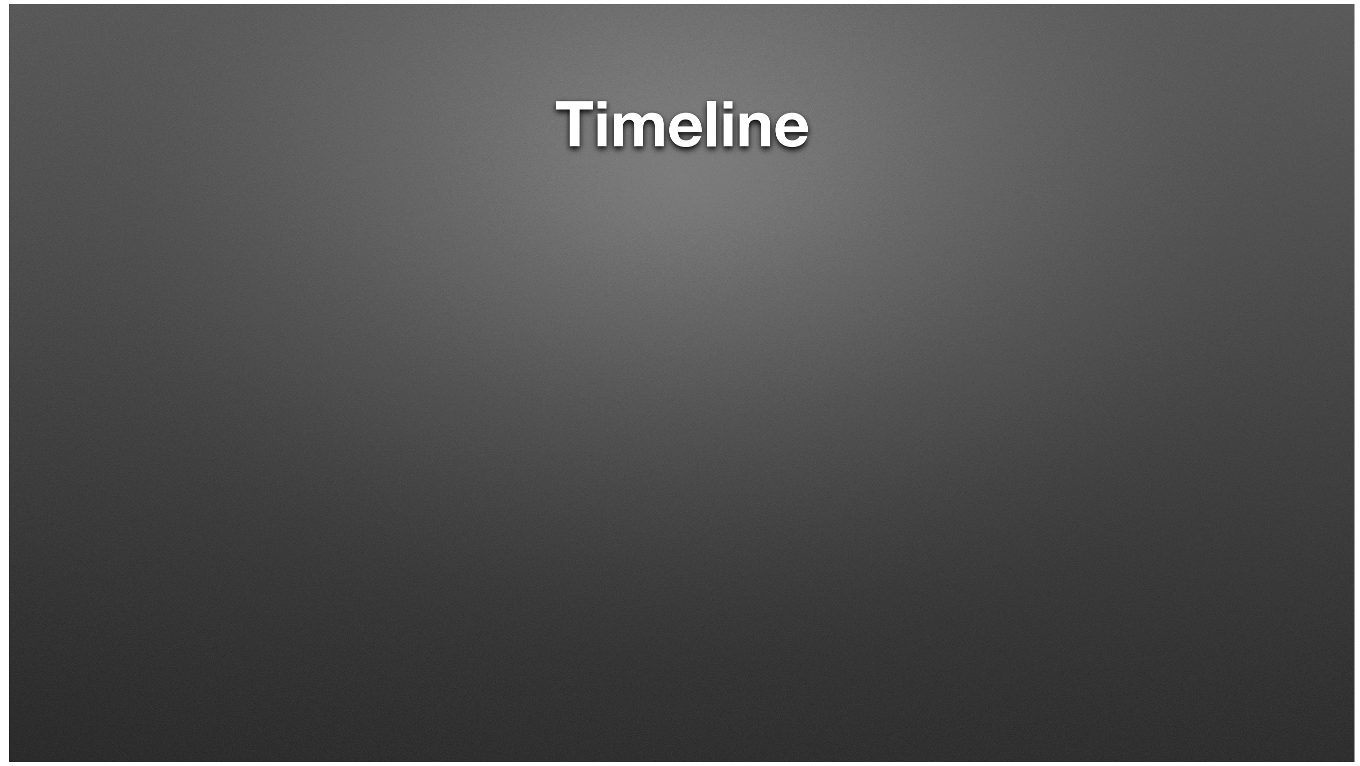


Andreas: Each graph is generated in **two versions**, one **without debugging** code (for final game) and one **with automatically generated debug probe code** for every node. We emit these **DEBUG_PROBE macros** into the graph functions, so we can easily **disable the debug support at compile time**.

When **executing** a graph **with debugging** enabled, the **code** in the macro **collects the values of the inputs and outputs of every node** (and of the graph itself) and **passes them to the engine**, which stores the information.

Post Mortem

So what when wrong and what went right?



Andreas: We started working on PS4 technology very early, many parts of the **PS4 hardware** were still **undefined** when we started. At one point we even expected the **hardware** to do more than decoding, i.e. we assumed that there might be a way of executing our own DSP code on it. We thought we might have to write **custom DSP code for that chip**. At that time there were **no middleware vendors disclosed** yet, so we just couldn't really talk to them about any of this.
We therefore had to **play it safe**, so we designed the **new sound system** around an **abstract synth**. We've created an initial prototype implementation, for use in our PC build. Later on we've switched to PS4 hardware.
Anton: become used to being alpha tester for hardware as a designer. became good audio tester. Other designers hardly noticed the transition from PC to PS4 hardware, as the synth was the same on both.

Timeline

February 2011	Pre-production begins, hardly any PS4 hardware details yet

Andreas: We started working on PS4 technology very early, many parts of the **PS4 hardware** were still **undefined** when we started. At one point we even expected the **hardware** to do more than decoding, i.e. we assumed that there might be a way of executing our own DSP code on it. We thought we might have to write **custom DSP code for that chip**. At that time there were **no middleware vendors disclosed** yet, so we just couldn't really talk to them about any of this.

We therefore had to **play it safe**, so we designed the **new sound system** around an **abstract synth**. We've created an initial prototype implementation, for use in our PC build. Later on we've switched to PS4 hardware.

Anton: become used to being alpha tester for hardware as a designer. became good audio tester. Other designers hardly noticed the transition from PC to PS4 hardware, as the synth was the same on both.

Timeline

February 2011	Pre-production begins, hardly any PS4 hardware details yet
November 2011	PC prototype of sound engine

Andreas: We started working on PS4 technology very early, many parts of the **PS4 hardware** were still **undefined** when we started. At one point we even expected the **hardware** to do more than decoding, i.e. we assumed that there might be a way of executing our own DSP code on it. We thought we might have to write **custom DSP code for that chip**. At that time there were **no middleware vendors disclosed** yet, so we just couldn't really talk to them about any of this.

We therefore had to **play it safe**, so we designed the **new sound system** around an **abstract synth**. We've created an initial prototype implementation, for use in our PC build. Later on we've switched to PS4 hardware.

Anton: become used to being alpha tester for hardware as a designer. became good audio tester. Other designers hardly noticed the transition from PC to PS4 hardware, as the synth was the same on both.

Timeline

February 2011	Pre-production begins, hardly any PS4 hardware details yet
November 2011	PC prototype of sound engine
August 2012	Moved over to PS4 hardware

Andreas: We started working on PS4 technology very early, many parts of the **PS4 hardware** were still **undefined** when we started. At one point we even expected the **hardware** to do more than decoding, i.e. we assumed that there might be a way of executing our own DSP code on it. We thought we might have to write **custom DSP code for that chip**. At that time there were **no middleware vendors disclosed** yet, so we just couldn't really talk to them about any of this.

We therefore had to **play it safe**, so we designed the **new sound system** around an **abstract synth**. We've created an initial prototype implementation, for use in our PC build. Later on we've switched to PS4 hardware.

Anton: become used to being alpha tester for hardware as a designer. became good audio tester. Other designers hardly noticed the transition from PC to PS4 hardware, as the synth was the same on both.

Timeline

February 2011	Pre-production begins, hardly any PS4 hardware details yet
November 2011	PC prototype of sound engine
August 2012	Moved over to PS4 hardware
February 2013	PS4 announcement demo (still using software codecs)

Andreas: We started working on PS4 technology very early, many parts of the **PS4 hardware** were still **undefined** when we started. At one point we even expected the **hardware** to do more than decoding, i.e. we assumed that there might be a way of executing our own DSP code on it. We thought we might have to write **custom DSP code for that chip**. At that time there were **no middleware vendors disclosed** yet, so we just couldn't really talk to them about any of this.

We therefore had to **play it safe**, so we designed the **new sound system** around an **abstract synth**. We've created an initial prototype implementation, for use in our PC build. Later on we've switched to PS4 hardware.

Anton: become used to being alpha tester for hardware as a designer. became good audio tester. Other designers hardly noticed the transition from PC to PS4 hardware, as the synth was the same on both.

Timeline

February 2011	Pre-production begins, hardly any PS4 hardware details yet
November 2011	PC prototype of sound engine
August 2012	Moved over to PS4 hardware
February 2013	PS4 announcement demo (still using software codecs)
June 2013	E3 demo (now using ACP hardware decoding)

Andreas: We started working on PS4 technology very early, many parts of the **PS4 hardware** were still **undefined** when we started. At one point we even expected the **hardware** to do more than decoding, i.e. we assumed that there might be a way of executing our own DSP code on it. We thought we might have to write **custom DSP code for that chip**. At that time there were **no middleware vendors disclosed** yet, so we just couldn't really talk to them about any of this.

We therefore had to **play it safe**, so we designed the **new sound system** around an **abstract synth**. We've created an initial prototype implementation, for use in our PC build. Later on we've switched to PS4 hardware.

Anton: become used to being alpha tester for hardware as a designer. became good audio tester. Other designers hardly noticed the transition from PC to PS4 hardware, as the synth was the same on both.

Timeline

February 2011	Pre-production begins, hardly any PS4 hardware details yet
November 2011	PC prototype of sound engine
August 2012	Moved over to PS4 hardware
February 2013	PS4 announcement demo (still using software codecs)
June 2013	E3 demo (now using ACP hardware decoding)
November 2013	Shipped as launch title

Andreas: We started working on PS4 technology very early, many parts of the **PS4 hardware** were still **undefined** when we started. At one point we even expected the **hardware** to do more than decoding, i.e. we assumed that there might be a way of executing our own DSP code on it. We thought we might have to write **custom DSP code for that chip**. At that time there were **no middleware vendors disclosed** yet, so we just couldn't really talk to them about any of this.

We therefore had to **play it safe**, so we designed the **new sound system** around an **abstract synth**. We've created an initial prototype implementation, for use in our PC build. Later on we've switched to PS4 hardware.

Anton: become used to being alpha tester for hardware as a designer. became good audio tester. Other designers hardly noticed the transition from PC to PS4 hardware, as the synth was the same on both.

Timeline

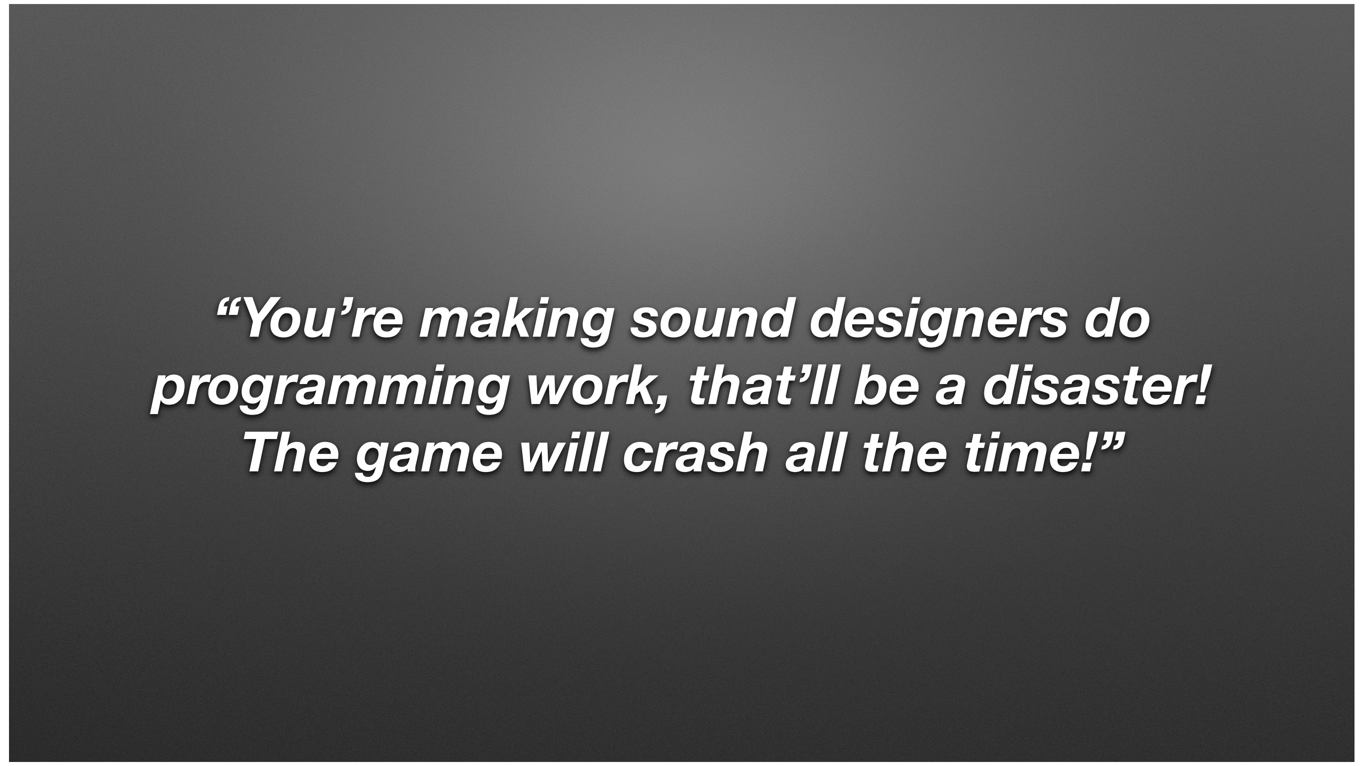
February 2011	Pre-production begins, hardly any PS4 hardware details yet
November 2011	PC prototype of sound engine
August 2012	Moved over to PS4 hardware
February 2013	PS4 announcement demo (still using software codecs)
June 2013	E3 demo (now using ACP hardware decoding)
November 2013	Shipped as launch title

New system was up and running within 6 months!

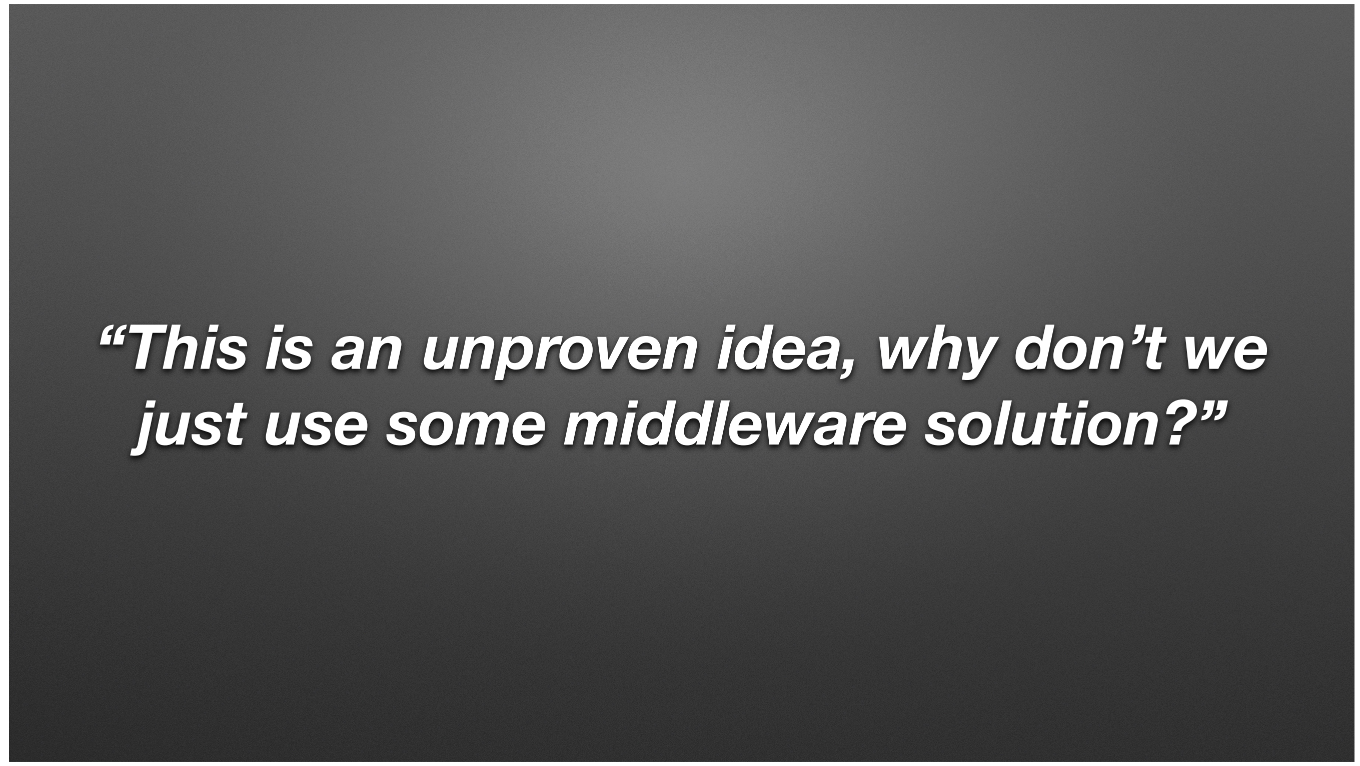
Andreas: We started working on PS4 technology very early, many parts of the **PS4 hardware** were still **undefined** when we started. At one point we even expected the **hardware** to do more than decoding, i.e. we assumed that there might be a way of executing our own DSP code on it. We thought we might have to write **custom DSP code for that chip**. At that time there were **no middleware vendors disclosed** yet, so we just couldn't really talk to them about any of this.

We therefore had to **play it safe**, so we designed the **new sound system** around an **abstract synth**. We've created an initial prototype implementation, for use in our PC build. Later on we've switched to PS4 hardware.

Anton: become used to being alpha tester for hardware as a designer. became good audio tester. Other designers hardly noticed the transition from PC to PS4 hardware, as the synth was the same on both.



This didn't really happen. We had **a few buggy nodes**, but in general, **things worked** out fine. What we did is to make sure that if **new nodes are peer-reviewed** like any other code, especially **if** it's **written by** a **non-programmer**. Also since nodes are so simple and limited in their scope, it's hard to make something that truly breaks the game.



There were **doubts** that we can **create new tech and a toolset** with a user friendly workflow in time. The safer thing would’ve been to **license a middleware** solution, but at the time we had to make this decision, **no 3rd parties** were **disclosed** about PS4 yet, and we couldn’t be sure if they will properly support the PS4 audio hardware.
Also we were keen on having a **solution** that’s **integrated with** our normal **asset pipeline**, which allowed us to **share the tech** built for audio with other disciplines.
Sound designers using the **same workflow** as artists and game designers is a **benefit that middleware can't give us**. New **nodes** created **by** e.g. **game programmers** are immediately **useful for sound designers**, and vice versa.
All of these reasons led us to push forward with our own tech, to make something that really fits the game and our way of working.

Anton: Also we wouldn’t have been able to do the kind of deep tool integration that we have now.

Used by other Disciplines

The fact that the graph system was available in the editor that the whole studio is using meant that it was directly available to all other disciplines.

Used by other Disciplines

- Graph system became popular

The fact that the graph system was available in the editor that the whole studio is using meant that it was directly available to all other disciplines.

Used by other Disciplines

- Graph system became popular
- Used for procedural rigging

The fact that the graph system was available in the editor that the whole studio is using meant that it was directly available to all other disciplines.

Used by other Disciplines

- Graph system became popular
- Used for procedural rigging
- Used for gameplay behavior

The fact that the graph system was available in the editor that the whole studio is using meant that it was directly available to all other disciplines.

Used by other Disciplines

- Graph system became popular
- Used for procedural rigging
- Used for gameplay behavior
- More nodes were added

The fact that the graph system was available in the editor that the whole studio is using meant that it was directly available to all other disciplines.

Used by other Disciplines

- Graph system became popular
- Used for procedural rigging
- Used for gameplay behavior
- More nodes were added
- Bugs were fixed quicker

The fact that the graph system was available in the editor that the whole studio is using meant that it was directly available to all other disciplines.

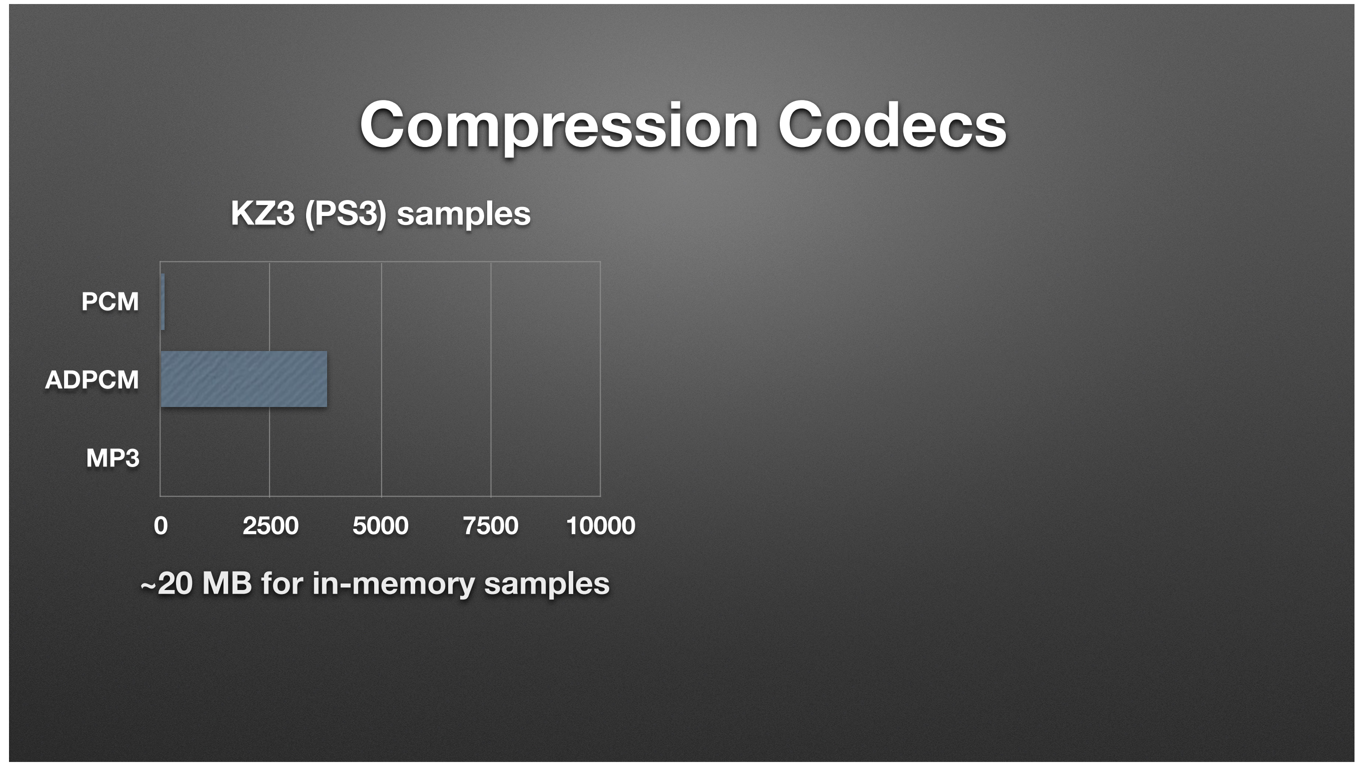
Compression Codecs

A little bit more information to show were we came from:

On PS3 **most** sounds were **ADPCM**, a **few** were **PCM**, only **streaming** sounds used **MP3** with various bit rates. We had about **20 MB budgeted for in-memory samples**.

This generation we ended up using a combination of **PCM** and **MP3** for in-memory sounds, with a total of about **300MB of memory** used at run-time, that's roughly 16x as much as on PS3, but still the same percentage of the whole memory **(roughly 4%)**. We didn't use any ADPCM samples anymore (good riddance). The split between PCM and MP3 is roughly 50:50. We used PCM for anything that needs to loop or seek with sample accuracy, and **MP3 for larger sounds** that **don’t require precise timing**, such as the tails of guns, for example. Obviously we relied on the audio coprocessor of the PS4 to decode our MP3 data.

We’ve used ATRAC9 for surround streams, but those are not in-memory.

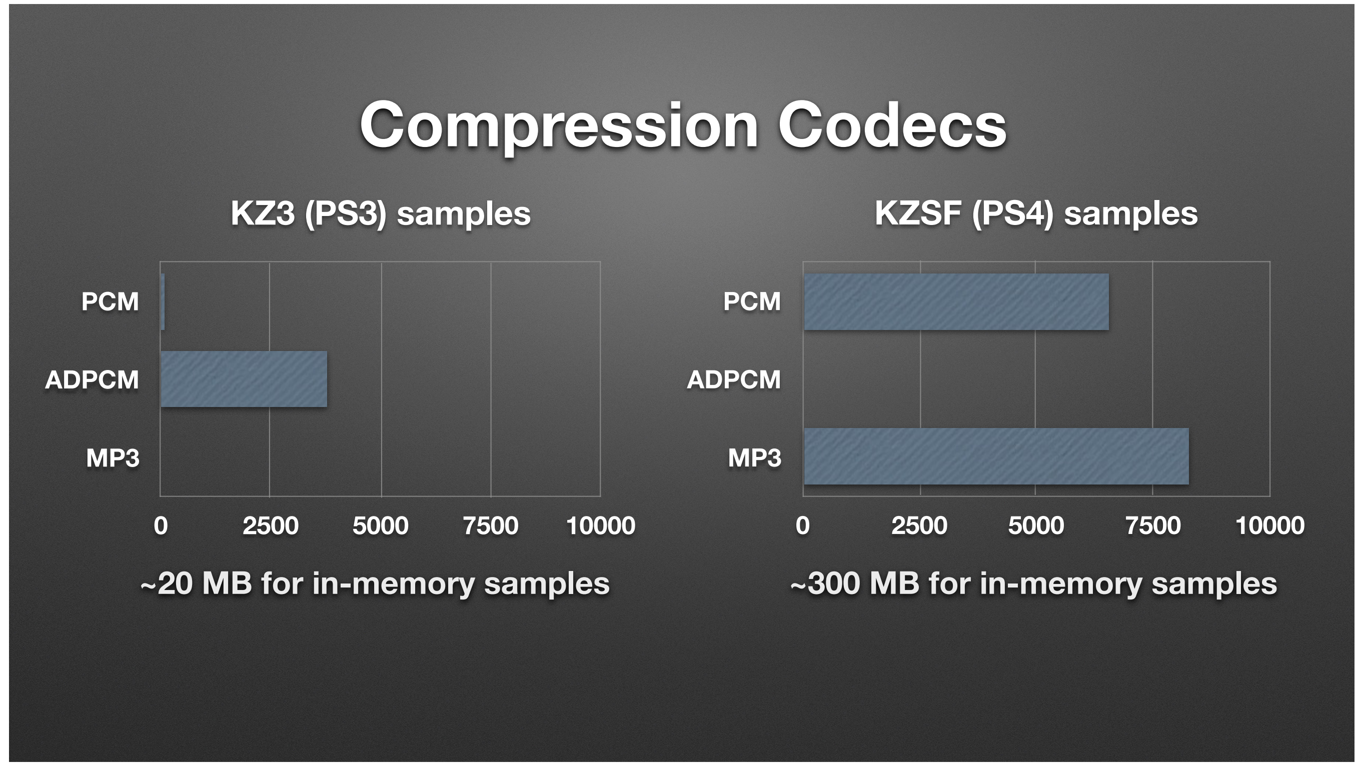


A little bit more information to show were we came from:

On PS3 **most** sounds were **ADPCM**, a **few** were **PCM**, only **streaming** sounds used **MP3** with various bit rates. We had about **20 MB budgeted for in-memory samples**.

This generation we ended up using a combination of **PCM** and **MP3** for in-memory sounds, with a total of about **300MB of memory** used at run-time, that's roughly 16x as much as on PS3, but still the same percentage of the whole memory **(roughly 4%)**. We didn't use any ADPCM samples anymore (good riddance). The split between PCM and MP3 is roughly 50:50. We used PCM for anything that needs to loop or seek with sample accuracy, and **MP3 for larger sounds** that **don't require precise timing**, such as the tails of guns, for example. Obviously we relied on the audio coprocessor of the PS4 to decode our MP3 data.

We've used ATRAC9 for surround streams, but those are not in-memory.

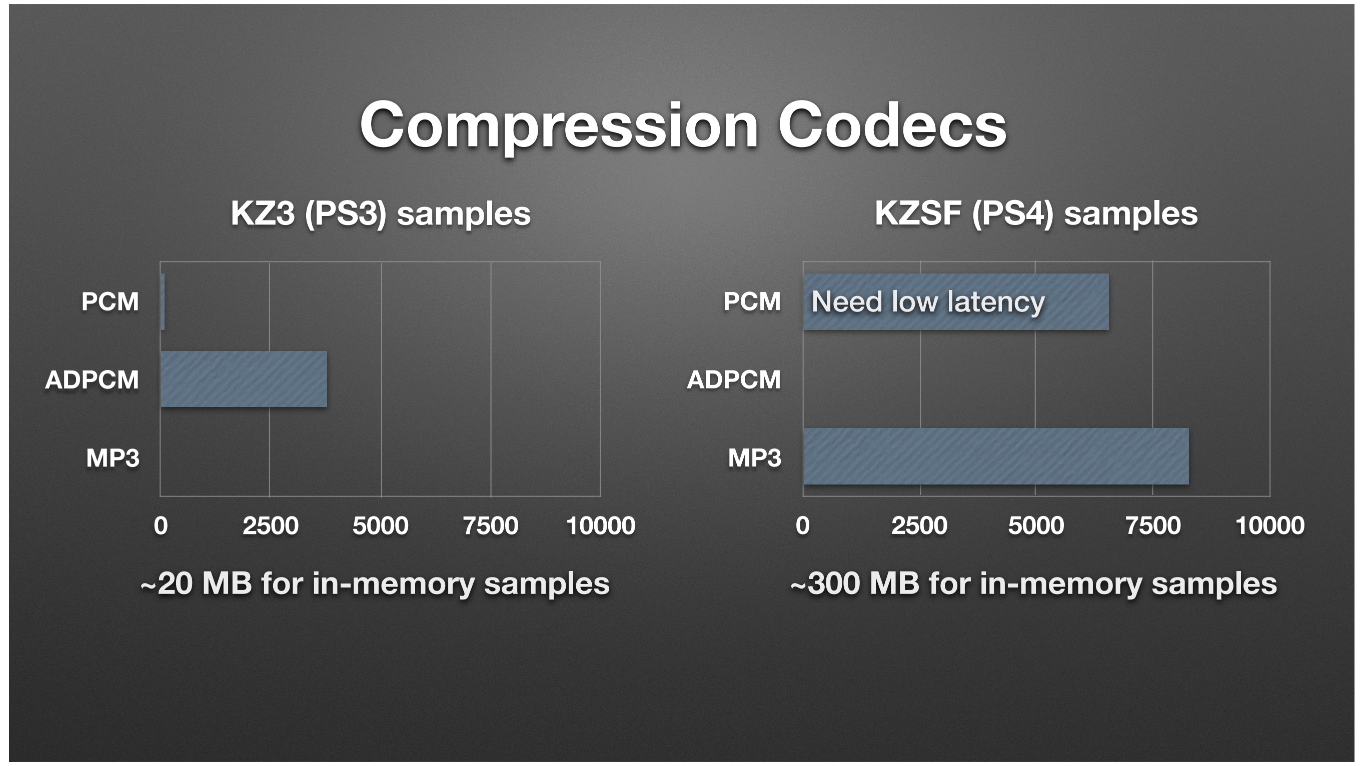


A little bit more information to show were we came from:

On PS3 **most** sounds were **ADPCM**, a **few** were **PCM**, only **streaming** sounds used **MP3** with various bit rates. We had about **20 MB budgeted for in-memory samples**.

This generation we ended up using a combination of **PCM** and **MP3** for in-memory sounds, with a total of about **300MB of memory** used at run-time, that's roughly 16x as much as on PS3, but still the same percentage of the whole memory **(roughly 4%)**. We didn't use any ADPCM samples anymore (good riddance). The split between PCM and MP3 is roughly 50:50. We used PCM for anything that needs to loop or seek with sample accuracy, and **MP3 for larger sounds** that **don't require precise timing**, such as the tails of guns, for example. Obviously we relied on the audio coprocessor of the PS4 to decode our MP3 data.

We've used ATRAC9 for surround streams, but those are not in-memory.

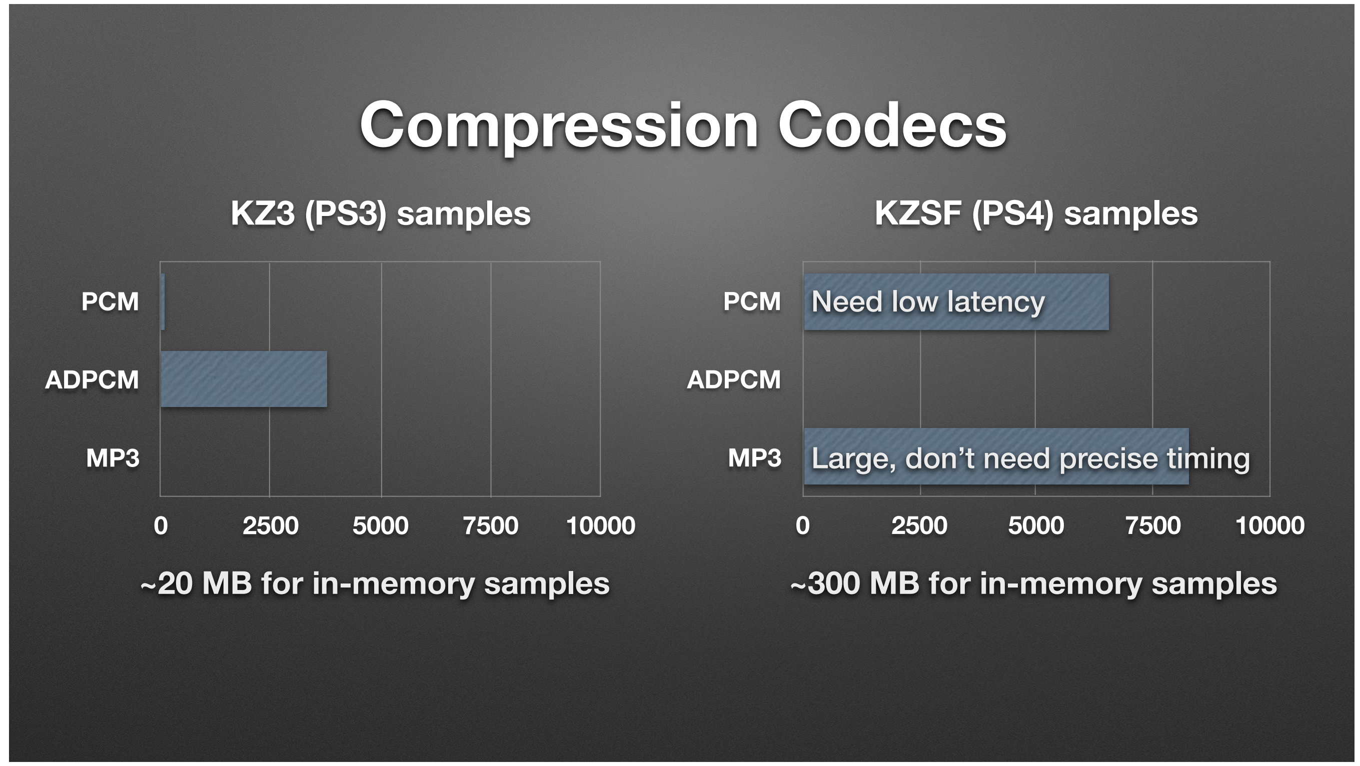


A little bit more information to show were we came from:

On PS3 **most** sounds were **ADPCM**, a **few** were **PCM**, only **streaming** sounds used **MP3** with various bit rates. We had about **20 MB budgeted for in-memory samples**.

This generation we ended up using a combination of **PCM** and **MP3** for in-memory sounds, with a total of about **300MB of memory** used at run-time, that's roughly 16x as much as on PS3, but still the same percentage of the whole memory **(roughly 4%)**. We didn't use any ADPCM samples anymore (good riddance). The split between PCM and MP3 is roughly 50:50. We used PCM for anything that needs to loop or seek with sample accuracy, and **MP3 for larger sounds** that **don't require precise timing**, such as the tails of guns, for example. Obviously we relied on the audio coprocessor of the PS4 to decode our MP3 data.

We've used ATRAC9 for surround streams, but those are not in-memory.

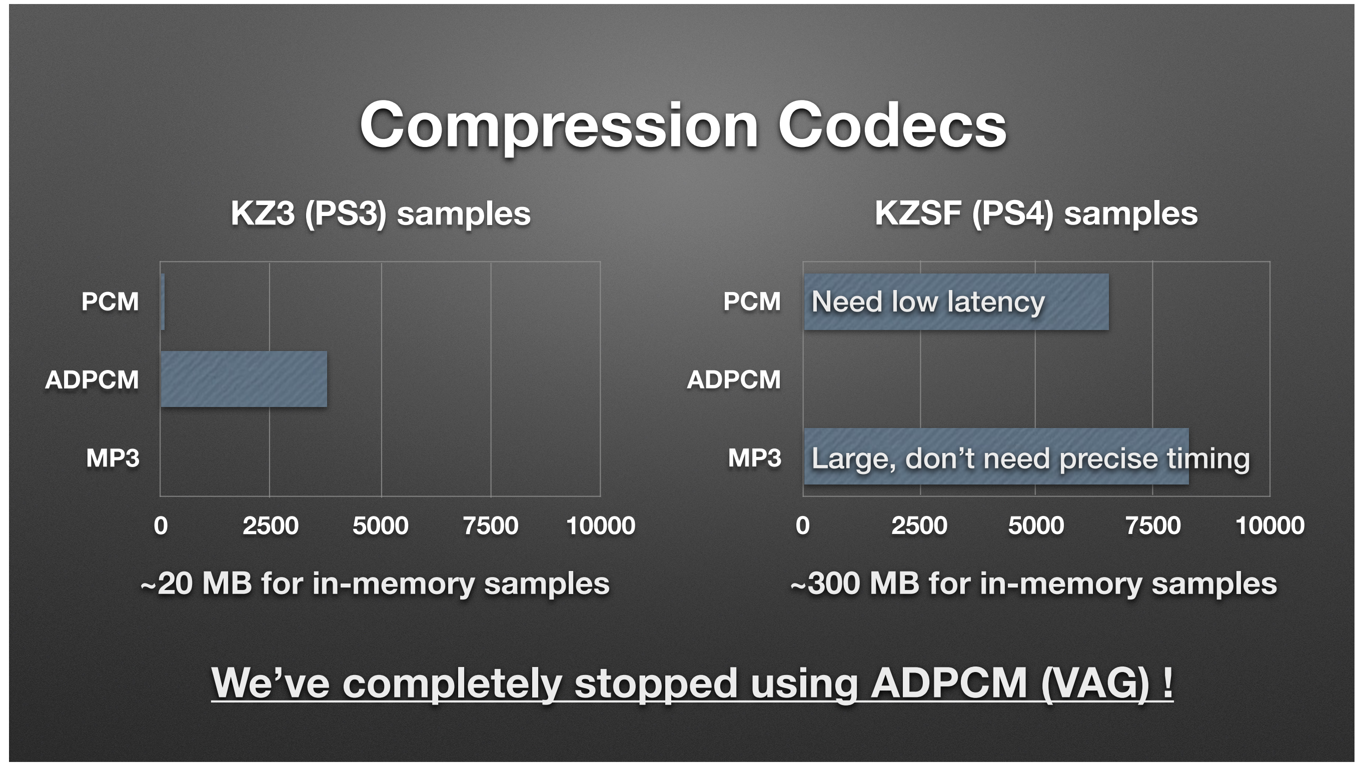


A little bit more information to show were we came from:

On PS3 **most** sounds were **ADPCM**, a **few** were **PCM**, only **streaming** sounds used **MP3** with various bit rates. We had about **20 MB budgeted for in-memory samples**.

This generation we ended up using a combination of **PCM** and **MP3** for in-memory sounds, with a total of about **300MB of memory** used at run-time, that's roughly 16x as much as on PS3, but still the same percentage of the whole memory **(roughly 4%)**. We didn't use any ADPCM samples anymore (good riddance). The split between PCM and MP3 is roughly 50:50. We used PCM for anything that needs to loop or seek with sample accuracy, and **MP3 for larger sounds** that **don’t require precise timing**, such as the tails of guns, for example. Obviously we relied on the audio coprocessor of the PS4 to decode our MP3 data.

We’ve used ATRAC9 for surround streams, but those are not in-memory.



A little bit more information to show were we came from:

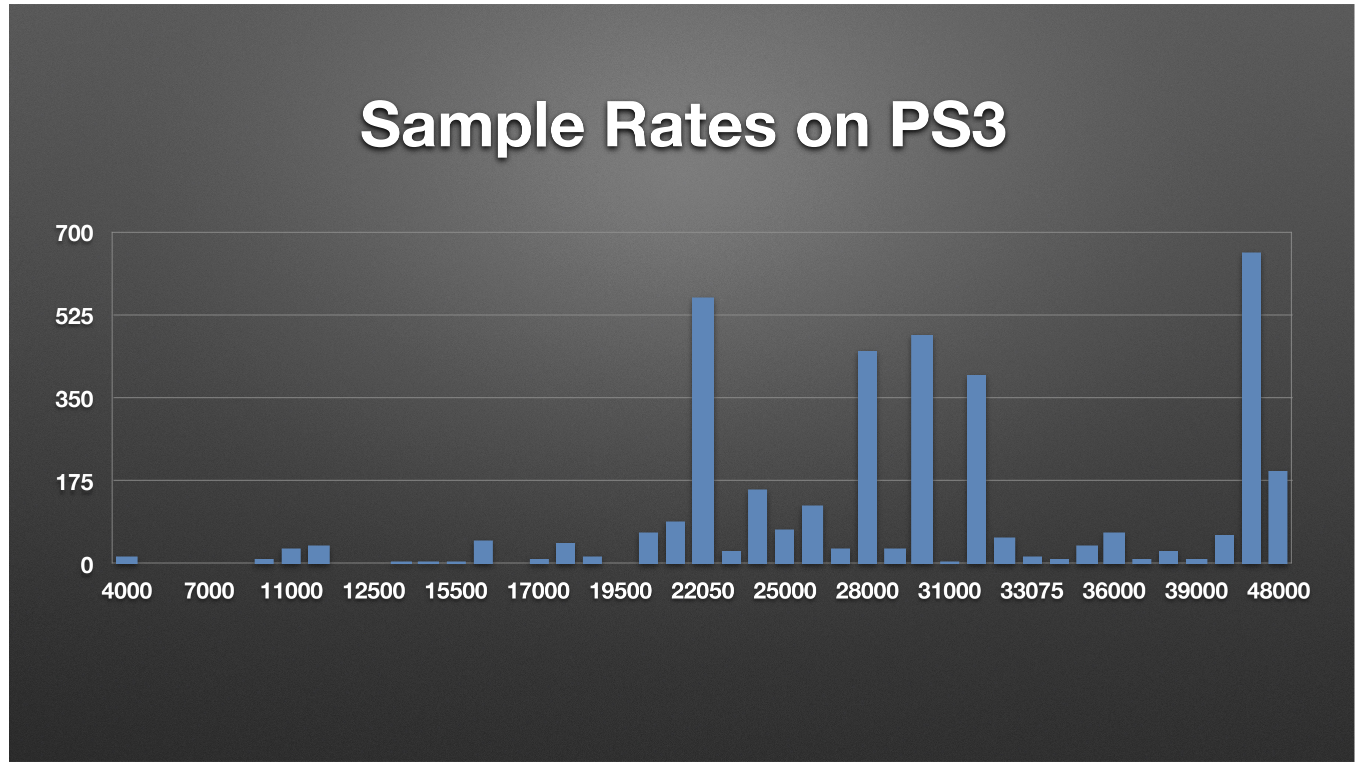
On PS3 **most** sounds were **ADPCM**, a **few** were **PCM**, only **streaming** sounds used **MP3** with various bit rates. We had about **20 MB budgeted for in-memory samples**.

This generation we ended up using a combination of **PCM** and **MP3** for in-memory sounds, with a total of about **300MB of memory** used at run-time, that's roughly 16x as much as on PS3, but still the same percentage of the whole memory **(roughly 4%)**. We didn't use any ADPCM samples anymore (good riddance). The split between PCM and MP3 is roughly 50:50. We used PCM for anything that needs to loop or seek with sample accuracy, and **MP3 for larger sounds** that **don’t require precise timing**, such as the tails of guns, for example. Obviously we relied on the audio coprocessor of the PS4 to decode our MP3 data.

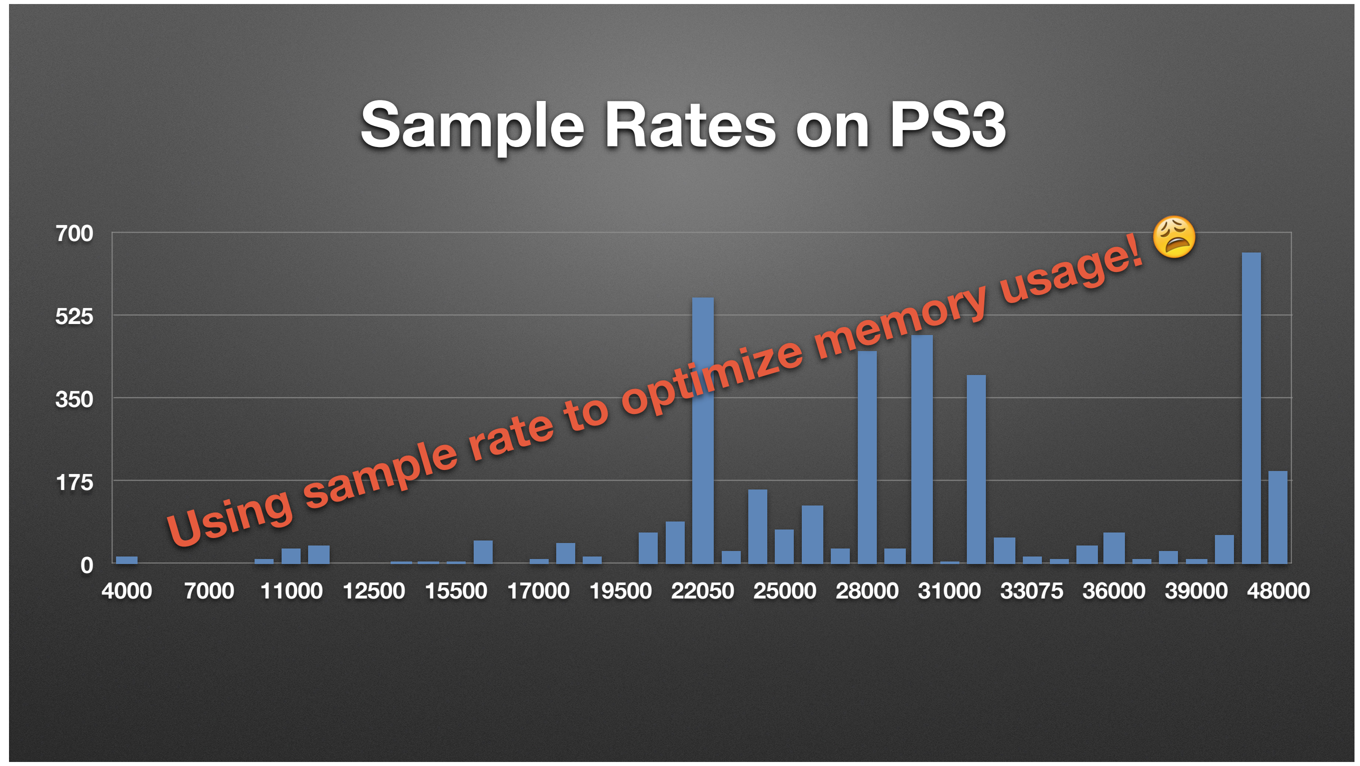
We’ve used ATRAC9 for surround streams, but those are not in-memory.

Sample Rates on PS3

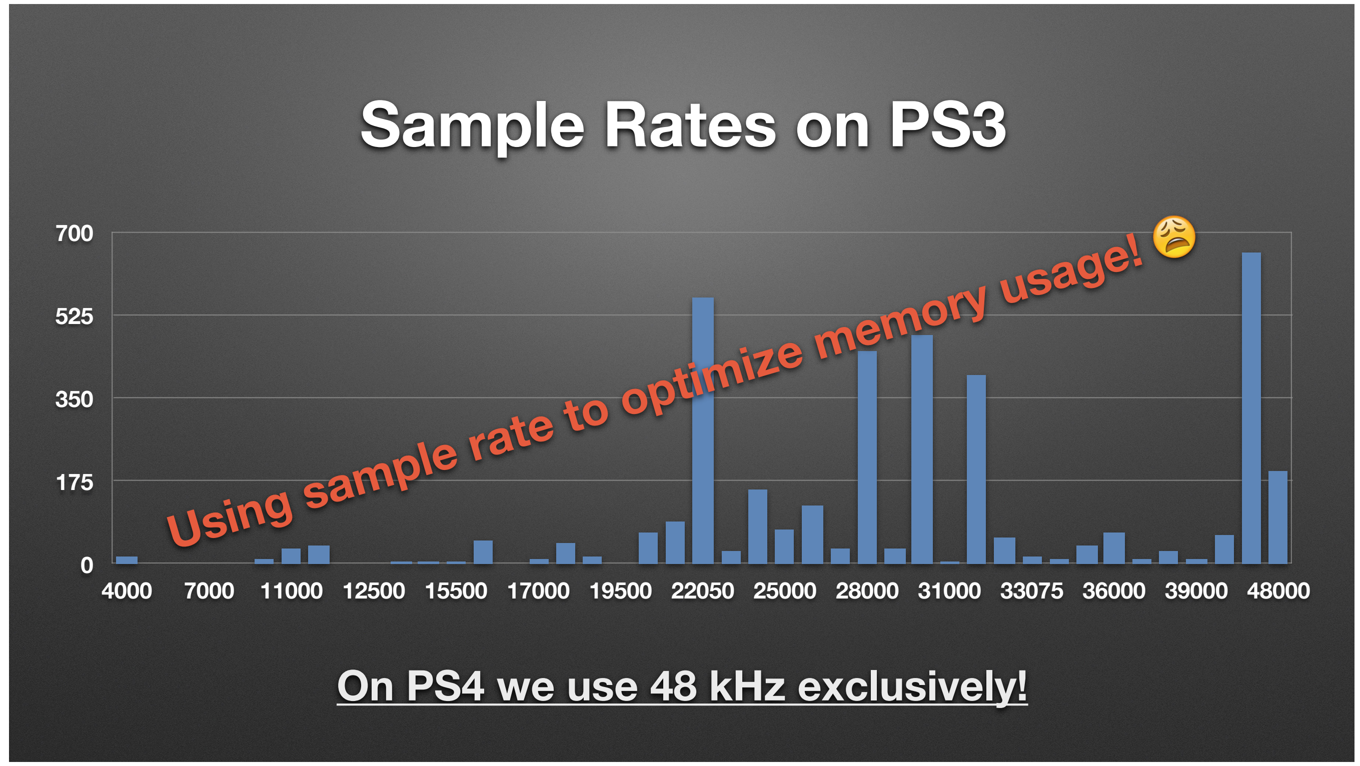
We had a wide **variety of sample rates** on the PS3, because it was the main parameter we used to optimize memory usage of sounds. This led to some **very low quality samples** in the game.
On the **PS4** we **only** used samples with **48kHz** rate. The main way to optimize was to switch them to **MP3 encoding** and adjust the bit rate.



We had a wide **variety of sample rates** on the PS3, because it was the main parameter we used to optimize memory usage of sounds. This led to some **very low quality samples** in the game.
On the **PS4** we **only** used samples with **48kHz** rate. The main way to optimize was to switch them to **MP3 encoding** and adjust the bit rate.



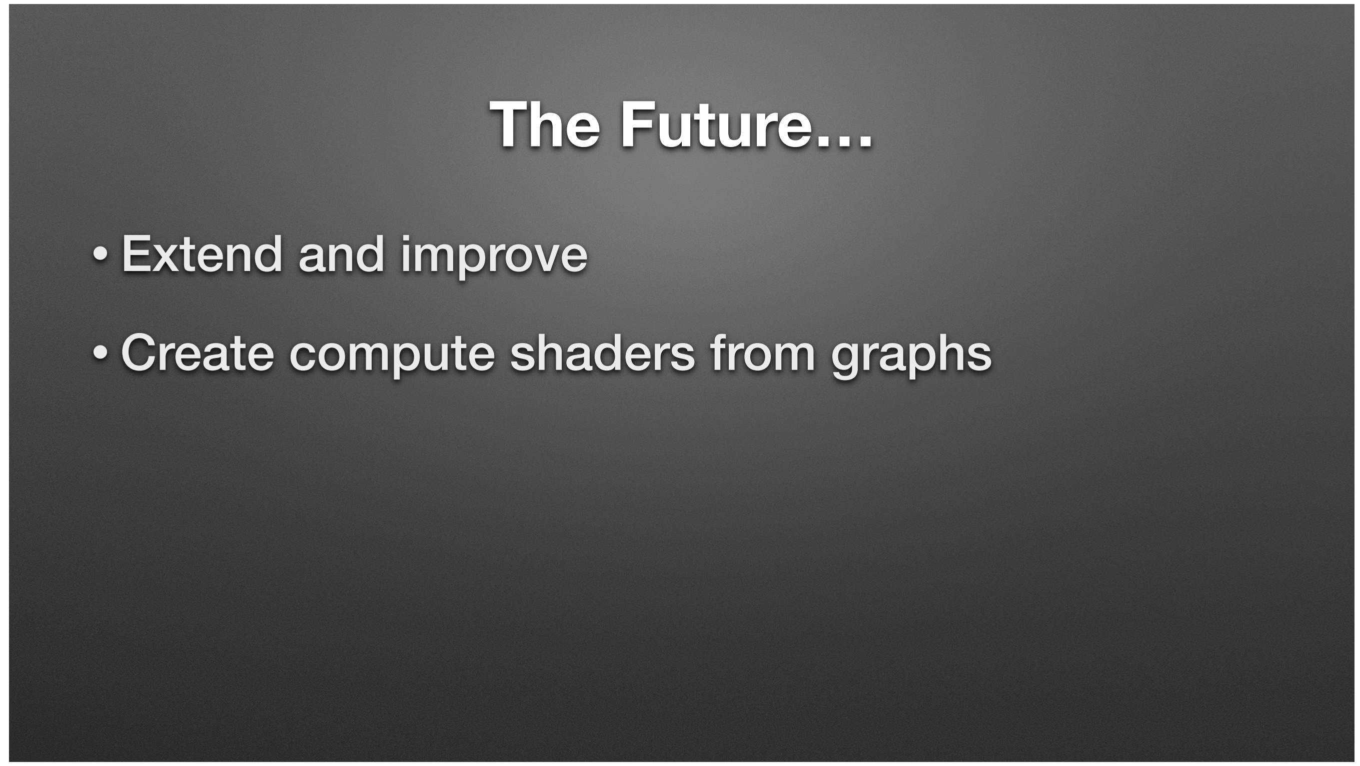
We had a wide **variety of sample rates** on the PS3, because it was the main parameter we used to optimize memory usage of sounds. This led to some **very low quality samples** in the game.
On the **PS4** we **only** used samples with **48kHz** rate. The main way to optimize was to switch them to **MP3 encoding** and adjust the bit rate.



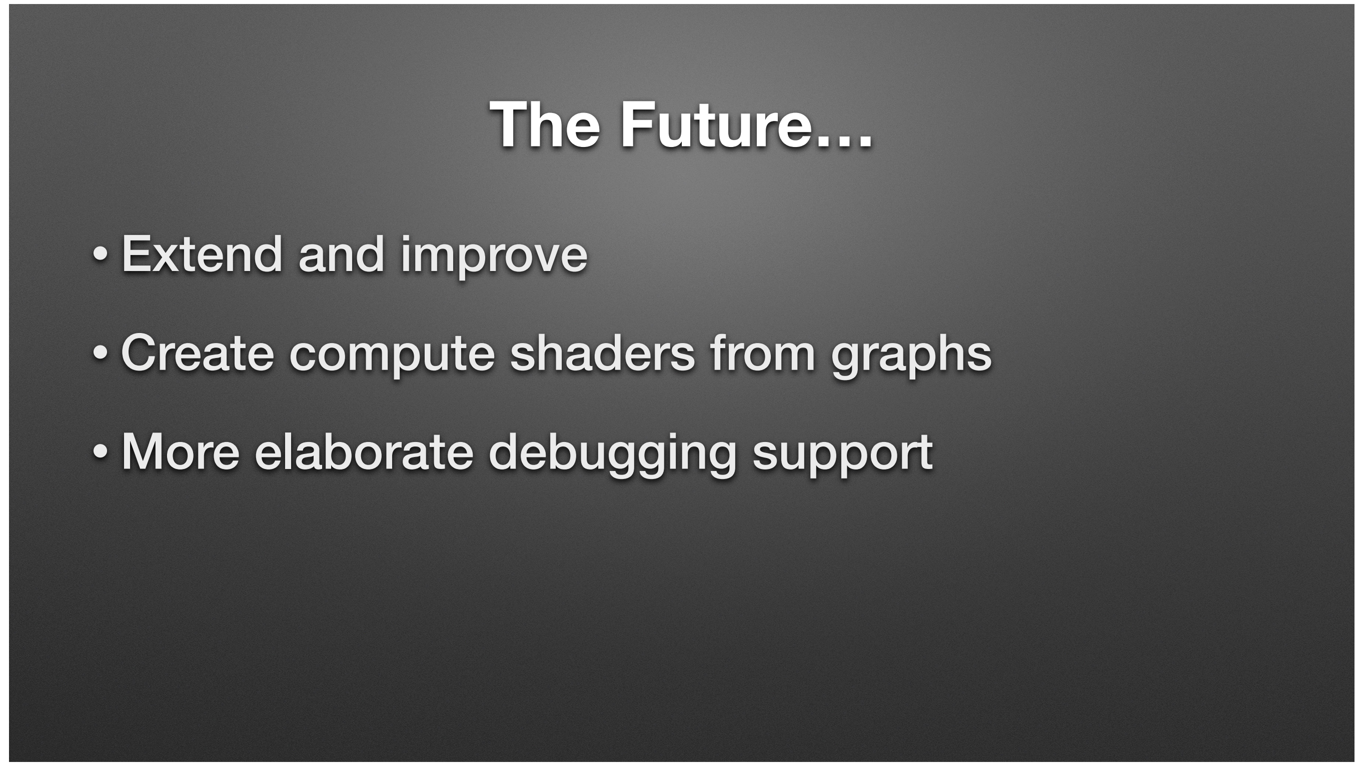
We had a wide **variety of sample rates** on the PS3, because it was the main parameter we used to optimize memory usage of sounds. This led to some **very low quality samples** in the game.
On the **PS4** we **only** used samples with **48kHz** rate. The main way to optimize was to switch them to **MP3 encoding** and adjust the bit rate.

Andreas: Were do we go from here?
We’re planning to **extend and improve** this system in the future, and use it for **more purposes**. E.g. changing the **code generator** to allow the creation of **compute shaders**. Also we'll add more **elaborate debugging support**, and lots of new nodes of course. We’ll have **more detailed profiling support**, to measure the execution time of the graph for each node, to allow us to **identify bottleneck** nodes that need to be optimised.
We’re also thinking about different ways to use this system, to **create samples on the fly** for variations. In this scenario we would add nodes to allow waveforms to be output into buffers, which can then be played at a later point in time.

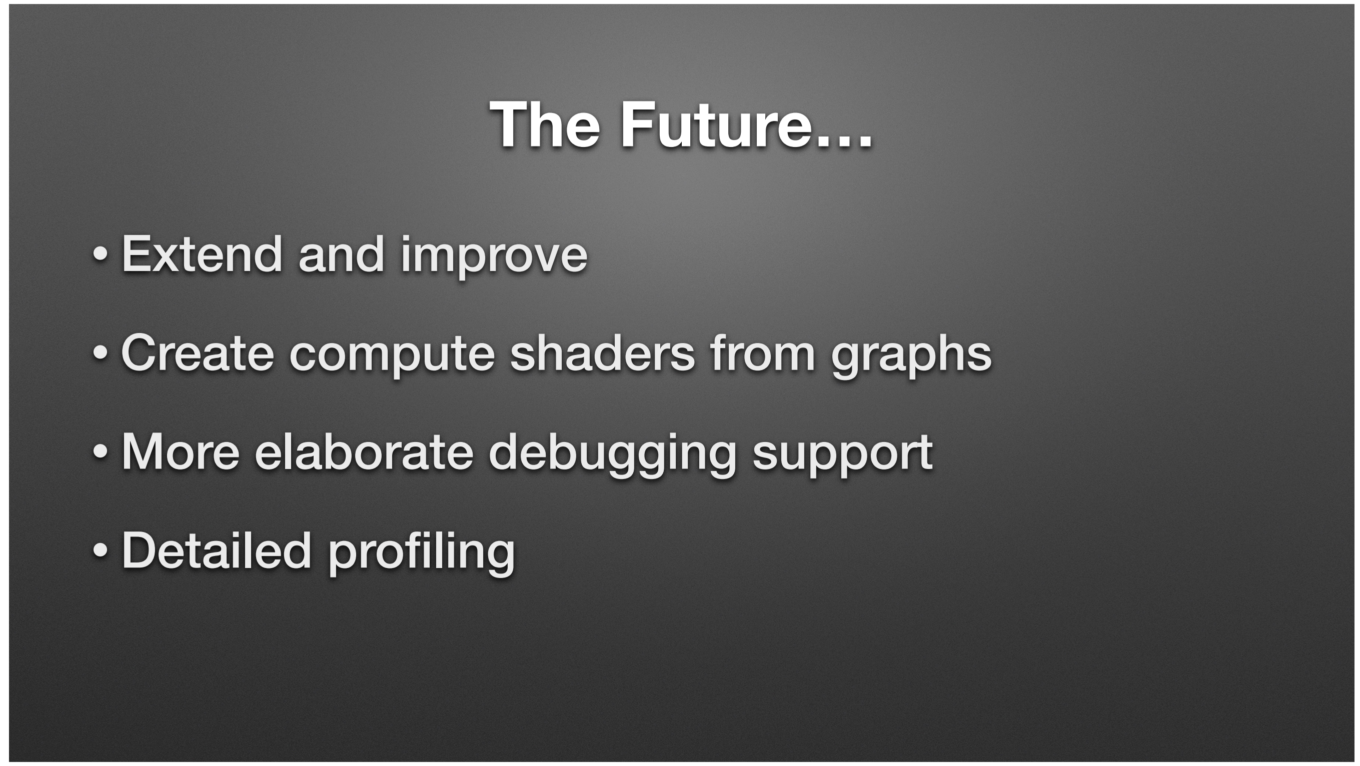
Andreas: Were do we go from here?
We're planning to **extend and improve** this system in the future, and use it for **more purposes**. E.g. changing the **code generator** to allow the creation of **compute shaders**. Also we'll add more **elaborate debugging support**, and lots of new nodes of course. We'll have **more detailed profiling support**, to measure the execution time of the graph for each node, to allow us to **identify bottleneck** nodes that need to be optimised.
We're also thinking about different ways to use this system, to **create samples on the fly** for variations. In this scenario we would add nodes to allow waveforms to be output into buffers, which can then be played at a later point in time.



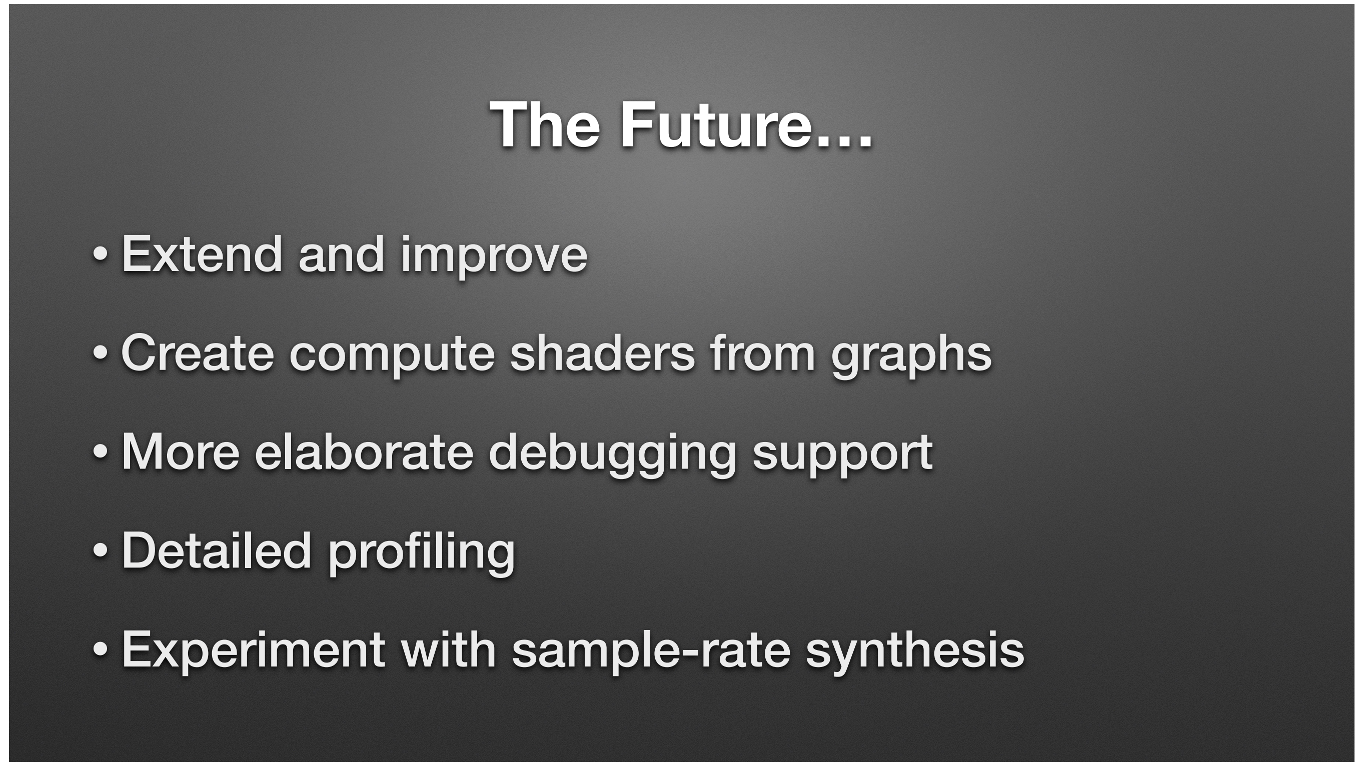
Andreas: Were do we go from here?
We're planning to **extend and improve** this system in the future, and use it for **more purposes**. E.g. changing the **code generator** to allow the creation of **compute shaders**. Also we'll add more **elaborate debugging support**, and lots of new nodes of course. We'll have **more detailed profiling support**, to measure the execution time of the graph for each node, to allow us to **identify bottleneck** nodes that need to be optimised.
We're also thinking about different ways to use this system, to **create samples on the fly** for variations. In this scenario we would add nodes to allow waveforms to be output into buffers, which can then be played at a later point in time.



Andreas: Were do we go from here?
We're planning to **extend and improve** this system in the future, and use it for **more purposes**. E.g. changing the **code generator** to allow the creation of **compute shaders**. Also we'll add more **elaborate debugging support**, and lots of new nodes of course. We'll have **more detailed profiling support**, to measure the execution time of the graph for each node, to allow us to **identify bottleneck** nodes that need to be optimised.
We're also thinking about different ways to use this system, to **create samples on the fly** for variations. In this scenario we would add nodes to allow waveforms to be output into buffers, which can then be played at a later point in time.

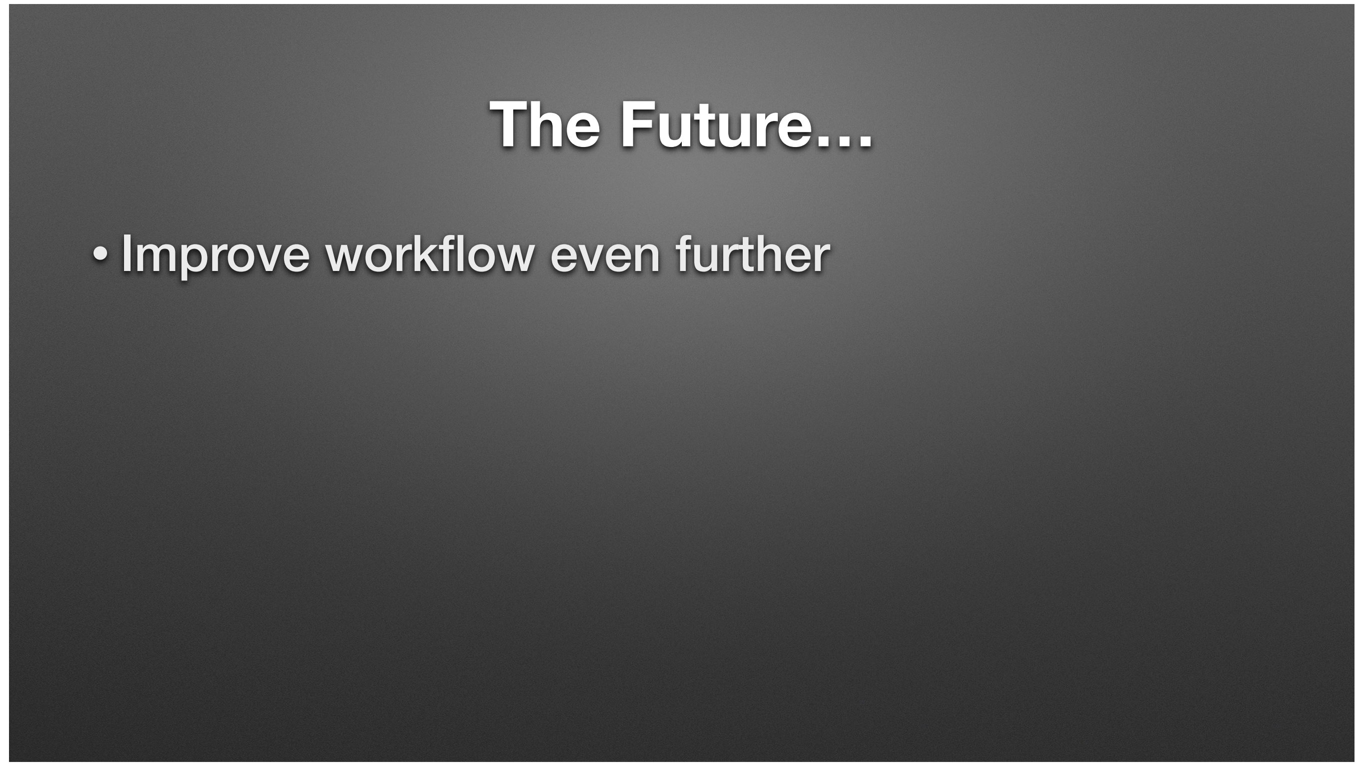


Andreas: Were do we go from here?
We’re planning to **extend and improve** this system in the future, and use it for **more purposes**. E.g. changing the **code generator** to allow the creation of **compute shaders**. Also we'll add more **elaborate debugging support**, and lots of new nodes of course. We’ll have **more detailed profiling support**, to measure the execution time of the graph for each node, to allow us to **identify bottleneck** nodes that need to be optimised.
We’re also thinking about different ways to use this system, to **create samples on the fly** for variations. In this scenario we would add nodes to allow waveforms to be output into buffers, which can then be played at a later point in time.

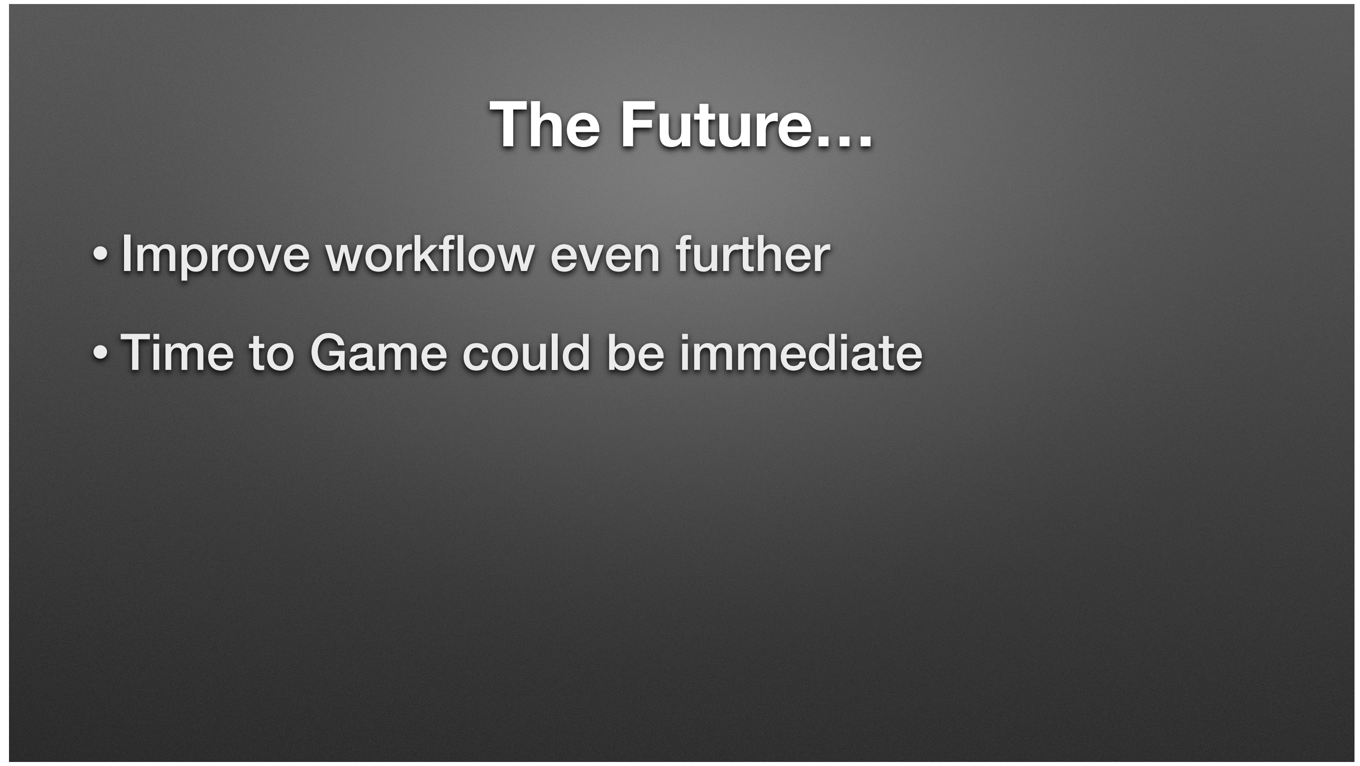


Andreas: Were do we go from here?
We're planning to **extend and improve** this system in the future, and use it for **more purposes**. E.g. changing the **code generator** to allow the creation of **compute shaders**. Also we'll add more **elaborate debugging support**, and lots of new nodes of course. We'll have **more detailed profiling support**, to measure the execution time of the graph for each node, to allow us to **identify bottleneck** nodes that need to be optimised.
We're also thinking about different ways to use this system, to **create samples on the fly** for variations. In this scenario we would add nodes to allow waveforms to be output into buffers, which can then be played at a later point in time.

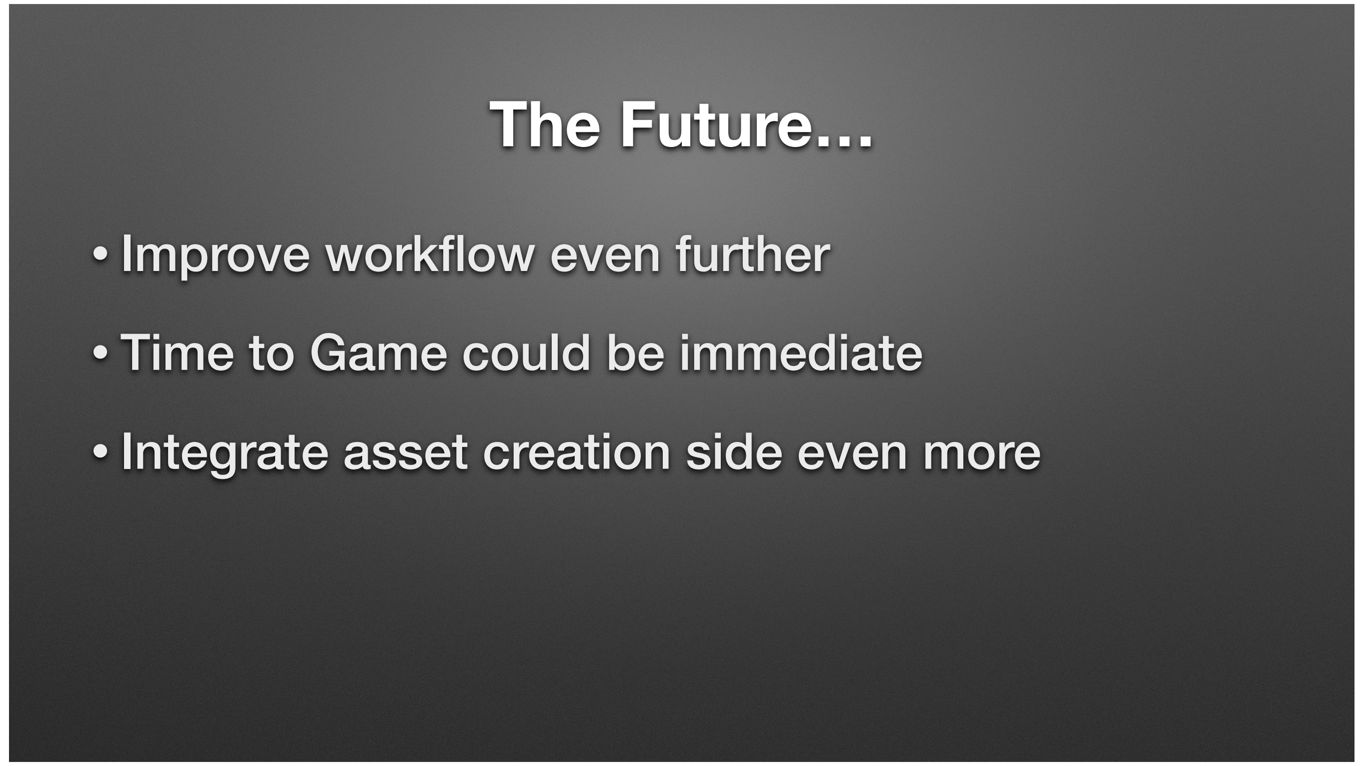
Andreas: Were do we go from here?
We're planning to **extend and improve** this system in the future, and use it for **more purposes**. E.g. changing the **code generator** to allow the creation of **compute shaders**. Also we'll add more **elaborate debugging support**, and lots of new nodes of course. We'll have **more detailed profiling support**, to measure the execution time of the graph for each node, to allow us to **identify bottleneck** nodes that need to be optimised.
We're also thinking about different ways to use this system, to **create samples on the fly** for variations. In this scenario we would add nodes to allow waveforms to be output into buffers, which can then be played at a later point in time.



Andreas: Were do we go from here?
We're planning to **extend and improve** this system in the future, and use it for **more purposes**. E.g. changing the **code generator** to allow the creation of **compute shaders**. Also we'll add more **elaborate debugging support**, and lots of new nodes of course. We'll have **more detailed profiling support**, to measure the execution time of the graph for each node, to allow us to **identify bottleneck** nodes that need to be optimised.
We're also thinking about different ways to use this system, to **create samples on the fly** for variations. In this scenario we would add nodes to allow waveforms to be output into buffers, which can then be played at a later point in time.



Andreas: Were do we go from here?
We're planning to **extend and improve** this system in the future, and use it for **more purposes**. E.g. changing the **code generator** to allow the creation of **compute shaders**. Also we'll add more **elaborate debugging support**, and lots of new nodes of course. We'll have **more detailed profiling support**, to measure the execution time of the graph for each node, to allow us to **identify bottleneck** nodes that need to be optimised.
We're also thinking about different ways to use this system, to **create samples on the fly** for variations. In this scenario we would add nodes to allow waveforms to be output into buffers, which can then be played at a later point in time.

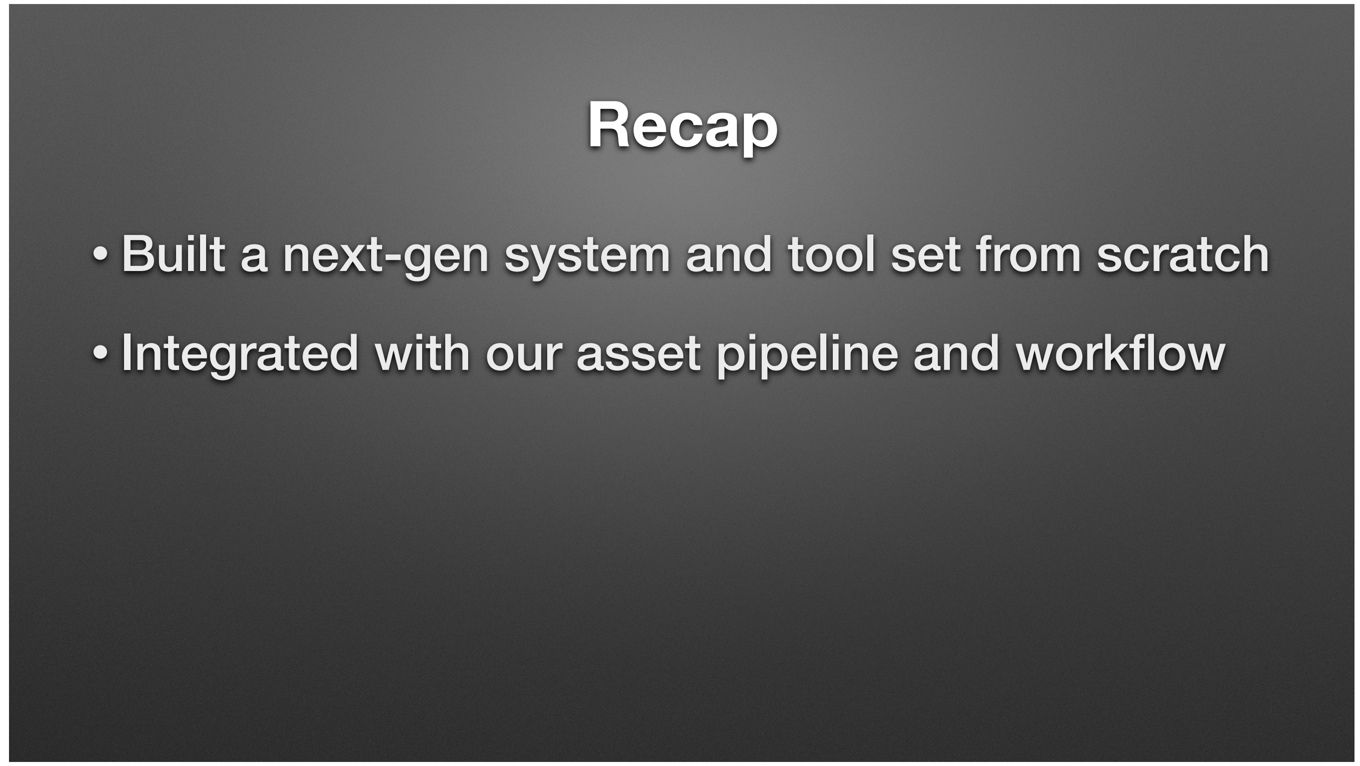


Andreas: Were do we go from here?
We're planning to **extend and improve** this system in the future, and use it for **more purposes**. E.g. changing the **code generator** to allow the creation of **compute shaders**. Also we'll add more **elaborate debugging support**, and lots of new nodes of course. We'll have **more detailed profiling support**, to measure the execution time of the graph for each node, to allow us to **identify bottleneck** nodes that need to be optimised.
We're also thinking about different ways to use this system, to **create samples on the fly** for variations. In this scenario we would add nodes to allow waveforms to be output into buffers, which can then be played at a later point in time.

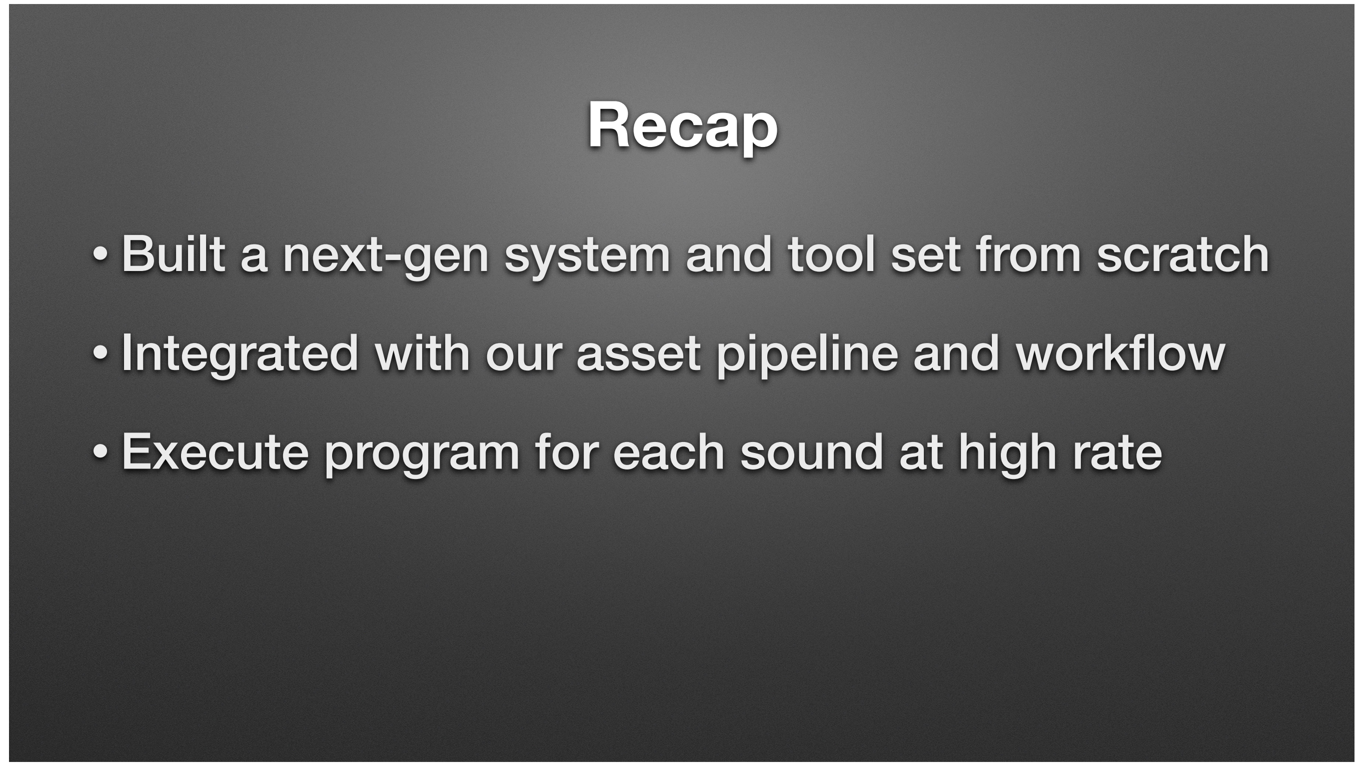
Recap

Andreas: So to quickly recap. In order to allow us to **fulfil the vision** we had for a **next-gen sound system and toolset**, we've created an engine that plays sounds by **executing a dataflow logic graph**, which was done by **generating C++** code which is compiled to **native code**, for performance reasons. We've used this system with **lots of inputs** from the **game engine** to create **sounds** that **dynamically change** to reflect the environment they're in and to make them **adapt** more directly **to** the **game situation**. Thus we've allowed sound designers to create behaviour for their sounds themselves, with the possibility to reuse and share this logic among different sounds.

Andreas: So to quickly recap. In order to allow us to **fulfil the vision** we had for a **next-gen sound system and toolset**, we've created an engine that plays sounds by **executing a dataflow logic graph**, which was done by **generating C++** code which is compiled to **native code**, for performance reasons. We've used this system with **lots of inputs** from the **game engine** to create **sounds** that **dynamically change** to reflect the environment they're in and to make them **adapt** more directly **to** the **game situation**. Thus we've allowed sound designers to create behaviour for their sounds themselves, with the possibility to reuse and share this logic among different sounds.



Andreas: So to quickly recap. In order to allow us to **fulfil the vision** we had for a **next-gen sound system and toolset**, we've created an engine that plays sounds by **executing a dataflow logic graph**, which was done by **generating C++** code which is compiled to **native code**, for performance reasons. We've used this system with **lots of inputs** from the **game engine** to create **sounds** that **dynamically change** to reflect the environment they're in and to make them **adapt** more directly **to** the **game situation**. Thus we've allowed sound designers to create behaviour for their sounds themselves, with the possibility to reuse and share this logic among different sounds.

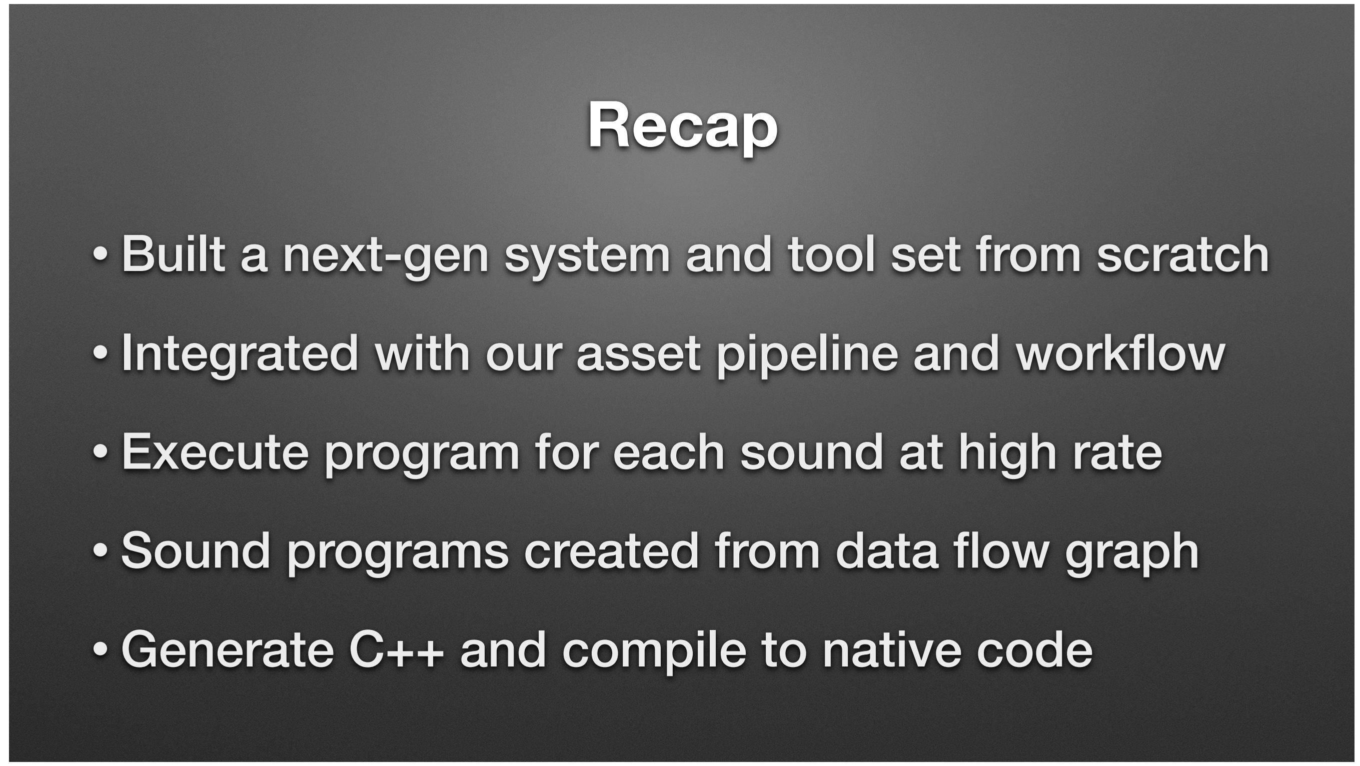


Andreas: So to quickly recap. In order to allow us to **fulfil the vision** we had for a **next-gen sound system and toolset**, we've created an engine that plays sounds by **executing a dataflow logic graph**, which was done by **generating C++** code which is compiled to **native code**, for performance reasons. We've used this system with **lots of inputs** from the **game engine** to create **sounds** that **dynamically change** to reflect the environment they're in and to make them **adapt** more directly **to** the **game situation**. Thus we've allowed sound designers to create behaviour for their sounds themselves, with the possibility to reuse and share this logic among different sounds.

Recap

- Built a next-gen system and tool set from scratch
- Integrated with our asset pipeline and workflow
- Execute program for each sound at high rate
- Sound programs created from data flow graph

Andreas: So to quickly recap. In order to allow us to **fulfil the vision** we had for a **next-gen sound system and toolset**, we've created an engine that plays sounds by **executing a dataflow logic graph**, which was done by **generating C++** code which is compiled to **native code**, for performance reasons. We've used this system with **lots of inputs** from the **game engine** to create **sounds** that **dynamically change** to reflect the environment they're in and to make them **adapt** more directly **to** the **game situation**. Thus we've allowed sound designers to create behaviour for their sounds themselves, with the possibility to reuse and share this logic among different sounds.



Andreas: So to quickly recap. In order to allow us to **fulfil the vision** we had for a **next-gen sound system and toolset**, we've created an engine that plays sounds by **executing a dataflow logic graph**, which was done by **generating C++** code which is compiled to **native code**, for performance reasons. We've used this system with **lots of inputs** from the **game engine** to create **sounds** that **dynamically change** to reflect the environment they're in and to make them **adapt** more directly **to** the **game situation**. Thus we've allowed sound designers to create behaviour for their sounds themselves, with the possibility to reuse and share this logic among different sounds.

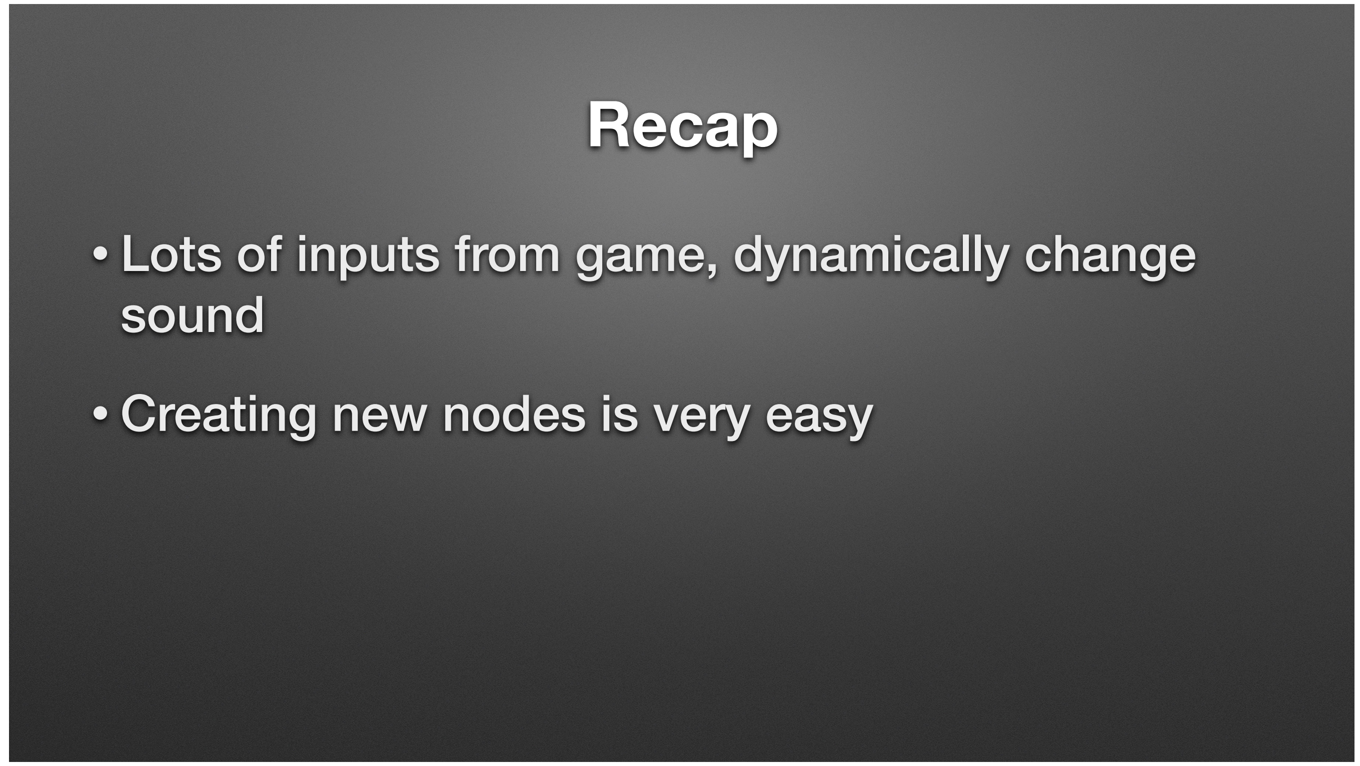
Recap

New nodes for the graph representation can be **easily added** as game assets and allow the **system to scale** with the needs of their users. It has been embraced by other disciplines in our company for various purposes.

Recap

- Lots of inputs from game, dynamically change sound

New nodes for the graph representation can be **easily added** as game assets and allow the **system to scale** with the needs of their users. It has been embraced by other disciplines in our company for various purposes.



New nodes for the graph representation can be **easily added** as game assets and allow the **system to scale** with the needs of their users. It has been embraced by other disciplines in our company for various purposes.

Recap

- Lots of inputs from game, dynamically change sound
- Creating new nodes is very easy
- Allows system to improve and designers to be creative

New nodes for the graph representation can be **easily added** as game assets and allow the **system to scale** with the needs of their users. It has been embraced by other disciplines in our company for various purposes.

Recap

- Lots of inputs from game, dynamically change sound
- Creating new nodes is very easy
- Allows system to improve and designers to be creative
- Has been embraced by other disciplines

New nodes for the graph representation can be **easily added** as game assets and allow the **system to scale** with the needs of their users. It has been embraced by other disciplines in our company for various purposes.

Any questions?