Synthspace Audio Layer

License

Discord

For inspiration, support and general discussions about the future of making music in VR, please join the **Synthspace discord server**: https://discord.gg/x5k6Ng5MtC

Collaboration

This is the first step towards opening up Synthspace for collaboration! I've outlined my vision here: https://projectshares.org

Repository

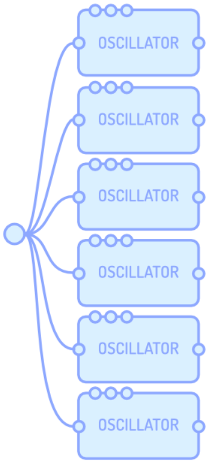
The source code to the *Synthspace Audio Layer* is hosted at https://github.com/brightlightrx/synthspace-audio-layer

Alright! Let's get started!

Basics

This is a **module**

Each **module** is a visualization of one or more **nodes**
(For example: the pictured *Value Controlled Oscillator* module contains 6 different *Oscillator* nodes all hooked up to the same Frequency input)



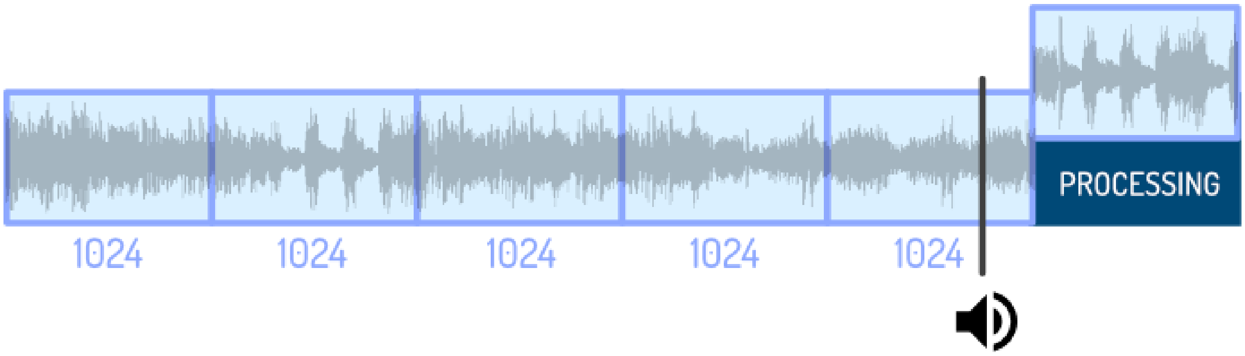
Nodes are coded in **C#** or **FAUST**

Option 1 - C#: AudioDataNodes

The base class for all nodes is **AudioDataNode** - this is where the data processing happens. A node can take in multiple streams of data and output multiple streams of data. There are 4 different types of inputs/outputs:

- **AudioDataInlet**: An inlet that holds a full buffer of values (can be visualized as a blue socket, a knob or socket + knob)
- **SimpleData**: An inlet that holds only a single value (can be visualized as a knob, switch or button)
- **AudioData**: An outlet - holds one full buffer of values (visualized as a red socket)
- **ContainerSlot**: Holds a single Container (Visualized as a cartridge) - at the moment there are 2 types: **Sample** (Audio Data) or **Pattern** (an array of values representing MIDI Pitches)

Hold on - What's a buffer? Audio processing happens in batches. We feed the audio system with new data at regular intervals. For example: If we run at a SampleRate of 48000 (48000 individual values per second) with a buffersize of 1024 that means that we're sending 1024 new values to the audio system every 0.0213 seconds. AudioDataInlets and AudioData each hold one full buffer of data.



How do multiple modules play together?

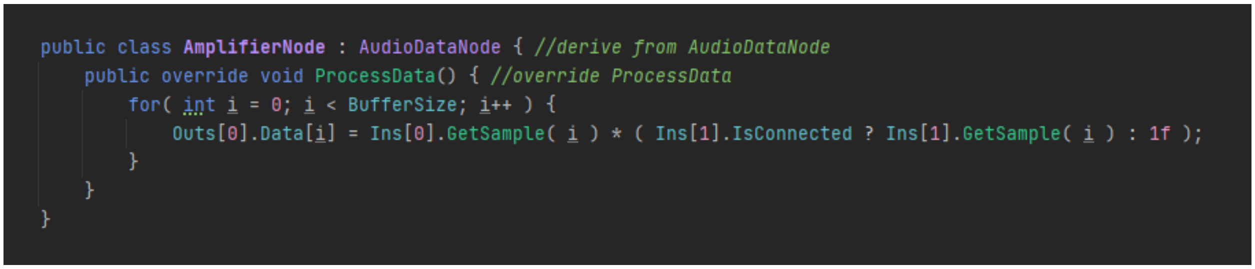
(The system does all of this for you)

It all starts with an OUT module containing an **AudioOutputNode** that feeds data into an *AudioSource*. To be able to play sound it first needs to know what to play, so it calls ***PrepareData*** on itself. This, in turn, goes through all its inputs and calls ***GetData*** on them before calling ***ProcessData*** on itself. In essence: it makes sure it has the latest data from all connected nodes before processing the data and filling its outputs (or in the case of the AudioOutputNode: playing it back).

Now if a node has an input and something is connected to that input, then the ***GetData*** call will call ***PrepareData*** on the connected node, which in turn will go through their inputs and so on, all the way up the node-tree.
By the time the AudioOutputNode is done with ***PrepareData***, all connected nodes have in turn prepared and processed their data and everything is ready to go!
Now it can do **ProcessData** and fill its outputs.

The cool thing about this system is that **you don't have to worry about any of this** when writing nodes. All you really need to know is to derive from **AudioDataNode** and override ***ProcessData*** - once ***ProcessData*** is called you can rely on all your Inputs having the latest data and all you need to do is fill your Outputs.

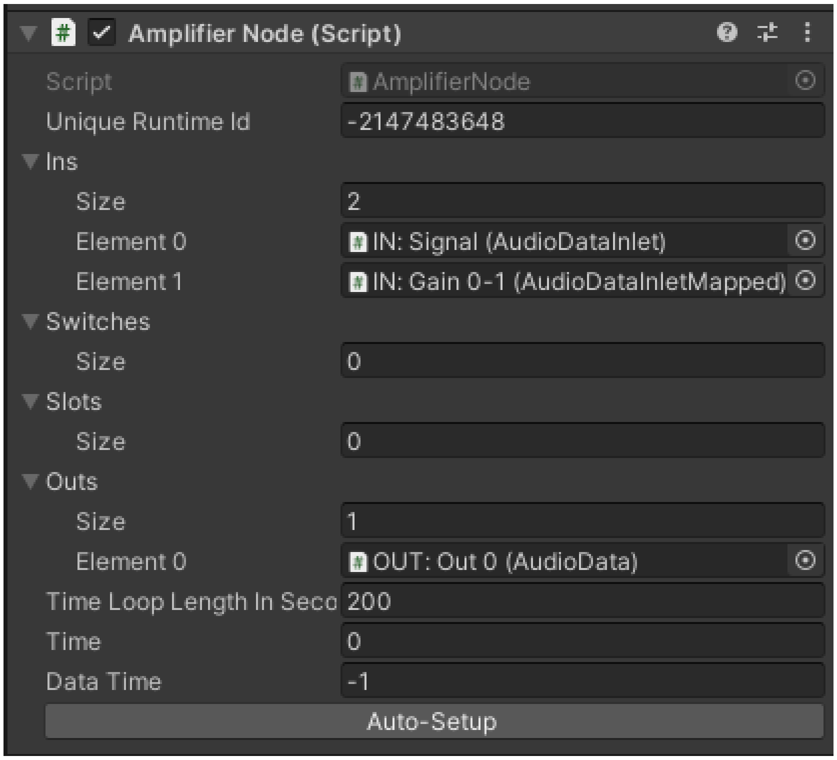
With this system you can write an entire *Amplifier Node* in under 10 lines of code:



```
public class AmplifierNode : AudioDataNode { //derive from AudioDataNode
    public override void ProcessData() { //override ProcessData
        for( int i = 0; i < BufferSize; i++ ) {
            Outs[0].Data[i] = Ins[0].GetSample( i ) * ( Ins[1].IsConnected ? Ins[1].GetSample( i ) : 1f );
        }
    }
}
```

All it does is step through the buffer and fill each sample of its Output. What it fills in is simply Input 1 (*Ins[0]*) multiplied by Input 2 (*Ins[1]*) - or multiplied by 1 if nothing is connected to Input 2.

Then all you need to do is set up the node and it's ready to be turned into a module! (Just change the array size of your Ins/Outs to the desired number and click Auto-Setup and it will create and fill in the necessary pieces for you!)



Script	AmplifierNode
Unique Runtime Id	-2147483648
Ins	
Size	2
Element 0	IN: Signal (AudioDataInlet)
Element 1	IN: Gain 0-1 (AudioDataInletMapped)
Switches	
Size	0
Slots	
Size	0
Outs	
Size	1
Element 0	OUT: Out 0 (AudioData)
Time Loop Length In Seco	200
Time	0
Data Time	-1

Signals

All signals are in a range between -1 and 1. This ensures that everything can be connected to everything else.
But different signals can represent very different things:

A **Frequency** signal represents a range of 0 to 8000 Hz (-1 is 0 Hz, 1 is 8000Hz)
A **Pitch** signal represents a range of 0 to 127 (-1 is 0, 1 is 127)
A **Gate** signal represents a range of 0/off to 1/on (-1 is 0, 1 is 1)

Does it make sense to connect a Frequency output to a Gate input? Probably not, but you can!

So all signals are between -1 and 1, but really mean different things. To make our lives easier when working with this I created a special version of **AudioDataInput** called **AudioDataInputMapped** - this version automatically maps the -1 to 1 range to a custom range you set. When you call ***GetSample***(i) on it, it will automatically remap it to your custom range, so you can use it in your code and get the actual value you want.

Option 2 - FAUST (Functional Programming Language for Real Time Signal Processing)

The other way of coding nodes for Synthspace is using **FAUST** (https://faust.grame.fr/)

The workflow looks like this:

1) Use the Faust web IDE (https://faustide.grame.fr/) to create a node
2) Compile it to a library
3) Import that into unity
4) Turn the wrapper into an AudioDataNode.

The repository contains the Flanger as an example - have a look at how ProcessData assigns parameters and tells the Faust Context to process its data and write the result directly into the Flanger Output.

The **challenge** here is that we need to **make sure it supports Synthspace for PC as well as Quest (Android)**. We need to export a Windows64 dll as well as an Android arm64 library and (at this point) the web IDE can't export the second one. To solve this, I've created a VDI disk image containing Ubuntu 20.04 with Faust set up correctly. You can start this up in VirtualBox and export everything from there. Here's the link to the VDI:
https://drive.google.com/file/d/1Eq8lY1l86hRfZYCqyzmyRiUg2VL8O63w/view