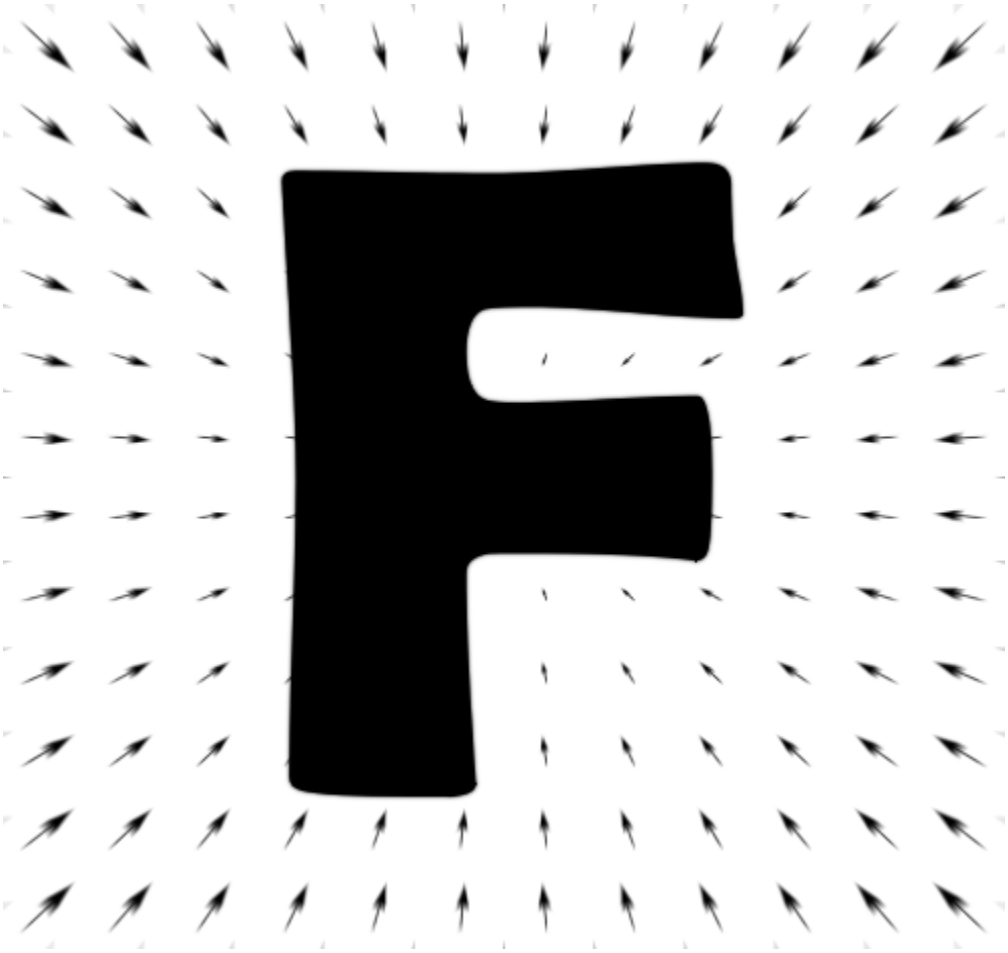
Spring 2022

A New Way to Make Music: Processing Digital Audio in Virtual Reality

Gavin E. Payne
Bard College

A New Way to Make Music: Processing Digital Audio in Virtual Reality

Senior Project Submitted to The Division of Science, Math, and Computing of Bard College
Annandale-on-Hudson, New York



Gavin Payne

May 2022

Abstract

The work of this project attempts to provide new methods of creating music with technology. The product, Fields, is a functional piece of virtual reality software, providing users an immersive and interactive set of tools used to build and design instruments in a modular manner. Each virtual tool is analogous to musical hardware such as guitar pedals, synthesizers, or samplers, and can be thought of as an effect or instrument on its own. Specific configurations of these virtual audio effects can then be played to produce music, and then even saved by the user to load up and play with at a later time. There are still goals to expand this project, and turn it into something more akin to a professional digital audio workstation, but for the time being it remains a modular synthesizer with nearly limitless configurations in a virtual 3D environment. Included is a developer library, giving users the ability to integrate their own musical software solutions into the virtual environment and expand the work of this project.

Table of Contents

```
Paper A_New_Way_to_Make_Music(Human reader)
{
    Section Introduction() {
        Introduction_To_Fields();

        Software_Architecture_Diagram();
        Background_Knowledge();

        Fields_Library();
        Materials();
    }
    Section User_Guide() {
        Using_Fields();









    }
    Section Mechanics() {
        DSPGraph();


        Touching_The_Execute_Function();

        AudioProcessor_and_ProcessorBundle();









    }
    Section Developer_Guide() {
        Building_a_ProcessorBundle();


        Building_an_IAudioKernel();









        Building_an_AudioProcessor();



        Building_an_IAudioKernelUpdate();


        Building_a_ParameterController();
    }
    Section Making_Music() {
        Synthesizer();









        Creative_Environment();

    }
    Page Bibliography;
}
```

Introduction

Introduction to Fields

What is it?

Fields is going to be a digital audio workstation (DAW) in virtual reality. Currently however, it only accomplishes part of what a DAW does, and in a sense, is just a glorified modular synthesizer. In Fields, you can generate and manipulate sounds, from playing virtual instruments to mixing and processing the instruments with effects. Similar to how modular synthesizers run on a large analog circuit, the audio engine in Fields simulates the signal flow architecture of those circuits by processing audio samples along a directed and acyclic graph. Below is the graph of a basic synthesizer:

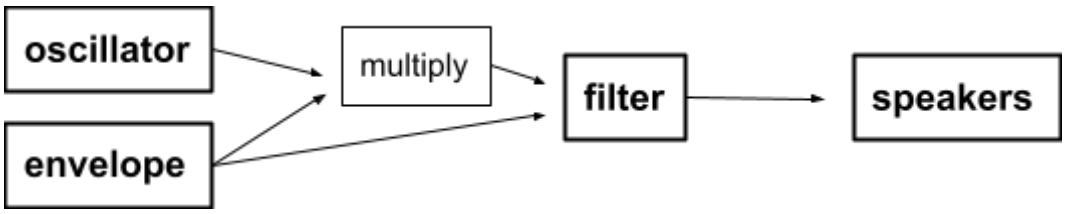


Figure 1: A signal flow graph of a basic audio synthesizer

In analog audio synthesis, Figure 2 represents a circuit of electronics that generates and processes sound, such as a network of guitar pedals connecting a guitar to its amplifier. In digital audio, the above graph represents the architecture of a signal processing algorithm, where each node on the graph acts as either the guitar (generates sound), the pedal (alters the sound), or the amplifier/speaker (makes the sound audible). Fields comes included with a set of tools ready to make music with. To generate sound, Fields includes a polyphonic synthesizer with multiple voices and stereo detune capabilities. To further manipulate the sound, Fields includes a waveshaping saturator, a stereo delay with

feedback, a filter module with multiple modes, and a stereo panning device. The audio devices included can be configured in any configuration, and can be put together in both parallel and serial circuits that include duplicate instances of each effect. Fields is a modular audio sandbox with limitless expansion.

Equally important as the software itself, is the Fields Library, which abstracts the core elements of the Fields audio system into templated packages that developers can use to construct their own audio processing devices within the Fields Virtual Environment. A developer is able to not only implement their own custom audio processing algorithms into a custom 3D model that can sit inside the audio processing graph, but can also implement their own virtually interactable control devices for their audio effect. This means that if someone has a unique idea for a way to interact with a virtual instrument in virtual reality, they can implement their control device into Fields as well.

Fields is a live audio processing system, in an interactive 3D virtual environment, with a developer toolbox, that facilitates the creation of music, entire new instruments, and novel audio processing algorithms.

Software Architecture Diagram

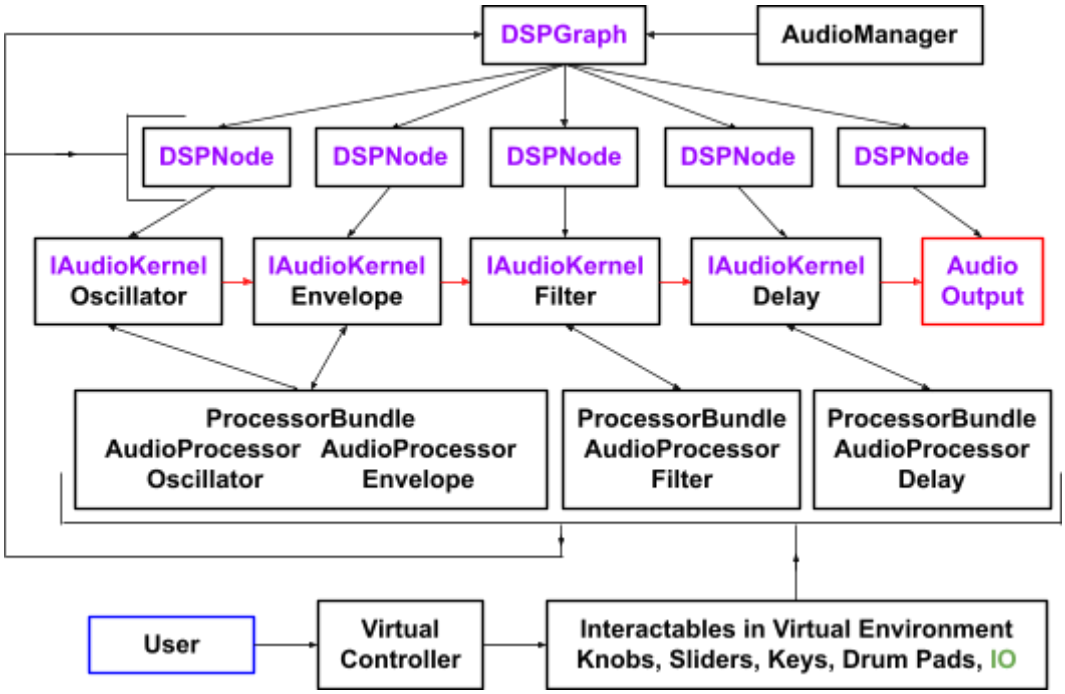


Figure 2: Diagram of Fields. This is the underlying structure of the audio processing system, from user input (blue box) to audio output (red box). The black arrows show data communication between the different objects, and the arrows in red show audio sample information being processed. In purple are structures from external sources, and in black are new structures built for the final software, available through the Fields Library. The Input/Output system, IO, is in green because there are many structures of that system that are not shown on this graph, but mediate the connections between the Virtual Controller, AudioProcessors, and DSPNodes. For simplicity, the ProcessorBundle class has been paired with the AudioProcessor. The ProcessorBundle can contain multiple AudioProcessors in a complex arrangement, and it manages the behaviors of those AudioProcessors as one distinct audio effect object in the Virtual Environment.

Background Knowledge

This project involves topics such as virtual reality, digital audio, and multithreaded programming. Most niche terms will be defined as they come up, however some background will be useful. For the purposes of this project, virtual reality is in reference to a 3D computer generated environment, like found in many modern video games, that is interacted with through hardware that is strapped to the body, tracking movement and translating it into the world. There are two screens, one for each eye, that display stereographic information directly into the eyes. Disclaimer: the content of this project is entirely software, and meant to run on a pre built virtual reality device.

More relevant to this project specifically is how the audio is generated and provided to the listener in this experience. In short, digital audio is a collection of measurements of sound pressure, measured so quickly that the continuous waveform of sound is reconstructed without loss of information audible in the human range. The processing of sound in this project is not through electrical signals, or vibrations in the air, but through manipulating the recorded samples of sound with math. There will be more on the specifics of digitally sampled audio in a later section, but it is helpful to know the collection of values stored on a computer represent a sound wave traveling in air, and vice versa.

How all this audio work comes together in a virtual reality environment, relies on some programming techniques that are not processing audio, nor generating the 3D world, but rather performing a type of organization that allows all the code of interesting and important things to run smoothly and communicate with each other. Some of the description of this project will stray away from the fun of audio processing and virtual reality, to describe a little more practically how Fields is able to run on a real device in the

real world. In short, modern computing occurs on multiple processor cores, and programs that have strict and short deadlines for computing values, such as real time audio processing, have multiple things going on at once. There are unique problems that occur when running multiple bits of code at the same time, and some solutions will be explored further on.

Here is a list of topics that could be useful to know more about in reference to this project:
The reader is encouraged to search for these terms as the need for greater background arises.

Virtual Reality
Modular Synthesis
Effects Unit
Game Engine
Reverb Effect
Digital Filter
Audio Envelope

Fields Library

Most of the code written for this project has been organized into a library for the ability for end-users to build custom audio effects for the Fields platform. The library will also allow anyone to take advantage of Unity's experimental audio package DSPGraph, in a simplified interface that lends itself towards building real-time audio editing applications. Using the API documentation found here: https://zggof1999.github.io/Fields-VR-Audio/, you can find instructions, examples, and a download link for the library itself.

The documentation will be referenced throughout the rest of this paper.

Materials

VR Headset

The headset used for development is the Oculus Quest 2 from the company now known as Meta. It was chosen for a variety of reasons, from the low cost to the ability to act as both a standalone android device as well as a tethered input and output device for a personal computer. For visuals, it has two 120Hz LCD screens at 1832 x 1920 per eye, and for audio, well, it has an audio jack to make up for the loud but crunchy speakers.

GameEngine

The environment is made in Unity, a cross-platform game engine developed by Unity Technologies. Because VR is in an early state with few standards, a game engine with a history of VR support for multiple platforms, such as Android and Windows in this case, is extremely useful for focusing on the audio technology itself rather than the hardware compatibility . Unity uses C# as its primary programming language, and even offers its own system to write multithreaded code that is significantly faster than C#'s own way of doing so (for most use cases).

- ➔ DSPGraph

  DSPGraph is a node based system provided by Unity that provides a backbone for scheduling audio processing automatically in both serial and parallel circuits. When provided processing nodes and their connections, executing the graph sends each node's output into the next node's input, in the form of an multi-channel array of audio samples. DSPGraph was first available to the public in early 2019, but has since been abandoned and stuck in Unity's experimental development stage, with little to no documentation.

➔ OpenXR

The system that handles communication between the VR headset and the game engine is OpenXR. It is an open source standard that allows for universal programming for many vr headsets. With OpenXR, Unity can communicate indirectly to all vr controllers and headsets instead of directly with each in a different standard of their own.

3D Modeling Software

Cinema 4D is a professional 3D modeling, animation, simulation and rendering software solution from Maxon. It was used to build all the models used, and exported .fbx models for Unity to render in real time.

Code Editor

All of the code was written in Visual Studio 2019 from Microsoft, which integrates seamlessly into Unity.

The project was developed and compiled on a x86 machine running Windows

User Guide

Using Fields

Figure 3: The welcome menu of Fields

When you first enter the application, you will be in a welcome area with a sign and menu straight ahead. Following common virtual reality conventions, you will see representations of your game controllers. The controllers' buttons are color coordinated to make remembering what each button does easier. To interact with the main menu, you can point the controllers towards the menu like you would use a laser pointer. A beam will shine onto the menu, and the menu icon pointed at will shine blue. To select that menu item, press the **pink** trigger button. The laser beam user interface was made to provide interactive menus out of reach from the user. If something looks like a button that can be 'clicked' try pointing at it and a beam will most likely appear.

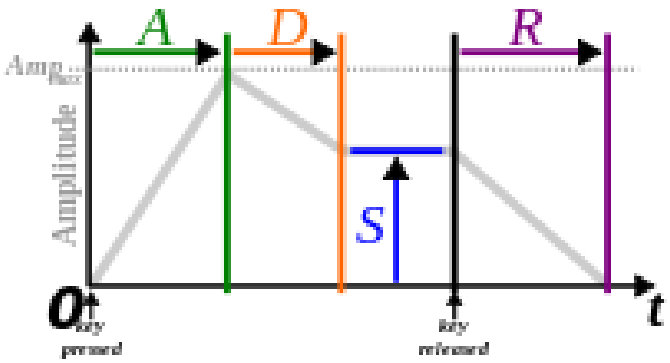
Figure 4: The virtual controller devices within Fields

Figure 5: A disconnected synthesizer inside Fields. 4.1: Input/output orbs, 4.2: Parameter knob, 4.3: Envelope curve, 4.4: Virtual Midi Keyboard, 4.5: X Button (delete/remove processor)

Selecting ‘Create Field’ from the title menu will transport you into a new room. This room is called a Field. The Field will initialize with a synthesizer in front. To the right of the synthesizer, is the output node of the graph. The graph starts off disconnected. To make sound from the synthesizer come out of the speakers, it must first be connected to the output. Connecting Audio Processors in Fields is easy. Similar to other node based creative programs, such as material design in rendering software, an output port exists on the right hand side of every AudioProcessor (Figure 1.1). Touching the output port with the cursor (the round ball at the end of the controller) will initiate a connection. The cursor will highlight, and simply touching the input port of another Audio Processor (port on the left hand side, Figure 1.1) will connect the two processors. Creating a cycle (a circular path that begins and ends on the same node) on the graph is not allowed, and will do nothing, including connecting a processor to itself. To disconnect two processors, simply repeat the process: touch the output port of one, and then the input port of another (or vice versa). If the processors are connected they will disconnect, and if they aren’t connected they will connect. To keep track of connections, the connected ports will glow in a coordinated color. If an Audio Processor is connected to multiple others, multiple highlighted color coordinated orbs will show.

Once the synthesizer is connected to the output, simply touching the keys on the keyboard with the cursor will play a note (Figure 1.4). The key will also highlight bright blue to signify its activation. Besides the keyboard and activating notes, the synthesizer has a few other ways of changing the parameters of the sound processing. The most common controller in Fields is the knob (Figure 1.2). The synthesizer has many knobs on its face, each labeled with their value and parameter name. For example, the pitch knob on the

synthesizer transposes the notes played by the amount shown under the knob. To interact with the knob, simply insert the cursor into the knob and rotate your wrist. The knob reacts to the relative z-axis rotation of the controller, so rotating your forearm as you would a key in a door works great. Just insert, rotate, remove, and repeat. The last thing on the synthesizer control panel is the envelope curve (Figure 1.3).



https://en.wikipedia.org/wiki/Envelope_(music)

Figure 6: Amplitude Time curve visualizing an Envelope and its Attack, Decay, Sustain, and Release parameters

Above is an example of an envelope curve. On the synthesizer, there are three white control points on the envelope. To move them around, place the cursor onto the point, and grab with the controller's **purple** grab button. The first control point is locked to the top of the curve (amplitude = 1). This is the attack point, and controls how long it takes for the audio signal to reach full amplitude from zero. The second point is the decay and sustain point. Its y-position controls the amplitude level the sound will end up at while holding down a note, while its x-position controls how long it takes for the amplitude to descend from the maximum to the sustained amplitude level. As long as a note is held down, the note will remain playing at this sustained amplitude, and only once released will the sound

slope down to zero amplitude. The final control point is pinned to the bottom of the curve (amplitude = 0), and controls how long it takes for a note to go quiet after letting go.

By changing the parameters of the synthesizer, fairly different sounds can be generated. For example, a common synthesizer sound is the 'supersaw'. To make a supersaw with the Fields synthesizer, the user can change the waveform to 'saw' (left waveform knob, 3rd position)[1], and increase the number of voices (right waveform knob). This will cause multiple sawtooth voices to be played at the same time. Then, the user can change the detune and width parameters. Detune controls slight pitch deviation for each voice, where more detune will cause the superaw to be bigger and rougher, while less detune will give the sound a more metallic color. The width parameter adjusts stereo deviations of each voice, making the sound more stereo or more mono. A common thing to do to a supersaw sound is to filter out the higher frequencies making it less harsh on the ears.

In Fields, adding effects to sounds is easy and limitless. In the room is a wall of effect icons. Just like any UI menu in Fields, pointing at an icon will cause the selection beam to appear, and pulling the trigger button on the controller while an icon is selected will cause the effect to appear in front of the user. Simply clicking the filter icon will cause a filter to appear. Then, like hooking up guitar pedals, the filter must be integrated into the signal. To filter out sound from the synth, the synthesizer needs to be plugged into the filter as shown in Figure 6. Then, to hear the sound, the filter needs to be plugged into the output.

[1] The waveform knobs will be replaced with a visual window that displays the waveform, as of now they are unlabeled and unintuitive.

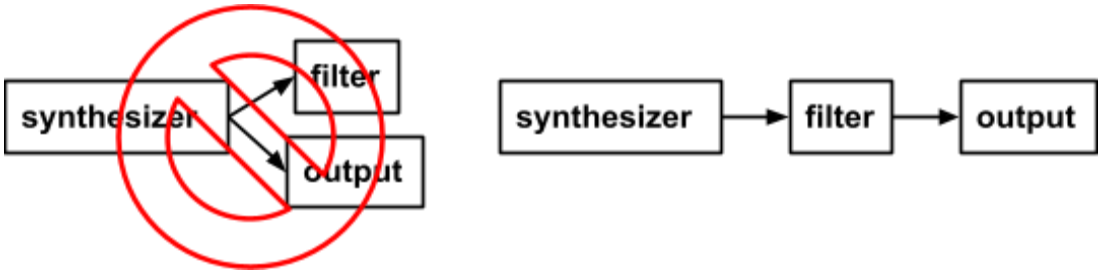


Figure 7: To filter a signal, the signal must feed into the filter, then to hear the filtered signal and not the original signal, the filtered signal must feed into the output.

When you are done with an AudioProcessor, inserting the cursor into the red 'x' on the top right corner and pressing the blue button will automatically disconnect the processor from the graph and delete it from the game world (Figure 1.5). Don't forget to save your project using the save wall and nifty custom built virtual QWERTY keyboard!

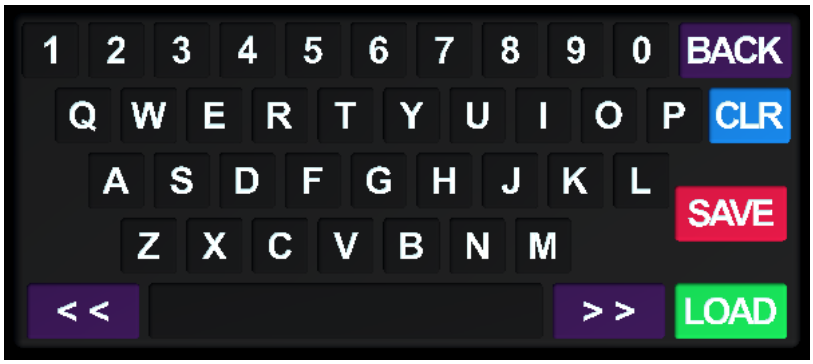


Figure 8: Virtual typing keyboard. The keyboard was entirely built from scratch for use in this project. This includes the modeling, interaction behaviors (haptics, lighting response, and key travel), and text input/display console. While not as effective as a physical keyboard, the virtual keyboard is a necessity for text input (without integrating speech processing), and turned out feeling responsive and fluid.

On the opposite side of the effects browser, there is a save file browser. On the top of the browser is a console that is permanently linked to the keyboard. To save a project, simply type in a name, and hit save. You will see it automatically appear in the browser. To load a file, simply type in the name of the file you want to load (it will be displayed in the browser but currently there is no UI beam selection support for the file system) and click load on the

keyboard. Stored locally on your device is a `.field` file, which you can transfer to other machines. It is planned to implement multiplayer functionality to Fields, where not only multiple people can join the same studio space at the same time, but where people can share save files with each other within the fields app. For now however, as long as the two versions of Fields you are transferring a save file between are the same, then the file will load with no issues.

Save Warning: There is no overwrite protection, so do not actually save over your favorite project by accident (maybe the save and load button are too close together, but a better solution for a file system is in the works)

Mechanics

DSPGraph

What is it?

DSPGraph is a package offered by Unity to schedule audio processing. DSPGraph is at the heart of the audio processing of this software. DSPGraph precisely provides a method of driving audio buffers between different processing nodes, automatically scheduling the computations in parallel when available. When acting on the graph, the AudioDriver starts at sink nodes, and works its way backwards to determine the schedule of the audio processing. It does so to ensure that the only audio processed is audio that will make its way to the sink. You can however flag the driver to compute all nodes of the graph, despite their output connection, and if so the graph will schedule as shown by Figure 8.

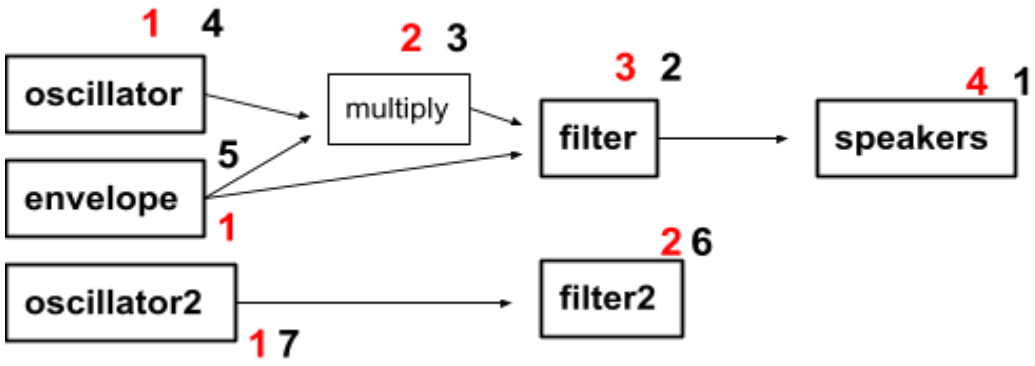


Figure 9: Graph depicting the scheduling process. A scheduling algorithm runs through the audio graph and determines the order the nodes process audio. Black: order seen by scheduler. Red: order scheduled

In the example instance of a graph shown in Figure 8, 3 audio buffers are allocated memory, and sent by reference to each of the audio processors marked with a red '1'. The nodes then execute their processing function on the audio buffers, and the audio driver combines the necessary buffers together if required, to be sent to the processors marked with red '2's. Here, processors marked with black '4' and '5' merge their buffers together when sent to the multiply function.

DSPNode, IAudioKernel, IAudioKernelUpdate

The DSPNode, IAudioKernel, and IAudioKernelUpdate, are the three objects that construct an interactable audio processing node on DSPGraph. The DSPNode is what sits on the graph, and is constructed using an IAudioKernel. IAudioKernels contain the method that does the actual audio processing. This 'Execute' function takes in pointers to input and output buffers, and sample accurate parameters, which are arrays of floats that use a sample offset as an index value. Using the parameter data, as well as any other data stored in the IAudioKernel, a processing algorithm on the input data will create output data that can be written into the output buffers. There are two ways of feeding data into an IAudioKernel, by parameter float buffers, and by IAudioKernelUpdates. The former are fed into the IAudioKernel in a similar way to the audio buffers themselves, but are limited to floating point number values. The ladder provides a way of sending any data type into the kernel, but can only send one value per block, due to the lock-free[2] nature of scheduling an update method to inject data into the kernel precisely before/after the kernel runs its own processing function on an audio block.

These objects provided from DSPGraph offer a seamless audio processing and scheduling system, when it comes to executing processing jobs. However, unless your audio system is static and hardcoded into the environment, like many video games would implement an audio system, there is no abstracted way of interfacing with DSPGraph from

[2] Serious issues arise when data gets simultaneously overwritten while another part of the program is reading that same data. In a multithreaded audio workload, any samples written using data read while being written to will be chaotic and result in terrible audio artifacts in the best case scenario, and the program will crash in the worst. Because of this multithreaded consideration, IAudioKernelUpdates are the best way to send data updates into the IAudioKernel.

the 3D virtual game environment. A game designer might program unique game objects that add sound processing to the graph, like a body of water that filters out high frequencies when the player is swimming, but the player has no way to interact with the sound of the water. DSPGraph does not provide a framework to edit the graph outside of code. With very limited interface capabilities, this is where DSPGraph hit its limits for providing an effective way to build an audio processing graph in real time. The only true innovation of this project is building a system that allows a user to interact with DSPGraph on a fundamental real time level, where the entire audio processing graph is continuously sculpted during runtime using intuitive 3D objects.

IAudioKernels are meant to be implemented by a developer. Their execute function is customized for the specific type of processing that effect does. This is the window into Unity's Audio System that allows the manipulation of sound. When making an audio effect for Fields, IAudioKernels hold the processing, and pair together with AudioProcessors and IAudioKernelUpdates to make a complete running effect that can be used in the Fields Virtual Environment.

Touching The Execute Function

User Input

Considering the solution of continuous manipulation of the audio processing graph, the problem can be narrowed down to finding a path between user input and the IAudioKernel's Execute Function.

User Input is handled by OpenXR, an open standard for communicating with a multitude of VR headsets. Once integrated into Unity, OpenXR offers scripts that can pull tracking data from a headset, such as Δx, Δy, and Δz coordinates of a controller, from a base point in the space the headset is in (the center of the floor in a room). Using rotation and position data from components of the headset, a virtual object can be simulated in place of the headset (head) and controllers (hands). With this type of user input data, all other traditional forms of user input can be simulated.

For example, to a computer, a button is a simple switch. The computer measures some voltage differential from the cable, and software translates that voltage differential over time into commands that tell the computer when an action was performed by the person pressing the button. In a virtual 3D environment, the physical switch that altered the circuit read from the computer does not exist, and the entire circuit can be bypassed. Instead, position values of a virtual finger can intersect with the bounding coordinates of a virtual button. When this occurs, a virtual collision can be measured, and any command can be executed from there on.

An interesting example of simulating input in Fields, is the knob. The knob requires a virtual collision with the cursor to activate, and then reads the relative z-axis rotation value between the cursor and the knob. While the collision has not been broken, the initial

rotation is used to measure further deviations from this rotation, $\Delta\Theta$, by the cursor. The knob then translates by $\Delta\Theta$, and rests at a final angle Θ, which is linearly mapped to a value v where v_{min} and v_{max} correlate to some Θ_{min} and Θ_{max}. User Input has just been successfully translated into some useful value that an audio effect could take advantage of, such as volume, filter frequency, or delay time.

The Fields Library offers an abstract ParameterController object that facilitates the communication of the parameter value into an object on the audio processing graph. A custom implementation of this object, such as the Knob, must then create its own collision system with the cursor, and its own mapping function to arrive at a value from user input. Then it sends its value through the ParameterController's ApplyValue function and the parameter is sent off. To work with the loading system of Fields, the controller must work in reverse, and map a given value to a specific configuration of the controller, and can use the OnValueUpdate() function to trigger that process.

AudioProcessor and ProcessorBundle

Following the story of our data, after measured from the user, and translated into a value the audio processing algorithm can use, the data must get into the graph, and into a specific IAudioKernel. This is where most of the work of this project comes into the picture, and is the core technology created for Fields. The AudioProcessor manages an IAudioKernel. It turns the kernel into a DSPNode, adds it to the DSPGraph, and handles all of the node's connections, parameters, and data. DSPGraph is extremely particular about how the graph is constructed. The particularities are a non-issue when building a static graph, and for Unity's mission, video game levels can always run a baked-in graph that only worries about turning nodes on and off dynamically. However, when the product is the graph, and it needs to be malleable and constructable in real time, their particularities become problematic and require a proper management system.

A big reason DSPGraph can be problematic when editing the graph in real time, is that the edges of the graph are stored in indexed adjacency lists from the node, and if a node has multiple output edges, but an edge with a small index is removed, the edges with higher indexes do not get scheduled.

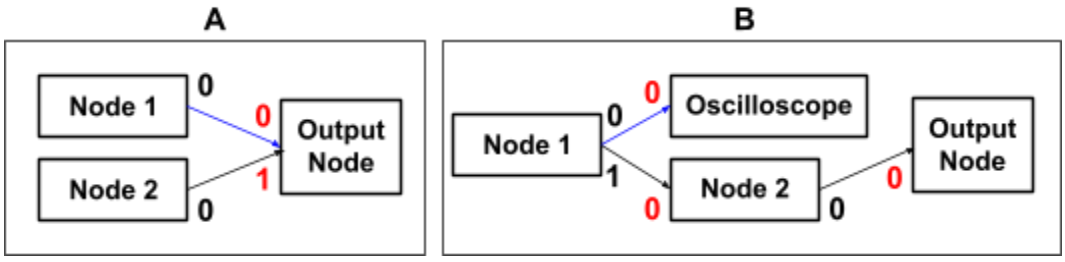


Figure 10: Showing a particular aspect of DSPGraph, where if any of the blue edges in the graph are removed, the audio processing of the entire graph halts despite other nodes still being connected to the output.

As shown in Figure 10A, if Node 1 is disconnected from the output, the graph freezes. When the output node is taking in inputs, it loops through its input edges in order of index, and if an edge is missing, the processing halts. This could be an issue the developers are still working on, or maybe it is low priority since it does not impact statically built graphs. DSPGraph is still an experimental package after all and is considered to be abandoned by many users on the Unity forums. This odd effect is also seen in Figure 10B, where Node 1 stops sending any audio to Node2 if disconnected from the oscilloscope, even though logically the audio should still be processed.

To fix this, it seems important to keep track of a Node's number of inputs and outputs, and get the index value of each edge. However, DSPGraph provides no method of getting any information about a DSPNode. Because of this, keeping a separate copy of the entire graph is necessary for keeping track of the graph's information. Therefore, each AudioProcessor contains lists of both its inputs and outputs, providing an adjacency list representation of the graph. There are also special methods of handling connections between the nodes, that sort connections by index and keep track of filling gap indexes as they are created. Solutions for managing DSPGraphs odd requirements have essentially recreated the graph part of DSPGraph, and could nearly be a complete standalone system if a multithreaded scheduling algorithm was implemented as well.

Another aspect of DSPGraph is to deallocate the memory of any IAudioKernel when it is removed from the graph. This makes sense for efficiently allocating memory in a game environment. However this causes some issues in a modular music production environment. Therefore, for Fields, the data stored in an IAudioKernel is ultimately just a reference to manually allocated and managed memory kept in the AudioProcessor. This is

to protect data from the behavior of DSPGraph, where data is deallocated if a node is not affecting the output signal of the graph. This would require data stored in the IAudioKernel to be reallocated and initialized every time its node is disconnected and reconnected from the graph. A good example of why this behavior is unwanted is a delay effect with some feedback. A very common audio effect, a delay with a high feedback, relies on an internal buffer of audio samples that it stores, and recalls many processed samples later. If this effect were disconnected from the graph to be reconnected somewhere else, it would be expected to run while disconnected, and maintain the sample data in its buffer. Reconnecting the delay to hear corrupt samples would surely surprise any user.

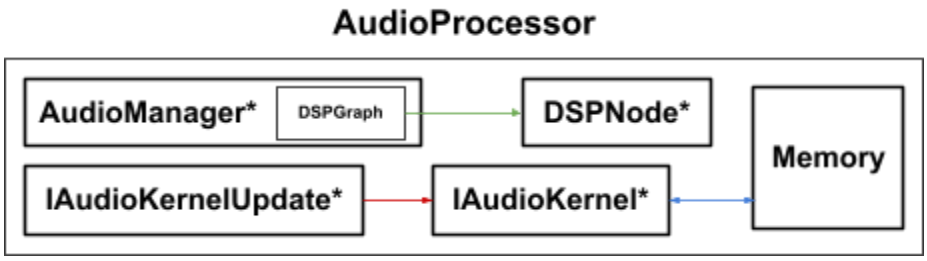


Figure 11: The AudioProcessor class holds a reference to the AudioManager, which it uses to access the DSPGraph. It also holds references to a unique DSPNode, IAudioKernel, and multiple IAudioKernelUpdates. The AudioProcessor also holds and manages any necessary memory used by the IAudioKernel. The green arrow represents functions of the AudioProcessor class which manipulate the graph, such as connecting and disconnecting the node to and from other nodes. The red arrow represents publicly accessible functions of the AudioProcessor class which utilize IAudioKernelUpdate structs to send or receive data to and from the IAudioKernel. Finally, the blue arrow represents a link via reference of the memory that the IAudioKernel holds, which is initialized via one of the IAudioKernelUpdates.

Besides working around DSPGraph to make the graph editable in real time, the AudioProcessor also provides an abstract framework for any ParameterController to send its value to the IAudioKernel. Sending sample accurate parameter floats that run directly through the execute function of an IAudioKernel must be done using the DSPGraph itself.

So, each AudioProcessor holds a reference to the DSPGraph, and any ParameterController connected to the AudioProcessor can use the AudioProcessor's UpdateParameter method to send data directly into the IAudioKernel. The AudioProcessor can also contain implementations of IAudioKernelUpdates to send any data type into the IAudioKernel. Since each IAudioKernel has a unique processing algorithm, and unique parameters and data, each AudioProcessor can contain unique methods to use IAudioKernelUpdates for specific cases. For example, the MultiVoiceSynth, part of the default processors in Fields, contains special update functions that activate and deactivate different notes for the synth. That is why when the user hits a note on the keyboard, where each key is a data type that implements ParameterController, the synth is able to understand a note is being played when it goes to generate a waveform inside its execute function.

So the story goes, from a musicians mind to a sound coming out of the speaker, the data flows through the headset, OpenXR, Unity's environment coordinates, the ParameterController, the AudioProcessor, DSPGraph, the IAudioKernel, then finally, the output device, the air, and the musician's ears. However, this is a bit simplified, in nearly all areas of the system. It is useful however, to better understand Fields, to understand the role of the ProcessorBundle in all this.

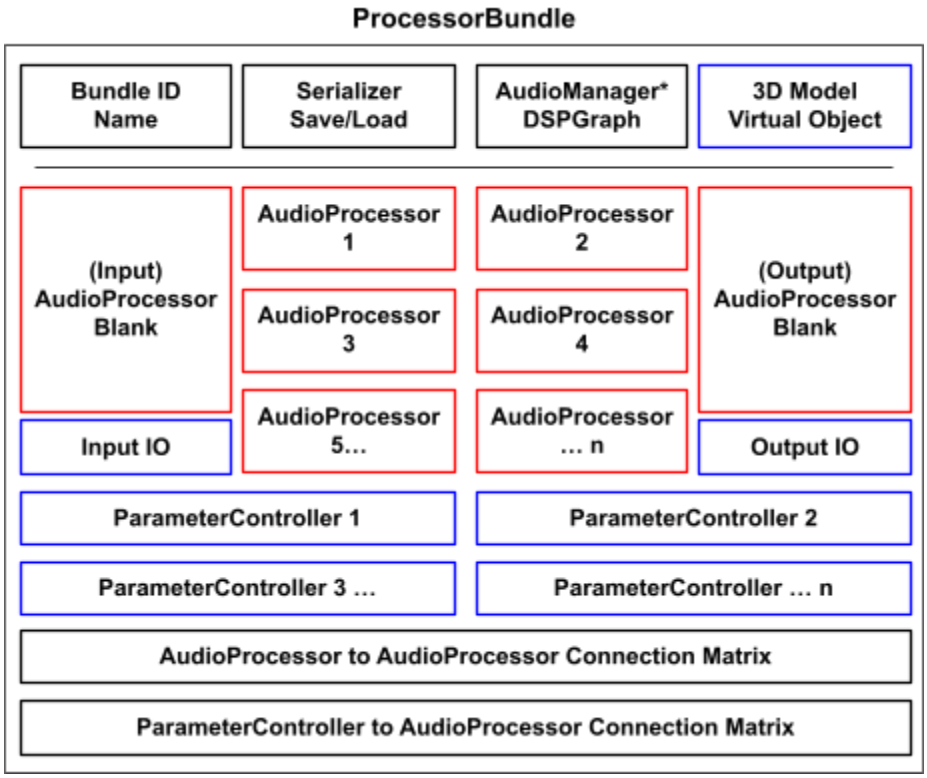


Figure 12: The ProcessorBundle is a packaged audio effect that is interactable in the virtual environment. It is also fully serializable and able to be loaded from binary files or online servers (online functionality not implemented as of yet) via the appropriate ID. In red, the ProcessorBundle contains an arbitrary number of AudioProcessor which represent a subset of the processing graph. The ProcessorBundle contains its own connection matrix configurable by a developer to build signal processing architecture for the overall audio effect. In blue, are devices represented in the virtual environment, where the ParameterControllers also have a configurable connection matrix for which processors they update. The IO ports are the same from Figure 5.1.

The ProcessorBundle encapsulates a subgraph of audio processing for the end user to interact with as a single virtual object with a unique set of custom controls. In digestion, Figure 12 shows a ProcessorBundle and its internal components. The bundle packages together an arrangement of AudioProcessors and ParameterControllers into one object, a condensed set of processing nodes with one input and one output. The user is then able to see this object in the virtual environment with a name, 3D body, IO, and controls (ex. Filter

in Figure 13, Synthesizer in Figure 5). The ProcessorBundle is the object that gets loaded into the world when the player selects an effect from the wall, and this is the object that gets serialized into a save file, to be respawned and reconfigured when loading a save file. It is also the object that can be packaged and sent to other people to use in their world when a developer creates a custom audio effect, although a system to import custom effects into Fields is not yet complete, and is one of the things planned for a future version of Fields.

Figure 13: The Default Package Filter. This is an effect inside Fields, the result of a completed ProcessorBundle applied to a 3D model inside Unity. There are 4 Knobs from the Default Package, implementations of ParameterControllers, placed onto the 3D model, as well as two IO objects behaving as Input and Output ports, left and right respectively. Once a developer has created an AudioProcessor along with its respective IAudioKernel, one which filters frequencies for example, the developer can apply their AudioProcessor script to the AudioBundle inside Unity, and fill out the connection matrices for the AudioProcessors and ParameterControllers such that the Knobs, in this case, point to the appropriate AudioProcessor.

Developer Guide

Building a ProcessorBundle

Fields is an open and expandable system that is intended for users/developers to build their own effects. Not only is the software a place for musicians to experiment with sound, but a place for developers to use as a sandbox for creating their own musical tools. To package together a unique audio effect for Fields, a developer must construct a ProcessorBundle. Figure 14 shows how a ProcessorBundle is configured on a 3D object within Unity. To convert a 3D model of an audio effect into a ProcessorBundle, just add "ProcessorBundle.cs" from the library onto the object in Unity. This will show an AudioBundle component on the object in the Unity Inspector shown in Figure 14.

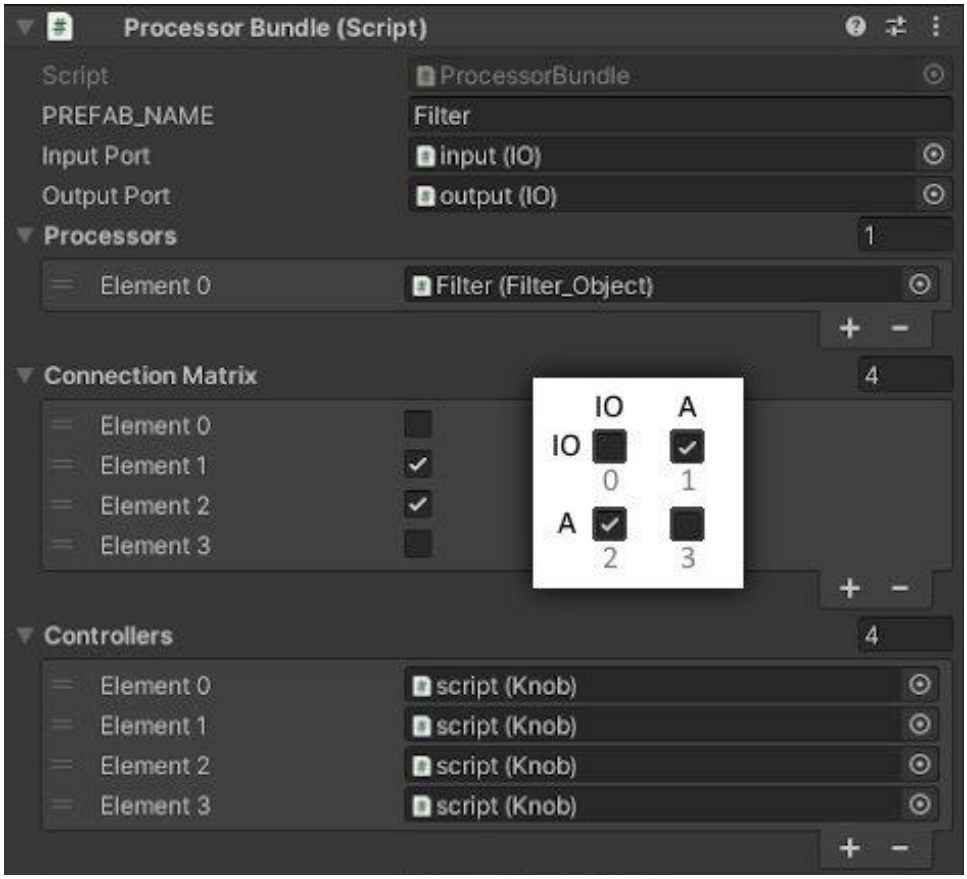


Figure 14: The implementation of the Filter ProcessorBundle of Figure 13 inside Unity's Inspector. The AudioProcessor scripts are 'drag and dropped' into the "Processors" list once fully implemented. Similarly are the ParameterController scripts placed into the "Controllers" List. Finally, the

> “Connection Matrix” is filled out according to the desired internal connections. Element 0 is the IO port, where IO -> A treats IO as the input, and A -> IO treats IO as the output. In this example, the simplest example, the ProcessorBundle only contains one AudioProcessor, which takes in audio via the input, and sends audio out via the output.

To start, each ProcessorBundle needs a name, which will be used as a unique identifier for Fields to recognize the final object. This name will be entered into the top text field in Figure 14. As of now, conflicts will arise if the name is the same as any other ProcessorBundle in Fields, but adapting the system to combine the individual name with the name of the package can fix this issue in the future:

ex. “Fields.Default.Filter” != “DeveloperName.PackageName.Filter”

The next thing each ProcessorBundle needs is IO. IO orbs from Figure 5.1 are available from the Fields Library as a Unity Prefab that can be dragged into the scene and placed appropriately aside the 3D model of the ProcessorBundle. Then, they can be dragged into the ProcessorBundle’s inspector as in Figure 14.

To get the ProcessorBundle to actually process audio, custom implementations of AudioProcessors must be added onto the ProcessorBundle object. For example, the “Fields.Default.Filter” AudioBundle in the Default Package that comes with the Fields Library, there is a “Filter_Object.cs” script which implements AudioProcessor and IAudioKernel specifically for a filter effect. This script is added to the same object the AudioBundle is on, or a child object for more organization, and then added to the Audio Bundles “Processors” list in the Unity Inspector seen in Figure 14. Once the AudioProcessors are added into the ProcessorBundle, the “Connection Matrix” must be filled out according to the desired internal signal flow of the audio effect. Figure 14 explains

the formatting of the connection matrix, and shows a 2D reconstruction of the 1D list of boxes shown in the Unity Inspector.

Finally, the ProcessorBundle needs a way of interacting with the processors, unless of course the desired custom effect takes in no parameters and is a mysterious black box of static processing. To add ParameterControllers to the ProcessorBundle, place the specific controllers, such as the “Fields.Default.Knob” implementation of ParameterController, onto the 3D model of the audio effect wherever seems fit, then add them to the “Controllers” list in the Unity Inspector shown in Figure 14. Then, simply configure the connection matrix to send each ParameterControllers value to their respective AudioProcessor.

Once the entire ProcessorBundle script is filled out accordingly in the Inspector, it will be able to process audio running on DSPGraph, and show up as an interactable game object inside Fields. Currently unless a custom version of the Fields Application is built including the new ProcessorBundle, there is no way to use it. This is an important next step for the project, and the goal is to provide an online server to store and load Unity’s Asset Bundles, which are specifically designed by Unity to load dynamic assets into a game during runtime.

Building an IAudioKernel

To fully develop an AudioProcessor to integrate into a ProcessorBundle's signal architecture, all a developer needs to do is construct the three main components of a Fields Audio Effect: AudioProcessor, IAudioKernel, IAudioKernelUpdate, and then link their AudioProcessor to their ProcessorBundle. An example of how to do so exists in the library here: Fields.Examples.AudioEffect. The code was written as an instructional template, intended to be used as a skeleton needing to be filled out, so diving into the file might be more useful to a more hands-on developer than reading this section. Nevertheless, the first thing a developer must do to start writing software objects for Fields is to import the proper dependencies (shown in Figure 15).

```
//Include Package
using Fields.Audio;
//Include DSPGraph from Unity, then Include Unity packages.
using Unity.Audio;
using Unity.Collections;
using Unity.Collections.LowLevel.Unsafe;
using Unity.Burst;
```

Figure 15: Other than the default imports that initialize when creating a new C# Unity script, these are the extra packages required to build an AudioProcessor and IAudioKernel.

After importing, it can be easiest to build the IAudioKernel first, since the AudioProcessor will rely on that struct. It is also probably best to build the actually signal processing algorithm first, so it is understood what data storage and update functions

are needed to be implemented in the AudioProcessor. To start building an IAudioKernel, create a new struct, and inherit from the IAudioKernel generic as shown in Figure 16.

```
[BurstCompile(CompileSynchronously = true)]
public struct ExampleAudioProcessor_Kernel :
    IAudioKernel<
        ExampleAudioProcessor_Kernel.Parameters,
        ExampleAudioProcessor_Kernel.Providers
    >
```

Figure 16: The construction of an IAudioKernel class, flagged with a BurstCompile tag to tell Unity that this struct executes on the multithreaded Unity Jobs system.

As shown in Figure 16, the IAudioKernel generic takes in TypeNames of Parameters, and Providers. These are just enum data types which DSPGraph uses to organize floating point arrays of either sample or parameter data. They can be initialized externally and referenced here, or for simplicity, can be constructed inside the IAudioKernel struct itself as shown in Figure 17.

```
public enum Parameters
{
    [ParameterDefault(1.0f)]
    [ParameterRange(0.0f, 1.2f)]
    inputGain,
}
```

Figure 17: ExampleAudioProcessor_Kernel.Parameters

The Parameters enum contains meta-data about the different floating point parameters you will use for the audio processing inside the IAudioKernel. For example, to adjust the gain, or volume, of sound, a gain parameter can be set up here, and will automatically be readable inside the IAudioKernal, and automatically be writable from a ParameterController on the ProcessorBundle if its connection matrix points to the index of this parameter on the AudioProcessor this IAudioKernel is attached to.

As well as parameters, the IAudioKernel inheritance also requires the use of several public void methods within the struct: Initialize(), Dispose(), and Execute(ExecuteContext<Parameters, Providers> context). In a traditional use of

DSPGraph, the Initialize() and Dispose() functions would be used for memory management, such as allocating and releasing arrays, but for persistence that will be done in the AudioProcessor. References of any data that needs persistent allocation must be created inside the IAudioKernel however, and would look like Figure 18.

```
//Do not initialize this here,
//this should be set to reference external data
public NativeArray<float> persistentBuffer;
```

Figure 18: Array in IAudioKernel that is never initialized, but rather set to reference an external array from the AudioProcessor.

This will become more clear in the sections about using IAudioKernelUpdates inside the AudioProcessor.

The Execute function is where the actual processing will take place, and where the IAudioKernel will have access to the audio buffers that are sent in from inputs, as well as to the empty audio buffers that will be sent to the outputs. Shown in Figure 19, the Execute function takes in an ExecuteContext reference. This ExecuteContext is passed through each IAudioKernel as scheduled in Figure 9, by a Fields.Audio.AudioDriver which implements DSPGraphs IAudioOutput. This AudioDriver is managed with the DSPGraph itself in the AudioManager shown in Figure 1.

```
public void Execute(ref ExecuteContext<Parameters, Providers> context)
{

    int numOutputs = context.Outputs.Count;
    //If no outputs then do nothing, this helps protect accessing context.Outputs.GetSampleBuffer(0)
    //If you want continuous processing even when disconnected, another way is possible
    if (numOutputs == 0) return;

    int numInputs = context.Inputs.Count;
    int blockSize = context.DSPBufferSize;
    int sampleRate = context.SampleRate;

    SampleBuffer outputBuffer = context.Outputs.GetSampleBuffer(0);
    int numChannels = context.Outputs.GetSampleBuffer(0).Channels;
    int numSamples = context.Outputs.GetSampleBuffer(0).GetBuffer(0).Length;
```

Figure 19: The execute function inside an IAudioKernel, along with a demonstration of pulling important information from the ExecuteContext parameter.

When writing an algorithm inside the Execute function, the ExecuteContext object provides everything, from the incoming audio signal and the output buffers, to important system information about the audio settings of the machine running the algorithm, such as the sample rate and block size. The number of samples will most likely range from 128 to 1024, but can also be as low as 16 samples per block. This is important when considering algorithms that operate in the time domain, and holding a larger buffer in memory such as the persistentBuffer in Figure 18, can allow access to samples that were in previous blocks, a history with variable length. As for the number of channels, Fields currently runs in stereo, so the number of audio channels per buffer will always be two, however there are plans to offer a multi-channel spatial audio experience that accounts for the 3D positions of virtual sound sources. Most likely, this spatial audio will be simulated from tracking the head position relative to a set of mono audio sources placed around the 3D environment, so the algorithms themselves could be written in mono, via duplication to the second channel. Then, custom ParameterControllers could provide sophisticated ways of passing parameters to the individual mono AudioProcessors that provide a spatial audio experience. So, the point is, writing sophisticated multi-channel algorithms within a single Execute function will most likely be unsupported for a while, and sticking to two channels, for either a mono or stereo experience, is best practice.

After all the meta-data of the sound is pulled from the ExecuteContext, you are ready to start peeking at samples from the inputs. The most common thing to do with the input samples, is to sum them all together so that the algorithm operates on everything fed into

the AudioProcessor. Figure 20 shows an example of mixing the audio inputs into a single buffer, while also showing the use of a parameter read from the ExecuteContext, written to by a ParameterController through the AudioProcessor. Parameters in this context are float arrays that take a sample offset as an input, which can be configured with a sample delta for linearly interpolating between an old value and a new value being written. This is useful for changing a parameter such as the volume of a signal, because any change in amplitude quicker than roughly fifty milliseconds will result in a burst of an audible frequency. For example, a common sample rate of audio is 44100 Hz, which can represent frequencies up to 22050 Hz, just above the accepted high frequency range of human hearing. This means a 20 Hz wave, the slowest wave audible by humans, makes one revolution in 2205 samples. Any drastic change in volume in less samples will produce sound in the audible range of human hearing, so it is important to interpolate between written volume values with a distance of at least 2205 samples.

```
//Mix inputs into output
for (int channel = 0; channel < numChannels; channel++)
{
    NativeArray<float> outputChannel = outputBuffer.GetBuffer(channel);
    for (int input = 0; input < numInputs; input++)
    {
        NativeArray<float> inputChannel = context.Inputs.GetSampleBuffer(input).GetBuffer(channel);
        for (int sample = 0; sample < numSamples; sample++)
        {
            //Load parameter values inside sample loop for single sample precision
            float inputGain = context.Parameters.GetFloat(Parameters.inputGain, sample);

            //Mix inputs into output
            outputChannel[sample] += inputChannel[sample] * inputGain;
        }
    }
}
```

Figure 20: An example of reading input buffers in the Execute function of an IAudioKernel. The first output buffer is being used as a placeholder for the samples before they are processed.

Finally we now have all our audio in a single buffer, ready to be processed. This is where creative bounds dissolve, and sonic ideas are converted into code. There are many traditional digital signal processing algorithms that are inherently musical, such as filtering or frequency shifting, while there are also algorithms that only make sense in a musical context, such as reverb or randomized granular modulation. Figure 21 shows a block of code that loops through all the audio samples in a buffer, but does nothing to the sound. One of the simplest algorithms that can be implemented here is a saturation effect. This effect is static, and can be implemented in one line of code. Running the samples through a sigmoid function such as `tanh()`, will add harmonics in the frequency domain of the sound, by applying a non-linear transformation on the amplitude domain, shown in Figure 22. Since sigmoid functions have a limited range, a scalar can be combined inside the function to increase the amount of non-linear distortion, without causing the output to be greater than one, which is the maximum value for digital audio signals. This would look like:

```
outputChannel[sample] = tanh(drive * outputChannel[sample]);
```

```
//Process Samples
for (int channel = 0; channel < numChannels; channel++)
{
    NativeArray<float> outputChannel = outputBuffer.GetBuffer(channel);
    for (int sample = 0; sample < numSamples; sample++)
    {
        //Do something to this sample
        outputChannel[sample] = outputChannel[sample];
    }
}
```

Figure 21: Block of code looping through every audio sample of the first output buffer, which now holds the combined signal from all of the input buffers combined, from the loop in Figure 20. This provides processing access to that combined signal.

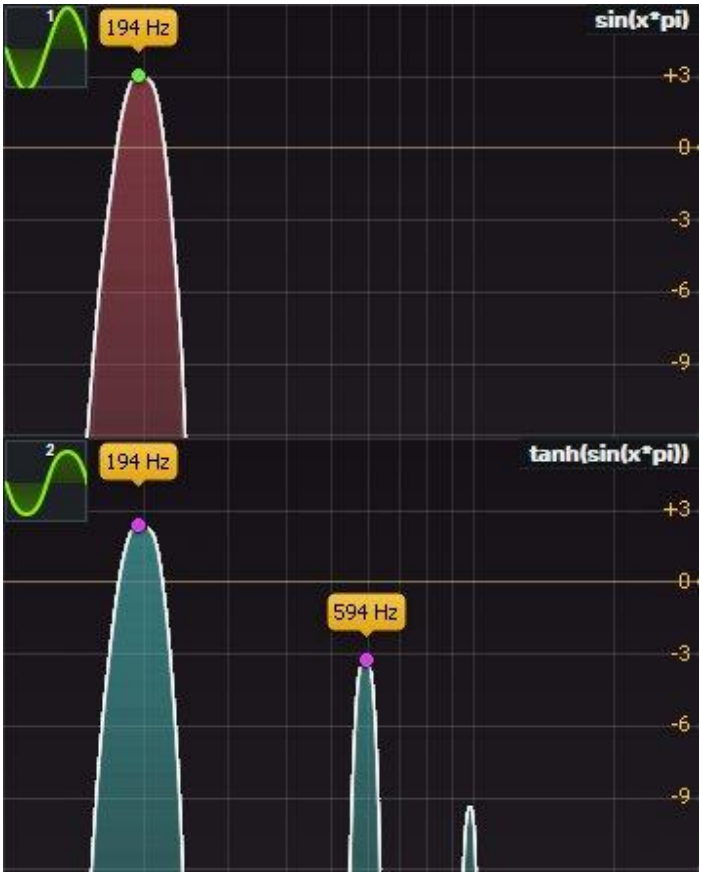


Figure 22: X-axis is frequency from 100 Hz to 3000Hz logarithmic scale, while the Y-axis is from -12 dB to +6 dB linear. A sine wave of 194 Hz was processed through a tanh() function, and the waveform was squashed. This produced a harmonic frequency at 594 Hz, roughly 3 times the frequency and 6 db lower amplitude of the initial sine wave. A harmonic 5 times higher frequency was also introduced, and theoretically, an infinite series of odd multiple harmonics were generated.

Once satisfied with the sound of an algorithm, the final step is to ensure the new signal is sent to all of the AudioProcessor's outputs. Similar to merging all the input buffers into a single buffer for processing, the goal now is to do the inverse, where what is in the first output buffer must be copied into the rest, shown in Figure 23. Sending the processed signal to all of the outputs of the processing node is the final step of any processing algorithm for Fields, and the end of the Execute function.

```
//Send to all outputs
for (int channel = 0; channel < numChannels; channel++)
{
    NativeArray<float> outputSourceChannel = outputBuffer.GetBuffer(channel);
    //Output starts at 1 since output 0 is the source
    for (int outputDest = 1; outputDest < numOutputs; outputDest++)
    {
        NativeArray<float> outputDestChannel = context.Outputs.GetSampleBuffer(outputDest).GetBuffer(channel);
        for (int sample = 0; sample < numSamples; sample++)
        {
            //Mix into output
            outputDestChannel[sample] += outputSourceChannel[sample];
        }
    }
}
```

Figure 23: An example of copying the processed output buffer the the rest of the output buffers. Note that the output loop starts at 1 instead of 0, since the first buffer is the one copied from, and not a destination.

Building an AudioProcessor

Now that the algorithm is implemented, and the IAudioKernel is complete, the developer can build a custom AudioProcessor class (construction pictured in Figure 24). This class represents a modular audio processing unit that can be added into any configuration inside a ProcessorBundle. It also declares which IAudioKernel, processing algorithm on the graph, is being used for this particular effect. To link an IAudioKernel to the AudioProcessor, simply use the TypeName of the IAudioKernel in the implementation of the AudioProcessor (shown in Figure 24).

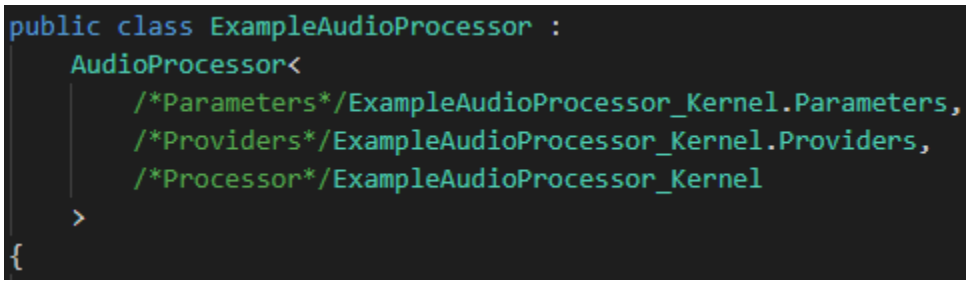


Figure 24: This is the construction of an AudioProcessor class, which takes 3 TypeNames as arguments. The types must be of the parameter enum, provider enum, and the IAudioKernel itself.

The typing of the AudioProcessor itself, provides types for some of its vital methods. The initialization method uses the type of the IAudioKernel to instantiate a DSPNode object on the DSPGraph. The method that writes parameters to the IAudioKernel uses the type of the parameters enum to convert indices from a connection matrix into parameter types that feed directly into DSPGraph, along with a value, and DSPNode. This generic abstract AudioProcessor class that is inherited to make a custom AudioProcessor, itself inherits an abstract AudioProcessor without a custom typing, so all AudioProcessors can be of the same type, and relate to other objects in Fields, all in the same ways. This is how the custom implementation of an AudioProcessor that a developer creates, can easily integrate into the

rest of Fields, because on a lower level, it is the same object as every other AudioProcessor, and is packaged into a ProcessorBundle all the same. This style of a generic abstract class inheriting a sublevel abstract class is the core principle driving the modularity of Fields development. This is also how ParametersControllers work, and provides the foundation for any custom controller device to be easily implemented into the rest of the system architecture of Fields.

Once an AudioProcessor is constructed with specific types, and therefore bound together with the processing algorithm built inside the IAudioKernel type, a developers next step is to construct any persistent memory the IAudioKernel needs to access, then feed it into the kernel itself through reference. Shown in Figure 25, a NativeArray of floats is constructed. The NativeArray is the data type used in Unity's Job System for allocating and accessing unmanaged memory in C#, with a simple protocol to track memory leaks. The NativeArray can be constructed with an Allocator, which is a flag to specify a minimum period for how long the memory needs to stay allocated. For an AudioProcessor, if the data needing to be stored should remain throughout multiple connections and disconnections from the graph, or even just for multiple audio blocks, then it is important to use the Allocator.Persistent flag.

Once allocated, any memory needs to be sent into the IAudioKernel, so it can be used by the audio processing algorithm. Besides sending a Parameter float into the ExecuteContext, there is only one other way to communicate with an IAudioKernel, by using an IAudioKernelUpdate. An IAudioKernelUpdate is a struct with an Update function, that similarly to the IAudioKernel's Execute function, gets scheduled by the same AudioDriver that schedules the audio processing jobs for DSPGraph. To prevent any read tears, the

AudioDriver schedules the IAudioKernelUpdate to interact with a specific IAudioKernel, precisely when the IAudioKernel is not running its Execute job. Figure 25 shows the use of an IAudioKernelUpdate, the ExampleAudioProcessor_MemoryAllocator which inherits from IAudioKernelUpdate, constructed with a reference to the persistentBuffer, and scheduled using a Command Block generated from the DPSGraph. All IAudioKernelUpdates are executed the same way, using a command block generated as such (Figure 25). A 'MemoryAllocator' is not native to DSPGraph or Fields, and is not a requirement for an AudioProcessor, but is a recommended use of an IAudioKernelUpdate. If a persistent array of data is vital to a processing algorithm, then an implementation of an IAudioKernelUpdate must exist that sends references of the data into an IAudioKernel.

Finally, once the reference is sent, and the IAudioKernel is accessing externally managed memory, it is important to dispose of the memory after its allocation is no longer needed. Figure 25 shows the use of the Dispose function to dispose of any unmanaged memory that has been allocated. If this step is not completed, then memory will be leaked every time an instance of the AudioProcessor is deleted from the virtual environment within Fields.

```
NativeArray<float> persistentBuffer;
bool trigger;

public override void Initialize()
{
    //the size of this buffer can be anything needed
    //to access the block size, use "audioManager.audioGraph.DSPBufferSize"
    persistentBuffer = new NativeArray<float>(
        audioManager.audioGraph.DSPBufferSize, Allocator.Persistent);

    //Send reference to IAudioKernel
    using (var block = audioManager.audioGraph.CreateCommandBlock())
    {
        block.UpdateAudioKernel<
        ExampleAudioProcessor_MemoryAllocator,
        ExampleAudioProcessor_Kernel.Parameters,
        ExampleAudioProcessor_Kernel.Providers,
        ExampleAudioProcessor_Kernel
        >(new ExampleAudioProcessor_MemoryAllocator(ref persistentBuffer), node);
    }
}
public override void Dispose()
{
    //must manually deallocate memory when using Allocator.Persistent
    if (persistentBuffer.IsCreated) persistentBuffer.Dispose();
}
```

Figure 25: Example of constructing persistent memory inside an AudioProcessor class. The Initialize() and Dispose() functions are also present, showing how to initialize Unity's NativeArray with a persistent memory allocation flag, send the reference of the array to the IAudioKernel via an IAudioKernelUpdate, and finally deallocate the array when the AudioProcessor is destroyed.

Building an IAudioKernelUpdate

Once the IAudioKernel and AudioProcessor are inherited into custom classes and constructed, At least one IAudioKernelUpdate is required if managed memory is needed, and additional IAudioKernelUpdates are needed for data communication other than the floating point parameter buffers in the ExecuteContext. One common use of an IAudioKernelUpdate, other than a memory allocator, is to pass trigger events to an IAudioKernel. Such an event might be a virtual midi key being pressed, where an event containing a variety of information must be passed into the IAudioKernel, and sub-block sample accuracy is not important. Figure 26 shows an example of an IAudioKernelUpdate being used to set a NativeArray inside an IAudioKernel, to reference a NativeArray managed from the AudioProcessor. Figure 27, shows an example of sending a trigger event to the IAudioKernel, which can contain any number of parameters of any unmanaged data type[3].

Once constructed, to use the IAudioKernelUpdate, a public function, accessible by a ParameterController, can be made inside the AudioProcessor, which utilizes a command block UpdateAudioKernel function. Figure 28 shows a function inside an AudioProcessor which can be called from a ParameterController, such as a virtual midi note, which sends its trigger state, note value, and velocity information into the IAudioKernel.

[3] According to Microsoft's C# Documentation, a type is an unmanaged type if it's any of the following types:

- sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, or bool
- Any enum type
- Any pointer type
- Any user-defined struct type that contains fields of unmanaged types only and, in C# 7.3 and earlier, is not a constructed type (a type that includes at least one type argument)

```
public struct ExampleAudioProcessor_MemoryAllocator : IAudioKernelUpdate<
    ExampleAudioProcessor_Kernel.Parameters,
    ExampleAudioProcessor_Kernel.Providers,
    ExampleAudioProcessor_Kernel
> {
    NativeArray<float> _persistentBuffer;

    public ExampleAudioProcessor_MemoryAllocator(ref NativeArray<float> persistentBuffer)
    {
        _persistentBuffer = persistentBuffer;

    }
    public void Update(ref ExampleAudioProcessor_Kernel kernel)
    {
        kernel.persistentBuffer = _persistentBuffer;
    }
}
```

Figure 26: An IAudioKernelUpdate struct used to send a reference of persistent memory managed in an AudioProcessor, to an IAudioKernel

```
public struct ExampleAudioProcessor_UpdateTrigger : IAudioKernelUpdate<
    ExampleAudioProcessor_Kernel.Parameters,
    ExampleAudioProcessor_Kernel.Providers,
    ExampleAudioProcessor_Kernel
> {
    bool _trigger;
    int _note;
    float _velocityl

    public ExampleAudioProcessor_UpdateTrigger(bool trigger, int note, float velocity)
    {
        _trigger = trigger;
        _note = note;
        _velocity = velocity;
    }
    public void Update(ref ExampleAudioProcessor_Kernel kernel)
    {
        kernel.trigger = _trigger;
        kernel.note = _note;
        kernel.velocity = _velocity;
    }
}
```

Figure 27: An IAudioKernelUpdate struct used to send information into an IAudioKernel in between process blocks.

```
//To change data in IAudioKernel other than the parameters,
//use IAudioKernelUpdate struct in custom public function that can be called externally
public void DoSomething(bool trigger, int note, float value)
{
    using (var block = audioManager.audioGraph.CreateCommandBlock())
    {
        block.UpdateAudioKernel<
        ExampleAudioProcessor_UpdateTrigger,
        ExampleAudioProcessor_Kernel.Parameters,
        ExampleAudioProcessor_Kernel.Providers,
        ExampleAudioProcessor_Kernel
        >(
            new ExampleAudioProcessor_UpdateTrigger(trigger, note, value),
            node
        );
    }
}
```

Figure 28: A public function inside an AudioProcessor that schedules an IAudioKernelUpdate to inject data into an IAudioKernel. This function is accessible to a custom implementation of a ParameterController, to allow a custom interactable object in the virtual environment to interact with the processing algorithm of an IAudioKernel.

Building a ParameterController (in development)

The default package of Fields includes a set of ParameterControllers such as knobs, collision pads, and a virtual midi controller, so any traditional effect or synthesizer can be created without creating new ParameterControllers. Although a ParameterController is intended to be as easily customizable as an AudioProcessor, the ParameterController was not able to be fully abstracted in the work of this project. Right now, the ParameterController class can be inherited to create a single value parameter setter, such as a knob or slider, and only supports sending a parameter update containing a single float value. However, trigger based colliders such as a drum pad, inherent from CollisionPad, which contains trigger on and off functions that can be set with lambdas, to execute any custom functionality when colliding with the object. Figure 29 shows a MidiKey CollisionPad, which sends a note value to an AudioProcessor when collided with.

```
public class MidiKey : CollisionPad<AudioProcessor>
{
    public MusicalEnums.NoteLetter note;
    public MusicalEnums.Accent accent;
    public MusicalEnums.Octave octave;

    public void Start()
    {
        Initialize();
        SetFunctions(
            processor => { processor.ActivateNote(getValue(), 1); },
            processor => { processor.DeactivateNote(getValue()); });
    }

    private int getValue()
    {
        return (int)note + (12 * (int)octave) + (int)accent;
    }
}
```

Figure 29: A CollisionPad implementation which takes advantage of SetFunctions, which can call public AudioProcessorFunctions such as the one in Figure 28. This allows an action in the virtual environment to interact with an AudioProcessor in a complex way.

This lambda functionality is intended to be accessible in ParameterController, but it and many other specific controller functionalities are scattered throughout several different classes. The envelope in Figure 5.3 for example, runs on an Envelope script, which reads single values, such as attack or decay, from VolumePoint scripts which construct movable objects on specific axes of a volume, and map value ranges according to the size of the volume's lengths. Because the ParameterController lacks functionality crucial to some types of interaction in the virtual environment, custom controllers are not yet supported in Fields, and the knobs and collision pads in the default package are the only virtual controllers available for a ProcessorBundle at this time. It is of course possible to inherit from CollisionPad directly and make use of the custom functions, but this will be deprecated and integrated into ParameterController soon.

Making Music

Synthesizer

The culmination of Fields is using the entire system to make music. From the low level signal processing happening inside the IAudioKernel's Execute function, to the higher level network of virtual objects sending information into a ProcessorBundle's AudioProcessors, all of the elements made for this expedition into virtual reality and musical interaction come together to provide an experience unlike any musical instrument. To conclude the discussion of this work, the inner workings of a polyphonic digital synthesizer will be explored.

Like any ProcessorBundle, the Fields.Default.MultiVoiceSynth is a collection of AudioProcessors, ParameterControllers, IO, and a 3D body. During the course of this project, there have been two major iterations of the synthesizer. The first version was a bundle of 50 AudioProcessors, and had 24 voice polyphony (able to play 24 notes at once). Figure 30 shows the signal flow of that first synthesizer.

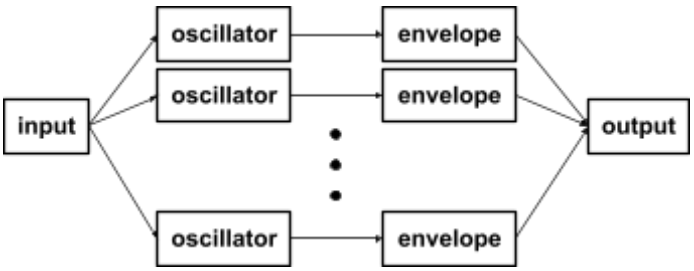


Figure 30: Subgraph inside first iteration synthesizer ProcessorBundle. There were 50 AudioProcessors, a blank input, a blank output, 24 oscillators, and 24 envelopes. Each oscillator envelope pair represented one of the 24 synthesizer voices.

One really big advantage that this architecture had was its largely parallel nature. Since the scheduling of IAudioKernels is a multithreaded workload handled by DSPGraph and Unity's Job System, parallel substructures in the graph were automatically optimized to run on multiple threads. This was the initial motivation for constructing such a synth, but in the

end, the parallel branches needed too much intercommunication, that the parameter passing became much too complex to further optimize the system. Heavily parallel, each oscillator envelope pair had very little knowledge about the state of the entire synthesizer, and this made it difficult to manage the state of each pair. For example, each branch was running all the time, and was waiting for its respective note to trigger the envelope to allow a note to play. So even though the system was optimized at full load, while not playing the synth, it was still using valuable time. On top of this, each ParameterController, such as the knobs for pitch and volume, had to update 24 different AudioProcessors at once, and expanding the capabilities of the synthesizer became taxing during development to manage the connection matrices of all the different components.

The solution, although giving up the automatic multithreading support by Unity, was to make a single AudioProcessor unit that managed all of the voices. This unified serial processing system would become slow as many notes were played at once, but when small numbers of notes were being played, the gains of a multithreaded workload diminished. Since in Fields, each arm is a single digit of control, like a big thumb with no fingers, it was difficult to play a bunch of notes at once anyway. When no notes were being played, the entire system was aware of each note's state, and processing was shut down completely. The only time this system is truly slower, is when the release value of the synthesizer is long, making notes continue long after being pressed, and more and more notes begin to play on top of each other over time. Standalone on the Oculus Quest 2, if more than 12 notes are played simultaneously with more than one voice per note, the buffer is not finished being written to before the clock times out, and clicking occurs. However, sub job scheduling is supported in Unity, and a system can be implemented to generate the waves

of each note in parallel threads within a single AudioProcessor, although this has not been implemented at this point in time.

The final synthesizer holds a reference to a list of Note structs and Envelope structs. It then loops through the notes which are active, samples their waveform with their current phase, and multiplies the amplitude by an envelope value calculated from a sample delta representing the lifetime of the note. Figure 31 shows how a sine wave is generated in the synthesizer.

```
case WaveForm.Sine:
    for (int v = 0; v < voices; v++)
        outputBuffer[s] +=
            Pan(v + 0.5f, c, voices, width)
            * math.sin(phases[v] * 2 * math.PI)
            * envelopes[n].GetValue(context.DSPClock + s);
```

```
private float Pan(float v, int c, int voices, float width)
{
    return (width * (((2 * c * v) - v + voices - (c * voices)) / voices))
           + ((1 - width) * 0.5f);
}
```

```
public float GetValue(long clockTime) {
    double gain; //pos = distance in samples from start of note (t1 - t0)
    //_pos2 = attack + hold;                    ADSR units are samples
    long pos = clockTime - initClock;
    if (_on) {
        //ATTACK
        if (pos < _attack) gain = (double)pos / _attack;

        //HOLD
        else if (pos < _pos2) gain = 1.0;

        //DECAY
        else if (pos < _pos2 + _decay)
            gain = (((_sustain - 1.0) / (_decay)) * (pos - _attack)) + 1.0;

        //SUSTAIN
        else gain = _sustain;
    }
    //RELEASE
    else if (pos < _release)
        gain = (((double)(0.0 - _init) / _release) * pos) + _init;
    else return 0;
    return (float)gain;
}
```

Figure 31: The generation of a sine wave written into the output buffer. This single note generates multiple sine waves, each known as a voice, with slight pitch and stereo field deviations to make the sound more complex. Each sine wave is multiplied by the envelope and panning values. The panning function evenly distributes each voice around the center with a given width. The GetValue function returns an interpolated gain value between each stage of the envelope, using a sample clock to keep track of envelope position.

In action, the synthesizer appears in the world as in Figure 32. Figure 32 shows the synthesizer not only connected to the output of the graph, with matching yellow orbs (like a traditional patch cable but yellow and wireless), but also shows the controllers of a player interacting with the synthesizer. The left handed controller is in the middle of turning the pitch knob on the synth, while the right handed controller is in the middle of playing a note. Figure 33 shows a closer look at how a controller turns a knob.

On the left hand, the Knob implementation of ParameterController is signaling the synthesizing AudioProcessor inside the ProcessorBundle to update the wave generating IAudioKernel to use a value of -2400 as its transposition.

Figure 32: A look into Fields while a user is actively playing the synthesizer. The left controller is transposing the synthesizer down two octaves by turning the 'Pitch' knob down to '-2400 cents'. The right controller is inserting its cursor into a note on the virtual keyboard, playing an F note on the synthesizer. Note that the synthesizer is playing sound since its output orb is connected to the input orb of the output, highlighted by a wireless yellow connection.

Figure 33: The left controller turns the 'Pitch' knob counterclockwise to '-2400 cents' with a rotation of the wrist while the cursor is semi or fully inserted into the knob.

When the pitch knob sends the value of -2400 into the synthesizer's IAudioKernel, the frequency of each note is calculated using the transposition value as shown by Figure 34. The frequency of each note is calculated from a base frequency value of 440 Hz, an A note, and a transposition value with some detune for added complexity. Note intervals are measured in 'cents' where 100 cents is one semitone (distance from A to A#). Since the relationship between frequencies is a doubling of frequency for every twelve notes, an octave, there is a logarithmic relationship of base 2 between the two scales. Since notes on a keyboard are measured in intervals of semitones, a conversion between semitones and frequency is vital to synthesize musically relevant frequencies. Starting from the base frequency of 440 Hz, the frequency of an octave higher would be 880 Hz, or double the frequency. This is also an interval of twelve semitones. To convert from the semitone distance, **ST**, to the scalar factor between the two frequencies, **FS**, we can take an exponential of two to the power of some semitone interval over twelve: **FS = 2^(ST/12)**.

Plugin in an octave up, twelve, gives two , and plugging in an octave down, negative twelve, gives one half.

```
var detune = context.Parameters.GetFloat(Parameters.detune, s);
for (int v = 0; v < voices; v++)
{
    _detune = Pan(v, 1, voices, detune) - 0.5f;
    frequency = parameters.GetFloat(Parameters.rootFrequency, s)
                * math.pow(
                    2,
                    (
                        parameters.GetFloat(Parameters.pitch, s)
                        + notes[n].GetNote() + _detune
                    ) / 12
                );
    delta = frequency / context.SampleRate;
    tmpPhase = phases[v];
    tmpPhase += delta;
    tmpPhase -= math.floor(tmpPhase);
    phases[v] = tmpPhase;
}
```

$$\frac{\text{oscillations}}{\text{seconds}} \div \frac{\text{samples}}{\text{seconds}} = \frac{\text{oscillations}}{\text{seconds}} \times \frac{\text{seconds}}{\text{samples}} = \frac{\text{oscillations}}{\text{samples}}$$

Figure 34: Each sample is calculated with a given phase, and each sample the phase is increased by a deltaphase, measured in oscillations per sample. This delta is calculated from converting the frequency, measured in oscillations per second, to oscillations per sample by dividing by the sample rate, measured in samples per second. This ensures frequency can change at the sample level. Note the detune is calculated using the same distribution function as the stereo deviations. As detune and width increase, the lower pitched notes end up on the left of the stereo field, and the higher pitched notes end up on the right.

Since the audio is multichannel, the phase increase by each sample calculation must be reset every channel but the last channel calculated. This ensures the phase evolves naturally through time, and doesn't snap to different values every block. Finally, after a buffer is filled with oscillations, the phases stay put for the next block to process, and the whole cycle repeats itself, generating waveforms on the fly with manipulatable characteristics, at a sample rate of 44100 samples per second.

Creative Environment

In the end, the synthesizer converts packages of note information received from an IAudioKernelUpdate into phase information for the wave generators to calculate amplitudes from. Then, to make the sound seem more interesting, multiple voices are stacked together with stereo and pitch deviations, and all fed through an envelope curve designed by the player. Notes are automatically managed through on and off trigger functions set up by the CollisionPads in the virtual keyboard, and then sent into the output of the synthesizer. The best part about all of this synthesis, is that it doesn't stop after leaving the synthesizer. The synthesizer is just one node in the graph, and only generates a seed waiting to be branched out and blossomed into different sonic configurations. After the waveforms leave the synthesizer, it is up to the user to use any of the default sound effects to make the sound of their dreams, or to even dive into the Fields Library to develop their own effects that bring more complexity to the entire sound environment.

The synthesizer is not the culmination of this project, and although it takes advantage of the entire system, the true culmination of this project has not been developed yet, because the system it will run on hasn’t been developed yet. This project is an exploration into sound editing in virtual reality, but is really the synthesis of an environment built for synthesizing synthesizers, that can be interacted with inside of a digitally synthesized environment. The synthesizer built in Fields represents the blank canvas that digital virtual environments represent, and the synthesizer is to its unlimited modules of effects as Fields is to its unlimited possibility of expanding the digital environments we are creative in. From

traditional digital audio workstations, to photo and video editing programs, the space in which creative expression is taking place is expanding into worlds of more capabilities and freedom of expression. Fields is a world to be expressive, as a developer, musician, or someone who wants to play around with a new toy.

Bibliography

aksyr, *Unity-DSP-Graph-Modular-Synth*, Github Repository, most recent access 5/2/2022,
<https://github.com/aksyr/Unity-DSP-Graph-Modular-Synth>

Brackeys, *Basics of Shader Graph*, Youtube, most recent access 5/2/2022,
<https://www.youtube.com/watch?v=Ar9eIn4z6XE&ab_channel=Brackeys>

DocFX, *DocFX Documentation Generator Tutorial*, most recent access 5/2/2022,
<https://dotnet.github.io/docfx/tutorial/docfx_getting_started.html>

Justin P Barnett, *How To MAKE A VR GAME: Beginner's guide to Virtual Reality & Unity XR Plugin*, Youtube, most recent access 5/2/2022,
<https://www.youtube.com/watch?v=1VC3ZOxn2Lo&ab_channel=JustinPBarnett>

Microsoft, *C# Documentation*, most recent access 5/2/2022,
<https://docs.microsoft.com/en-us/dotnet/csharp/>

Public Forum, *DOTS Audio Discussion*, Unity Forums, most recent access 5/2/2022,
<https://forum.unity.com/threads/dots-audio-discussion.651982/>

Steven W. Smith, Ph.D., *The Scientist and Engineer's Guide to Digital Signal Processing*, Online Textbook, most recent access 5/2/2022,
<https://www.dspguide.com/ch1.htm>

Unity Technologies, *ECS Track: Graph Driven Audio in an ECS World - Unite LA, Youtube*, most recent access 5/2/2022,
<https://www.youtube.com/watch?v=kDE-KxQBiYQ&t=440s&ab_channel=Unity>

Unity Technologies, *Unity Documentation*, most recent access 5/2/2022,
<https://docs.unity3d.com/Manual/index.html>

Valem, *How To Make a VR Game in 2021 - New Input System and OpenXR Support*, Youtube, most recent access 5/2/2022,
<https://www.youtube.com/watch?v=u6Rlr2021vw&ab_channel=Valem>