University of Massachusetts at Dartmouth

3DVerb

Three-dimensional Visualization of Algorithmic Reverb in Real Time

CIS 600 Master's Project.
Professor Jiawei Yuan, Advisor

Joseph McCann, MSCS candidate
8-15-2025

Abstract

Audio plugins for digital audio workstations (DAWs) traditionally prioritize functionality over aesthetics, and for practical purposes—CPU cycles are precious and multiple plugins running at once quickly saturate the available computing power of the average recording workstation PC. With increasing computing power available and a push towards performing all digital signal processing (DSP) on a general-purpose CPU, the ability to visualize audio effect parameter changes in real time becomes a possibility.

Traditionally, plugins with visualizers are typically simple spectrograms in 2D—they give you the technical information required to perform a function, e.g., an equalizer plugin that shows an accurate representation of human-hearable frequencies present in the signal. With the rise of the bedroom musician and performing music artists using real-time visual effects, there is a demand for more elaborate real-time visual effects.

3DVerb addresses this demand by proving conceptually that 3D audio visualization is straightforward with the relatively new JUCE WebView UI, which enables JavaScript events to communicate in real-time with a JUCE C++ backend. Because the UI is JavaScript and not the traditional C++ widget-based UI, this frees the developer to do with the backend data what they please. In this case, an existing WebGL library, ThreeJS, is utilized to respond to real-time data in a visually pleasing manner.

While some applications already exist for real-time 3D visualization of real-time audio data, such as T3X2R, these applications are not typically simple to install VST3 or AU plugins, but more complex devices—in the case of T3X2R, a Max for Live device only available to users of Ableton Live Suite, a popular but expensive DAW.

Using JUCE's reference reverb algorithm, 3DVerb demonstrates responsive parameter to visualization mapping that balances performance with perceptual fidelity. 3DVerb achieves real-time audio-visual synchronization through an event-driven, object-oriented architecture that prioritizes reliability, maintainability, and scalability while maintaining 165 fps on the development system (consistently maxing out the development machine monitor's 165 Hz refresh rate on a 4-year-old mid-range system with an AMD Ryzen 5 3600 CPU, 32GB RAM, and Radeon 6600XT GPU).

Table of Contents

Introduction

An audio plug-in is a computer software program that can process or synthesize a sound signal digitally. Sometimes these programs are offered as standalone applications, while more often they require a plug-in host, such as a digital audio workstation application (DAW), or as in the case of this project, the host included with the JUCE framework, *AudioPluginHost*. *AudioPluginHost* is a lightweight plug-in host, which allows for rapid iteration and debugging of the plug-in without requiring a resource intensive application like a DAW to be launched. Examples of DAWs include Pro Tools, Ableton Live, and Apple Logic Pro. This project utilizes the JUCE Framework in a Windows 11 environment with Visual Studio 2022 Community Edition as an IDE. The development machine has a 6-core AMD Ryzen 5 3600 CPU, 32GB RAM, and a Radeon 6600XT GPU.

The JUCE Framework, an open-source C++ codebase, is the “most widely used framework for audio application and plug-in development .” It includes a library of commonly used DSP “building blocks” so you can “quickly prototype and release native applications and plug-ins with a consistent user experience across all supported platforms [juce.com].”

The JUCE Framework simplifies deployment and compatibility with various plug-in APIs, which are responsible for handling communication between the host and the plug-in program. JUCE also provides everything necessary for a project to successfully be built in any popular IDE such as Visual Studio or Xcode.

For this project, a custom Visual Studio command launches the executable for *AudioPluginHost*. *AudioPluginHost* then receives audio data from an audio source of some kind, such as an *Audio Input* (microphone/audio-interface), a sound-synthesizing plug-in, or an audio file player. The plug-in host then feeds into 3DVerb which feeds to an *Audio Output* so that the processed audio may be heard.

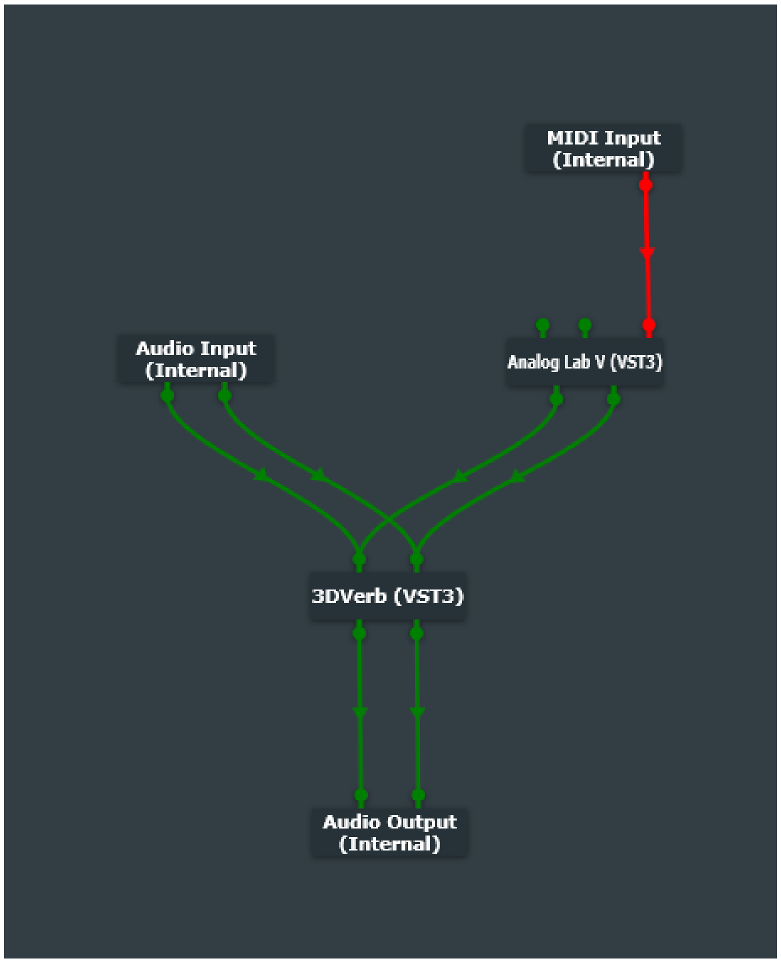


Figure 1: AudioPluginHost facilitates the creation of a filtergraph for 3DVerb. This allows for various audio sources to be tested with the plugin.

In Figure 1, the filter graph shows how the plug-in produces sound: The left and right channels of 3DVerb receive data from Audio Input and from another plugin, Arturia Analog Lab V, which synthesizes sounds via a MIDI controller (*MIDI Input* Above). These sounds are further processed in 3DVerb, where the frontend animation system is responsible for animating the live parameter data from the backend and extracting new features from that data.

3DVerb implements a novel approach to developing the visualization. With the relatively new JUCE WebView UI functionality, developers can build the project once and rapidly iterate on the front-end design without having to wait for a rebuild every time they want to see their UI design changes. Instead, they can simply refresh a web page. Since UI development is often the most time-consuming part of plug-in development, it makes sense to choose JUCE WebView UI for a plug-in that focuses on visualization. This decouples the DSP processing on the backend from the frontend. In Figure 2, a component diagram reflects how 3DVerb transfers and manages incoming audio data.

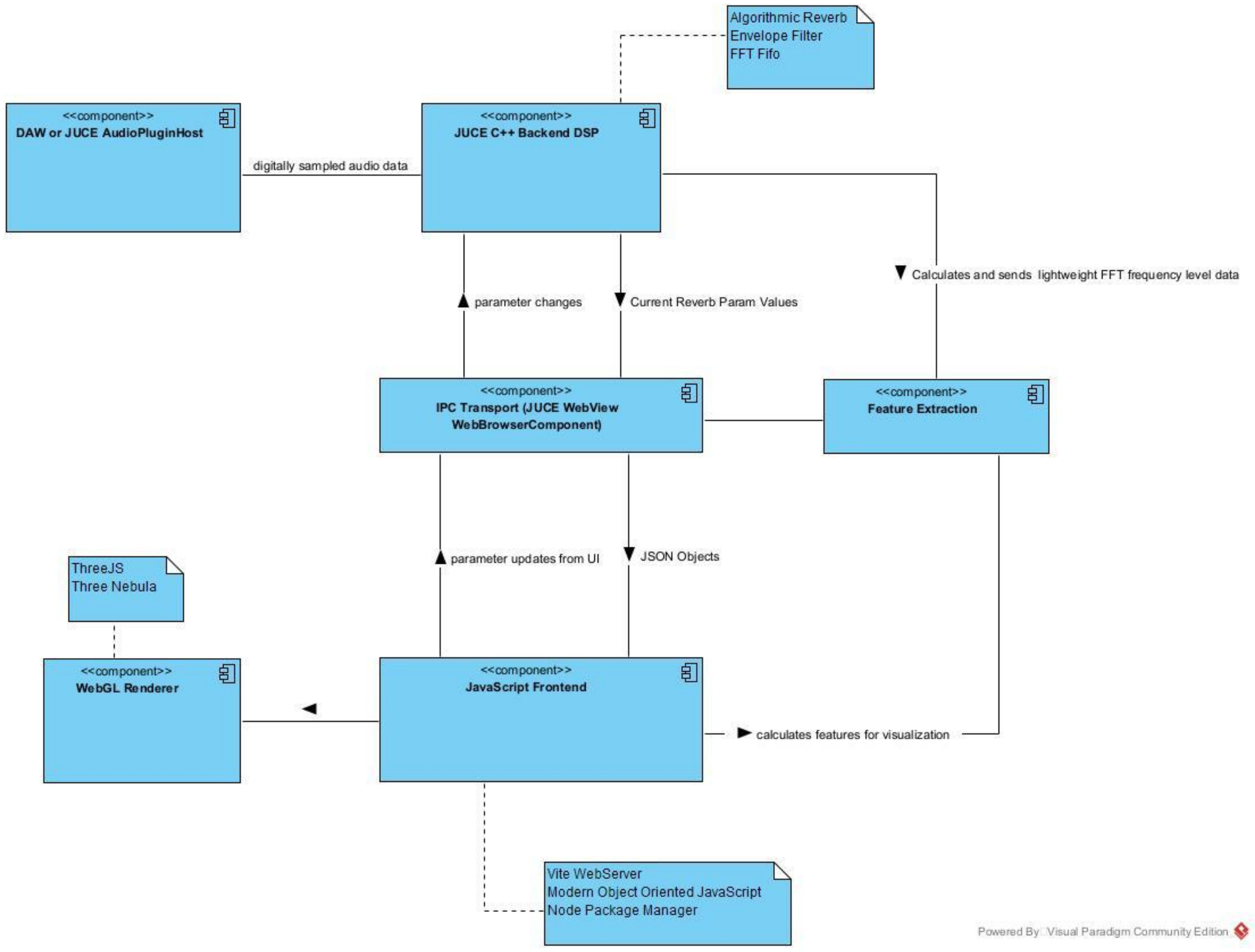

Algorithmic Reverb
Envelope Filter
FFT Fifo
<<component>>
DAW or JUCE AudioPluginHost
digitally sampled audio data
<<component>>
JUCE C++ Backend DSP
Calculates and sends lightweight FFT frequency level data
parameter changes
Current Reverb Param Values
<<component>>
IPC Transport (JUCE WebView WebBrowserComponent)
<<component>>
Feature Extraction
parameter updates from UI
JSON Objects
ThreeJS
Three Nebula
<<component>>
WebGL Renderer
<<component>>
JavaScript Frontend
calculates features for visualization
Vite WebServer
Modern Object Oriented JavaScript
Node Package Manager
Powered By Visual Paradigm Community Edition

Figure 2: Component Diagram of 3DVerb Architecture. The plug-in host continuously feeds audio data to the backend which interacts with the frontend via the IPC Transport bridge.

The audio is processed as usual in the backend, while parameter changes are propagated to the IPC transport via the JUCE WebView *WebBrowserComponent*, which relays parameter information to and from the backend. This frees the frontend to perform most of the feature extraction required for a coherent visualization, allowing the audio thread in the backend to remain performant.

While developing this plugin, I was constantly making parameter adjustments to the frontend visualization features, features derived from the reference reverb algorithm and real-time output level measurements. With some creative tweaks, features can display perceptually meaningful changes, for example, increasing room size results in the emitted sprites of Figure 3 to have a longer life. The WebView UI is refreshed like a typical webpage via a ctrl-shift-R command so that feature alterations are easily perceived without having to rebuild.

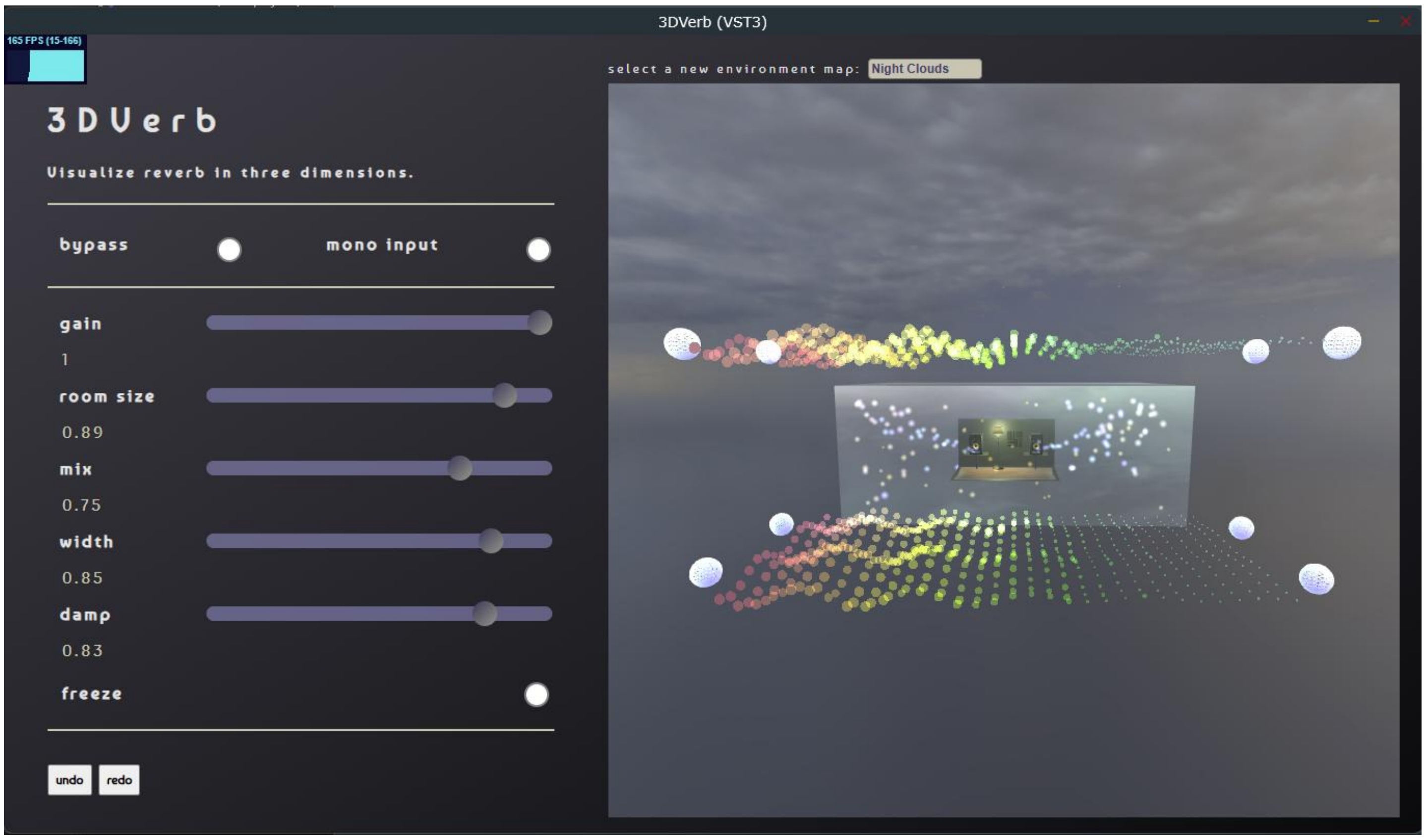


Figure 3: The 3DVerb UI with visualizer in-action. Parameter controls on the left are propagated to the backend via the IPC bridge, while current audio data and parameter values influence the behavior of the visualizer on the right.

This project helps demonstrate that three-dimensional visualization of processed audio is now feasible for audio plug-ins. We can not only visualize with somewhat utilitarian 2D spectrograms, but in three-dimensional spaces with more complex animations that interact with their environment. The primary challenge here is portraying the effect in a perceptually meaningful way while maintaining high performance.

With 3DVerb, perceptual accuracy is achieved through frequency mapping and various methods. Balancing perceptual accuracy with aesthetics continues to be a major challenge for visualizers. By leveraging the recent development of the JUCE WebView UI, 3DVerb interacts with the established JavaScript WebGL library, ThreeJS, to visualize digitally processed, real-time audio data in three dimensions, while maintaining high performance on a mid-range PC typical of most hobbyist recording artists or musicians.

This project should be of general interest to plug-in developers, audio-visual researchers, and real time system researchers. Potential applications for this plugin are for therapeutic applications in the music therapy field, acoustic education, and providing music performers with more visual tools.

In the following sections we will explore the digital signal processing (DSP) involved in algorithmic reverb, examine software system design and pertinent implementation details, evaluate plug-in performance and audio-visual analysis, and investigate the applications and extensions for future three-dimensional visualization plug-ins.

DSP Background and Algorithmic Reverb

To visualize a reverb effect, it is important to first define reverb and understand its attributes.

Defining Reverb

The reverb effect is one of the most well-known and utilized effects in acoustics and music production. A reverb effect simulates the natural acoustics phenomenon of reverberation, defined as “the persistence of sound after it is produced [Reverberation definition].” Reverberation occurs in physical spaces, both outdoors and indoors. Imagine how sound behaves at the beach or in a canyon or cavern. Now contrast that with how sound behaves in a living room compared to an empty hall or warehouse. Sound persists in unique ways in these spaces. The reverberation resulting from shouting “hello” into a canyon has distinctive characteristics than the reverberation of shouting “hello” in one’s living room or a concert hall. The sound will persist longer in the concert hall than in the living room and will have different echo characteristics in the canyon vs the cavern. This is due to the physics of sound.

Sound is reflected or absorbed by physical materials such as walls, sound-proofing material, carpets, couches, etc. These physical materials affect how reverb effects attempt to simulate this behavior of sound in many ways. A reverb effect can be achieved physically in several ways. These include using echo chambers, such as an empty bathroom with reflective tile materials; using spring reverb where a transducer and pickup capture the sound of a set of springs vibrating inside a box; or by using plate reverb which is made by capturing the vibrations made by a large plate of sheet metal [Reverb effect].

Digital Reverb

The reverberation effect can be simulated digitally. In modern sound design and music production, digital reverb effects are the most common. The technical term for these effects is *artificial reverberation*, initiated in the early 1960s by Manfred Schroeder.

Fundamental Building Blocks of Artificial Reverberation

Artificial reverberation is achieved through various means of digital signal processing (DSP). In the case of the *Schroeder Reverberator*, the following DSP techniques are deployed:

- Parallel bank of feedback comb filters
- Series connection of multiple all pass filters
- Mixing matrix [dsprelated.com]

A popular implementation of the *Schroeder Reverberator* is a public domain C++ program called *Freeverb*. JUCE provides a "simple stereo reverb based on the technique and tunings used in Freeverb [juce_audio_basics module]." *Freeverb* is often utilized because of its permissive license, ease of implementation, and low performance overhead. *Freeverb* and the JUCE implementation do not however use a mixing matrix, and instead use wet, dry, and width scaling after the two filter stages.

3DVerb uses this algorithm as a reference algorithm since the primary goal of this project is not to develop a new or improved reverb algorithm, but to explore how to visualize its parameters in three dimensions while following best practices for software development.

The Freeverb algorithm works by taking an input signal and delaying it through a set of parallel comb filters and then sequentially through a series of all pass filters.

Below we can see how the reference algorithm processes an input signal through 8 filtered-feedback-comb-filters, specifically a lowpass-feedback-comb-filter (LBCF). Each LBCF processes the same input signal and so it is said to process them in parallel.

In the primary function of the algorithm, *processStereo()*, shown below, *outL* and *outR* are summed in the loop with the same input. In the first iteration of the allPass loop, *outL* and *outR* are passed the values of *outL* and *outR* of the last comb filter output and sequentially fed that result *numAllPasses* times.

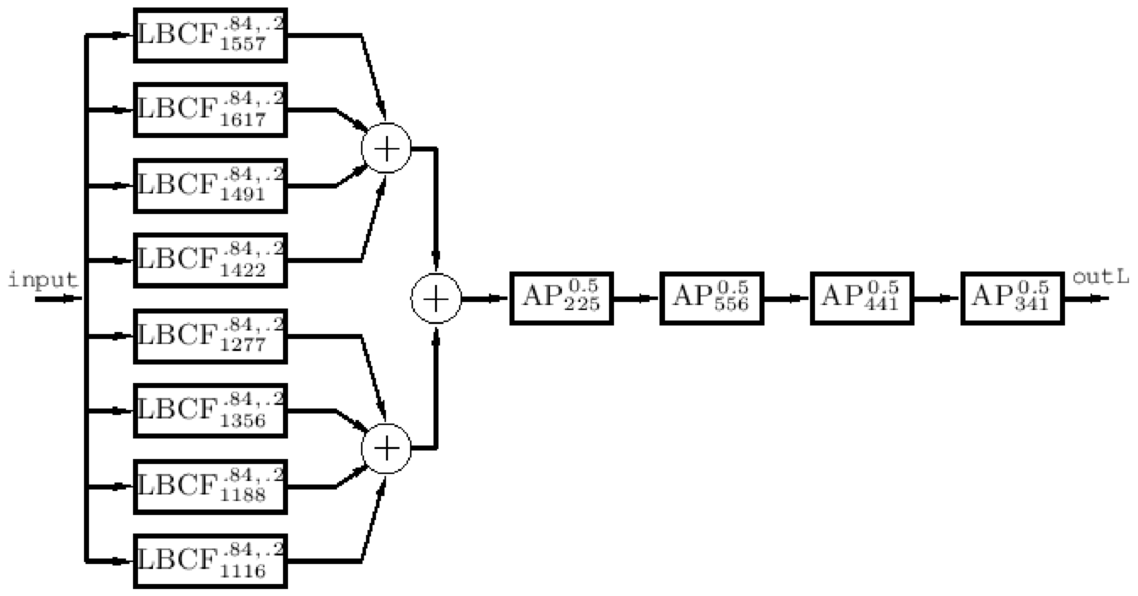
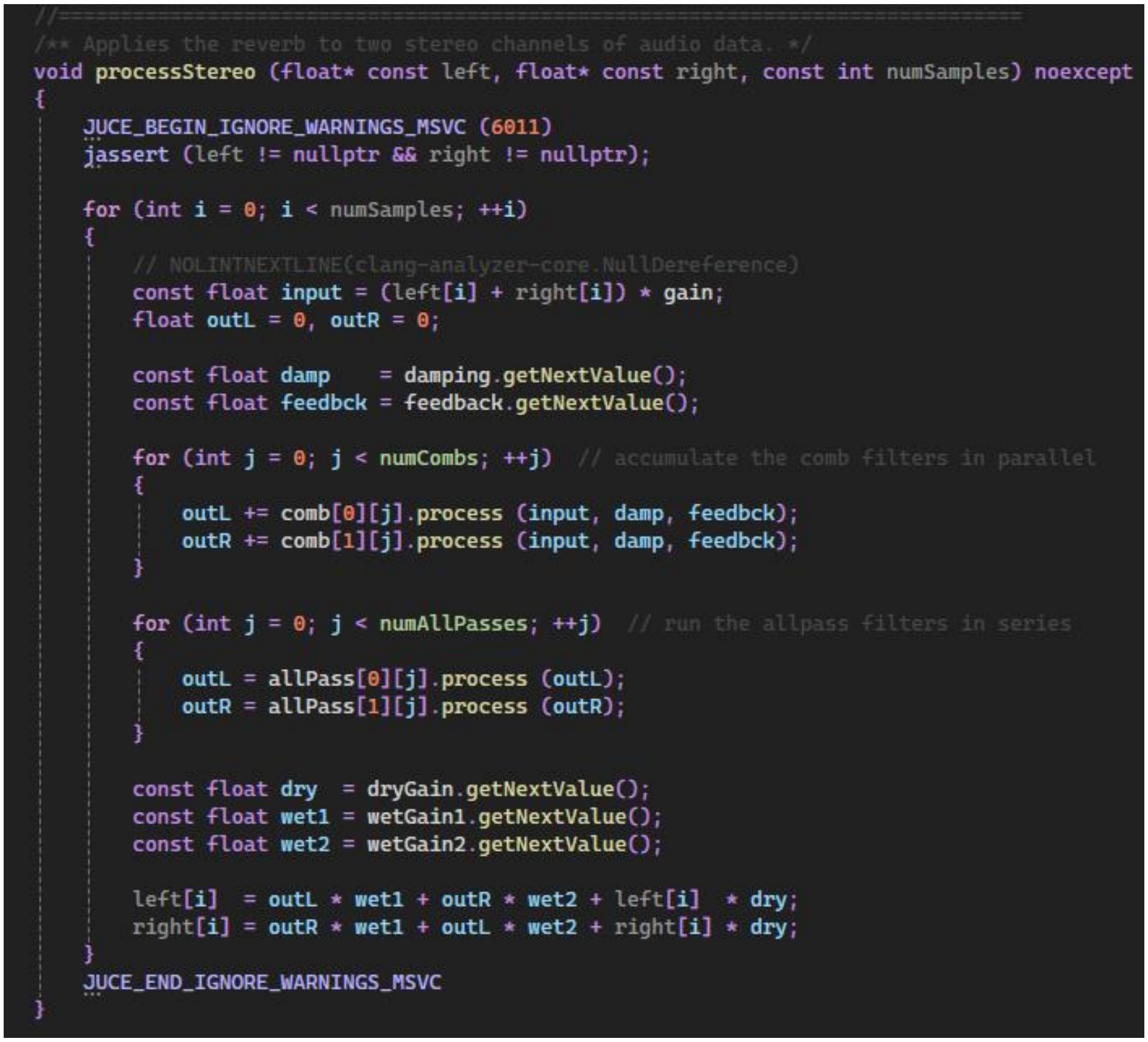


Figure 4: Audio input feeds into 8 parallel low-pass feedback comb filters, whose output is passed sequentially to 4 all-pass filters [DSPrelated.com]

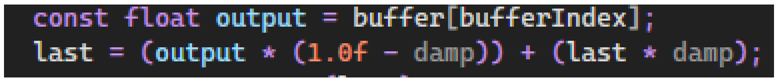
Primary Reverb Algorithm



```
//==============================================================================
/** Applies the reverb to two stereo channels of audio data. */
void processStereo (float* const left, float* const right, const int numSamples) noexcept
{
    JUCE_BEGIN_IGNORE_WARNINGS_MSVC (6011)
    jassert (left != nullptr && right != nullptr);

    for (int i = 0; i < numSamples; ++i)
    {
        // NOLINTNEXTLINE(clang-analyzer-core.NullDereference)
        const float input = (left[i] + right[i]) * gain;
        float outL = 0, outR = 0;

        const float damp    = damping.getNextValue();
        const float feedbck = feedback.getNextValue();

        for (int j = 0; j < numCombs; ++j)  // accumulate the comb filters in parallel
        {
            outL += comb[0][j].process (input, damp, feedbck);
            outR += comb[1][j].process (input, damp, feedbck);
        }

        for (int j = 0; j < numAllPasses; ++j)  // run the allpass filters in series
        {
            outL = allPass[0][j].process (outL);
            outR = allPass[1][j].process (outR);
        }

        const float dry  = dryGain.getNextValue();
        const float wet1 = wetGain1.getNextValue();
        const float wet2 = wetGain2.getNextValue();

        left[i]  = outL * wet1 + outR * wet2 + left[i]  * dry;
        right[i] = outR * wet1 + outL * wet2 + right[i] * dry;
    }
    JUCE_END_IGNORE_WARNINGS_MSVC
}
```

Figure 5: JUCE implementation of Freeverb, a type of Schroeder Reverberator. First the input is processed in parallel in the comb filter loop; then it is processed in a sequential series in the second nested loop.

The reverb algorithm [See *process()* method of Figure 7] works by using a circular buffer to delay the output samples, *output* in the code. The *last* variable is a private member of *CombFilter.* The "*last* line" below has the effect of gradually reducing the high frequencies, hence the low-pass behavior. The *last* variable now represents the filtered signal. It is scaled by *feedbackLevel* and then added back into the delayed buffer input [Figure 7].

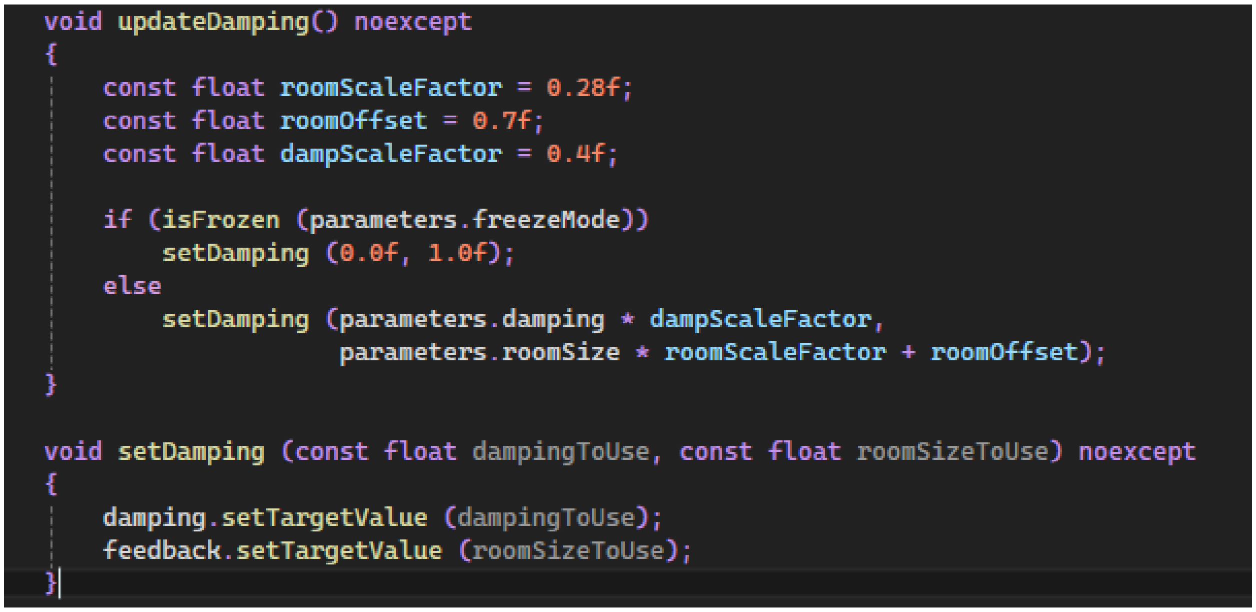


```
const float output = buffer[bufferIndex];
last = (output * (1.0f - damp)) + (last * damp);
```

damp is a value between 0 and 1 and controls how much of the old output stays and how much the high frequencies are reduced in the feedback signal. High *damp* means that more of the old output remains, resulting in more high frequency attenuation. Low *damp* results in less high frequency attenuation. When *damp* is low, this results in a more reflective sound, simulating a room without much sound absorbing material. When damp is high, high frequencies get reduced more quickly.

Later, we will explore the behavior of *ParticleWave* [see the color spectrum particles of Figure 3], which gathers frequency data from a frequency-only forward FFT. When damp is high, the particles representing high frequencies scale down more quickly, and when damp is low, they get reduced more quickly.

damping and roomSize have the biggest impact on the sound of this reverb algorithm. Damping directly interacts with the roomSize parameter:

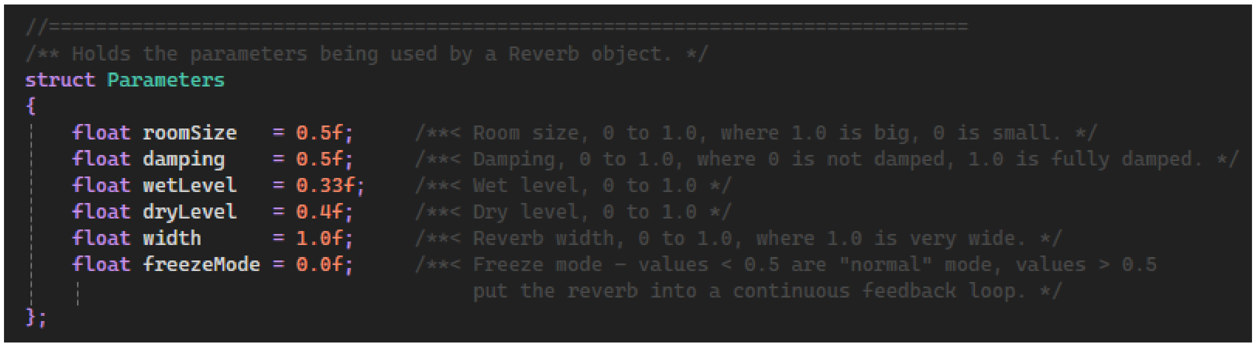


```
void updateDamping() noexcept
{
    const float roomScaleFactor = 0.28f;
    const float roomOffset = 0.7f;
    const float dampScaleFactor = 0.4f;

    if (isFrozen (parameters.freezeMode))
        setDamping (0.0f, 1.0f);
    else
        setDamping (parameters.damping * dampScaleFactor,
                    parameters.roomSize * roomScaleFactor + roomOffset);
}

void setDamping (const float dampingToUse, const float roomSizeToUse) noexcept
{
    damping.setTargetValue (dampingToUse);
    feedback.setTargetValue (roomSizeToUse);
}
```

feedbackLevel scales the filtered signal but also is directly controlled by the *roomSize* parameter which is set when the plug-in backend calls *setParameters()*. *setDamping()* calls *feedback.setTargetvalue(roomSizeToUse)*. Therefore, a higher *roomSize* value results in a longer decay time of the filtered signal.

The comb filters delay the signal in a buffer and gradually attenuate the high frequencies based on damping, while the *allpass* filters [Figure 8] modify their input and add it to the delayed signal multiplied by a constant factor of 0.5. This has the effect of increasing the number of echoes and diffusing the output of the comb filters. In the *processStereo()* method, their output is passed sequentially to the next allpass filter, increasing echo density and making the output of the comb filters sound more natural.

Reverb Parameters



```
//==============================================================================
/** Holds the parameters being used by a Reverb object. */
struct Parameters
{
    float roomSize   = 0.5f;     /**< Room size, 0 to 1.0, where 1.0 is big, 0 is small. */
    float damping    = 0.5f;     /**< Damping, 0 to 1.0, where 0 is not damped, 1.0 is fully damped. */
    float wetLevel   = 0.33f;    /**< Wet level, 0 to 1.0 */
    float dryLevel   = 0.4f;     /**< Dry level, 0 to 1.0 */
    float width      = 1.0f;     /**< Reverb width, 0 to 1.0, where 1.0 is very wide. */
    float freezeMode = 0.0f;     /**< Freeze mode - values < 0.5 are "normal" mode, values > 0.5
                                      put the reverb into a continuous feedback loop. */
};
```

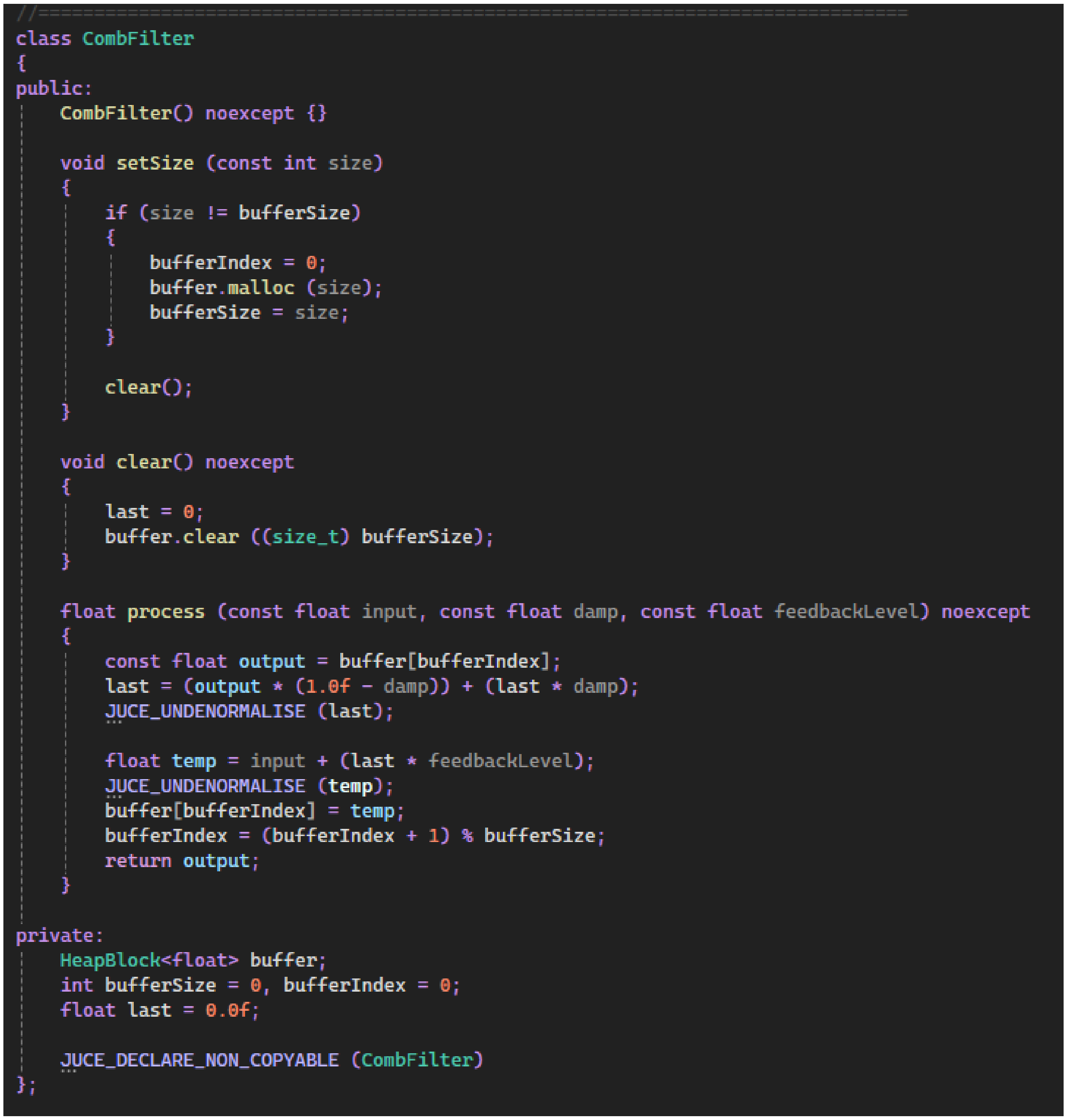
Figure 6: Embedded in the Reverb class is this Parameters struct. 3DVerb uses these same parameters, except wetLevel and dryLevel are combined into a single parameter called mix, where 0 is completely dry and 1 is completely wet, and the addition of a gain parameter.

Reverb Algorithm Parameter	**Parameter Effect on Sound**
Gain (3DVerb)	Controls input gain. Included for future expansion with effects that interact more with gain such as distortion.
roomSize	Small room = shorter decay of filtered signal Big room = longer decay of filtered signal
damping	Low damp = more echo/reflective sound; less attenuation of high frequencies High damp = less echo/reflective sound; more attenuation of high frequencies

mix (3DVerb)	mix = 1 == 100 % wet signal
	mix = 0 == 100% dry signal

width	High width = more stereo effect. Low width = less stereo effect
freeze	On = value > 0.5 == sends reverb into feedback loop where new incoming signal cannot be perceived if mix is 100%. Off = ‘normal mode.’

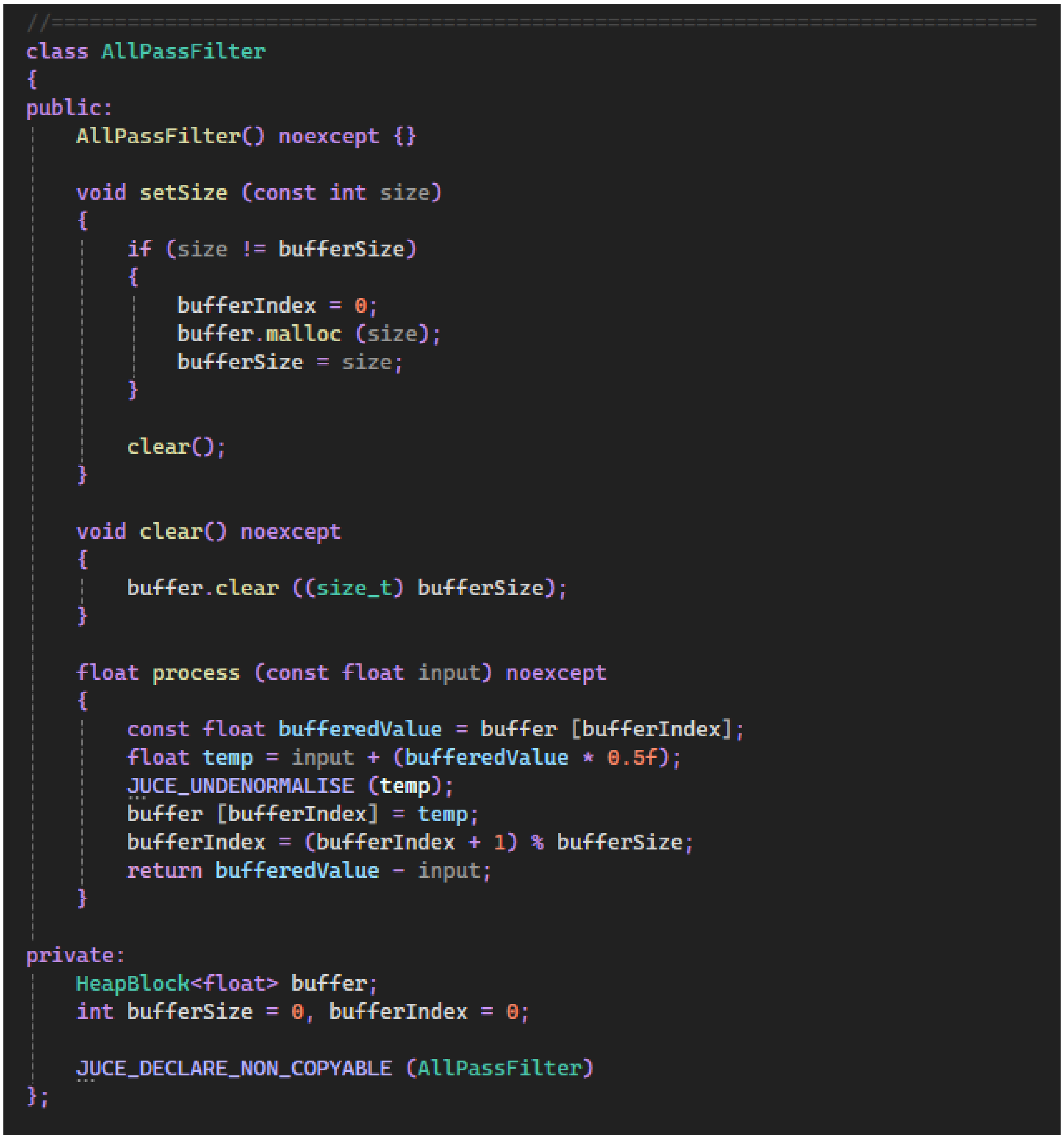
Comb Filter



```
//==============================================================================
class CombFilter
{
public:
    CombFilter() noexcept {}

    void setSize (const int size)
    {
        if (size != bufferSize)
        {
            bufferIndex = 0;
            buffer.malloc (size);
            bufferSize = size;
        }

        clear();
    }

    void clear() noexcept
    {
        last = 0;
        buffer.clear ((size_t) bufferSize);
    }

    float process (const float input, const float damp, const float feedbackLevel) noexcept
    {
        const float output = buffer[bufferIndex];
        last = (output * (1.0f - damp)) + (last * damp);
        JUCE_UNDENORMALISE (last);

        float temp = input + (last * feedbackLevel);
        JUCE_UNDENORMALISE (temp);
        buffer[bufferIndex] = temp;
        bufferIndex = (bufferIndex + 1) % bufferSize;
        return output;
    }

private:
    HeapBlock<float> buffer;
    int bufferSize = 0, bufferIndex = 0;
    float last = 0.0f;

    JUCE_DECLARE_NON_COPYABLE (CombFilter)
};
```

Figure 7: The CombFilter class is responsible for delaying the input signal and attenuating high frequencies modulated by the damp parameter.

All Pass Filter



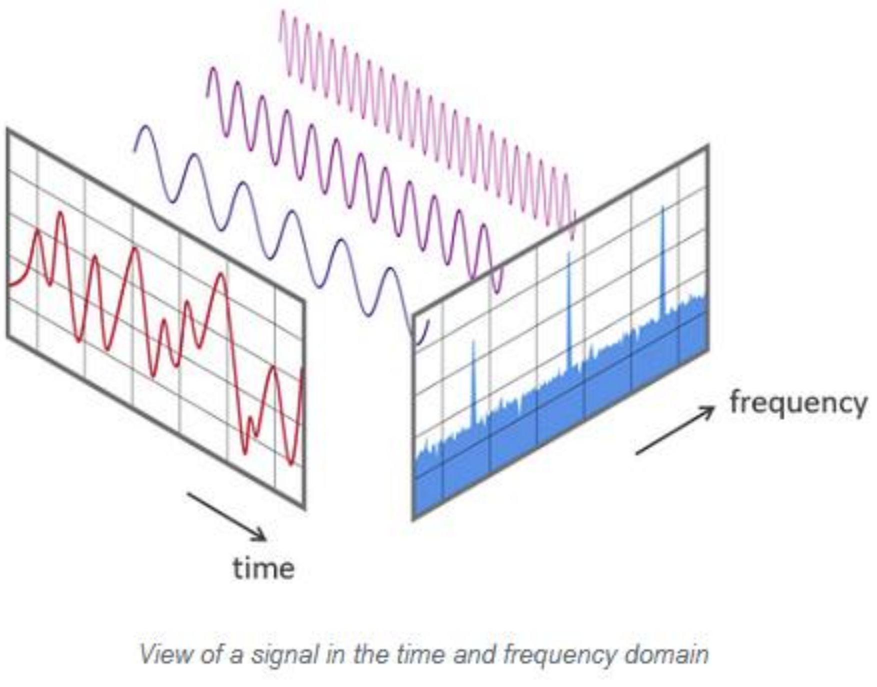
```cpp
//==============================================================================
class AllPassFilter
{
public:
    AllPassFilter() noexcept {}

    void setSize (const int size)
    {
        if (size != bufferSize)
        {
            bufferIndex = 0;
            buffer.malloc (size);
            bufferSize = size;
        }

        clear();
    }

    void clear() noexcept
    {
        buffer.clear ((size_t) bufferSize);
    }

    float process (const float input) noexcept
    {
        const float bufferedValue = buffer [bufferIndex];
        float temp = input + (bufferedValue * 0.5f);
        JUCE_UNDENORMALISE (temp);
        buffer [bufferIndex] = temp;
        bufferIndex = (bufferIndex + 1) % bufferSize;
        return bufferedValue - input;
    }

private:
    HeapBlock<float> buffer;
    int bufferSize = 0, bufferIndex = 0;

    JUCE_DECLARE_NON_COPYABLE (AllPassFilter)
};
```

Figure 8: The AllPassFilter is responsible for processing the combined output of the comb filters, increasing the number of echoes, and diffusing the sound, making individual echoes less noticeable.

Extracting Frequency Data using a Fast Fourier Transform (FFT)

Once the reverb algorithm has processed a block of audio in *processBlock()* [See Implementation Details, Backend], 3DVerb pushes the block's averaged left and right stereo samples into a custom FIFO queue where they are processed by the juce::dsp module's FFT algorithm. The goal is to then extract frequency information as a feature that can be consumed by the frontend visualization.

The Fast Fourier Transform is a commonly used mathematical technique used in DSP that "deconstructs a complex waveform into its component sine waves. [More about FFTs]." The FFT "converts a signal into [its] individual spectral components and thereby provides frequency information about the signal [Fast Fourier Transformation FFT – Basics]."



The FFT transforms time domain data into the frequency domain. [image taken from nti-audio.com]

The FFT transforms sound information from the time domain to the frequency domain. Frequency data are gathered in frequency bins that represent the magnitude levels of a given frequency over the duration of the block.

How the FFT Collects Frequency Data into Frequency Bins

A discrete section of the audio block is stored in a FIFO queue for processing once the queue is full. Below are the parameters relevant for FFT collection of frequency data.

Audio Hardware Sampling Rate (fs)

During development, this project uses a device with a sampling rate set to **fs** = **48 kHz**. This means that the device is digitally sampling 48000 samples of audio data every second.

FFT Size / Block Length (BL)

The block length is the selected number of samples and is always a positive base 2 integer called the FFT order. 3DVerb uses an FFT order of 11.

BL = 2^{11} = **2048** samples.

Nyquist frequency (fn)

The Nyquist frequency is the max frequency that the FFT can calculate due to its algorithm design. The Nyquist theorem posits that when sampling an analog signal, the sampling frequency must be at least double the highest frequency of the signal.

Nyquist frequency **fn** = fs / 2 = 48 kHz / 2 = **24 kHz**.

This means that with a sampling rate of 48 kHz, the FFT can give us frequency data from frequencies 0 Hz to 24 kHz. This is plenty as the best human ears cannot hear above 20 kHz and most musical sound data is in the mid range from 250 Hz to 4 kHz.

Frequency resolution (df)

The frequency resolution is the spacing from one frequency bin to the next. The smaller the frequency bin the more accurate the spectrogram. Increasing **BL** increases accuracy at the cost of increased computation time. 3DVerb uses a moderate **BL** of 2048 samples.

Frequency resolution **df** = fs / BL = 48000 / 2048 = **23.44 Hz**

3DVerb accumulates BL / 4 = 512 of these frequency bins of **df** width and maps them logarithmically (for human hearing perceptual accuracy) to a level between 0 and 1. These levels are then used by the frontend animation thread for visualization in *ParticleWave*.

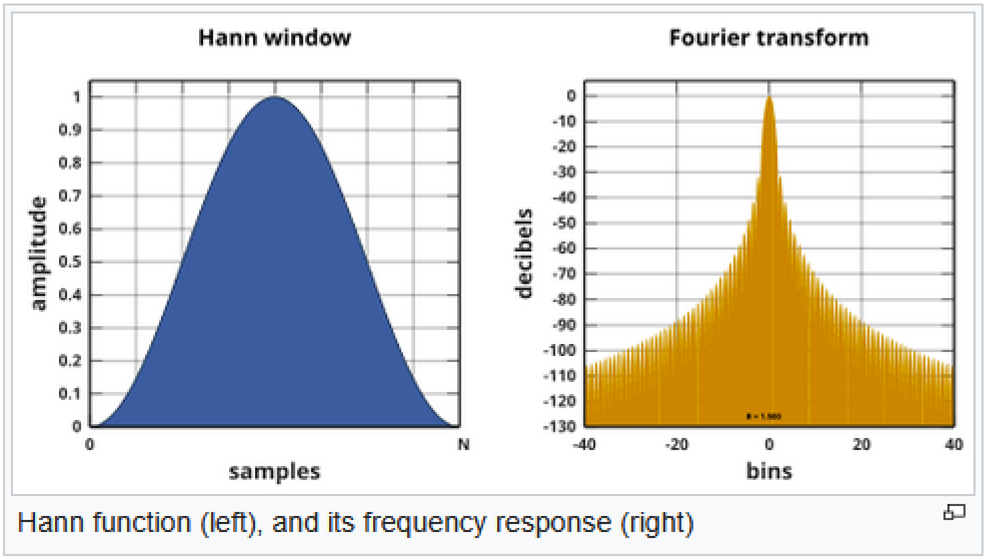
[Reference: Fast Fourier Transformation FFT – Basics]

Frequency Bin (Hz) df = 23.44	**Example Level (normalized)**
0 – 23.44	0.01
23.44 – 46.88	0.20
…	
234.4 – 257.84	0.80
257.84 – 281.28	0.70
…	
4688.0 – 4711.44	0.9
….	
7032.0 – 7055.44	0.60
7055.44 – 7078.88	0.62
…	
23976.56 – 24000 (**fn**)	0.04

Windowing

The stream of sampled audio data is passed to the audio buffer at a continuous rate, but the FFT algorithm expects a finite data sample of size **BL**. FFT assumes that a signal will be a sine wave with a nice round integer number of periods, but when those sine waves get partially cut off due to a discrete **BL**, a phenomenon called spectral leakage occurs when a jump in the time signal occurs. The result is that some energy “leaks” into the wrong frequency bins and can lead to inaccurate frequency representation.

A mathematical windowing function can be applied to the signal sample to prevent this leakage and resulting inaccuracy. A window function works by ensuring the signal begins and ends at zero amplitude. This prevents hard transitions from sampling continuous data at discrete intervals.



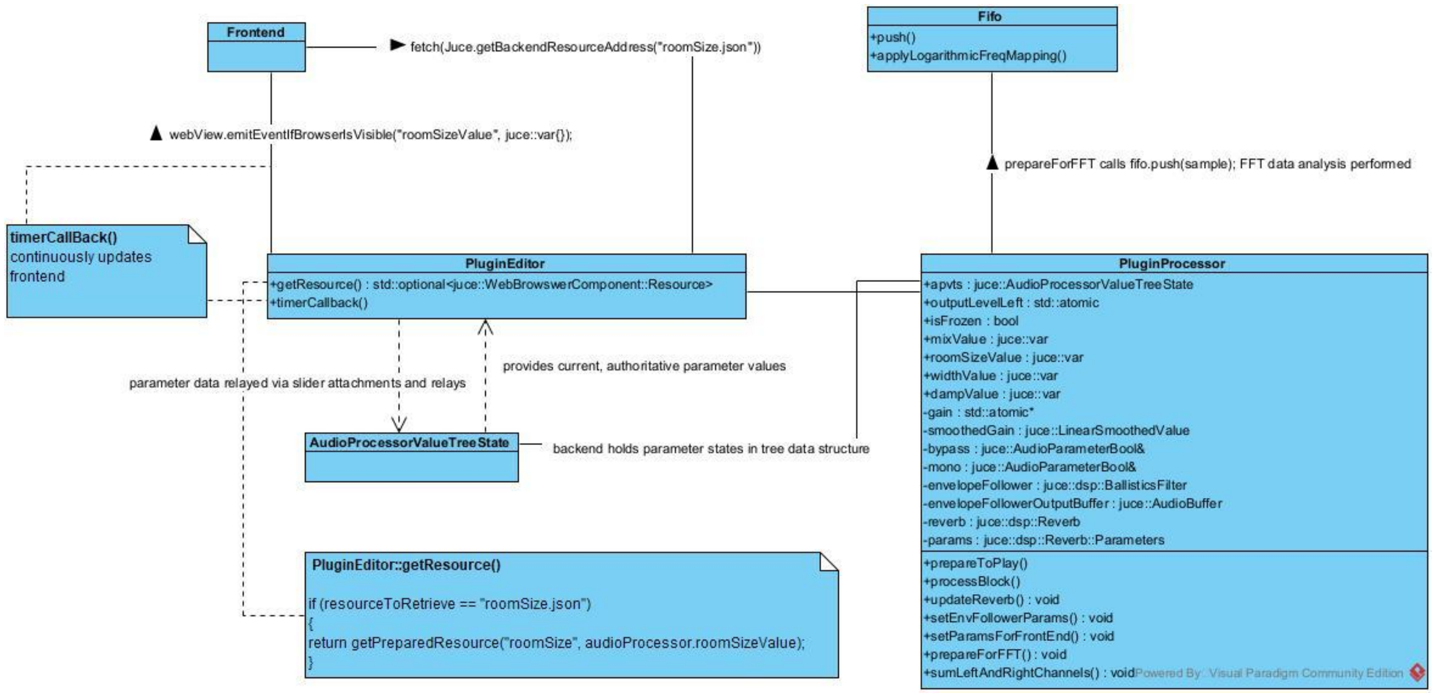
Pictured is the Hann Windowing function. In 3DVerb, only positive frequencies are collected in frequency bins. *https://en.wikipedia.org/wiki/Hann_function*

3DVerb utilizes the Hann window function, which is “extensively used in spectral analysis to improve the frequency domain representation of signals [Hanning window dsp].” The window function is applied before the FFT is performed to maximize accurate frequency representation.

The code for the FFT, windowing, and logarithmic frequency mapping will be discussed in [Implementation Details] below.

Software System Design

Backend architecture


Frontend
fetch(Juce.getBackendResourceAddress("roomSize.json"))
Fifo
+push()
+applyLogarithmicFreqMapping()
webView.emitEventIfBrowserIsVisible("roomSizeValue", juce::var{});
prepareForFFT calls fifo.push(sample); FFT data analysis performed
timerCallBack()
continuously updates
frontend
PluginEditor
+getResource() : std::optional<juce::WebBrowswerComponent::Resource>
+timerCallback()
PluginProcessor
+apvts : juce::AudioProcessorValueTreeState
+outputLevelLeft : std::atomic
+isFrozen : bool
+mixValue : juce::var
+roomSizeValue : juce::var
+widthValue : juce::var
+dampValue : juce::var
-gain : std::atomic*
-smoothedGain : juce::LinearSmoothedValue
-bypass : juce::AudioParameterBool&
-mono : juce::AudioParameterBool&
-envelopeFollower : juce::dsp::BallisticsFilter
-envelopeFollowerOutputBuffer : juce::AudioBuffer
-reverb : juce::dsp::Reverb
-params : juce::dsp::Reverb::Parameters
+prepareToPlay()
+processBlock()
+updateReverb() : void
+setEnvFollowerParams() : void
+setParamsForFrontEnd() : void
+prepareForFFT() : void
+sumLeftAndRightChannels() : void
provides current, authoritative parameter values
parameter data relayed via slider attachments and relays
AudioProcessorValueTreeState
backend holds parameter states in tree data structure
PluginEditor::getResource()
Powered By: Visual Paradigm Community Edition

```
if (resourceToRetrieve == "roomSize.json")
{
return getPreparedResource("roomSize", audioProcessor.roomSizeValue);
}
```

Figure 9: General System Architecture of a JUCE backend exchanging data with a WebView frontend

Most interaction on the backend occurs between two classes, (1) *ThreeDVerbAudioProcessorEditor* and (2) *ThreeDVerbAudioProcessor*. *PluginEditor* and *PluginProcessor* will be shorthand for (1) and (2), respectively. The convention in JUCE is to handle frontend code in *PluginEdito*r and backend code in *PluginProcessor*. With JUCE WebView, PluginEditor becomes more of an intermediary between frontend and backend, especially the *AudioProcessorValueTreeState*'s interaction with the WebBrowserComponent class, which allows parameter values to be exchanges between the frontend and backend.

These are the primary workhorse classes of any JUCE plug-in that processes audio. [Figure 9] shows the inner workings of how the backend communicates with the frontend and vice versa. The two most important methods of *PluginProcessor* are *prepareToPlay()* and *processBlock()*. *prepareToPlay()* is responsible for preparing the audio program for receiving audio data, while processBlock() processes the audio buffer.

PluginProcessor: Important Elements

prepareToPlay()

"An *AudioSource* has two states: 'prepared' and 'unprepared.' When a source needs to be played, it is first put into a 'prepared' state by a call to *prepareToPlay()* and then repeated calls will be made to *its getNextAudioBlock()* method to process the audio data [docs.juce.com]" See [Implementation Details, Backend]. This is also where DSP filters are initialized and prepared with things like the proper sample rate, block size, and the proper number of channels.

processBlock()

Once the audio block is obtained, it can be processed. This is the job of *processBlock()*. *processBlock()* receives an audio buffer and "renders the next block." This method is called by the audio thread, "so any kind of interaction with the UI is absolutely out of the question."

At first glance 3DVerb may appear to violate this statement. Inside *processBlock(),* accumulates frequency levels gathered from a Fast Fourier transform (FFT) inside *processBlock()* and then PluginEditor sends those levels as a JSON object to the frontend UI. However, in [Implementation Details, Backend], we will explore using a mutex lock to ensure thread safety.

The audio thread calls this function continuously if the *AudioSource,* in this case a *ReverbAudioSource,* is in a prepared state. This means that parameters are also continuously retrieved from the *AudioProcessorValueTreeState* (*apvts*) and updated here to ensure the perceived audio is using the correct audio.

Fifo

In my 3DVerb implementation, *processBlock()* calls a method called *prepareForFFT()* that pushes an averaged stereo sample into a Fifo queue implemented as a struct and initialized in stack memory. The *Fifo* struct is an important part of the backend design because it decouples *PluginProcessor* from the feature extraction of FFT data processing. The feature extraction of frequency level data is performed in the backend where there is a continuous and reliable access to raw sample data ensuring accurate data reaches the frontend.

AudioProcessorValueTreeState (apvts)

The *apvts* is responsible for an *AudioProcessor* object's state. It uses a *ValueTree*, a "powerful tree structure that can be used to hold free-form data, and which can handle its own undo and redo behaviour [docs.juce.com]." The *apvts* is initialized in *PluginProcessor* and *PluginEditor* holds a reference to *PluginProcessor* as *audioProcessor*. This design is sensible because the processor is the entity that alters the audio data based on the current state of the parameters, but the editor is the entity that alters the parameters.

In *PluginEditor*'s initialization list, Web Relay objects are initialized with IDs and Web Attachment objects are initialized with references to specific *apvts* parameters and the corresponding Web Relay object. The *PluginEditor WebBrowserComponent* is then initialized with options that contain data from each of the relays. Therefore, it is ultimately the Web Attachment objects that are responsible for updating the apvts [See Figure 10]. This guarantees that PluginEditor and PluginProcessor always access the most current apvts.

PluginEditor: Important Elements

AudioProcessorValueTreeState (apvts)

Since there is some crossover between PluginProcessor and PluginEditor regarding the *apvts*, I won't cover that again but will reiterate its importance to simplifying backend parameter state management.

PluginEditor::timerCallBack()

Event-Driven Architecture

The timer callback function enables the backend to continuously perform a task at a programmer-defined interval. The timer callback is initiated by calling `` `*startTimer(intervalTime)*` ``. For 3DVerb, with its WebView paradigm enabled architecture, this allows the *WebBrowserComponent* object, *webView*, to emit events every 60 milliseconds, like so:

- webView.emitEventIfBrowserIsVisible("outputLevel", juce::var{});

The frontend listens for these events and when they are received, requests the appropriate resource, e.g., `outputLevel.json` using the JUCE Frontend library code. The frontend is then free to use this data in its own thread. We will examine how this timer mechanism interacts with a try-lock mutex pattern in [Implementation Details], which prevents blocking between the audio and visual systems and guarantees consistent visual updates regardless of audio host timing.

PluginEditor::getResource()

The *getResource()* callback is registered when initializing the *webView* (*WebBrowserComponent*) options. The frontend retrieves the backend resource using the fetch() API and the JUCE frontend library code function `` `*getBackendResourceAddress()*` ``, which interacts with *WebBrowserComponent* to return the requested resource. Essentially, JUCE sends a specially formatted http request that JUCE intercepts before it gets sent over the network.

- Frontend event listener on window object listens for event emitted inside *timerCallback()*
- Fetch API calls *getBackenddResourceAddress(‘outputLevel.json’)*
- Inside *getBackendResourceAddress()*:

```
-      if (platform == "windows" || platform == "android")
            return "https://juce.backend/" + path;
```

- The *WebBrowserComponent* object *webView* intercepts the request, strips the string of everything except the path and passes the path to the registered *getResource()* callback, which then creates the JSON response. The HTTP response is then formatted and delivered by the *WebBrowserComponent webView* object and consumed in the frontend thread.

Frontend Architecture

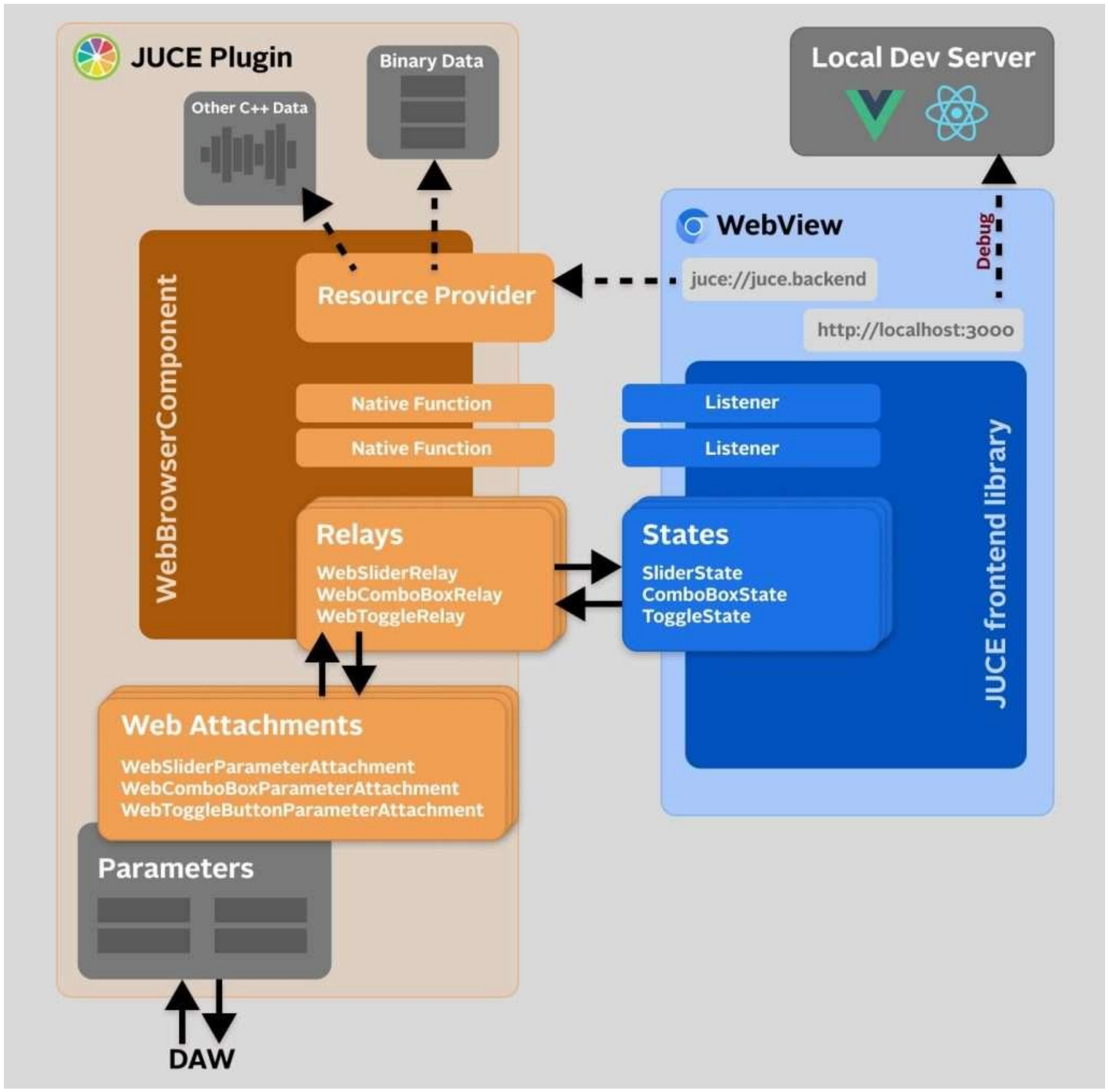


Figure 10: The JUCE WebView paradigm. The WebBrowserComponent is initialized in PluginEditor. It is a component that displays an embedded web browser. On windows, this is WebView2 and on Mac it is WebKit. The Web Attachments connect to the parameters of PluginProcessor, while the Relays relay data to and from the JUCE frontend library. [image from JUCE.com]

As discussed earlier, the frontend and backend are separated by the IPC transport JUCE *WebBrowserComponent*. This abstraction allows developers familiar with C++ to perform common C++ tasks without having to worry about learning how to use custom C++ UI widgets, resulting in faster UI iterative development.

The benefits of faster UI iterative development come at the cost of increased software complexity. However, since most development time is spent on UI development, and most developers are more familiar with HTML/CSS/JS than specialized C++ UI widgets, this tradeoff is justified. It’s one less thing for a frontend development team to learn, since according to the 2024 Stack Overflow Developer Survey, 64.6% of professional developers have extensive development experience in JavaScript [Stack Overflow Survey]. This has the added benefit of allowing the developer to run the JavaScript debugger on the local web server, which is helpful when managing the interactions of several features.

Additionally, this approach allows for more creative visual expression of audio data in real time, as evidenced by this project’s use of the ThreeJS library for creating a highly reactive WebGL animation. Figure 10 shows a more detailed look at the *WebView* paradigm provided by JUCE 8. It gives a clear visual of how the Web Attachment objects transfer a UI-updated parameter to the *apvts* in the backend.

It is also possible to have the C++ sliders and Web Sliders to co-exist, if practically necessary. This would allow the web slider’s state to mirror the C++ GUI slider’s state. This might be useful for some DAWs that might have compatibility issues with WebView parameter controls. The C++ GUI counterpart to a *WebSliderParameterAttachment* is simply a *SliderParameterAttachment*. See Figure 11 for a breakdown of the relationship between a web slider (HTML/JS), an *AudioParameterFloat*, and a JUCE GUI slider.

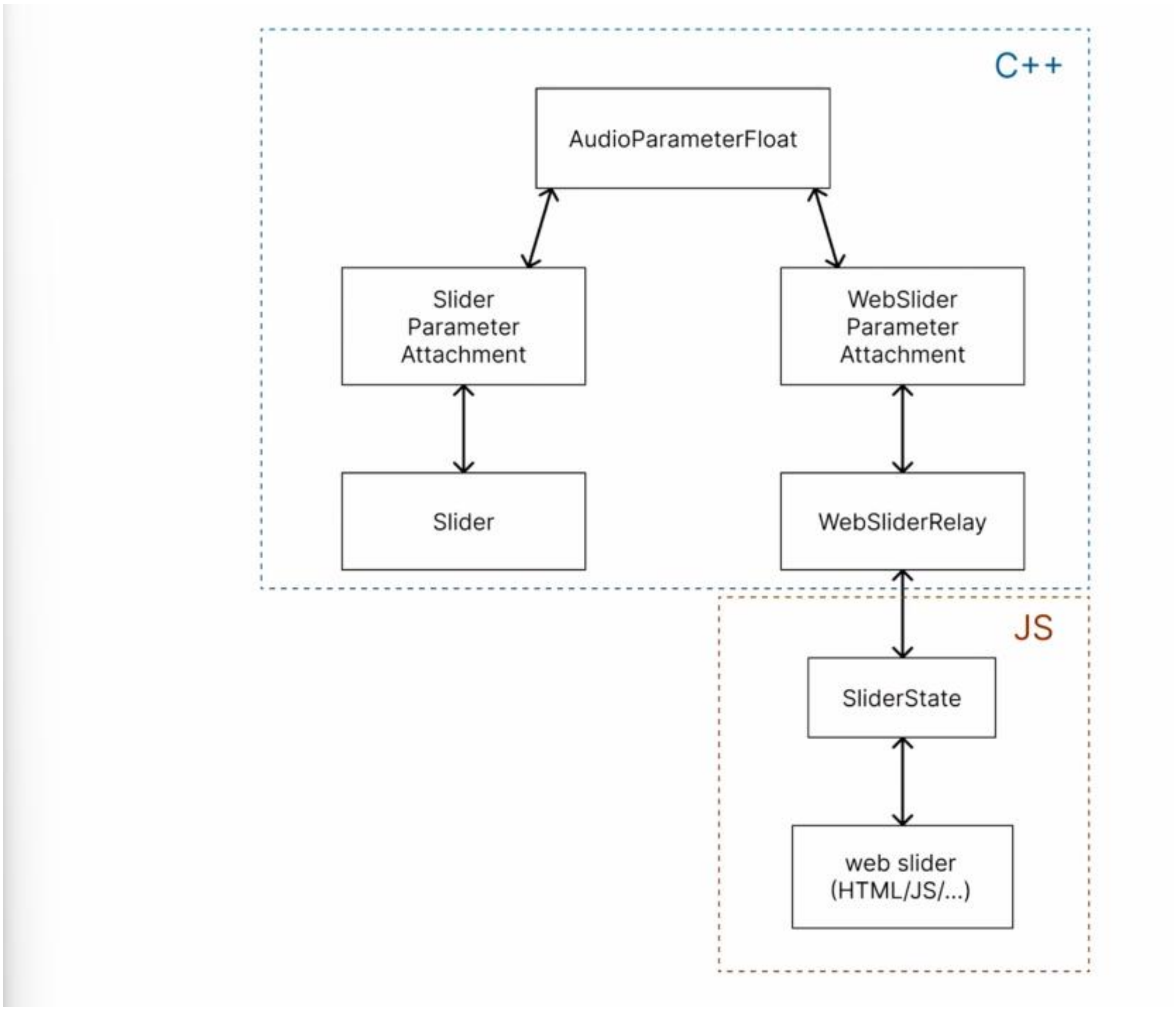


Figure 11: A JUCE GUI Slider is mirrored by a web slider. The utilization of a JUCE GUI (C++) slider is optional because the web slider combined with the slider relays of the WebBrowserComponent and the WebSliderParameterAttachment is enough to update the AudioProcessorValueTreeState of PluginProcessor [image from JUCE.com].

Design Patterns Employed

Once the backend code is complete and the plug-in is processing audio, organizing, and maintaining the state of the frontend's animation becomes the primary development challenge. I developed the following arrangement after needing to organize a somewhat monolithic code base that was loosely spread out across modules.

The problem was I was having to import and export too many variables, and the code was becoming difficult to extend and manage. By adopting a combination of the design patterns below, I was able to minimize importing and exporting by utilizing object-oriented JavaScript techniques, paired with well-established design patterns.

While the 3DVerb frontend does not purely adapt itself to a single design pattern, it utilizes elements of several and takes a hybrid approach appropriate for the real-time constraints of a complex audio-visualization system. I will argue why *AnimationController*, shown in Figure 13, is a candidate for each pattern below, and then show how it is a hybrid combination of these.

Mediator Pattern

This pattern "[defines] an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently [Gamma et al]." The Mediator pattern allows for the changing of a system's behavior without the need for defining many subclasses. Gamma et al describe the mediator object as one that "is responsible for controlling and coordinating the interactions of a group of objects [colleagues]. The mediator serves as an intermediary that keeps objects in the group from referring to each other explicitly [Gamma et al]."

3DVerb, as shown in Figure 13, utilizes a mediator-style object called *AnimationController*. *AnimationController* is responsible for initializing and coordinating animation events. The client, index.js, initializes the mediator, and now all animation events are mediated through *AnimationController*. Coupling between the animation classes is reduced through dependency injection and a single point of control is provided for initiating an animation behavior [Appendix, Custom JavaScript Animation Code: animation_controller.js].

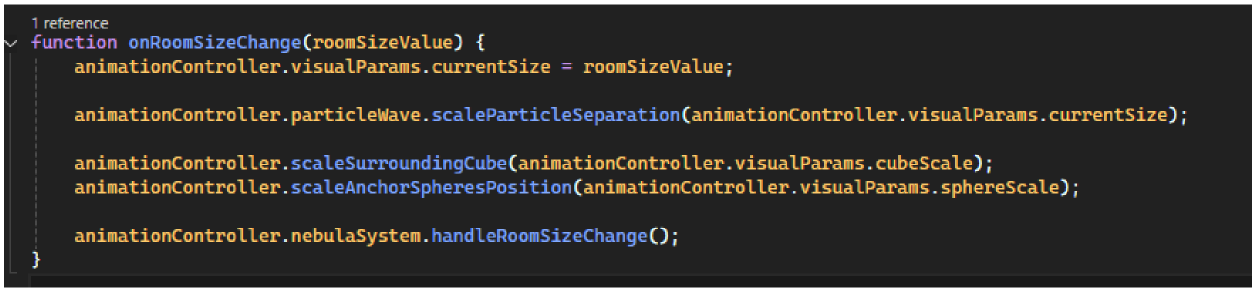
AnimationController, the mediator, coordinates the interactions between the colleagues *NebulaSystem*, *NebulaParams, ParticleWave*, *VisualParams*, and the 3D mesh objects. It is also responsible for synchronizing the state of the animation system for all these colleagues.

As an example, see Figure 12 for the client-side code in index.js, where all commands propagate through the *animationController* object [Appendix, Custom JavaScript Animation Code: index.js].
When room size changes:

1. The client only directly references *animationController*.
2. State changes are propagated through four distinct aspects of the animation system:
    a. *VisualParams*
    b. *ParticleWave*
    c. *3D Mesh Objects*

   d. *NebulaSystem*

3. All interaction happens within AnimationController, which holds an instance variable reference to each system.



```
1 reference
function onRoomSizeChange(roomSizeValue) {
    animationController.visualParams.currentSize = roomSizeValue;

    animationController.particleWave.scaleParticleSeparation(animationController.visualParams.currentSize);

    animationController.scaleSurroundingCube(animationController.visualParams.cubeScale);
    animationController.scaleAnchorSpheresPosition(animationController.visualParams.sphereScale);

    animationController.nebulaSystem.handleRoomSizeChange();
}
```

Figure 12: AnimationController behaving like a mediator with several different colleagues.

Façade Pattern

The Façade pattern "provides a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use." 3DVerb shares the motivation behind the Façade pattern, which is "to minimize the communication and dependencies between subsystems." The façade object "provides a single, simplified interface to the more general facilities of a subsystem."

In this case, 3DVerb client communicates with the animation subsystems by sending requests to the Façade, the *AnimationController*. Gamma et al describe the Façade pattern as one where clients that use the façade don't have access to its subsystem objects directly. This is true for 3DVerb, where the client only initializes an *AnimationController* [Appendix, Custom JavaScript Animation Code: index.js]. The client has no knowledge of ThreeJS, the *ParticleWave*, the *NebulaSystem*, or how *VisualParams* and *NebulaParams* interact. It only sends requests to the Façade, *Animation Controller*.

Controller style Mediator-Façade Hybrid

The *AnimationController* has two roles:

1. As a controller style mediator that coordinates complex parameter changes across various aspects of the visual system (*VisualParams, NebulaParams, NebulaSystem, ParticleWave*, 3D Mesh Objects).
2. As a Façade that provides the client with a unified interface that can handle the complexity of the animation.

Although it is not purely a Mediator, since *AnimationController* directly manages state instead of reacting to notifications sent by colleagues. This behavior makes it more of a controller than a pure mediator, which would depend, for example, on *VisualParams*, to send a notification to *AnimationController* that its params have changed. With the notification style approach of a pure Mediator design, excess latency might be introduced which would be a hindrance to a real-time audio-reactive visualization system. The controller approach allows for direct updates of features in real-time.

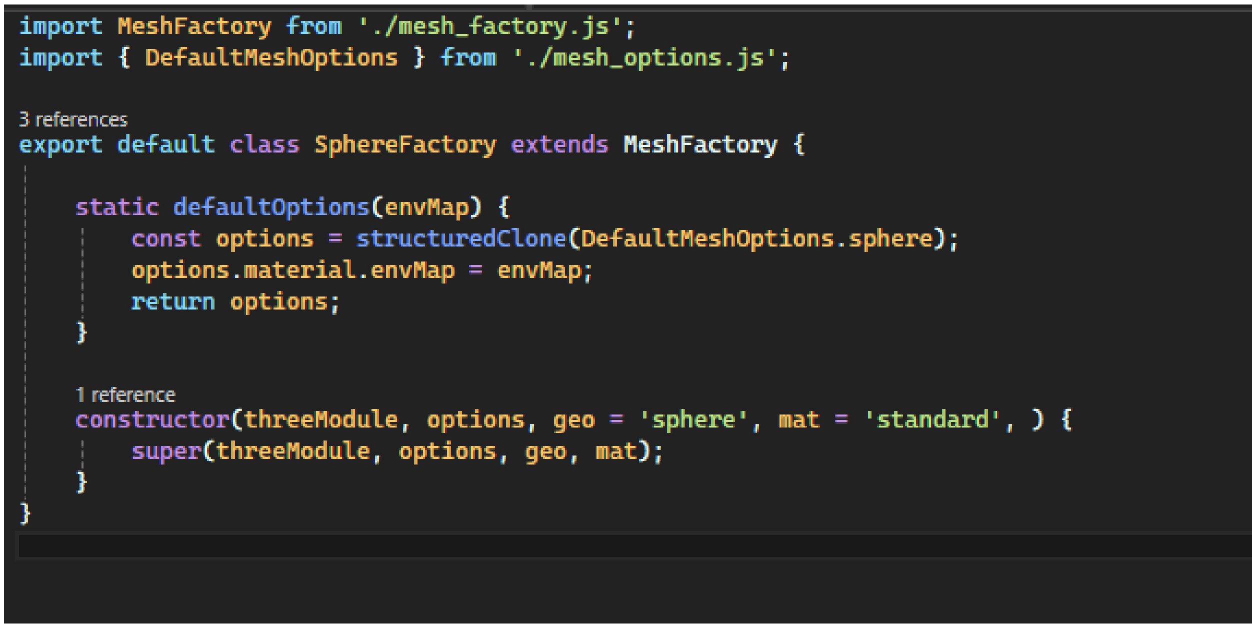
AnimationController is therefore a controller style mediator-façade hybrid. This hybrid design addresses concerns that neither pattern could solve on its own. There is loose coupling achieved through the Mediator approach and a simplified interface offered by the Façade approach, while the controller aspect ensures real-time data is synchronized across all the visualization systems.

Abstract Factory

MeshFactory below follows the Abstract Factory design pattern. When calling *sphereFactory.getMesh()* or *planeFactory.getMesh(), etc.,* an options object is passed and *getGeometry()* and *getMaterial()* are called internally in the *MeshFactory* constructor and automatically provide the correct geometry or material for the given concrete factory.

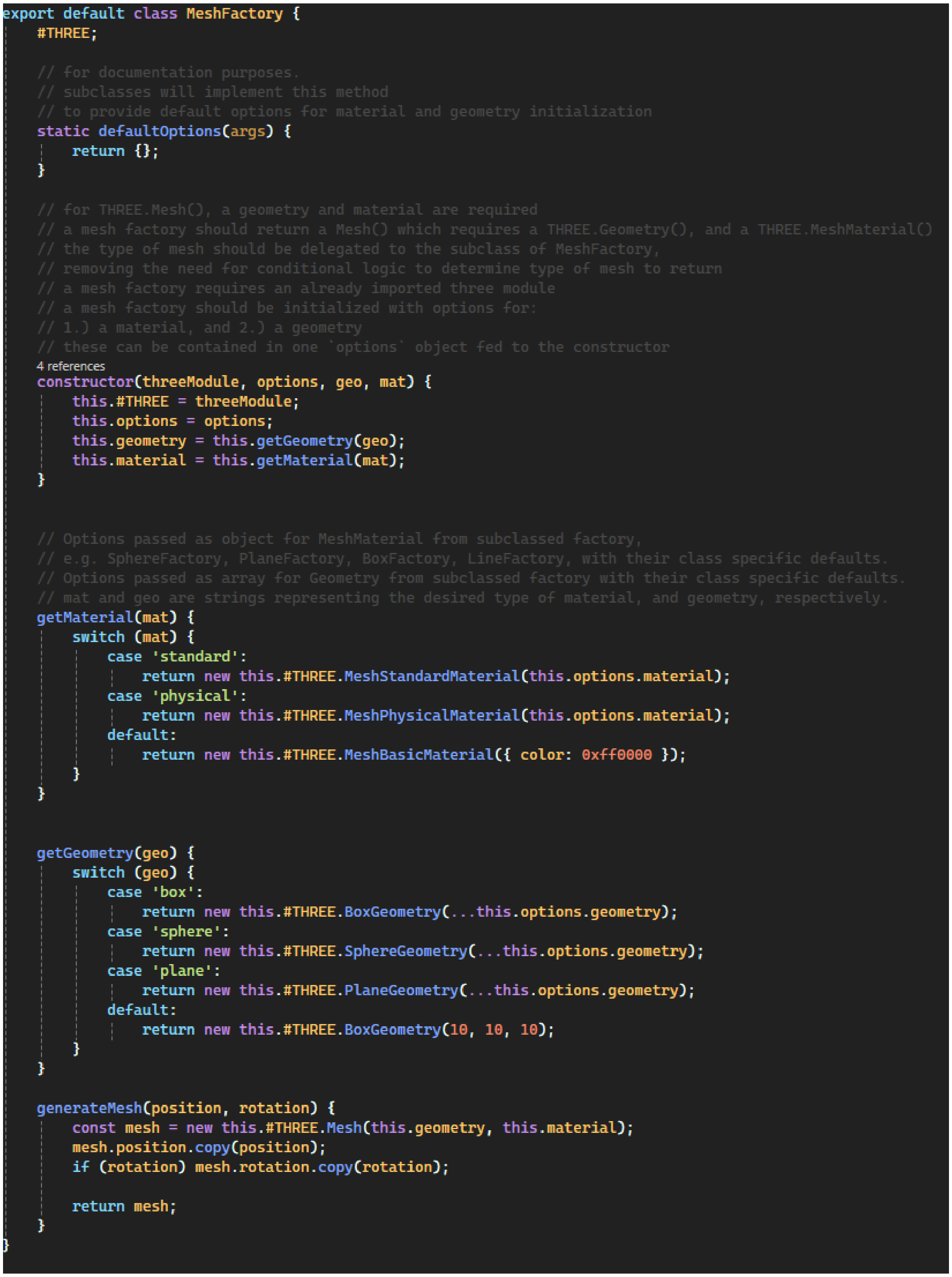
In the subclass constructors, *super() is called in their respective constructor* with the passed options from the caller. This pattern allows for easily adding more ThreeJS mesh objects to the scene later without the need for adding custom functions.

First a subclass:



```
import MeshFactory from './mesh_factory.js';
import { DefaultMeshOptions } from './mesh_options.js';

3 references
export default class SphereFactory extends MeshFactory {

    static defaultOptions(envMap) {
        const options = structuredClone(DefaultMeshOptions.sphere);
        options.material.envMap = envMap;
        return options;
    }

    1 reference
    constructor(threeModule, options, geo = 'sphere', mat = 'standard', ) {
        super(threeModule, options, geo, mat);
    }
}
```

Next, the base class:



```
export default class MeshFactory {
    #THREE;

    // for documentation purposes.
    // subclasses will implement this method
    // to provide default options for material and geometry initialization
    static defaultOptions(args) {
        return {};
    }

    // for THREE.Mesh(), a geometry and material are required
    // a mesh factory should return a Mesh() which requires a THREE.Geometry(), and a THREE.MeshMaterial()
    // the type of mesh should be delegated to the subclass of MeshFactory,
    // removing the need for conditional logic to determine type of mesh to return
    // a mesh factory requires an already imported three module
    // a mesh factory should be initialized with options for:
    // 1.) a material, and 2.) a geometry
    // these can be contained in one `options` object fed to the constructor
    4 references
    constructor(threeModule, options, geo, mat) {
        this.#THREE = threeModule;
        this.options = options;
        this.geometry = this.getGeometry(geo);
        this.material = this.getMaterial(mat);
    }


    // Options passed as object for MeshMaterial from subclassed factory,
    // e.g. SphereFactory, PlaneFactory, BoxFactory, LineFactory, with their class specific defaults.
    // Options passed as array for Geometry from subclassed factory with their class specific defaults.
    // mat and geo are strings representing the desired type of material, and geometry, respectively.
    getMaterial(mat) {
        switch (mat) {
            case 'standard':
                return new this.#THREE.MeshStandardMaterial(this.options.material);
            case 'physical':
                return new this.#THREE.MeshPhysicalMaterial(this.options.material);
            default:
                return new this.#THREE.MeshBasicMaterial({ color: 0xff0000 });
        }
    }


    getGeometry(geo) {
        switch (geo) {
            case 'box':
                return new this.#THREE.BoxGeometry(...this.options.geometry);
            case 'sphere':
                return new this.#THREE.SphereGeometry(...this.options.geometry);
            case 'plane':
                return new this.#THREE.PlaneGeometry(...this.options.geometry);
            default:
                return new this.#THREE.BoxGeometry(10, 10, 10);
        }
    }

    generateMesh(position, rotation) {
        const mesh = new this.#THREE.Mesh(this.geometry, this.material);
        mesh.position.copy(position);
        if (rotation) mesh.rotation.copy(rotation);

        return mesh;
    }
}
```

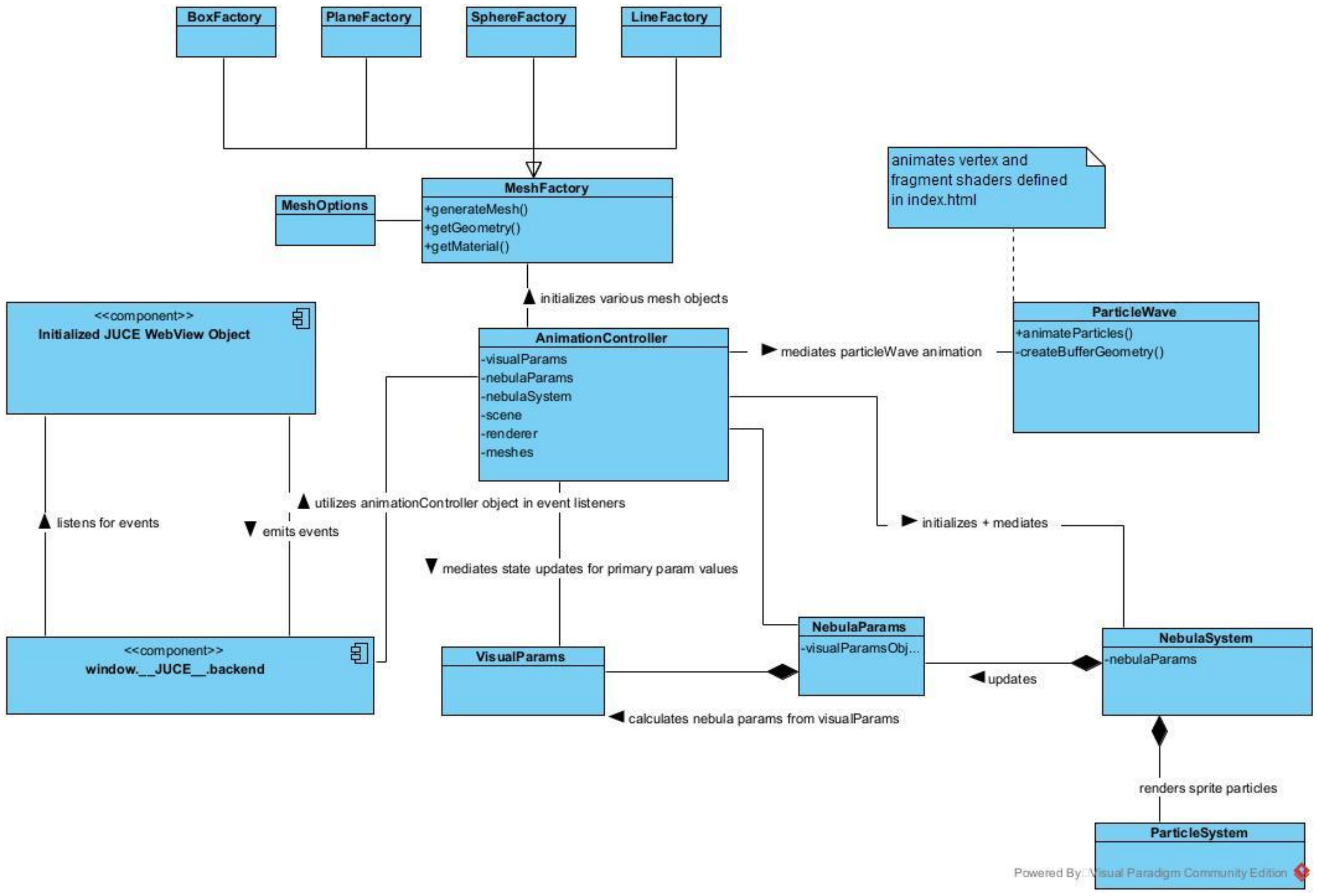


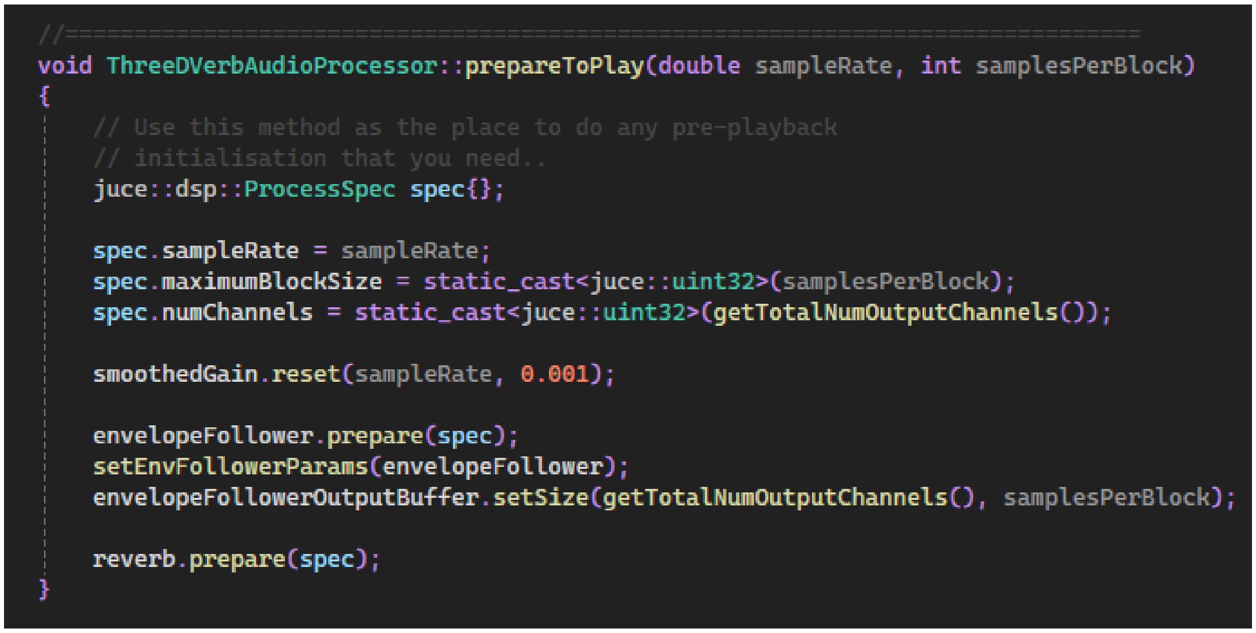
Figure 13: A high-level view of 3DVerb's frontend architecture. Note AnimationController's role as a mediator.

Implementation Details

Backend

PluginProcessor::prepareToPlay()

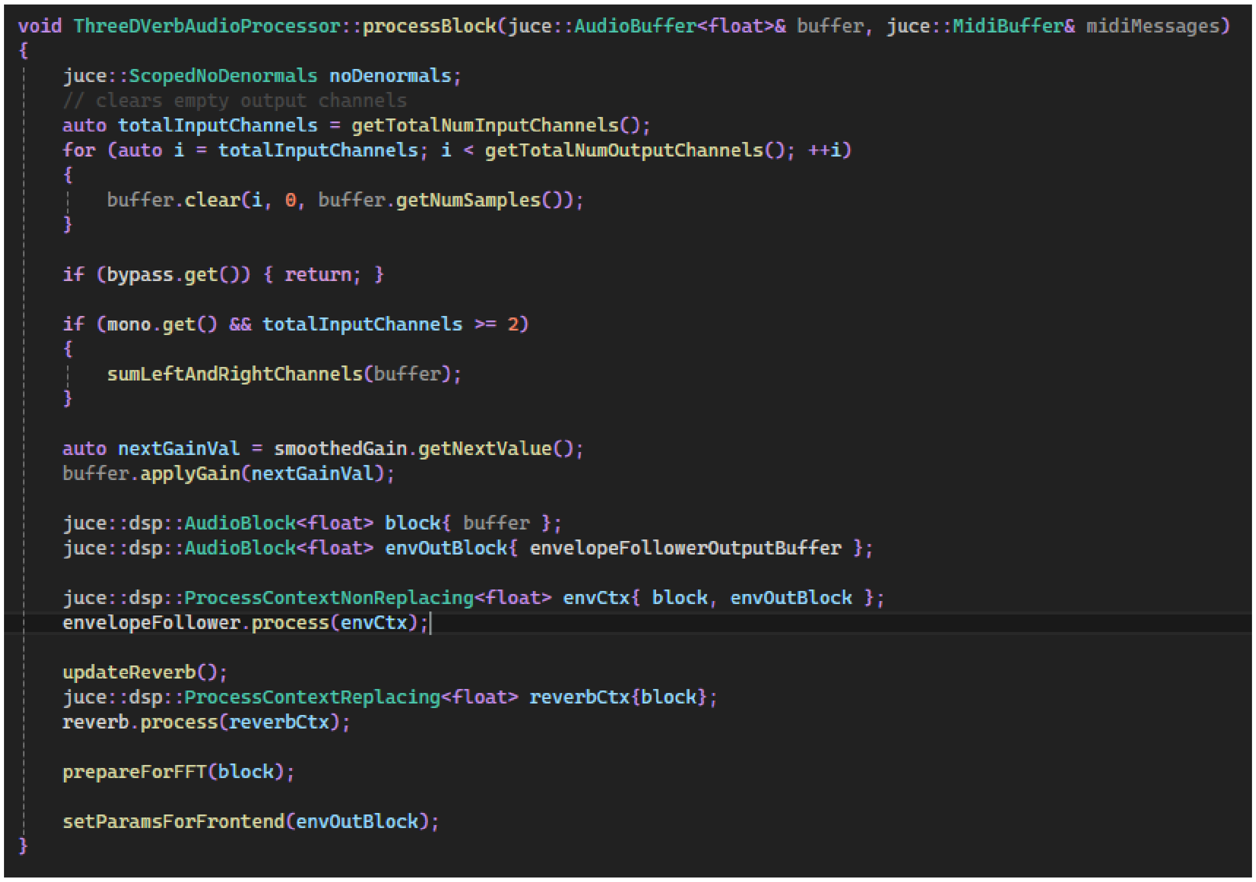
Figure 14 shows 3DVerb's customized *prepareToPlay()* method, although the general steps are similar across plugins. The *sampleRate* is set by the plug-in host and *samplesPerBlock* is dependent on audio hardware callbacks. Audio is continuously buffered through the computer's audio hardware and must be prepared to be played back digitally. In JUCE, an *AudioSource* is the "base class for objects that can produce a continuous stream of audio [juce.com]." Since it's assumed the audio will be processed in some way, these processing filters must be prepared somehow. In 3DVerb, *prepareToPlay()* calls *prepare()* on two filters, an envelope follower, for measuring output level, and the JUCE reference reverb algorithm provided by the *juce::dsp::Reverb* wrapper.



```
//==============================================================================
void ThreeDVerbAudioProcessor::prepareToPlay(double sampleRate, int samplesPerBlock)
{
    // Use this method as the place to do any pre-playback
    // initialisation that you need..
    juce::dsp::ProcessSpec spec{};

    spec.sampleRate = sampleRate;
    spec.maximumBlockSize = static_cast<juce::uint32>(samplesPerBlock);
    spec.numChannels = static_cast<juce::uint32>(getTotalNumOutputChannels());

    smoothedGain.reset(sampleRate, 0.001);

    envelopeFollower.prepare(spec);
    setEnvFollowerParams(envelopeFollower);
    envelopeFollowerOutputBuffer.setSize(getTotalNumOutputChannels(), samplesPerBlock);

    reverb.prepare(spec);
}
```

Figure 14: prepareToPlay() is called by JUCE internally to prepare an AudioSource for playback. An AudioSource must be in a 'prepared' state before any calls are made to getNextAudioBlock().

PluginProcessor::processBlock()

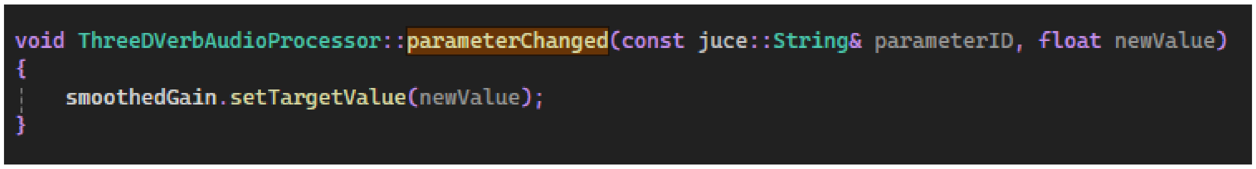


```
void ThreeDVerbAudioProcessor::processBlock(juce::AudioBuffer<float>& buffer, juce::MidiBuffer& midiMessages)
{
    juce::ScopedNoDenormals noDenormals;
    // clears empty output channels
    auto totalInputChannels = getTotalNumInputChannels();
    for (auto i = totalInputChannels; i < getTotalNumOutputChannels(); ++i)
    {
        buffer.clear(i, 0, buffer.getNumSamples());
    }

    if (bypass.get()) { return; }

    if (mono.get() && totalInputChannels >= 2)
    {
        sumLeftAndRightChannels(buffer);
    }

    auto nextGainVal = smoothedGain.getNextValue();
    buffer.applyGain(nextGainVal);

    juce::dsp::AudioBlock<float> block{ buffer };
    juce::dsp::AudioBlock<float> envOutBlock{ envelopeFollowerOutputBuffer };

    juce::dsp::ProcessContextNonReplacing<float> envCtx{ block, envOutBlock };
    envelopeFollower.process(envCtx);

    updateReverb();
    juce::dsp::ProcessContextReplacing<float> reverbCtx{block};
    reverb.process(reverbCtx);

    prepareForFFT(block);

    setParamsForFrontend(envOutBlock);
}
```

Figure 15: The function definition for processBlock(). This function is critical to the functionality of any juce plugin that applies any sort of DSP filter. In this case, a Ballistics Filter is utilized for envelopeFollower and the juce::dsp::Reverb wrapper around the reference algorithm is used for reverb.

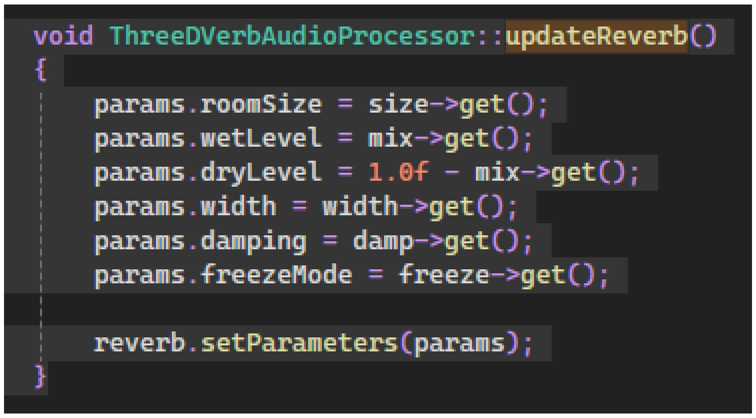
Once an *AudioSource* is prepared, it can be processed by *processBlock()*. Function definition is shown in [Figure 15].

- empty output channels are cleared to prevent old data from accidentally being processed.
- no processing occurs if the Boolean parameter *bypass* is true.
- The left and right channels are summed in a helper function if the Boolean parameter *mono is checked.* Normally a DAW handles this functionality but without customizing the *juce_audio_devices* module, the easiest way to support mono devices like an electric guitar is by including this simple helper function.
- To prevent sudden spikes in gain, gain level is smoothed in a separate callback function:

- 

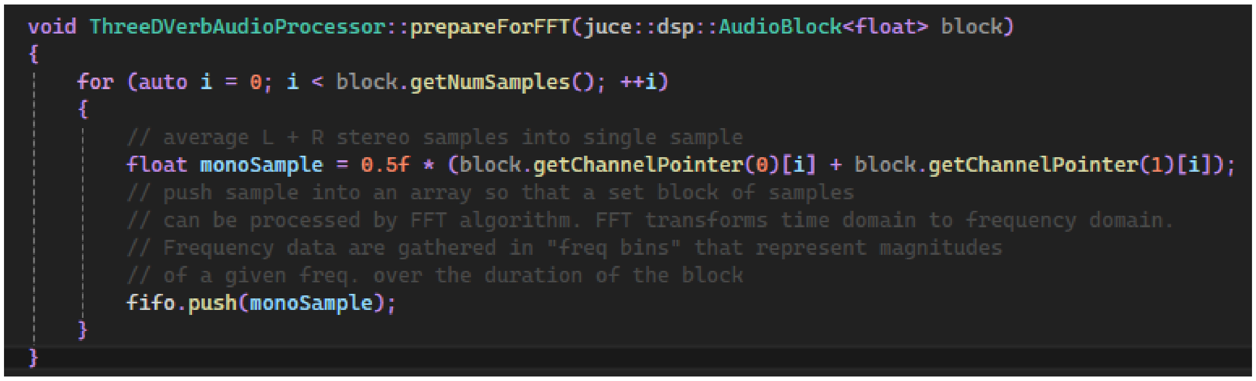
```cpp
void ThreeDVerbAudioProcessor::parameterChanged(const juce::String& parameterID, float newValue)
{
    smoothedGain.setTargetValue(newValue);
}
```

- Two blocks are initialized: One for *reverb* data, one for *envelopeFollower* data.
- The envelopeFollower needs data unaltered by the Reverb algorithm and so processes its context first. It is initialized with block as its input and envOutBlock at its output.
- Reverb params are updated with authoritative values from *AudioProcessorValueTreeState* (*apvts)* and then the reference algorithm uses those parameters:



```cpp
void ThreeDVerbAudioProcessor::updateReverb()
{
    params.roomSize = size->get();
    params.wetLevel = mix->get();
    params.dryLevel = 1.0f - mix->get();
    params.width = width->get();
    params.damping = damp->get();
    params.freezeMode = freeze->get();

    reverb.setParameters(params);
}
```

- reverb object processes its context (context information is passed to the reference algorithm's process method).
- In a helper function that receives the block pertaining to the reverb context, *prepareForFFT() is called:*



```cpp
void ThreeDVerbAudioProcessor::prepareForFFT(juce::dsp::AudioBlock<float> block)
{
    for (auto i = 0; i < block.getNumSamples(); ++i)
    {
        // average L + R stereo samples into single sample
        float monoSample = 0.5f * (block.getChannelPointer(0)[i] + block.getChannelPointer(1)[i]);
        // push sample into an array so that a set block of samples
        // can be processed by FFT algorithm. FFT transforms time domain to frequency domain.
        // Frequency data are gathered in "freq bins" that represent magnitudes
        // of a given freq. over the duration of the block
        fifo.push(monoSample);
    }
}
```

- Finally, params are prepared for consumption by the frontend in a format it expects.

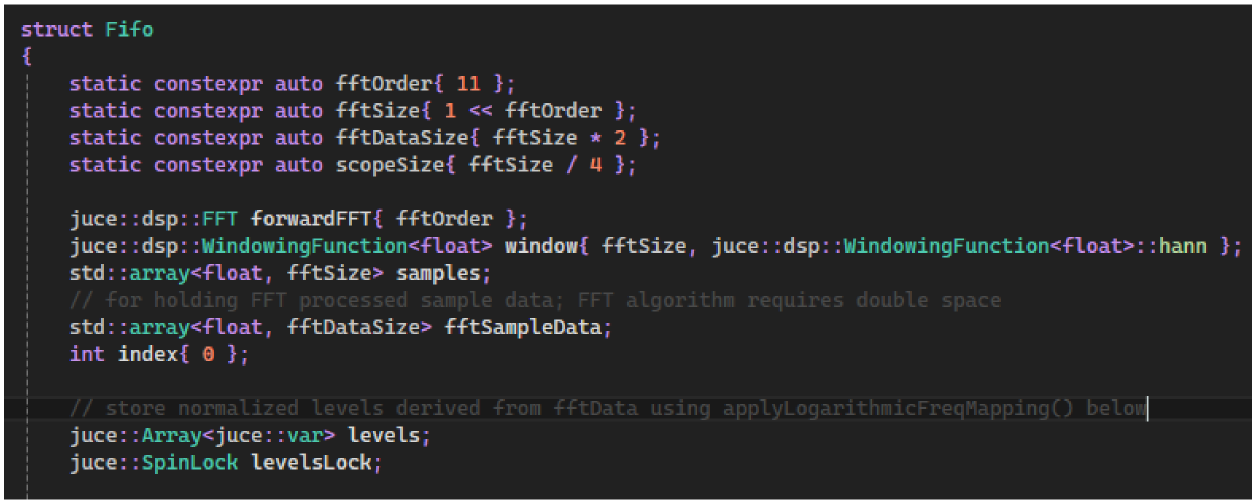
FFT Frequency Data Feature Extraction

After the reverb object has processed its context, its frequency data can be analyzed. This is interesting because different parameter settings will result in different frequencies being boosted or lowered. The "damp" parameter has the biggest influence on high frequencies, with low damp resulting in less attenuation of high frequencies and vice versa.

FIFO Queue

A FIFO Queue is the standard approach to handling FFT data.

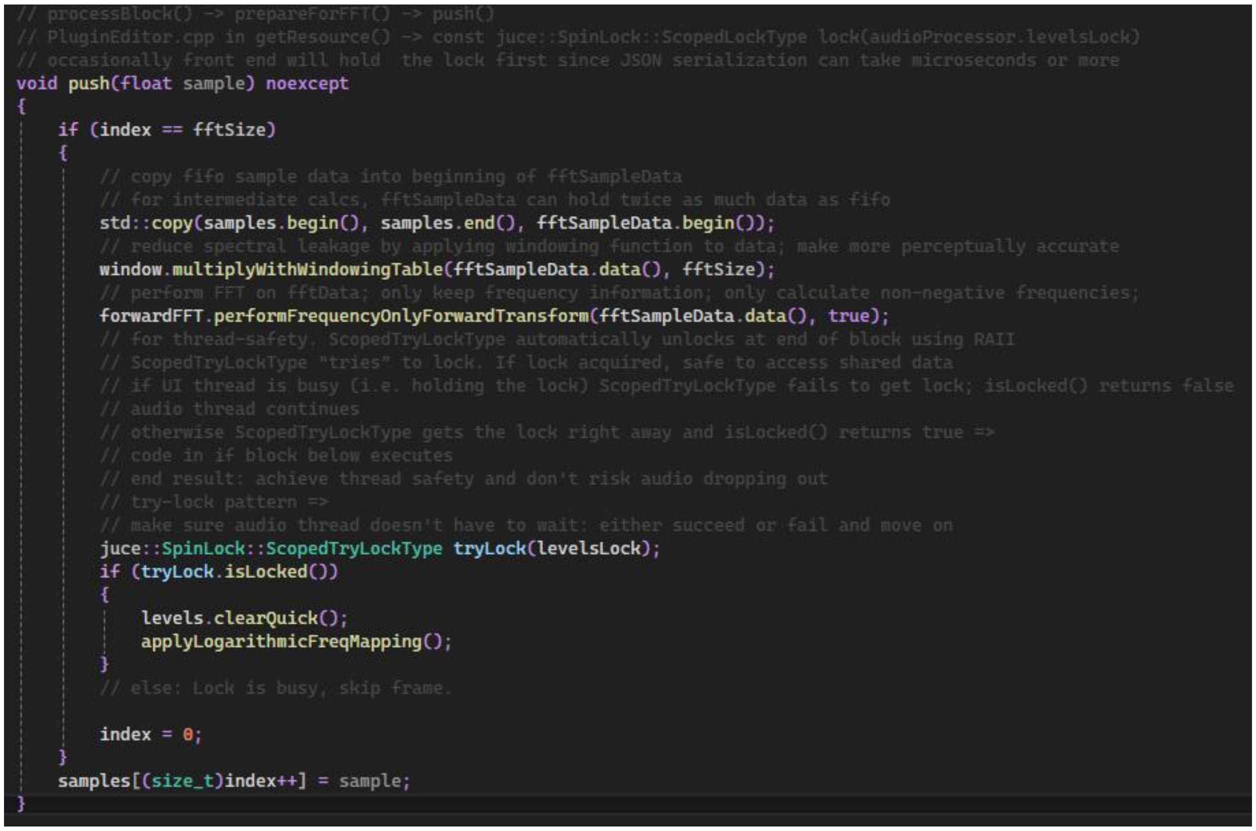
- The *Fifo* struct is initialized with its static and instance members:



```
struct Fifo
{
    static constexpr auto fftOrder{ 11 };
    static constexpr auto fftSize{ 1 << fftOrder };
    static constexpr auto fftDataSize{ fftSize * 2 };
    static constexpr auto scopeSize{ fftSize / 4 };

    juce::dsp::FFT forwardFFT{ fftOrder };
    juce::dsp::WindowingFunction<float> window{ fftSize, juce::dsp::WindowingFunction<float>::hann };
    std::array<float, fftSize> samples;
    // for holding FFT processed sample data; FFT algorithm requires double space
    std::array<float, fftDataSize> fftSampleData;
    int index{ 0 };

    // store normalized levels derived from fftData using applyLogarithmicFreqMapping() below
    juce::Array<juce::var> levels;
    juce::SpinLock levelsLock;
```

-
- *fftSize* is equivalent to Block Length **BL** discussed in [DSP Background]. For consistency 1 is bitshifted left *fftOrder* times. It equals 2^{11} = 2048
- The FFT algorithm requires *fftSize* * 2 space for holding intermediate values.
- *scopeSize* refers to the number of bins that will be displayed. fftSize is 2048. When we perform the actual FFT, we will discard negative frequency values, so that means 1024 bins will remain. In *applyLogarithmicFreqMapping()* function below, only the critical frequency information will be retained, and 512 levels will be sent to the frontend for consumption.
- *forwardFFT* is initialized with *fftOrder.*
- The window object is initialized with fftSize and the specified Hann windowing function discussed in [DSP Background].
- The *samples* array is initialized with size *fftSize.*
- *fftSampleData* is initialized with size twice the length of *samples*.
- The *levels* array (for frontend consumption) and the mutex lock are initialized.

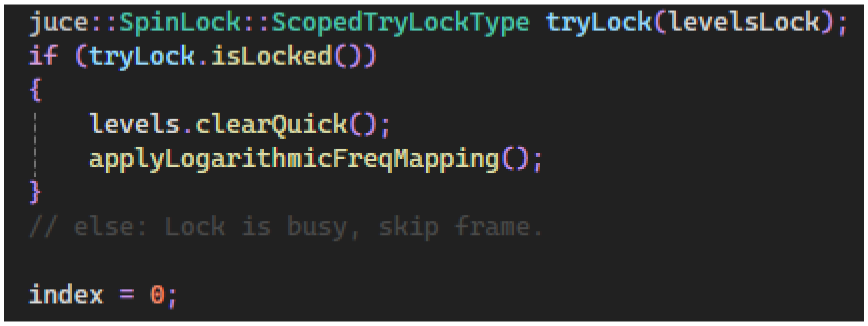
Fifo#push() function



```
// processBlock() -> prepareForFFT() -> push()
// PluginEditor.cpp in getResource() -> const juce::SpinLock::ScopedLockType lock(audioProcessor.levelsLock)
// occasionally front end will hold  the lock first since JSON serialization can take microseconds or more
void push(float sample) noexcept
{
    if (index == fftSize)
    {
        // copy fifo sample data into beginning of fftSampleData
        // for intermediate calcs, fftSampleData can hold twice as much data as fifo
        std::copy(samples.begin(), samples.end(), fftSampleData.begin());
        // reduce spectral leakage by applying windowing function to data; make more perceptually accurate
        window.multiplyWithWindowingTable(fftSampleData.data(), fftSize);
        // perform FFT on fftData; only keep frequency information; only calculate non-negative frequencies;
        forwardFFT.performFrequencyOnlyForwardTransform(fftSampleData.data(), true);
        // for thread-safety. ScopedTryLockType automatically unlocks at end of block using RAII
        // ScopedTryLockType "tries" to lock. If lock acquired, safe to access shared data
        // if UI thread is busy (i.e. holding the lock) ScopedTryLockType fails to get lock; isLocked() returns false
        // audio thread continues
        // otherwise ScopedTryLockType gets the lock right away and isLocked() returns true =>
        // code in if block below executes
        // end result: achieve thread safety and don't risk audio dropping out
        // try-lock pattern =>
        // make sure audio thread doesn't have to wait: either succeed or fail and move on
        juce::SpinLock::ScopedTryLockType tryLock(levelsLock);
        if (tryLock.isLocked())
        {
            levels.clearQuick();
            applyLogarithmicFreqMapping();
        }
        // else: Lock is busy, skip frame.

        index = 0;
    }
    samples[(size_t)index++] = sample;
}
```

- The *push()* function allows for *fftSize* (2048) samples to be pushed to the *samples* array.
- When the Fifo queue is full, *samples* is copied over to *fftSampleData*.
- The Hann windowing function is applied to *fftSampleData* to prepare for the frequency only forward Fourier transform.
- FFT is performed on *fftSampleData*. Only frequency magnitude information is kept. Phase information and negative frequencies are discarded.

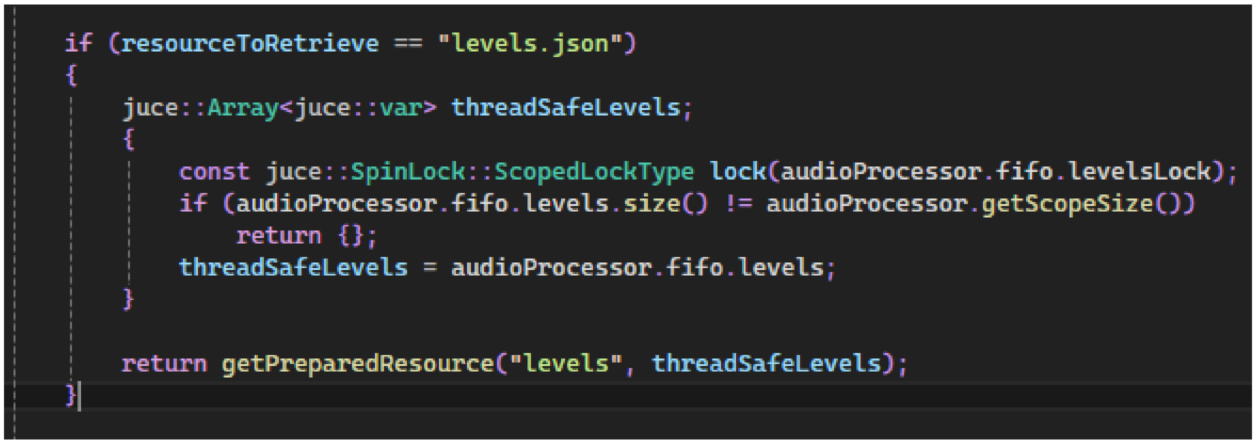
Thread Safety / SpinLock



```
juce::SpinLock::ScopedTryLockType tryLock(levelsLock);
if (tryLock.isLocked())
{
    levels.clearQuick();
    applyLogarithmicFreqMapping();
}
// else: Lock is busy, skip frame.

index = 0;
```

- Above, *levelsLock,* initialized earlier is passed to a *ScopedTryLockType#tryLock()* function. If the lock is acquired, the levels array is reset and *applyLogarithmicFreqMapping()* is called, pushing fresh level data to the *levels* array.

- In *PluginEditor*, which prepares the *levels* feature for the frontend, a similar mutex is employed. *ScopedLockType* uses RAII (Resource Acquisition Is Initialization), so the lock is released when *lock* goes out of scope after *threadSafeLevels is copied the levels from audioProcessor.fifo*:



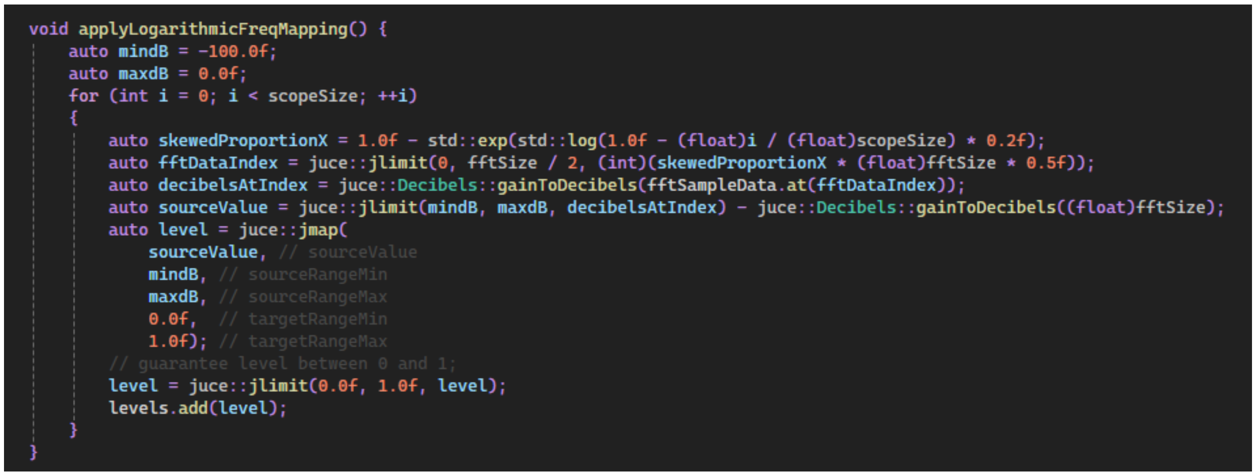
```
if (resourceToRetrieve == "levels.json")
{
    juce::Array<juce::var> threadSafeLevels;
    {
        const juce::SpinLock::ScopedLockType lock(audioProcessor.fifo.levelsLock);
        if (audioProcessor.fifo.levels.size() != audioProcessor.getScopeSize())
            return {};
        threadSafeLevels = audioProcessor.fifo.levels;
    }

    return getPreparedResource("levels", threadSafeLevels);
}
```

-
- The juce::SpinLock is a non-blocking lock. If it fails to get the lock, the frontend simply goes one frame without most recent data, which will be imperceptible.

Fifo#applyLogarithmicFreqMapping() function

The following function is adapted from a juce tutorial: [Spectrum Analyzer]. I have customized it to make it more readable and adapted it to work the *levels* array.

The primary use for this function is to match the output bins with human hearing sensitivity. More detail is preserved at the lower end of the frequency spectrum where the human ear is more sensitive.



```
void applyLogarithmicFreqMapping() {
    auto mindB = -100.0f;
    auto maxdB = 0.0f;
    for (int i = 0; i < scopeSize; ++i)
    {
        auto skewedProportionX = 1.0f - std::exp(std::log(1.0f - (float)i / (float)scopeSize) * 0.2f);
        auto fftDataIndex = juce::jlimit(0, fftSize / 2, (int)(skewedProportionX * (float)fftSize * 0.5f));
        auto decibelsAtIndex = juce::Decibels::gainToDecibels(fftSampleData.at(fftDataIndex));
        auto sourceValue = juce::jlimit(mindB, maxdB, decibelsAtIndex) - juce::Decibels::gainToDecibels((float)fftSize);
        auto level = juce::jmap(
            sourceValue, // sourceValue
            mindB, // sourceRangeMin
            maxdB, // sourceRangeMax
            0.0f,  // targetRangeMin
            1.0f); // targetRangeMax
        // guarantee level between 0 and 1;
        level = juce::jlimit(0.0f, 1.0f, level);
        levels.add(level);
    }
}
```

- There are 512 particles in *ParticleWave*. We therefore need to send 512 points of data. The data of *fftSampleData* contains gain units that represent the amplitude of each frequency bin.
- The x-axis is skewed to use a logarithmic scale since that is what *gainToDecibels()* outputs. We store this value in *skewedProportionX*.
- *skewedProportionX* is then used to obtain the amplitude at the appropriate index for the fftData.
- This amplitude is then normalized to a float range between 0 and 1.

Frontend

There are many animation features and functions, so only select major features will be covered: one from *ParticleWave,* and two from *NebulaSystem*.
For full code see [Appendix, Custom JavaScript Animation Code];

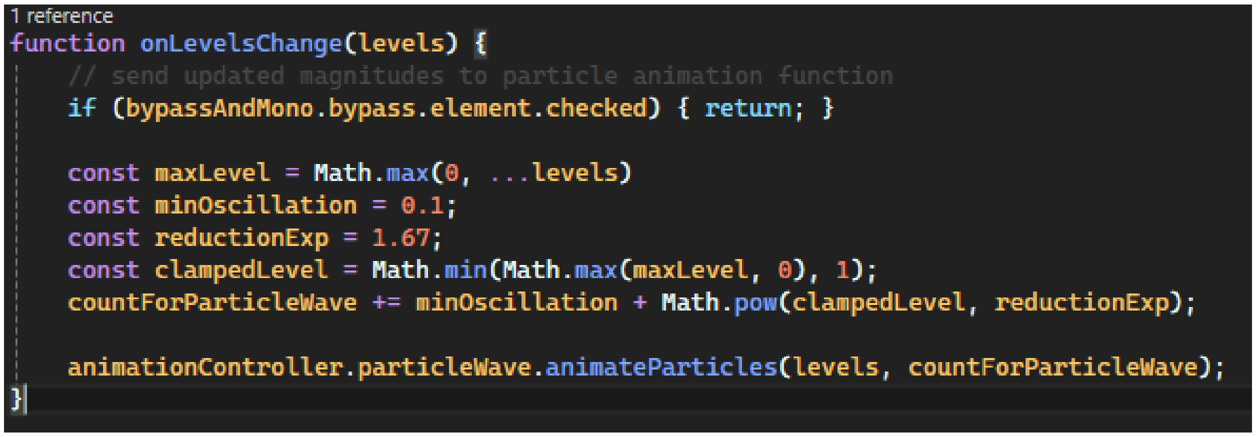
As discussed in [Frontend Architecture], event handling follows a similar procedure for each respective parameter change or feature received.

Standard course of events:

1. Frontend receives a *PluginEditor::webView* emitted event.
2. Frontend requests the appropriate resource.
3. The client calls an appropriate event handler.
4. The event handler tells *AnimationController* to do something with the received resource.

ParticleWave

Starting at step 4:



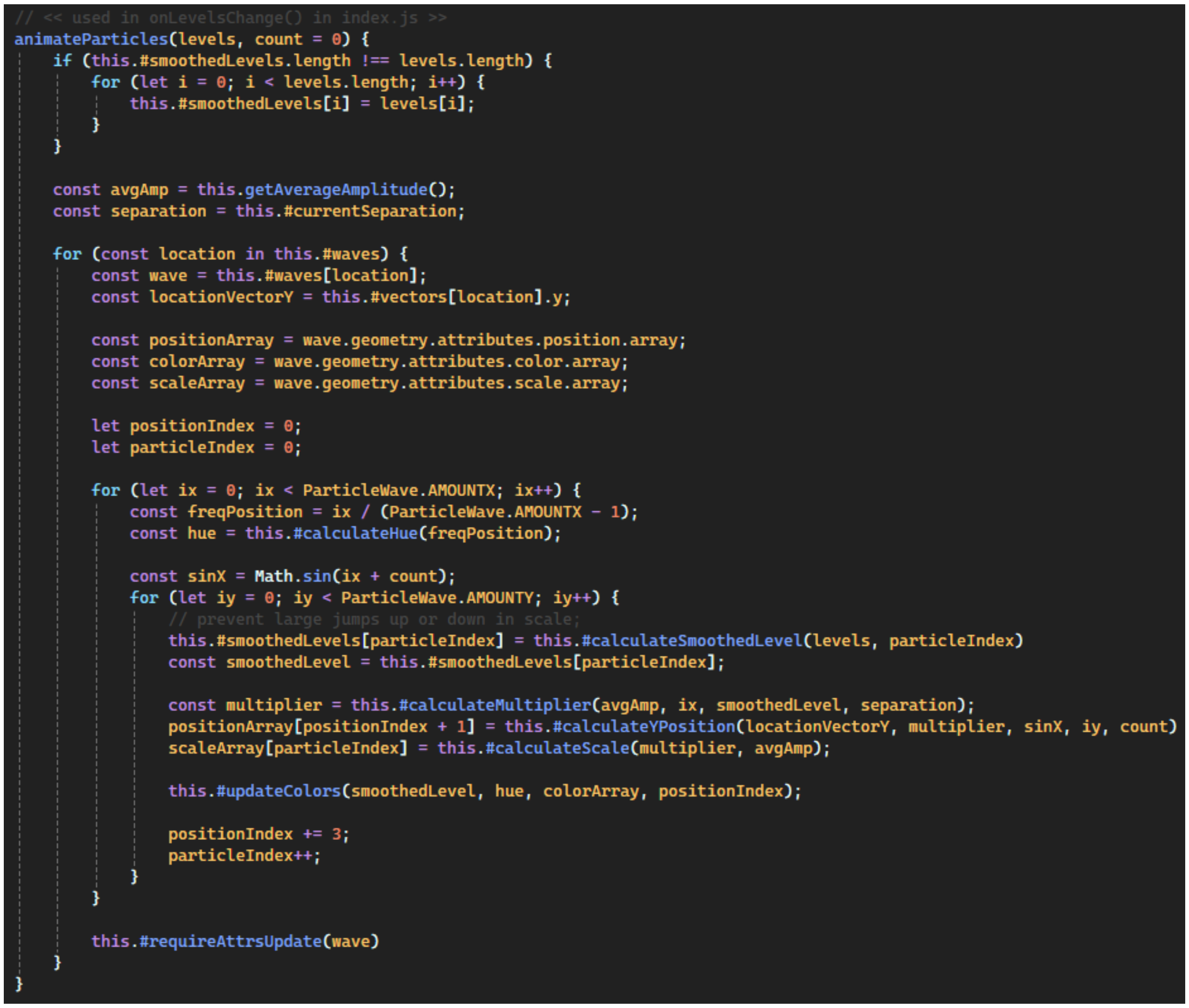
```
1 reference
function onLevelsChange(levels) {
    // send updated magnitudes to particle animation function
    if (bypassAndMono.bypass.element.checked) { return; }

    const maxLevel = Math.max(0, ...levels)
    const minOscillation = 0.1;
    const reductionExp = 1.67;
    const clampedLevel = Math.min(Math.max(maxLevel, 0), 1);
    countForParticleWave += minOscillation + Math.pow(clampedLevel, reductionExp);

    animationController.particleWave.animateParticles(levels, countForParticleWave);
}
```

- New levels are received from the backend.

- Some setup is performed to ensure the particle wave remains animated. *maxLevel* determines how fast the wave oscillates.
- *countForParticleWave*
- animationController tells its particleWave member to animate the particles.

ParticleWave#animateParticles()

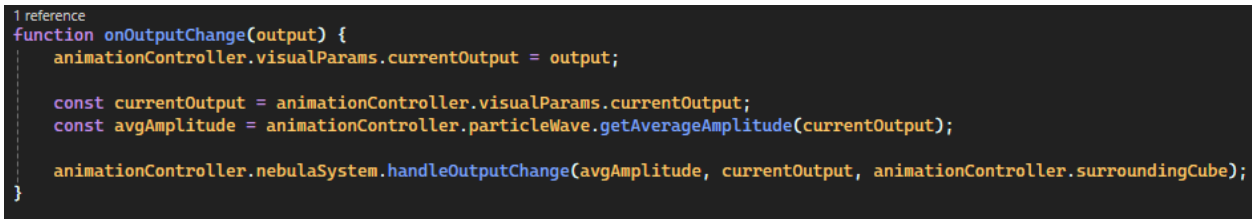
The particle wave itself is a ThreeJS *Points* Mesh composed with a *BufferGeometry* and a ThreeJS *ShaderMaterial, which is initialized with* vertex and fragment shaders defined in index.html. A *BufferGeometry* is a flexible, high performance geometry object in ThreeJS. It allows custom attributes. For this implementation, we continuously update the scale, color, and position of the *Points* mesh.



```
// << used in onLevelsChange() in index.js >>
animateParticles(levels, count = 0) {
    if (this.#smoothedLevels.length !== levels.length) {
        for (let i = 0; i < levels.length; i++) {
            this.#smoothedLevels[i] = levels[i];
        }
    }

    const avgAmp = this.getAverageAmplitude();
    const separation = this.#currentSeparation;

    for (const location in this.#waves) {
        const wave = this.#waves[location];
        const locationVectorY = this.#vectors[location].y;

        const positionArray = wave.geometry.attributes.position.array;
        const colorArray = wave.geometry.attributes.color.array;
        const scaleArray = wave.geometry.attributes.scale.array;

        let positionIndex = 0;
        let particleIndex = 0;

        for (let ix = 0; ix < ParticleWave.AMOUNTX; ix++) {
            const freqPosition = ix / (ParticleWave.AMOUNTX - 1);
            const hue = this.#calculateHue(freqPosition);

            const sinX = Math.sin(ix + count);
            for (let iy = 0; iy < ParticleWave.AMOUNTY; iy++) {
                // prevent large jumps up or down in scale
                this.#smoothedLevels[particleIndex] = this.#calculateSmoothedLevel(levels, particleIndex)
                const smoothedLevel = this.#smoothedLevels[particleIndex];

                const multiplier = this.#calculateMultiplier(avgAmp, ix, smoothedLevel, separation);
                positionArray[positionIndex + 1] = this.#calculateYPosition(locationVectorY, multiplier, sinX, iy, count)
                scaleArray[particleIndex] = this.#calculateScale(multiplier, avgAmp);

                this.#updateColors(smoothedLevel, hue, colorArray, positionIndex);

                positionIndex += 3;
                particleIndex++;
            }
        }

        this.#requireAttrsUpdate(wave)
    }
}
```

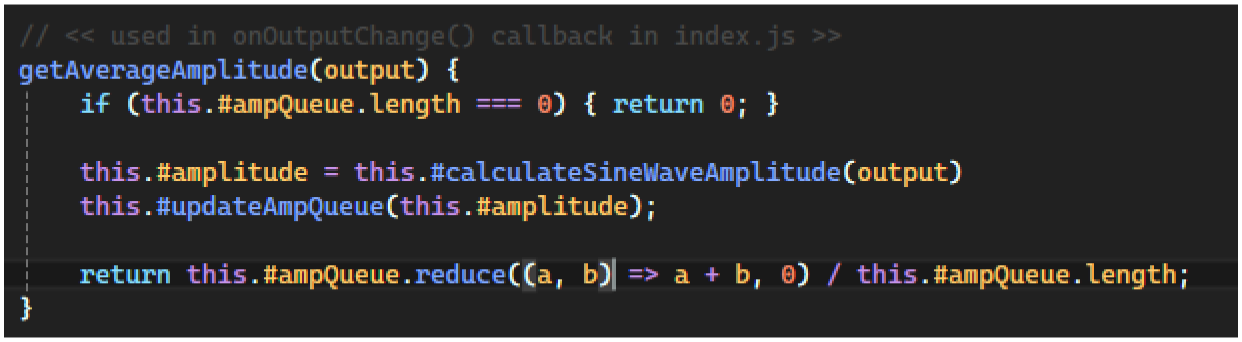
- If *smoothedLevels array* is empty, fill it with incoming *levels* data.

- Amplitude refers to the amplitude of the particle wave's sine wave position animation (see ParticleWave#calculateYPosition()). Amplitude depends on the currentOutput level which is set in the client's output change handler:



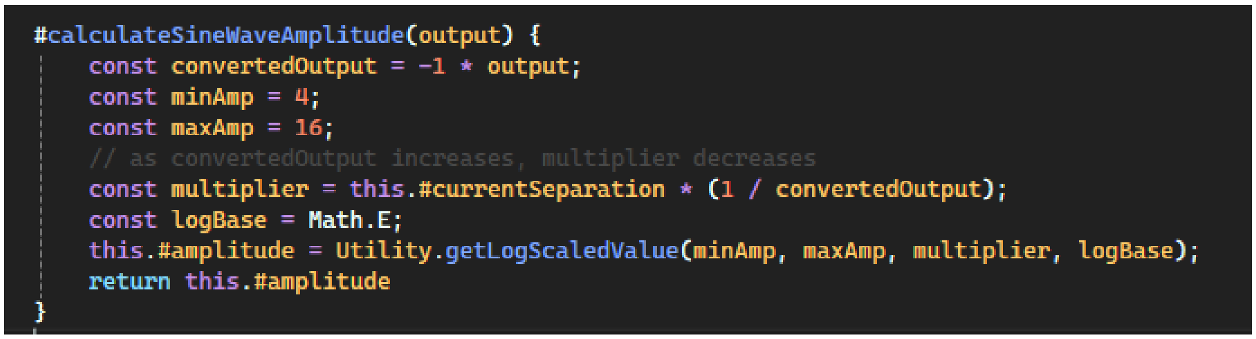
```
1 reference
function onOutputChange(output) {
    animationController.visualParams.currentOutput = output;

    const currentOutput = animationController.visualParams.currentOutput;
    const avgAmplitude = animationController.particleWave.getAverageAmplitude(currentOutput);

    animationController.nebulaSystem.handleOutputChange(avgAmplitude, currentOutput, animationController.surroundingCube);
}
```

- *VisualParams* is responsible for maintaining the current state of all primary features derived from backend events. *animationController* tells *visualParams* to update the *currentOutput.*
- *currentOutput* is used to calculate the average amplitude required for the sine wave animation of *ParticleWave:*



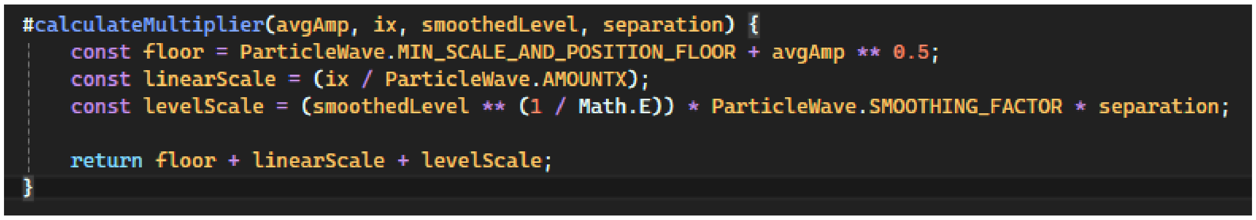
```
// << used in onOutputChange() callback in index.js >>
getAverageAmplitude(output) {
    if (this.#ampQueue.length === 0) { return 0; }

    this.#amplitude = this.#calculateSineWaveAmplitude(output)
    this.#updateAmpQueue(this.#amplitude);

    return this.#ampQueue.reduce((a, b) => a + b, 0) / this.#ampQueue.length;
}
```

- An *ampQueue* holds the 5 most recent amplitude values so that there are no "spikey jumps" in the animation when output level suddenly increases.
- output is on a decibel scale with 0 being the highest possible output and so it is converted to positive. *currentSeparation* is how much space is between each 'point' of the *Points* mesh:



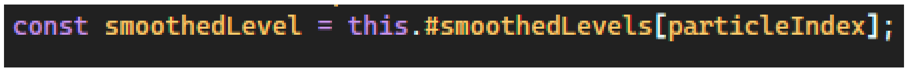
```
#calculateSineWaveAmplitude(output) {
    const convertedOutput = -1 * output;
    const minAmp = 4;
    const maxAmp = 16;
    // as convertedOutput increases, multiplier decreases
    const multiplier = this.#currentSeparation * (1 / convertedOutput);
    const logBase = Math.E;
    this.#amplitude = Utility.getLogScaledValue(minAmp, maxAmp, multiplier, logBase);
    return this.#amplitude
}
```

- Next in *animateParticles(),* a multiplier is calculated using the current average amplitude of the particle wave and the current smoothed level. This multiplier is then used to change the Y position and the scale of the respective particle at *particleIndex.*



```
#calculateMultiplier(avgAmp, ix, smoothedLevel, separation) {
    const floor = ParticleWave.MIN_SCALE_AND_POSITION_FLOOR + avgAmp ** 0.5;
    const linearScale = (ix / ParticleWave.AMOUNTX);
    const levelScale = (smoothedLevel ** (1 / Math.E)) * ParticleWave.SMOOTHING_FACTOR * separation;

    return floor + linearScale + levelScale;
}
```

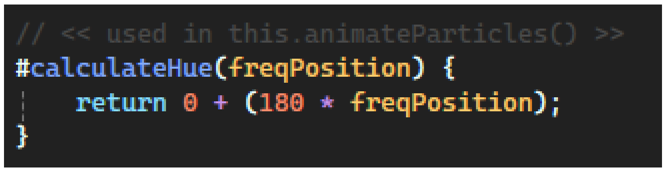
- Levels are smoothed to prevent sudden jumps in scale. A *smoothedLevel* corresponds to the current *particleIndex* since this function loops over a size 32 x 16 = 512 collection of points.



```
const smoothedLevel = this.#smoothedLevels[particleIndex];
```

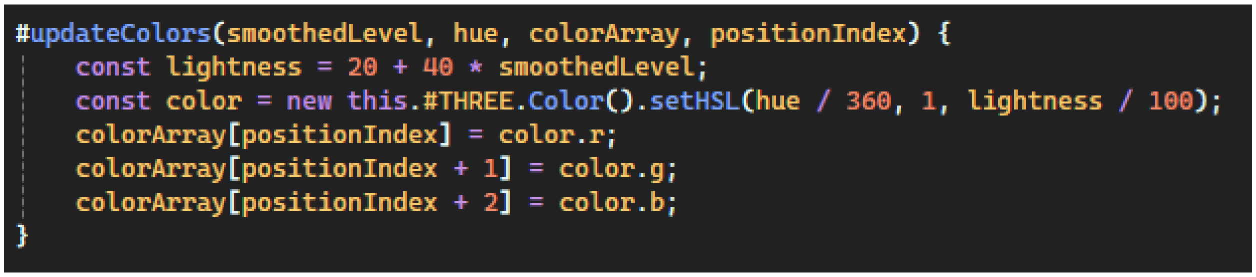
This ensures that a given point in the wave scales according to its FFT-calculated magnitude.

- The color of these points is calculated using a simple function that shifts on a gradient based on a particle point's corresponding frequency position. Low frequencies are red, and the highest frequencies are cyan. This resulted in a gentler visual experience than including the entire color spectrum.



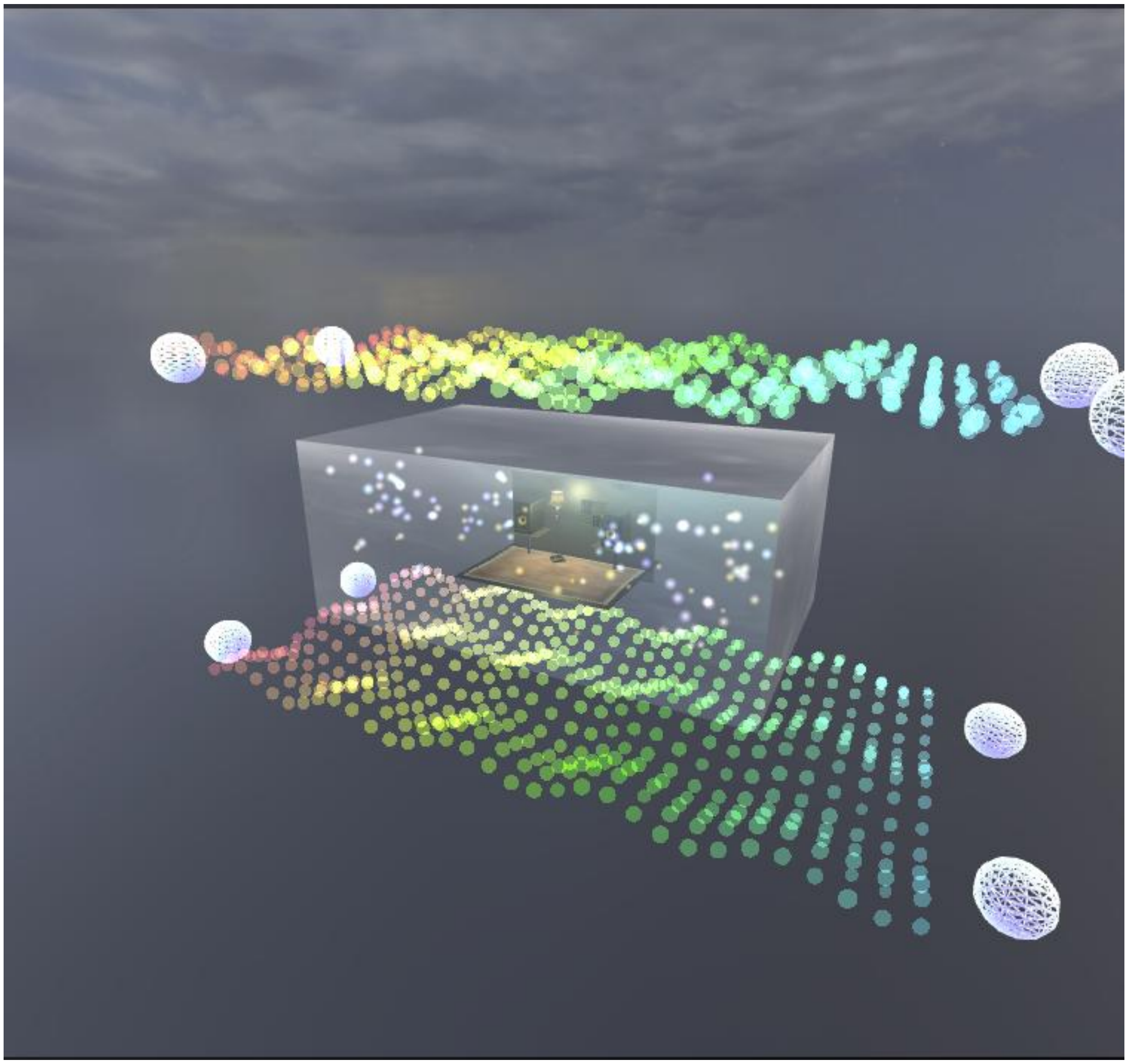
```
// << used in this.animateParticles() >>
#calculateHue(freqPosition) {
    return 0 + (180 * freqPosition);
}
```

and then the color attributes are updated:



```
#updateColors(smoothedLevel, hue, colorArray, positionIndex) {
    const lightness = 20 + 40 * smoothedLevel;
    const color = new this.#THREE.Color().setHSL(hue / 360, 1, lightness / 100);
    colorArray[positionIndex] = color.r;
    colorArray[positionIndex + 1] = color.g;
    colorArray[positionIndex + 2] = color.b;
}
```

- Red points on the left of [Figure 3] are low frequencies, and the green/cyan points are high-mid/high frequencies, and the *Points* mesh responds to environment lighting. Below is a screenshot where a virtual instrument lights up the entire frequency spectrum:

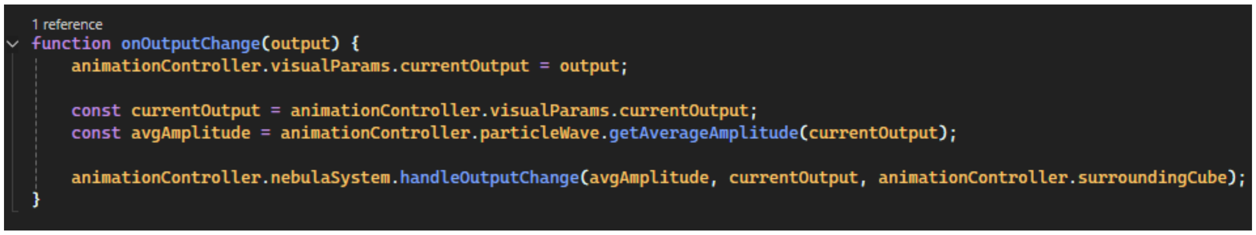


Nebula System

NebulaSystem#handleOutputChange()

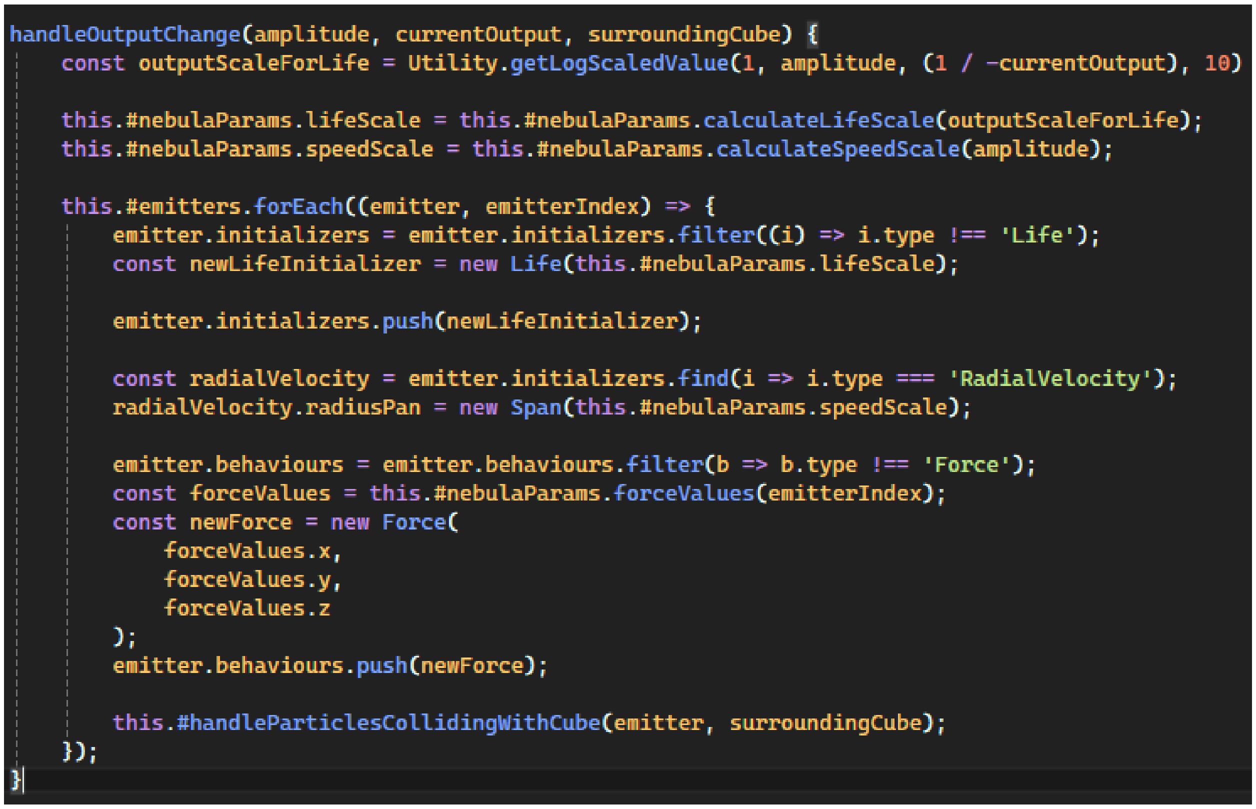
The output level has an influence on several features. It controls the amplitude of the particle wave, the following Three Nebula features: the life of emitted sprites, the radial velocity with which particles are emitted, and the x, y, and z force behavior.

We start again on step 4 with the client telling animationController to perform a task:



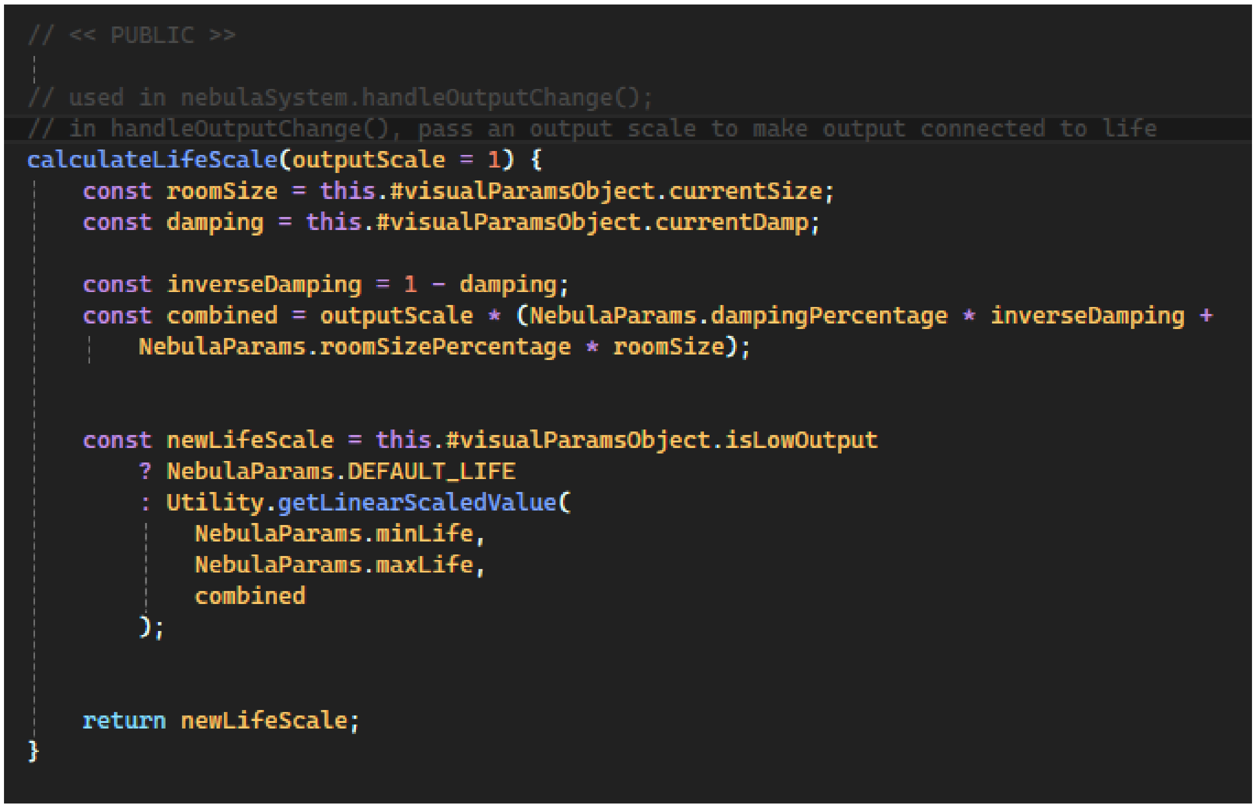
```
1 reference
function onOutputChange(output) {
    animationController.visualParams.currentOutput = output;

    const currentOutput = animationController.visualParams.currentOutput;
    const avgAmplitude = animationController.particleWave.getAverageAmplitude(currentOutput);

    animationController.nebulaSystem.handleOutputChange(avgAmplitude, currentOutput, animationController.surroundingCube);
}
```

Here we focus on the last command, *NebulaSystem.handleOutputChange()*.



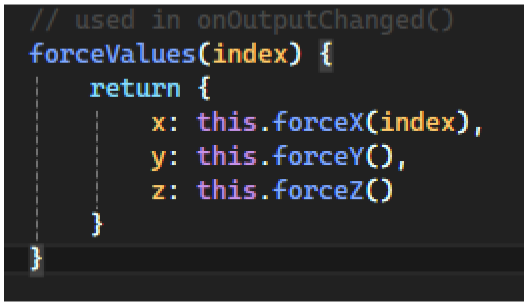
```
handleOutputChange(amplitude, currentOutput, surroundingCube) {
    const outputScaleForLife = Utility.getLogScaledValue(1, amplitude, (1 / -currentOutput), 10)

    this.#nebulaParams.lifeScale = this.#nebulaParams.calculateLifeScale(outputScaleForLife);
    this.#nebulaParams.speedScale = this.#nebulaParams.calculateSpeedScale(amplitude);

    this.#emitters.forEach((emitter, emitterIndex) => {
        emitter.initializers = emitter.initializers.filter((i) => i.type !== 'Life');
        const newLifeInitializer = new Life(this.#nebulaParams.lifeScale);

        emitter.initializers.push(newLifeInitializer);

        const radialVelocity = emitter.initializers.find(i => i.type === 'RadialVelocity');
        radialVelocity.radiusPan = new Span(this.#nebulaParams.speedScale);

        emitter.behaviours = emitter.behaviours.filter(b => b.type !== 'Force');
        const forceValues = this.#nebulaParams.forceValues(emitterIndex);
        const newForce = new Force(
            forceValues.x,
            forceValues.y,
            forceValues.z
        );
        emitter.behaviours.push(newForce);

        this.#handleParticlesCollidingWithCube(emitter, surroundingCube);
    });
}
```

- A log-scaled value of output is used to calculate the life of an emitted particle. The inverse of negative *currentOutput* is used because output is greater as it approaches zero, e.g., an output of -5 dB will turn into 1/5 => 0.2; an output of -60 dB will turn into 1/60 => 0.0167.
- *NebulaParams* is the class responsible for maintaining the state of parameters that control the behavior of the Three Nebula particle system. Most of the parameters of *NebulaParams* are influenced by *VisualParams* and so when *NebulaParams* is initialized, it is injected with *visualParamsObject* as a dependency. In *NebulaParams#calculateLifeScale(),* this dependency is accessed to get the current roomSize and damping values.



```
// << PUBLIC >>

// used in nebulaSystem.handleOutputChange();
// in handleOutputChange(), pass an output scale to make output connected to life
calculateLifeScale(outputScale = 1) {
    const roomSize = this.#visualParamsObject.currentSize;
    const damping = this.#visualParamsObject.currentDamp;

    const inverseDamping = 1 - damping;
    const combined = outputScale * (NebulaParams.dampingPercentage * inverseDamping +
        NebulaParams.roomSizePercentage * roomSize);


    const newLifeScale = this.#visualParamsObject.isLowOutput
        ? NebulaParams.DEFAULT_LIFE
        : Utility.getLinearScaledValue(
            NebulaParams.minLife,
            NebulaParams.maxLife,
            combined
        );


    return newLifeScale;
}
```

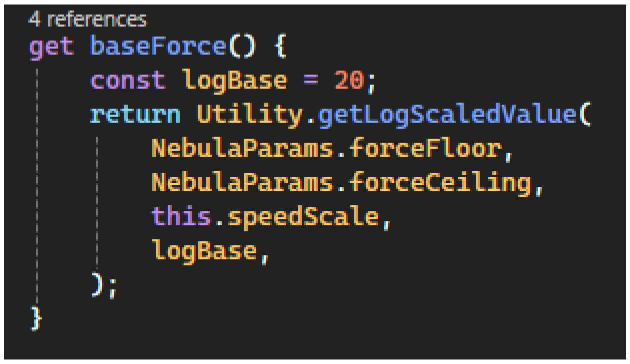
- Inverse damping is used so that as damping decreases, the life of the emitted sprites increases and vice versa. *dampingPercentage* defaults to 0.6; *roomSizePercentage* defaults to 0.4. These values influence how much of an effect either parameter has on the life of an emitted particle. Large rooms with low damp values will have particles that live the longest.
- When *calculateLifeScale()* is called from *handleOutputChange()* it is passed the log-scaled value discussed above so that *outputLevel* also influences the life of the emitted particle. This is observed in the variable *combined,* which is then used as a linear scale multiplier to return *newLifeScale* to *nebulaParams*.
- *nebulaParams.speedScale* is calculated similarly but with a log-scale.
- nebulaParams.forceValues returns an object with each respective x, y, and z force as a key:



```
// used in onOutputChanged()
forceValues(index) {
    return {
        x: this.forceX(index),
        y: this.forceY(),
        z: this.forceZ()
    }
}
```

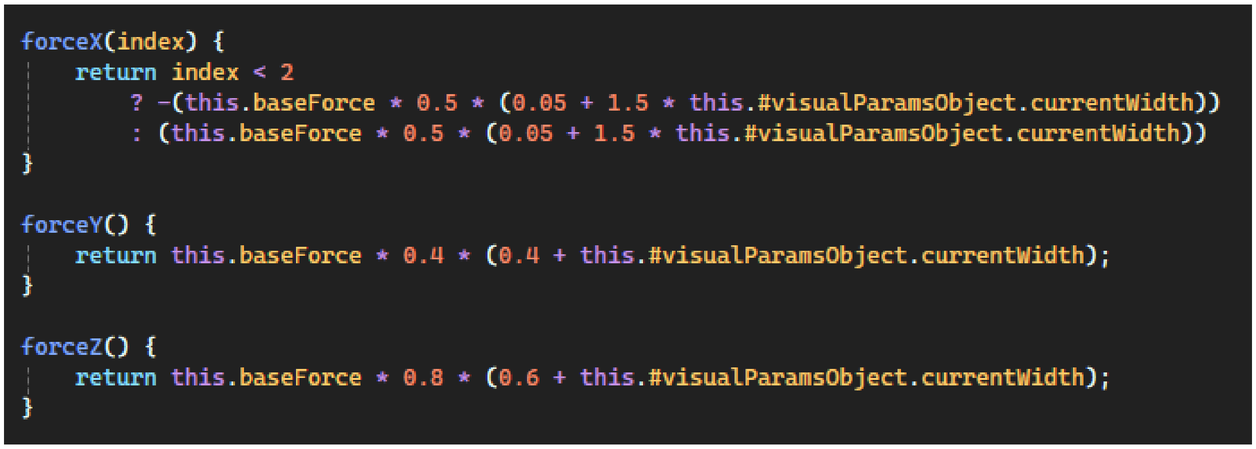
-

- Each method forceX(), forceY(), and forceZ() utilizes a base force that uses nebulaParams.speedScale as a multiplier:



```
4 references
get baseForce() {
    const logBase = 20;
    return Utility.getLogScaledValue(
        NebulaParams.forceFloor,
        NebulaParams.forceCeiling,
        this.speedScale,
        logBase,
    );
}
```

-
- The force for each is then modulated by the injected visualParamsObject's currentWidth value so that force doesn't dominate width changes too much and width changes can be perceived. When width parameter decreases, there is less stereo effect and vice versa.
- This is represented visually as:
    - low width => particles emitted from left and right speaker converge towards a center line:
    - high width => particles emitted from left and right speaker are spread more towards left and right walls



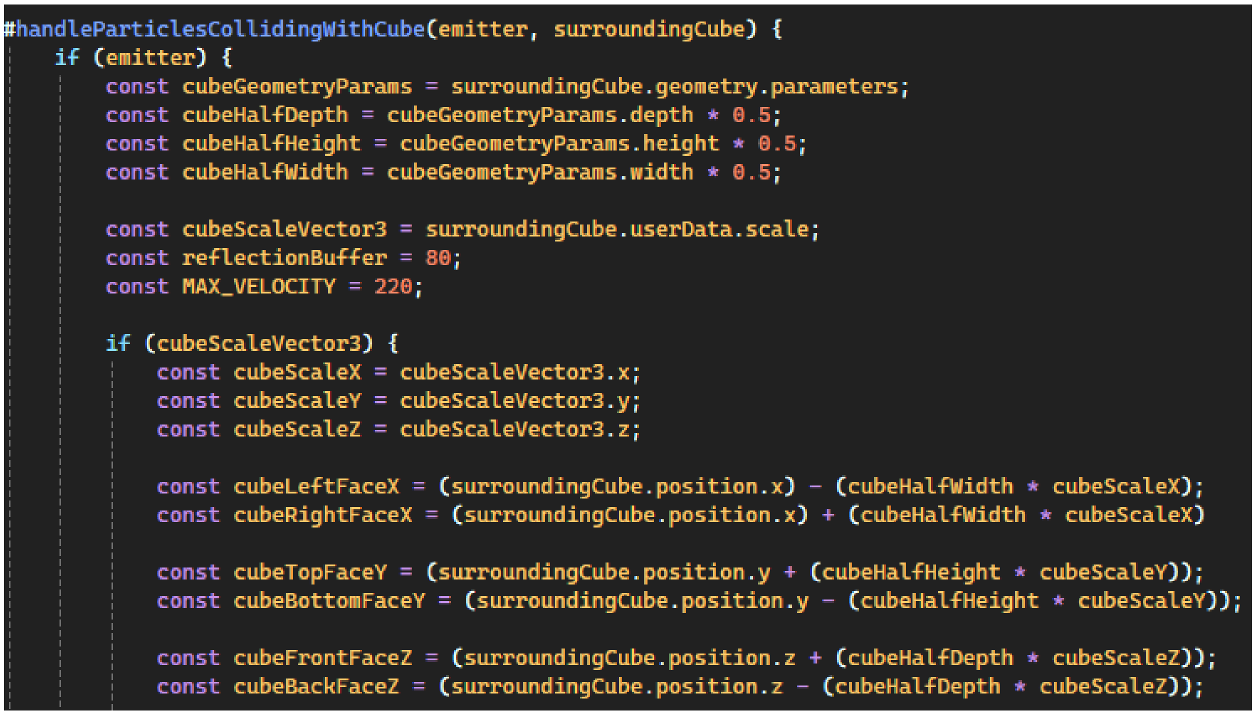
```
forceX(index) {
    return index < 2
        ? -(this.baseForce * 0.5 * (0.05 + 1.5 * this.#visualParamsObject.currentWidth))
        : (this.baseForce * 0.5 * (0.05 + 1.5 * this.#visualParamsObject.currentWidth))
}

forceY() {
    return this.baseForce * 0.4 * (0.4 + this.#visualParamsObject.currentWidth);
}

forceZ() {
    return this.baseForce * 0.8 * (0.6 + this.#visualParamsObject.currentWidth);
}
```

-
- The final call of *NebulaSystem.handleOutputChange(), handleParticlesCollidingWithCube(),* deserves its own section.

NebulaSystem.handleParticlesCollidingWithCube()

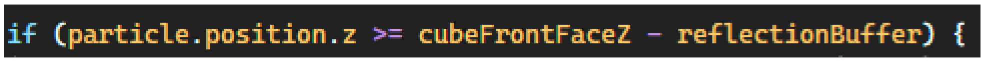
A large part of the visual metaphor for this project is using emitted particles to simulate sound reflections in a room. *AnimationController*'s *surroundingCube* is a transparent glass-like mesh generated by the *BoxFactory*. The idea is that as *roomSize* scales up and down, so does the *surroundingCube*. Acoustic engineers sometimes demonstrate sound reflections with nerf guns or tennis balls. A major challenge was figuring out how to "bounce" the emitted sprites inside the box without escaping and doing so in a moderately aesthetically pleasing way.

1. *surroundingCube* needs to be parameterized so that it doesn't have to be a member variable of *NebulaSystem* and have its state managed in multiple places.
2. The dimensions of *surroundingCube* can be updated at any moment. This function needs the most up-to-date state. This is achieved by accessing the custom *userData* object that is created in *AnimationController*.
3. Determine arithmetic necessary to define the boundary of the wall that a sprite should not cross. The screenshot below demonstrates the straightforward arithmetic involved in solving this problem. A reflection buffer is to prevent any strays that may escape due to sudden force changes by frame.



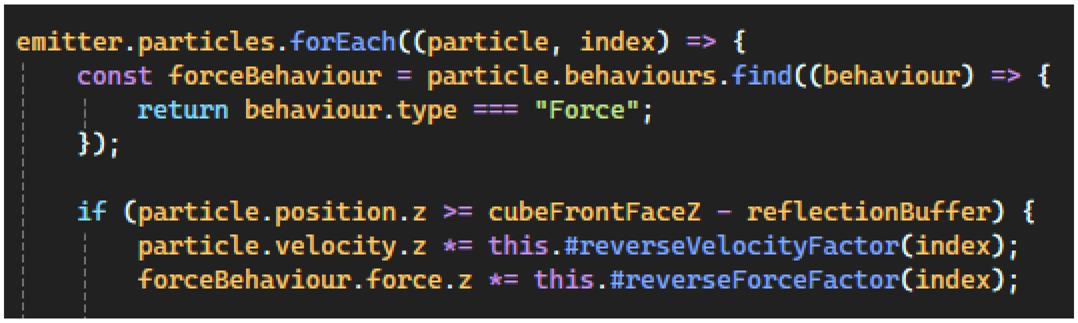
```
#handleParticlesCollidingWithCube(emitter, surroundingCube) {
    if (emitter) {
        const cubeGeometryParams = surroundingCube.geometry.parameters;
        const cubeHalfDepth = cubeGeometryParams.depth * 0.5;
        const cubeHalfHeight = cubeGeometryParams.height * 0.5;
        const cubeHalfWidth = cubeGeometryParams.width * 0.5;

        const cubeScaleVector3 = surroundingCube.userData.scale;
        const reflectionBuffer = 80;
        const MAX_VELOCITY = 220;

        if (cubeScaleVector3) {
            const cubeScaleX = cubeScaleVector3.x;
            const cubeScaleY = cubeScaleVector3.y;
            const cubeScaleZ = cubeScaleVector3.z;

            const cubeLeftFaceX = (surroundingCube.position.x) - (cubeHalfWidth * cubeScaleX);
            const cubeRightFaceX = (surroundingCube.position.x) + (cubeHalfWidth * cubeScaleX)

            const cubeTopFaceY = (surroundingCube.position.y + (cubeHalfHeight * cubeScaleY));
            const cubeBottomFaceY = (surroundingCube.position.y - (cubeHalfHeight * cubeScaleY));

            const cubeFrontFaceZ = (surroundingCube.position.z + (cubeHalfDepth * cubeScaleZ));
            const cubeBackFaceZ = (surroundingCube.position.z - (cubeHalfDepth * cubeScaleZ));
```

4. Iterate over each of the emitter's particles and check it a boundary has been crossed:



```
if (particle.position.z >= cubeFrontFaceZ - reflectionBuffer) {
```

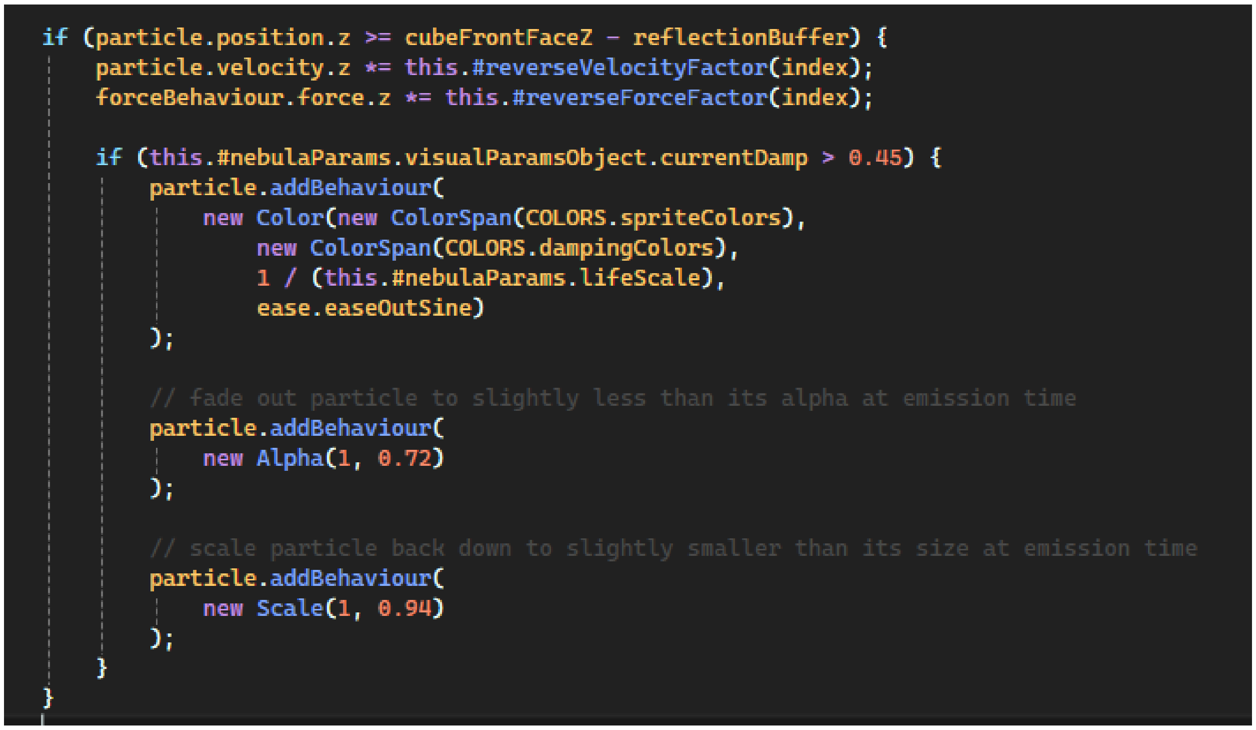
5. Determine how to reflect particles off boundaries.
    a. Reverse their force behavior and reverse their velocity:



```
emitter.particles.forEach((particle, index) => {
    const forceBehaviour = particle.behaviours.find((behaviour) => {
        return behaviour.type === "Force";
    });

    if (particle.position.z >= cubeFrontFaceZ - reflectionBuffer) {
        particle.velocity.z *= this.#reverseVelocityFactor(index);
        forceBehaviour.force.z *= this.#reverseForceFactor(index);
```

6. Incorporate existing features that are relevant to an emitted sprite colliding with a boundary in the visual reverb metaphor. Again, damping is important to both the sound and visual metaphor of the reverb effect.

   The following code snippet has the effect of changing the color when a particle collides with the front facing wall of *surroundingCube* if *currentDamp* is greater than 0.45 (where I personally perceived the damping effect start to kick in more noticably). *nebulaParams.lifeScale* determines how long the color change lasts. Additionally, a particle will fade out from 1 to slightly less than its alpha at emission time and the particle will scale down slightly:

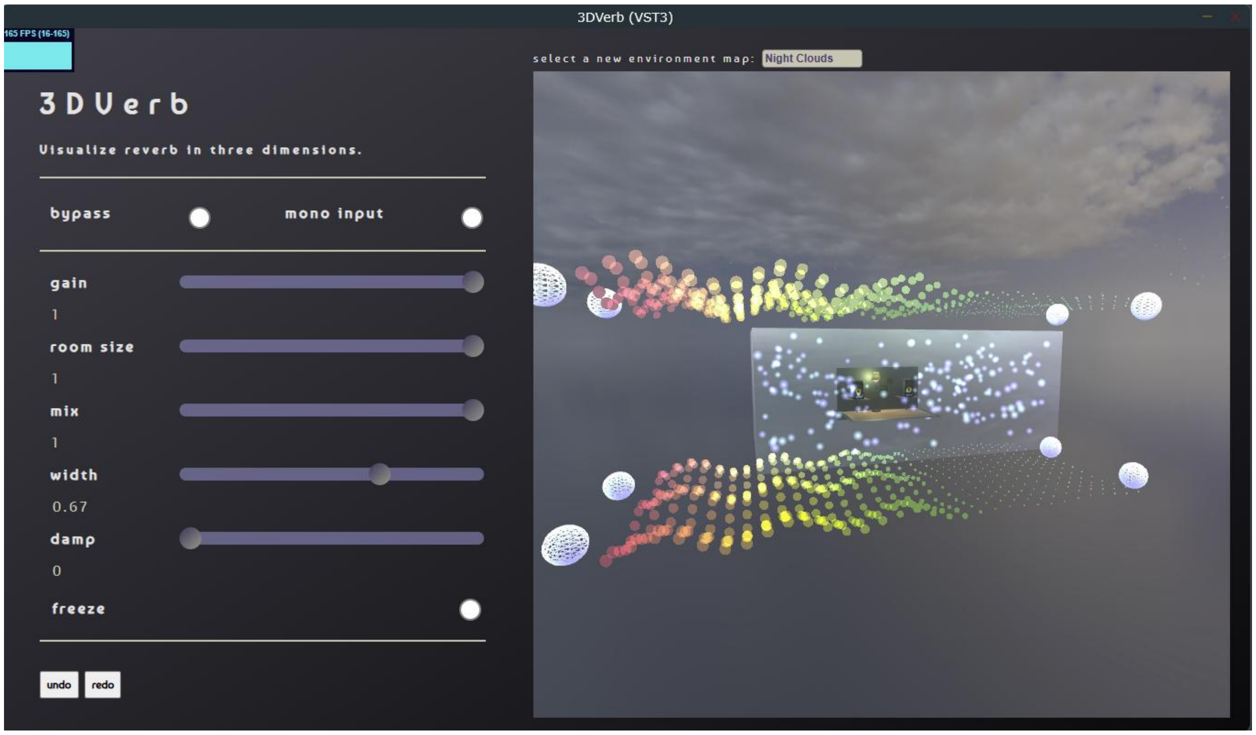


```
if (particle.position.z >= cubeFrontFaceZ - reflectionBuffer) {
    particle.velocity.z *= this.#reverseVelocityFactor(index);
    forceBehaviour.force.z *= this.#reverseForceFactor(index);

    if (this.#nebulaParams.visualParamsObject.currentDamp > 0.45) {
        particle.addBehaviour(
            new Color(new ColorSpan(COLORS.spriteColors),
                new ColorSpan(COLORS.dampingColors),
                1 / (this.#nebulaParams.lifeScale),
                ease.easeOutSine)
        );

        // fade out particle to slightly less than its alpha at emission time
        particle.addBehaviour(
            new Alpha(1, 0.72)
        );

        // scale particle back down to slightly smaller than its size at emission time
        particle.addBehaviour(
            new Scale(1, 0.94)
        );
    }
}
```

Audio-Visual Analysis

We will now explore the visual differences of the animation as the parameters change. This is difficult to do with an interactive application that adjusts in real time, but for a report format, screenshots will have to suffice. Please see [Appendix, Video Demo] for a short demonstration with audio. (It's best to maximize screen since it's a desktop recording.)

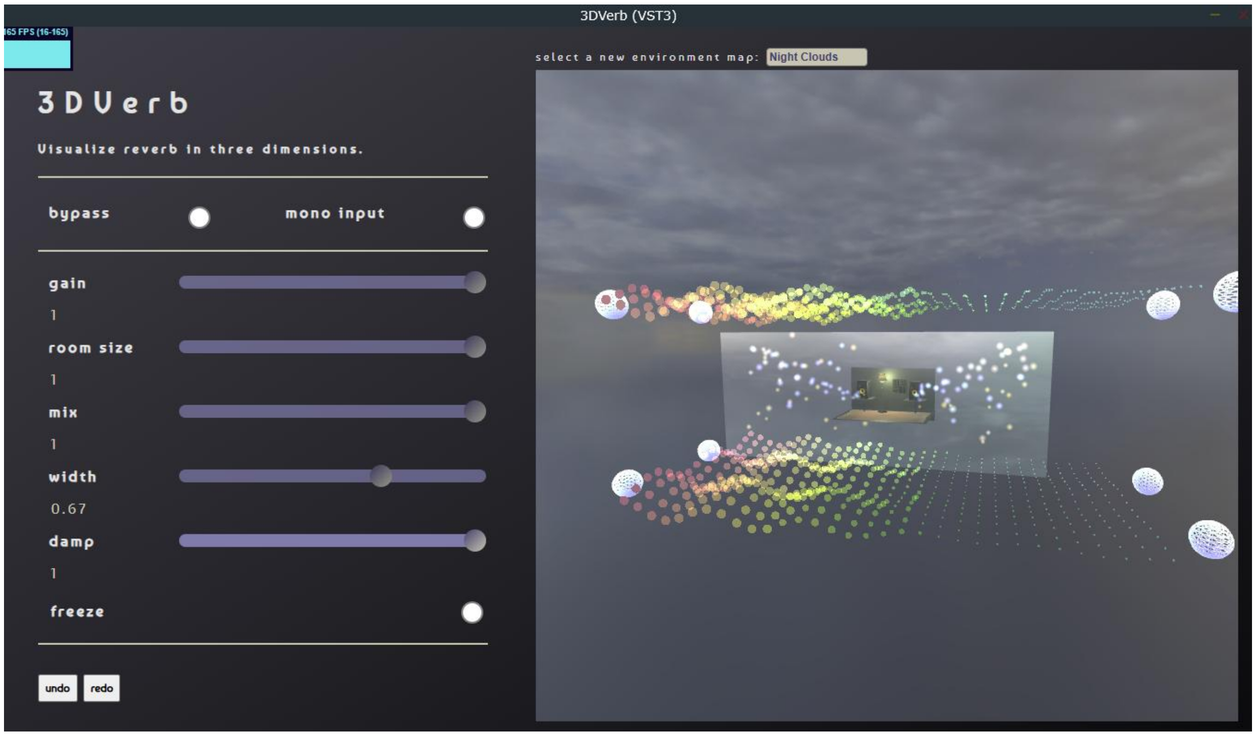
1. A low-mid string synthesizer single note is held. In first screenshot below there are more particles and they haven't changed color. In the second screenshot with equal room size but maxed out damp value, there are fewer particles, and some have changed color to indicate sound dampening:



select a new environment map: Night Clouds

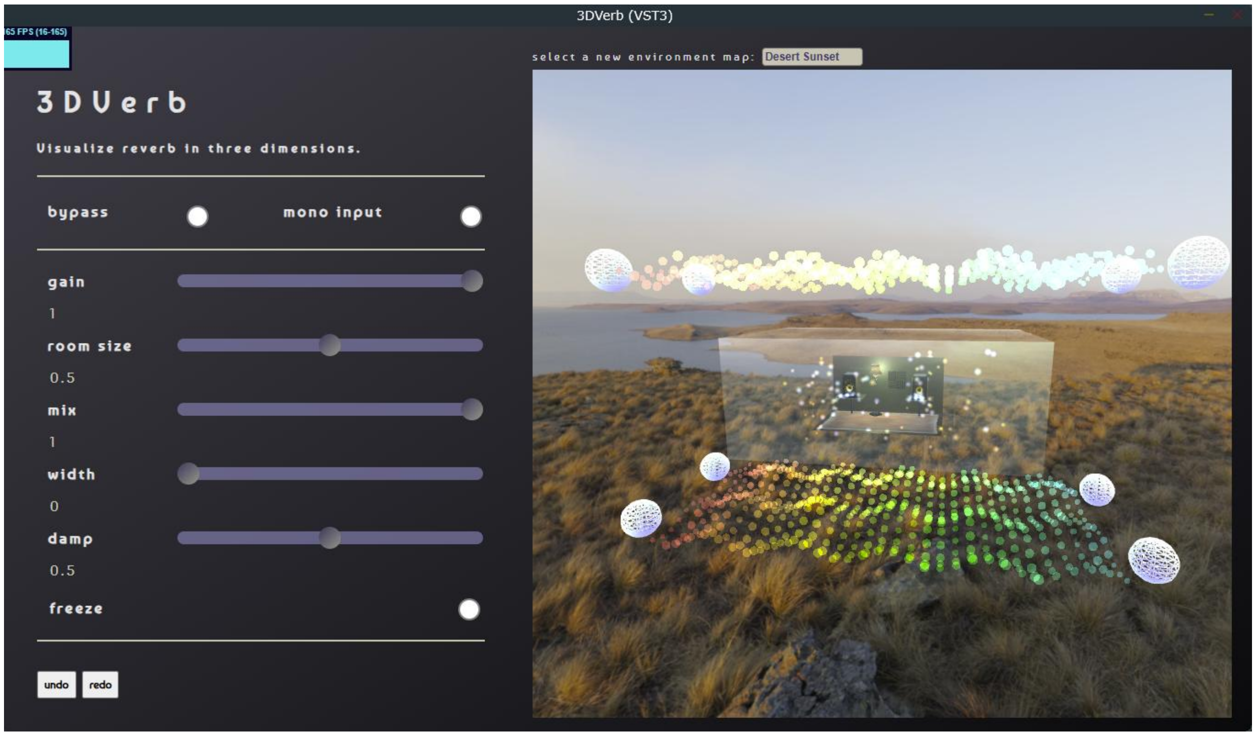
gain 1
room size 1
mix 1
width 0.67
damp 0

damp = 0; room size = 1;

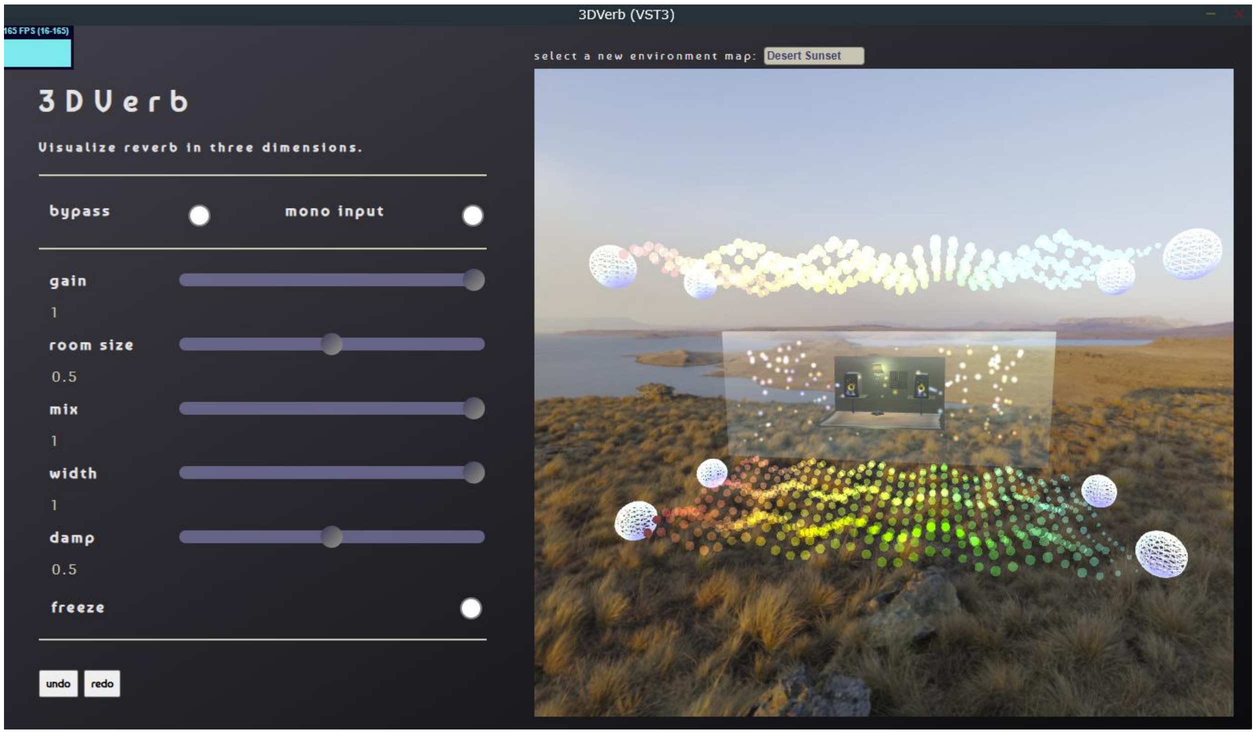


damp = 1; room size = 1;

2. A high mid string synthesizer note is played first with no width and then with width maxed out. The emission pattern converges more towards the center in the first.



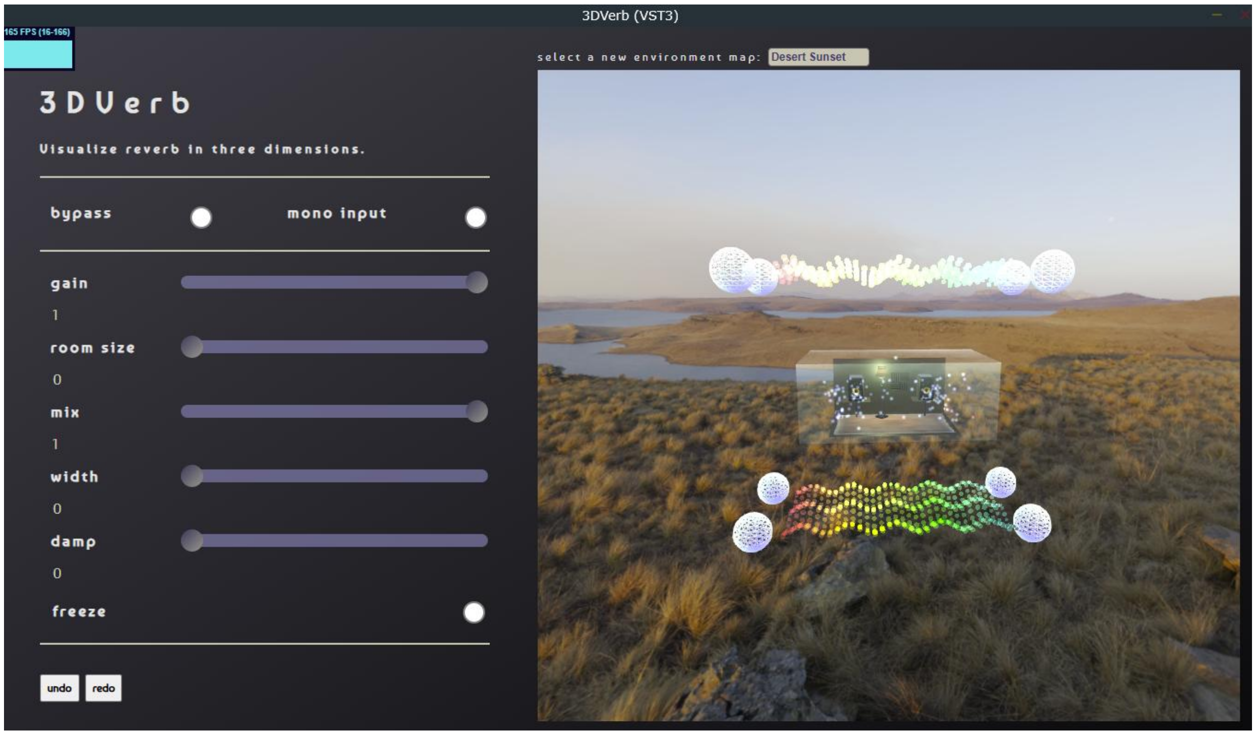
width: 0; damp 0.5


3DVerb (VST3)
165 FPS (16-165)
3DVerb
Visualize reverb in three dimensions.
select a new environment map: Desert Sunset
bypass
mono input
freeze
undo
redo

gain 1
room size 0.5
mix 1
width 1
damp 0.5

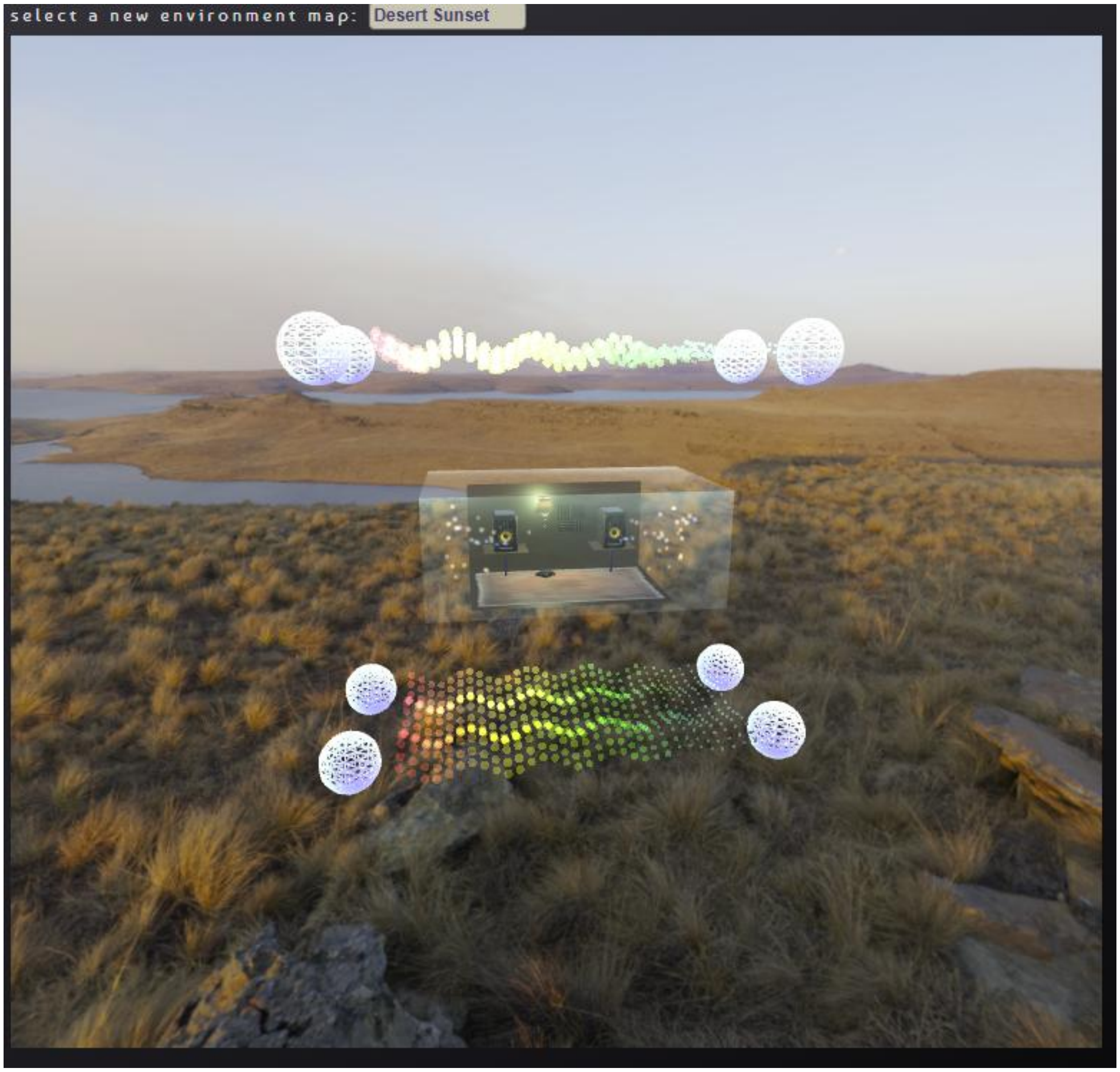
width: 1; damp: 0.5

3. Testing smallest room size. Note how the radius of the emitted sprite decreases with a smaller room size.

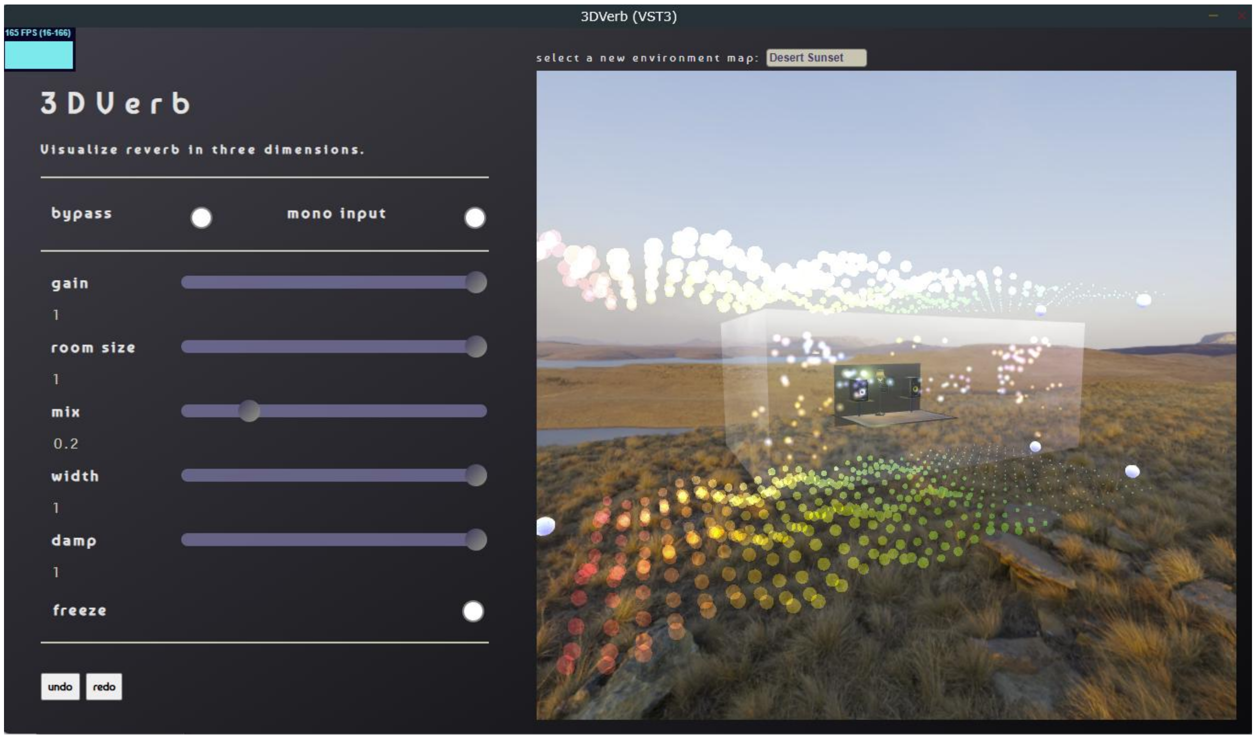

3DVerb (VST3)
165 FPS (16-166)
3DVerb
Visualize reverb in three dimensions.
select a new environment map: Desert Sunset
bypass
mono input
freeze
undo
redo

gain 1
room size 0
mix 1
width 0
damp 0

damp: 0; room size: 0



width: 1; damp: 1; room size: 0



3DVerb (VST3)

165 FPS (16-166)

3DVerb

Visualize reverb in three dimensions.

bypass

mono input

gain

1

room size

1

mix

0.2

width

1

damp

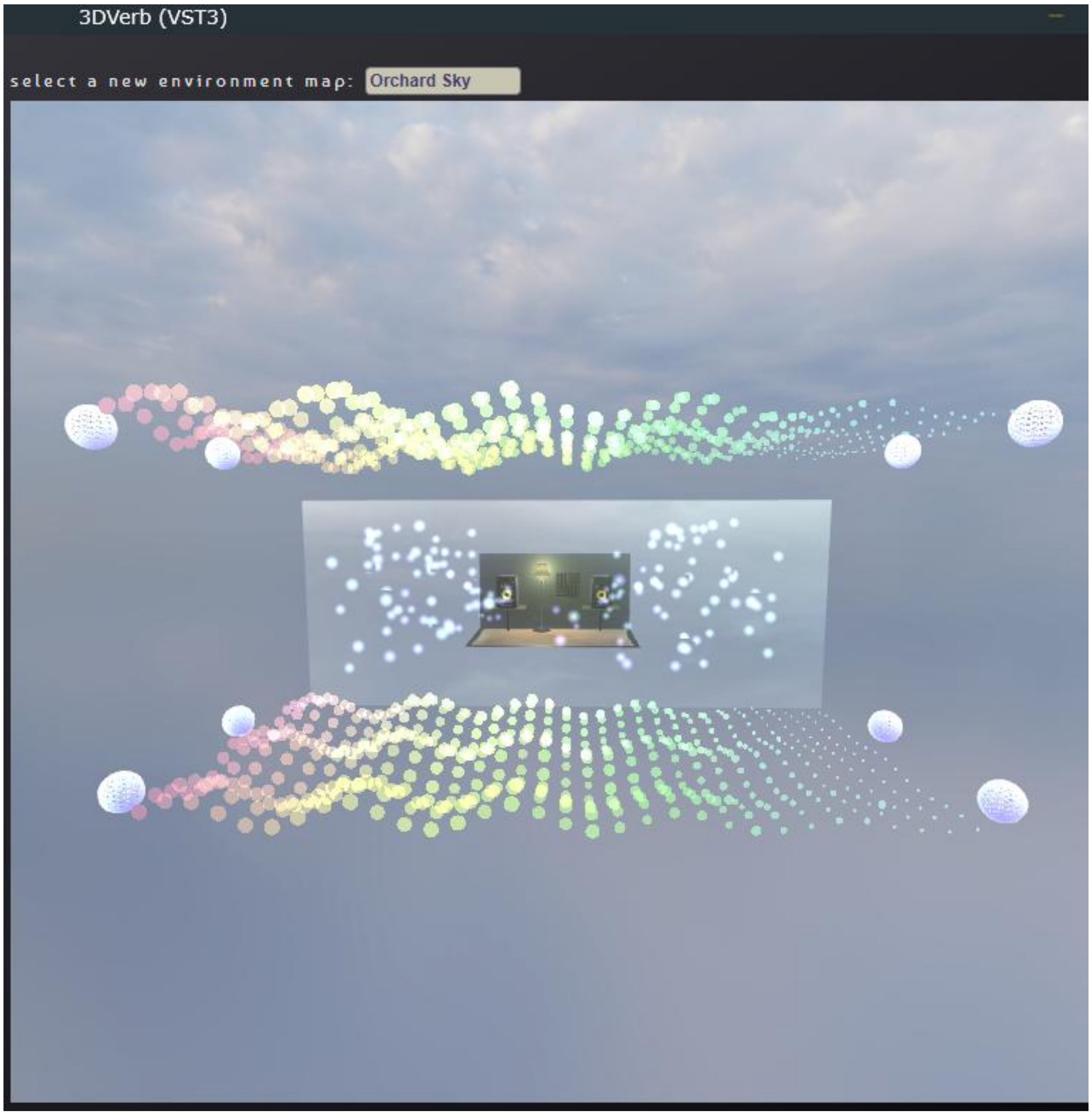
1

freeze

undo redo

select a new environment map: Desert Sunset

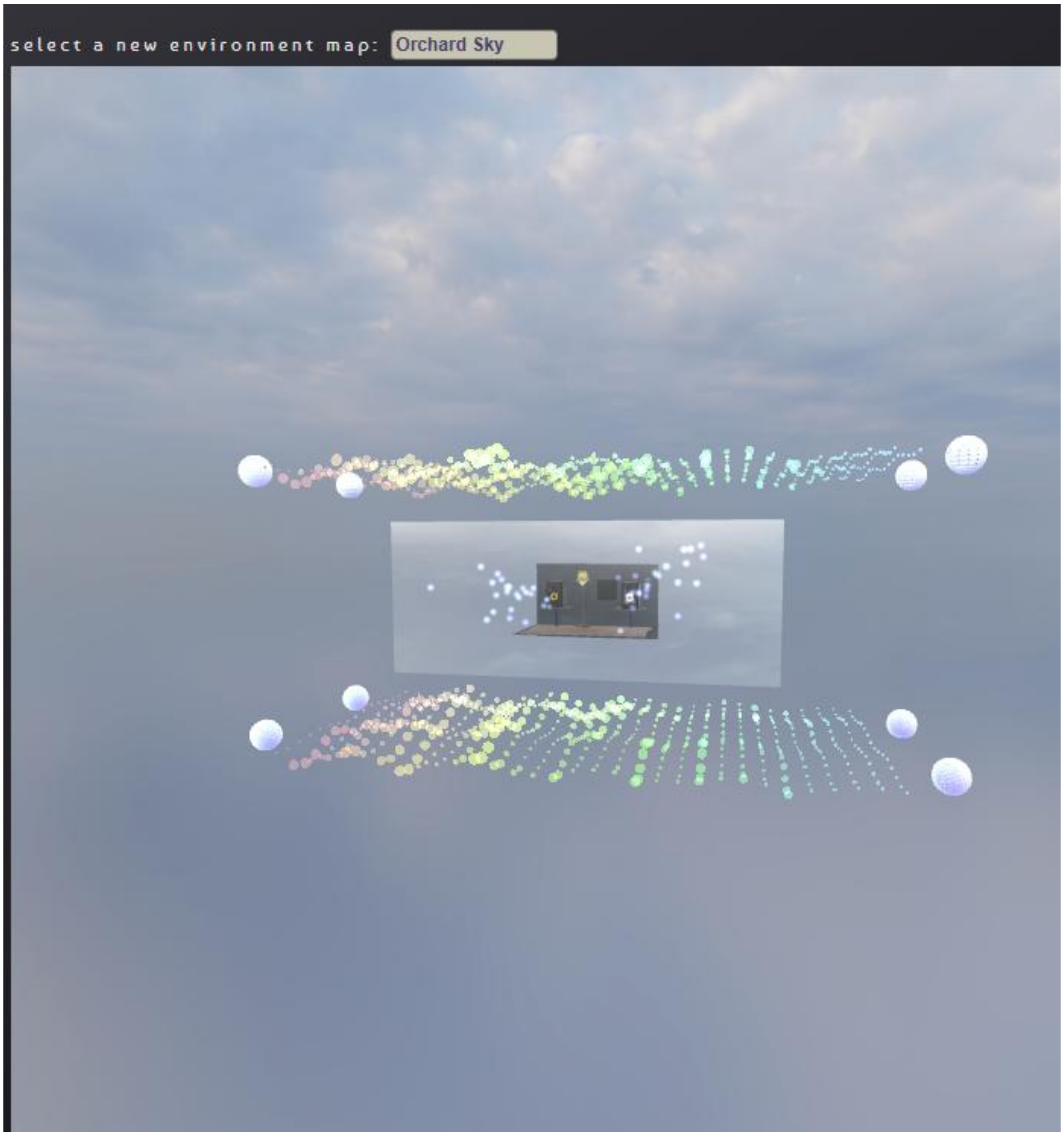
room size: 1; mix; 0.2; width: 1; damp: 1.



select a new environment map: Orchard Sky

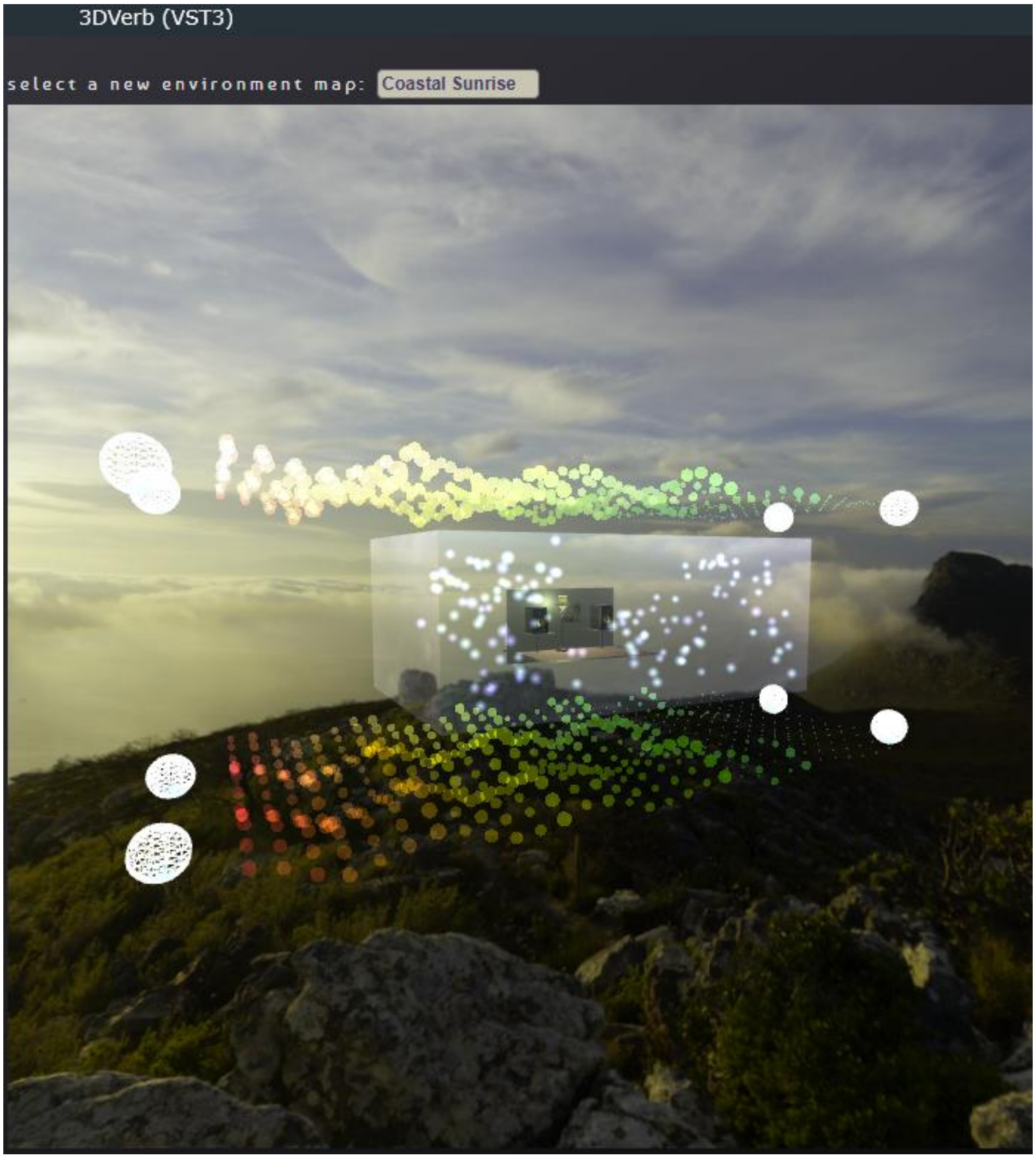
room size: 0.75; mix: 0.61; width: 0.25; damp: 0.4.

4. Playing sound quickly, letting it fade, and then enabling bypass (lamp point light disables; particle wave and emitters stop animating):



bypass: true; room size: 0.75; mix: 0.61; width: 0.25; damp: 0.4

5. This is after playing a high “C” note and some high frequencies have started to attenuate. Enabling freeze mode freezes the emitted particles in place but the particle wave keeps animating. The anchor spheres change color to a lighter shade of blue and the sound continues to sustain after enabling freeze:



freeze: true; room size: 1, mix: 1; width: 0.8; damp: 0.36

Applications and Extensions

Applications

Performance

With some refinement and the ability to upload a custom environment map (currently can only select from limited list) and custom 3D models (GLTF), a WebView plugin connected to a powerful WebGL animation library like ThreeJS could be utilized for live music or theatrical performances. The decoupling of the backend from the frontend is what makes this possible.

Education

3DVerb makes it possible to view reverb topology in real time, e.g., witnessing high frequencies attenuate faster when damp is set to a high value. With audio-visual feedback in real time as parameters change, students can perceive and understand how reverb works at an intuitive level.

Music Therapy

Many hobbyist musicians use their personal computer as their primary music device. Especially keyboardists and guitarists due to how easy it is to practice silently connected through MIDI or an audio interface. The PC is ubiquitous and utilizing it for music therapy is sensible. When developing this plug-in, I personally found it relaxing to view environment maps of natural settings, while seeing particles interact to what I was playing. I think with some improvements to the visualizer (less clutter and tweaks to transparency), 3DVerb has real the potential for actual music-therapy uses.

Mixing

Although extreme precision was not the goal of the visualizer, the visualizer does provide useful visual feedback on when high or low frequencies might be saturating the mix. The struggle was balancing how much low frequencies should dominate the visualization, while still animating the high frequency portion of *ParticleWave*.

Extensions

Feedback Delay Network (FDN) Algorithm

Explore a different technique for artificial reverberation. Fork 3DVerb and Implement an FDN algorithm. Use it instead of the reference algorithm. Tweak backend parameter mappings and test the extensibility of the frontend architecture when deploying new parameter mappings.

Complexity Controller

Implement a controller that monitors performance and scales animation features up or down depending on a user's computer hardware and observed performance. The goal is to maintain 60 fps on all modern PCs.

Preset Management

Preset management is well-established in JUCE, but it would be interesting to see how WebView affects this. Perhaps presets could be saved to JSON and then exported to a key-value database. This would likely require user authentication and authorization.

Custom environment maps and sprites

Allow users to upload their own custom environment maps and sprites for maximum customizability.

Simplified mobile version

Create an iOS version of the plug-in with just the *ParticleWave* to maintain high performance. Explore mobile plug-in development with JUCE.

Conclusion

3DVerb demonstrates a functional real-time 3D visualizer of a reference reverb algorithm. Parameters are mapped thoughtfully to various animation features, and a solid effort is made to balance aesthetics with perceptual accuracy while maintaining software development best practices. As a relatively new paradigm, JUCE WebView has potential as the future of plugin development since most development time for a plugin is spent on UI development.

As an MSCS candidate, there are many skills I honed while making this project over the course of the past few months. I learned better UI/UX, refined my JavaScript skills, learned new C++ features, and got a better grasp of DSP fundamentals and algorithmic reverb techniques. I was forced to recall several topics from my graduate studies including mutex locks, thread safety, design patterns, object-oriented programming, and the benefits of UML diagrams for the planning and organization of complex software. I will continue to explore audio DSP and its creative applications in the music world.

Appendix

Code Repository on GitHub

https://github.com/joe-mccann-dev/3DVerb/

Backend source code

https://github.com/joe-mccann-dev/3DVerb/tree/main/Source

Frontend source code

https://github.com/joe-mccann-dev/3DVerb/tree/main/Source/ui

Custom JavaScript Animation Code

https://github.com/joe-mccann-dev/3DVerb/tree/main/Source/ui/js

Video Demo

Link: 3DVerb Demo 1

References

1. Reverberation definition. Wikipedia: https://en.wikipedia.org/wiki/Reverberation
2. Reverb effect: https://en.wikipedia.org/wiki/Reverb_effect
3. T3X2R 3D Audio-Reactive Visualizer: https://www.t3x2r.com
4. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design patterns: elements of reusable Object-Oriented software. "Factory Method," page 107.
5. JUCE Framework: https://juce.com.
6. JUCE WebView Paradigm: https://juce.com/blog/juce-8-feature-overview-webview-uis/
7. Stack Overflow Developer Survey: https://survey.stackoverflow.co/2024/technology#most-popular-technologies-language-prof
8. ThreeJS: https://threejs.org/
9. Three Nebula: https://github.com/creativelifeform/three-nebula/tree/master
10. Code for animating emitter positions derived from: https://github.com/creativelifeform/three-nebula/blob/master/website/components/Examples/EightDiagrams/init.js
11. Code for particle wave derived from: https://github.com/mrdoob/three.js/blob/master/examples/webgl_points_waves.html

12. 3D Models utilized in animation
    (all created by Sketchfab.com users and licensed under Creative Commons CC BY 4.0):
    a. "Fine Persian Heriz Carpet" by mfb64. Source: https://skfb.ly/o76nI
    b. "Sound Proofing Dampening Foam" by MiningMark48. Source: https://skfb.ly/6tsrX
    c. "Floor Lamp" by bilgehan.korkmaz. Source: https://skfb.ly/OAQK
    d. "KRK Classic 5 Studio Monitor Speaker", by TheGiwi. Source: https://skfb.ly/6WMwT
13. Loading ThreeJS environment maps: https://threejs-journey.com/lessons/environment-map#model
14. Tool used to convert HDRIs to CubeMap textures: https://matheowis.github.io/HDRI-to-CubeMap/
15. Free-to-use environment maps obtained from:
    a. https://polyhaven.com/a/rogland_clear_night
    b. https://hdri-skies.com/free-hdris/
16. Analysis of Freeverb functionality: https://www.dsprelated.com/freebooks/pasp/Freeverb.html
17. More about FFTs: https://www.ap.com/news/more-about-ffts
18. Hanning window dsp: https://www.numberanalytics.com/blog/mastering-hanning-window-dsp
19. Fast Fourier Transformation FFT – Basics
20. Spectrum Analyzer: https://juce.com/tutorials/tutorial_spectrum_analyser/.
21. Details on comb filtering: https://unison.audio/comb-filter/
22. STFT: https://en.wikipedia.org/wiki/Short-time_Fourier_transform
23. Window functions: https://en.wikipedia.org/wiki/Window_function