MAP Final Project
Reverb and Distortion Guitar Effects on Bela

Yazhou Li

May 13, 2023

Abstract

In this project, I implement two reverberations, image source reverberation and the spring reverb, and a distortion guitar effect in real time. The levels of reverberation and distortion can be adjusted through either the Bela GUI or the breadboard. The system supports both playing a loaded audio file and playing guitar sounds in real time. The experiment shows real-time image source reverberation can only support reverberation orders lower than two, and the resulting reverberation is not that realistic. A demo video shows the performance of the system. The spring reverberation used in this project is limited to a single spring, so if we want to produce reverberation with a long reverberation time, it will not be dense enough and sound like delays. Future work includes using more springs with different lengths and adding the scattering effects in the spring reverb.

1 Image source reverberation

Image source reverberation is a geometrical method to estimate the impulse responses with the room's early reflections [1]. Each speaker is mapped to an image source symmetrical to each wall. The image sources and original speakers are together mapped against the wall again. The number of mapping times is called the reverberation order. The schematic of image sources is shown in Fig. 1.1.

The transfer function $T(\omega)$ from a source to a receiver at the frequency ω in the free field is

$$T(\omega) = \frac{1}{4\pi r} e^{-jkr}, \tag{1}$$

where $k = 2\pi\omega/c$ is the wave number, r is the distance from the speaker to the microphone.

The sound at one microphone can be seen as the sum of direct sounds in the free field from every image source to the microphone. The transfer functions at this microphone can be seen as the sum of transfer functions in the free field from every image source to the microphone.

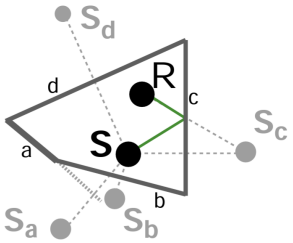


Figure 1.1: Image source method schematic, reproduced from [2]. R is the receiver, S is the source, a, b, c, d are four reflectors, and S_a, S_b, S_c, S_d are four image sources of S against four walls.

I refer to [3] for image source reverberation implementation. It is important to notice that due to the principle of image source reverberation, the current source can't be mirrored to the parent source again. The material absorption is added in the end, by multiplying the result of absorption by the power of the reverberation order times. The implementation is realized using a Tree data structure, each image source is stored as a tree node, and the reverberation order is the depth of the current tree node.

I first tried using a standard Tree class to implement the image source reverberation. The image sources of the current source are stored as children of the current node. If the child of the current node is the same as its parent, it will be ignored. After I get the impulse response from the speaker to the microphone, I can convolve the impulse response with the input audio. However, the queue push and pop operations in C++ may be too time-consuming and when the reverberation order is larger than 2, the system is unable to work in real time. Also, because the impulse response is consisted of discrete impulses, it may not be efficient to do the convolution.

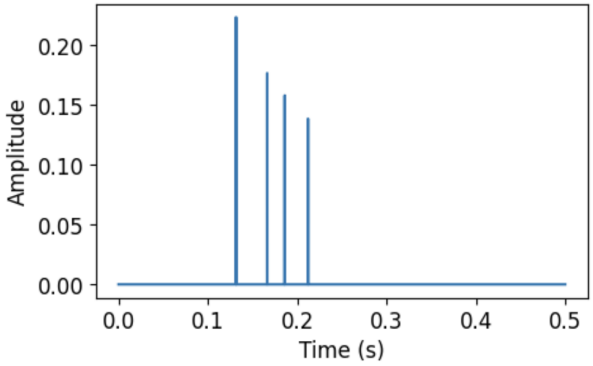


Figure 1.2: impulse response produced by image source reverberation

If we generate the impulse responses in advance and then convolve them with the input signal, this work will not be meaningful.

Therefore, I implement only the image source reverberation of the first order using delay buffers. The reverberation is not dense enough and sounds like delays. An example of the impulse response produced by the image source reverberation is shown in Figure

1.2. The room size is (50 m, 50 m), the microphone is placed at (5 m, 5 m) and the speaker is placed at (15 m, 25 m).

We can adjust the amount of reverberation in real time, which means the size of the room. The room is a shoebox and its side length can change from 5 m to 100 m. If we define the room size as s, the default position of the speaker is ($s/10$ m, $s10$ m) and the default position of the microphone is ($s/10 * 3$ m, $s/10 * 5$ m). We can also adjust the positions of the speaker and the microphone in real time easily, but we can't hear much difference, so this is not shown in the final user interface.

2 Spring Reverb

The spring's structure is shown in Figure 2.1. When the driver end receives the signal, the signal will pass along the spring to the pick-up end. Then the signal is reflected from the pick-up end and propagates to the driver end again, thus producing reverberation.

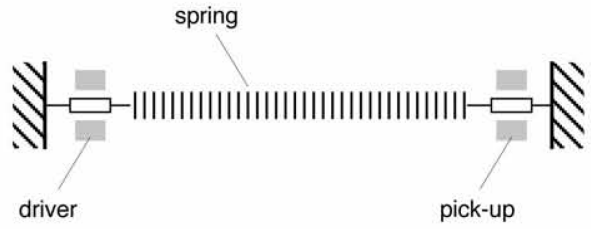


Figure 2.1: The structure of the spring in the spring reverb, reproduced from [4].

The system is shown in Figure 2.2. The waveguide section, the end reflection element (termination) and the scattering junction element are shown in Figure 2.3, Figure 2.4 separately.

Figure 2.2: The spring implemented using the waveguide model, reproduced from [4].

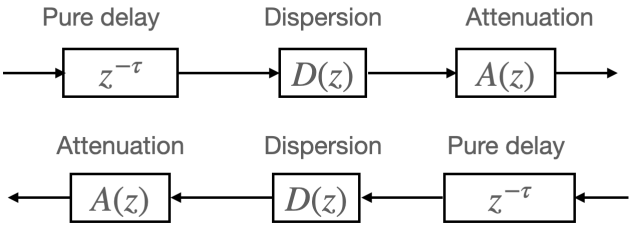


Figure 2.3: The waveguide section, reproduced from [4].

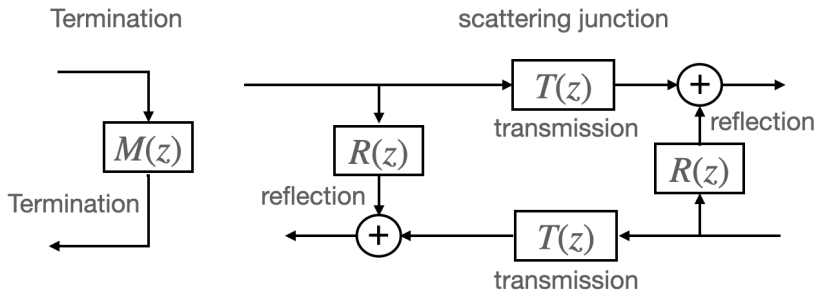


Figure 2.4: The scattering element and the end element, reproduced from [4].

The dispersion can be modeled using an all-pass filter [5], but because of the lack of the real spring's parameters, I did not implement this part. The scattering junctions are inserted into the waveguides to account for spring imperfections and to model the scattering between counter-wound springs [4].

To simplify the problem, there is just one waveguide section in the spring model, and no scattering section between the delay lines. The termination $M(z)$ is just 1 if we assume the end is stiff and lossless.

In my implementation of the waveguide, the attenuation is 0.5 and the delay is the traveling time from the driver to the pickup (which can change from 0 to 0.5 s). If the attenuation is near 1, there will be comb filtering effect and the signal will be distorted. We use two delay lines to model the reverberation in the guitar reverb pedal. Two circular buffers are used for the two delay lines. One buffer represents the forward-propagating delay line, and the other buffer represents the backward-propagating delay line. When we write the input to delay line 1, we write the delayed and attenuated input into delay line 2 (which is also the output of the spring reverb); when we write the input into delay line 2, we also write the delayed and attenuated input into delay line 1. In this way, the delays are fed back and forth.

3 Distortion

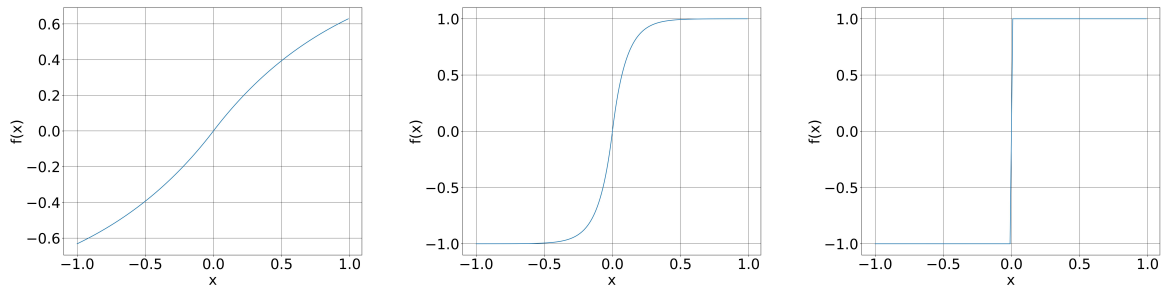
The distortion is implemented using a nonlinear function [6].

$$f(x) = sgn(x)(1 - e^{-|x|*m}), \tag{2}$$

where m is the amount of distortion, which ranges from 1 to 1000. The function's plots with different m are shown in Figure 3.1. If the signal is not distorted, the plot should be a linear line. Therefore, when m is increased, the signal is distorted more.

When $m = 0$, there will be no output signal, so we also combine the distorted signal with the original signal as the output. The final output o is

$$o = (1 - m) \times x + m \times f(x). \tag{3}$$



(a) m=1 (b) m=10 (c) m=1000

Figure 3.1: The nonlinear function for distortion when the distortion parameter is 1, 10 and 100.

4 Project Details

4.1 Input, Output

If reading an audio file, we use gPlayer.process() to get the input; If reading from the guitar output directly, we use audioRead(context, n, 0) to read from the analog input. The guitar is input to Bela through a mono jack. Although no amplifier is added before the guitar's output is fed into Bela, the sound is still large enough. Because the guitar's input is mono, we just read from the left channel (this has been tested by listening to the left channel and the right channel separately).

If we use the breadboard to control the system, we use two buttons to trigger the effects: one to turn on or turn off the distortion effect; the other to select the reverberation effects (no reverberation, spring reverb, or image source reverberation). Also, two potentiometers are added to change the amount of reverberation and distortion.

The user interface is shown in Figure 4.1. If we want to adjust the parameters from the Bela GUI, we read four parameters to represent the breadboard. The "reverb" and "distortion" are the two buttons that only include discrete values with a step of 1, the meanings of different values in these two parameters are shown in Table 1. "reverb amount" and "distortion amount" are two sliders with continuous values, representing the potentiometers on the breadboard. Although the slider's ranges are from 0 to 1, they are mapped to different values in the two reverberations, representing the spring length and the room size separately.

Table 1: Effect types with different values in the "reverb" and "distortion" sliders.

	0	1	2
distortion	off	on	-
reverb	off	spring reverb	image source reverb

However, when using the potentiometer on the breadboard to control the reverberation's amount, maybe because the potentiometer's output is not very stable, the

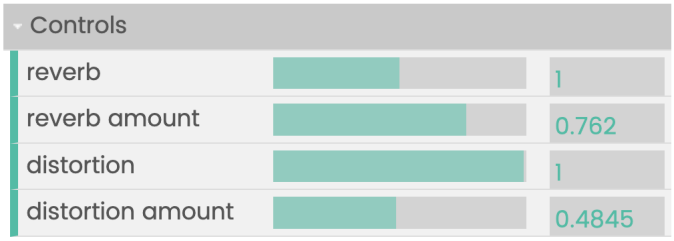


Figure 4.1: User Interface

reverberation sounds distorted (we can also observe this distortion when we are changing the slider "reverb amount" in Bela GUI). Therefore, for better performance, we just use Bela GUI to adjust the amount of reverberation at present.

The button's previous status is stored. If the button's previous status is HIGH and its current status is LOW, it means the button is just pressed, and we change the reverb and distortion types. The connections from the breadboard's electronic components to Bela PINs are not discussed in detail here.

4.2 C++ Classes

Each effect is written in a class that includes a process() function. We pass the input at each audio frame to the parameter of the process() function, and get the output of the effect from the process() function at each audio frame.

In the class initializing function, we define the default parameters and initialize delay buffers. When the reverberation button is pressed, we reset all elements in the delay buffers to zero, and set the read pointers and write pointers to zero.

The three objects, distortion, image source reverberation and spring reverb, are initialized at the beginning of the render file, and are initialized just once. *isr_simple* is the class for image source reverberation, *Reverberation* is the class for the spring reverb and *Distortion* is the class for distortion. Classes *Tree* and *ImageSource* are the original implementation of the image source reverberation, but they can't be run in real time successfully and are not used in the final implementation.

5 Conclusion

A spring reverb, an image source reverberation and a distortion are implemented in real time to form a guitar pedal chain. Three effects are written in three different classes, which can be adjusted and combined flexibly.

The image source reverberation can only be implemented in real time when the reverberation order is smaller than two (which includes just the direct sound and four image sources against four walls). The produced reverberation is not dense enough and sounds like delays. This suggests the image source reverberation may not be suitable for real-time reverberation.

The spring reverb can produce a small amount of reverberation, but if we increase the spring's length in order to produce a longer reverberation time, the result is also not dense enough and sounds like delays. Further work includes implementing the dispersion and scattering effects in the spring reverb. We can also use more parallel springs with different lengths to produce denser late reverberation.

References

[1] Jont B Allen and David A Berkley. Image method for efficiently simulating small-room acoustics. *The Journal of the Acoustical Society of America*, 65(4):943–950, 1979.

[2] Thomas Funkhouser, Ingrid Carlbom, et al. A beam tracing approach to acoustic modeling for interactive virtual environments. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 21–32, 1998.

[3] Lauri Savioja and U Peter Svensson. Overview of geometrical room acoustic modeling techniques. *The Journal of the Acoustical Society of America*, 138(2):708–730, 2015.

[4] Jonathan S Abel, David P Berners, Sean Costello, and Julius O Smith III. Spring reverb emulation using dispersive allpass filters in a waveguide structure. In *Audio Engineering Society Convention 121*. Audio Engineering Society, 2006.

[5] Jonathan S Abel and Julius O Smith. Robust design of very high-order allpass dispersion filters. In *Proc. of the Int. Conf. on Digital Audio Effects (DAFx-06), Montreal, Quebec, Canada*, pages 13–18. Citeseer, 2006.

[6] Udo Zölzer, Xavier Amatriain, Daniel Arfib, Jordi Bonada, Giovanni De Poli, Pierre Dutilleux, Gianpaolo Evangelista, Florian Keiler, Alex Loscos, Davide Rocchesso, et al. *DAFX-Digital audio effects*. John Wiley & Sons, 2002.