Assignment 1: Scattering Delay Network Reverb

Chris Yeoward

Introduction

The task for this assignment was to implement an original audio effect, either using the Juce framework or otherwise. For this I chose to implement a state-of-the-art reverberation algorithm known as a Scattering Delay Network (SDN), presented in [1]. Since Jot introduced the Feedback Delay Nework (FDN) in the early 90s, it has become an industry standard for the implementation of artificial reverbs [2]. It offered independent control over the parameters of the reverb, such as energy storage, damping and diffusion component. However, it remained difficult to bridge the gap between the parameters and perceptually significant features such as room layout and surface composition, requiring considerable effort to tune the network parameters [3]. The algorithm proposed in [1] addresses these issues, providing a network with independently tuneable characteristics that are directly informed by room properties. Additionally, SDNs promise to be an order of magnitude lower in computational complexity than FDNs [1]. As someone particularly interested in realistic reverb, I wished to have an effect that would efficiently model a room with a moveable source. This particular effect could then be used to model virtual spaces, and could see application in both virtual and augmented reality.

Scattering Delay Network

In essence, the scattering delay network is combination of a digital waveguide network (DWN) and the ray-tracing image source model (IM) [3], shown in figure 1. It consists of a DWN with scattering junction nodes placed at positions of 1st order reflections, thereby accurately modelling the early reflections and decreasing in accuracy for higher orders (fig 2).

Delay lines connect the source to each node and microphone, each node to the microphone, and bidirectional delay lines between the nodes. The length of each delay line is determined by the real physical distance between two points. For example the delay between the locations of the source $\mathbf{x}_S$ and microphone $\mathbf{x}_M$, $D_{S,M}$, is given by:

$$D_{S,M} = \lfloor \frac{F_s \|\mathbf{x_S} - \mathbf{x_M}\|}{c} \rfloor$$

Where c is the speed of sound in air, and F_s is the sampling frequency.

Source

In this implementation, the source is assumed to radiate sound in all directions equally, though a directivity vector could also easily be applied [1]. As such, the inputs from the source to each node are all applied equally, alongside an attenuation factor $g_{S,k}$ that models the spherical spreading of acoustic energy through space (e.g. $1/r$):

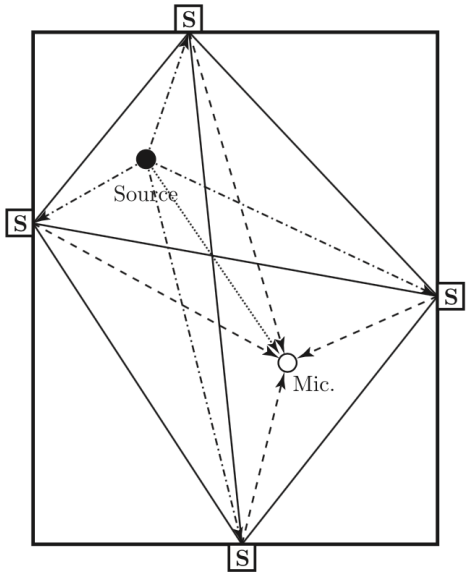

Fig. 1: Conceptual model of a scattering delay network. Scattering junctions (S) are at positions of 1st order reflections. Delay lines connect the source to the microphone and each node, the nodes to each other node, and each node to the microphone. Delay line lengths are informed by the time taken for sound to propagate the space between each point. [1]

$$g_{S,k} = \frac{1}{\|\mathbf{x_S} - \mathbf{x_k}\|}$$

where $\mathbf{x_k}$ is the position vector of node k. The attenuation factor $g_{S,M}$ is also applied to the source-microphone delay line.

Nodes

Each node receives an input from each of the other nodes, with the addition of the contribution from the source. This input vector is then multiplied by a scattering matrix $\mathbf{A}$, with the resultant vector defining the outputs to each of the other nodes in the network. Different scattering matrices can be used, and they could be different for each node. However, in this instance, each are chosen to be the same, *isotropic scattering matrix*:

$$A = \frac{2}{K}\mathbf{1}\mathbf{1}^{\mathbf{T}} - \mathbf{I}$$

where K is the network order (5 for a cuboidal room), $\mathbf{1} = [1_1, ..., 1_k]^T$ and $\mathbf{I}$ the identity matrix. This matrix uniformly distributes an sample incoming from one node to each of the others, reflecting a small portion back to the originating node. The matrix is lossless, such that without

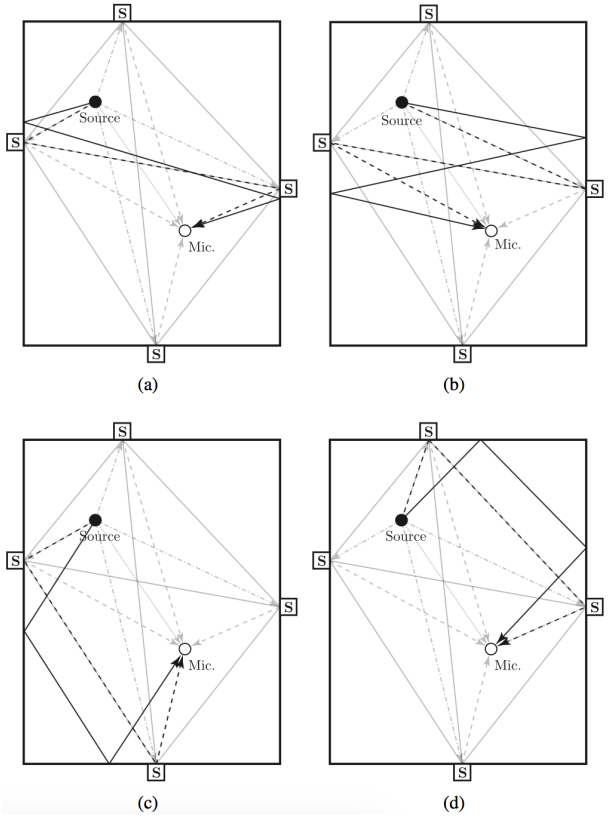

Fig. 2: Paths taken by the second order reflections. The solid lines show the real physical path, and the dotted lines show the paths taken through the SDN. [1]

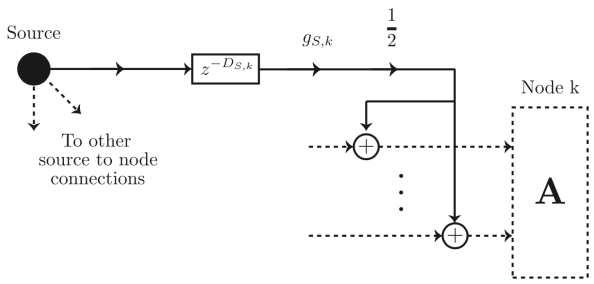

Fig. 3: Block diagram of the source to node connections. $D_{S,k}$ is the delay between the source and the node, in samples. [1]

attenuation the samples would propagate through the network forever. As such, an absorption coefficient β is applied to the outgoing samples, which controls the rate of decay of the diffuse sound field. Frequency dependent filters $H_k(z)$ can also be applied instead to colour the sound and give more realistic reflection characteristics.

The locations of the nodes are determined by simple geometric calculations, relative to the distance between the source and microphone along a given axis, and their perpendicular distance from a given wall. For example, the displacement of the node along a wall in the ZY plane from the position of the mic is given by:

$$y_k = \frac{x_M(y_S - y_M)}{x_M + x_S}, z_k = \frac{x_M(z_S - z_M)}{x_M + x_S}$$

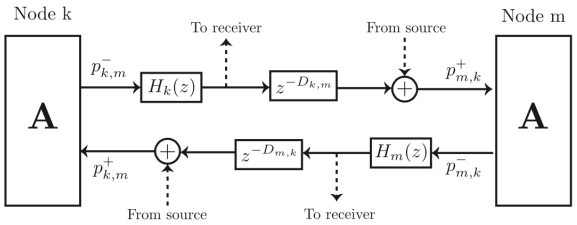

Fig. 4: Block diagram of the connection between two nodes connections. The input from the source is inserted at each junction before the scattering matrix is applied. [1]

where x_M and x_S are the distances of the microphone and source to the wall, and y_k is the displacement of the node along the y axis from the microphone's position y_M, similarly applying to z_k. As the wall is at a fixed position on the x axis, the x coordinate for the node is the same as the wall's.

Microphone

In a similar manner to the source, a directivity vector could be applied to the microphone input to scale the contribution from input. In this implementation however, I assume that the listener is facing forward, oriented parallel to the y axis. The source and node signals are then combined at the output and scaled for each stereo channel by gains g_l and g_r. These are determined by the position's azimuth relative to the y-axis[4]:

$$g_l(\theta) = \frac{1 - sin(\theta)}{\sqrt{2(1 + sin^2(\theta))}}$$

$$g_r(\theta) = \frac{1 + sin(\theta)}{\sqrt{2(1 + sin^2(\theta))}}$$

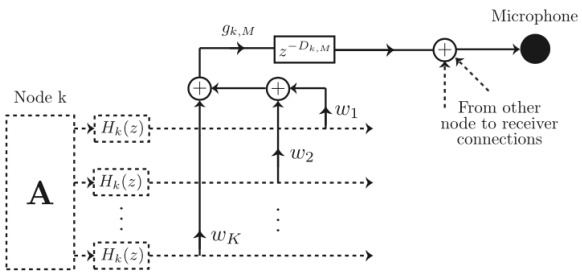

Fig. 5: Block diagram of the connection between a node and the microphone. Each node can have a weighting w_k applied to it, though in this implementation these are all set to 1. [1]

IMPLEMENTATION

This effect was implemented in C++ using the Juce framework. The UI is shown in 6

The DWN model lends itself well to object oriented programming, as each component represents a high-level, intuitive concept. The application runs from the $Network.h$ class, and is split into the following composite classes:

- **Node.h**: Scattering node

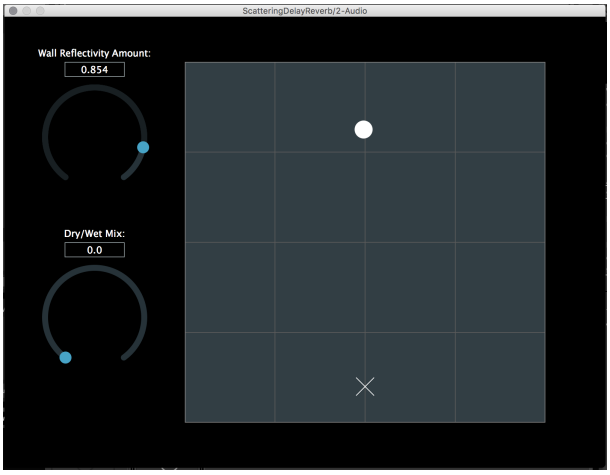

Fig. 6: The user interface. The solid circle represents the source position in the room, and the cross the microphone. The reflectivity controls the length of the reverb.

- **Connection.h**: Bidirectional delay line between two nodes
- **Terminal.h**: The end of a connection, with pointers to the delays for a node to read from and write to.
- **Boundary.h**: Bounds of the room. Has position and orientation, and determines where to place a node for the first order reflection.
- **ModulatingDelay.h**: Delay line with the length of the delay slightly modulated, so as to avoid the metallic sound and flutter characteristic of many artificial reverbs [2]

which are built from the following base classes:

- **Point.h**: 3D cartesian coordinate.
- **Delay.h**: Contains a buffer, read and write pointers, and the ability to change the delay length.

Network

The delay network first initialises the nodes by determining their positions and creating connections between them based on their distances. The number of connections N_c between nodes is determined programatically, and is given by:

$$N_c = \sum_{n=1}^{K} n$$

where K is the order of the network, related to the number of nodes N_n as $K = N_n - 1$. For a cuboidal room, the number of connections is therefor 15, comprising of 30 delay lines, in addition to the unidirectional delay lines from the source and to the microphone. Each connection is created with a length defined by the distance between each node. Upon creation of each connection, the terminal at one end is given to the first node, and the other to the second:

```
int connection = 0;
for(int node = 0; node < nodeCount - 1; node++)
{
  for(int otherNode = node + 1; otherNode < nodeCount;
      otherNode++)
  {
    connections[connection] = SDN::Connection(nodes[node],
        nodes[otherNode], sampleRate);
    nodes[node].addTerminal(
        connections[connection].getStartTerminal());
    nodes[otherNode].addTerminal(
        connections[connection].getEndTerminal());
    connection++;
  }
}
```

The main processing is done in a *scatter* method. This function first writes the input to the source-mic delay line, and to each of the source-node delays. For each node, the input wave vector is then gathered from each of the node connections, scattered, and redistributed to the network. So that each scatter is performed at the same time step, the wave vectors are first all gathered, and then all processed.

```
void Network::scatter(float in)
{
  in *= 0.1; // accommodate 10x gain factor in 1/r volume
      adjustment
  in *= 0.5;

  sourceMicDelay->write(in);

  for(int node = 0; node < nodeCount; node++)
  {
    sourceToNodeDelays[node].write(in);
    nodes[node].gatherInputWaveVectorFromNodes();
  }

  for(int node = 0; node < nodeCount; node++)
  {
    float nodeOut = nodes[node].getNodeOutput();
    nodes[node].scatter(sourceToNodeDelays[node].read() /
        (fmax(mic.distanceTo(nodes[node].getPosition()),
        0.1))); // add distance attenuation
    nodes[node].distributeOutputWaveVectorToNodes();
    nodeToMicDelays[node].write(nodeOut);
  }
}
```

To avoid an infinitely gained output at distances close to 0, the distance attenuation is clamped to 0.1. As such, input is scaled by a factor of 0.1, to accommodate the maximum potential gain factor of 10.

Node

The *Node* class encapsulates the behaviour for each scattering junction. It contains an array of terminals, that represent interfaces with *Connections*, and are where the nodes read from and write to. It gathers the inputs from each of the terminals, scatters by multiplying the input wave vector by the scattering matrix, and then distributes the wave vector back to the terminals.

```
void Node::gatherInputWaveVectorFromNodes() {
  for(int terminal = 0; terminal < terminalCount; terminal
      ++) {
    waveVector[terminal] = 0;
    waveVector[terminal] = terminals[terminal]->read();
  }
}
```

```
void Node::distributeOutputWaveVectorToNodes() {
  for(int terminal = 0; terminal < terminalCount;
       terminal++) {
    terminals[terminal]->write(waveVector[terminal]);
  }
}
void Node::scatter(float sourceInput) {

  float absorptionFactor = 0.7;

  float networkInput = sourceInput / numberOfOtherNodes;

  float tempVector[numberOfOtherNodes];

  for(int i = 0; i < numberOfOtherNodes; i ++) {
    tempVector[i] = 0.0;
    for(int j = 0; j < numberOfOtherNodes; j++) {
      tempVector[i] +=
           scatteringMatrix[(numberOfOtherNodes * i) + j] *
           (networkInput + waveVector[j]);
    }
    tempVector[i] *= absorptionFactor;
  }

  for(int i = 0; i < numberOfOtherNodes; i ++) {
    waveVector[i] = tempVector[i];
  }
}
```

The scattering matrix is calculated programatically during the construction of each node, dependent on K.

Modulating Delay

The delay lines between the source and the nodes and between the nodes and the mic are subjected to a small modulation amount. This prevents an undesirably metallic sound being present in the reverb tail. To achieve this effectively, the read pointer is set as a floating point number, and linear interpolation is then applied between the samples.

```
float Delay::read() {
  int highPointer = floor(readPointer + 1.5);
  if(highPointer >= bufferLength) highPointer -=
       bufferLength;

  float high = buffer[highPointer];
  float low = buffer[(int) floor(readPointer)];
  float out = (1 - (readPointer - floor(readPointer))) *
       low + (readPointer - floor(readPointer)) * high; //
       interpolate

  incrementReadPointer();

  if(bufferLength < 0)
    readPointer += bufferLength;
  else if (readPointer >= bufferLength)
    readPointer -= bufferLength;

  return out;
}
```

This also means that the changes in delay length happen smoothly, avoiding artefacts from rounding a floating point change to an integer.

Source positioning

As the algorithm permits real time modification of high level parameters, it was simple to add in the controls for moving the source. The most intuitive way to do this was to create an XY pad that would control the movement of the source along the x and y axes. To achieve this, a custom component was created called `XYContainer` that handles mouse drag events, adjusting the x and y coordinates accordingly.

```
void XYContainer::mouseDrag(const MouseEvent& event)
{

    float x = (event.getMouseDownX() +
         event.getDistanceFromDragStartX()) / (float)
         getWidth();
    float y = 1 - ((event.getMouseDownY() +
         event.getDistanceFromDragStartY()) / (float)
         getHeight());
    processor.updateSourcePosition(x, y, 1.5); // hard
         code z position to be 1.5 metres
}
```

The coordinates are sent as a proportion of the total dimensions, with $updateSourcePosition(...)$ calling $setValueNotifyingHost()$ to update the parameters accordingly.

Due to the limitations of mouse control, there is not an elegant way to handle changes along the z-axis, though a multi-touch device would improve this experience.

Evaluation

An impulse response $impulse.wav$ was first generated in Matlab, and then used in Ableton Live to measure the effect. The audio samples were also recorded in Ableton. The submission includes all raw and processed samples. A room size of 10m x 10m x 3m was chosen for the tests.

Reverberation

Figures 7 to 10 show the stereo impulse responses for several wall reflectivity coefficient β values. The reverberation times for coefficients of 0.5, 0.7, 0.8 and 0.9 are 195ms, 357ms, 538ms and 1026ms respectively.

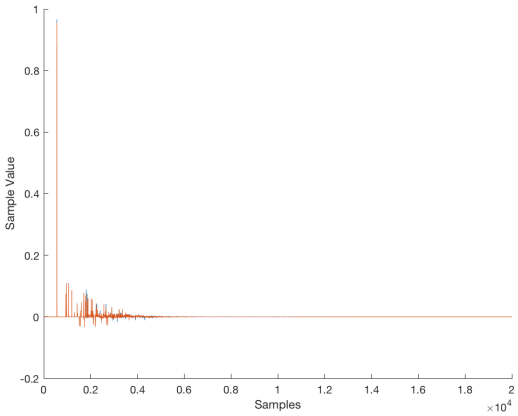

Fig. 7: Impulse response for β = 0.5. Left channel is in blue, right in red.

The reverb tail clearly increases with increasing β, with a different diffusion appearing between the two channels.

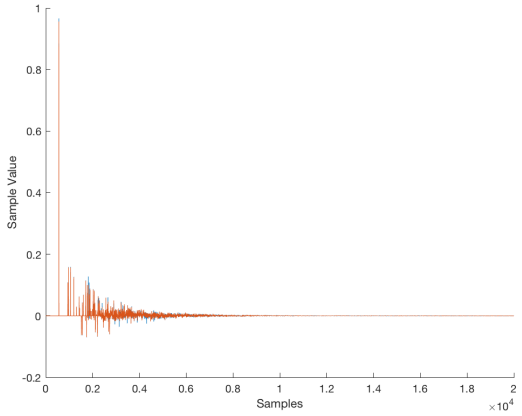

Fig. 8: Impulse response for β = 0.7. Left channel is in blue, right in red.

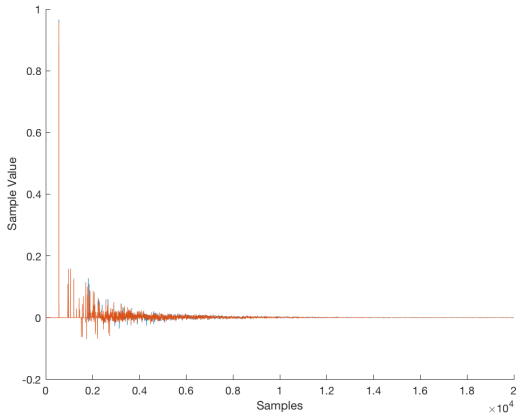

Fig. 9: Impulse response for β = 0.8. Left channel is in blue, right in red.

Limitations

The VST occasionally crashes while moving the source around, or throws out noise. This is likely caused by a delay line pointer pointing to some unallocated piece of memory, but the true cause and solution are unknown. This happens much more frequently when the room size is set to be orders of magnitude greater than the default 10m x 10m x 3m, resulting in large buffer sizes. I had originally wanted to be able to vary the room size, but decided to not to on account of this issue.

As the source moves close to a boundary, the gain of the reverberation increases. This is due to the $1/r$ factor being applied to the delay line, and increasing for values greater than 1. A solution could be limit the distance to 1, but I chose to leave it as I liked the generated sound.

References

[1] De Sena, E et al. (2015) *Efficient Synthesis of Room Acoustics via Scattering Delay Networks* IEEE
[2] Valimaki, V. et al (2012) *Fifty Years of Artificial Reverb* IEEE
[3] Valimaki, V. et al (2016) *More than Fifty Years of Artificial Reverb* AES
[4] Pulkki, V. (1997) *Virtual Sound Source Positioning Using Vector Base Amplitude Panning* AES

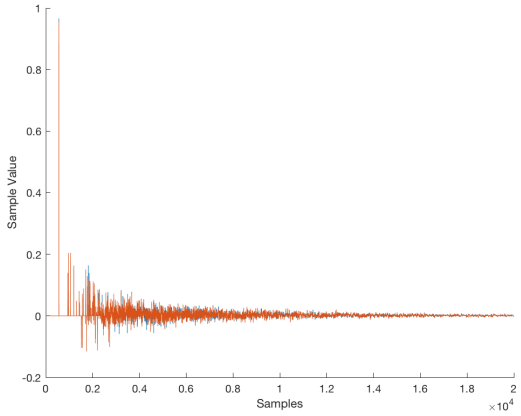

Fig. 10: Impulse response for β = 0.9. Left channel is in blue, right in red.

[5] Reiss, J. (2019) *Digital Audio Effects*
[6] Reiss, J. and Macpherson, A. (2014) *Audio Effects: Theory, Implementation and Application* CRC Press
[7] Plug In Doctor *https://ddmf.eu/plugindoctor/*