ES3 Lab 7
Real-time physically modelled sound synthesis
This Lab

• You will build a fairly realistic physically-modelled guitar

• User interface will be provided, along with a simplified audio driver

• You will have to implement all the synthesis code!
Outline of steps

• Get the template, and check that it works
• Synthesise a simple sine wave
• Construct a digital delay line class
• Fill it with noise and make it recirculate to generate a simple "pluck"
• Create a guitar string class
• Link it to the UI
• Set the string tuning from the UI
• Make six strings
• Implement a realistic modelled pluck
• Add a pick position model
Structure of provided code

- lab7.zip has the template code
  - **SoundHandler**
    - provides the basic sound driver and wave loading functions
  - **DigitalGuitarAppDelegate**
    - the app delegate, just creates a **GuitarViewController** and shows it
  - **GuitarSynthesizer**
    - the skeleton synthesizer, opens up the audio driver
    - you have to fill in the **fillBuffer** method to make sounds!
  - **UIFontBoard**
    - provides a fretboard display (allows you to tap on strings)
    - sends messages to the **GuitarSynthesizer** when frets change
  - **UIGuitarView**
    - provides a control which can be strummed
    - sends messages to **GuitarSynthesizer** saying which string has been plucked
  - **GuitarViewController**
    - just instantiates the **UIGuitarView** and **UIFontBoard** and shows them
    - links the **GuitarSynthesizer** instance to **UIGuitarView** and **UIFontBoard**
Your task

• You will only need to modify **GuitarSynthesizer**
  ▫ All the rest of the provided classes should remain unchanged!
  ▫ You will have to create new classes though, to represent the string models

• You just have to create a simple waveguide string model, which can be used in **GuitarSynthesizer** to produce sounds when it receives pluck events from the **UIGuitarView**
Getting Started

• Build the project and run it

• Note that the UI is like a guitar folded in half
  ▫ the fretboard runs horizontally instead of vertically
  ▫ as does the strumming area

• You should be able to click on notes in the fretboard (top half) and circles will indicate where fingers are "down"
Creating a sound

- To test the sound is working, add some simple code to generate a tone in the `fillBuffer` method of `GuitarSynthesizer`
  - At the moment, the buffer is filled with zeros (which is obviously silent...)

- After the "INSERT SYNTHESIS CODE HERE" comment, replace the v=0.0 with
  \[
  v = \sin(i \times 440 \times 2 \pi / (44100.0));
  \]
  - Change the 440 to something else for a different frequency
  - the divide by 44100.0 is because we are using a 44100.0 sample rate
  - the scale by 2*Pi is because a sine wave takes an increment of 2pi to make a full cycle
    - so this function does a full cycle 440 times a second

- Note that the code immediately below automatically rescales v to -32768 -- 32767
  - we are using a 16 bit, 44100Hz PCM format
  - we will work with floating point numbers in the range -1.0 -- 1.0 and rescale at the end
  - all your computation should work with `double` values in -1.0 -- 1.0
First sounds!

• Test it!
  ▫ You should hear a clear tone

• Note: the tone will have continuous, annoying clicks

• Think about why this is happening
  ▫ Hint: what happens when one buffer finishes and another one starts?

• Fix it, by introducing a new member variable...
On to strings

- OK, sound is working
  - you can safely comment out the sine generation now

- A simple physical model uses a single delay and some attenuation and filtering

- We need to create a delay line
  - this is an object which takes a value and returns the value passed \( n \) steps ago
  - where \( n \) is the length of the delay
Creating a delay line

• A delay line is very simple
  ▫ It can be modelled as just an array of previous samples
  ▫ Each step, we put a new sample on the start of the line
    • and read out the sample n steps ago

• Create a new class **DelayLine** to represent the delay line
  ▫ It will need a **double** * member variable to store the delayed samples
    • Note: we use C arrays for efficiency here, not NSArrays!
  ▫ And a variable representing the length of the delay line

• Add methods to initialise the delay line (**initDelay**), get the current delayed output (**getOutput**), put a new value onto the delay line (**newSample**), and set the delay length (**setDelay**)
  ▫ **newSample** should take a double, **getOutput** should return one

![Diagram of Delay Line and Damping](image-url)
Allocating delay memory

- The delay array is a C array
  - Allocate it in `initDelay` with `calloc` (like `malloc`, but zeros the array)
    
    ```
    samples = calloc(sizeof(*samples), maxDelay);
    ```
  - Remember to `free` it in `dealloc`!
  - **Note:** you should allocate an array of a fixed length (`maxDelay`) which should be say 2048 samples (this is much longer than we'll ever use)
  - We will use some subset of this when the delay is shorter, but we don't want to have to keep reallocating arrays...
  - You will need an instance variable to represent the currently used delay length, which can be set by `setDelay`

- Every time we get a sample, we could shift the whole array down, then put the new sample on the end...
  - This is terribly inefficient!

- Instead, we use a pair of indices
  - A *read head* and a *write head*
  - *The write head follows behind the read head*
  - *both heads wrap around when they reach the end*
Delay Line

- Create instance variables for the `readHead` and `writeHead`
  - just `ints`: they represent indices in the `samples` array

- Initialise the read head to 1 and the write head to 0
- every time `newSample` is called
  - write the new value into `samples[writeHead]`
  - increment `readHead` and `writeHead`
  - check if `readHead` or `writeHead` is equal to the delay line length
    - if so, reset it to zero (so it wraps around)

- To get the current output, just return `samples[readHead]`

- One subtle issue: if you change the delay line length and make it smaller, both `readHead` and `writeHead` might be greater the new delay length, and both get reset to 0
  - This would be very bad!
  - Always check if `readHead` == `writeHead`, and if so, reset them to 1 and 0, respectively
Making a noise!

• Add a method `fillWithNoise` to `DelayLine`
  - fill each element of samples with a number between -1.0 and 1.0
  - this can be done with
    - \( r = \text{arc4random}() / (\text{double})0xffffffff \ast 2.0 - 1.0 \)

• Now, in `GuitarSynth`, add an instance of `DelayLine`
  - Initialise it, and set it to 140 samples long
  - Call `fillWithNoise` on it immediately
  - in the `fillBuffer` routine, we need to:
    - read out the current value of the delay line, and feed back that value scaled by some value <1.0
    - this represents the damping of the string -- closer to 1 is more "resonance"
    - try 0.99
    - the new sample output (v) is the value we read from the delay line

• Try it!
  - it should sound like a "pling"
The **UIGuitarView** object communicates with the synthesizer
- The view sends the synthesizer a message when a string is stroked
- This connection is already established in the template (in **GuitarViewController**)

**GuitarViewController** instantiates both the synth and the **UIGuitarView**
- in `loadView` it sets the synth delegate to the **GuitarSynthesizer**
- It also links the **UIFretBoard** object (which we'll use later)

The **UIGuitarView** sends a **stringPlucked** message to its delegate
- It has parameters for the string number (0-5), the x-position of the pluck (0.0 -- 1.0) (we will use this later), and the velocity of the pluck

**GuitarSynthesizer** needs to respond to this message
- fill in the empty method definition
- for the moment, just call **fillWithNoise** on the delay line
- later, we will use the other parameters to control the sound
Creating DigitalString

- Create a class to encapsulate a whole string model (a single string of the guitar)
  - The guitar will eventually need six of these
  - Call it DigitalString

- It should be initialised with a and a damping value
  - It should have a `newSample` method which returns a new sample
    - computed exactly as it was in the `fillBuffer` method
  - And a `setDelay` method, which sets the delay of the delay line
  - And a `pluck()` method, which takes a velocity (i.e. how hard it is plucked)
    - should range from 0.0 -- 1.0
  - Modify the DelayLine's `fillWithNoise` to take a scale parameter
    - pass the scale parameter from `pluck` to `fillWithNoise`
    - just multiply the random value by this scale!

- Now replace the instance of DelayLine in GuitarSynthesizer with a DigitalString
  - call `pluck` on it every time a string is touched (when a `pluckString` message is received)
    - use the velocity passed in from the UIGuitarView and pass it to `pluck`
  - and use `newSample` to compute the new output in `fillBuffer`
6 string guitar

• Instead of one single `DigitalString`, add an array of 6 `DigitalStrings` to `GuitarSynthesizer`
  
  ```c
  DigitalString *strings[6];
  ```
  
  ▫ **remember to initialise all of them!**

• In `fillBuffer`, the output value `v` is just the sum of the string values, divided by six

• Initialise each string to a different delay line length (but same damping)
  ▫ choose any delays (less than the maximum delay for the delay line!)
  ▫ Now, use the string index from the pluck detection to pluck the appropriate string

• Test it!
  ▫ You should have a tinny sounding, hopelessly tuned, but responsive guitar!
Tuning the guitar

- For each string, we need to work out how to tune it
  - The tuning is given by the delay line length
  - longer delays -> lower pitch
  - actually delayLength = sampleRate/frequency

- Add a `setNote` method to `DigitalString` to take a note number rather than a delay length
  - (it will still be an int though)
  - we will compute the appropriate delay and then call `setDelay`

- We will use MIDI note numbers
  - MIDI note 60 == middle C
  - each increase by 1 is a semitone up, decrease by 1 is a semitone down

- We need to compute `delayLength == sampleRate/frequency`
  - The sampleRate is 44100
  - Computing the `frequency` is trickier...
Frequency computation

- To convert a note number to a frequency, you need to be aware that note numbers are exponential in frequency
  - high C (note 72) is twice the frequency of middle C (60) which is twice that of low C (note 48)
  - Each 12 steps represents a doubling in frequency (an octave)

- As a reference point, middle C is 261 Hz by definition
  - So we can compute other notes relative to that
    frequency = 261*adjustment;

- The adjustment must be 1 for +12, 0.5 for -12, 0.25 for -24 and so on
  - i.e. $2^\left(\frac{\text{noteNumber}-60}{12}\right)$
    adjustment = pow(2.0, (noteNumber-60.0)/12.0);

- That's it!
- Initialise the strings with note numbers now
  - Standard open guitar strings are note numbers 40 45 50 55 59 64 (E A D G B E)

- Try it!
String damping model

• The strings sound very "sharp" and tinny, because they have no frequency damping  
  ▫ in real life, high frequencies decay away more quickly

• We can add a loop pass filter to the loop to simulate this

  ![Diagram of Delay Line and Lowpass filter with damping]

• We will use a very simple one-pole filter  
  ▫ \( x = \alpha x + (1-\alpha)\text{newSample} \)
Adding the filter

- In `DigitalString`, add a (double) `filterTemp` variable to hold the previous output of the string
  - And an `alpha` variable to represent the filter coefficient

- The coefficient `alpha` of the filter can be computed by
  - `alpha = exp(-2*M_PI*frequency/44100.0)`
  - where `frequency` is the filter we want to cutoff at
  - Use `32*noteFrequency` for this value — this is fairly realistic

- In `newSample`, add a line like
  - `filterTemp = alpha * filterTemp + (1-alpha) * v`

- use `filterTemp` as the feedback into the delay line, and also as the return value from `newSample`

- Compile and test
  - The strings should sound much better now!
Using the fretboard

• To use the fretboard, implement the method `fretsUpdated` in `GuitarSynthesizer`
  ▫ This will receive a `int[]` array with 6 elements
  ▫ Each element specifies the number of notes (semitones) above the base string to play for each string
  ▫ an array of `[0 0 2 0 0 1]` means second string +2 semitones, sixth string +1 semitone, all others unchanged

• The synthesizer will get a message each time the frets are changed
  ▫ At this point, check all `DigitalString` instances and see if the note needs to change
  ▫ If so, send them a `setNote` message to update their notes
    ▪ The note is the string base note + the fret offset for that string

• Test!
  ▫ The guitar should now be playable!
Correcting the tuning

• One problem with this model is that the delay lines are always integer length
  ▫ This means the possible frequencies are quite limited
  ▫ Because we throw away the fractional part when computing the delay line length, the notes are all out of tune!

• We will use an allpass filter
  ▫ this filter passes all frequencies equally, but induces a phase shift (a delay)

• The formula for computing a allpass filtered sample with the type of allpass filter we will be using is:
  ▫ \( y(t) = \tau u(x(t) - y(t-1)) + x(t-1) \)

• Extend DelayLine, adding instance variables for \( \tau \), the previous allpass output \( y(t) \) (e.g. called lastAllpass) and the previous delay line value (e.g. lastDelay)
Using the allpass filter

- Now, in `getOutput` compute the allpass output with something like:
  - \[ v = \tau \cdot (\text{samples}[\text{readHead}] - \text{lastAllpass}) + \text{lastDelay}; \]
  - \[ \text{lastDelay} = \text{samples}[\text{readHead}]; \]
  - \[ \text{lastAllpass} = v; \]
- `newSample` remains unchanged! It's only the readout which needs to be modified

- To compute \( \tau \) for a given delay, compute the fractional part of the delay (i.e. the total delay - the integer part)
  - \[ \tau = \frac{1 - \text{fractionalPart}}{1 + \text{fractionalPart}} \]

- Make sure you pass a `double` to the delay line `setDelay` when you modify it in `DigitalString` (e.g. from `setNote`)
  - Now delay values like 140.43 should work fine

- The strings will be in tune now -- the difference might be small, but it is important!
Better pick model

- Using white noise isn't very realistic
  - much better results can be had by using measured impulses

- Measuring impulses is relatively hard
  - There is, handily enough, a guitar impulse provided for you in the project
  - It's called \texttt{pluck.wav}

- We need to load it into a form where we can put into the delay line
  - \texttt{loadWaveFileRaw} takes a filename (without the .wav extension) and returns a \texttt{WaveData} structure
  - has the PCM data in \texttt{data}, and the length in \texttt{nSamples}

- Note: to use the values in \texttt{data}, you must first cast it to \texttt{SInt16} *

\begin{verbatim}
SInt16 *pcmData = (SInt16*) wavefile->data;
pcmData[0]; // first sample -- OK
pcmData[wavefile->nSamples-1]; // last sample -- OK
data[0]; // DO NOT DO THIS -- you must cast the data!
\end{verbatim}
Using the pick model

• Add a `WaveData` member to `GuitarSynthesizer`, and load it when you initialise the synthesizer
  ▫ Note: we only want one impulse to be shared among all strings
  ▫ it would be wasteful to load multiple copies of the impulse

• Pass a pointer (i.e. `WaveData *`) to the WaveData structure into each `DigitalString` when you construct it

• In `DigitalString` add variables to represent the `WaveData *` structure, the current sample index inside the impulse, and for the current pluck velocity

• Now, instead of calling `fillWithNoise` on the delay line in pluck
  ▫ set the current pick velocity to the value passed in
  ▫ reset the sample index for the impulse to zero (restart it)
Picking

• In **newSample**
  ▫ check if there are still samples to play in the impulse
  ▫ if so, copy a sample into the delay line (scaled by the current velocity), and advance the pointer

• Test it!

• The guitar should sound much, much better

• Try using pluck-reverb.wav instead
  ▫ This adds some reverberation, so that the sound sustains for longer...
Modelling pick position

• If you pluck a string near the end, it sounds different than if you pluck it in the middle

• We can simulate this using "comb filtering"
  ○ comb filtering is just adding a delayed copy of a signal
  ○ \[ y(t) = x(t) + x(t-n) \]

• Add another delay line to the DigitalString
  ○ Each time the strings frequency changes, set the delay length to \[ \text{xposition} \times (\text{string delay time}) \]
  ○ \text{xposition} is the value passed into pluckString which ranges from 0.0 -- 1.0
  ○ \text{string delay time} is just the length of the delay calculated for the main feedback loop

• Just feed the output from the impulse to this delay line
  ○ Add the delay line output to the result just before injecting it into the delay line

```plaintext
// get impulseValue from the impulse wave...
[combDelay newSample:impulseValue];
impulseValue = v + [combDelay getOutput];
// feed impulseValue into the delay line...
```

• Test it!
  ○ It should sound different near the middle of the string
Optional Extras

• If you’re feeling ambitious:
• Make the string play a note as soon as the fret changes, so you can preview the note
  ▫ or make it play the whole chord on all six strings...

• Replace the UIFretView with a view that lets you select chords directly
  ▫ e.g. from a list view
  ▫ chord tables can be found online

• Put the GuitarViewController in a UITabBarController and add a controls page
  ▫ Add controls to select different picks
    ▪ e.g. choose either the pluck model or the simple white noise model
    ▪ noise model is better with distortion...
  ▫ Allow the user to select different tunings
    ▪ different tunings simply require different base values for each string
    ▪ e.g. get drop D by using the set 38, 45, 50, 55, 59, 64