Generating Sound

EECS 4462 - Digital Audio

October 22, 2018

Plugins vs Apps

- So far, we have implemented plugins that receive input from a host and provide output back to the host
- With JUCE, one can also create standalone apps
- We will look at how to do that in the context of generating sound

Important Class: AudioAppComponent

- Our app must inherit from AudioAppComponent
- AudioAppComponent takes care of connecting to the audio inputs and outputs of your computer
- Need to implement three methods that should sound familiar

```
prepareToPlay()
releaseResources()
getNextAudioBlock()
```


YORK UNIVERSITÉ UNIVERSITY

Two important methods to call

- To get access to two audio inputs and outputs, we need to call

```
setAudioChannels(2,2);
```

- This call will also start the loop of calling **`getNextAudioBlock()`**
- When finished, we need to call

```
shutDownAudio();
```

White noise generation

- Let's examine the code in the white noise generator tutorial
- Important: **`getNextAudioBlock()`** receives an **`AudioSourceChannelInfo`** as an argument
    - Just a struct that contains an audio buffer and two ints: The first sample to write at, and how many samples to write

Sine wave generation

- Let's examine the code in the sine wave generator tutorial (3 different versions)
- V1 creates a sampled version of the sine function
- V2 adds smooth transitioning to the new frequency when the slider is changed
- V3 adds a level slider

Virtual Instruments

- A virtual instrument is a piece of software that receives MIDI events as input, and produces audio samples as output
- This can be quite complicated if we want to produce sounds rich in frequency content
  - Let's listen to some examples…
- We will use JUCE to create a sine wave based virtual instrument

YORK U UNIVERSITÉ UNIVERSITY

Important Class: Synthesiser

- The base class for virtual instruments in JUCE
- Contains a collection of **`SynthesiserSound`**
  - Each sound can apply to specific notes or specific MIDI channels
- Contains a collection of **`SynthesiserVoice`**
  - Each voice can sound independently
  - When playing multiple notes at the same time, each note is a different voice
  - All audio rendering happens in method **`renderNextBlock`** of **`SynthesiserVoice`**

MIDI Synthesiser Tutorial

- Let's examine the code in the MIDI Synthesiser tutorial
- The main app makes a **`MidiKeyboardComponent`** visible, and delegates all audio rendering to a subclass of **`AudioSource`** called **`SynthAudioSource`**
- **`AudioSource`** is a superclass of **`AudioAppComponent`** and is the one that declares methods

```
prepareToPlay()
releaseResources()
getNextAudioBlock()
```

MIDI Synthesiser Tutorial

- **`SynthAudioSource`** contains a **`Synthesiser`** object
- Four voices and one sound are added to the synthesiser
- **`SineWaveVoice`** inherits from **`SynthesiserVoice`**
- **`SineWaveSound`** inherits from **`SynthesiserSound`**
- **`getNextAudioBlock`** receives a **`MidiBuffer`** from the keyboard and passes it to the **`renderNextBlock`** function of the synthesiser, which in turn calls the **`renderNextBlock`** method of each voice

YORK U UNIVERSITÉ UNIVERSITY

SynthesiserVoice::renderNextBlock

- **`renderNextBlock`** creates sample values as before
- Rather than setting direct values to samples, use the **`addSample`** method
- If the voice has finished producing sound, call **`clearCurrentNote`** to free the voice for the next note
- Can choose to have sound trail off slowly by continuing to produce sound with decreasing level (see code)

SynthesiserVoice (other methods)

- **`canPlaySound`** determines if the voice can play a particular sound
- **`startNote`** initializes class attributes for the next note to be rendered
- **`stopNote`** indicates what happens when a note has ended
- **`pitchWheelMoved`**, **`controllerMoved`** etc. react to the corresponding MIDI events

YORK UNIVERSITÉ UNIVERSITY

Wavetable Synthesis

- Real, complex sounds are composed of hundreds of frequencies
- Creating these by calculating and adding as many sine waves is very computation-intensive
  - See v1 of the wavetable synthesis tutorial
- Using wavetables, i.e. precomputed signals, we can accelerate computation by interpolating on the precomputed signal rather than computing directly
  - See v2 of the wavetable synthesis tutorial