Coding workshop: Making some sound in JUCE

Introduction

In this worksheet, we will add some basic sound synthesis code to our JUCE application. We will hook up the buttons and sliders so that they control the synthesis. We are not going to get too deep into sound synthesis as this is not a principal aim for this course, but we'll certainly implement some basic sounds.

About the audio buffer

The standard way to get audio from your program out to speakers or headphones connected to the computer is by generating a sequence of blocks of numbers. Each block might contain 512 or 1024 numbers or some other power of 2. The block is sent to the soundcard and played, then we are given another block to write into. The blocks are played sequentially, resulting in continuous sound.

This is how things work in JUCE. Periodically, the getNextAudioBlock will be called for you. It passes you a block. Your job is to fill up the block with numbers.

White noise

The most straightforward sound to generate is random noise. To make random noise, we just fill up the buffer with a sequence of random numbers. Let's try it. We can use the Random class from JUCE to generate random numbers (there are other ways, of course).

In the MainComponent.h file, private section:

```
Random random;
```

Add this code to MainComponent.cpp in the getNextAudioBlock function:

```
void MainComponent::getNextAudioBlock (const AudioSourceChannelInfo& bufferToFill)
{
    auto* leftChannel = bufferToFill.buffer->getWritePointer(0, bufferToFill.startSample);
    auto* rightChannel = bufferToFill.buffer->getWritePointer(1, bufferToFill.startSample);

    for (auto i=0;i<bufferToFill.numSamples; ++i)
    {
        leftChannel[i] = random.nextFloat() * 0.125f;
        rightChannel[i] = random.nextFloat() * 0.125f;
    }
}
```

Notes:

- This assumes there are only two channels (check your call to setAudioChannels in the MainComponent constructor and look up the parameters for that function in the JUCE documentation)
- It makes a nasty noise, so please take off headphones before running this code.

Controlling the sound with the buttons

Now let's control the sound with our user interface.

Stop and start

Add a boolean to the private section of MainComponent.h:

```
bool playing;
```

Set it to false in the prepareToPlay function:

```
playing = false;
```

Only generate the output if playing is true - put this at the top of the getNextAudioBlock function's code:

```
if(!playing)
{
  bufferToFill.clearActiveBufferRegion();
  return;
}
```

Toggle playing using the play and stop buttons:

```
void MainComponent::buttonClicked(Button* button)
{
    if (button == &playButton )
    {
        playing = true;
    }
    if (button == &stopButton )
    {
        playing = false;
    }
}
```

You should be able to switch the sound on and off using the play and stop buttons.

Gain slider

Now for the gain slider. Add a variable to store the current gain to the private section of MainComponent.h

```
double gain;
```

Initialise in prepareToPlay:

```
gain = 0.5;
```

Update when they move the slider:

```
void MainComponent::sliderValueChanged(Slider* slider)
{
    if (slider == &gainSlider)
    {
        DBG("MainComponent::sliderValueChanged: gainSlider " << gainSlider.getValue() );
        gain = gainSlider.getValue();
    }
}
```

Use the gain variable in the audio generator loop (in getNextAudioBlock):

```
for (auto i=0;i<bufferToFill.numSamples; ++i)
{
  leftChannel[i] = random.nextFloat() * 0.125 * gain;
  rightChannel[i] = random.nextFloat() * 0.125 * gain;
}
```

One problem - ideally, the signal we send to the audio output should not go higher than 1. So, we need to change the range of the slider to be 0 to 1. In the MainComponent constructor:

```
gainSlider.setRange(0, 1);
```

You should now be able to vary the gain/level of the sound using the slider.

Making a (slightly) better sound

White noise is not a very interesting sound. Babies like it, but I prefer periodic waveforms. To make a periodic (repeating) waveform, we need to remember where we were in the waveform between calls to getNextAudioBlock. Add a new data member variable to the private section of MainComponent:

```
float phase;
```

Initialise it in prepareToPlay:

```
phase = 0;
```

Now update the audio generating loop to this:

```
for (auto i=0;i<bufferToFill.numSamples; ++i)
{
  auto sample = fmod(phase, 1.0f);
  phase += 0.005;
  leftChannel[i] = sample * 0.125 * gain;
  rightChannel[i] = sample * 0.125 * gain;
}
```

This should generate a repeating linear ramp. Can you work out what fmod is doing here?

Then a classic reggae sound system siren

We are going to add one more level to the sound synthesis, so we can create a reggae sound system style 'police siren'. To do that, we need to increase the frequency of the waveform over time as well as the phase. Add a new variable to private in MainComponent:

```
double dphase;
```

Set it to zero in prepareToPlay:

```
dphase = 0;
```

Change the audio generating loop to this, so that it uses dphase to increase the frequency of the waveform:

```
for (auto i=0;i<bufferToFill.numSamples; ++i)
{
  auto sample = fmod(phase, 1.0f);
  phase += fmod(dphase, 0.01f);
  dphase += 0.0000005f;
  leftChannel[i] = sample * 0.125 * gain;
  rightChannel[i] = sample * 0.125 * gain;
}
```

Then finally, reset dphase when they click the play button in buttonClicked:

```
if (button == &playButton )
{
  playing = true;
  dphase = 0;
}
```

Check it out!

Next steps

- Can you implement a sound based on a sine wave?
- Can you implement a variable speed siren sound?
- Can you separate the sound generation into functions so it is possible to switch between the different sounds via some sort of GUI?

Conclusion

In this worksheet, we have explored some simple sound synthesis techniques, to familiarise ourselves with the audio buffer in JUCE. We created a white noise sound, a saw tooth sound and a siren sound.