Coding workshop: Playing audio files

Introduction

In this worksheet, we will move from basic sound synthesis to playing back audio files. Playing audio files is similar to audio synthesis, in that we need to fill up an audio buffer with some numbers. The difference is that we need to obtain the numbers from an audio file instead of generating/ synthesizing them ourselves.

Setting up the AudioFormatManager

The first step is to set up an AudioFormatManager. This is a class from the JUCE API that helps us to work with different audio file formats. For example, mp3 files and WAV files.

Add this to the MainComponent.h file, in the private section of the class definition:

```
AudioFormatManager formatManager;
```

Then add this to the MainComponent constructor:

```
formatManager.registerBasicFormats();
```

If you are wondering what the basic formats are, try adding this code to the constructor, after you call registerBasicFormats:

```
for (int i=0;i<formatManager.getNumKnownFormats(); i++){
  std::string s =
  formatManager.getKnownFormat(i)->getFormatName().toStdString();
  std::cout << i << " " << s << std::endl;
}
```

I see output like this when I run the program:

```
0 WAV file
1 AIFF file
2 FLAC file
3 Ogg-Vorbis file
4 MP3 file
```

You might see a different list if you are running on a different operating system. I am running on Linux in this case.

When you get to playing audio files later, those are the formats JUCE will be able to play.

Set up an AudioTransportSource

Now we are going to use some more JUCE classes to create a source of audio data from an audio file.

Add this variable to your MainComponent.h file, in the private section of the class definition:

```
AudioTransportSource transportSource;
```

In the prepareToPlay function in MainComponent.cpp, set up the add the following:

```
transportSource.prepareToPlay (samplesPerBlockExpected, sampleRate);
```

Then clean things up in MainComponent::releaseResources:

```
transportSource.releaseResources();
```

Now we'll ask the transportSource for some audio whenever we are asked for audio. In the MainComponent getNextAudioBlock function, comment out your sound synthesis code, and add this code:

```
transportSource.getNextAudioBlock (bufferToFill);
```

The complete function should look something like this, including the commented out sound synthesis code:

```
void MainComponent::getNextAudioBlock (const AudioSourceChannelInfo& bufferToFill)
{
    if(!playing)
    {
        bufferToFill.clearActiveBufferRegion();
        return;
    }
    transportSource.getNextAudioBlock (bufferToFill);
}
```

Now, this is where it gets interesting from an object-oriented perspective. Note that our MainComponment class inherits from AudioSource:

```
class DJAudioPlayer  : public AudioSource
```

This forces us to implement the audio methods prepareToPlay, getNextAudioBlock and leaseResources. Now - check out the JUCE API documentation for AudioTransportSource. You will notice that it also inherits from AudioSource, so it has those methods too. That's why it has a getNextAudioBlock function. We basically delegate the buffer filling to the AudioTransportSource instead of doing it ourselves.

Providing a file to the AudioTransportSource

Now we need an audio file to play. We will start by hard coding in a file. Reading audio files in JUCE involves the use of several objects. We wrap the objects around each other, and each layer we wrap increases the abstraction.

In MainComponent.h, put this in the private section of the class definition:

```
std::unique_ptr<AudioFormatReaderSource> readerSource;
```

Notes:

- this is a pointer as we don't want to construct it until we have a file to play
- it is a unique ptr as that should be what pointers always start as until there is an apparent reason why they should not be unique (i.e. need to be passed around)

In the MainComponent constructor:

```
// linux/ mac (unix) style
URL audioURL{"file:///home/scratch/audio/samples/cr/RIDED0.wav"};
```

You will need to edit that file path so it points at an audio file on your system. If you are on Windows, use a URL like this, making sure this file exists on your system:

```
// windows style
URL audioURL{"file:///C:\Documents%20and%20Settings\myusername/mywavfile.wav"};
```

Then on any platform, continue with this code:

```
AudioFormatReader* reader =
formatManager.createReaderFor (audioURL.createInputStream (false));

if (reader != nullptr)
{
  std::unique_ptr<AudioFormatReaderSource> newSource
      (new AudioFormatReaderSource (reader, true));
  transportSource.setSource (newSource.get(),
                     0, nullptr, reader->sampleRate);
  readerSource.reset (newSource.release());
}
else
{
  std::cout << "Something went wrong loading the file " << std::endl;
}
```

There is a lot of clever pointer work going on in that code to make it more robust. Here are some notes about that:

- AudioFormatReader* reader is the lowest level layer in the file reading. It does the hard work of reading numbers out of the audio file
- reader is a pointer as that is what we get back from createReaderFor.formatManager
- Since reader is a pointer, the data it points to is not destructed when it goes out of scope. We need it to persist as we need to keep streaming data in from the file.

- We create a temporary, local scope unique_ptr called newSource. Being local, if anything goes wrong, it will be correctly destructed when it goes out of scope (when we exit the function). We call this 'exception-safe'.
- Once (if) the newSource variable has been created successfully, we hand over the object it points to readerSource, which has object level scope. That means it can persist as long as the MainComponent object does. The handover is readerSource.reset (newSource.release());
- When we create newSource, we pass it the reader and the value true. It needs the reader to be able to read numbers from the file. true tells it that it should destruct the reader when it has finished with it. That is why reader needs to be a pointer - to allow newSource to clear up when it needs to
- Why is reader not in object scope? We could put reader in object scope instead of function scope. Still, we don't because newSource could destruct reader at any time, and by keeping it out of object scope, we prevent the object's other functions from accessing reader after it has been destructed (dangling pointer). In function scope, it can only be seen in this function.

Making the start, stop and gain control work

Let's hook up the buttons. We'll make the start button rewind and play the file. We'll make the stop button stop playback, and we'll hook up the gain slider.

Play button

In MainComponent.cpp: MainComponent::buttonClicked

```
if (button == &playButton )
{
  playing = true;
  dphase = 0;
  transportSource.setPosition(0);
  transportSource.start();
}
```

Now clicking the play button should rewind the player and play from the start.

Stop button

In MainComponent.cpp: MainComponent::buttonClicked

```
if (button == &stopButton )
{
  playing = false;
  transportSource.stop();
}
```

Now clicking the stop button should stop playback.

Gain slider

In MainComponent::sliderValueChanged:

```
if (slider == &gainSlider)
{
  std::cout << gainSlider.getValue() << std::endl;
  gain = gainSlider.getValue();
  transportSource.setGain(gain);
}
```

Moving the gain slider should change how loud the sound is.

Making a file chooser

Finally, let's allow the user to select a file for playback.

Add a new button to the interface, in MainComponent.h, in the private section of the class definition:

```
juce::TextButton loadButton;
```

Now add a file chooser component to the private section of MainComponent.h:

```
juce::FileChooser fChooser{"Select a file..."};
```

Set the load button up in MainComponent::MainComponent:

```
addAndMakeVisible(loadButton);
loadButton.addListener(this);
loadButton.setButtonText("LOAD");
```

Set the size in MainComponent::resized:

```
playButton.setBounds(0, 0, getWidth(), getHeight()/5);
stopButton.setBounds(0, getHeight()/5, getWidth(), getHeight()/5);
loadButton.setBounds(0, getHeight()/5 * 2, getWidth(), getHeight()/5);
gainSlider.setBounds(0, getHeight()/5 * 3, getWidth(), getHeight()/5);
```

Set up the event listener in MainComponent::buttonClicked:

```
if (button == &loadButton)
{

    // - configure the dialogue
    auto fileChooserFlags =
    FileBrowserComponent::canSelectFiles;
    // - launch out of the main thread
    // - note how we use a lambda function which you've probably
    // not seen before. Please do not worry too much about that
    // but it is necessary as of JUCE 6.1
    fChooser.launchAsync(fileChooserFlags,
                         [this](const FileChooser& chooser)
    {
        auto chosenFile = chooser.getResult();
        auto* reader = formatManager.createReaderFor (chosenFile);

        std::unique_ptr<AudioFormatReaderSource> newSource
              (new AudioFormatReaderSource (reader, true));
        transportSource.setSource (newSource.get(), 0,
                                   nullptr, reader->sampleRate);
        readerSource.reset (newSource.release());
    });
}
```

Now when you click the load button, it should pop up a file browser with which you can select an audio file to play.

Some notes:

- chooser.browseForFileToOpen() triggers the file browser
- chooser.browseForFileToOpen() returns false if they click cancel
- The code for dealing with the file is more or less the same as above, except this time we pass a file to createReaderFor instead of an URL.

Clear it up with a loadURL function

We now have very similar code in the constructor and the button listener function. It would be better to put this code in one place. Add this line to the private section of your MainComponent.h file:

```
void loadURL(URL audioURL);
```

Now go into MainComponent.cpp and put this code at the end:

```
void MainComponent::loadURL(URL audioURL)
{
```

```
    auto* reader = formatManager.createReaderFor(audioURL.createInputStream(false));
    if (reader != nullptr) // good file!
    {
        std::unique_ptr<AudioFormatReaderSource> newSource
             (new AudioFormatReaderSource (reader, true));
        transportSource.setSource (newSource.get(), 0, nullptr, reader->sampleRate);
        readerSource.reset (newSource.release());
    }
}
```

Now remove the code from the MainComponent constructor that deals with the file loading - we don't need to load a file there any more. Update the loadButton code in the MainComponent::buttonClicked function to this:

```
if (button == &loadButton)
{
    auto fileChooserFlags =
    FileBrowserComponent::canSelectFiles;
    fChooser.launchAsync(fileChooserFlags, [this](const FileChooser& chooser)
    {
        auto chosenFile = chooser.getResult();
        loadURL(URL{chosenFile});
    });
}
```

Now compile and test that the loadButton works.

Challenges:

Can you create a slider position control that lets you move the 'play head' of the audio player to where you want it? (clue - call setPosition on AudioTransportSource)

Conclusion

In this worksheet we have learned how to play audio files using classes from the JUCE library.