Buubuu Connect

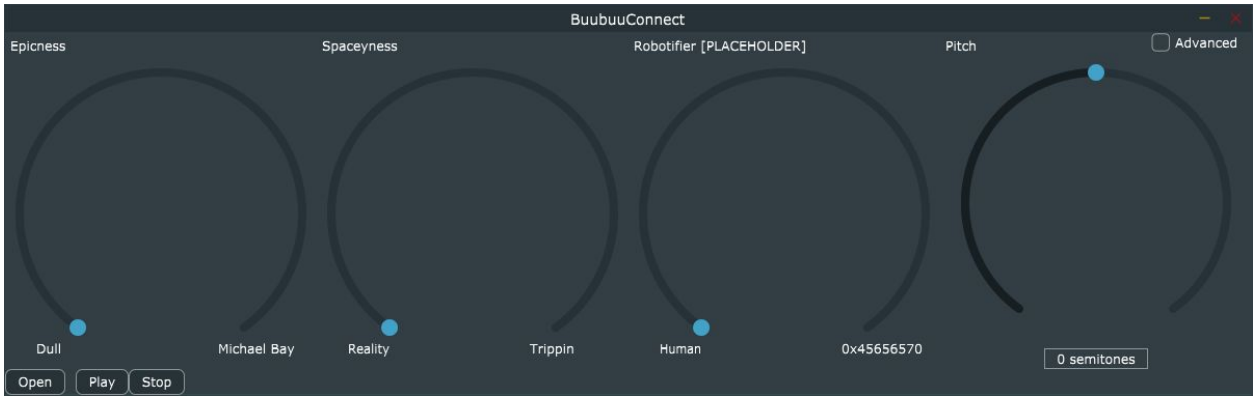
Kaya Bax
Buubuu Sounds
11-05-1996

Index

Interface

This program has 2 modes, set with the toggle button at the top right corner:

Simple Mode



The main mode this application starts in. This mode has the following 4 parameters:

- **Epicness**: Changes the *Bass* & *Mids* of the EQ, the *Threshold*, *Ratio*, and *Gain* of the Compressor, and the *Length* and *Mix* of the Reverb, for exact values, look at the MainComponent.cpp page.
- **Spaceyness**: Lowers the *Sub* & *Treble* frequencies in the EQ, sets the *Attack* & *Release* of the Compressor, and alters the *Length* and *Mix* of the Reverb. Values can be found on the MainComponent.cpp page.
- **Robotifier**: Placeholder, has no current usage.
- Pitch: Placeholder, has no current usage.

Changing the parameters of these dials will affect the underlying effect modules. These effects are shown when toggling the upper-right button:

Advanced Mode

This mode lets you change the individual sliders of the modules.

Compressor:

- Threshold: Sets the threshold in dB where the compressor should start compressing;
- Ratio: Sets a value from 1 to 100 on how much the compressor should change. (eg. a Ratio of 10 means you need 10dB on the input signal for 1dB on the output signal)
- Attack: Sets the attack time in ms.
- Release: Sets the release time in ms.
- Gain: Compensate the loss in volume by applying a gain post-compression.

(NOTE: Setting the Attack or Release shorter than 1/1000th of your samplerate will result in audio glitches.)

EQ:

- Sub: Changes a low shelf -10 to +10 dB.
  Cutoff Frequency: 75 Hz
- Bass: Changes a parametric peak filter -10 to +10 dB.
  Cutoff Frequency: 100 Hz

- Mids: Changes a parametric peak filter -10 to +10 dB.
  Cutoff Frequency: 900 Hz
- Treble: Changes a high shelf -10 to +10 dB
  Cutoff Frequency: 8000 Hz

All filters have a Q of 0.5.

Reverb:

- Length: sets the length in seconds, from 0 to +- 2 seconds of length, RT60.
- Mix: Sets the Wet/Dry balance, from 0/100 to 100/0.

The damping and width are pre-determined, for a fixed sound of the reverb.

MainComponent.h

This is the header of the main window, where we put all sub components, windows, and general functionality of our program.
Interesting are the following Private Functions:

/* SimpleMode Components */
Used to declare the names for alle the dials (called with the Slider class in JUCE) and their labels and texts.

/* Player Components */
Here we have the buttons for opening .wav files, and to start and stop the playback.

```
TextButton openButton;
TextButton playButton;
TextButton stopButton;
```

Following are the functions to change where the application listens to and gets his blocks of audio from:

```
enum transportState
{
    Stopped,
    Starting,
    Stopping,
    Playing
};

transportState state;
void transportStateChanged(transportState newState);
void changeListenerCallback(ChangeBroadcaster* source) override;

AudioFormatManager manager;
std::unique_ptr<AudioFormatReaderSource> playSource;
AudioTransportSource transport;
```

/* Effect Components */
Here we add the 3 child components with both their dials and their respective effect modules.

```
Compressor comp;
EQ eq;
Verb reverb;
```

MainComponent.cpp

Constructor()

We start by setting the size of the SimpleMode window:

```
setSize(1200, 350);
```

Sliders

We then set the sliders. example given is with the first slider, the **dramaSlider**.
Setting the slider to be a dial:

```
dramaSlider.setSliderStyle(Slider::SliderStyle::RotaryVerticalDrag);
```

Making sure we have no textbox showing the value of our slider, since we want to use non-scientific dials in our Simple Mode:

```
dramaSlider.setTextBoxStyle(Slider::NoTextBox, true, 0, 0);
```

Set the range from 0 to 100, with increments of 1.

```
dramaSlider.setRange(0, 100, 1);
```

This next one is interesting. It's a lambda function, added from C++14 and onwards, that lets us assign a special listener, that lets us change other parameters when the value of our dial changes:

```
dramaSlider.onValueChange = [this]
{
    eq.setBass(dramaSlider.getValue() * 0.03);
    eq.setMids(0 - (dramaSlider.getValue() * 0.03));
    comp.setThreshold(0 - (dramaSlider.getValue() * 0.1));
    comp.setRatio(1 + dramaSlider.getValue() * 0.02);
    comp.setGain(dramaSlider.getValue() * 0.003);
    reverb.setLength((dramaSlider.getValue() * 0.1) + (spaceySlider.getValue() * 0.7));
    reverb.setMix((dramaSlider.getValue() * 0.2) + (spaceySlider.getValue() * (0.6 - (dramaSlider.getValue() * 0.002))));
};
```

Lastly, we set the starting value at 0, so it updates itself on start to prevent bugs.

```
dramaSlider.setValue(0);
```

Adding the effect modules

We then add the dials and effects of the compressor, eq, and reverb, with the addChildComponent() function.
The reason we use this function over addAndMakeVisible() is because we want their dials to be hidden in the Simple Mode, but their functionality working.

```
addChildComponent(comp);
addChildComponent(eq);
addChildComponent(reverb);
```

modeSwitch toggle:

This toggle lets us switch between Simple and Advanced mode.
First we set the location and size for our button:

```
modeSwitch.setBounds(getRight() - 100, 0, 30, 20);
```

We then set the text and the location of the text, letting it readjust it's size to fit the text:

```
modeSwitch.setButtonText("Advanced");
modeSwitch.changeWidthToFitText();
addAndMakeVisible(modeSwitch);
```

This lambda-listener function toggles the visibility of the 3 effect components, and rescales the MainComponent window to fit these child components (with a cool animation).

```
modeSwitch.onStateChange = [this]
{
    if (modeSwitch.getToggleState() == false)
    {
        comp.setVisible(false);
        eq.setVisible(false);
        reverb.setVisible(false);
        if (getHeight() != 350) {
            int growth = 16;
            for (int i = 750; i > 350; i-=growth) {
                setSize(1200, i);
                growth *= 2;
            }
        }
        setSize(1200, 350);
    }
    else if (modeSwitch.getToggleState() == true)
    {
        comp.setVisible(true);
        eq.setVisible(true);
        reverb.setVisible(true);
```

```
                if (getHeight() != 750) {
                        int growth = 16;
                        for (int i = 350; i < 750; i+=growth) {
                                setSize(1200, i);
                                growth *= 2;
                        }
                }
                setSize(1200, 750);
        }
};
```

Player functions

Up next are the 3 buttons *Open, Play,* and *Stop.*
We set their location, size, text, their listener lambda function, and their visibility:

```
openButton.setBounds(0, getHeight() - 25, 25, 25);
openButton.setButtonText("Open");
openButton.changeWidthToFitText();
openButton.onClick = [this] {openButtonClicked(); };
addAndMakeVisible(openButton);

playButton.setBounds(openButton.getRight() + 10, getHeight() - 25, 25, 25);
playButton.setButtonText("Play");
playButton.changeWidthToFitText();
playButton.onClick = [this] {playButtonClicked(); };
addAndMakeVisible(playButton);

stopButton.setBounds(playButton.getRight(), getHeight() - 25, 25, 25);
stopButton.setButtonText("Stop");
stopButton.changeWidthToFitText();
stopButton.onClick = [this] {stopButtonClicked(); };
addAndMakeVisible(stopButton);
```

We then add a Listener for changes to the transport class:

```
transport.addChangeListener(this);
```

We make sure our program can open .wav and .aiff files with:

```
manager.registerBasicFormats();
```

Base JUCE stuff

We open 2 unused input channels, and 2 (stereo) output channels:

```
        // Some platforms require permissions to open input channels so request that here
    if (RuntimePermissions::isRequired (RuntimePermissions::recordAudio)
        && ! RuntimePermissions::isGranted (RuntimePermissions::recordAudio))
    {
        RuntimePermissions::request (RuntimePermissions::recordAudio,
                                     [&] (bool granted) { if (granted)  setAudioChannels (2, 2); });
    }
    else
    {
        // Specify the number of input and output channels that we want to open
        setAudioChannels (2, 2);
```

prepareToPlay()

Here we initialise our audio thread, where we set the samplerate of our child components with the samplerate we got from our audio device:

```
void MainComponent::prepareToPlay (int samplesPerBlockExpected, double sampleRate)
{
        double playBackRate = sampleRate;
        comp.setSampleRate(playBackRate);
        eq.setSampleRate(playBackRate);
        reverb.setSampleRate(playBackRate);

        transport.prepareToPlay(samplesPerBlockExpected, playBackRate);

        dramaSlider.setValue(0);
        spaceySlider.setValue(0);

}
```

Note: The playBackRate is added for the pitch-shift function, which will be added in a later update.

xButtonClicked()

openButton:

When the open button is clicked, it will let you search an audio file, and make sure the file is ready to play (filling the sample blocks of our audio thread with sample blocks from our file.):

```
        //choose a file
        FileChooser chooser("Choose a WAV file.",
File::getSpecialLocation(File::userDesktopDirectory), "*.wav;*.aiff", true, false);
        //if the user chooses a file
        if (chooser.browseForFileToOpen())
        {
                File myFile;
                //what did the user choose?
                myFile = chooser.getResult();
                //read the file
                AudioFormatReader* reader = manager.createReaderFor(myFile);

                if(reader != nullptr)
                {
                        //get the file ready to play
                        std::unique_ptr<AudioFormatReaderSource> tempSource(new
AudioFormatReaderSource(reader, true));

                        transport.setSource(tempSource.get());
                        transportStateChanged(Stopped);

                        playSource.reset(tempSource.release());
                }

        }
```

playButton:

All we do here, is change the transport state to it's Starting sequence, as well as update the compressor (added due to a bug of it otherwise not giving sound):

```
        comp.update();
        transportStateChanged(Starting);
```

stopButton:

This function changes the transportState to it's Stopping sequence:

```
transportStateChanged(Stopping);
```

transportStateChanged()

First we make sure the program only changes on switches, and not every time this function is called with the same state:

```
if (newState != state)
{
    state = newState;
```

We then set the transport position, as well as enabling/disabling the play & stop buttons as intended:

```
    switch (state)
    {
    case Stopped:
        playButton.setEnabled(true);
        stopButton.setEnabled(false);
        transport.setPosition(0.0);
        break;

    case Starting:
        playButton.setEnabled(false);
        stopButton.setEnabled(true);
        transport.start();
        break;

    case Stopping:
        playButton.setEnabled(true);
        stopButton.setEnabled(false);
        transport.stop();
        break;

    case Playing:
        playButton.setEnabled(false);
        stopButton.setEnabled(true);
        break;

    }
```

```
}
```

getNextAudioBlock()

Here, we get our sample blocks from the Audio Source, and we process the buffer with the compressor, EQ, and reverb effects:

```
bufferToFill.clearActiveBufferRegion();
transport.getNextAudioBlock(bufferToFill);
int channelAmount = bufferToFill.buffer->getNumChannels();
int sampleAmount = bufferToFill.buffer->getNumSamples();

comp.compress(channelAmount, sampleAmount, bufferToFill);
eq.process(channelAmount, sampleAmount, bufferToFill);
reverb.reverberate(channelAmount, sampleAmount, bufferToFill);
```

paint()

Used for the graphical interface, when it's static and not resizable. We only make sure there's a background, and a line underneath our SimpleMode to divide it from the child components:

```
g.fillAll (getLookAndFeel().findColour (ResizableWindow::backgroundColourId));
g.setColour(Colours::darkslategrey);
Line<float> line(Point<float>(0, 350), Point<float>(1200, 350));
g.drawLine(line, 2.0f);
```

resized()

We set the locations and sizes of our child components:

```
comp.setBounds(0, 350, 1200, 200);
eq.setBounds(0, 550, 600, 200);
reverb.setBounds(600, 550, 600, 200);
```

Compressor

Constructor()

We set the base size of this component, and set the title of it:

```
setSize(1200, 200);
compTitle.setText("Compressor", dontSendNotification);
compTitle.setBounds(0, 0, 100, 20);
addAndMakeVisible(compTitle);
```

We then set all stuff from our sliders, the same way we did in the MainComponent.
Our lambda function converts the dB value to a gain ratio value of the thresholdValue variable:

```
thresholdSlider.onValueChange = [this]
{
    thresholdValue = Decibels::decibelsToGain(thresholdSlider.getValue());
};
```

setX()

All the sliders and their value are private members, so to change them (especially from a mother component), we need functions to change them. For example:

```
void Compressor::setSampleRate(float sampleRate)
{
    this->sampleRate = sampleRate;
}
```

compress()

Since the buffer comes with two lists, we need to split them into two seperate channels first. Then, we need to check for every sample in that channel's buffersize (which is the same as the other buffersize.)
We change the samples if it's over the thresholdvalue, by compressing it over the duration of the Attack value, and removing the compresison if the input samples are under the threshold value, over the length of the Release value:

```
for (int channel = 0; channel < numChannels; channel++)
{
```

```
        float* const buffer = mainBuffer.buffer->getWritePointer(channel, mainBuffer.startSample);

        for (int sample = 0; sample < numSamples; sample++)
        {
            if (fabs(buffer[sample]) > thresholdValue)
            {
                if (compressionAmount < ratioSlider.getValue())
                {
                    compressionAmount += (ratioSlider.getValue() / attackValue);
                }
                else
                {
                    compressionAmount = ratioSlider.getValue();
                }

            }
            else if(fabs(buffer[sample]) <= thresholdValue)
            {

                if (compressionAmount > 1.0)
                {
                    compressionAmount -= (ratioSlider.getValue() / releaseValue);
                }
                else
                {
                    compressionAmount = 1.0;
                }

            }
            buffer[sample] *= thresholdValue + ((1 - thresholdValue) / compressionAmount);
            buffer[sample] *= gainValue;
        }
    }
```

update()

We can call this function in the audio initialisation, to initialise the thresholdValue and gainValue:

```
thresholdValue = thresholdValue = Decibels::decibelsToGain(thresholdSlider.getValue());
    gainValue = 1.0;
```

EQ

4-band graphic equalizer, for small Sub, Bass, Mid, and Treble changes.

All 4 bands have a duplicate: One set is for the left channel, one is for the right.
This means this component will only work in it's current state on a stereo setup.

Constructor()

Only used to set the sliders, theyr style, location, size, visibility etc.
Check the MainComponent on how this is done.
All sliders also have lambda functions, that converts the sliders value from dB to a gain ratio.
Eg:

```
subSlider.onValueChange = [this]
        {
            subGain = Decibels::decibelsToGain(subSlider.getValue());
            updateFilter(sampleRate);
        };
```

setX()

All the sliders and their value are private members, so to change them (especially from a mother component), we need functions to change them.

updateFilter()

We set the coefficients of our sub filters to be a low shelf, our bass filters to be a bell filter, our mids filters to also be a bell filter, and our treble filters to be a high shelf, with values gotten from the sliderGain values:

```
subFilterL.setCoefficients(coeffs.makeLowShelf(sampleRate, 75, 0.5, subGain));
        subFilterR.setCoefficients(coeffs.makeLowShelf(sampleRate, 75, 0.5, subGain));
        bassFilterL.setCoefficients(coeffs.makePeakFilter(sampleRate, 100, 0.5, bassGain));
        bassFilterR.setCoefficients(coeffs.makePeakFilter(sampleRate, 100, 0.5, bassGain));
        midFilterL.setCoefficients(coeffs.makePeakFilter(sampleRate, 900, 0.5, midGain));
        midFilterR.setCoefficients(coeffs.makePeakFilter(sampleRate, 900, 0.5, midGain));
        trebleFilterL.setCoefficients(coeffs.makeHighShelf(sampleRate, 8000, 0.5, trebleGain));
        trebleFilterR.setCoefficients(coeffs.makeHighShelf(sampleRate, 8000, 0.5, trebleGain));
```

Process()

We use the JUCE IIR DSP function processSampples(channel, sampleAmount) to process all samples through our filters:

```
float*  left = mainBuffer.buffer->getWritePointer(0, mainBuffer.startSample);
float*  right = mainBuffer.buffer->getWritePointer(1, mainBuffer.startSample);
subFilterL.processSamples(left , numSamples);
subFilterR.processSamples(right, numSamples);
bassFilterL.processSamples(left, numSamples);
bassFilterR.processSamples(right, numSamples);
midFilterL.processSamples(left, numSamples);
midFilterR.processSamples(right, numSamples);
trebleFilterL.processSamples(left, numSamples);
trebleFilterR.processSamples(right, numSamples);
```

Verb

This function is written exactly the same as the EQ, but instead of using the DSP::IIR class from JUCE, we use the DSP::Reverb class. This means:

Constructor()

We set the damping and width to be a fixed value:

```
params.damping = 0.5f;
params.width = 0.5f;
```

And change the length and dry/wet balance with lambda functions. Eg:

```
lengthSlider.onValueChange = [this]
{
    params.roomSize = lengthSlider.getValue() / 100.0;
    verb.setParameters(params);
};
```

Reverberate()

We call this function to process the left and right buffer with the DSP::Reverb::processStereo function:

```
float* const left = mainBuffer.buffer->getWritePointer(0, mainBuffer.startSample);
```

```
        float* const right = mainBuffer.buffer->getWritePointer(1,
mainBuffer.startSample);
        verb.processStereo(left, right, numSamples);
```