the demo project will display the incoming audio data as a three-dimensional spectrogram in the time (x-axis), frequency (y-axis) and amplitude (colour) domains. The values displayed on the screen will be updated 60 times a second and the window at any time frame may look something like this:

A screenshot of a computer screen

Description automatically generated

**The demo project window**

The code presented here is broadly similar to the **SimpleFFTExample** from the JUCE Examples.

**The Fast Fourier Transform**

A time or space domain signal can be converted to the frequency domain by using a transformation formula called the Fourier transform. A common efficient implementation of this transformation function is the Fast Fourier Transform or FFT, which is included in the JUCE DSP module and which we will use in this tutorial.

The FFT allows us to decompose an audio signal into its frequencies and represent the magnitude and phase information for each of these frequencies. Using its inverse function, we can revert the signal into its original domain thus making it really useful to process individual frequency components such as for filtering.

Since this only deals with displaying the audio data without actual processing for output, we focus on the forward FFT rather than the inverse FFT.

**Processing Audio Data**

**FFT Initialisation**

Currently our application does not display nor process any incoming audio signals so start by defining some useful constants for the FFT implementation:

static constexpr auto fftOrder = 10; // [1]

static constexpr auto fftSize = 1 << fftOrder; // [2]

private:

* [1] : The FFT order designates the size of the FFT window and the number of points on which it will operate corresponds to 2 to the power of the order. In this case, let's use an order of 10 which will produce an FFT with 2 ^ 10 = 1024 points.
* [2] : To calculate the corresponding FFT size, we use the left bit shift operator which produces 1024 as binary number 10000000000.

Next, declare private member variables required for the FFT implementation as shown below:

juce::dsp::FFT forwardFFT; // [3]

juce::Image spectrogramImage;

std::array<float, fftSize> fifo; // [4]

std::array<float, fftSize \* 2> fftData; // [5]

int fifoIndex = 0; // [6]

bool nextFFTBlockReady = false; // [7]

[JUCE\_DECLARE\_NON\_COPYABLE\_WITH\_LEAK\_DETECTOR](https://docs.juce.com/master/group__juce__core-system.html#ga851b02724a8158a7f746c66b5758b3dd) (SpectrogramComponent)

};

* [3] : Declare a [**dsp::FFT**](https://docs.juce.com/master/classdsp_1_1FFT.html) object to perform the forward FFT on.
* [4] : The fifo float array of size 1024 will contain our incoming audio data in samples.
* [5] : The fftData float array of size 2048 will contain the results of our FFT calculations.
* [6] : This temporary index keeps count of the amount of samples in the fifo.
* [7] : This temporary boolean tells us whether the next FFT block is ready to be rendered.

Now let's initialise these variables in the member initialisation list of our constructor like so:

SpectrogramComponent() : forwardFFT (fftOrder), spectrogramImage (juce::Image::RGB, 512, 512, true)

{

The FFT object has to be explicitly initialised with the correct order at this point.

In the overriden getNextAudioBlock() function, we simply push all the samples contained in our current audio buffer block to the fifo to be processed at a later time:

void getNextAudioBlock (const juce::AudioSourceChannelInfo& bufferToFill) override

{

if (bufferToFill.buffer->getNumChannels() > 0)

{

auto\* channelData = bufferToFill.buffer->getReadPointer (0, bufferToFill.startSample);

for (auto i = 0; i < bufferToFill.numSamples; ++i)

pushNextSampleIntoFifo (channelData[i]);

}

}

To push the sample into the fifo, implement the pushNextSampleIntoFifo() function as described below:

void pushNextSampleIntoFifo (float sample) noexcept

{

// if the fifo contains enough data, set a flag to say

// that the next line should now be rendered..

if (fifoIndex == fftSize) // [8]

{

if (! nextFFTBlockReady) // [9]

{

std::fill (fftData.begin(), fftData.end(), 0.0f);

std::copy (fifo.begin(), fifo.end(), fftData.begin());

nextFFTBlockReady = true;

}

fifoIndex = 0;

}

fifo[(size\_t) fifoIndex++] = sample; // [9]

}

* [8] : If the fifo contains enough data in this case 1024 samples, we are ready to copy the data to the fftData array for it to be processed by the FFT. We also set a flag to say that the next line should now be rendered and always reset the index to 0 to start filling the fifo again.
* [9] : Every time this function gets called, a sample is stored in the fifo and the index is incremented.

The fifo data now occupies the first half of the FFT input array and is ready to be processed and displayed.

**Displaying the Spectrogram**

In the drawNextLineOfSpectrogram() function, insert the pixel drawing implementation as explained below:

void drawNextLineOfSpectrogram()

{

auto rightHandEdge = spectrogramImage.getWidth() - 1;

auto imageHeight = spectrogramImage.getHeight();

// first, shuffle our image leftwards by 1 pixel..

spectrogramImage.moveImageSection (0, 0, 1, 0, rightHandEdge, imageHeight); // [1]

// then render our FFT data..

forwardFFT.performFrequencyOnlyForwardTransform (fftData.data()); // [2]

// find the range of values produced, so we can scale our rendering to

// show up the detail clearly

auto maxLevel = juce::FloatVectorOperations::findMinAndMax (fftData.data(), fftSize / 2); // [3]

for (auto [y](https://docs.juce.com/master/group__juce__audio__plugin__client-Unity.html#gae9ace15db486f33243a35ce689893fd9) = 1; [y](https://docs.juce.com/master/group__juce__audio__plugin__client-Unity.html#gae9ace15db486f33243a35ce689893fd9) < imageHeight; ++[y](https://docs.juce.com/master/group__juce__audio__plugin__client-Unity.html#gae9ace15db486f33243a35ce689893fd9)) // [4]

{

auto skewedProportionY = 1.0f - std::exp (std::log ((float) [y](https://docs.juce.com/master/group__juce__audio__plugin__client-Unity.html#gae9ace15db486f33243a35ce689893fd9) / (float) imageHeight) \* 0.2f);

auto fftDataIndex = (size\_t) juce::jlimit (0, fftSize / 2, (int) (skewedProportionY \* fftSize / 2));

auto level = juce::jmap (fftData[fftDataIndex], 0.0f, juce::jmax(maxLevel.getEnd(), 1e-5f), 0.0f, 1.0f);

spectrogramImage.setPixelAt (rightHandEdge, [y](https://docs.juce.com/master/group__juce__audio__plugin__client-Unity.html#gae9ace15db486f33243a35ce689893fd9), juce::Colour::fromHSV (level, 1.0f, level, 1.0f)); // [5]

}

}

[1] : First, shuffle the image leftwards by 1 pixel using the moveImageSection() function on the [**Image**](https://docs.juce.com/master/classImage.html) object. Specify the image section as the whole width minus one pixel and the whole height.

[2] : Then, render the FFT data using the performFrequencyOnlyForwardTransform() function on the FFT object with the fftData array as an argument.

[3] : Find the range of values produced, so that we can scale our rendering to show up the detail clearly. We can do so using the FloatVectorOperations::findMinAndMax() function.

[4] : Now in the for loop for every pixel in the spectrogram height, calculate the level proportionally to the sample set. To do this, we first need to skew the y-axis to use a logarithmic scale to better represent our frequencies. We can then feed this scaling factor to retrieve the correct array index and use the amplitude value to map it to a range between *0.0 .. 1.0* .

[5] : Finally set the appropriate pixel with the correct colour to display the FFT data.

As a final step, update the spectrogram using the timer callback function by calling the drawNextLineOfSpectrogram() only when the next FFT block is ready, reset the flag and update the GUI using the repaint() function:

void timerCallback() override

{

if (nextFFTBlockReady)

{

drawNextLineOfSpectrogram();

nextFFTBlockReady = false;

repaint();

}

}

**Exercise:** Try to increase the resolution of the FFT and change the rate at which the spectrogram updates.