demo project presents an on-screen keyboard that can be used to play a simple sine wave synthesiser. on-screen keyboard can be controlled using keys A, S, D, F and so on to control musical notes C, D, E, F and so on. This allows you to play the synthesiser polyphonically.

A screenshot of a keyboard

Description automatically generated

**The application window containing a MidiKeyboardComponent**

# The Synthesiser class

JUCE [**Synthesiser**](https://docs.juce.com/master/classSynthesiser.html) class implements polyphonic synthesiser. Various classes needed to get this to work:

* SynthAudioSource: implements custom **[AudioSource](https://docs.juce.com/master/classAudioSource.html" \o "Base class for objects that can produce a continuous stream of audio.)** class called SynthAudioSource, which contains the **[Synthesiser](https://docs.juce.com/master/classSynthesiser.html" \o "Base class for a musical device that can play sounds.)** class itself. This outputs all of the audio from the synthesiser.
* SineWaveVoice: This is a custom **[SynthesiserVoice](https://docs.juce.com/master/classSynthesiserVoice.html" \o "Represents a voice that a Synthesiser can use to play a SynthesiserSound.)** class called SineWaveVoice. A voice class renders one of the voices of the synthesiser mixing it with the other sounding voices in a **[Synthesiser](https://docs.juce.com/master/classSynthesiser.html" \o "Base class for a musical device that can play sounds.)** object. A single instance of a voice class renders one voice.
* SineWaveSound: contains custom **[SynthesiserSound](https://docs.juce.com/master/classSynthesiserSound.html" \o "Describes one of the sounds that a Synthesiser can play.)** class called SineWaveSound. Sound class is description of the sound that can be created as a voice. For example, this may contain sample data for a sampler voice or wavetable data for a wavetable synthesiser.

## Setting up the keyboard

Our MainContentComponent class contains the following data members.

juce::MidiKeyboardState keyboardState;

SynthAudioSource synthAudioSource;

juce::MidiKeyboardComponent keyboardComponent;

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

};

The synthAudioSource and keyboardComponent members are initialised in the MainContentComponent constructor.

MainContentComponent()

: synthAudioSource (keyboardState),

keyboardComponent (keyboardState, [juce](https://docs.juce.com/master/namespacejuce.html)::[MidiKeyboardComponent](https://docs.juce.com/master/classMidiKeyboardComponent.html)::horizontalKeyboard)

{

addAndMakeVisible (keyboardComponent);

setAudioChannels (0, 2);

setSize (600, 160);

startTimer (400);

}

See [**Tutorial: Handling MIDI events**](https://docs.juce.com/master/tutorial_handling_midi_events.html) for more information on the **[MidiKeyboardComponent](https://docs.juce.com/master/classMidiKeyboardComponent.html" \o "A component that displays a piano keyboard, whose notes can be clicked on.)** class.

In order that we can start playing the keyboard from the computer's keyboard we grab the keyboard focus just after the application starts. To do this we use a simple timer that fires after 400 ms:

void [timerCallback](https://docs.juce.com/master/group__juce__gui__basics-detail.html#gac56f546a808974ae71f0aeb42c7fc847)() override

{

keyboardComponent.grabKeyboardFocus();

stopTimer();

}

## AudioAppComponent functions

The application uses the **[AudioAppComponent](https://docs.juce.com/master/classAudioAppComponent.html" \o "A base class for writing audio apps that stream from the audio i/o devices.)** to set up a simple audio application (see [**Tutorial: Build a white noise generator**](https://docs.juce.com/master/tutorial_simple_synth_noise.html) for the most basic application). The three required [pure virtual](http://www.learncpp.com/cpp-tutorial/126-pure-virtual-functions-abstract-base-classes-and-interface-classes/) functions simply call the corresponding functions in our custom **[AudioSource](https://docs.juce.com/master/classAudioSource.html" \o "Base class for objects that can produce a continuous stream of audio.)** class:

void prepareToPlay (int samplesPerBlockExpected, double sampleRate) override

{

synthAudioSource.prepareToPlay (samplesPerBlockExpected, sampleRate);

}

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

{

synthAudioSource.getNextAudioBlock (bufferToFill);

}

void releaseResources() override

{

synthAudioSource.releaseResources();

}

## The SynthAudioSource class

The SynthAudioSource class does a little more work:

class SynthAudioSource : public juce::AudioSource

{

public:

SynthAudioSource (juce::MidiKeyboardState& keyState)

: keyboardState (keyState)

{

for (auto i = 0; i < 4; ++i) // [1]

synth.addVoice (new SineWaveVoice());

synth.addSound (new SineWaveSound()); // [2]

}

void setUsingSineWaveSound()

{

synth.clearSounds();

}

void prepareToPlay (int /\*samplesPerBlockExpected\*/, double sampleRate) override

{

synth.setCurrentPlaybackSampleRate (sampleRate); // [3]

}

void releaseResources() override {}

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

{

bufferToFill.clearActiveBufferRegion();

juce::MidiBuffer incomingMidi;

keyboardState.processNextMidiBuffer (incomingMidi, bufferToFill.startSample,

bufferToFill.numSamples, true); // [4]

synth.renderNextBlock (\*bufferToFill.buffer, incomingMidi,

bufferToFill.startSample, bufferToFill.numSamples); // [5]

}

private:

juce::MidiKeyboardState& keyboardState;

juce::Synthesiser synth;

};

1. We add voices to our synthesiser. number of voices added determines polyphony of synthesiser.
2. We add the sound so that the synthesiser knows which sounds it can play.
3. The synthesiser needs to know the sample rate of the audio output.
4. In getNextAudioBlock() we pull buffers of MIDI data from the **[MidiKeyboardState](https://docs.juce.com/master/classMidiKeyboardState.html" \o "Represents a piano keyboard, keeping track of which keys are currently pressed.)** object.
5. These buffers of MIDI are passed to the synthesiser which will be used to render the voices using the timestamps of the note-on and note-off messages (and other MIDI channel voice messages).

[**SynthesiserVoice**](https://docs.juce.com/master/classSynthesiserVoice.html) objects must be added to one and only one **[Synthesiser](https://docs.juce.com/master/classSynthesiser.html" \o "Base class for a musical device that can play sounds.)** object. The **[Synthesiser](https://docs.juce.com/master/classSynthesiser.html" \o "Base class for a musical device that can play sounds.)** object manages the lifetime of the voices.

[**SynthesiserSound**](https://docs.juce.com/master/classSynthesiserSound.html) objects can be shared between [**Synthesiser**](https://docs.juce.com/master/classSynthesiser.html) objects if you wish. The **[SynthesiserSound](https://docs.juce.com/master/classSynthesiserSound.html" \o "Describes one of the sounds that a Synthesiser can play.)** class is a type of **[ReferenceCountedObject](https://docs.juce.com/master/classReferenceCountedObject.html" \o "A base class which provides methods for reference-counting.)** class therefore the lifetime of **[SynthesiserSound](https://docs.juce.com/master/classSynthesiserSound.html" \o "Describes one of the sounds that a Synthesiser can play.)** objects is handled automatically.

**Note**

If you need to keep a pointer to a **[SynthesiserSound](https://docs.juce.com/master/classSynthesiserSound.html" \o "Describes one of the sounds that a Synthesiser can play.)** object you should store it in a YourSoundClass::Ptr variable for this memory management to work.

## The SineWaveSound class

Our sound class is very simple, it doesn't even need to contain any data. It just needs to report whether this sound should play on particular MIDI channels and specific notes or note ranges on that channel. In our simple case, it just returns true for both the appliesToNote() and appliesToChannel() functions. As mentioned above, the sound class might be where you would store data that is needed to create the sound (such as a wavetable).

struct SineWaveSound : public juce::SynthesiserSound

{

SineWaveSound() {}

bool appliesToNote (int) override { return true; }

bool appliesToChannel (int) override { return true; }

};

## Sine wave voice state

The SineWaveVoice class is a bit more complex. It needs to maintain the state of one of the voices of the synthesiser. For our sine wave, we need these data members:

private:

double currentAngle = 0.0, angleDelta = 0.0, level = 0.0, tailOff = 0.0;

};

See [**Tutorial: Build a sine wave synthesiser**](https://docs.juce.com/master/tutorial_sine_synth.html) for information on the first three. The tailOff member is used to give each voice a slightly softer release to its amplitude envelope. This gives each voice a slight fade out at the end rather than stopping abruptly.

A graph on a black background

Description automatically generated

**Exponential release envelope**

## Checking which sound can be played

The **[SynthesiserVoice::canPlaySound()](https://docs.juce.com/master/classSynthesiserVoice.html" \l "aa9fb097ca96dc39fae8e6f7b9139df9f" \o "Must return true if this voice object is capable of playing the given sound.)** function must be overriden to return whether the voice can play a sound. We could just return true in this case but our example illustrates how to use dynamic\_cast to check the type of the sound class being passed in.

bool canPlaySound (juce::SynthesiserSound\* sound) override

{

return dynamic\_cast<SineWaveSound\*> (sound) != nullptr;

}

## Starting a voice

A voice is started by the owning synthesiser by calling our **[SynthesiserVoice::startNote()](https://docs.juce.com/master/classSynthesiserVoice.html" \l "af3b872622dd9bb540030420189175762" \o "Called to start a new note.)** function, which we must override:

void startNote (int midiNoteNumber, float velocity,

juce::SynthesiserSound\*, int /\*currentPitchWheelPosition\*/) override

{

currentAngle = 0.0;

level = velocity \* 0.15;

tailOff = 0.0;

auto cyclesPerSecond = juce::MidiMessage::getMidiNoteInHertz (midiNoteNumber);

auto cyclesPerSample = cyclesPerSecond / getSampleRate();

angleDelta = cyclesPerSample \* 2.0 \* juce::MathConstants<double>::pi;

}

Again, most of this should be familar to your from [**Tutorial: Build a sine wave synthesiser**](https://docs.juce.com/master/tutorial_sine_synth.html). The tailOff value is set to zero at the start of each voice. We also use the velocity of the MIDI note-on event to control the level of the voice.

## Rendering a voice

[**SynthesiserVoice::renderNextBlock()**](https://docs.juce.com/master/classSynthesiserVoice.html#a72ab7856c1e7651b1ce955388645a0a1) must be overriden to generate the audio.

void renderNextBlock (juce::AudioSampleBuffer& outputBuffer, int startSample, int numSamples) override {

if (angleDelta != 0.0) {

if (tailOff > 0.0) { // [7]

while (--numSamples >= 0) {

auto currentSample = (float) (std::sin (currentAngle) \* level \* tailOff);

for (auto i = outputBuffer.getNumChannels(); --i >= 0;)

outputBuffer.addSample (i, startSample, currentSample);

currentAngle += angleDelta;

++startSample;

tailOff \*= 0.99; // [8]

if (tailOff <= 0.005) {

clearCurrentNote(); // [9]

angleDelta = 0.0;

break;

}

}

} else {

while (--numSamples >= 0) { // [6]

auto currentSample = (float) (std::sin (currentAngle) \* [level](https://docs.juce.com/master/namespacegl.html#a3037cfec75442657d7a3f8830cb97c34));

for (auto i = outputBuffer.getNumChannels(); --i >= 0;)

outputBuffer.addSample (i, startSample, currentSample);

currentAngle += angleDelta;

++startSample;

}

}

}

}

1. Loop used for normal state of voice while key held down. Use AudioSampleBuffer::addSample(), which mixes currentSample  with value already at index startSample, since synthesiser will iterate over all voices. It is responsibility of each voice to mix its output with current contents of buffer.
2. When the key has been released the tailOff value will be greater than zero. You can see the synthesis algorithm is similar.
3. We use a simple exponential decay envelope shape.
4. Voice ends when tailOff small. Must call [**SynthesiserVoice::clearCurrentNote()**](https://docs.juce.com/master/classSynthesiserVoice.html#a55d1e7a062f8dc9a6cd78526b0da049c) function at this point so voice is reset and available to be reused.

Note startSample argument. Synthesiser may call renderNextBlock() mid-way through output block, because notes may start on any sample. These start times are based on timestamps of MIDI data received.

## Stopping a voice

A voice is stopped by the owning synthersiser calling our **[SynthesiserVoice::stopNote()](https://docs.juce.com/master/classSynthesiserVoice.html" \l "ae1faf9289de24cdeb1d42289754601e6" \o "Called to stop a note.)** function, which we must override:

void stopNote (float /\*velocity\*/, bool allowTailOff) override {

if (allowTailOff) {

if (tailOff == 0.0) tailOff = 1.0;

} Else {

clearCurrentNote();

angleDelta = 0.0;

}

}

This may include velocity information from the MIDI note-off message, but in many cases we can ignore this. We may be being asked to stop the voice immediately in which case we call the the **[SynthesiserVoice::clearCurrentNote()](https://docs.juce.com/master/classSynthesiserVoice.html" \l "a55d1e7a062f8dc9a6cd78526b0da049c" \o "Resets the state of this voice after a sound has finished playing.)** function straight away. Under normal circumstances the synthesiser will allow our voices to end naturally. In our case we have the simple tail-off envelope. We trigger our tail-off by setting our tailOff member to 1.0.

Try adding a slower attack to the voices such that they don't start abruptly.

# Adding external MIDI input

Let's allow external MIDI source to control our synthesiser in addition to the on-screen keyboard.

## Providing a MIDI input to the SynthAudioSource

Add **[MidiMessageCollector](https://docs.juce.com/master/classMidiMessageCollector.html" \o "Collects incoming realtime MIDI messages and turns them into blocks suitable for processing by a bloc...)** object as member of SynthAudioSource class. This provides somewhere that MIDI messages can be sent and that the SynthAudioSource class can use them:

juce::MidiMessageCollector midiCollector;

};

In order to process the timestamps of the MIDI data the **[MidiMessageCollector](https://docs.juce.com/master/classMidiMessageCollector.html" \o "Collects incoming realtime MIDI messages and turns them into blocks suitable for processing by a bloc...)** class needs to know the audio sample rate. Set this in the SynthAudioSource::prepareToPlay() function [10]:

void prepareToPlay (int /\*samplesPerBlockExpected\*/, double sampleRate) override

{

synth.setCurrentPlaybackSampleRate (sampleRate);

midiCollector.reset (sampleRate); // [10]

}

Then you can pull any MIDI messages for each block of audio using the [**MidiMessageCollector::removeNextBlockOfMessages()**](https://docs.juce.com/master/classMidiMessageCollector.html#ac72b6cf4965e63b90d1a2402b73b1798) function [11]:

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

{

bufferToFill.clearActiveBufferRegion();

juce::MidiBuffer incomingMidi;

midiCollector.removeNextBlockOfMessages (incomingMidi, bufferToFill.numSamples); // [11]

keyboardState.processNextMidiBuffer (incomingMidi, bufferToFill.startSample,

bufferToFill.numSamples, true);

synth.renderNextBlock (\*bufferToFill.buffer, incomingMidi,

bufferToFill.startSample, bufferToFill.numSamples);

}

need access to this **[MidiMessageCollector](https://docs.juce.com/master/classMidiMessageCollector.html" \o "Collects incoming realtime MIDI messages and turns them into blocks suitable for processing by a bloc...)** object from outside SynthAudioSource class, so add an accessor to SynthAudioSource class like this:

juce::MidiMessageCollector\* getMidiCollector() { return &midiCollector; }

Add this **[MidiMessageCollector](https://docs.juce.com/master/classMidiMessageCollector.html" \o "Collects incoming realtime MIDI messages and turns them into blocks suitable for processing by a bloc...)** object as [**MidiInputCallback**](https://docs.juce.com/master/classMidiInputCallback.html) object In MainContentComponent class to application's **[AudioDeviceManager](https://docs.juce.com/master/classAudioDeviceManager.html" \o "Manages the state of some audio and midi i/o devices.)** object.

## Listing MIDI inputs

To present list of MIDI input devices to user, we'll use some code from [**Tutorial: Handling MIDI events**](https://docs.juce.com/master/tutorial_handling_midi_events.html). Add some members to our MainContentComponent class:

juce::ComboBox midiInputList;

juce::Label midiInputListLabel;

int lastInputIndex = 0;

Then add the following code to the MainContentComponent constructor.

addAndMakeVisible (midiInputListLabel);

midiInputListLabel.setText ("MIDI Input:", juce::dontSendNotification);

midiInputListLabel.attachToComponent (&midiInputList, true);

auto midiInputs = juce::MidiInput::getAvailableDevices();

addAndMakeVisible (midiInputList);

midiInputList.setTextWhenNoChoicesAvailable ("No MIDI Inputs Enabled");

juce::StringArray midiInputNames;

for (auto input : midiInputs) midiInputNames.add (input.name);

midiInputList.addItemList (midiInputNames, 1);

midiInputList.onChange= [this] { setMidiInput(midiInputList.getSelectedItemIndex()); };

for (auto input : midiInputs) {

if (deviceManager.isMidiInputDeviceEnabled (input.identifier)) {

setMidiInput (midiInputs.indexOf (input));

break;

}

}

if (midiInputList.getSelectedId() == 0)

setMidiInput (0);

Add the setMidiInput() function that is called in the code above:

void setMidiInput (int [index](https://docs.juce.com/master/group__juce__gui__basics-detail.html#ga32b76ccceab9e1bba3e8c4ef2947d1d8))

{

auto [list](https://docs.juce.com/master/group__juce__gui__basics-accessibility.html#ggaf1349587ab6cac7e882cd63e1976682ca10ae9fc7d453b0dd525d0edf2ede7961) = juce::MidiInput::getAvailableDevices();

deviceManager.removeMidiInputDeviceCallback ([list](https://docs.juce.com/master/group__juce__gui__basics-accessibility.html#ggaf1349587ab6cac7e882cd63e1976682ca10ae9fc7d453b0dd525d0edf2ede7961)[lastInputIndex].identifier,

synthAudioSource.getMidiCollector()); // [12]

auto newInput = [list](https://docs.juce.com/master/group__juce__gui__basics-accessibility.html#ggaf1349587ab6cac7e882cd63e1976682ca10ae9fc7d453b0dd525d0edf2ede7961)[[index](https://docs.juce.com/master/group__juce__gui__basics-detail.html#ga32b76ccceab9e1bba3e8c4ef2947d1d8)];

if (! deviceManager.isMidiInputDeviceEnabled (newInput.identifier))

deviceManager.setMidiInputDeviceEnabled (newInput.identifier, true);

deviceManager.addMidiInputDeviceCallback(newInput.identifier, synthAudioSource.getMidiCollector()); // [13]

midiInputList.setSelectedId ([index](https://docs.juce.com/master/group__juce__gui__basics-detail.html#ga32b76ccceab9e1bba3e8c4ef2947d1d8) + 1, juce::dontSendNotification);

lastInputIndex = [index](https://docs.juce.com/master/group__juce__gui__basics-detail.html#ga32b76ccceab9e1bba3e8c4ef2947d1d8);

}

We add **[MidiMessageCollector](https://docs.juce.com/master/classMidiMessageCollector.html" \o "Collects incoming realtime MIDI messages and turns them into blocks suitable for processing by a bloc...)** object from our SynthAudioSource object as a **[MidiInputCallback](https://docs.juce.com/master/classMidiInputCallback.html" \o "Receives incoming messages from a physical MIDI input device.)** object [13] for specified MIDI input device. We also need to remove the previous [**MidiInputCallback**](https://docs.juce.com/master/classMidiInputCallback.html) object for previously selected MIDI input device if the user changes the selected device using the combo-box [12].

We need to position this **[ComboBox](https://docs.juce.com/master/classComboBox.html" \o "A component that lets the user choose from a drop-down list of choices.)** object and adjust the position of the **[MidiKeyboardComponent](https://docs.juce.com/master/classMidiKeyboardComponent.html" \o "A component that displays a piano keyboard, whose notes can be clicked on.)** object in our resized() function:

void resized() override

{

midiInputList .setBounds (200, 10, getWidth() - 210, 20);

keyboardComponent.setBounds (10, 40, getWidth() - 20, getHeight() - 50);

}

Run the application again and it should look something like this:

A screenshot of a keyboard

Description automatically generated

**The application window showing the MIDI input device list**

Of course, the devices listed will depend on your specific system configuration.

The source code for this modified version of the application can be found in the SynthUsingMidiInputTutorial\_02.h file of the demo project.

Try adding a slider to control the length of the tail-off for each voice. You could also add a slider to control the length of the attack, if you added this in the earlier exercise.