Project Report

Basic Sound Synthesiser Application

Object-Oriented Programming Concepts and Programming

13016209

Arut Jinadit 60090002

Software Engineering , International College

King Mongut's Institute of Technology Ladkrabang

13016209 Object-oriented Concepts and Programming

Second semester, 2017

Project Assignment – Proposal

Project developer

Student ID : 60090002

Name : Arut Jinadit

Project title

Basic Sound Synthesiser Application

Description :

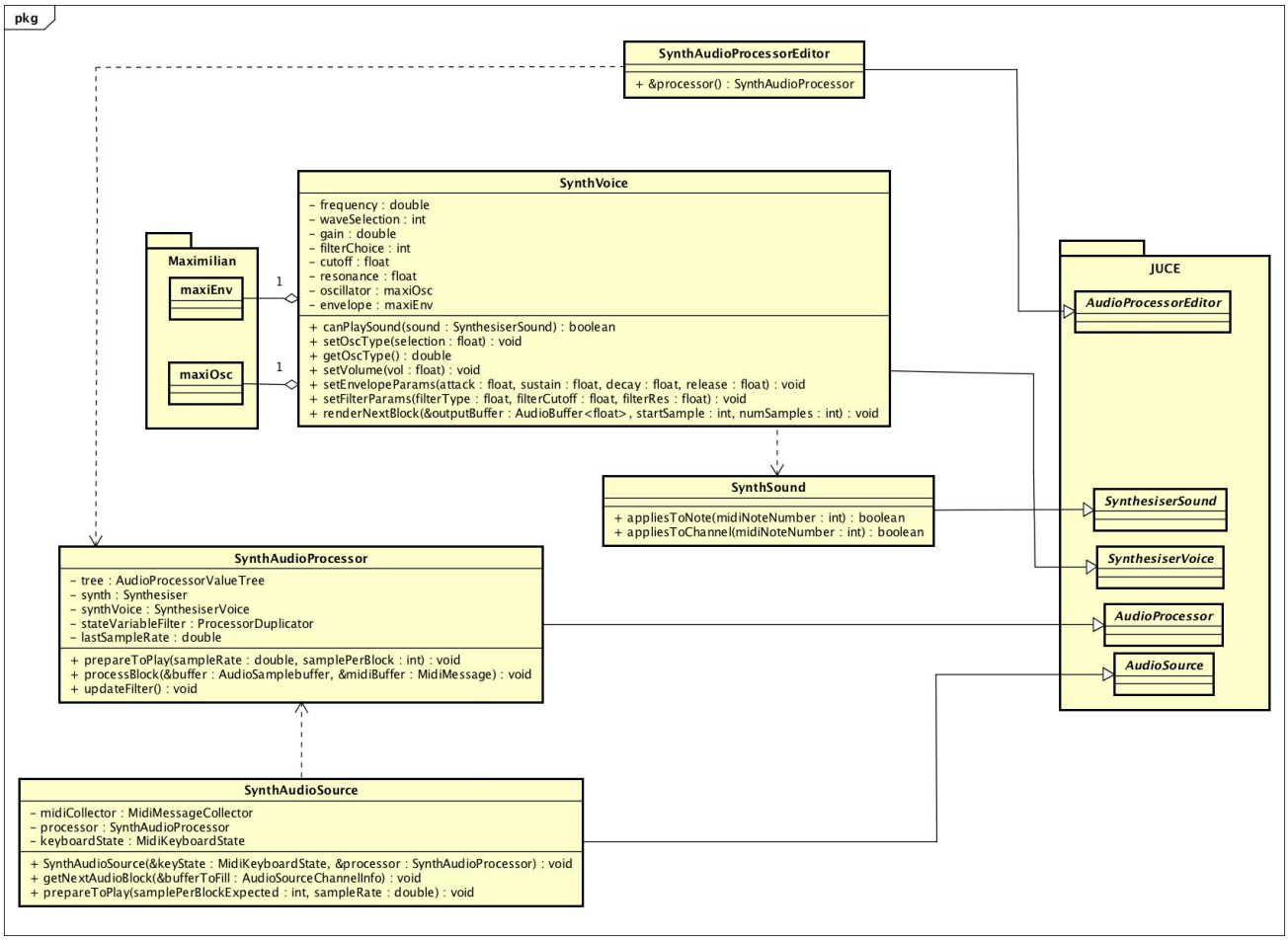
In music production, nowadays, sound engineers and artists are using an instrument that can generate sound, which its characteristics can be modified by using Signal Processing process, so called “Synthesiser”. However, these instruments are expensive. For beginners, who newly entre this industry, to afford these instrument would be a heavy-weighted investment. Therefore, this application would help in making a synthesized sound with a cheaper way.

Basic Sound Synthesiser Application application is an application that provides users an ability to generate and synthesise sound with basic synthesizer operation. Users are provided with options to use a virtual keyboard on the application or use a plugged midi controlling instrument.

Requirements

- JUCE Library (GUI and audio management)
- Maximilian Library (Sound Manipulation)
- This application needed to be run on mac OS.

Class Diagram (containing only important attributes and methods)



Main.cpp

```
//Main.cpp

#include "../JuceLibraryCode/JuceHeader.h"
#include "MainComponent.h"

class oopSynthApplication  : public JUCEApplication
{
public:
    oopSynthApplication() {}

    const String getApplicationName() override       { return
ProjectInfo::projectName; }
    const String getApplicationVersion() override    { return
ProjectInfo::versionString; }
    bool moreThanOneInstanceAllowed() override       { return true; }

    void initialise (const String& commandLine) override
    {
        //Initialise application window
        mainWindow = new MainWindow (getApplicationName());
    }

    void shutdown() override { mainWindow = nullptr; }

    void systemRequestedQuit() override { quit(); }

    void anotherInstanceStarted (const String& commandLine) override {}


    //MainWindow Application
    class MainWindow    : public DocumentWindow
    {
    public:
        MainWindow (String name)  : DocumentWindow (name,
                                                    Desktop::getInstance()
.getDefaultLookAndFeel()

.findColour (ResizableWindow::backgroundColourId),
                                                    5)
        {
            setUsingNativeTitleBar (true);
            setContentOwned (new MainComponent(), true);
            setResizable (false, false);
            centreWithSize (getWidth(), getHeight());
            setVisible (true);
        }

        void closeButtonPressed() override
        {
            JUCEApplication::getInstance()->systemRequestedQuit();
        }
    private:
        JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (MainWindow)
    };

private:
```

```
    ScopedPointer<MainWindow> mainWindow;
};

// This macro generates the main() routine that launches the app.
START_JUCE_APPLICATION (oopSynthApplication)
```

MainComponent.h

```
//MainComponent.h

#pragma once

#include "maximilian.h"
#include "../JuceLibraryCode/JuceHeader.h"
#include "PluginProcessor.h"
#include "PluginEditor.h"
#include "SynthAudioSource.h"

class MainComponent  : public AudioAppComponent
{
public:
    //==========================================================================
==========
    MainComponent();
    ~MainComponent();
    //==========================================================================
==========
    void resized() override;
    void prepareToPlay(int samplesPerBlockExpected,double sampleRate)
override;
    void getNextAudioBlock (const AudioSourceChannelInfo& bufferToFill)
override;
    void releaseResources() override;
    void paint(Graphics& g) override;
private:
    AudioDeviceManager audioDeviceManager;
    MidiKeyboardState keyboardState;
    AudioSourcePlayer audioSourcePlayer;
    SynthAudioSource synthAudioSource { keyboardState,processor };
    MidiKeyboardComponent keyboardComponent { keyboardState,
MidiKeyboardComponent::horizontalKeyboard};

    ComboBox midiInputList;
    Label midiInputListLabel;
    SynthAudioProcessor processor;
    SynthAudioProcessorEditor processorEditor{processor};
    int lastMidiInputIndex = 0;
    void setMidiInput (int index);
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (MainComponent)
};
```

MainComponent.cpp

```
//MainComponent.cpp
#include "MainComponent.h"
//==============================================================================
MainComponent::MainComponent() :
synthAudioSource  (keyboardState,processor),
                                keyboardComponent (keyboardState,
MidiKeyboardComponent::horizontalKeyboard)
{
    addAndMakeVisible(midiInputListLabel);
    midiInputListLabel.setText("MIDI Input;", dontSendNotification);
    midiInputListLabel.attachToComponent(&midiInputList, true);

    addAndMakeVisible(midiInputList);
    midiInputList.setTextWhenNoChoicesAvailable("No MIDI Inputs Enabled");
    auto midiInputs = MidiInput::getDevices();
    midiInputList.addItemList(midiInputs, 1);
    midiInputList.onChange = [this] { setMidiInput
(midiInputList.getSelectedItemIndex());};
    for (auto midiInput : midiInputs)
    {
        if (deviceManager.isMidiInputEnabled (midiInput))
        {
            setMidiInput (midiInputs.indexOf (midiInput));
            break;
        }
    }
    if (midiInputList.getSelectedId() == 0) setMidiInput (0);

    addAndMakeVisible(keyboardComponent);

    setAudioChannels (0, 2);
    addAndMakeVisible(processorEditor);
    setSize (800, 450);
}

MainComponent::~MainComponent()
{
    shutdownAudio();
}

//==============================================================================prepare
void MainComponent::prepareToPlay (int samplesPerBlockExpected, double
sampleRate)
{
    synthAudioSource.prepareToPlay(samplesPerBlockExpected, sampleRate);
}

void MainComponent::getNextAudioBlock (const AudioSourceChannelInfo&
bufferToFill)
{
    synthAudioSource.getNextAudioBlock(bufferToFill);
}

void MainComponent::releaseResources()
```

```
{
    synthAudioSource.releaseResources();
}

//==============================================================================
void MainComponent::paint (Graphics& g)
{
    g.fillAll (getLookAndFeel().findColour
(ResizableWindow::backgroundColourId));

}

void MainComponent::resized()
{
    Rectangle<int> screenArea = getLocalBounds();
    auto lowerArea = screenArea.removeFromBottom(195);
    auto upperArea = screenArea.removeFromTop(800 - 195);
    auto keyboardArea = lowerArea.removeFromBottom(130);
    auto midiInputArea = lowerArea.removeFromTop(65);
    midiInputArea = midiInputArea.removeFromTop(45);
    midiInputArea = midiInputArea.removeFromBottom(40);
    midiInputArea = midiInputArea.removeFromRight(790);
    midiInputArea = midiInputArea.removeFromLeft(780);
    midiInputList.setBounds(midiInputArea);
    keyboardComponent.setBounds(keyboardArea);
    processorEditor.setBounds(upperArea);
}

void MainComponent::setMidiInput(int index) {
    auto list = MidiInput::getDevices();
    deviceManager.removeMidiInputCallback (list[lastMidiInputIndex],
synthAudioSource.getMidiCollector());
    auto newMidiInput = list[index];

    if (! deviceManager.isMidiInputEnabled (newMidiInput))
        deviceManager.setMidiInputEnabled (newMidiInput, true);

    deviceManager.addMidiInputCallback (newMidiInput,
synthAudioSource.getMidiCollector());
    midiInputList.setSelectedId (index + 1, dontSendNotification);
    lastMidiInputIndex = index;
}
```

PlugInEditor.h

```
//PlugInEditor.h
#pragma once

#include "../JuceLibraryCode/JuceHeader.h"
#include "PluginProcessor.h"
#include "Oscillator.h"
#include "Envelope.h"
#include "Filter.h"
#include "Volume.h"

class SynthAudioProcessorEditor  : public AudioProcessorEditor

{
public:
    SynthAudioProcessorEditor (SynthAudioProcessor&);
    ~SynthAudioProcessorEditor();

    void paint (Graphics&) override;
    void resized() override;

private:
    SynthAudioProcessor& processor;
    Oscillator oscGui;
    Envelope envGui;
    Filter filterGui;
    Volume volumeControlGui;
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR
(SynthAudioProcessorEditor)
};
```

PlugInEditor.cpp

```
#include "PluginProcessor.h"
#include "PluginEditor.h"

SynthAudioProcessorEditor::SynthAudioProcessorEditor (SynthAudioProcessor&
p)
    : AudioProcessorEditor (&p), processor (p), oscGui(p), envGui(p),
filterGui(p),volumeControlGui(p)
{
    setSize (800,800 - 195);

    addAndMakeVisible(&oscGui);
    addAndMakeVisible(&envGui);
    addAndMakeVisible(&filterGui);
    addAndMakeVisible(&volumeControlGui);
}

SynthAudioProcessorEditor::~SynthAudioProcessorEditor()
{

}
```

```
void SynthAudioProcessorEditor::paint (Graphics& g)
{
    g.fillAll (getLookAndFeel().findColour
(ResizableWindow::backgroundColourId));
}

void SynthAudioProcessorEditor::resized()
{
    Rectangle<int> area = getLocalBounds();
    auto leftSide = area.removeFromLeft(120 + 350).removeFromRight(350);
    auto rightSide = area.removeFromRight(800 - 120 -
350).removeFromLeft(220);

    oscGui.setBounds(rightSide.removeFromTop(255 - 170));
    filterGui.setBounds(leftSide.removeFromTop(130));
    envGui.setBounds(leftSide.removeFromBottom(125));
    volumeControlGui.setBounds(rightSide.removeFromBottom(170));
}
```

PlugInProcessor.h

```
#pragma once

#include "../JuceLibraryCode/JuceHeader.h"
#include "SynthVoice.h"
#include "SynthSound.h"

class SynthAudioProcessor : public AudioProcessor
{
public:
    SynthAudioProcessor();
    ~SynthAudioProcessor();

    void prepareToPlay (double sampleRate, int samplesPerBlock) override;
    void releaseResources() override {}

   #ifndef JucePlugin_PreferredChannelConfigurations
    bool isBusesLayoutSupported (const BusesLayout& layouts) const
override;
   #endif

    void processBlock (AudioSampleBuffer&, MidiBuffer&) override;

    AudioProcessorEditor* createEditor() override;
    bool hasEditor() const override {return true;}
    const String getName() const override { return JucePlugin_Name; }

    bool acceptsMidi() const override;
    bool producesMidi() const override;
    bool isMidiEffect () const override;
    double getTailLengthSeconds() const override {return 0.0;}

    int getNumPrograms() override {return 1;}
    int getCurrentProgram() override {return 0;}
    void setCurrentProgram (int index) override {}
    const String getProgramName (int index) override {return {};}
    void changeProgramName (int index, const String& newName) override {}

    void getStateInformation (MemoryBlock& destData) override {}
```

```
    void setStateInformation (const void* data, int sizeInBytes) override
{}

    void updateFilter();

    AudioProcessorValueTreeState tree;

private:
    Synthesiser synth;
    SynthVoice* synthVoice;
    dsp::ProcessorDuplicator<dsp::StateVariableFilter::Filter<float> ,
dsp::StateVariableFilter::Parameters<float>> stateVariableFilter;
    double lastSampleRate;

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (SynthAudioProcessor)
};
```

PluginProcessor.cpp

```
#include "PluginProcessor.h"
#include "PluginEditor.h"

SynthAudioProcessor::SynthAudioProcessor()
#ifndef JucePlugin_PreferredChannelConfigurations
     : AudioProcessor (BusesProperties()
                     #if ! JucePlugin_IsMidiEffect
                      #if ! JucePlugin_IsSynth
                       .withInput  ("Input",  AudioChannelSet::stereo(),
true)
                      #endif
                       .withOutput ("Output", AudioChannelSet::stereo(),
true)
                     #endif
                       ),
tree(*this, nullptr)
#endif
{
    //Set range for envelope params
    NormalisableRange<float> attackParam (0.1f, 5000.0f);
    NormalisableRange<float> decayParam (1.0f, 2000.0f);
    NormalisableRange<float> sustainParam (0.0f, 1.0f);
    NormalisableRange<float> releaseParam (0.1f, 5000.0f);

    //Add envelope params to AudioProcessorValueTree
    tree.createAndAddParameter("attack", "Attack", "attack", attackParam,
0.1f, nullptr, nullptr);
    tree.createAndAddParameter("decay", "Decay", "decay", decayParam,
1.0f, nullptr, nullptr);
    tree.createAndAddParameter("sustain", "Sustain", "sustain",
sustainParam, 0.8f, nullptr, nullptr);
    tree.createAndAddParameter("release", "Release", "release",
releaseParam, 0.1f, nullptr, nullptr);

    //Set range for oscillator params
    NormalisableRange<float> wavetypeParam (0, 2);

    //Add envelope params to AudioProcessorValueTree
    tree.createAndAddParameter("wavetype", "WaveType", "wavetype",
wavetypeParam, 0, nullptr, nullptr);
```

```
    //Set range for filter params
    NormalisableRange<float> filterTypeVal (0, 2);
    NormalisableRange<float> filterVal (20.0f, 10000.0f);
    NormalisableRange<float> resVal (1, 5);

    //Add filter params to AudioProcessorValueTree
    tree.createAndAddParameter("filterType", "FilterType", "filterType",
filterTypeVal, 0, nullptr, nullptr);
    tree.createAndAddParameter("filterCutoff", "FilterCutoff",
"filterCutoff", filterVal, 400.0f, nullptr, nullptr);
    tree.createAndAddParameter("filterRes", "FilterRes", "filterRes",
resVal, 1, nullptr, nullptr);

    //Set range for volume params
    NormalisableRange<float> volumeParam(0,10);

    //Add volume params to AudioProcessorValueTree
    tree.createAndAddParameter("volume", "Volume", "volume", volumeParam,
5, nullptr, nullptr);

    //Clear and add voices to the synth
    synth.clearVoices();
    for (int i = 0; i < 5; i++)
    {
        synth.addVoice(new SynthVoice());
    }
    synth.clearSounds();
    synth.addSound(new SynthSound());
}

SynthAudioProcessor::~SynthAudioProcessor()
{

}

bool SynthAudioProcessor::acceptsMidi() const
{
   #if JucePlugin_WantsMidiInput
    return true;
   #else
    return false;
   #endif
}

bool SynthAudioProcessor::producesMidi() const
{
   #if JucePlugin_ProducesMidiOutput
    return true;
   #else
    return false;
   #endif
}

bool SynthAudioProcessor::isMidiEffect() const
{
   #if JucePlugin_IsMidiEffect
    return true;
   #else
    return false;
```

```
   #endif
}

void SynthAudioProcessor::prepareToPlay (double sampleRate, int
samplesPerBlock)
{
    //prepare processor settings before process a new audio block.
    ignoreUnused(samplesPerBlock);
    lastSampleRate = sampleRate;
    synth.setCurrentPlaybackSampleRate(lastSampleRate);

    dsp::ProcessSpec spec;
    spec.sampleRate = lastSampleRate;
    spec.maximumBlockSize = samplesPerBlock;
    spec.numChannels = getTotalNumOutputChannels();

    stateVariableFilter.reset();
    stateVariableFilter.prepare(spec);
    updateFilter();
}

#ifndef JucePlugin_PreferredChannelConfigurations
bool SynthAudioProcessor::isBusesLayoutSupported (const BusesLayout&
layouts) const
{
  #if JucePlugin_IsMidiEffect
    ignoreUnused (layouts);
    return true;
  #else
    // This is the place where you check if the layout is supported.
    // In this template code we only support mono or stereo.
    if (layouts.getMainOutputChannelSet() != AudioChannelSet::mono()
     && layouts.getMainOutputChannelSet() != AudioChannelSet::stereo())
        return false;

    // This checks if the input layout matches the output layout
   #if ! JucePlugin_IsSynth
    if (layouts.getMainOutputChannelSet() !=
layouts.getMainInputChannelSet())
        return false;
   #endif

    return true;
  #endif
}
#endif


void SynthAudioProcessor::updateFilter()
{
    int menuChoice = *tree.getRawParameterValue("filterType");
    int freq = *tree.getRawParameterValue("filterCutoff");
    int res = *tree.getRawParameterValue("filterRes");

    if (menuChoice == 0)
    {
        stateVariableFilter.state->type =
dsp::StateVariableFilter::Parameters<float>::Type::lowPass;
        stateVariableFilter.state->setCutOffFrequency(lastSampleRate,
freq, res);
```

```
    }

    if (menuChoice == 1)
    {
        stateVariableFilter.state->type =
dsp::StateVariableFilter::Parameters<float>::Type::highPass;
        stateVariableFilter.state->setCutOffFrequency(lastSampleRate,
freq, res);
    }

    if (menuChoice == 2)
    {
        stateVariableFilter.state->type =
dsp::StateVariableFilter::Parameters<float>::Type::bandPass;
        stateVariableFilter.state->setCutOffFrequency(lastSampleRate,
freq, res);
    }
}

void SynthAudioProcessor::processBlock (AudioSampleBuffer& buffer,
MidiBuffer& midiMessages)
{
    ScopedNoDenormals noDenormals;
    for (int i = 0; i < synth.getNumVoices(); i++)
    {
        //Modify all parameters with values which are gotten from GUI.
        if ((synthVoice = dynamic_cast<SynthVoice*>(synth.getVoice(i))))
        {
            synthVoice-
>getEnvelopeParams(tree.getRawParameterValue("attack"),
                                        tree.getRawParameterValue("decay"),
                                        tree.getRawParameterValue("sustain"
),
                                        tree.getRawParameterValue("release"
));

            synthVoice->setOscType(tree.getRawParameterValue("wavetype"));

            synthVoice-
>setFilterParams(tree.getRawParameterValue("filterType"),
                                       tree.getRawParameterValue("filterCuto
ff"),
                                        tree.getRawParameterValue("filterRes"
));

            synthVoice->setVolume(tree.getRawParameterValue("volume"));
        }
    }

    buffer.clear();
    synth.renderNextBlock(buffer, midiMessages, 0,
buffer.getNumSamples());
    updateFilter();
    dsp::AudioBlock<float> block (buffer);
    stateVariableFilter.process(dsp::ProcessContextReplacing<float>
(block));
}

AudioProcessorEditor* SynthAudioProcessor::createEditor()
{
```

```
    return new SynthAudioProcessorEditor (*this);
}

AudioProcessor* JUCE_CALLTYPE createPluginFilter()
{
    return new SynthAudioProcessor();
}
```

SynthAudioSource.h

```
#pragma once

#include "../JuceLibraryCode/JuceHeader.h"
#include "PluginProcessor.h"

class SynthAudioSource : public AudioSource {
public:
    SynthAudioSource(MidiKeyboardState& keyState,SynthAudioProcessor& p);
    void prepareToPlay(int samplePerBlockExpected,double sampleRate)
override;
    void releaseResources() override {};
    void getNextAudioBlock(const AudioSourceChannelInfo& bufferToFill)
override;
    MidiMessageCollector* getMidiCollector();
private:
    MidiMessageCollector midiCollector;
    MidiKeyboardState& keyboardState;
    Synthesiser synth;
    SynthAudioProcessor& processor;
};
```

SynthAudioSource.cpp

```
#include "../JuceLibraryCode/JuceHeader.h"
#include "PluginProcessor.h"
#include "SynthAudioSource.h"


//Attach processor and MidiKeyboardState to the SynthAudioSource instance.
SynthAudioSource::SynthAudioSource(MidiKeyboardState&
keyState,SynthAudioProcessor& p) : keyboardState(keyState),processor(p) {

}

void SynthAudioSource::prepareToPlay(int samplePerBlockExpected, double
sampleRate) {
    //Prepare initial settings.
    midiCollector.reset(sampleRate);
    processor.prepareToPlay(sampleRate, samplePerBlockExpected);
}

void SynthAudioSource::getNextAudioBlock(const AudioSourceChannelInfo&
bufferToFill) {
    //Clear the audio buffer.
    bufferToFill.clearActiveBufferRegion();

    //Create a buffer for the incoming MidiMessage.
    MidiBuffer incomingMidi;
```

```
    //Fetch MidiMessage from midiCollector into the prepared buffer.
    midiCollector.removeNextBlockOfMessages(incomingMidi,
bufferToFill.numSamples);

    keyboardState.processNextMidiBuffer(incomingMidi, 0,
bufferToFill.numSamples, true);
    processor.processBlock(*bufferToFill.buffer, incomingMidi);
}

MidiMessageCollector*  SynthAudioSource::getMidiCollector() {
    return &midiCollector;
}
```

SynthSound.h

```
#pragma once

#include "../JuceLibraryCode/JuceHeader.h"

class SynthSound : public SynthesiserSound
{
public:
    bool appliesToNote (int midiNoteNumber) override { return true; }
    bool appliesToChannel (int midiNoteNumber) override { return true; }
};
```

SynthVoice.h

```
#pragma once

#include "../JuceLibraryCode/JuceHeader.h"
#include "SynthSound.h"
#include "maximilian.h"


class SynthVoice : public SynthesiserVoice
{
public:
    bool canPlaySound (SynthesiserSound* sound) override;
    void setOscType(float* selection);
    double getOscType ();
    void setVolume(float* vol);
    void getEnvelopeParams(float* attack, float* decay, float* sustain,
float* release);
    double getEnvelope();
    void setFilterParams (float* filterType, float* filterCutoff, float*
filterRes);
    void startNote (int midiNoteNumber, float velocity, SynthesiserSound*
sound, int currentPitchWheelPosition) override;
    void stopNote (float velocity, bool allowTailOff) override;
    void pitchWheelMoved (int newPitchWheelValue) override {}
    void controllerMoved (int controllerNumber, int newControllerValue)
override {}
    void renderNextBlock (AudioBuffer <float> &outputBuffer, int
startSample, int numSamples) override;
```

```
private:
    double frequency;
    int waveSelection;
    double gain{0.5};
    int filterChoice;
    float cutoff;
    float resonance;
    maxiOsc oscillator;
    maxiEnv envelope;
    maxiDistortion distort;
};
```

SynthVoice.cpp

```
#include "SynthVoice.h"

bool SynthVoice::canPlaySound(SynthesiserSound* sound) {
    return dynamic_cast <SynthSound*>(sound) != nullptr;
}

void SynthVoice::setOscType(float* selection){
    waveSelection = *selection;
}

void SynthVoice::setVolume(float* vol) {
    gain = *vol;
}

double SynthVoice::getOscType() {
        if (waveSelection == 0)return oscillator.sinewave(frequency);
        if (waveSelection == 1)return oscillator.saw(frequency);
        if (waveSelection == 2)return oscillator.square(frequency);
        else return oscillator.sinewave(frequency);
}

void SynthVoice::getEnvelopeParams(float *attack, float *decay, float
*sustain, float *release) {
    envelope.setAttack(*attack);
    envelope.setDecay(*decay);
    envelope.setSustain(*sustain);
    envelope.setRelease(*release);
}

double SynthVoice::getEnvelope()
{
    return envelope.adsr(getOscType(), envelope.trigger);
}

void SynthVoice::setFilterParams (float* filterType, float* filterCutoff,
float* filterRes) {
    filterChoice = *filterType;
    cutoff = *filterCutoff;
    resonance = *filterRes;
}
```

```
void SynthVoice::startNote (int midiNoteNumber, float velocity,
SynthesiserSound* sound, int currentPitchWheelPosition) {
    envelope.trigger = 1;
    frequency = MidiMessage::getMidiNoteInHertz(midiNoteNumber);
}

void SynthVoice::stopNote (float velocity, bool allowTailOff) {
    envelope.trigger = 0;
    allowTailOff = true;
    if (velocity == 0)
        clearCurrentNote();
}

void SynthVoice::renderNextBlock (AudioBuffer <float> &outputBuffer, int
startSample, int numSamples) {
    for (int sample = 0; sample < numSamples; ++sample)
    {
        for (int channel = 0; channel < outputBuffer.getNumChannels();
++channel)
        {
            outputBuffer.addSample(channel, startSample, getEnvelope() *
(gain / 10.0f));
        }
        ++startSample;
    }
}
```

Oscillator.h

```
#pragma once

#include "../JuceLibraryCode/JuceHeader.h"
#include "PluginProcessor.h"

class Oscillator    : public Component,
                      private ComboBox::Listener
{
public:
    Oscillator(SynthAudioProcessor&);
    ~Oscillator();

    void paint (Graphics&) override;
    void resized() override;
    void comboBoxChanged(ComboBox*) override {};

private:


    ComboBox oscMenu;
    ScopedPointer<AudioProcessorValueTreeState::ComboBoxAttachment>
waveSelection;

    //Processor Referrence
    SynthAudioProcessor& processor;

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (Oscillator)
};
```

Oscillator.cpp

```
#include "../JuceLibraryCode/JuceHeader.h"
#include "Oscillator.h"

Oscillator::Oscillator(SynthAudioProcessor& p) :
processor(p)
{
    setSize(220,85);
    oscMenu.addItem("Sine", 1);
    oscMenu.addItem("Saw", 2);
    oscMenu.addItem("Square", 3);
    oscMenu.setJustificationType(Justification::centred);
    addAndMakeVisible(&oscMenu);
    oscMenu.addListener(this);
    waveSelection = new AudioProcessorValueTreeState::ComboBoxAttachment
(processor.tree, "wavetype", oscMenu);
}

Oscillator::~Oscillator()
{
}

void Oscillator::paint (Graphics& g)
{
    Rectangle<int> titleArea =
getLocalBounds().removeFromTop(20).removeFromBottom(10);
    g.fillAll (getLookAndFeel().findColour
(ResizableWindow::backgroundColourId));
    g.setColour(Colours::white);
    g.drawText("Oscillator", titleArea, Justification::centredTop);
}

void Oscillator::resized()
{
    Rectangle<int> area = getLocalBounds().removeFromBottom(65);
    area = area.removeFromTop(40).removeFromBottom(20);
    oscMenu.setBounds(area.reduced(20, 0));
}
```

Filter.h

```
#pragma once

#include "../JuceLibraryCode/JuceHeader.h"
#include "PluginProcessor.h"

class Filter : public Component
{
public:
    Filter(SynthAudioProcessor&);
    ~Filter();

    void paint (Graphics&) override;
    void resized() override;
```

```
private:
    Slider filterCutoff;
    Slider filterRes;

    ComboBox filterMenu;

    ScopedPointer<AudioProcessorValueTreeState::ComboBoxAttachment>
filterTypeVal;
    ScopedPointer<AudioProcessorValueTreeState::SliderAttachment>
filterVal;
    ScopedPointer<AudioProcessorValueTreeState::SliderAttachment> resVal;

    //Processor Referrence
    SynthAudioProcessor& processor;

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (Filter)
};
```

Filter.cpp

```
#include "../JuceLibraryCode/JuceHeader.h"
#include "Filter.h"

Filter::Filter(SynthAudioProcessor& p) : processor(p)
{
    setSize(350,125);
    filterMenu.addItem("Low Pass", 1);
    filterMenu.addItem("High Pass", 2);
    filterMenu.addItem("Band Pass", 3);
    filterMenu.setJustificationType(Justification::centred);
    addAndMakeVisible(&filterMenu);
    filterTypeVal = new AudioProcessorValueTreeState::ComboBoxAttachment
(processor.tree, "filterType", filterMenu);

    filterCutoff.setSliderStyle(Slider::SliderStyle::RotaryHorizontalVerti
calDrag);
    filterCutoff.setRange(20.0, 10000.0);
    filterCutoff.setValue (400.0);
    filterCutoff.setTextBoxStyle(Slider::NoTextBox, false, 0, 0);
    addAndMakeVisible(&filterCutoff);
    filterVal = new AudioProcessorValueTreeState::SliderAttachment
(processor.tree, "filterCutoff", filterCutoff);
    filterCutoff.setSkewFactorFromMidPoint(1000.0);

    filterRes.setSliderStyle(Slider::SliderStyle::RotaryHorizontalVertical
Drag);
    filterRes.setRange(1, 5);
    filterRes.setValue(1);
    filterRes.setTextBoxStyle(Slider::NoTextBox, false, 0, 0);
    addAndMakeVisible(&filterRes);
    resVal = new AudioProcessorValueTreeState::SliderAttachment
(processor.tree, "filterRes", filterRes);
}

Filter::~Filter()
{
}
```

```
void Filter::paint (Graphics& g)
{
    Rectangle<int> titleArea =
getLocalBounds().removeFromTop(20).removeFromBottom(10);
    //g.fillAll (getLookAndFeel().findColour
(ResizableWindow::backgroundColourId));
    g.setColour(Colours::white);
    g.drawText("Filter", titleArea, Justification::centredTop);
}

void Filter::resized()
{
    Rectangle<int> area = getLocalBounds().removeFromBottom(105);
    filterMenu.setBounds(area.removeFromTop(30).removeFromBottom(20).reduc
ed(80, 0));
    auto sliderArea = area.removeFromBottom(area.getHeight());
    filterCutoff.setBounds
(sliderArea.removeFromRight(sliderArea.getWidth() / 2));
    filterRes.setBounds(sliderArea);

}
```

Envelope.h

```
#pragma once

#include "../JuceLibraryCode/JuceHeader.h"
#include "PluginProcessor.h"

class Envelope : public Component
{
public:
    Envelope(SynthAudioProcessor&);
    ~Envelope();

    void paint (Graphics&) override;
    void resized() override;

private:
    Slider attackSlider;
    Slider decaySlider;
    Slider sustainSlider;
    Slider releaseSlider;

    ScopedPointer<AudioProcessorValueTreeState::SliderAttachment>
attackVal;
    ScopedPointer<AudioProcessorValueTreeState::SliderAttachment>
decayVal;
    ScopedPointer<AudioProcessorValueTreeState::SliderAttachment>
sustainVal;
    ScopedPointer<AudioProcessorValueTreeState::SliderAttachment>
releaseVal;

    //Processor Referrence
    SynthAudioProcessor& processor;

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (Envelope)
};
```

Envelope.cpp

```
#include "../JuceLibraryCode/JuceHeader.h"
#include "Envelope.h"

Envelope::Envelope(SynthAudioProcessor& p) :
processor(p)
{
    setSize(350,255 - 130);

        //slider initialization values
        attackSlider.setSliderStyle(Slider::SliderStyle::LinearVertical);
        attackSlider.setRange(0.1f, 5000.0f);
        attackSlider.setValue(0.1f);
        attackSlider.setTextBoxStyle(Slider::NoTextBox, true, 0, 0);
        addAndMakeVisible(&attackSlider);

        decaySlider.setSliderStyle(Slider::SliderStyle::LinearVertical);
        decaySlider.setRange(1.0f, 2000.0f);
        decaySlider.setValue(1.0f);
        decaySlider.setTextBoxStyle(Slider::NoTextBox, true, 0, 0);
        addAndMakeVisible(&decaySlider);

        sustainSlider.setSliderStyle(Slider::SliderStyle::LinearVertical);
        sustainSlider.setRange(0.0f, 1.0f);
        sustainSlider.setValue(0.8f);
        sustainSlider.setTextBoxStyle(Slider::NoTextBox, true, 0, 0);
        addAndMakeVisible(&sustainSlider);

        releaseSlider.setSliderStyle(Slider::SliderStyle::LinearVertical);
        releaseSlider.setRange(0.1f, 5000.0f);
        releaseSlider.setValue(0.8f);
        releaseSlider.setTextBoxStyle(Slider::NoTextBox, true, 0, 0);
        addAndMakeVisible(&releaseSlider);

        //sends value of the sliders to the tree state in the processor
        attackVal = new AudioProcessorValueTreeState::SliderAttachment
(processor.tree, "attack", attackSlider);
        decayVal = new AudioProcessorValueTreeState::SliderAttachment
(processor.tree, "decay", decaySlider);
        sustainVal = new AudioProcessorValueTreeState::SliderAttachment
(processor.tree, "sustain", sustainSlider);
        releaseVal = new AudioProcessorValueTreeState::SliderAttachment
(processor.tree, "release", releaseSlider);
}

Envelope::~Envelope()
{
}
```

```
void Envelope::paint (Graphics& g)
{
    Rectangle<int> titleArea (0, 10, getWidth(), 20);

    g.fillAll (getLookAndFeel().findColour
(ResizableWindow::backgroundColourId));
    g.setColour(Colours::white);
    g.drawText("Envelope", titleArea, Justification::centredTop);

    Rectangle<int> area = getLocalBounds().removeFromBottom(getHeight() -
20);
    g.drawText ("Attack",area.removeFromLeft(87),
Justification::centredBottom);
    g.drawText ("Sustain",area.removeFromLeft(87),
Justification::centredBottom);
    g.drawText ("Decay",area.removeFromLeft(87),
Justification::centredBottom);
    g.drawText ("Release",area.removeFromLeft(87),
Justification::centredBottom);

}

void Envelope::resized()
{

    Rectangle<int> area =
getLocalBounds().removeFromBottom(getLocalBounds().getHeight() - 20);
    attackSlider.setBounds(area.removeFromLeft(87).reduced(0, 10));
    sustainSlider.setBounds(area.removeFromLeft(87).reduced(0, 10));
    decaySlider.setBounds(area.removeFromLeft(87).reduced(0, 10));
    releaseSlider.setBounds(area.removeFromLeft(87).reduced(0, 10));

}
```

Volume.h

```
#pragma once

#include "../../oopSynth/JuceLibraryCode/JuceHeader.h"
#include "PluginProcessor.h"

class Volume    : public Component
{
public:
    Volume(SynthAudioProcessor&);
    ~Volume();

    void paint (Graphics&) override;
    void resized() override;

private:
    Slider volumeSlider;
    ScopedPointer<AudioProcessorValueTreeState::SliderAttachment>
volumeLevel;
    //Processor Referrence
    SynthAudioProcessor& processor;
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (Volume)
};
```

Volume.cpp

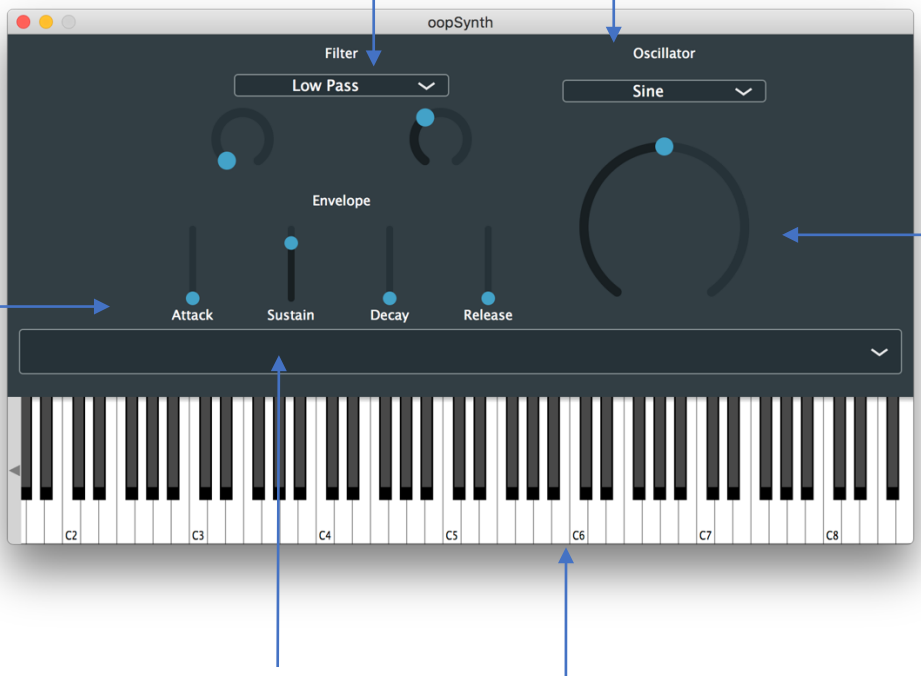
```
#include "../../oopSynth/JuceLibraryCode/JuceHeader.h"
#include "Volume.h"

//==============================================================================
Volume::Volume(SynthAudioProcessor& p) : processor(p)
{
    setSize(200, 200);
    volumeSlider.setRange(0, 10);
    volumeSlider.setValue(5);
    volumeSlider.setSliderStyle(Slider::SliderStyle::RotaryHorizontalVerti
calDrag);
    volumeSlider.setTextBoxStyle(Slider::TextBoxBelow, true, 0, 0);
    addAndMakeVisible(&volumeSlider);

    volumeLevel = new
AudioProcessorValueTreeState::SliderAttachment(processor.tree,"volume",vol
umeSlider);
}

Volume::~Volume()
{
}

void Volume::paint (Graphics& g)
{
    g.fillAll (getLookAndFeel().findColour
(ResizableWindow::backgroundColourId));
}

void Volume::resized()
{
    Rectangle<int> area = getLocalBounds();
    volumeSlider.setBounds(area);
}
```

Filter

- Filter type selector
- Resonance knob
- Cutoff frequency knob

Oscillator

- Wave type selector



Envelope

- Attack slider
- Sustain slider
- Decay slider
- Release slider

MIDI Input

- External MIDI input selector

Volume

- Volume level knob

Keyboard

- Virtual MIDI input keyboard