Implementation

Umang Hirani 92200133025

SimpleSynth is a full-featured audio synthesizer plugin built using the JUCE framework. It provides a set of audio processing capabilities including oscillator synthesis, envelopes, filters, and multiple effects processors (reverb, distortion, chorus, and limiting etc…..)

Architecture and Code Structure

Component Hierarchy

The application follows a modular, component-based architecture:

```
SimpleSynthAudioProcessor
├── SynthVoice (Voice Management & DSP)
│   ├── Oscillator (Sound Generation)
│   ├── Envelope (ADSR)
│   ├── Filter (Signal Processing)
│   └── Effects Chain
│       ├── Crunch (Distortion)
│       ├── Chorus
│       ├── Reverb
│       └── Limiter
├── SimpleSynthAudioProcessorEditor (UI Layer)
│   ├── OscillatorComponent
│   ├── EnvelopeComponent
│   ├── FilterComponent
│   ├── CrunchComponent
│   ├── ChorusComponent
│   ├── ReverbComponent
│   ├── LimiterComponent
│   └── PresetPanel
└── PresetManager (State Management)
```

The above structure was made using the tree command in fedora linux.

<u>File Organization</u>

Core Processing:

PluginProcessor.h/cpp - Main audio processor, parameter management

SynthVoice.h - Voice implementation with Maximilian DSP and Signalsmith effects

SynthSound.h - Sound definition for synthesizer

UI Components:

PluginEditor.h/cpp - Main editor window and component layout

Oscillator.h/cpp - Waveform selection and oscillator controls

Envelope.h/cpp - ADSR envelope controls

Filter.h/cpp - Filter frequency and resonance controls

Crunch.h/cpp - Distortion/saturation effect controls

Chorus.h/cpp - Chorus effect parameters

ReverbComponent.h/cpp - Reverb effect controls

Limiter.h/cpp - Dynamic range limiting controls

State Management:

PresetManager.h/cpp - Preset save/load functionality

PresetPanel.h/cpp - Preset UI interface

Implementation Details

Audio Processing Pipeline

The audio processing follows this signal flow:

<u>Note Generation:</u> MIDI input triggers voice allocation

<u>Oscillator:</u> Generates waveform (sine, square, saw) with octave/semitone transpose

<u>Noise Generator:</u> Optional white noise source mixed with oscillator

Envelope: ADSR envelope shapes amplitude over time

<u>Filter:</u> Low-pass filter with cutoff frequency and resonance

Effects Chain:

Crunch (distortion/saturation)

Chorus (modulation)

Reverb (spatial processing)

Limiter (dynamic control)

Key Algorithms

ADSR Envelope Implementation (Maximilian library):

```
double theSound = env1.adsr(setOscType(), env1.trigger) * level * oscAmp;
```

The envelope applies attack, decay, sustain, and release phases to control amplitude dynamics.

Filter Processing:

```
float filteredSound = filter1.lores(theSum, cutoffFrequency, filterResonance);
```

A low-pass filter with resonance control for frequency content shaping.

Effects Processing (Signalsmith library):

```
crunch1.process(channels, numSamples);
chorus1.process(channels, numSamples);
reverb1.process(channels, numSamples);
limiter1.process(channels, numSamples);
```

Effects are applied in series to the stereo output buffer.

Parameter Management

The system uses JUCE's AudioProcessorValueTreeState for thread-safe parameter management:

```
tree(*this, nullptr, "PARAMETERS", {
   std::make_unique<AudioParameterFloat>("attack", "Attack", 0.1f, 5000.0f, 100.0f),
   std::make_unique<AudioParameterFloat>("frequency", "Cutoff Frequency", 20.0f, 10000.0f,
1000.0f),
})
```

UI components connect to parameters via attachments:

```
attackAttachment = std::make_unique<AudioProcessorValueTreeState::SliderAttachment>(
    processor.tree, "attack", AttackSlider
);
```

Voice Management

The synthesizer uses polyphonic voice allocation with 5 voices:

```
int numChannels = 5;
for (int i = 0; i < numChannels; ++i) {
    mySynth.addVoice(new SynthVoice());
}
```

Each voice independently processes MIDI note events and renders audio.

Integration Across Components

Audio-to-UI Integration

The processor communicates parameter changes to all active voices:

```
for (int i = 0; i < mySynth.getNumVoices(); i++) {
    if ((myVoice = dynamic_cast<SynthVoice*>(mySynth.getVoice(i)))) {
        myVoice->setAttack(tree.getRawParameterValue("attack")->load());
        myVoice->setReverbWet(tree.getRawParameterValue("reverbWet")->load());
    }
}
```

Visualizer Integration

Audio is sent to the waveform visualizer for real-time display:

```
if (auto* editor = dynamic_cast<SimpleSynthAudioProcessorEditor*>(getActiveEditor())) {
    const int downsampleFactor = 4;
    for (int sample = 0; sample < buffer.getNumSamples(); sample += downsampleFactor) {
        float sampleValue = buffer.getSample(0, sample);
        editor->pushNextSample(sampleValue);
    }
}
```

Preset System Integration

The preset manager integrates with the parameter tree for state:

```
void PresetManager::savePreset(const String& presetName) {
    auto state = valueTreeState.copyState();
    state.setProperty(presetNameProperty, presetName, nullptr);
    if (auto xml = state.createXml()) {
        File presetFile = defaultDirectory.getChildFile(presetName + "." + extension);
        xml->writeTo(presetFile);
    }
}
```

There are many snippets of code that I have missed but really, this is the core gist of the whole project that there is to offer.