SUBSYNTH

SINTETIZZATORE CON FILTRI ED EFFETTI.
REALIZZATO DA CRISTIAN PIETRONIRO.

SOMMARIO

INTRODUZIONE

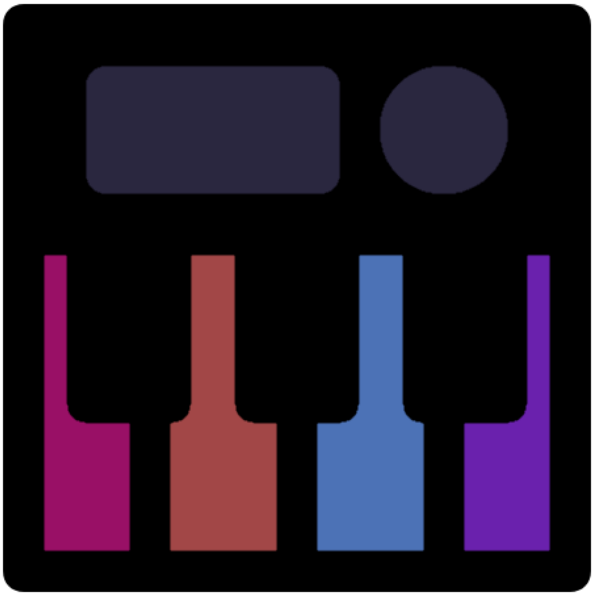
SubSynth è un sintetizzatore con supporto MIDI, voci polifoniche, filtri ed effetti post-synth disattivabili.

Permette il salvataggio e caricamento di preset tramite file.
Viene fornito con dei preset di fabbrica per testare velocemente i suoni che possono essere raggiunti cambiando forme d'onda, inviluppi, filtri, effetti e altro ancora.

I vari parametri sono gestiti tramite AudioProcessorValueTreeState (APVTS) e possono essere regolati mediante interfaccia grafica.

L'interfaccia grafica è realizzata interamente in Juce.
Ogni pezzo fondamentale del sintetizzatore ha il proprio componente: mediante funzioni personalizzate ogni componente viene correttamente spaziato, impostato e colorato.

Durante la progettazione, è stata rispettata per i principali componenti una netta suddivisione tra UI e logica.
Altre classi helper contengono funzioni utili per la GUI (Utils) o i parametri centralizzati (Parameters).

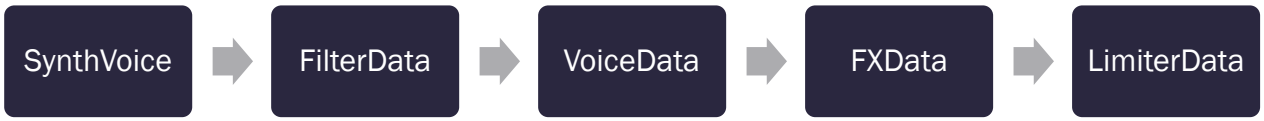


ARCHITETTURA E LOGICA

PLUGINPROCESSOR

Il core DSP per-voce di SubSynth ruota attorno a **SynthVoice**, coordinato dal SubSynthAudioProcessor.

PluginProcessor gestisce l'orchestrazione: prepara la catena DSP con prepareToPlay, aggiorna i parametri e rende i blocchi audio in processBlock, e applica la catena post-synth (FXData + LimiterData).



Anche la gestione della polifonia è a carico del Processor.
In processBlock, il numero di voci desiderate viene letto dall'APVTS e aggiornato in tempo reale.
In renderNextBlock, ogni voce somma i propri samples nel buffer di uscita, ottenendo la resa polifonica complessiva.
Poiché l'utente può in qualsiasi momento aumentare e ridurre le voci a disposizione mediante GUI, ho inserito in updateSynthVoices dei controlli: in particolare, per la rimozione, sono privilegiate quelle inattive.

SYNTHVOICE

Ogni voce (SynthVoice) incapsula la generazione del segnale e la sua modellazione:

- **Oscillatore (OscData):** produce il campione grezzo.
  In OscData troviamo le funzioni di preparazione dell'oscillatore principale e dell'oscillatore per la modulazione FM.
  La frequenza dell'oscillatore principale viene aggiornata in base alla nota MIDI corrente sommata all'eventuale modulazione FM.
  La forma d'onda principale può essere scelta tra sinusoide (Sine), dente di sega (Saw), quadra (Square) o triangolare (Triangle).
  L'**oscillatore secondario (fmOsc)** è sempre sinusoidale e implementa una semplice modulazione di frequenza, applicata per sample.
  Permette di ottenere un timbro arricchito da variazioni di pitch a velocità e intensità controllabili.
- **Guadagno (GainData):** gestisce il livello di uscita con smussatura per-sample.
  La smussatura è necessaria per evitare salti udibili quando il gain viene modificato.
  - In base al numero di voci attive, è stato implementato uno scaling polifonico.
    Gestisce automaticamente il livello per-voce quando più note suonano contemporaneamente, prevenendo il clipping e mantenendo un bilanciamento sonoro coerente.
    L'energia complessiva cresce in modo controllato al crescere della polifonia, con un fattore di riduzione equivalente a $\frac{1}{\sqrt{n}}$, con $n = numero\ voci$.
- **Inviluppo d'ampiezza / ADSR (ADSRData):** controlla l'andamento temporale dell'ampiezza della voce e viene applicato direttamente al campione generato dall'oscillatore prima della filtratura.
  Può essere regolato secondo i parametri di Attack, Delay, Sustain, Release.
- **Inviluppo di modulazione / modADSR (ADSRData):** controlla dinamicamente il filtro per-voce, modellando nel tempo la sua frequenza di taglio.

FILTERDATA

Il filtro è uno State Variable (Low-Pass/Band-Pass/High-Pass) con cutoff e risonanza (Q) aggiornati di continuo in funzione del modulatore (modADSR); espone sia **process(buffer)** per elaborazioni a blocco sia **processSample(channel, sample)** per l'uso strettamente per-voce.
Sia il tipo di filtro che i valori di cutoff e risonanza possono essere regolati mediante la GUI.
La cutoff viene modulata e clampata tra $20\ Hz$ e $20\ kHz$ per evitare instabilità numeriche e prevenire valori inaudibili.

VOICEDATA

Gestisce la configurazione e l'aggiornamento dei parametri di ogni voce.
Questa classe è molto snella: contiene semplicemente due funzioni ed è stata pensata principalmente per migliorare la leggibilità di PluginProcessor.

FXDATA

Incapsula la catena di effetti post-synth del plugin e gestisce selezione, configurazione e mix Dry / Wet in modo centralizzato.
Internamente usa una juce::dsp::ProcessorChain, che permette di eseguire vari classi di processing in sequenza.
Ogni classe rappresenta uno degli effetti a disposizione dell'utente:

1. **Chorus:** realizzato a partire da juce::dsp::Chorus, può essere regolato in rate, spessore, delay e feedback.
   Duplica e ritarda il segnale modulando il delay con un LFO, creando spessore e movimento.
2. **Flanger:** anche questo è un Chorus e ha gli stessi parametri ma con range più stretti.
   La modulazione del micro-delay genera le cancellazioni / rafforzamenti periodici tipiche del flanger.
3. **Reverb:** realizzato con juce::dsp::Reverb, può essere regolato in dimensione stanza, damping (attenuazione alte frequenze) e larghezza stereo.
   Simula un campo riverberante.

Gli effetti possono essere bypassati (e quindi non applicati).
Durante il processing degli effetti, inizialmente vengono impostati tutti gli effetti come bypassati.
L'effetto viene selezionato tramite GUI con un menù stile dropdown (a tendina) e il tipo attualmente selezionato diventa l'unico non bypassato e quindi processato.

Ho deciso di inserire tra la scelta degli effetti anche **l'effetto nullo**: il funzionamento è equivalente al bypass, ma nasconde anche alcuni elementi grafici (quali slider, mix wet / dry e toggle del bypass).
A prescindere che ci sia bypass o effetto nullo, i lavori di processing vengono interrotti sul nascere.

LIMITERDATA

Incapsula il limiter finale della catena per proteggere l'uscita da picchi e clipping.

Come per VoiceData, è una classe leggera in cui ho deciso di contenere la preparazione e il processing del limiter. I parametri chiave di LimiterData sono:

- **Threshold (dB):** controlla il livello di soglia oltre cui i picchi vengono limitati.
- **Release (ms):** definisce il tempo di rilascio del limiter dopo un evento di limiting.

PARAMETRI

Lavorando con i parametri, è indispensabile la consistenza e avere meno punti di modifica possibili.

Parameters.h definisce gli ID dei parametri del plugin e costruisce il ParameterLayout dell'**AudioProcessorValueTreeState**, centralizzando la configurazione delle varie parti.

Ogni parametro è creato con il tipo appropriato (AudioParameterChoice, AudioParameterFloat, AudioParameterInt, AudioParameterBool), con range normalizzati, step, skew e valori di default sensati; gli ID stringa (costanti constexpr) garantiscono coerenza tra GUI e DSP.

La funzione **createParameters()** ritorna il layout completo, usato dal processor per inizializzare l'APVTS e consentire attacchi dei controlli UI e salvataggio/ripristino dello stato.

GESTIONE DEI PRESET

PresetManager gestisce i preset del plugin integrando l'**AudioProcessorValueTreeState**.

Si occupa della creazione della directory predefinita e della scrittura, lettura e cancellazione dei preset.

La directory predefinita, dove vengono salvati i preset degli utenti, è calcolata da juce:

```
const File PresetManager::defaultDirectory{
File::getSpecialLocation(File::commonDocumentsDirectory).getChildFile(ProjectInfo::companyNam
e).getChildFile(ProjectInfo::projectName) };
```

Nel caso di Windows, potrebbe essere:

```
"C:\Users\Public\Documents\univpm\SubSynth"
```

I preset di fabbrica vengono invece caricati tramite BinaryData e vanno caricati prima della compilazione nella cartella Resources (sia in Projucer che nei file della soluzione).
I preset di fabbrica non possono essere cancellati o sovrascritti.

Oltre alle funzioni già citate, in PresetManager troviamo anche:

- **setMissingParamsToDefaults:** se un preset non ha alcun valore salvato per un determinato parametro, quel parametro viene impostato al suo valore di default;
    - Questa funzione è stata implementata per garantire retrocompatibilità: può essere usato senza problemi un preset di versioni precedenti, creato prima che venissero implementati alcuni componenti;
- **purgeUnknownParameters:** ripulisce eventuali parametri obsoleti al caricamento;
- Funzioni che restituiscono come StringArray i preset di fabbrica, i preset utente o entrambi;
- Funzioni di controllo sul preset (se è di fabbrica, se il nome è valido);
- **juce::Value currentPreset:** un wrapper reattivo che punta alla property "presetName" dell'APVTS.
  Viene collegato nel costruttore e aggiornato da un listener se la radice viene reindirizzata, in modo da garantire coerenza tra UI e stato senza gestire manualmente sincronizzazioni e notifiche.

INTERFACCIA E COMPONENTI

L'interfaccia è stata realizzata in Juce, senza elementi grafici esterni (eccetto l'icona).

Presenta tutti i parametri necessari per modificare il suono (come quelli descritti nei capitoli precedenti), oltre a una barra superiore che permette di salvare, caricare e cancellare i preset.

Il focus è stato quello di riutilizzare più codice possibile tra i vari componenti, sia per garantire coerenza tra gli stili e le spaziature, sia per ridurre ridondanze ed errori.

Ho deciso di colorare i componenti correlati tra loro con lo stesso colore: ad esempio, l'inviluppo relativo al filtro (Mod Envelope) ha lo stesso tema del componente filtro (Filter).

UTILS (CLASSE HELPER)

I file **Utils.h** e **Utils.cpp** raccolgono utilità UI basate su JUCE per rendere più rapido e coerente il layout e lo styling dei controlli:

- definiscono costanti di **padding / colore** e funzioni helper per gestire bordi, contorni colorati, titolo e spaziatura;
- offrono tema colore per **ComboBox** e **Button** (themeComboBox, themeButton) basati sui colori usati per i bordi del componente;
- implementano componenti **wrapper** riutilizzabili
    - **LabeledSlider** (slider con label e SliderAttachment integrato, configurazione stile/LookAndFeel, layout e visibilità);
    - **DropDown** (combo con ComboBoxAttachment, gestione delle scelte, placeholder e tema colore con LookAndFeel dedicato);
- Forniscono **funzioni di layout** come **layoutVisibleRow** per disporre in riga solo gli slider visibili e **comboAndSliderRow** per posizionare una DropDown con una riga di slider sotto, adattando dimensioni all'area contenuti.

L'utilizzo di Utils ha aiutato enormemente nella gestione dei vari componenti.
Ad esempio:

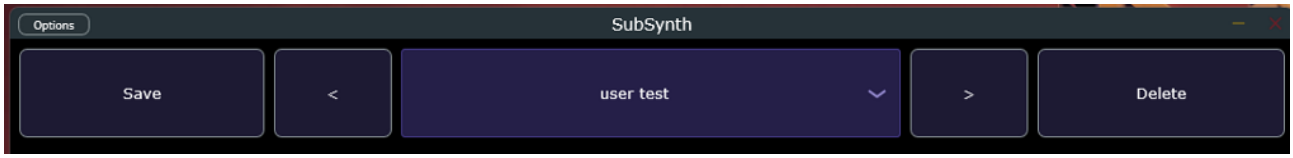
- **comboAndSliderRow** viene utilizzato sia per i filtri che per gli oscillatori, cambiando semplicemente i parametri;
- **LayoutVisibleRow** è flessibile: mostra n LabeledSlider su una singola riga, calcolando le opportune spaziature.
  Viene usato per gli ADSR, per il Limiter, per gli FX e persino all'interno di comboAndSliderRow;
- **setButton** viene usato sia per i pulsanti del PresetPanel che per i pulsanti della DialogBox;
- I **wrapper**, incorporando slider e attachment, ci risparmiano tanto codice superfluo e rendono più veloce l'implementazione di futuri componenti;
- Le **spaziature** sia tra i componenti che dentro i componenti sono coerenti.

PRESETPANEL

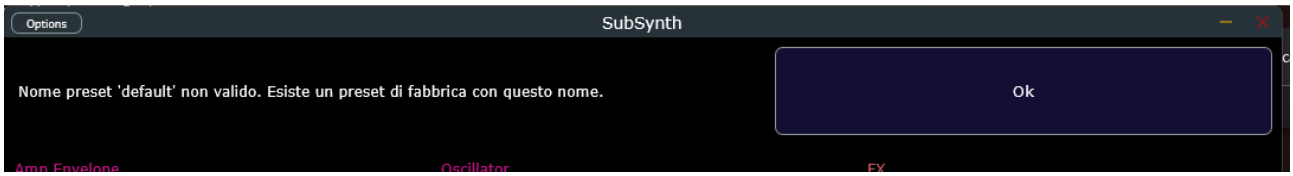
PresetPanel permette di salvare, cancellare e caricare i vari preset (di fabbrica o creati dall'utente), usando le funzioni di PresetManager.
Presenta alcuni sistemi per evitare che l'utente cancelli o crei conflitti con i preset di fabbrica.

Innanzitutto, un preset di fabbrica viene visualizzato con sfondo del dropdown nero e il pulsante di cancellazione viene disattivato.

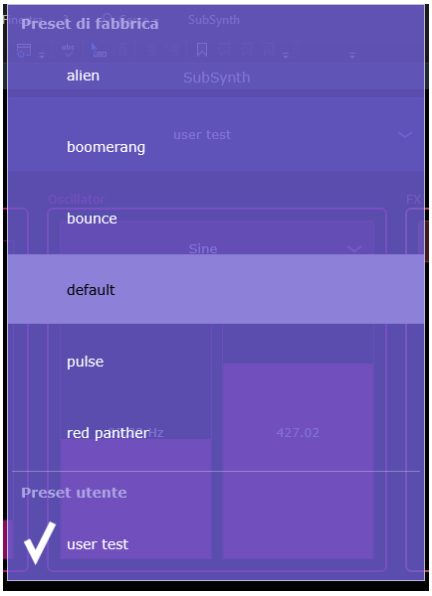
Non appena viene selezionato un preset utente, il pulsante viene riattivato, e il colore di sfondo si adatta al resto del pannello.



Se un utente prova a salvare un preset con un nome già usato da un preset di fabbrica, viene visualizzato un messaggio di errore tramite DialogBox.



Aprendo il dropdown, i preset di fabbrica e i preset utente sono separati da spaziature e titoli.

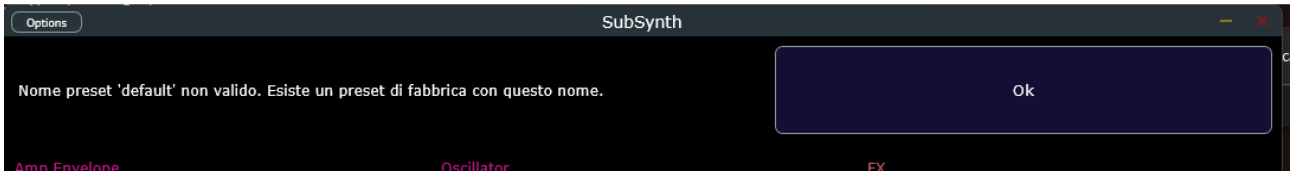


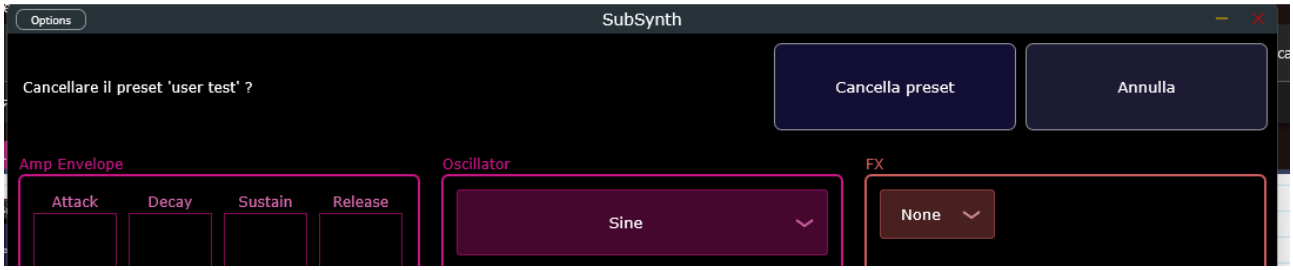
DIALOGBOX

DialogBox è una classe estremamente versatile che permette di visualizzare un messaggio seguito da 1 o 2 pulsanti.
Le funzioni dei pulsanti sono passate come parametro al costruttore di DialogBox.

In questo plugin, DialogBox compare al posto di PresetPanel per:

- Stampare messaggio di errore, con pulsante per chiudere DialogBox
- Chiedere conferma della cancellazione preset, con doppio pulsante





Esempio di implementazione: in PresetPanel, DialogBox viene mostrato al click del pulsante “Delete”:

```
juce::String msg = "Cancellare il preset '" + current + "' ?";

auto deleteFunction = [this]()
    {
        presetManager.deletePreset(presetManager.getCurrentPreset());
        loadPresetList();
        if (dialogBox) dialogBox->close();
    };

showDialogBox(msg, "Cancella preset", "Annulla", deleteFunction);
}
```

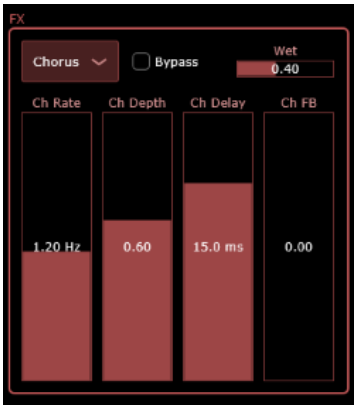
ALTRI COMPONENTI

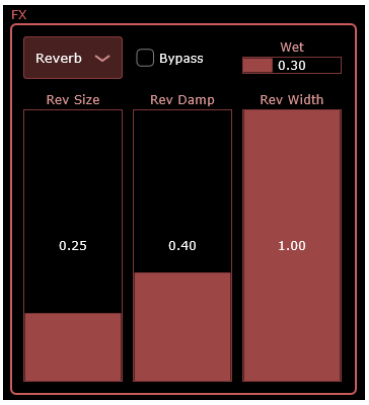
Come detto in precedenza, gli altri componenti riutilizzano le funzioni di Utils e variano solo per colori, nomi e parametri.

- **ADSRComponent** viene riutilizzata sia per l'Amp Envelope che per Mod Envelope, passando il nome dell'envelope da stampare come parametro al costruttore.
- **GainComponent** e **VoiceComponent**, a differenza delle altre, non mostrano la label ma solo il titolo sul bordo del rettangolo.
  Tramite parametri del costruttore, **GainComponent** è orizzontale e **VoiceComponent** è verticale.
- **OscComponent** e **FilterComponent** hanno stesso identico layout.
- **LimiterComponent** utilizza semplicemente LayoutVisibleRow, con impostazioni minime.

FX COMPONENT

Un caso più particolare è FXComponent.
Viene usato per tutti i tipi di effetti, incluso nullo.
In base all'effetto corrente, cambia la visualizzazione degli slider al suo interno.
Gli effetti possono avere 3 o 4 slider, ma è tutto gestito automaticamente da LayoutVisibleRow.





In caso di effetto nullo, invece degli slider viene visualizzato il logo di SubSynth.

Vengono inoltre nascosti il toggle di Bypass e lo slider del Wet.

SCELTE PROGETTUALI

Durante le fasi iniziali del progetto, ho avuto dei problemi di clicking e artefatti.
Per risolvere il problema, ho optato per un'elaborazione "per-sample", distaccandomi dalle guide e tutorial che stavo seguendo.

L'elaborazione per-sample ha un miglior controllo temporale, e mi ha permesso di evitare i click al cambio di gain, attacco e rilascio.

Un grosso svantaggio è che però richiede prestazioni maggiori rispetto all'elaborazione a blocchi.
Il costo computazionale risulta evidente specialmente in caso di molte voci (128), ma durante le fasi iniziali di progettazione non era ancora stata implementata.
Alla fine ho testato vari livelli di voci e optato per 48 voci come massimo.

Enorme importanza durante tutto il progetto è stata data al **versioning**.
Ho usato GitHub per caricare e gestire le varie commit.
Il progetto può essere visionato e scaricato da questo link:

https://github.com/PCristian00/progetto_CAES