Računalna aplikacija za parametarsko kašnjenje audio signala

Antolović, Nikola

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1255

RAČUNALNA APLIKACIJA ZA PARAMETARSKO KAŠNJENJE AUDIO SIGNALA

Nikola Antolović

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1255

RAČUNALNA APLIKACIJA ZA PARAMETARSKO KAŠNJENJE AUDIO SIGNALA

Nikola Antolović

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

ZAVRŠNI ZADATAK br. 1255

Pristupnik:	**Nikola Antolović (0036537942)**
Studij:	Elektrotehnika i informacijska tehnologija i Računarstvo
Modul:	Računarstvo
Mentor:	prof. dr. sc. Kristian Jambrošić

Zadatak: **Računalna aplikacija za parametarsko kašnjenje audio signala**

Opis zadatka:

VST dodaci su uobičajena metoda implementacije obrade audio signala u procesu audio produkcije na digitalnim audio radnim stanicama. JUCE je najčešće korišten okvir za razvoj audio aplikacija i dodataka temeljena na C++ jeziku. Korištenjem JUCE razvojne okoline, izradite aplikaciju u obliku VST dodatka koja će omogućiti promjenjivo kašnjenje audio signala ovisno o njegovoj frekvenciji kao parametar duljine kašnjenja. Testirajte rješenje za različite funkcije promjene kašnjenja signala ovisno o frekvencijskom pojasu. Optimizirajte kod kako bi se mogao što brže izvoditi, ali bez čujnih artefakata u samom audio signalu. Izmjerite koliko iznosi trajanje izvođenja koda i usporedite ga s brzinom rada komercijalnog rješenja VST dodatka za istu vrstu efekta. Analizirajte i potvrdite da li je rješenje moguće upotrijebiti i za obradu audio signala u stvarnom vremenu.

Rok za predaju rada: 14. lipnja 2024.

Sadržaj

1. Uvod

Digitalna obrada signala (DSP, *engl. Digital Signal Processing*) postala je ključni alat u modernoj glazbenoj produkciji, omogućujući glazbenicima i producentima da stvaraju sofisticirane zvučne efekte i manipuliraju zvukom na načine koji su prije bili nezamislivi. Jedan od najpopularnijih i najkorištenijih efekata u DSP-u je efekt kašnjenja (*engl. delay*) koji stvara jeku dodavanjem odgođenih kopija ulaznog signala originalnom signalu. Međutim, standardni *delay* efekt često ima ograničenja kada je riječ o obradi širokopojasnih signala što može rezultirati nejasnim zvukom.

Motivacija za ovaj projekt proizašla je iz potrebe za naprednijim rješenjima koja omogućuju precizniju kontrolu nad različitim frekvencijskim opsezima signala. Multiband Delay je *plugin* koji koristi više frekvencijskih opsega za odvojenu obradu i modulaciju. Korištenjem razvojne okoline JUCE, moćnog alata za razvoj audio aplikacija, cilj ovog projekta je stvoriti prilagodljivi *multiband delay* efekt koji može zadovoljiti potrebe suvremenih dizajnera zvučnih efekata te glazbenih profesionalaca.

Cilj završnog rada je najprije bilo razumjeti i implementirati osnovne principe DSP-a koji stoje iza efekta kašnjenja. Osim toga, bilo je potrebno istražiti i primijeniti filtere koji bi omogućili preciznu frekvencijsku podjelu signala. Na kraju, cilj je bio razviti korisnički orijentirano grafičko sučelje (GUI, *engl. Graphical User Interface*) koje omogućuje intuitivnu kontrolu parametara efekta.

Struktura završnog rada podijeljena je u nekoliko ključnih poglavlja. Nakon uvoda, slijedi detaljan opis projekta koji uključuje pregled Multiband Delay-a i tehnologija korištenih za razvoj, s posebnim naglaskom na razvojnu okolinu JUCE. Teorijska pozadina pruža kratak uvod u teoriju digitalne obrade signala, s posebnim fokusom na *delay* efekt i Linkwitz-Riley filtere. U poglavlju o implementaciji opisani su ključni aspekti razvoja, uključujući strukturu koda za DSP i GUI komponente. Na kraju, rad završava evaluacijom postignutih rezultata i zaključkom. Ovim projektom došlo bi se do jednostavnog rješenja za preciznu i fleksibilnu obradu zvuka. Koristeći napredne tehnologije i alate, cilj je bio stvoriti alat koji će biti koristan u dizajniranju zvuka te audio produkciji.

2. Opis projekta

U ovom poglavlju detaljno će biti opisan Multiband Delay *plugin* te tehnologije korištene za njegov razvoj. Multiband Delay je audio *plugin* koji koristi više frekvencijskih opsega za odvojenu obradu i modulaciju signala omogućujući preciznu kontrolu nad korištenjem *delay* efekta na različitim frekvencijama. Na taj način, korisnici mogu optimizirati zvuk i postići željene efekte uz minimalno preklapanje i zamućenje. U nastavku ćemo prvo predstaviti karakteristike i funkcionalnosti Multiband Delay-a, a zatim ćemo se fokusirati na tehnologije korištene za razvoj projekta, s posebnim naglaskom na razvojnu okolinu JUCE.

2.1. Multiband Delay plugin

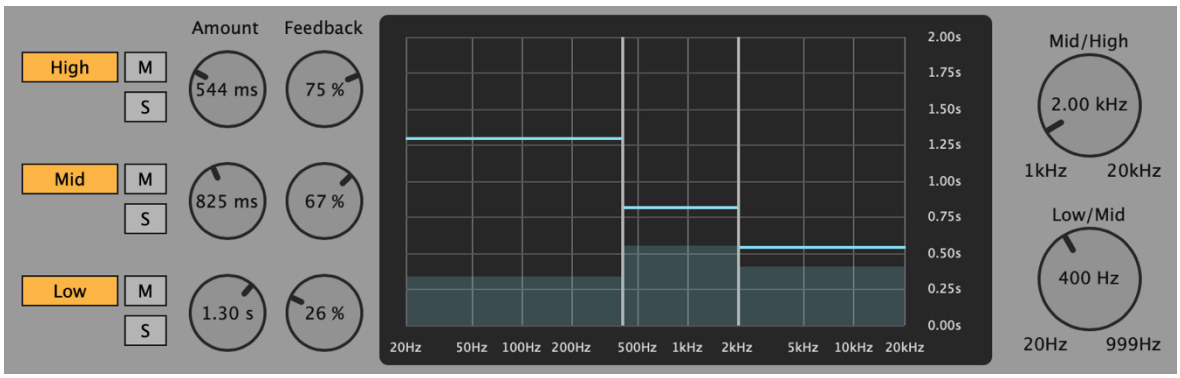
Multiband Delay je napredni audio *plugin* dizajniran za pružanje većeg stupnja kontrole nad efektom kašnjenja primijenjenim na različite frekvencijske opsege ulaznog signala. Umjesto da cijeli signal prolazi kroz jedan *delay*, Multiband Delay ga dijeli na tri zasebna frekvencijska opsega: niski, srednji i visoki. Svaki od tih opsega može se individualno obraditi s vlastitim parametrima kašnjenja, omogućujući preciznu i detaljnu kontrolu nad zvukom.

Podjela signala na tri frekvencijska opsega postiže se Linkwitz-Riley *crossover* filterima koji osiguravaju glatku i neprimjetnu tranziciju između opsega bez faznih izobličenja. Nakon podjele, svaki frekvencijski opseg prolazi kroz vlastiti *delay* modul. To omogućuje zasebne postavke količine kašnjenja (*delay amount*) te povratne veze (*feedback*) za svaki od opsega. Na primjer, korisnik može postaviti kratko kašnjenje za niske frekvencije kako bi zadržao ritmički integritet basa, dok može primijeniti duže kašnjenje za visoke frekvencije kako bi stvorio šire i ambijentalnije efekte.

Osim osnovnih parametara, Multiband Delay nudi dodatne kontrole kao što su *Bypass*, *Mute* i *Solo* za svaki opseg. *Bypass* omogućuje korisnicima da privremeno isključe efekt kašnjenja za određeni pojas, *Mute* utišava pojas, a *Solo* omogućuje slušanje samo jednog pojasa što je korisno za preciznu prilagodbu parametara. Također, tu su i *dial* kontrole za

količinu kašnjenja (*Amount*) i povratnu vezu (*Feedback*) koje korisnicima omogućuju finu regulaciju intenziteta i trajanja *delay* efekta.

Grafičko korisničko sučelje (*engl. Graphical User Interface - GUI*) Multiband Delay-a dizajnirano je tako da bude intuitivno i lako za korištenje. Sve kontrole su jasno označene i raspoređene na način koji omogućuje brz i jednostavan pristup svim funkcijama. Korisnici mogu vizualno pratiti promjene u realnom vremenu što olakšava proces prilagodbe i eksperimentiranja s različitim postavkama.



Slika 2.1 Prikaz grafičkog korisničkog sučelja Multiband Delay-a

Primjena Multiband Delay-a je široka i može se koristiti u raznim situacijama, od produkcije glazbe do postprodukcije zvuka za film i televiziju. Njegova sposobnost da precizno kontrolira *delay* efekt na različitim frekvencijama čini ga moćnim alatom za svakog audio profesionalca koji želi dodati dubinu, širinu i kompleksnost u svojim zvučnim projektima.

2.2. Razvojna okolina JUCE

Razvojna okolina JUCE (*Jules' Utility Class Extensions*) je ključni alat za razvoj audio efekt *plugin*-ova u programskom jeziku C++, pružajući širok spektar funkcionalnosti koje pojednostavljuju složeni proces razvoja.

U području digitalne obrade audio signala, JUCE se izdvaja svojim bogatim bibliotekama i alatima koji programerima omogućuju učinkovitu manipulaciju zvučnim podacima. Pored osnovnih funkcija, JUCE nudi napredne mogućnosti filtriranja i ekvilizacije koje su ključne za razvoj kvalitetnih audio efekata.

Osim toga, JUCE omogućuje jednostavnu integraciju s drugim audio API-ima (*engl. Application Programming Interface*) pružajući programerima širok raspon mogućnosti za proširenje funkcionalnosti svojih *plugin*-ova. Njegova modularna arhitektura olakšava dodavanje novih značajki i prilagodbu postojećih čineći ga idealnim alatom za iterativni razvoj audio efekata.

3. Teorija digitalne obrade signala

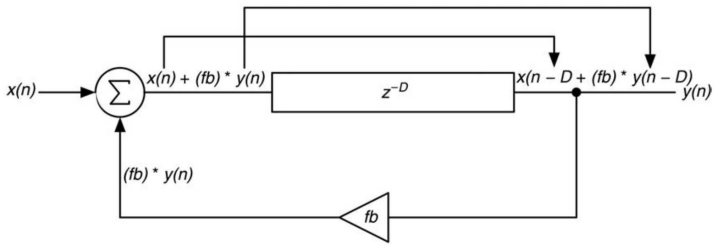
3.1. Osnovno digitalno kašnjenje

Digitalni efekt kašnjenja (*engl. delay*), ili digitalna linija kašnjenja (DDL, *engl. Digital Delay Line*), sastoji se od međuspremnika (*buffer*) za pohranu audio uzoraka. Uzorci ulaze na jedan kraj međuspremnika i izlaze na drugi kraj nakon *D* uzoraka kašnjenja, što odgovara *D* vremenskih perioda uzoraka. Put povratne veze omogućava ponavljanje odjeka kao što je prikazano na slici 3.1. Ponavljani odjeci se formiraju brzinom koja odgovara duljini linije kašnjenja *D* [1].

Izlaz digitalni *delay*-a opisan je sljedećom diferencijalnom jednadžbom:

$$y(n) = x(n - D) + fb \cdot y(n - D) \qquad (1)$$

Iz diferencijalne jednadžbe (1) može se vidjeti da se izlaz sastoji od ulaznog uzorka odgođenog za *D* uzoraka plus skalirane verzije izlaza u tom trenutku.



Slika 3.1 Shema osnovnog digitalnog kašnjenja

Sekvenca pristupa liniji kašnjenja tijekom funkcije obrade audio uzoraka:

1. Pročita se izlazna vrijednost *y(n)* iz DDL-a i napiše se u izlazni međuspremnik.

2. Izračuna se produkt $fb \cdot y(n)$.

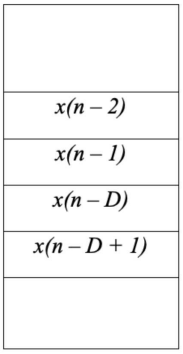
3. Ulazna vrijednost $x(n) + fb \cdot y(n)$ piše se u liniju kašnjenja.

Ako se koristi kružni međuspremnik (*engl. circular buffer*), moraju se pratiti lokacije za čitanje i pisanje u kružni međuspremnik te prema potrebi prilagoditi vrijednosti indeksa.

Pretpostavlja se da je *circular buffer* oblikovan kao lista veličine 1024 u koju su uzorci upisivani sekvencijalno u svakom vremenskom periodu uzorka, pri čemu se indeks za pisanje automatski povećava i prilagođava vraćanjem na početak međuspremnika po potrebi. U ovom slučaju, indeks za pisanje uvijek će biti povećavan za točno jedan.

```
// --- upis u međuspremnik
buffer[writeIndex] = audioSample;
// --- povećanje indeksa
writeIndex++;
// --- provjera: ako izađemo izvan međuspremnika
if(writeIndex == 1024)
// <- onda vraćamo pokazivač na početak
    writeIndex = 0;
```

Nakon provedbe koda iznad, neposredno prije upisa trenutne vrijednosti, *x(n)*, međuspremnik bi izgledao kao na slici 3.2. U tom stanju, može se vidjeti vrijednost koja je upisana u prethodnom vremenskom periodu uzorka, sada *x(n − 1)*, koja se nalazi jedno mjesto iza našeg mjesta za pisanje. Vrijednost *x(n − 2)* nalazi se dva mjesta unatrag i tako dalje. Ako se nastavi kretati kroz niz, pristupajući sve starijim podacima, na kraju se vraća na vrh i ponovno kreće ispočetka. Najstariji uzorak *x(n – D)* upravo se nalazi na indeksu na koji pokazivač za pisanje pokazuje.



Slika 3.2 Prikaz kružnog međuspremnika prije pisanja na lokaciji *x(n – D)*

Nakon operacije pisanja prepisan je najstariji uzorak najnovijim uzorkom, *x(n)*. To vodi do prvog pravila prilikom pristupa linijama kašnjenja u *plugin*-ovima: međuspremnik se uvijek čita prije nego što se ažurira novom vrijednošću kako bi se osigurao pristup posljednjem uzorku.

Nakon operacije pisanja u međuspremnik, odvija se operacija čitanja. Poznato je da se svakom odgođenom uzorku može pristupiti do maksimalne duljine linije kašnjenja ili *D* uzoraka. Pri dizajniranju efekta *plugin*-a prvo se određuje maksimalno vrijeme kašnjenja koje će *plugin* omogućiti postavljanjem veličine kružnog međuspremnika na brzinu uzorkovanja pomnoženu s brojem sekundi koje predstavljaju maksimalno vrijeme kašnjenja.

```
circularBufferSize = sampleRate * maximumDelayTimeInSeconds;
```

Indeks za pisanje postavlja se kako bi upravljao pisanjem u liniju kašnjenja, povećavajući ga i prilagođavajući prema potrebi.

Za stvaranje vremenskog kašnjenja, lokacija indeksa za čitanje bazira se tako da uvijek zaostaje za indeksom za pisanje za broj uzoraka potreban za stvaranje željenog vremena kašnjenja. Kada se podešava kontrola količine kašnjenja (*engl. delay amount*) na GUI-u, manipulira se udaljenost između indeksa za pisanje i indeksa za čitanje. Kako se indeks za čitanje pomiče dalje iza indeksa za pisanje, vrijeme kašnjenja se povećava.

Sljedeći koraci prikazuju svaku fazu operacije čitanja i pisanja u međuspremnik za fiksno vrijeme kašnjenja od 100 uzoraka:

1. Izlazni uzorak *y(n)* pročita se 100 uzoraka iza trenutne lokacije pisanja.
2. Ulaz u liniju kašnjenja formira se kao $d(n) = x(n) + fb \cdot y(n)$ i upisuje se u međuspremnik na mjestu pisanja.
3. Indeks za pisanje povećava se za +1 i prilagođava prema potrebi.
4. Indeks za čitanje povećava se za +1 i prilagođava prema potrebi. Ova lokacija indeksa za čitanje ostaje 100 uzoraka iza trenutne lokacije pisanja.

Kada se promijeni vrijeme kašnjenja, ponovno se izračunava pomak između indeksa za čitanje i pisanje. Tijekom ove operacije, kašnjenje u uzorcima oduzima se od indeksa za pisanje, a zatim se provjerava treba li ovaj novi indeks prilagoditi unatrag od vrha prema dnu međuspremnika.

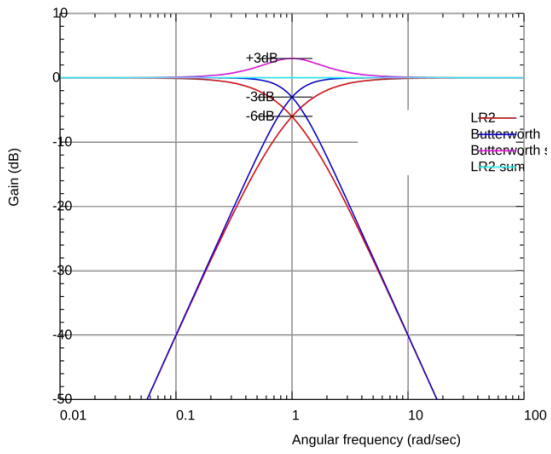
```
// --- pomakni indeks za čitanje iza indeksa za pisanje za
broj uzoraka
readIndex = writeIndex - numSamples;
// --- provjeri i prilagodi UNATRAG ako je indeks negativan
if (readIndex < 0)
      // količina prilagodbe je –Read + Length
      readIndex += bufferLength;
```

Na ovaj način, učinkovit kružni međuspremnik kreirat će se za korištenje u *plugin*-ovima, omogućujući kvalitetnu i pouzdanu implementaciju efekta kašnjenja.

3.2. Linkwitz-Riley filteri

Linkwitz-Riley (LR) filteri su vrsta IIR (*engl. Infinite Impulse Response*) filtera često korištenih za precizno razdvajanje frekvencijskih pojasa u audio sustavima kao što je Multiband Delay. Ovi filteri se ističu svojom sposobnošću da održe linearnu faznu karakteristiku unutar prolaznog pojasa, što je ključno za očuvanje kvalitete zvuka pri prelasku između različitih frekvencijskih područja.

Linkwitz-Riley filter je tip IIR filtera koji se često konstruira kao kombinacija dva Butterworth filtera, od kojih svaki ima pad od 3 dB po oktavi. Kombinacijom ova dva filtera postiže se pad od 6 dB/oct na *crossover* frekvenciji prikazan slikom 3.3. Ove filtere karakterizira linearna fazna karakteristika unutar prolaznog pojasa, što znači da se faza signala unutar svakog pojedinog frekvencijskog pojasa ne mijenja, čime se izbjegavaju fazni problemi te se time očuva vremenska usklađenost signala.



Slika 3.3 Usporedba amplitudne karakteristike zbrojenih Butterworth i Linkwitz–Riley *low-pass* i *high-pass* filtera 2. reda [2].

Prednosti Linkwitz-Riley filtera:

1. Linearna fazna karakteristika
   Zahvaljujući svojoj konstrukciji, LR filteri održavaju linearnu faznu karakteristiku unutar prolaznog pojasa. To je ključno svojstvo koje osigurava da se faza signala ne mijenja dok prolazi kroz filter što doprinosi jasnoći i transparentnosti zvuka.
2. Precizno razdvajanje frekvencijskih pojasa
   Kombinacija Butterworth filtera u LR filteru osigurava jednake amplitude signala u na *crossover* frekvenciji. Ovo je bitno za glatko i precizno razdvajanje frekvencijskih područja bez dodatnih artefakata.
3. Minimalno fazno kašnjenje
   LR filteri su *minimum phase* filteri, što znači da imaju najmanje moguće fazno kašnjenje za dani red filtera. Ovo svojstvo je korisno u situacijama gdje je važno održati vremensku usklađenost signala što je posebno bitno u audio *plugin*-ovima poput Multiband Delay-a.
4. Asimetrično zbrajanje pojaseva
   Prilikom zbrajanja pojaseva, LR filteri omogućuju glatko spajanje signala bez stvaranja prepoznatljivih artefakata ili izobličenja. Ovo svojstvo je ključno za dobivanje čistog i prirodnog zvuka prilikom ponovnog kombiniranja frekvencijskih pojaseva.

Prilikom primjene Linkwitz-Riley filtera, signal se prvo dijeli na dva dijela pomoću nisko-propusnog (*engl. low-pass*) i visoko-propusnog (*engl. high-pass*) filtera. *Low-pass* filter propušta niske frekvencije, dok *high-pass* filter propušta visoke frekvencije. Nakon toga, jedan od tih dijelova ponovno dijelimo koristeći još jedan set *low-pass* i *high-pass* filtera čime se dodatno razdvajaju srednje i visoke frekvencije.

Međutim, da bi se izbjeglo fazno kašnjenje koje bi moglo nastati zbog višestrukog filtriranja, drugi dio signala prolazi kroz *all-pass* filter. *All-pass* filter ne mijenja amplitudu signala zbog svoje ravne frekvencijske karakteristike, ali uvodi fazno kašnjenje koje kompenzira za kašnjenje uzrokovano dodatnim filtriranjem drugog dijela signala. Na taj način, sve tri frekvencijske grupe (niske, srednje i visoke) ostaju u fazi.

4. Implementacija

U ovom poglavlju analizirat će se tehnički detalji izrade *multiband delay plugin*-a. Razmotrit će se dvije ključne komponente: DSP (*engl. Digital Signal Processing*) i GUI (*engl. Graphical User Interface*).

DSP uključuje implementaciju svih ključnih funkcionalnosti koje omogućuju pravilno funkcioniranje Multiband Delay-a. Fokus će biti na strukturu i funkcionalnost datoteka kao što su pluginProcessor.h, pluginProcessor.cpp, delay.h, delay.cpp, delayBand.h i delayBand.cpp. Ove datoteke sadrže kod odgovoran za obradu audio signala, primjenu *delay* efekta na različite frekvencijske opsege i kontrolu nad parametrima efekta.

GUI pokriva dizajn i implementaciju korisničkog sučelja *plugin*-a. Korisničko sučelje je ključno za interakciju korisnika s *plugin*-om i omogućuje podešavanje parametara kao što su *Low/Mid Crossover* i *Mid/High Crossover*, kontrola gumbi za *Bypass*, *Mute* i *Solo*, kao i dijelova za postavke *Amount* i *Feedback* te središnji spektralni analizator. Detaljno će biti opisano kako je sučelje dizajnirano, koji alati i tehnike su korišteni te kako su implementirane različite kontrole i vizualni elementi.

U nastavku poglavlja, detaljno se objašnjava implementacija svake od ovih komponenti kako bi se dobio cjeloviti uvid u proces razvoja Multiband Delay *plugin*-a.

4.1. DSP

4.1.1. Delay.h i Delay.cpp

Programiranje Multiband Delay-a započeto je upravo funkcionalnošću efekta kašnjenja. Klasa `Delay` upravlja procesom efekta kašnjenja nad ulaznim audio međuspremnikom. Na procesiranje utječu različiti audio parametri poput *delayAmount*, *feedback* te *bypassed*. Te funkcionalnosti su implementirane upravo u ovoj klasi.

Datoteka delay.h sadrži deklaraciju klase `Delay` te dekleraciju varijabli i metoda koje koristimo u delay.cpp.

Delay.h:

```
class Delay
{
public:
    Delay();

    void setDelayAmount(juce::AudioParameterFloat*
newDelayAmount);
    void setFeedback(juce::AudioParameterFloat* newFeedback);
    void setBypassed(juce::AudioParameterBool* newBypassed);

    float getDelayAmount() const;
    float getFeedback() const;
    bool isBypassed() const;

    void reset(double sampleRateSize);
    void process(juce::AudioBuffer<float>& buffer,
juce::AudioBuffer<float>& delayBuffer, int
totalNumInputChannels);

private:
    void fillBuffer(juce::AudioBuffer<float>& buffer,
juce::AudioBuffer<float>& delayBuffer, int channel);
    void readFromBuffer(juce::AudioBuffer<float>& buffer,
juce::AudioBuffer<float>& delayBuffer, int channel);
    void updateBufferPositions(juce::AudioBuffer<float>&
buffer, juce::AudioBuffer<float>& delayBuffer);

    juce::AudioParameterFloat* delayAmount { nullptr };
    juce::AudioParameterFloat* feedback { nullptr };
    juce::AudioParameterBool* bypassed { nullptr };
    int writePosition { 0 };
    double sampleRate { 44100.0 };
};
```

U delay.cpp definiraju se metode deklarirane u delay.h. U središtu *delay* efekta je metoda `process` koja radi čitanje iz međuspremnika te pisanje u međuspremnik.

Delay.cpp:

```
...
void Delay::process(juce::AudioBuffer<float>& buffer,
juce::AudioBuffer<float>& delayBuffer, int
totalNumInputChannels)
{
    if (isBypassed()) // ako je bypassed nemoj nista citati
niti pisati
        return;

    for (int channel = 0; channel < totalNumInputChannels;
++channel)
    {
// prvo čitamo iz međuspremnika, onda pišemo u njega
        readFromBuffer(buffer, delayBuffer, channel);
        fillBuffer(buffer, delayBuffer, channel);
    }
// na kraju pomičemo writePointer
    updateBufferPositions(buffer, delayBuffer);
}
...
```

U glavnoj `process()` funkciji zovu se tri funkcije `readFromBuffer()`, `fillBuffer()` te `updateBufferPositions()`.

`readFromBuffer()` upravlja čitanjem iz `delayBuffer`-a te dodavanjem tog signala u `buffer` s odgovarajućim iznosom povratne veze (*feedback*).

Isto provjerava ako je trenutna pozicija čitanja (`readPosition`) plus veličina međuspremnika (`bufferSize`) manja od veličine kružnog međuspremnika (`delayBufferSize`), čitav signal se kopira iz *delayBuffer*-a u *buffer* koristeći funkciju `addFromWithRamp()` koja uzima u obzir parametar `feedbackAmount`. U slučaju da je `readPosition` blizu kraja kružnog međuspremnika i prelazi njegovu veličinu, signal se dijeli u dva dijela: prvi dio zapisuje do kraja kružnog međuspremnika i drugi dio od početka. Oba dijela se dodaju u glavni međuspremnik koristeći `addFromWithRamp()` osiguravajući glatku tranziciju i kontinuirani signal kroz cirkularni *delayBuffer*.

```
...
void Delay::readFromBuffer(juce::AudioBuffer<float>& buffer,
juce::AudioBuffer<float>& delayBuffer, int channel)
{
    auto bufferSize = buffer.getNumSamples();
    auto delayBufferSize = delayBuffer.getNumSamples();

// određivanje količinu kašnjenja
    auto delayAmountInSamples =
static_cast<int>((getDelayAmount() / 1000) * sampleRate);

    auto readPosition = writePosition - delayAmountInSamples;
    if (readPosition < 0) readPosition += delayBufferSize;

    float feedbackAmount = getFeedback() / 100;
    if (readPosition + bufferSize < delayBufferSize) // ako
    {
        buffer.addFromWithRamp(channel, 0,
delayBuffer.getReadPointer(channel, readPosition),
bufferSize, feedbackAmount, feedbackAmount);
    }
    else
    {
        auto numSamplesToEnd = delayBufferSize -
readPosition;
        buffer.addFromWithRamp(channel, 0,
delayBuffer.getReadPointer(channel, readPosition),
numSamplesToEnd, feedbackAmount, feedbackAmount);

        auto numSamplesAtStart = bufferSize -
numSamplesToEnd;
        buffer.addFromWithRamp(channel, numSamplesToEnd,
delayBuffer.getReadPointer(channel, 0), numSamplesAtStart,
feedbackAmount, feedbackAmount);
    }
}
```

`fillBuffer()` upravlja pisanjem sadržaja iz *buffer*-a u *delayBuffer* na način da osigurava kontinuirano kružno popunjavanje. Prvo se dohvaća broj uzoraka u *buffer*-u i *delayBuffer*-u. Slično kao u metodi `readFromBuffer()`, ako trenutna pozicija za

pisanje (`writePosition`) plus veličina *buffer*-a (`bufferSize`) stane unutar veličine *delayBuffer*-a (`delayBufferSize`), cijeli *buffer* se direktno kopira u *delayBuffer* od trenutne pozicije za pisanje. Ako zapisivanje prelazi kraj *delayBuffer*-a, signal se dijeli u dva dijela: prvi dio se kopira od trenutne pozicije do kraja *delayBuffer*-a, a preostali dio se kopira od početka *delayBuffer*-a. Ovaj pristup osigurava da se *delayBuffer* puni na način koji odgovara *circular buffer*-u što je ključno za *delay* efekt.

```
...
void Delay::fillBuffer(juce::AudioBuffer<float>& buffer,
juce::AudioBuffer<float>& delayBuffer, int channel)
{
    auto bufferSize = buffer.getNumSamples();
    auto delayBufferSize = delayBuffer.getNumSamples();

    if (delayBufferSize >= bufferSize + writePosition)
    {
        delayBuffer.copyFrom(channel, writePosition,
buffer.getWritePointer(channel), bufferSize);
    }
    else
    {
        auto numSamplesToEnd = delayBufferSize -
writePosition;
        jassert(numSamplesToEnd > 0 && numSamplesToEnd <=
bufferSize);
        delayBuffer.copyFrom(channel, writePosition,
buffer.getWritePointer(channel), numSamplesToEnd);

        auto numSamplesAtStart = bufferSize -
numSamplesToEnd;
        jassert(numSamplesAtStart > 0 && numSamplesAtStart <=
bufferSize);
        delayBuffer.copyFrom(channel, 0,
buffer.getWritePointer(channel, numSamplesToEnd),
numSamplesAtStart);
    }
}
```

Na kraju `process()` metode zove se `updateBufferPositions()` koja ažurira poziciju pisanja u *buffer*.

```
...
void Delay::updateBufferPositions(juce::AudioBuffer<float>&
buffer, juce::AudioBuffer<float>& delayBuffer)
{
    auto bufferSize = buffer.getNumSamples();
    auto delayBufferSize = delayBuffer.getNumSamples();
    writePosition += bufferSize;
    writePosition %= delayBufferSize;
}
```

4.1.2. DelayBand.h i DelayBand.cpp

Klasa `DelayBand` oblikuje niski, srednji i visoki pojas Multiband Delay-a te omogućuje individualnu kontrolu i manipulaciju nad pojasevima Multiband Delay-a. Svaki pojas ima svoju instancu `Delay` klase, svoj cirkularni `delayBuffer` te svoje parametre koji određuju karakteristiku *delay* efekta.

Postoje tri metode u klasi `DelayBand`:

1. `prepare()` - priprema `delayBuffer` i druge resurse za obradu zvuka, koristeći specifikacije koje se prenose putem `spec` objekta
2. `updateDelaySettings()` - ažurira postavke *delay* efekta (`delay_ms` i `feedback_pct`) na temelju trenutnih vrijednosti parametara
3. `process()` - Ova metoda obrađuje audio signal koristeći *delay* efekt za specificirani broj ulaznih kanala

`DelayBand.h:`

```
#include "Delay.h"
struct DelayBand
{
// definiranje parametra DelayBanda
    juce::AudioBuffer<float> delayBuffer;
    juce::AudioParameterFloat* delay_ms {nullptr};
    juce::AudioParameterFloat* feedback_pct {nullptr};
    juce::AudioParameterBool* bypassed {nullptr};
    juce::AudioParameterBool* mute {nullptr};
    juce::AudioParameterBool* solo {nullptr};
```

```
    void prepare(juce::dsp::ProcessSpec spec);

    void updateDelaySettings();

    void process(juce::AudioBuffer<float>& buffer, int
totalNumInputChannels);
private:
// instanciranje objekta delay klase Delay
    Delay delay;
};
```

DelayBand.cpp:

```
#include "DelayBand.h"

void DelayBand::prepare(juce::dsp::ProcessSpec spec) {
    delay.reset(spec.sampleRate);
//brisanje garbage sadržaja
    delayBuffer.clear();delayBuffer-a

// određivanje maksimalnog kašnjenja
    auto delayBufferSize = static_cast<int>(spec.sampleRate *
2);
    delayBuffer.setSize(spec.numChannels, delayBufferSize);
}

void DelayBand::updateDelaySettings()
{
    delay.setDelayAmount(delay_ms);
    delay.setFeedback(feedback_pct);
    delay.setBypassed(bypassed);
}

void DelayBand::process(juce::AudioBuffer<float>& buffer, int
totalNumInputChannels)
{
    delay.process(buffer, delayBuffer,
totalNumInputChannels);
}
```

4.1.3. PluginProcessor.h i PluginProcessor.cpp

PluginProcessor.h i PluginProcessor.cpp čine osnovnu strukturu i logiku za obradu signala unutar Multiband Delay plugin-a. U PluginProcessor.h nalazi se deklaracija klase `MBDelayAudioProcessor` koja je centralna komponenta za obradu audio signala.

```
class MBDelayAudioProcessor : public juce::AudioProcessor
```

Također nalaze se razne metode za inicijalizaciju funkcija kao što su `prepareToPlay()`, `processBlock()`, `getStateInformation()`, `setStateInformation()`.

```
void prepareToPlay(double sampleRate, int samplesPerBlock);
void processBlock(juce::AudioBuffer<float>&,
juce::MidiBuffer&);

void getStateInformation(juce::MemoryBlock& destData);

void setStateInformation(const void* data, int sizeInBytes);
```

Deklarira se klasa `juce::AudioProcessorValueTreeState` koja služi za upravljanje cjelokupnim stanjem *AudioProcessor*-a.

```
juce::AudioProcessorValueTreeState apvts{*this, nullptr,
"Parameters", createParameterLayout()};
```

U privatnom dijelu klase deklariraju se polja `delays` i `filterBuffers`. Oba polja sadrže tri objekta gdje svaki predstavlja jedan pojas audio signala (niski, srednji, visoki):

```
  std::array<DelayBand, 3> delays;
//referenca na niski, srednji i visoki pojas
  DelayBand& lowBandDelay = delays[0];
  DelayBand& midBandDelay = delays[1];
  DelayBand& highBandDelay = delays[2];

  std::array<juce::AudioBuffer<float>, 3>
filterBuffers;
```

Također se deklariraju filteri koji se koriste za dijeljenje signala te *crossover* frekvencije po kojim se dijeli signal:

```
using Filter = juce::dsp::LinkwitzRileyFilter<float>;
Filter  LP1,    AP2,
        HP1,    LP2,
                HP2;
juce::AudioParameterFloat* lowMidCrossover {nullptr};
juce::AudioParameterFloat* midHighCrossover {nullptr};
```

Datoteka PluginProcessor.cpp sadrži definicije metoda deklariranih u PluginProcessor.h.

Jedna od tih metoda `prepareToPlay()` postavlja specifikacije za obradu pomoću klase `juce::dsp::ProcessSpec` koja sadrži informacije o različitim aspektima konteksta, poput brzine uzorkovanja i broja kanala. Također priprema objekte za obradu te postavlja veličine međuspremnika. Ova metoda je izuzetno važna jer osigurava pravilno postavljanje svih parametara prije reproduciranja audio signala čime se izbjegavaju nepoželjni i potencijalno štetni zvučni artefakti.

```
void MBDelayAudioProcessor::prepareToPlay (double sampleRate,
int samplesPerBlock) {
    juce::dsp::ProcessSpec spec;
    spec.maximumBlockSize = samplesPerBlock;
    spec.numChannels = getTotalNumOutputChannels();
    spec.sampleRate = sampleRate;

    for( auto& d : delays )
        d.prepare(spec);

    LP1.prepare(spec);
    HP1.prepare(spec);
    AP2.prepare(spec);
    LP2.prepare(spec);
    HP2.prepare(spec);

    for ( auto& buffer : filterBuffers )
        buffer.setSize(spec.numChannels, samplesPerBlock);
}
```

Metoda `processBlock()` čini srž Multiband Delay plugin-a. To je glavna metoda koja je odgovorna za procesiranje audio signala. Prvo, ažurira stanje parametara te pomoću filtera dijeli audio signal u tri frekventna opsega (niski, srednji, visoki). Zatim, na svaki opseg primjenjuje *delay* efekte.

```
void MBDelayAudioProcessor::processBlock
(juce::AudioBuffer<float>& buffer, juce::MidiBuffer&
midiMessages)
{
    auto totalNumInputChannels  = getTotalNumInputChannels();
    auto totalNumOutputChannels =
getTotalNumOutputChannels();
    for (auto i = totalNumInputChannels; i <
totalNumOutputChannels; ++i)
        buffer.clear (i, 0, buffer.getNumSamples());

    updateState(); // ažurira crossover frekvencije

    splitBands(buffer); // dijeli audio signal po crossover
frekvencijama

    for (size_t i = 0; i < filterBuffers.size(); ++i)
    {
        delays[i].process(filterBuffers[i],
totalNumInputChannels); // procesiranje delay-a svakog pojasa
    }

// slijedi zbrajanje svih pojaseva u jedan buffer pomoću
lambda funkcije.
    auto numSamples = buffer.getNumSamples();
    auto numChannels = buffer.getNumChannels();
    buffer.clear();

    auto addFilterBand = [nc = numChannels, ns =
numSamples](auto& inputBuffer, const auto& source)
    {
        for (auto i = 0; i < nc; ++i)
        {
            inputBuffer.addFrom(i, 0, source, i, 0, ns);
        }
    };
```

Dvije metode koje se koriste kako bi signal bio podijeljen na tri dijela su `updateState()` koja ažurira *crossover* frekvencije te ih postavlja na odgovarajuće filtere te `splitBands()` koja primjenjuje filtere na audio signal.

```
void MBDelayAudioProcessor::updateState () {
    for (auto& d : delays)
        d.updateDelaySettings();

    auto lowMidCutoffFreq = lowMidCrossover->get();
    LP1.setCutoffFrequency(lowMidCutoffFreq);
    HP1.setCutoffFrequency(lowMidCutoffFreq);

    auto midHighCutoffFreq = midHighCrossover->get();
    AP2.setCutoffFrequency(midHighCutoffFreq);
    LP2.setCutoffFrequency(midHighCutoffFreq);
    HP2.setCutoffFrequency(midHighCutoffFreq);
}
```

U funkciji `splitBands()`, koriste se objekti `block` i `context` za filtriranje signala u različite frekvencijske pojaseve. `juce::dsp::AudioBlock<float>` je struktura podataka koja olakšava pristup i manipulaciju audio podacima. Omogućava direktan rad s blokom audio podataka unutar DSP modula.

`juce::dsp::ProcessContextReplacing<float>` objekt služi za prijenos informacija o kontekstu procesiranja. Koristi se za primjenu DSP algoritama direktno na blok audio podataka, gdje se ulazni podaci zamjenjuju izlaznim podacima nakon obrade.

```
void MBDelayAudioProcessor::splitBands(const
juce::AudioBuffer<float> &inputBuffer) {
    for ( auto& fb : filterBuffers)
    {
      // priprema filterBuffera za svaki pojas
        fb = inputBuffer;
    }

    auto fb0Block =
juce::dsp::AudioBlock<float>(filterBuffers[0]);
    auto fb1Block =
juce::dsp::AudioBlock<float>(filterBuffers[1]);
    auto fb2Block =
juce::dsp::AudioBlock<float>(filterBuffers[2]);
```

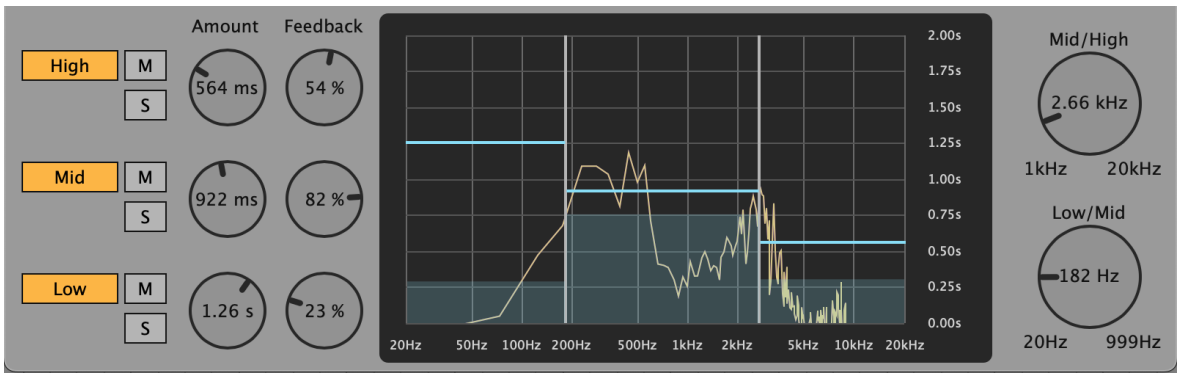
```
    auto fb0Ctx =
juce::dsp::ProcessContextReplacing<float>(fb0Block);
    auto fb1Ctx =
juce::dsp::ProcessContextReplacing<float>(fb1Block);
    auto fb2Ctx =
juce::dsp::ProcessContextReplacing<float>(fb2Block);

// Niskopropusni i svepropusni filter primjenjen na prvi blok
rezultira niskim pojasom
    LP1.process(fb0Ctx);
    AP2.process(fb0Ctx);

// Visokopropusni filter primjenjen na drugi blok
    HP1.process(fb1Ctx);
// Kopiranje drugog bloka u treći (signal bez niskog pojasa)
    filterBuffers[2] = filterBuffers[1];

// Niskopropusni filter primjenjen na drugi blok rezultira
srednjim pojasom
    LP2.process(fb1Ctx);
// Visokopropusni filter primjenjen na treći blok rezultira
visokim pojasom
    HP2.process(fb2Ctx);
}
```

Metode u pluginProcessor.h i pluginProcessor.cpp omogućuju pravilno funkcioniranje Multiband Delay plugina. Kroz inicijalizaciju, pripremu resursa, obradu audio blokova i upravljanje stanjima, ove ključne metode osiguravaju da *plugin* može efikasno podijeliti signal u različite pojaseve i primijeniti *delay* efektna svaki pojas, pružajući korisnicima fleksibilan alat za obradu zvuka.

4.2. GUI

Grafičko korisničko sučelje (*engl. Graphical User Interface*) Multiband Delay-a (prikazano slikom 4.1) podijeljeno je u tri dijela – kontrola individualnih pojaseva na lijevoj strani, spektralni analizator u sredini te kontrola *crossover* frekvencija na desnoj

strani *plugin*-a. Cilj ove faze programiranja je bio ostvariti intuitivno grafičko sučelje koje izgleda čisto te ga je zabavno koristiti.



Slika 4.1 Prikaz grafičkog sučelja Multiband Delay-a

4.2.1. PluginEditor.h i PluginEditor.cpp

Izgled grafičkog korisničkog sučelja definiran je u datotekama PluginEditor.h i PluginEditor.cpp. U PluginEditor.h definirani su dijelovi korisničkog sučelja u obliku zasebnih klasa, dok su u PluginEditor.cpp definirane implementacije tih dijelova.

PluginEditor.h:

```
...
BandControls bandControls {audioProcessor.apvts};
GlobalControls globalControls {audioProcessor.apvts};
SpectrumAnalyzer analyzer {audioProcessor};
```

PluginEditor.cpp:

```
...
MBDelayAudioProcessorEditor::MBDelayAudioProcessorEditor
(MBDelayAudioProcessor& p)
    : AudioProcessorEditor (&p), audioProcessor (p)
{
    setLookAndFeel(&lnf);
    addAndMakeVisible(analyzer);
    addAndMakeVisible(globalControls);
    addAndMakeVisible(bandControls);

    setSize (800, 250); // postavljanje veličine prozora
}
```

```
MBDelayAudioProcessorEditor::~MBDelayAudioProcessorEditor()
{
    setLookAndFeel(nullptr);
}

void MBDelayAudioProcessorEditor::resized()
{   // određivanje veličine dijelova plugin-a
    bandControls.setBounds(bounds.removeFromLeft(250));
    globalControls.setBounds(bounds.removeFromRight(120));
    analyzer.setBounds(bounds);
}
```

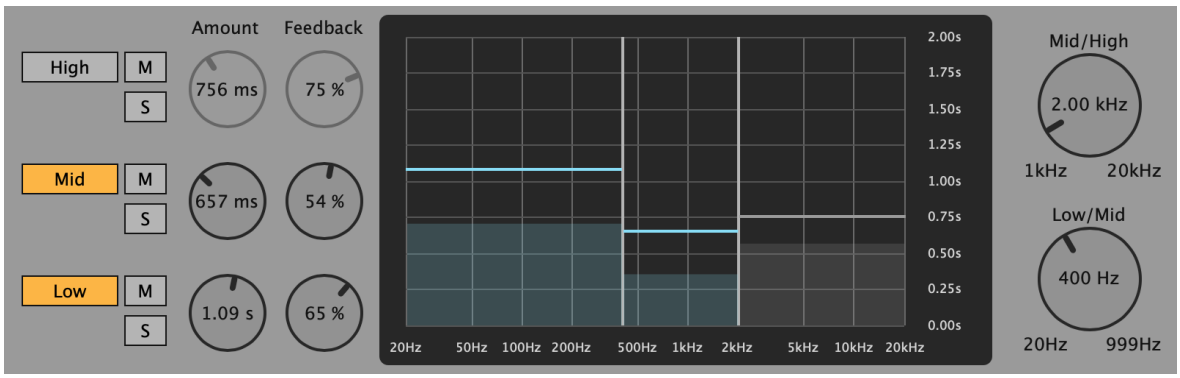
4.2.2. Kontrole individualnih pojaseva

Datoteke BandControls.h i BandControls.cpp definiraju i implementiraju klasu `BandControls` te ponašanje kontrola za pojedinačne frekvencijske pojaseve Multiband Delay-a. Svaki pojas ima niz kontrola koje omogućuju detaljno podešavanje efekata, pružajući korisnicima fleksibilnost i preciznost u manipulaciji zvukom.

Ključne kontrole za svaki pojas:

1. *Bypass*

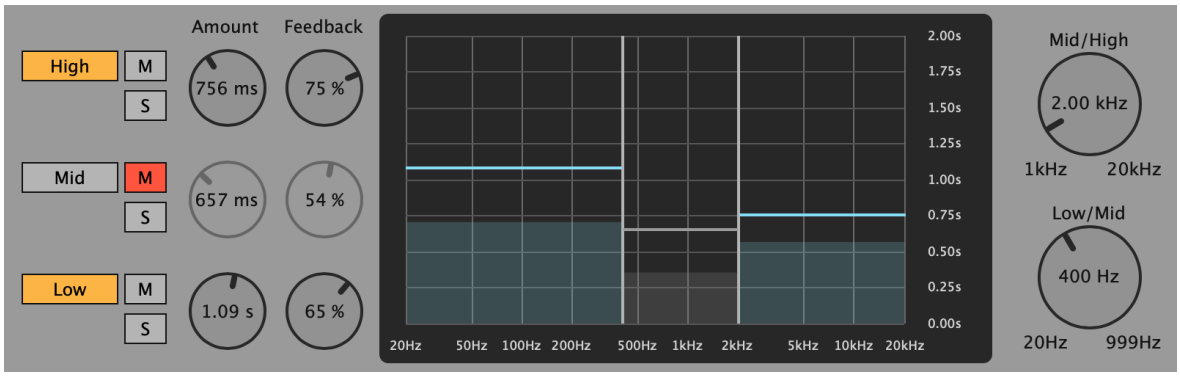
   Gumb *Bypass* omogućuje korisniku da zaobiđe *delay* efekt za određeni pojas. Kada je *Bypass* aktiviran, signal prolazi kroz pojas bez primjene *delay* efekta. Gumb to stanje reflektira sivom bojom. Također, kontrole *delay*-a za taj pojas postanu sive i blokirane. Kada *Bypass* nije aktiviran, gumb ima žutu boju na kojoj piše ime pojasa kojem pripada.



Slika 4.2 Prikaz grafičkog sučelja sa zaobiđenim visokim pojasom

2. *Mute*

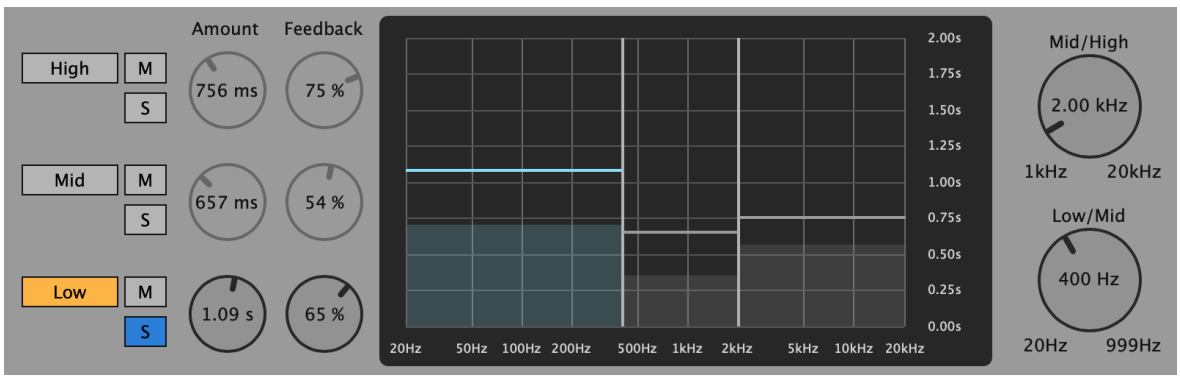
   Gumb *Mute* utišava signal za određeni pojas. Kada je *Mute* aktiviran, signal tog pojasa neće biti čujan, a gumb će poprimiti crvenu boju. Osim toga, sve ostale kontrole za taj pojas bit će blokirane i postat će sive kako bi se izbjegla zbunjenost korisnika i jasno označilo da je pojas utišan.



Slika 4.3 Prikaz grafičkog sučelja sa utišanim srednjim pojesom

3. *Solo*

   Gumb *Solo* izolira signal iz određenog pojasa. Kada je *solo* aktiviran, samo signal iz tog pojasa će biti čujan dok će svi ostali pojasevi biti utišani. Samo jedan *Solo* gumb može biti aktivan odjednom. Također, gumb će poprimiti plavu boju dok će ostale kontrole drugih pojaseva biti sive kako bi se jasno označilo da su utišani.



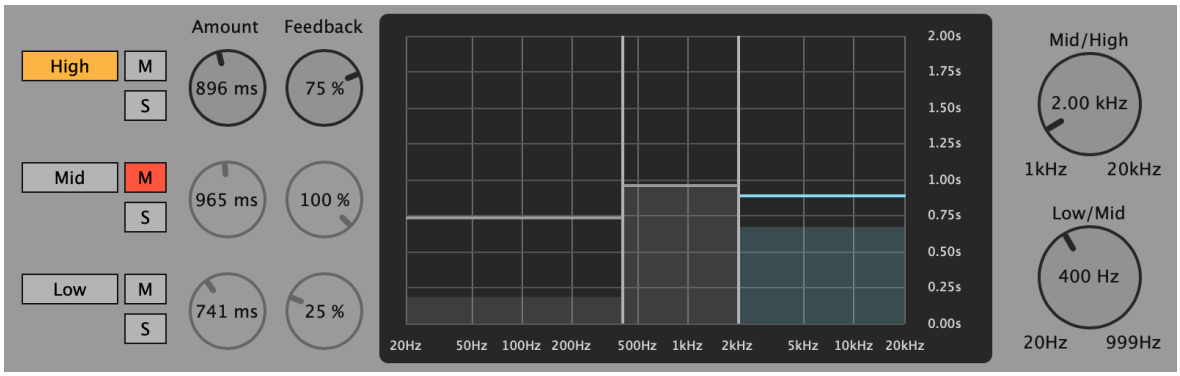
Slika 4.4 Prikaz grafičkog sučelja sa izoliranim niskim pojasom

4. *Amount*

   Kontrola *Amount* podešava količinu kašnjenja (*delay amount*) za određeni pojas. Korisnik može precizno postaviti vrijeme kašnjenja, što određuje koliko će dugo trajati vrijeme prije nego što se odgođeni signal reproducira. Ova kontrola je aktivna samo ako pojas na kojeg se odnosi nije zaobiđen ili utišan. Spektralni analizator preslikava stanje ove kontrole u obliku plave ili sive horizontalne linije na grafu ovisno o stanju kontrole.

5. *Feedback*

   Kontrola *Feedback* određuje koeficijent povratne veze (*feedback*) za određeni pojas. Povratna veza kontrolira koliko se izlaznog signala vraća natrag u ulaz *delay* efekta, stvarajući višestruke odjeke i bogatiji zvuk. Isto kao *Amount*, ova kontrola je aktivna samo ako pojas na kojeg se odnosi nije zaobiđen ili utišan. Spektralni analizator preslikava stanje ove kontrole u obliku plavog ili sivog pravokutnika na grafu ovisno o stanju kontrole.



Slika 4.5 Prikaz grafičkog sučelja s utišanim srednjim pojasom te zaobiđenim niskim pojasom. Količina kašnjenja visokog pojasa postavljena je na 896 ms dok je *feedback* postavljen na 75%.

Funkcionalnost grafičkog korisničkog sučelja za kontrolu bendova implementirana je u datotekama BandControls.cpp i BandControls.h.

U BandControls.h deklarirani su svi gumbi i kontrole te se instancira objekt `AudioProcessorValueTreeState` koji omogućuje dohvaćanje stanja parametara. Ova datoteka također sadrži metode za iscrtavanje `paint()`, prilagodbu veličine `resized()` i rukovanje klikovima na gumbima `buttonClicked()`.

Ključni objekti u deklarirani u BandControls.h su:

- `apvts`: Referenca na `AudioProcessorValueTreeState` koji upravlja stanjem parametara.
- `lowDelayDial`, `midDelayDial`, `highDelayDial`: Klizači za podešavanje kašnjenja za niske, srednje i visoke frekvencije.
- `lowFeedbackDial`, `midFeedbackDial`, `highFeedbackDial`: Klizači za podešavanje povratne veze za niske, srednje i visoke frekvencije.
- `amtLabel`, `fbLabel`: Natpisi za označavanje kontrola količine (*amount*) i povratne veze (*feedback*).
- `lowBypassButton`, `midBypassButton`, `highBypassButton`: Gumbi *bypass* za niski, srednji i visoki pojas.
- `lowSoloButton`, `midSoloButton`, `highSoloButton`: Gumbi za *solo* način rada za niski, srednji i visoki pojas.
- `lowMuteButton`, `midMuteButton`, `highMuteButton`: Gumbi za gašenje (*mute*) niskog, srednjeg i visokog pojasa.

U BandControls.cpp definirane su implementacije metoda deklariranih u BandControls.h. Ove metode upravljaju interakcijama korisnika s grafičkim sučeljem i omogućuju promjene u realnom vremenu na osnovu korisničkih ulaza.

```
BandControls.h:
            ...
                void paint(juce::Graphics& g) override;
                void resized() override;

                void buttonClicked(juce::Button* button) override;
            private:
```

```
        juce::AudioProcessorValueTreeState& apvts;

        RotarySliderWithLabels lowDelayDial,
                               midDelayDial,
                               highDelayDial,
                               lowFeedbackDial,
                               midFeedbackDial,
                               highFeedbackDial;

    ...
        juce::Label amtLabel {"", "Amount"};
        juce::Label fbLabel {"", "Feedback"};

        juce::ToggleButton lowBypassButton,
                           midBypassButton,
                           highBypassButton,
                           lowSoloButton,
                           midSoloButton,
                           highSoloButton,
                           lowMuteButton,
                           midMuteButton,
                           highMuteButton;
        ...
```

Metoda `paint()` odgovorna je za iscrtavanje sučelja, dok metoda `resized()` postavlja pozicije i veličine svih kontrola. Metoda `buttonClicked()` upravlja klikovima na gumbima i izvršava odgovarajuće radnje na temelju korisničkih interakcija.

```
paint():
        void BandControls::paint(juce::Graphics &g) {
            auto bounds = getLocalBounds();
            drawModuleBackground(g, bounds);
        }
resized():
        ...
              FlexBox flexBox;
              flexBox.flexDirection = FlexBox::Direction::column;
```

```
                flexBox.flexWrap = FlexBox::Wrap::noWrap;
                ...
                FlexBox flexBoxLow;
                flexBoxLow.flexDirection = FlexBox::Direction::row;
                flexBoxLow.flexWrap = FlexBox::Wrap::noWrap;

                auto endCap = FlexItem().withWidth(10);
                auto spacer = FlexItem().withWidth(2);

                flexBoxLow.items.add(endCap);
                flexBoxLow.items.add(FlexItem(lowBypassButton).withWidt
                h(70.f).withHeight(25.f));

                flexBoxLow.items.add(FlexItem(lowBandButtonControlBox).
                withWidth(33.f));


                flexBoxLow.items.add(FlexItem(lowBandDialControlBox).wi
                thWidth(132.f).withHeight(80.f));
                ...
                flexBox.items.add(FlexItem(flexBoxText).withHeight(25.f
                ));
                flexBox.items.add(FlexItem(flexBoxHigh).withHeight(80.f
                ));
                flexBox.items.add(FlexItem(flexBoxMid).withHeight(80.f)
                );
                flexBox.items.add(FlexItem(flexBoxLow).withHeight(80.f)
                );
                flexBox.performLayout(bounds);
            }

buttonClicked():
            void BandControls::buttonClicked(juce::Button *button)
            {
                updateSliderEnablements();
                updateSoloMuteBypassToggleStates(*button);
            }
```

4.2.3. Globalne kontrole

Desna strana GUI-a sastoji se od dva klizača (*dial*) koji su odgovorni za kontrolu *crossover* frekvencija filtera. Ove kontrole omogućuju korisniku da precizno podešava prijelazne frekvencije između niskih/srednjih i srednjih/visokih frekvencijskih pojaseva.

Kontrole frekvencija su raspoređene vertikalno, jedan iznad drugoga, s razmakom između njih kako bi se osigurala preglednost i lakoća korištenja.

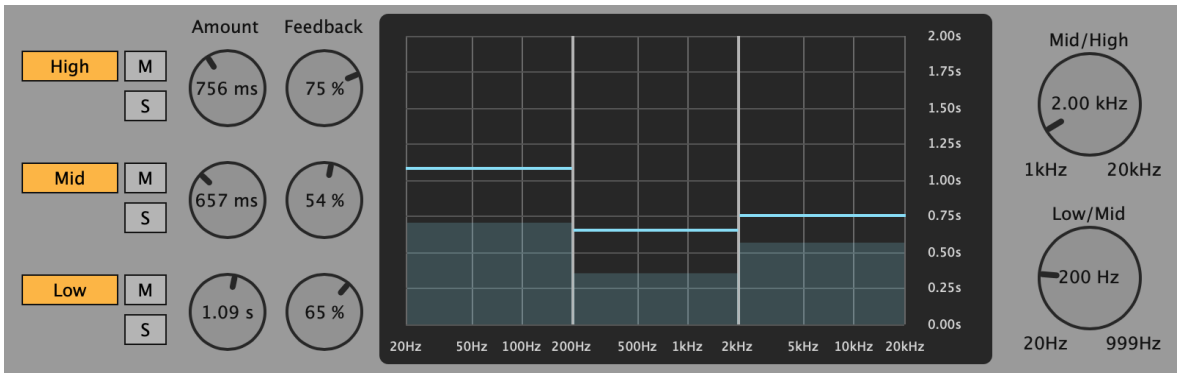
Ovakav dizajn omogućuje korisnicima intuitivnu i preciznu kontrolu frekvencijskih pojaseva što je ključno za postizanje željenih zvučnih efekata i optimizaciju audio performansi.

```
GlobalControls.h:
            ...
            private:
                using RSWL = RotarySliderWithLabels;
                std::unique_ptr<RSWL> lowMidDial, midHighDial;
            ...
```

U konstruktoru se inicijaliziraju dvije instance klase `RotarySliderWithLabels` za kontrolu *crossover* frekvencija između niskih i srednjih te srednjih i visokih frekvencija, te ih povezuje s odgovarajućim parametrima u `AudioProcessorValueTreeState`.

```
GlobalControls.cpp:
            ...
            GlobalControls::GlobalControls(juce::AudioProcessorValueTreeS
      tate& apvts)
            {
                using namespace Params;
                const auto& params = GetParams();

                auto getParamHelper = [&params, &apvts](const auto& name)
            -> auto&
                {
                    return getParam(apvts, params, name);
                };
```

```
        auto& lowMidParam =
    getParamHelper(Names::Low_Mid_Crossover_Freq);
        auto& midHighParam =
    getParamHelper(Names::Mid_High_Crossover_Freq);

        lowMidDial = std::make_unique<RSWL>(&lowMidParam, "Hz",
    "Low/Mid");
midHighDial = std::make_unique<RSWL>(&midHighParam, "Hz", "Mid/High")
```



Slika 4.6 Grafičko sučelje Multiband Delay-a s postavljenim *crossover* frekvencijama na 200 Hz i na 2 kHz

4.2.4. Spektralni analizator

Spektralni analizator je središnja komponenta Multiband Delay *plugin*-a koji vizualno predstavlja globalne kontrole *crossover* frekvencija te stanje frekvencijskih pojaseva i njihovih parametara kašnjenja na grafu. Tijekom reprodukcije zvučnog signala, analizator prikazuje signal u frekvencijskom spektru, omogućujući korisnicima da vide kako se frekvencije mijenjaju u realnom vremenu.

Datoteka `SpectrumAnalyzer.h` definira klasu `SpectrumAnalyzer` koja se koristi za vizualizaciju frekvencijskog spektra audio signala unutar GUI-a Multiband Delay *plugin*-a. Ova klasa nasljeđuje nekoliko JUCE klasa kako bi implementirala potrebnu funkcionalnost za crtanje spektralnog analizatora i ažuriranje njegovog prikaza u stvarnom vremenu.

```
struct SpectrumAnalyzer: juce::Component,
juce::AudioProcessorParameter::Listener, juce::Timer
```

```
{
...
```

Klasa također sadrži privatne članove za pohranu referenci na procesor i parametre potrebne za analizu:

```
...
private:
MBDelayAudioProcessor& audioProcessor;
...
juce::AudioParameterFloat* lowMidXoverParam { nullptr };
juce::AudioParameterFloat* midHighXoverParam { nullptr };
juce::AudioParameterFloat* lowDelayAmtParam { nullptr };
juce::AudioParameterFloat* midDelayAmtParam { nullptr };
juce::AudioParameterFloat* highDelayAmtParam { nullptr };
juce::AudioParameterFloat* lowFeedbackAmtParam { nullptr };
juce::AudioParameterFloat* midFeedbackAmtParam { nullptr };
juce::AudioParameterFloat* highFeedbackAmtParam { nullptr };
juce::AudioParameterBool* lowBandOnParam { nullptr };
juce::AudioParameterBool* midBandOnParam { nullptr };
juce::AudioParameterBool* highBandOnParam { nullptr };
```

Datoteka SpectrumAnalyzer.cpp implementira funkcionalnost definiranu u SpectrumAnalyzer.h, omogućujući crtanje i ažuriranje spektralnog analizatora u stvarnom vremenu. Ključne funkcije uključuju `drawBackgroundGrid()`, `drawTextLabels()`, `drawCrossovers()` i `drawFFTAnalysis()`, koje zajedno omogućuju cjelovitu vizualizaciju frekvencijskog sadržaja i parametara signala.

`drawBackgroundGrid()` crta vertikalne linije koje predstavljaju frekvencijske pojaseve i horizontalne linije koje označavaju količinu kašnjenja *delay*-a na grafu, pružajući vizualne referentne točke za analizu signala.

`drawBackgroundsGrid()`:

```
...
for( auto x : xs ) {
    g.drawVerticalLine(x, top, bottom);
}

for( auto gDb : gain ) {
    auto y = jmap(gDb, NEGATIVE_INFINITY, 12.f,
float(bottom), float(top));
```

```
            g.drawHorizontalLine(y, left, right);
        }
        ...
```

`drawTextLabels()` prikazuje tekstualne oznake na frekvencijskoj osi i osi kašnjenja, čime korisnicima daje jasne vrijednosti frekvencija i kašnjenja.

```
drawTextLabels():
        ...
        for( int i = 0; i < freqs.size(); ++i ) {
            auto f = freqs[i];
            auto x = xs[i];
            if( f > 999.f ) {
                f /= 1000.f;
            }
            str << f << "Hz";
            g.drawFittedText(str, r, juce::Justification::centred,
        1);
        }

        for( float dAmt : delayAmt ) {
            auto y = jmap(dAmt, 0.f, 2.00f, float(bottom),
        float(top));
            str << dAmt << "s";
            g.drawFittedText(str, r,
        juce::Justification::centredLeft, 1);
        ...
```

`drawCrossovers()` crta vertikalne linije koje predstavljaju *crossover* frekvencije između različitih frekvencijskih pojaseva, horizontalne linije koje prikazuju razine kašnjenja te pravokutnike koji prikazuju razinu *feedback*-a za svaki pojas.

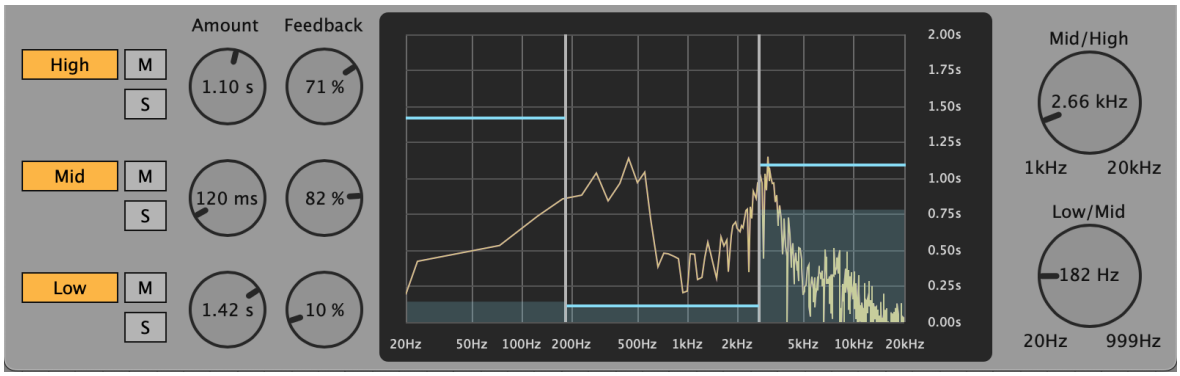
```
drawCrossovers():
        ...
        auto lowMidX = mapX(lowMidXoverParam->get());
        g.drawVerticalLine(lowMidX, top, bottom);

        auto midHighX = mapX(midHighXoverParam->get());
        g.drawVerticalLine(midHighX, top, bottom);

        auto lowMapY = mapY(lowDelayAmtParam->get());
        g.drawHorizontalLine(lowMapY, left, lowMidX);
        ...
```

```
auto lowFbMapY = jmap(lowFeedbackAmtParam->get(), 0.f, 100.f,
float(bottom), lowMapY);
auto midFbMapY = jmap(midFeedbackAmtParam->get(), 0.f, 100.f,
float(bottom), midMapY);
auto highFbMapY = jmap(highFeedbackAmtParam->get(), 0.f,
100.f, float(bottom), highMapY);

g.setColour(lowBandOnParam->get() ?
MyColours::lightBlue.withAlpha(0.2f) :
MyColours::grey.withAlpha(0.2f));
g.fillRect(Rectangle<float>::leftTopRightBottom(left + 1,
lowFbMapY, lowMidX, bottom));
g.setColour(midBandOnParam->get() ?
MyColours::lightBlue.withAlpha(0.2f) :
MyColours::grey.withAlpha(0.2f));
g.fillRect(Rectangle<float>::leftTopRightBottom(lowMidX + 1,
midFbMapY, midHighX, bottom));
g.setColour(highBandOnParam->get() ?
MyColours::lightBlue.withAlpha(0.2f) :
MyColours::grey.withAlpha(0.2f));
g.fillRect(Rectangle<float>::leftTopRightBottom(midHighX + 1,
highFbMapY, right - 1, bottom));
}
```

`drawFFTAnalysis()` crta analizu FFT (*engl. Fast Fourier Transform*) za lijevi i desni kanal zvučnog signala prikazujući njihov frekvencijski sadržaj na grafu.

`drawFFTAnalysis()`:

```
...
auto leftChannelFFTPath = leftPathProducer.getPath();
leftChannelFFTPath.applyTransform(AffineTransform().translati
on(responseArea.getX(), responseArea.getY()));
g.strokePath(leftChannelFFTPath, PathStrokeType(1.f));

auto rightChannelFFTPath = rightPathProducer.getPath();
rightChannelFFTPath.applyTransform(AffineTransform().translat
ion(responseArea.getX(), responseArea.getY()));
g.strokePath(rightChannelFFTPath, PathStrokeType(1.f));
...
```



Slika 4.7 Prikaz spektralnog analizator tijekom reprodukcije signala

Spektralni analizator pruža detaljan i intuitivan prikaz audio signala, omogućujući korisnicima da precizno prate i podešavaju frekvencijske pojaseve i njihove parametre unutar *plugin*-a.

Zaključak

Multiband Delay je napredni audio efekt dizajniran za pružanje većeg stupnja kontrole nad *delay* efektom primijenjenim na različite frekvencijske pojaseve ulaznog signala. Umjesto da cijeli signal prolazi kroz jedan *delay*, Multiband Delay ga dijeli na tri zasebna frekvencijska opsega: niski, srednji i visoki. Svaki od tih opsega može se individualno obraditi vlastitim *delay* parametrima omogućujući preciznu i detaljnu kontrolu nad zvukom.

Podjela signala na tri frekvencijska opsega postiže se korištenjem Linkwitz-Riley filtera koji osiguravaju glatku i neprimjetnu tranziciju između opsega bez faznih izobličenja. Nakon podjele, svaki pojas prolazi kroz vlastiti *delay* modul. To omogućuje postavke količine kašnjenja (*delay amount*) te povratne veze (*feedback*) za svaki od pojasa.

Osim osnovnih parametara, Multiband Delay nudi dodatne kontrole kao što su *Bypass*, *Mute* i *Solo* za svaki pojas. *Bypass* omogućuje korisnicima da privremeno isključe *delay* efekt za određeni pojas, *Mute* utišava pojas, a *Solo* omogućuje slušanje samo jednog pojasa, što je korisno za preciznu prilagodbu parametara.

Grafičko korisničko sučelje (GUI) Multiband Delay-a dizajnirano je tako da bude intuitivno i lako za korištenje. Sve kontrole su jasno označene i raspoređene na način koji omogućuje brz i jednostavan pristup svim funkcijama. Korisnici mogu vizualno pratiti promjene u realnom vremenu što olakšava proces prilagodbe i eksperimentiranja s različitim postavkama.

Pisanje ovog završnog rada rezultiralo je postizanjem početno postavljenih ciljeva. Prvo, stečeno je razumijevanje i implementacija osnovnih principa digitalne obrade signala (DSP) koji stoje iza *delay* efekta. Drugo, uspješno su primijenjeni Linkwitz-Riley filteri za preciznu frekvencijsku podjelu signala omogućujući glatku tranziciju između različitih frekvencijskih opsega bez faznih izobličenja. Konačno, razvijeno je korisnički orijentirano grafičko sučelje (GUI) koje omogućuje intuitivnu kontrolu parametara efekta čineći Multiband Delay alatom koji je istovremeno moćan i jednostavan za korištenje.

Literatura

[1] Pirkle, W. C. *Designing Audio Effect Plugins in C++*. 2. izdanje. New York and London: Routledge, 2019.

[2] *Linkwitz–Riley filter*, Wikipedia, (2024, svibanj). Poveznica: https://en.wikipedia.org/wiki/Linkwitz_Riley_filter; pristupljeno 14. lipnja 2024.

[3] JUCE, Poveznica: https://juce.com/, pristupljeno 12. lipnja 2024.

[4] The Audio Programmer, *JUCE Tutorial 16 - Creating a Basic Delay Effect*, YouTube, Poveznica: https://www.youtube.com/watch?v=eA5Mhbric6Y, pristupljeno 11. lipnja 2024.

[5] The Audio Programmer, *JUCE Tutorial 17 - Creating Delay Feedback*, YouTube, Poveznica: https://www.youtube.com/watch?v=eVg7EVmWBsE, pristupljeno 11. lipnja 2024.

Sažetak

Multiband Delay

Multiband Delay je audio *plugin* koji dijeli ulazni signal na tri pojasa (niski, srednji i visoki) i primjenjuje individualne *delay* parametre za svaki pojas. Koristeći Linkwitz-Riley crossover filtere osigurava se podjela ulaznog signala bez faznih izobličenja. Metoda omogućuje preciznu kontrolu nad zvukom, uključujući opcije *Bypass*, *Mute* i *Solo* za svaki opseg te kontrole za količinu kašnjenja i povratne veze. Grafičko korisničko sučelje je intuitivno i jednostavno za korištenje. Završni rad se fokusira na razumijevanje i implementaciju DSP principa, preciznu frekvencijsku podjelu te razvoj intuitivnog korisnički orijentiranog sučelja što rezultira moćnim alatom za kreativnu obradu zvuka.

Ključne riječi:

Multiband Delay, DSP (Digitalna obrada signala), grafičko korisničko sučelje (GUI), delay, filteri, kreativna obrada zvuka

Summary

Multiband Delay

Multiband Delay is an audio plugin that splits the input signal into three bands (low, mid, and high) and applies individual delay parameters to each band. Using Linkwitz-Riley crossover filters, it ensures the input signal is divided without phase distortion. This method allows for precise control over the sound, including options for *Bypass*, *Mute*, and *Solo* for each band, as well as controls for delay amount and feedback. The graphical user interface is intuitive and easy to use. The final project focuses on understanding and implementing DSP principles, achieving precise frequency splitting, and developing an intuitive user-oriented interface, resulting in a powerful tool for creative sound processing.

Keywords:

Multiband Delay, DSP (Digital Signal Processing), graphical user interface (GUI), delay, filters, creative sound processing

Skraćenice

DSP	*Digital Signal Processing*	digitalna obrada signala
GUI	*Graphical User Interface*	grafičko korisničko sučelje
API	*Application Programming Interface*	aplikacijsko programsko sučelje
DDL	*Digital Delay Line*	digitalna linija kašnjenja
IIR	*Infinite Impulse Response*	beskonačni odziv impulsa
FFT	*Fast Fourier Transform*	brza Fourierova tranformacija