Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Bakalářská práce

Návrh a implementace pluginu pro hudební produkční software

Lukáš Schöbel

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Lukáš Schöbel

Informatika

Název práce

Návrh a implementace pluginu pro hudební produkční software

Název anglicky

Design and implementation of a plugin for music production software

---

Cíle práce

Cílem práce je naprogramovat funkční VST (.dll) plugin pro software určený ke skládání a editaci hudby, jako například FL Studio, Ableton, apod. Výsledkem práce bude funkční ekvalizér, který se bude ovládat skrze grafické rozhraní. Ekvalizér je nástroj, který se v hudebním odvětví využívá k analýze a práci se zvukem, především díky grafickému znázornění frekvenčního spektra zvuku.

Dílčí cíle:
- Navrhnout kód pluginu – jak ekvalizér bude fungovat
- Navrhnout GUI pluginu – jak ekvalizér bude vypadat
- Užití ekvalizéru v praxi

Metodika

Práce sestává ze dvou částí, teoretické a praktické – vlastního řešení. V rámci zpracování teoretické části práce bude provedeno studium souvisejících odborných informačních zdrojů. Na jeho základě budou formulována teoretická východiska pro vlastní řešení.

Praktická část práce bude vycházet z teoretických východisek a v jejím rámci bude proveden návrh a implementace samotného pluginu. Pro vývoj bude využit jazyk C++ a multiplatformní aplikační framework JUCE. Vývoj bude primárně probíhat ve vývojovém prostředí Visual Studio Code. Testování aplikace bude probíhat v hudebním produkčním systému FL Studio 20. Výsledný plugin bude otestován, budou shrnuty zkušenosti a poznatky získané během jeho implementace a budou navržena případná další rozšíření do budoucna.

Doporučený rozsah práce

35-40 stran

Klíčová slova

vst, ekvalizér, plugin, fl studio, programování, c++, juce

Doporučené zdroje informací

Lippman, S., Lajoie, J., Moo, B., & Safari, an O'Reilly Media Company. (2012). C++ Primer, Fifth Edition.
Pirkle, W. C. (2013). Designing audio effect plug-ins in C++ with digital audio signal processing theory.
Pirkle, W., & Safari, an O'Reilly Media Company. (2019). Designing Audio Effect Plugins in C++, 2nd Edition.
Robinson, M. (2013). Getting Started with JUCE. Birmingham: Packt Publishing.

Předběžný termín obhajoby

2022/23 LS – PEF

Vedoucí práce

Ing. Jiří Brožek, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 31. 10. 2022

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 24. 11. 2022

doc. Ing. Tomáš Šubrt, Ph.D.

Děkan

V Praze dne 28. 02. 2023

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci "Návrh a implementace pluginu pro hudební produkční software" jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 15.3.2023 _____

Poděkování

Rád bych touto cestou poděkoval Ing. Jiřímu Brožkovi, Ph.D. za odborné vedení a koordinování práce, vstřícnost a možnost spojit obor studia s mým volnočasovým koníčkem – hudební produkcí.

Návrh a implementace pluginu pro hudební produkční software

Abstrakt

Bakalářská práce se zabývá návrhem a implementací funkčního VST pluginu pro hudební software, jako je například FL Studio a Ableton. Práce se zaměřuje na vývoj ekvalizéru, který je možné ovládat prostřednictvím grafického rozhraní. Pro dosažení tohoto cíle je práce rozdělena do dvou částí: teoretické a praktické.

Teoretická část zahrnuje související odborné informace s cílem vysvětlit charakteristiku zpracování zvuku počítačem, seznámit čtenáře se základními vlastnostmi zvuku, vysvětlit objektově orientovaný přístup k programování a tedy formulovat teoretické podklady pro vlastní řešení.

Praktická část vychází z teoretických podkladů. Zahrnuje návrh a implementaci samotného zásuvného modulu. Vývoj probíhá především ve vývojovém prostředí Visual Studio s využitím jazyka C++ a multiplatformního aplikačního frameworku JUCE. Poznatky a zkušenosti získané při implementaci ekvalizéru jsou shrnuty v kapitole Výsledky a diskuze, kde jsou také navržena možná budoucí rozšíření zásuvného modulu.

Klíčová slova: JUCE framework, ekvalizér, C++, vývoj, zvuk, VST, zásuvný modul, digitální zpracování zvuku, zvuková vlna

Design and implementation of a plugin for music production software

Abstract

The bachelor's thesis focuses on the development and implementation of a functional VST plug-in for FL Studio and Ableton. An equalizer controlled by a graphical user interface is the subject of the thesis. In order to accomplish this goal, the thesis consists of two parts: theoretical and practical.

The theoretical part includes related technical information in order to explain the characteristics of sound processing by computer, to introduce the reader to the basic properties of sound, to explain the object-oriented approach to programming and, thus, formulate the theoretical basis for the actual solution.

Based on the theoretical background, the practical part involves designing and implementing the plugin. Visual Studio is used to develop the application, along with the cross-platform application framework JUCE. The Results and Discussion chapter summarizes the knowledge and experience gained from the implementation of the equalizer, along with possible extensions.

Keywords: JUCE framework, equalizer, C++, development, sound, VST, plugin, digital audio processing, sound wave

Obsah

1 Úvod

Odvětví hudební produkce je plné různých zásuvných modulů pro úpravu zvuku, a najde se také pestrá škála zaměření těchto pluginů, od ozvěny a dozvuk, přes autokorekci hlasu (tzv. autotune), až po harmonizaci a saturaci. Přesto však existují takové typy, bez kterých se hudební producenti a další zvukoví inženýři zkrátka neobejdou. Mezi takové patří ekvalizér.

Ekvalizér je nástroj sloužící k úpravě hlasitosti různých frekvenčních pásem v rámci zvukového signálu. Existuje několik typů ekvalizéru, například analogové ekvalizéry, které mimo úpravy hlasitosti disponují také zabarvením zvuku, díky čemuž se odlišují od ostatních. Takové zabarvení je často nežádoucí, a právě proto se tato práce zabývá vývojem digitálního ekvalizéru, který mimo úpravu samotným nástrojem nechává zvuk nepozměněný. Oproti analogovým ekvalizérům může digitální ekvalizér mít zabudován analyzátor frekvenčního spektra, který monitoruje vstup zvuku a převádí ho do grafické podoby, což producentům a zvukovým inženýrům usnadňuje práci v mnoha ohledech.

Výsledkem této práce je digitální ekvalizér s frekvenčním spektrem, třemi hlavními parametry, několika vedlejšími parametry a vlastnostmi, které se při práci se zvukem běžně využívají.

2 Cíl práce a metodika

2.1 Cíl práce

Cílem práce je naprogramovat funkční VST plugin pro software určený ke skládání a editaci hudby, jako například FL Studio, Ableton, apod. Výsledkem práce bude funkční ekvalizér, který se bude ovládat skrze grafické rozhraní. Ekvalizér je nástroj, který se v hudebním odvětví využívá k analýze a práci se zvukem, především díky grafickému znázornění frekvenčního spektra zvuku.

Dílčí cíle:

- Návrh kódu pluginu – funkční část ekvalizéru
- Návrh grafického rozhraní – vizuální stránka ekvalizéru
- Použití ekvalizéru v praxi

2.2 Metodika

Práce sestává ze dvou částí, teoretické a praktické – vlastního řešení. V rámci zpracování teoretické části práce bude provedeno studium souvisejících odborných informačních zdrojů. Na jeho základě budou formulována teoretická východiska pro vlastní řešení.

Praktická část práce bude vycházet z teoretických východisek a v jejím rámci bude proveden návrh a implementace samotného pluginu. Pro vývoj bude využit jazyk C++ a multiplatformní aplikační framework JUCE. Vývoj bude primárně probíhat ve vývojovém prostředí Visual Studio. Testování aplikace bude probíhat v hudebním produkčním systému FL Studio 20. Výsledný plugin bude otestován, budou shrnuty zkušenosti a poznatky získané během jeho implementace a budou navržena případná další rozšíření do budoucna.

3 Teoretická východiska

Pro dosažení cílů této práce je zapotřebí obeznámit se s technologiemi, které se pro vývoj využívají. Pro lepší pochopení problematiky je vhodné uvést také základní charakteristiky a vlastnosti zvuku a to, jak je zvuk lidským uchem vnímán a jak počítač zvuk zpracovává.

3.1 Programovací jazyk C++

Jazyk C++ je multiparadigmatický jazyk, ale primárně je považován za objektově orientovaný. Tento jazyk poskytuje také podporu objektového nebo procedurálního programování. Díky těmto vlastnostem je člověk schopen poskytnout řešení, které nejlépe odpovídá danému problému. Vzniká tím ale jistá nevýhoda – větší a složitější programovací jazyk (Lippman, a další, 1998 str. 13).

3.1.1 Objektově-orientovaný jazyk

Objektově orientované programování je programovací paradigma, které je dnes základem mnoha programovacích jazyků. Objektově orientované programování spočívá v modelování systému jakožto kolekce objektů, kde každý objekt reprezentuje konkrétní aspekt systému. Objekty mohou obsahovat jak metody, tak data. Objekt poskytuje rozhraní dostupné ostatním částem kódu, které jej chtějí používat, zároveň si ale udržuje vlastní soukromý, vnitřní stav. Další části systému nezajímá, co se uvnitř objektu děje (Mozilla Contributors, 2021).

Objektově orientované programování rozšiřuje abstraktní datové typy pomocí mechanismů dědičnosti (opakované použití existující implementace) a dynamické vazby (opakované použití existujícího veřejného rozhraní) (Lippman, a další, 1998 str. 13).

3.1.2 Třídy a instance

Když se modeluje problém pomocí objektů v objektově orientovaném programování, vytváří se abstraktní definice představující typy objektů, které systém má obsahovat. Pokud se například modeluje společnost, mohou existovat objekty představující vedoucí týmů. Každý vedoucí má některé společné vlastnosti: všichni mají jméno, plat, přidělený tým, apod. Dále

může každý vedoucí dělat určité věci: všichni mohou hodnotit práci a mohou se například seznámit s novým zaměstnancem při nástupu. Vedoucí by tedy mohl být třídou. V definici třídy jsou uvedeny údaje a metody, které má každý vedoucí k dispozici (Mozilla Contributors, 2021).

3.1.3 Dědičnost

Když nějaká třída dědí z jiné třídy, dědící třída má přístup k datovým členům takové třídy a k funkcím jejích členů. Vývojář se tak zbaví nutnosti udržovat dvě kopie kódu. Nová třída musí poskytovat pouze datové členy a funkce nezbytné k implementaci její dodatečné sémantiky.

V jazyce C++ se o třídě, z níž se dědí, hovoří jako o bázové třídě. O nové třídě se hovoří, že je odvozena od bázové třídy. Říká se jí odvozená třída nebo podtyp bázové třídy. Podtyp sdílí se svou bázovou třídou společné rozhraní, tedy společnou sadu operací. Sdílení společného rozhraní umožňuje, aby bázová třída a odvozená třída byly používány v rámci programu zaměnitelně, aniž by bylo nutné se zajímat o skutečný typ objektu. V jistém smyslu společné rozhraní zapouzdřuje typově specifické detaily jednotlivých podtypů. Vztah bázových tříd a tříd odvozených tvoří hierarchii dědičnosti (Lippman, a další, 1998 str. 52).

3.1.4 JUCE Framework

JUCE je open-source[1] multiplatformní aplikační framework v jazyce C++ pro vytváření desktopových a mobilních aplikací, zvukových zásuvných modulů VST, VST3, AU, AUv3, AAX a LV2 včetně jejich hostitelů. JUCE lze integrovat do stávajících projektů, nebo jej lze použít jako nástroj pro generování projektů prostřednictvím Projucer, který podporuje export projektů pro Xcode, Visual Studio, Android Studio, Code::Blocks a Linux Makefiles. Obsahuje také editor zdrojového kódu.

[1] open-source – volně přístupný, modifikovatelný a distribuovatelný

JUCE rozhraní nabízí širokou paletu tříd, zaměřující se na prvky uživatelského rozhraní, grafiku, zvuk, vícevláknové zpracování[2], ale také kryptografii, a další oblasti. Nejvíce se JUCE ale využívá díky sadě zvukových funkcí a tříd. Toto rozhraní podporuje zvukové knihovny jako ASIO, WASAPI, ALSA nebo CoreAudio, a disponuje podporou přehrávání MIDI[3], běžných formátů zvukových souborů jako WAV, MP3, FLAC a podporuje nástroje k otevírání různých typů zvukových zásuvných modulů (Leffler, 2018).

JUCE obsahuje speciální třídy pro vytváření zvukových a prohlížečových zásuvných modulů. Díky uchování kódu ve speciálním formátu (tzv. wrapperu[4]), je možné z jediného kódu sestavit zásuvný modul typu VST, VST3, RTAS, AAX, AU, pro Mac i Windows (Schrag, 2018).

3.2 Zvuk

Pro vývoj ekvalizéru je nezbytné vedle technických pojmů a metodik spjatých z oboru infomačních technologií pochopit také to, co vlastně zvuk a zvukový signál je a jakým způsobem je zvukový signál počítačem zpracován, uchován a interpretován.

3.2.1 Charakteristika zvuku

Zvuk je ve své podstatě vibrace cestující okolím. Vibrace je vytvořena vibrujícím objektem nebo předmětem. Zvuk se okolím šíří jako vlna a může se šířit například kapalinou (voda), pevnou látkou (zeď) nebo plynem (vzduch). Ve vakuu se zvuk šířit nemůže. Ačkoli se zvuk šíří jako vlna, jednotlivé částice prostředí se nepohybují s vlnou, ale pouze kmitají (DOSITS Team).

---

[2] vícevláknové zpracování – technika, která umožňuje provádět více úloh současně na jednom procesoru

[3] MIDI – Musical Instrument Digital Interface – protokol, který umožňuje komunikaci a výměnu zvukových informací mezi zařízeními a softwarem

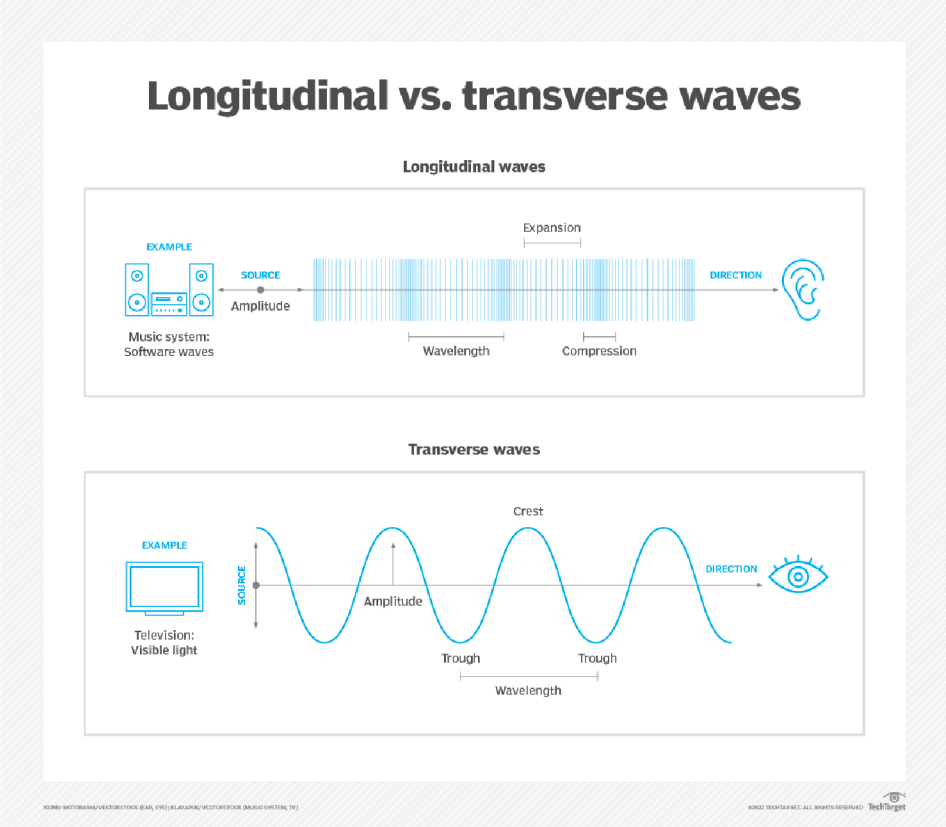
[4] wrapper – software, který umožňuje použití zásuvného modulu v hostitelské aplikaci, pro kterou nebyl původně navržen

3.2.1.1 Podélné vlnění

Při podélném vlnění všechny částice prostředí kmitají ve stejném směru, jakým se vlna pohybuje. Když se podélné vlny šíří daným prostředím, částice jsou v prostředí také stlačené a ztenčené. Ke kompresi dochází, když se částice pohybují blízko sebe a vytvářejí oblasti vysokého tlaku (Rouse, 2022).

3.2.1.2 Příčné vlnění

Při příčném vlnění částice v prostředí kmitají kolmo ke zvukové vlně, tedy směrem nahoru a dolů, a pohybují se ve směru vlnění. Příkladem může být vlnění na hladině vody. S výjimkou zvláštních podmínek se zvuk příčným vlněním nepohybuje (Rouse, 2022).

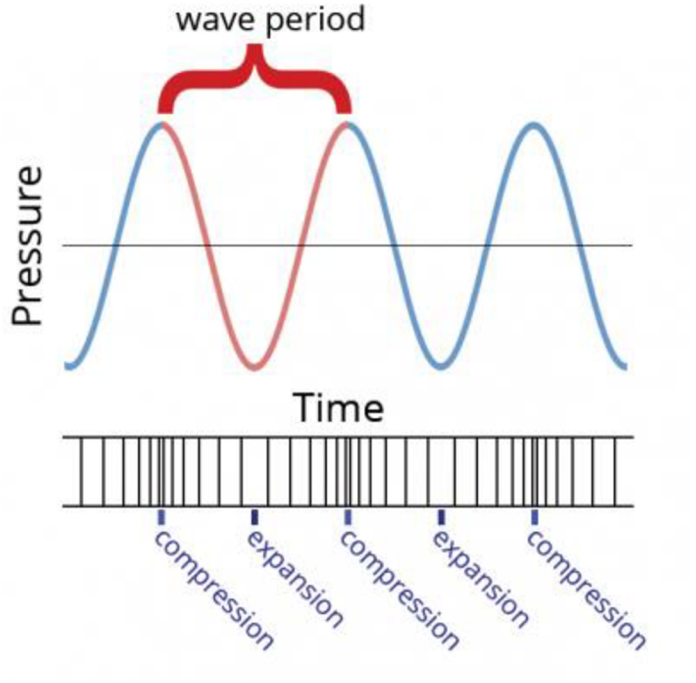


Obrázek 1 - Jak se liší podélné a příčné zvukové vlny (Rouse, 2022)

3.2.1.3 Amplituda

Amplituda zvukové vlny se vztahuje ke změně tlaku způsobené vlnou, která se měří v určitém místě. Zvuk je vnímán jako hlasitější, pokud se amplituda zvyšuje, a jako tišší, pokud se amplituda snižuje.

Amplituda vlny souvisí s množstvím energie, kterou nese. Vlna s vysokou amplitudou nese velké množství energie, vlna s nízkou amplitudou nese naopak malé množství energie. Intenzita zvukové vlny je definována jako energie přenesená jednotkovou plochou za jednotku času ve směru, kterým se zvuk šíří. Zvuky s vyšší intenzitou jsou tím pádem vnímány jako hlasitější. Relativní intenzita zvuku se často udává v jednotkách označovaných jako decibely (dB) (DOSITS Team).

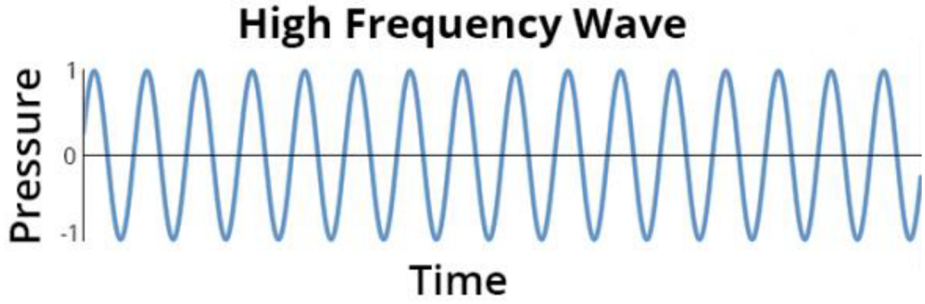


Obrázek 2 - Změna tlaku způsobená zvukovou vlnou (DOSITS Team)

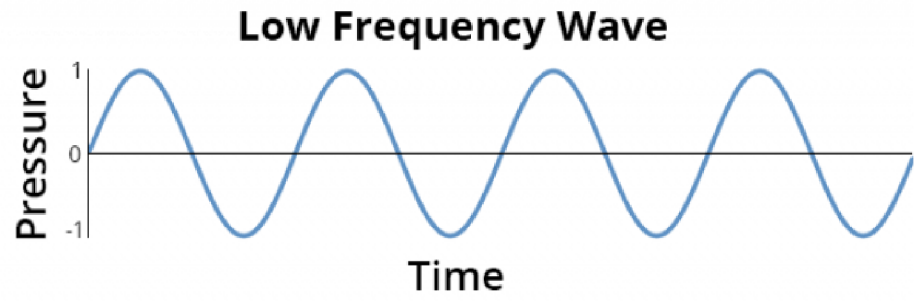
3.2.1.4 Frekvence

Nízké nebo vysoké tóny se vztahují k frekvenci zvukové vlny. Zvukové vlny mají opakující se vzorec, kde jednomu úplnému cyklu říkáme cyklus. Doba, za kterou se cyklus dokončí, se nazývá perioda. Frekvence je počet cyklů za sekundu. Jednotkou frekvence je hertz (Hz).

Na obrázku 3 je ukázána vysokofrekvenční vlna, na obrázku 4 je ukázána nízkofrekvenční vlna. Obě vlny jsou na obrázku zobrazeny jako závislost tlaku na čase. Vysokofrekvenční vlna má 15 cyklů a nízkofrekvenční vlna má pouze 3 cykly za stejný časový úsek.



Obrázek 3 - Vysokofrekvenční vlna (DOSITS Team)

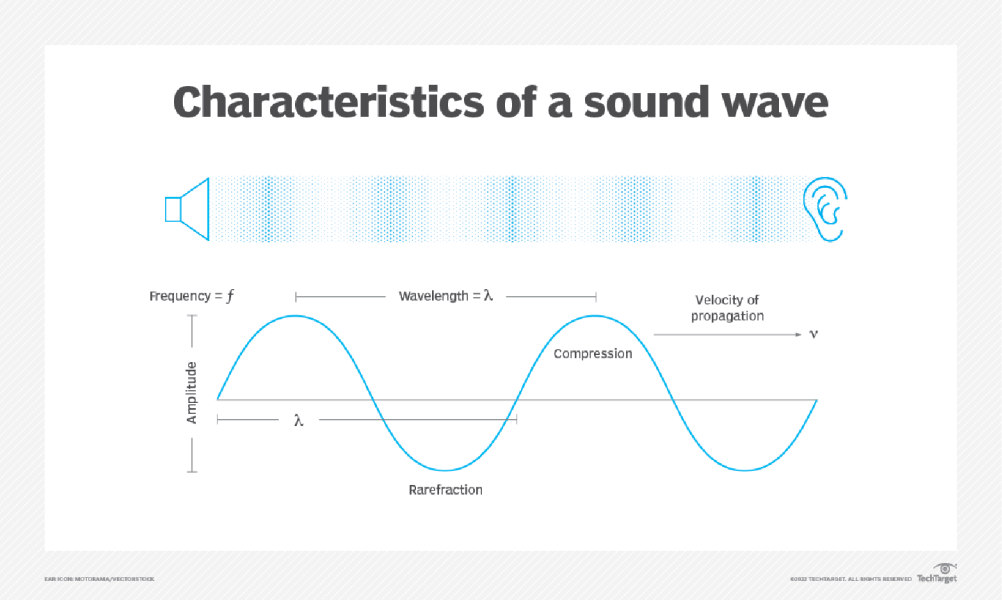


Obrázek 4 - Nízkofrekvenční vlna (DOSITS Team)

Se zvyšující se frekvencí se zvyšuje počet cyklů, který za sekundu proběhne, a zároveň se zvyšuje tón zvuku. Se snížením frekvencí se tón zvuku snižuje, vlnová délka prodlužuje a tím se i zmenšuje počet cyklů za sekundu (DOSITS Team).

3.2.1.5 Vlnová délka

Vlnová délka souvisí s rychlostí, kterou se zvuk šíří, a je dána poměrem rychlosti zvuku a frekvence zvuku. Vysokofrekvenční zvuk má kratší vlnovou délku než nízkofrekvenční zvuk.



Obrázek 5 – Zvukové vlny rozkmitávají vzduch a přenášejí zvuk vnějším uchem k ušnímu bubínku, který vibracemi posílá zvukové informace do mozku ke zpracování (Rouse, 2022)

Při rychlosti 1 500 metrů za sekundu (m/s) pro přibližnou rychlost zvuku v mořské vodě zjistíme následující vztahy mezi frekvencí a vlnovou délkou (DOSITS Team).

Frekvence (počet cyklů za jednu sekundu)	**Vlnová délka**
100 Hz	15 m
1 000 Hz	1,5 m
10 000 Hz	0,15 m

Tabulka 1 - Vztah mezi frekvencí a vlnovou délkou (DOSITS Team)

3.2.2 Lidské vnímání zvuku

Sluch je pro člověka důležitým smyslem, avšak lidský sluch má několik limitů oproti jiným druhům živočichů. Lidské ucho není schopno například zaznamenat zvuk psí píšťalky. To, jestli je člověk schopen zvuk slyšet, závisí na frekvenci a intenzitě zvuku. Lidské ucho vnímá nejlépe zvuky, které obsahuje frekvence, z jakých se skládá lidská řeč (DOSITS Team).

Lidské ucho je obvykle schopno slyšet frekvence v rozsahu od 20 Hz (20 cyklů za sekundu) do 20 000 Hz (tj. 20 kHz, tedy 20 000 cyklů za sekundu). Proto velké množství ekvalizérů podporuje úpravu zvuku v tomto rozsahu. V uchu je pak membrána se spoustou drobných kůstek, které se nachází v těsném okolí bubínku. Bubínek zesiluje změny tlaku vzduchu. Dále je v uchu membrána se spoustou drobných chloupků, které vibrují v souladu s tlakovými vlnami zvuku a detekují různé části slyšitelného spektra. Tyto chloupky rozeznávají změny tlaku vzduchu a převádí je na elektrické impulsy (respektive elektrochemické impulsy, protože různé zvuky vyvolávají různé pocity a emoce), které vedou do mozku a jsou vnímány jako zvuk.

Když jsou současně vysílány dva zvuky s rozdílem frekvencí větším než 7 Hz, většina lidí je schopna rozpoznat přítomnost složitější zvukové vlny. Vlny, které při současném poslechu vyvolávají obzvláště příjemné pocity, se přezdívají souzvučné. Takové vlny tvoří základ intervalů v hudbě. Dva zvuky, jejichž frekvence tvoří poměr 2:1, jsou odděleny oktávou (The Physics Classroom).

3.2.2.1 Růžový šum

Sluchová soustava, která zpracovává frekvence, nevnímá různé frekvence se stejnou citlivostí. Frekvence v rozmezí 2 kHz – 5 kHz znějí při dané intenzitě hlasitěji, než jiné, kvůli snazšímu rozpoznávání lidské řeči (Frontiers for Young Minds, 2022).

Kvůli této vlastnosti byl vytvořen růžový šum, ve kterém je na každou oktávu frekvence stejné množství energie. Energie však na každé frekvenční úrovni klesá o zhruba 1 dB až 3 dB na oktávu. Grafické ekvalizéry taktéž rozdělují zvuk po oktávách. Zvukoví

inženýři nechávají projít ekvalizérem růžový šum, aby otestovali, zda má testovaný zvuk ve spektru plochou frekvenční charakteristiku (The Sleep Foundation, 2021).

3.3 Zpracování zvuku počítačem

3.3.1 Převod zvukové vlny na zvukový signál

Při použití mikrofonu připojeného k počítači se převádí zvukové vlny na zvukový signál. Změny tlaku vzduchu pohybují membránou mikrofonu, která je spojena s magnetem. Pohyb magnetu lze snímat pomocí cívky, která pohyby převádí na elektrický signál díky tomu, že měnící se magnetické pole indikuje elektrické pole.

Zvuk se tedy převádí na velmi rychle se měnící napětí, které je posíláno do analogově-digitálního obvodu. Ten převádí napětí na řadu čísel, která se většinou ukládají v binární podobě. Tento proces se nazývá vzorkování (tzv. sampling).

Zvuk reprezentovaný jako velká řada čísel lze v počítači ukládat, manipulovat s ním, komprimovat, apod. Tuto řadu čísel lze zpracovat obráceným způsobem, díky čemuž se řada čísel převede do zvukového signálu, které reproduktor pomocí kmitů vysílá jako zvukové vlny (Feist, 2021).

3.3.1.1 Analogový zvukový signál

Analogový zvukový signál je reprezentovaný elektrickým napětím nebo proudem, který je analogický zvukovým vlnám. Zpracování signálu zahrnuje fyzickou změnu signálu změnou napětí nebo proudu prostřednictvím elektrických obvodů.

Před nástupem digitální technologie byl analog jediný způsob, jak manipulovat se signálem. V hudební tvorbě je však analog stále žádoucí, protože vytváří nelineární odezvy, které lze pomocí digitálních obvodů jen obtížně napodobit (Smith, 1999).

3.3.1.2 Digitální zvukový signál

Digitální zvukový signál reprezentuje průběh zvuku jako posloupnost symbolů, většinou ve formě binárních čísel. Signál je zpracován mikroprocesorem. Digitální zpracování zvuku je oproti analogovému mnohem efektivnější a výkonnější (Smith, 1999).

3.3.2 Vzorkovací frekvence

Kvalita digitálního zvukového signálu do značné míry závisí na dvou faktorech: vzorkovací frekvenci a formátu vzorku neboli bitové hloubce. Zvýšení vzorkovací frekvence nebo počtu bitů v každém vzorku zvyšuje kvalitu signálu, ale zároveň se tím zvyšuje velikost souboru (The Audacity Team).

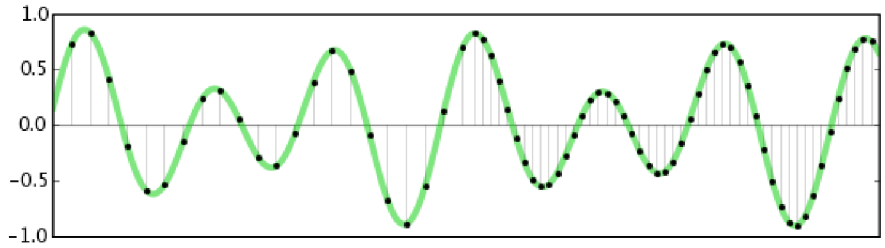
3.3.2.1 Vzorkovací frekvence

Vzorkovací frekvence se měří v hertzích. Tato hodnota představuje počet vzorků zachycených za sekundu za účelem reprezentace tvaru vlny. Vyšší vzorkovací frekvence umožňují reprezentovat vyšší zvukové frekvence. Za předpokladu, že vzorkovací frekvence je vyšší než dvojnásobek nejvyšší přítomné zvukové frekvence, lze průběh zvukové vlny přesně rekonstruovat z digitálních vzorků. Frekvence, které jsou vyšší než polovina vzorkovací frekvence, nelze v digitálních vzorcích správně reprezentovat, a pokud jsou v původním zvuku přítomny, musí být před převodem do digitální podoby odstraněny. Polovina vzorkovací frekvence tedy představuje horní hranici, která se nazývá „Nyquistova frekvence“, a analogový průběh musí být zcela pod touto hranicí, aby byl správně digitálně reprezentován. Analogové frekvence na této hranici nebo nad ní nelze správně reprezentovat digitálním vzorkováním a způsobily by zkreslení.

Vzhledem k tomu, že lidské ucho běžně slyší frekvence od 20 Hz do 20 000 Hz, vzorkovací frekvence 40 000 Hz je absolutním minimem pro reprodukci celého rozsahu slyšitelných zvuků.

Nejběžnější vzorkovací frekvence měřené v Hz jsou 8 000 (telefonické hovory), 16 000 (VoIP), 22 050 (PCM a MPEG), 44 100 (nejčastěji CD, MP3, apod.),

48 000 (nejčastěji WAV, standard v odvětví hudební produkce), 96 000 (DVD, Blu-Ray, HD DVD), 192 000 (DVD, LPCM DVD, Blu-Ray, HD DVD) (The Audacity Team).



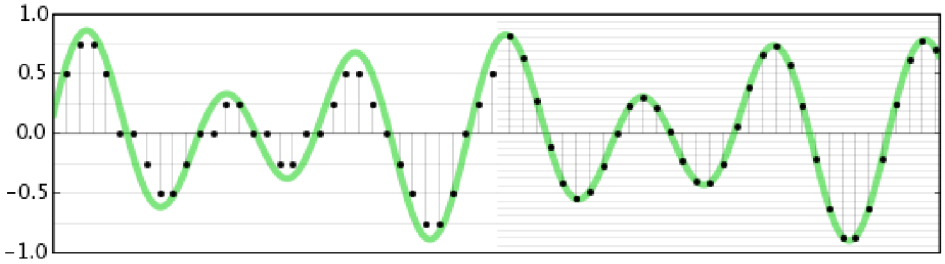
Obrázek 6 – Průběh zvukové vlny s vyšší a nižší vzorkovací frekvencí (The Audacity Team)

3.3.2.2 Formát vzorku

Dalším důležitým měřítkem kvality zvuku je formát vzorku neboli bitová hloubka. Bitová hloubka se měří počtem bitů použitých k reprezentaci každého vzorku. Čím více bitů je použito, tím přesnější je reprezentace každého vzorku. Zvyšování počtu bitů také zvyšuje maximální dynamický rozsah zvukového záznamu, tedy rozdíl v hlasitosti mezi nejhlasitějšími a nejtišími možnými zvuky, které lze reprezentovat.

Dynamický rozsah se měří v decibelech. Lidské ucho je schopno vnímat zvuky s dynamickým rozsahem nejméně 90 dB. Je však lepší nahrávat zvuk s větším dynamickým rozsahem, neboť při nízkých úrovních se nevyužívá celá bitová hloubka a tuto ztrátu nelze znovu zachytit například normalizací.

Mezi běžné formáty vzorků a jejich příslušný dynamický rozsah patří 8bitové celé číslo (48 dB), 16bitové celé číslo (96 dB), 24bitové celé číslo (145 dB), 32bitové číslo s desetinnou čárkou (téměř nekonečno dB) (The Audacity Team).



Obrázek 7 – Průběh zvukové vlny s nízkou vzorkovací frekvencí a vysokou vzorkovací frekvencí (The Audacity Team)

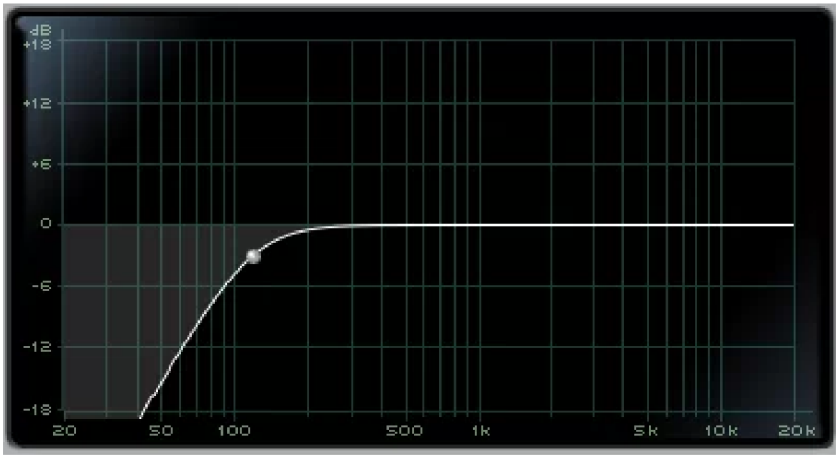
3.4 Ekvalizér

Ekvalizace je jednou z nejběžnějších forem zpracování zvuku v hudební produkci. Pomocí ekvalizéru lze upravit úroveň hlasitosti a rozsah frekvencí, což následně umožňuje zvuk ošetřit od nežádoucích artefaktů, nebo zvuk pomocí tohoto nástroje zásadně změnit. Mezi nejběžnější praktiky patří snižování nežádoucích frekvencí a posilování jiných, za účelem vyvážení frekvencí tak, aby spolu dobře zněly a pracovaly (Armada Music).

3.4.1 Ovládání ekvalizéru

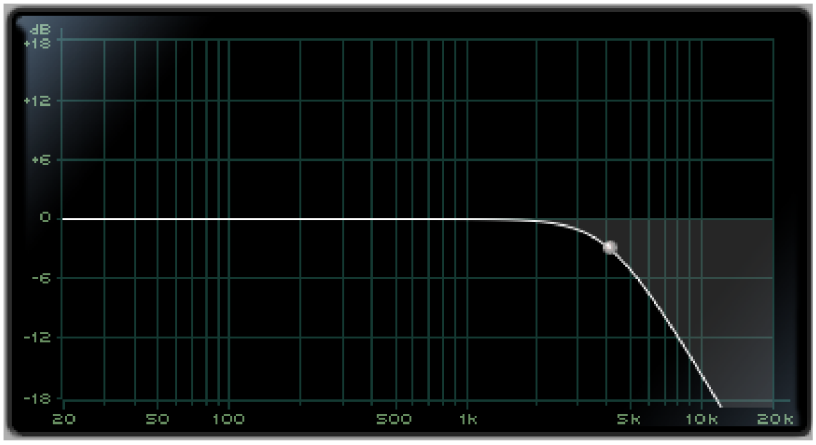
Ekvalizéry mají k dispozici různé typy parametrů, které se chovají podle toho, o jaký typ parametru jde. Počet parametrů, jaký je v ekvalizéru k dispozici se liší.

Low-cut filtr (též nazývaný high-pass filtr) umožňuje kompletně odstranit frekvence směrem zleva doprava na frekvenčním spektru, a proto se nazývá low-cut. Na frekvenčním spektru se totiž vlevo nachází nízké frekvence a směrem doprava jsou frekvence vyšší (Alex, 2019).



Obrázek 8 – Filtr high-pass (Alex, 2019)

High-cut filtr (též low-pass filtr) naopak odstraňuje frekvence směrem zprava doleva, počínaje od vyšších frekvencí až po nízké.



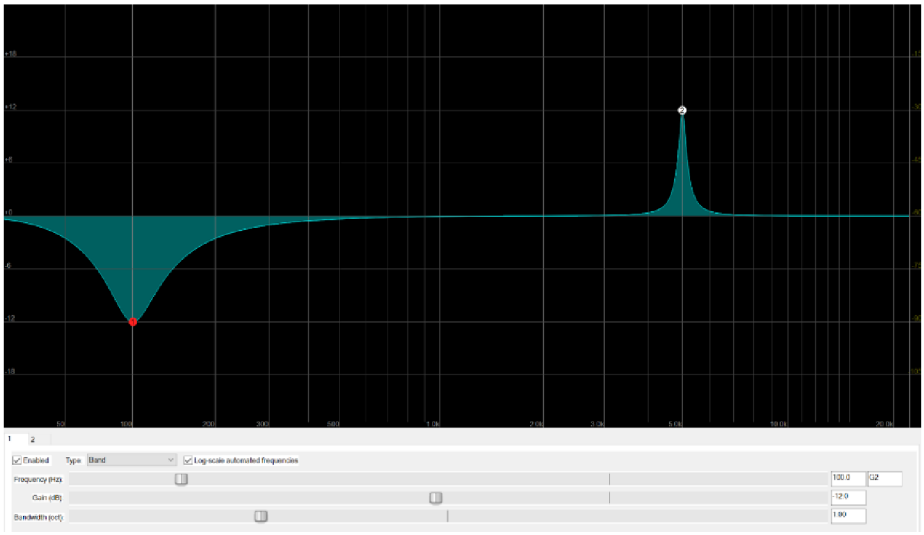
Obrázek 9 – Filtr low-pass (Alex, 2019)

Shelf je specifický typ parametru. Může ovlivňovat frekvence směrem doleva k nízkým frekvencím (low-shelf), nebo doprava směrem k vysokým frekvencím (high-shelf). Od low-cut a high-cut filtrů se liší sklonem křivky. S tímto typem lze zvyšovat/snižovat hlasitosti frekvencí (Jonathan).

Obrázek 10 – Filtr shelf (Jonathan)

Bell je nejpoužívanější typ parametru. S tímto parametrem lze docílit zvýšení/snížení hlasitosti frekvencí v rozsahu frekvenčního pásma, který se mu nastaví. Toto frekvenční pásmo se často označuje Q a představuje rozsah, který určuje okolí podléhající nastavení parametru. Čím je Q menší, tím menší má parametr rozsah působnosti, a v ideálním případě se rozsah může zúžit natolik, že v rozsahu je pouze několik frekvencí. Tento typ parametru se nejčastěji užívá právě pro odstraňování/snižování hlasitosti frekvencí, které ve zvuku

způsobují problémy nebo nezní dobře. Takových parametrů lze do ekvalizéru umístit několik (Wilson, 2021).



Obrázek 11 – Filtr bell (Wilson, 2021)

3.5 Zásuvné moduly

Zásuvné moduly (tzv. pluginy[5]) jsou doplňkové programy, které se zaměřují na specifické úlohy, které je nemožné nebo obtížné splnit pomocí samotného softwaru. Mezi současné populární rozhraní zásuvných modulů v odvětví hudební produkce patří formáty Virtual Studio Technology (VST) společnosti Steinberg, Audio Units (AU) společnosti Apple, a Avid Avid Audio eXtension (AAX). Rozhraní VST a AU jsou veřejně zdokumentována, zatímco rozhraní AAX nikoli. Zásuvné moduly VST jsou napsány v jazyce C++ bez potřeby externích frameworků nebo základních tříd, avšak existují různé frameworky, které lze pro vývoj VST pluginu využít. Vývoj modulů formátu AU vyžaduje dobrou znalost programování pro Mac (Pirkle, 2013).

VST formát představuje revoluční technologii, která umožňuje přidávání zvukových efektů v reálném čase do zvukových stop nahraných v počítači. Počítač se tak stal z MIDI sekvenceru a audio rekordéru plnohodnotnou platformou pro nahrávání, mixování

[5] plugin – zásuvný modul, který lze používat jako samostatnou aplikaci bez potřeby hostitelské aplikace nebo digitální zvukové pracovní stanice

a produkci zvuku. Velký podíl na úspěchu VST mělo to, že technologie byla vydána jako open-source a ostatní společnosti mohly vyvíjet vlastní software (Vincent, 2022).

VSTi označuje, že nejde o zásuvný modul zvukového efektu, ale o emulaci nástroje (instrument), případně syntetizéru (Laukkonen, 2021).

3.5.1 Vývoj zásuvného modulu

Existují dvě paradigmata pro psaní zásuvných modulů VST3, která zahrnují způsob rozdělení dvou hlavních úkolů zásuvných modulů: zpracování zvukového signálu a implementace grafického uživatelského rozhraní. Ve vzoru duální komponenty se implementuje zásuvný modul ve dvou objektech jazyka C++: jeden pro zpracování zvukového signálu, nazývaný procesor, a druhý pro grafické uživatelské rozhraní, nazývaný ovladač. Architekti VST3 si dali záležet na tom, aby tyto dva objekty od sebe navzájem izolovali, proto je komunikace mezi nimi téměř nemožná. To umožňuje, aby zpracování zvuku a připojení/implementace grafického uživatelského rozhraní běžely na oddělených procesorech (Pirkle, 2019).

4 Vlastní práce

Vlastní práce je popsána pro prostředí operačního systému Windows 11.

4.1 Vývojové prostředí

Pro vývoj ekvalizéru je jako vývojové prostředí vybráno Visual Studio. Visual Studio je integrované vývojové prostředí (IDE[6]) vyvinuté společností Microsoft. Nejedná se o IDE specifické pro určitý jazyk, protože s ním lze psát kód až v 36 programovacích jazycích.

4.1.1 Instalace prostředí Visual Studio

Visual Studio lze stáhnout na webu https://visualstudio.microsoft.com/cs/. Během instalace je potřeba vybrat komponentu *Desktop development with C++*.

4.2 Instalace JUCE Framework

JUCE Framework je k dispozici na GitHubu (https://github.com/juce-framework/JUCE). Po stažení lze vidět, že ve složce *JUCE/extras/Projucer/Builds* je k dispozici několik typů sestav. To, se kterou sestavou se pracuje, se odvíjí od operačního systému a verze nainstalovaného prostředí Visual Studio. V případě systému Windows s verzí Visual Studio 2022, se pracuje se sestavou ve složce *VisualStudio2022*.

V adresáři se nachází soubor *Projucer.sln*, který se otevře a zkompiluje ve Visual Studiu. Tím se v adresáři vytvoří položky *x64/Debug/App* a v této složce další soubory, včetně *Projucer.exe*, přes který lze JUCE následně spouštět. Po kompilaci se otevře okno nového projektu JUCE.

V otevřeném okně se nakonfigurují globální proměnné a je potřeba se ujistit, že položka *Path to JUCE* obsahuje cestu k adresáři *JUCE*, a položka *JUCE Modules* obsahuje cestu k adresáři *JUCE/modules*.

---

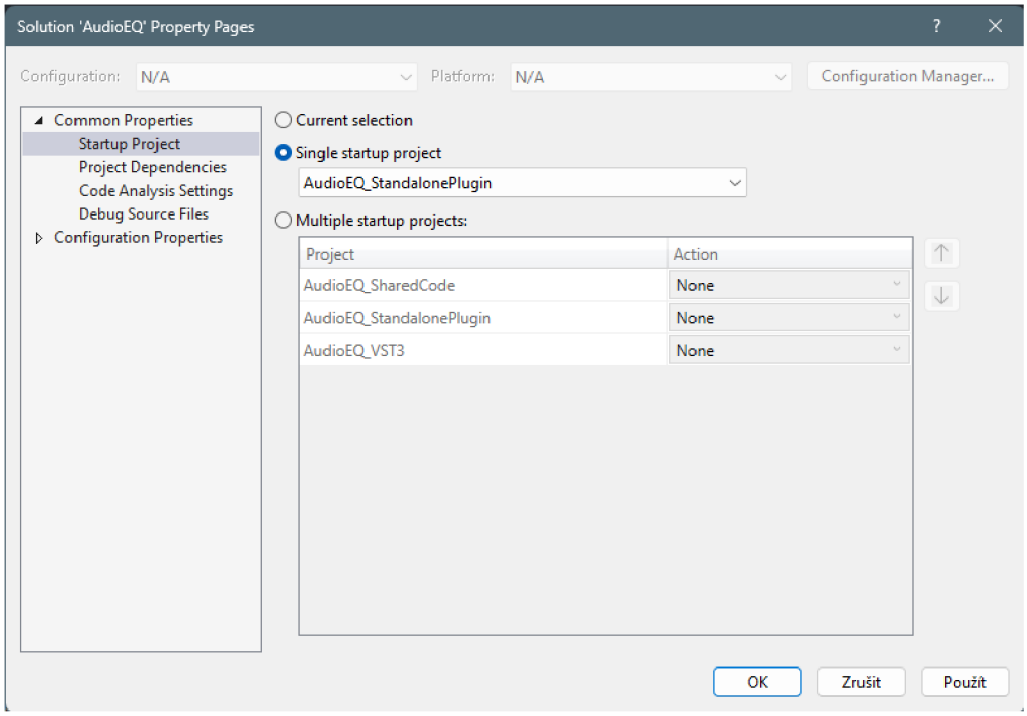
[6] IDE – Integrated Development Environment – softwarová aplikace, která poskytuje komplexní vybavení pro vývoj softwaru v jediném rozhraní

Následně se v levém menu v sekci *Plug-In* pojmenuje projekt a následně se vytvoří. Tímto se vytvoří adresář se soubory, které se vztahují k danému projektu.

V tento moment je projekt připravený k samotnému vývoji.

4.3 Vývoj

V nastavení projektu je možné nastavit, aby se program spouštěl jako samostatný program (tzv. standalone plugin). To je vhodné pro rychlejší testování aplikace.



Obrázek 12 – Nastavení spouštění programu v režimu ladění v prostředí Visual Studio (vlastní zdroj)

V adresáři ekvalizéru existují ve složce *Source* čtyři soubory, se kterými se primárně pracuje: *PluginProcessor.cpp*, *PluginProcessor.h*, *PluginEditor.cpp* a *PluginEditor.h*.

4.3.1 PluginProcessor.h

Mezi nejdůležitější funkce, které se v kódu volají, jsou funkce *prepareToPlay* a *processBlock*.

Funkce *prepareToPlay* je volána hostitelem v moment, kdy se chystá spustit přehrávání.

Funkce *processBlock* probíhá vždy po stisknutí tlačítka přehrání. Po stisku tlačítka přehrávání hostitel začne posílat vyrovnávací paměť zvuku[7] pravidelnou rychlostí do zásuvného modulu a jeho úkolem je vrátit hostiteli zpracovaný zvuk. Tento řetězec událostí nelze přerušit. Pokud je zpracování příliš pomalé, vznikne zpoždění, které může způsobit cvakání nebo praskání v reproduktorech, a to má potenciál poškodit reproduktory i poškodit sluch. Veškeré zpracování musí být provedeno v pevně stanoveném čase.

4.3.2 PluginProcessor.cpp

Tento soubor obsahuje tělo funkcí *prepareToPlay* a *processBlock*. Ve funkci *prepareToPlay* proběhne inicializace před přehráváním. Ve funkci *processBlock* se zpracovávají zvuková data například z vyrovnávací paměti zvuku nebo z MIDI ovladače.

4.3.3 PluginEditor.h

V tomto souboru primárně probíhá pouze volání funkcí, které jsou popsány v souboru *PluginEditor.cpp*.

4.3.4 PluginEditor.cpp

Skrze tento kód lze nastavit vizuální prvky zásuvného modulu, jako vykreslování, barvy, změny velikosti, a další. Zároveň zde probíhají aktualizace jednotlivých parametrů na úrovni grafického rozhraní, a to v reálném čase na základě zpracovávaného zvuku.

4.3.5 Vytvoření zvukových parametrů

Jako první se vytvoří ovladače pro ovládání parametrů low-cut, high-cut, sklon křivky (tzv. slope) a zesílení (tzv. gain). Parametr slope určuje, jak moc strmá křivka parametru low-cut nebo high-cut je. Parametr gain slouží k zesílení vybraného rozsahu frekvencí.

---

[7] vyrovnávací paměť zvuku – dočasná paměťová oblast v paměti počítače, která uchovává zvuková data pro zpracování a přehrávání a slouží ke snížení latence a zajištění plynulého výkonu zvuku

JUCE používá objekt nazvaný *audioProcessorValueTreeState* ke koordinaci synchronizace parametrů s parametry v GUI a proměnnými v DSP[8]. V *PluginProcessor.h* se taková funkce volá a nese název *apvts*. První parametr funkce je audio procesor, na který se má napojit, druhým parametrem je *Undo Manager*, který v tomto případě není potřeba, třetím parametrem je identifikátor a posledním parametrem je rozložení parametru.

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

Zdrojový kód 1 – Volání funkce pro vytvoření parametrů

[8] DSP – Digital Signal Processing – použití matematických algoritmů k manipulaci a transformaci digitálních signálů, jako je zvuk

V *PluginProcess.cpp* se následně nadefinuje rozložení parametrů, což znamená, že se vytvoří ovladače jednotlivých parametrů a přidají se do rozložení parametrů.

```
juce::AudioProcessorValueTreeState::ParameterLayout
AudioEQAudioProcessor::createParameterLayout()
{
    juce::AudioProcessorValueTreeState::ParameterLayout layout;
    layout.add(std::make_unique<juce::AudioParameterFloat>
    ("LowCut Freq", "LowCut Freq",
    juce::NormalisableRange<float>(20.f, 20000.f, 1.f, 1.f), 20.f));
    layout.add(std::make_unique<juce::AudioParameterFloat>
    ("HighCut Freq", "HighCut Freq",
    juce::NormalisableRange<float>(20.f, 20000.f, 1.f, 1.f), 20000.f));
    layout.add(std::make_unique<juce::AudioParameterFloat>
    ("Peak Freq", "Peak Freq", juce::NormalisableRange<float>
    (20.f, 20000.f, 1.f, 1.f), 750.f));
    layout.add(std::make_unique<juce::AudioParameterFloat>
    ("Peak Gain", "Peak Gain", juce::NormalisableRange<float>(-24.f, 24.f, 0.2f, 1.f), 0.0f));
    layout.add(std::make_unique<juce::AudioParameterFloat>
    ("Peak Quality", "Peak Quality",
    juce::NormalisableRange<float>(0.1f, 10.f, 0.05f, 1.f), 1.f));
    juce::StringArray stringArray;
    for (int i = 0; i < 4; ++i){
        juce::String str;
        str << (12 + i*12);
        str << " db/Oct";
        stringArray.add(str);
    }
    layout.add(std::make_unique<juce::AudioParameterChoice>
    ("LowCut Slope", "LowCut Slope", stringArray, 0));
    layout.add(std::make_unique<juce::AudioParameterChoice>
    ("HighCut Slope", "HighCut Slope", stringArray, 0));

    return layout;
}
```

Zdrojový kód 2 – Vytvoření a pojmenování parametrů

4.3.6 Nastavení digitálního zvukového zpracování (DSP)

Pro nastavení digitálního zvukového zpracování (DSP) je potřeba v JUCE v sekci modulů přidat modul *juce_dsp*.

Modul *juce_dsp* je knihovna pro zpracování digitálních signálů, která poskytuje sadu tříd a funkcí pro úlohy zpracování zvuku, jako je filtrování, ekvalizace, modulace a spektrální analýza. Knihovna je navržena tak, aby byla rychlá a efektivní, takže je vhodná pro aplikace zpracování zvuku v reálném čase. Knihovna poskytuje vývojářům snadno použitelné rozhraní API pro vytváření vlastních algoritmů zpracování zvuku.

Třídy této knihovny umí zpracovávat pouze monofonní zvuk, což znamená, že se bude zpracovávat pouze jeden kanál zvuku. Jelikož ekvalizér bude zpracovávat stereo zvuk, musí se duplikovat metody zpracování, které se do kódu zavedou.

4.3.6.1 PluginProcessor.h

Knihovna obsahuje spoustu tříd, ale pro účel bakalářské práce se využijí následující třídy, na které se lze odkazovat přes vlastní typový alias.

```
using Filter = juce::dsp::IIR::Filter<float>;
using CutFilter = juce::dsp::ProcessorChain<Filter, Filter, Filter, Filter>;
using MonoChain = juce::dsp::ProcessorChain<CutFilter, Filter, CutFilter>;
MonoChain leftChain, rightChain;
```

Zdrojový kód 3 – Definování typových aliasů

Typový alias *MonoChain* se odkazuje na třídu, která slouží ke spojení více procesorů DSP do jednoho celku, což usnadňuje správu navazujících procesů pro zpracování zvuku. Každý procesor v řetězci je na vstupní signál aplikován postupně a výstup jednoho procesoru se stává vstupem dalšího. Použití třídy může zjednodušit správu pořadí zpracování a toku zvukových dat mezi procesory.

Třída, na kterou se odkazujeme pomocí *CutFilter*, představuje řetězec filtrů pro zpracování zvuku. Jedná se o šablonovou třídu, která jako argumenty šablony přijímá jeden nebo více typů třídy *Filter* a představuje řadu propojených zvukových filtrů. Vstupní zvukový signál prochází jednotlivými filtry v řetězci v pořadí, v jakém byly přidány, a konečný výstup je výsledkem zpracování přes všechny filtry v řetězci. Třída *ProcessorChain* určuje způsob, jak propojit více filtrů dohromady, spravovat jejich pořadí a zpracovávat zvuk skrze celý řetězec.

Alias *Filter* se odkazuje na třídu, která implementuje filtr s nekonečnou impulzní odezvou (IIR[9]) pro zvukové signály. Jde o šablonovou třídu, která přijímá jeden argument typu *float*, který určuje typ zpracovávaných dat. Třída *IIR::Filter* se používá k filtrování zvukových signálů tak, že na vstupní zvuková data aplikuje matematickou přenosovou funkci, která vytvoří filtrovaný výstup. Tato přenosová funkce je definována sadou koeficientů, které popisují odezvu filtru na různé frekvence. Tato třída implementuje běžné typy filtrů IIR, jako jsou high-cut, low-cut nebo pásmové filtry.

4.3.6.2 PluginProcessor.cpp

Než se filtry aplikují, musí se připravit k přehrávání. K tomu lze použít třídu *juce::dsp::ProcessSpec*, která představuje specifikace jednotky pro zpracování zvuku. Obsahuje informace, jako je vzorkovací frekvence, velikost bloku, počet kanálů a maximální počet vzorků, které lze zpracovat najednou. Tyto informace slouží ke konfiguraci jednotky zpracování zvuku, alokaci paměti a zajištění kompatibility mezi různými komponentami zpracování zvuku.

```
juce::dsp::ProcessSpec spec;
spec.maximumBlockSize = samplesPerBlock;
spec.numChannels = 1;
spec.sampleRate = sampleRate;
leftChain.prepare(spec);
rightChain.prepare(spec);
```

Zdrojový kód 4 – Specifikace parametrů pro přípravu zpracování zvuku

Poté se vytvoří zvukový blok pro levý a pravý kanál - souvislé pole zvukových vzorků, které představuje vyrovnávací paměť zvuku, a vytvoří se kontext pro zpracování zvuku, aby se nahradila stará data v dané vyrovnávací paměti daty novými. Poté se předá nově vytvořený kontext řetězci filtru.

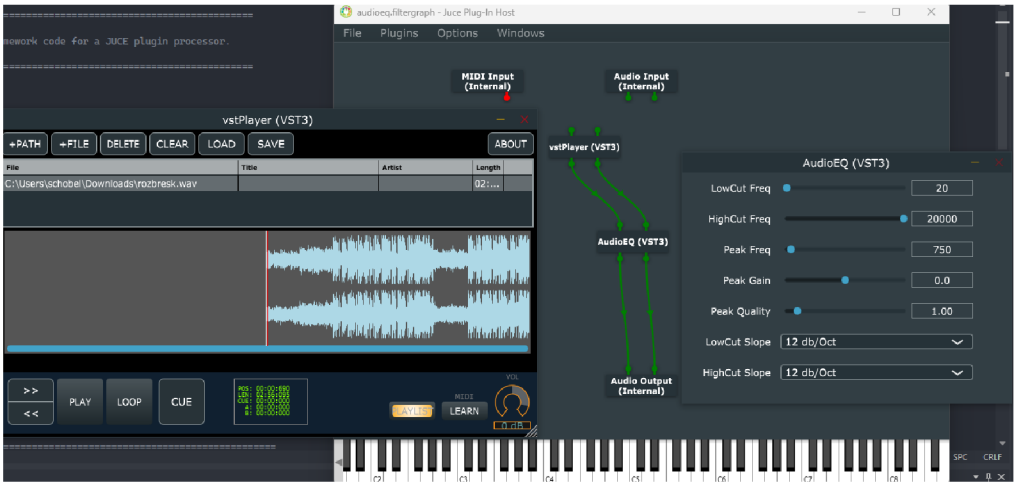
[9] IIR – Infinite Impulse Response – typ digitálního filtru používaného při zpracování signálu, který může poskytnout nekonečnou odezvu na konečný vstupní signál a který se vyznačuje zpětnovazební smyčkou

```
juce::dsp::AudioBlock<float> block(buffer);
auto leftBlock = block.getSingleChannelBlock(0);
auto rightBlock = block.getSingleChannelBlock(1);
juce::dsp::ProcessContextReplacing<float> leftContext(leftBlock);
juce::dsp::ProcessContextReplacing<float> rightContext(rightBlock);
leftChain.process(leftContext);
rightChain.process(rightContext);
```

Zdrojový kód 5 – Vytvoření zvukového bloku levého a pravého kanálu

4.3.7 Nastavení hostitele ekvalizéru

Aby se do ekvalizéru dostala zvuková data, je potřeba plugin spouštět v hostiteli – aplikaci, která je schopna zvuk přehrávat a přesměrovat jej do ekvalizéru, jako tomu je v profesionálních systémech pro produkci zvuku. JUCE framework již obsahuje hostitele, po zkompilování dosavadního kódu lze v hostiteli naskenovat počítač, což najde všechny zásuvné moduly podporované daným hostitelem pro daný systém a načte vytvořený ekvalizér. V hostiteli se pak připraví modul pro přehrávání zvuku ze souboru, což přesměruje zvuk do ekvalizéru a zvuk z ekvalizéru do finálního zvukového výstupu počítače.



Obrázek 13 – Ekvalizér (vpravo) spuštěný jako zásuvný modul v hostiteli (v pozadí) (vlastní zdroj)

4.3.8 Napojení parametrů peak na filtrování

4.3.8.1 PluginProcessor.h

Jako první se vytvoří struktura, která bude obsahovat parametry řetězce zvukových procesorů: rozsah frekvencí parametru peak, zesílení parametru peak, kvalitu výřezu parametru peak, low-cut, high-cut, sklon křivky filtru low-cut a sklon křivky filtru high-cut. Poté se implementuje funkce, která vrací instanci struktury inicializovanou hodnotami z objektu *apvts* předaného funkci jako argument. Funkce načte hodnoty parametrů z objektu *apvts* a vrátí strukturu s inicializovanými hodnotami.

```
struct ChainSettings
{
    float peakFreq{ 0 }, peakGainInDecibels{ 0 }, peakQuality{ 1.f };
    float lowCutFreq{ 0 }, highCutFreq{ 0 };
    int lowCutSlope{ 0 }, highCutSlope{ 0 };
};
ChainSettings getChainSettings(juce::AudioProcessorValueTreeState& apvts);
```

Zdrojový kód 6 – Definování struktury ChainSettings

Pro lepší čitelnost kódu lze pomocí výčtového typu pojmenovat hodnoty podle typu parametru. Hodnoty se použijí k reprezentaci pozic v řetězci a identifikaci pozici filtru v řetězci zpracování a na každý filtr se v závislosti na jeho pozici aplikují různé zpracování.

```
enum ChainPositions
{
    LowCut,
    Peak,
    HighCut
};
```

Zdrojový kód 7 – Pojmenování pozic parametrů

4.3.8.2 PluginProcessor.cpp

```
ChainSettings getChainSettings(juce::AudioProcessorValueTreeState& apvts)
{
    ChainSettings settings;
    settings.lowCutFreq = apvts.getRawParameterValue("LowCut Freq")->load();
    settings.highCutFreq = apvts.getRawParameterValue("HighCut Freq")->load();
    settings.peakFreq = apvts.getRawParameterValue("Peak Freq")->load();
    settings.peakGainInDecibels = apvts.getRawParameterValue("Peak Gain")->load();
    settings.peakQuality = apvts.getRawParameterValue("Peak Quality")->load();
    settings.lowCutSlope = apvts.getRawParameterValue("LowCut Slope")->load();
    settings.highCutSlope = apvts.getRawParameterValue("HighCut Slope")->load();
    return settings;
}
```

Zdrojový kód 8 – Načtení jednotlivých parametrů

Zdrojový kód 8 definuje funkci *getChainSettings*, která jako vstup přijímá odkaz na objekt *apvts*. Funkce vytvoří instanci struktury *ChainSettings* a inicializuje její členské proměnné hodnotami načtenými z objektu *apvts* pomocí metody *load()* vrácených objektů *AudioProcessorValue* z metody *getRawParameterValue()*. Proměnné struktury *ChainSettings* odpovídají různým nastavením řetězce zpracování zvuku. Nakonec funkce vrátí inicializovaný objekt *ChainSettings*.

```
auto chainSettings = getChainSettings(apvts);
auto peakCoefficients = juce::dsp::IIR::Coefficients<float>::makePeakFilter(
        sampleRate,
        chainSettings.peakFreq,
        chainSettings.peakQuality,
        juce::Decibels::decibelsToGain(chainSettings.peakGainInDecibels));
*leftChain.get<ChainPositions::Peak>().coefficients = *peakCoefficients;
*rightChain.get<ChainPositions::Peak>().coefficients = *peakCoefficients;
```

Zdrojový kód 9 – Vytvoření koeficientů pro filtr peak

Kód uvedený výše vytváří peak filtr pro zpracování zvuku. Ze třídy *IIR* se zavolá metoda *makePeakFilter()*, která vytvoří koeficienty peak filtru. Tato metoda přijímá čtyři argumenty: *sampleRate* (vzorkovací frekvence zpracovávaných zvukových dat), *peakFreq* (středová frekvence peak filtru), *peakQuality* (faktor Q peak filtru) a zesílení v decibelech (zesílení použité na peak filtr při jeho středové frekvenci, decibely se získají pomocí metody *decibelsToGain()*). Nakonec se získané koeficienty přiřadí sekcím peak filtru levého

i pravého řetězce. *ChainPositions::Peak* slouží k určení, která sekce řetězců má koeficienty obdržet. Operátor * se používá k dereferenci ukazatelů na koeficienty, aby bylo možné přiřadit skutečné koeficienty sekcím filtrů. Ten samý kód se může vložit také do funkce *processBlock*, aby se prováděné změny filtrů aplikovaly na přehrávaný zvuk.

V tento moment je ekvalizér ve stavu, kdy lze slyšet změny provedené na parametru peak filtru, zesílení nebo kvality.

4.3.9 Napojení parametru low-cut na filtrování

4.3.9.1 PluginProcessor.h

```
enum Slope
{
    Slope_12,
    Slope_24,
    Slope_36,
    Slope_48
};
```

Zdrojový kód 10 – Pojmenování pozic sklonu křivky

Zdrojový kód 10 pojmenuje hodnoty pomocí výčtového typu. Jednotlivé názvy určují úroveň sklonu křivky oříznutí frekvencí. Poté se ve struktuře řetězce mohou deklarovat proměnné pro sklon křivky filtrů low-cut a high-cut.

4.3.9.2 PluginProcessor.cpp

```
auto cutCoefficients =
juce::dsp::FilterDesign<float>::designIIRHighpassHighOrderButterworthMethod
(chainSettings.lowCutFreq, sampleRate, 2 * (chainSettings.lowCutSlope + 1));
auto& leftLowCut = leftChain.get<ChainPositions::LowCut>();
leftLowCut.setBypassed<0>(true);
leftLowCut.setBypassed<1>(true);
leftLowCut.setBypassed<2>(true);
leftLowCut.setBypassed<3>(true);
```

Zdrojový kód 11 – Nastavení koeficientů filtru low-cut

Zdrojový kód 11 zachycuje funkci *prepareToPlay*, kde se nastaví filtr low-cut pro zpracování zvuku. Koeficienty filtru jsou generovány voláním funkce, která přijímá tři parametry: frekvence filtru low-cut, vzorkovací frekvenci a sklon křivky filtru low-cut. Poté se nastaví objekt filtru low-cut a nastaví se jeho stav bypass[10] na *true*, což znamená, že je filtr vypnutý pro každou ze čtyř možných instancí filtru.

Ve zdrojovém kódu 12 se pomocí příkazu *switch* určí nastavení sklonu křivky filtru low-cut a na základě toho nastaví koeficienty filtru a stav bypass instancí filtru podle potřeby. Pokud je sklon nastaven na hodnotu 12, použije se pouze první instance filtru, pokud je hodnota sklonu 24, použijí se první dvě instance a tak dále, až se pro hodnotu 48 použijí všechny čtyři instance.

[10] bypass – parametr, který umožňuje uživateli zapnout nebo vypnout efekt ekvalizace, čímž efektivně obejde zpracování zvukového signálu ekvalizérem

```
switch (chainSettings.lowCutSlope)
    {
        case Slope_12:
        {
            *leftLowCut.get<0>().coefficients = *cutCoefficients[0];
            leftLowCut.setBypassed<0>(false);
            break;
        }
        case Slope_24:
        {
            *leftLowCut.get<0>().coefficients = *cutCoefficients[0];
            leftLowCut.setBypassed<0>(false);
            *leftLowCut.get<1>().coefficients = *cutCoefficients[1];
            leftLowCut.setBypassed<1>(false);
            break;
        }
        case Slope_36:
        {
            *leftLowCut.get<0>().coefficients = *cutCoefficients[0];
            leftLowCut.setBypassed<0>(false);
            *leftLowCut.get<1>().coefficients = *cutCoefficients[1];
            leftLowCut.setBypassed<1>(false);
            *leftLowCut.get<2>().coefficients = *cutCoefficients[2];
            leftLowCut.setBypassed<2>(false);
            break;
        }
        case Slope_48:
        {
            *leftLowCut.get<0>().coefficients = *cutCoefficients[0];
            leftLowCut.setBypassed<0>(false);
            *leftLowCut.get<1>().coefficients = *cutCoefficients[1];
            leftLowCut.setBypassed<1>(false);
            *leftLowCut.get<2>().coefficients = *cutCoefficients[2];
            leftLowCut.setBypassed<2>(false);
            *leftLowCut.get<3>().coefficients = *cutCoefficients[3];
            leftLowCut.setBypassed<3>(false);
            break;
        }
    }
```

Zdrojový kód 12 – Nastavení koeficientů filtru low-cut dle vybraného sklonu křivky

Jelikož se toto zpracování týká pouze levého kanálu, ten samý kód se analogicky napíše také pro pravý kanál. Celý nově přidaný kód v tomto kroku se zkopíruje a vloží do funkce *processBlock*.

Ekvalizér má funkční parametr filtru low-cut spolu s parametrem určujícím sklon křivky oříznutí.

4.3.10 Refaktorování kódu digitálního zpracování zvuku

4.3.10.1 PluginProcessor.h

```
void updatePeakFilter(const ChainSettings& chainSettings);
using Coefficients = Filter::CoefficientsPtr;
static void updateCoefficients(Coefficients& old, const Coefficients& replacements);
```

Zdrojový kód 13 – Volání funkce k aktualizování filtru peak a volání funkce k aktualizování jeho koeficientů

V kódu uvedeném výše se volá nová funkce *updatePeakFilter* s parametrem nastavení řetězce zpracování. S využitím pomocné funkce *updateCoefficients* se aktualizují staré koeficienty filtru novými.

```
template<typename ChainType, typename CoefficientType>
void updateCutFilter(ChainType& leftLowCut, const CoefficientType& cutCoefficients,
const Slope& lowCutSlope)
{
    leftLowCut.template setBypassed<0>(true);
    leftLowCut.template setBypassed<1>(true);
    leftLowCut.template setBypassed<2>(true);
    leftLowCut.template setBypassed<3>(true);
    switch (lowCutSlope)
    {
        case Slope_12:
        {
            *leftLowCut.template get<0>().coefficients = *cutCoefficients[0];
            leftLowCut.template setBypassed<0>(false);
            break;
        }
        case Slope_24:
        {
            *leftLowCut.template get<0>().coefficients = *cutCoefficients[0];
            leftLowCut.template setBypassed<0>(false);
            *leftLowCut.template get<1>().coefficients = *cutCoefficients[1];
            leftLowCut.template setBypassed<1>(false);
            break;
        }
        case Slope_36:
        {
            *leftLowCut.template get<0>().coefficients = *cutCoefficients[0];
            leftLowCut.template setBypassed<0>(false);
            *leftLowCut.template get<1>().coefficients = *cutCoefficients[1];
            leftLowCut.template setBypassed<1>(false);
            *leftLowCut.template get<2>().coefficients = *cutCoefficients[2];
            leftLowCut.template setBypassed<2>(false);
            break;
        }
```

Zdrojový kód 14 – Část šablonové funkce aktualizující koeficienty filtru low-cut

Zdrojový kód 14 zachycuje část šablonové funkce, která aktualizuje koeficienty filtru typu *ChainType* se zadaným typem koeficientu a sklonem křivky filtru low-cut. Na začátku funkce nastaví všechny filtry levého filtru low-cut na aktuální stav bypassed. Poté nastaví koeficienty a zruší vypnutí konkrétních filtrů na základě hodnoty sklonu křivky. Počet zapnutých filtrů a jejich příslušné koeficienty závisí na hodnotě sklonu křivky. Pokud je hodnota sklonu křivky *Slope_12*, pak se odpojí pouze první filtr a aktualizuje se první koeficient. Podobně pro ostatní hodnoty sklonu křivky jsou konkrétní filtry zapnuty a aktualizovány s odpovídajícími koeficienty.

```
template<int Index, typename ChainType, typename CoefficientType>
void update(ChainType& chain, const CoefficientType& coefficients)
{
    updateCoefficients(chain.template get<Index>().coefficients, coefficients[Index]);
    chain.template setBypassed<Index>(false);
}
```

Zdrojový kód 15 – Nastavení bypass parametru na hodnotu false

Funkcí *updateCoefficients* se aktualizují koeficienty. Pak se nastaví stav bypassed řetězce na *false*.

```
void updateLowCutFilters(const ChainSettings& chainSettings);
void updateHighCutFilters(const ChainSettings& chainSettings);
void updateFilters();
```

Zdrojový kód 16 – Volání funkcí k aktualizování filtrů

Pomocí příkazů v kódu výše se volá funkce k aktualizaci filtrů pro daný řetězec.

4.3.10.2 PluginProcessor.cpp

Nyní do souboru lze implementovat funkce volané z *PluginProcessor.h*.

```
void AudioEQAudioProcessor::updatePeakFilter(const ChainSettings& chainSettings)
{
    auto peakCoefficients = juce::dsp::IIR::Coefficients<float>::makePeakFilter(
        getSampleRate(),
        chainSettings.peakFreq,
        chainSettings.peakQuality,
        juce::Decibels::decibelsToGain(chainSettings.peakGainInDecibels));

    updateCoefficients(leftChain.get<ChainPositions::Peak>().coefficients, peakCoefficients);
    updateCoefficients(rightChain.get<ChainPositions::Peak>().coefficients, peakCoefficients);
}
void AudioEQAudioProcessor::updateCoefficients(Coefficients& old, const Coefficients&
replacements)
{
    *old = *replacements;
}
```

Zdrojový kód 17 – Funkce pro aktualizaci filtru peak

Funkce *updatePeakFilter* aktualizuje peak filtr v řetězci zpracování zvuku, vypočítá koeficienty filtru pro peak filtr za použití vzorkovací frekvence, peak frekvence, faktoru kvality a zesílení v decibelech. Funkce pak aktualizuje koeficienty levého a pravého řetězce pomocí nově vypočtených koeficientů.

Funkce *updateCoefficients* přijímá staré koeficienty a nahradí je novými koeficienty filtru.

```
void AudioEQAudioProcessor::updateLowCutFilters(const ChainSettings& chainSettings)
{
    auto cutCoefficients =
    juce::dsp::FilterDesign<float>::designIIRHighpassHighOrderButterworthMethod
    (chainSettings.lowCutFreq, getSampleRate(), 2 * (chainSettings.lowCutSlope + 1));
    auto& leftLowCut = leftChain.get<ChainPositions::LowCut>();
    auto& rightLowCut = rightChain.get<ChainPositions::LowCut>();

    updateCutFilter(leftLowCut, cutCoefficients, chainSettings.lowCutSlope);
    updateCutFilter(rightLowCut, cutCoefficients, chainSettings.lowCutSlope);

}
void AudioEQAudioProcessor::updateHighCutFilters(const ChainSettings& chainSettings)
{
    auto highCutCoefficients =
    juce::dsp::FilterDesign<float>::designIIRLowpassHighOrderButterworthMethod
    (chainSettings.highCutFreq, getSampleRate(), 2 * (chainSettings.highCutSlope + 1));
    auto& leftHighCut = leftChain.get<ChainPositions::HighCut>();
    auto& rightHighCut = rightChain.get<ChainPositions::HighCut>();
    updateCutFilter(leftHighCut, highCutCoefficients, chainSettings.highCutSlope);
    updateCutFilter(rightHighCut, highCutCoefficients, chainSettings.highCutSlope);
}
void AudioEQAudioProcessor::updateFilters()
{
    auto chainSettings = getChainSettings(apvts);
    updateLowCutFilters(chainSettings);
    updatePeakFilter(chainSettings);
    updateHighCutFilters(chainSettings);
}
```

Zdrojový kód 18 – Nové funkce k aktualizaci filtru high-cut a low-cut

Nově vytvořená funkce *updateLowCutFilter* je zodpovědná za aktualizaci koeficientů filtru low-cut. Nejprve vypočítá koeficienty filtru vysokých frekvencí, a aktualizuje koeficienty filtru předáním odkazů na filtry levého a pravého kanálu.

Funkce *updateHighCutFilters* je podobná předchozí funkci, ale aktualizuje koeficienty filtru high-cut.

Funkce *updateFilters* aktualizuje koeficienty všech tří typů filtrů: peak, low-cut a high-cut voláním nově vytvořených funkcí. Nejprve získá z objektu *apvts* řetězec s daty a poté zavolá ostatní funkce, kterým předá řetězec *Settings* jako argument.

Po zavedení těchto funkcí se může smazat spousta duplikátního a dlouhého kódu, a místo toho lze volat nově vytvořené funkce.

4.3.11 Přidání posuvníků do grafického rozhraní

V této části se implementuje ukládání a načítání stavu parametrů, a přidají se posuvníky do grafického rozhraní ekvalizéru.

4.3.11.1 PluginProcessor.cpp

```
juce::MemoryOutputStream mos(destData, true);
apvts.state.writeToStream(mos);
```

Zdrojový kód 19 – Vytvoření objektu mos, který slouží k zápisu do paměti

Zdrojový kód 19 vytvoří objekt *mos*, který se používá k zápisu dat do vyrovnávací paměti. První parametr slouží k určení vyrovnávací paměti zvuku, do které se má zapisovat, a druhý argument určuje, jestli má být vyrovnávací paměť před zápisem do ní vymazána. Následně se zavolá metoda *writeToStream()*, která zapíše informace o aktuálním stavu objektu *apvts* do vyrovnávací paměti určené pomocí *mos*.

```
auto tree = juce::ValueTree::readFromData(data, sizeInBytes);
if (tree.isValid())
{
    apvts.replaceState(tree);
    updateFilters();
}
```

Zdrojový kód 20 – Kontrola dat z vyrovnávací paměti zvuku

V kódu výše se vytvoří objekt s názvem *tree*, který bude obsahovat informace o vyrovnávací paměti zvuku, ze které se mají data načíst.

Dále se kontroluje, zda je objekt platný. Nakonec je zavolána funkce *updateFilters*, která použije všechny změny provedené v objektu *apvts*.

Tímto se docílí toho, že po úpravách posuvníků v ekvalizéru, zavření ekvalizéru a následného otevření se ekvalizér načte ve stavu, v jakém byl zavřen a ekvalizér tak načte provedené úpravy.

4.3.11.2 PluginEditor.h

```
struct CustomRotarySlider : juce::Slider
{
    CustomRotarySlider() :
    juce::Slider(juce::Slider::SliderStyle::RotaryHorizontalVerticalDrag,
    juce::Slider::TextEntryBoxPosition::NoTextBox){}
};
CustomRotarySlider
        peakFreqSlider,
        peakGainSlider,
        peakQualitySlider,
        lowCutFreqSlider,
        highCutFreqSlider,
        lowCutSlopeSlider,
        highCutSlopeSlider;

std::vector<juce::Component*> getComps();
```

Zdrojový kód 21 – Definování struktury pro parametr otočného posuvníku

Pomocí zdrojového kódu 21 se definuje struktura *CustomRotarySlider*, která dědí ze třídy *juce::Slider*. Konstruktor *CustomRotarySlider* nastaví styl posuvníku na rotační, který lze táhnout buď v horizontálním, nebo vertikálním směru a nebude mít žádné textové pole pro zadávání hodnot. V kódu se dále deklarují instance struktury.

4.3.11.3 PluginEditor.cpp

```
auto bounds = getLocalBounds();
auto responseArea = bounds.removeFromTop(bounds.getHeight() * 0.33);

auto lowCutArea = bounds.removeFromLeft(bounds.getWidth() * 0.33);
auto highCutArea = bounds.removeFromRight(bounds.getWidth() * 0.5);

lowCutFreqSlider.setBounds(lowCutArea.removeFromTop(lowCutArea.getHeight() * 0.5));
lowCutSlopeSlider.setBounds(lowCutArea);

highCutFreqSlider.setBounds(highCutArea.removeFromTop(highCutArea.getHeight() * 0.5));
highCutSlopeSlider.setBounds(highCutArea);

peakFreqSlider.setBounds(bounds.removeFromTop(bounds.getHeight() * 0.33));
peakGainSlider.setBounds(bounds.removeFromTop(bounds.getHeight() * 0.5));
peakQualitySlider.setBounds(bounds);
```

Zdrojový kód 22 – Nastavení hranic jednotlivých parametrů v grafickém rozhraní

V této části kódu se nastavuje hranice a pozice komponent v okně ekvalizéru. První řádek zjistí hranice okna, pak je odebrána horní třetina hranic okna která je přiřazena do *responseArea*. Dále se rozdělí zbývající ohraničení na dvě části: oblast pro low-cut posuvník a high-cut posuvník. Oblast pro low-cut zabírá levou třetinu zbývajícího prostoru a oblast pro high-cut zabírá pravou polovinu. Posuvníkům low-cut filtru a low-cut sklonu je pak přiřazena horní, resp. dolní polovina oblasti low-cut. Posuvníkům high-cut filtru a high-cut sklonu je přiřazena horní a dolní polovina plochy high-cut. Nakonec je posuvníkům peak frekvence, peak zesílení a peak kvality přiřazena horní třetina, polovina a dolní část zbývajících mezí.

```
for (auto* comp : getComps())
{
    addAndMakeVisible(comp);
}

std::vector<juce::Component*> AudioEQAudioProcessorEditor::getComps()
{
    return
    {
        &peakFreqSlider,
        &peakGainSlider,
        &peakQualitySlider,
        &lowCutFreqSlider,
        &highCutFreqSlider,
        &lowCutSlopeSlider,
        &highCutSlopeSlider
    };
}
```

Zdrojový kód 23 – Přidání objektů parametrů do zobrazitelné oblasti

Cyklus *for* přidává jednotlivé objekty z *CustomRotarySlider* do zobrazitelné oblasti. Funkce *getComps* vrací vektor ukazatelů na objekty v *CustomRotarySlider*.

4.3.12 Vykreslení křivky odezvy

Tato část práce obsahuje popis, jak vykreslit křivku odezvy do grafického rozhraní. K tomu je zapotřebí nastavit veškeré věci týkající se řetězce na veřejné, aby bylo možné vytvořit v editoru vlastní instance řetězce. Z důvodu velkého množství složitých úprav a dlouhých zápisů kódu jsou uvedeny pouze nejdůležitější části kódu.

Ačkoli je vhodné používat vestavěné komponenty, je možné vytvořit zcela novou vlastní komponentu. Může se jednat o provádění některých specifických kreslicích úloh nebo o jedinečný prvek uživatelského rozhraní. I s tím si JUCE elegantně poradí (Robinson, 2013).

4.3.12.1 PluginEditor.cpp

```
using namespace juce;
// (Our component is opaque, so we must completely fill the background with a solid colour)
g.fillAll (Colours::black);
auto bounds = getLocalBounds();
auto responseArea = bounds.removeFromTop(bounds.getHeight() * 0.33);
auto w = responseArea.getWidth();
auto& lowcut = monoChain.get<ChainPositions::LowCut>();
auto& peak = monoChain.get<ChainPositions::Peak>();
auto& highcut = monoChain.get<ChainPositions::HighCut>();
auto sampleRate = audioProcessor.getSampleRate();
std::vector<double> mags;
mags.resize(w);
for (int i = 0; i < w; ++i)
{
    double mag = 1.f;
    auto freq = mapToLog10(double(i) / double(w), 20.0, 20000.0);
    if (!monoChain.isBypassed <ChainPositions::Peak>())
        mag *= peak.coefficients->getMagnitudeForFrequency(freq, sampleRate);
    if (!lowcut.isBypassed<0>())
        mag *= lowcut.get<0>().coefficients->getMagnitudeForFrequency(freq, sampleRate);
    if (!lowcut.isBypassed<1>())
        mag *= lowcut.get<1>().coefficients->getMagnitudeForFrequency(freq, sampleRate);
    if (!lowcut.isBypassed<2>())
        mag *= lowcut.get<2>().coefficients->getMagnitudeForFrequency(freq, sampleRate);
    if (!lowcut.isBypassed<3>())
        mag *= lowcut.get<3>().coefficients->getMagnitudeForFrequency(freq, sampleRate);
    if (!highcut.isBypassed<0>())
        mag *= highcut.get<0>().coefficients->getMagnitudeForFrequency(freq, sampleRate);
    if (!highcut.isBypassed<1>())
        mag *= highcut.get<1>().coefficients->getMagnitudeForFrequency(freq, sampleRate);
```

Zdrojový kód 24 – Část kódu, která vykresluje oblast křivky odezvy

Kód výše zachycuje část funkce *paint*, která nejdříve vyplní pozadí grafického rozhraní černou barvou. Poté se definuje *responseArea*, což je oblast pro zobrazení křivky frekvenční odezvy. Tato oblast se vytvoří odstraněním horní části ohraničení komponenty. Metoda zjistí aktuální velikost odezvy filtrů low-cut, peak a high-cut v řetězci monofonního zvuku. Vypočítá se křivka odezvy pro každou frekvenci mezi 20 Hz a 20 kHz a namapuje se křivka

odezvy na oblast pro zobrazení křivky. Kolem této oblasti se nakreslí oranžovou barvou obdélník, do kterého se vykresluje křivka odezvy v oblasti *responseArea* pomocí bílé barvy.

```
void AudioEQAudioProcessorEditor::timerCallback()
{
    if (parametersChanged.compareAndSetBool(false, true))
    {
        //update the monochain
        auto chainSettings = getChainSettings(audioProcessor.apvts);
        auto peakCoefficients = makePeakFilter(chainSettings, audioProcessor.getSampleRate());
        updateCoefficients
        (monoChain.get<ChainPositions::Peak>().coefficients, peakCoefficients);
        auto lowCutCoefficients =
        makeLowCutFilter(chainSettings, audioProcessor.getSampleRate());
        auto highCutCoefficients =
        makeHighCutFilter(chainSettings, audioProcessor.getSampleRate());
        updateCutFilter(monoChain.get<ChainPositions::LowCut>(),
        lowCutCoefficients, chainSettings.lowCutSlope);
        updateCutFilter(monoChain.get<ChainPositions::HighCut>(),
        highCutCoefficients, chainSettings.highCutSlope);
        //signal a repaint
        repaint();
    }
}
```

Zdrojový kód 25 – Překreslování parametrů v grafickém rozhraní na základě změněných hodnot koeficientů parametrů

Účelem funkce *timerCallback* z kódu uvedeného výše je aktualizovat koeficienty filtru v objektu *monoChain* na základě aktuálních hodnot parametrů zvukového procesoru.

Prvním krokem je kontrola, zda se parametry změnily porovnáním hodnoty *parametersChanged* a nastavením této hodnoty na *true*, pokud je porovnání úspěšné. Poté funkce získá aktuální nastavení řetězce zvukového procesoru. Toto nastavení řetězce se pak použije k vytvoření koeficientů peak filtru, low-cut a high-cut filtru. Poté se aktualizují koeficienty peak filtru v objektu *monoChain* voláním *updateCoefficients* a předáním aktuálních koeficientů a nově vytvořených koeficientů peak filtru. Filtry low-cut a high-cut se aktualizují podobným způsobem. Nakonec se zavolá funkce *repaint*, která překreslí uživatelské rozhraní.

4.3.13 Sestavení křivky odezvy

4.3.13.1 PluginEditor.h

```
struct ResponseCurveComponent : juce::Component,
juce::AudioProcessorParameter::Listener, juce::Timer
{
    ResponseCurveComponent(AudioEQAudioProcessor&);
    ~ResponseCurveComponent();
    void parameterValueChanged(int parameterIndex, float newValue) override;

    void parameterGestureChanged(int parameterIndex, bool gestureIsStarting) override { }

    void timerCallback() override;

    void paint(juce::Graphics& g) override;
    private:AudioEQAudioProcessor& audioProcessor;
    juce::Atomic<bool> parametersChanged{ false };
    MonoChain monoChain;
};
```

Zdrojový kód 26 – Definování komponenty pro zobrazení křivky odezvy v grafickém rozhraní

Zdrojový kód 26 definuje vlastní komponentu pro zobrazení křivky odezvy ekvalizéru. Komponenta naslouchá změnám parametrů zvukového procesoru a podle toho aktualizuje své zobrazení. V případě potřeby používá časovač pro překreslení svého obsahu a implementuje funkci *paint* pro vykreslení své grafické reprezentace. Komponenta je také schopna detekovat změny parametrů zvukového procesoru a používá k tomu atomickou funkci *parametersChanged*.

4.3.13.2 PluginEditor.cpp

```
void ResponseCurveComponent::parameterValueChanged(int parameterIndex, float newValue)
{
    parametersChanged.set(true);
}

void ResponseCurveComponent::timerCallback()
{
    if (parametersChanged.compareAndSetBool(false, true))
    {
        //update the monochain
        auto chainSettings = getChainSettings(audioProcessor.apvts);
        auto peakCoefficients = makePeakFilter(chainSettings, audioProcessor.getSampleRate());
        updateCoefficients
        (monoChain.get<ChainPositions::Peak>().coefficients, peakCoefficients);
        auto lowCutCoefficients = makeLowCutFilter
        (chainSettings, audioProcessor.getSampleRate());
        auto highCutCoefficients = makeHighCutFilter
        (chainSettings, audioProcessor.getSampleRate());
        updateCutFilter(monoChain.get<ChainPositions::LowCut>(),
        lowCutCoefficients, chainSettings.lowCutSlope);
        updateCutFilter(monoChain.get<ChainPositions::HighCut>(),
        highCutCoefficients, chainSettings.highCutSlope);
        //signal a repaint
        repaint();
    }
}
```

Zdrojový kód 27 – Aktualizování řetězce zpracování zvuku a volání překreslení grafického rozhraní

První funkce v kódu 27 aktualizuje hodnotu *parametersChanged*, což indikuje, že se parametry změnily.

Druhá funkce provádí aktualizace řetězce zpracování zvuku, když je parametr *parametersChanged* nastaven na hodnotu *true*. Funkce aktualizuje členskou proměnnou *monoChain* voláním různých funkcí pro aktualizaci koeficientů filtru na základě parametrů zpracování zvuku. Nakonec funkce signalizuje překreslení voláním *repaint*.

4.3.14 Úprava posuvníků

V této části jsou uvedeny nejdůležitější části kódu odpovědné za vizuální podobu posuvníků.

4.3.14.1 PluginEditor.h

```
struct RotarySliderWithLabels : juce::Slider
{
    RotarySliderWithLabels
    (juce::RangedAudioParameter& rap, const juce::String& unitSuffix)
    :juce::Slider(juce::Slider::SliderStyle::RotaryHorizontalVerticalDrag,
    juce::Slider::TextEntryBoxPosition::NoTextBox),
    param(&rap), suffix(unitSuffix)
    {
        setLookAndFeel(&lnf);
    }
    ~RotarySliderWithLabels()
    {
        setLookAndFeel(nullptr);
    }
    struct LabelPos
    {
        float pos;
        juce::String label;
    };
    juce::Array<LabelPos> labels;
    void paint(juce::Graphics& g) override;
    juce::Rectangle<int> getSliderBounds() const;
    int getTextHeight() const { return 14; }
    juce::String getDisplayString() const;

    private:
        LookAndFeel lnf;
        juce::RangedAudioParameter* param;
        juce::String suffix;
};
```

Zdrojový kód 28 – Nastavení vzhledu otočných posuvníků

V kódu se definuje vlastní třída posuvníku, která dědí ze třídy *Slider* a implementuje funkce vlastního vzhledu, popisky a zobrazovací řetězec. Třída má také pole s názvem *labels* pro uložení popisků a jejich pozic. Na konci se volají funkce pro vykreslení posuvníků a popisků.

4.3.14.2 PluginEditor.cpp

```
void LookAndFeel::drawRotarySlider(juce::Graphics& g,
    int x,
    int y,
    int width,
    int height,
    float sliderPosProportional,
    float rotaryStartAngle,
    float rotaryEndAngle,
    juce::Slider& slider)
{
    using namespace juce;

    auto bounds = Rectangle<float>(x, y, width, height);

    //color pallete: https://coolors.co/palette/cb997e-ddbea9-ffe8d6-b7b7a4-a5a58d-6b705c

    g.setColour(Colour(183u, 182u, 164u)); //b7b7a4
    g.fillEllipse(bounds);

    g.setColour(Colour(107u, 112u, 92u)); //6b705c
    g.drawEllipse(bounds, 1.f);
    if (auto* rswl = dynamic_cast<RotarySliderWithLabels*>(&slider))
    {
        auto center = bounds.getCentre();
        Path p;

        Rectangle<float> r;
        r.setLeft(center.getX() - 2);
        r.setRight(center.getX() + 2);
        r.setTop(bounds.getY());
        r.setBottom(center.getY() - rswl->getTextBoxHeight() * 1.5);
```

Zdrojový kód 29 – Část kódu odpovědného za vykreslování vlastností otočných posuvníků

Zdrojový kód 29 vykresluje otočný posuvník s popisky. Začíná nastavením barvy pozadí posuvníku na světle šedou barvu a poté vykreslí dráhu posuvníku tmavší šedou barvou. Funkce kontroluje, jakého typu je kreslený posuvník, a případně vykresluje další prvky pro zobrazení popisků posuvníku. Nejprve vypočítá polohu posuvníku a úhel natočení, poté vytvoří dráhu, která bude znázorňovat parametr. Dále kód vytvoří obdélník pro textový

popisek a nastaví jeho barvu pozadí na černou. Poté nakreslí popisek bílou barvou a získá text, který se má zobrazit. Text je umístěn do středu obdélníku.

```
void RotarySliderWithLabels::paint(juce::Graphics& g)
{
    using namespace juce;

    auto startAng = degreesToRadians(180.f + 45.f);
    auto endAng = degreesToRadians(180.f - 45.f) + MathConstants<float>::twoPi;

    auto range = getRange();
    auto sliderBounds = getSliderBounds();
    /*g.setColour(Colours::red);
    g.drawRect(getLocalBounds());
    g.setColour(Colours::yellow);
    g.drawRect(sliderBounds);*/
    getLookAndFeel().drawRotarySlider
    (g,
        sliderBounds.getX(),
        sliderBounds.getY(),
        sliderBounds.getWidth(),
        sliderBounds.getHeight(),
        jmap(getValue(),
            range.getStart(),
            range.getEnd(),
            0.0, 1.0),
        startAng,
        endAng,
        *this);

    auto center = sliderBounds.toFloat().getCentre();
    auto radius = sliderBounds.getWidth() * 0.5f;
```

Zdrojový kód 30 – Část kódu odpovědného za vykreslování otočných posuvníků

Funkce *paint* vykresluje otočné posuvníky na obrazovce a přidává kolem nich popisky. Popisky jsou umístěny po obvodu posuvníku na základě jejich polohy, která je zadána jako hodnota mezi 0 a 1.

```
juce::Rectangle<int> RotarySliderWithLabels::getSliderBounds() const
{
    auto bounds = getLocalBounds();
    auto size = juce::jmin(bounds.getWidth(), bounds.getHeight());
    size -= getTextHeight() * 2;
    juce::Rectangle<int> r;
    r.setSize(size, size);
    r.setCentre(bounds.getCentreX(), 0);
    r.setY(2);

    return r;
}
```

Zdrojový kód 31 – Výpočet velikosti otočného posuvníku

Tato funkce vrací hranice posuvníku v rámci lokálních hranic komponenty, přičemž velikost je upravena s ohledem na výšku popisků.

4.3.15 Mřížka křivky odezvy

Pro lepší přehlednost a informovanost je přidána do okna obsahující křivku odezvy také mřížka, která dle osy X rozděluje určitá pásma frekvencí, a dle osy Y úroveň hlasitosti.

4.3.15.1 PluginEditor.cpp

```
void ResponseCurveComponent::resized()
{
    using namespace juce;
    background = Image(Image::PixelFormat::RGB, getWidth(), getHeight(), true);

    Graphics g(background);

    Array<float> freqs
    {
        20, 30, 40, 50, 100,
        200, 300, 400, 700, 1000,
        2000, 3000, 4000, 6000, 10000,
        14000, 20000
    };

    auto renderArea = getAnalysisArea();
    auto left = renderArea.getX();
    auto right = renderArea.getRight();
    auto top = renderArea.getY();
    auto bottom = renderArea.getBottom();
    auto width = renderArea.getWidth();

    Array<float> xs;
    for (auto f : freqs)
    {
        auto normX = mapFromLog10(f, 20.f, 20000.f);
        xs.add(left + width * normX);
    }
```

Zdrojový kód 32 – Vykreslení mřížky odezvy

Funkce *resized* vytvoří pozadí a ke kreslení na něj použije objekt *Graphics g*. Kód kreslí mřížky pro hodnoty frekvence a zesílení uložené v polích *freqs* a *gain*. Hodnoty frekvencí jsou mapovány z logaritmického měřítka do lineárního měřítka a je vypočtena jejich poloha na ose X. Na těchto pozicích se nakreslí svislé čáry, které představují mřížku frekvencí.

Pro hodnoty hlasitosti se nakreslí vodorovné čáry. Hodnoty hlasitosti se rovněž mapují v rozsahu od -24 do 24 dB na osu Y. Pro každou frekvenci a hodnotu zesílení je vedle příslušné linie mřížky vykresleno textové znázornění. Textové znázornění hodnoty zesílení

je nakresleno dvakrát na obou stranách, přičemž barva je zelená pro 0 dB a světle šedá pro ostatní hodnoty. Textové znázornění hodnot frekvence je nakresleno ve spodní části šedou barvou.

4.3.16 Spektrální analyzátor

V této části se implementuje skutečná křivka odezvy, která představuje grafické znázornění frekvenčního spektra právě zpracovávaného zvuku. Jde tak o nejobsáhlejší část vývoje, jelikož zde musí část kódu odpovědného za zpracovávání zvuku úzce spolupracovat s částí kódu odpovědného za vykreslování.

4.3.16.1 PluginEditor.h

```
enum FFTOrder
{
    order2048 = 11,
    order4096 = 12,
    order8192 = 13
};
```

Zdrojový kód 33 – Definování hodnot Rychlé Fourierovy transformace

FFTOrder definuje tři možné hodnoty pro pořadí Rychlé Fourierovy transformace (FFT[11]). Tyto hodnoty jsou: *order2048* s hodnotou 11, *order4096* s hodnotou 12 a *order8192* s hodnotou 13. *FFTOrder* se používá k určení velikosti výpočtu FFT, přičemž větší velikost FFT vede k většímu frekvenčnímu rozlišení.

[11] FFT – Fast Fourier Transform – matematický algoritmus používaný k transformaci signálu v časové oblasti na jeho frekvenční reprezentaci za účelem analýzy jeho frekvenčních složek

```
template<typename BlockType>
struct FFTDataGenerator
{
    void produceFFTDataForRendering(const juce::AudioBuffer<float>& audioData, const float
negativeInfinity)
    {
        const auto fftSize = getFFTSize();

        fftData.assign(fftData.size(), 0);
        auto* readIndex = audioData.getReadPointer(0);
        std::copy(readIndex, readIndex + fftSize, fftData.begin());

        window->multiplyWithWindowingTable(fftData.data(), fftSize);

        forwardFFT->performFrequencyOnlyForwardTransform(fftData.data());

        int numBins = (int)fftSize / 2;

        for (int i = 0; i < numBins; ++i)
        {
            fftData[i] /= (float)numBins;
        }

        for (int i = 0; i < numBins; ++i)
        {
            fftData[i] = juce::Decibels::gainToDecibels(fftData[i], negativeInfinity);
        }
```

Zdrojový kód 34 – Část kódu odpovědného za vytvoření struktury pro generování dat, které se budou vykreslovat do mřížky

Struktura *FFTDataGenerator* generuje data ve frekvenční oblasti z vyrovnávací paměti zvuku a provádí na nich Ryhlou Fourierovu transformaci (FFT). Obsahuje funkce pro provedení FFT, změnu pořadí FFT a získání dat FFT. Data jsou uložena ve frontě FIFO[12]. Funkce *produceFFTDataForRendering* zkopíruje zvuková data ze vstupní vyrovnávací paměti do vyrovnávací paměti, normalizuje data FFT a převede je na decibely, poté předá data do fronty FIFO. Funkce *changeOrder()* aktualizuje pořadí FFT.

[12] FIFO – First In, First Out – fronta vyrovnávací paměti, která slouží k ukládání zvukových dat v sekvenčním pořadí pro zpracování nebo přehrávání

```
template<typename PathType>
struct AnalyzerPathGenerator
{
    void generatePath(const std::vector<float>& renderData,
        juce::Rectangle<float> fftBounds,
        int fftSize,
        float binWidth,
        float negativeInfinity)
    {
        auto top = fftBounds.getY();
        auto bottom = fftBounds.getHeight();
        auto width = fftBounds.getWidth();

        int numBins = (int)fftSize / 2;

        PathType p;
        p.preallocateSpace(3 * (int)fftBounds.getWidth());

        auto map = [bottom, top, negativeInfinity](float v)
        {
            return juce::jmap(v,
                negativeInfinity, 0.f,
                float(bottom), top);
        };

        auto y = map(renderData[0]);
```

Zdrojový kód 35 – Část kódu odpovědného za generování cesty spektrálního analyzátoru

AnalyzerPathGenerator je další struktura, která generuje cesty analyzátoru na základě dat FFT. Má funkce pro generování cest a získávání cest. Vygenerované cesty jsou uloženy ve frontě FIFO. Funkce *generatePath* přijme data FFT, hranice zobrazení FFT a velikost FFT a vygeneruje cestu namapováním dat FFT z decibelové stupnice na stupnici zobrazení. Vytvoří cestu a nakreslí čáry od každého frekvenčního zásobníku k dalšímu. Počet zahrnutých zásobníků je určen parametrem *pathResolution*. Pak přesune vytvořenou cestu do fronty FIFO.

4.3.16.2 PluginEditor.cpp

```
void PathProducer::process(juce::Rectangle<float> fftBounds, double sampleRate)
{
    juce::AudioBuffer<float> tempIncomingBuffer;
    while (leftChannelFifo->getNumCompleteBuffersAvailable() > 0)
    {
        if (leftChannelFifo->getAudioBuffer(tempIncomingBuffer))
        {
            auto size = tempIncomingBuffer.getNumChannels();
            juce::FloatVectorOperations::copy(monoBuffer.getWritePointer(0, 0),
                monoBuffer.getReadPointer(0, size),
                monoBuffer.getNumSamples() - size);
            juce::FloatVectorOperations::copy
            (monoBuffer.getWritePointer(0, monoBuffer.getNumSamples() - size),
            tempIncomingBuffer.getReadPointer(0, 0), size);
            leftChannelFFTDataGenerator.produceFFTDataForRendering(monoBuffer, -48.f);
        }
    }

    const auto fftSize = leftChannelFFTDataGenerator.getFFTSize();
    const auto binWidth = sampleRate/ (double)fftSize;
    while (leftChannelFFTDataGenerator.getNumAvailableFFTDataBlocks() > 0)
    {
        std::vector<float> fftData;
        if (leftChannelFFTDataGenerator.getFFTData(fftData))
        {
            pathProducer.generatePath(fftData, fftBounds, fftSize, binWidth, -48.f);
        }
    }
    while (pathProducer.getNumPathsAvailable())
    {
        pathProducer.getPath(leftChannelFFTPath);
    }
}
```

Zdrojový kód 36 – Předávání zvukových dat ve vyrovnávací paměti a zpracování dat spektrálním analyzátorem

V této části kódu se provádí analýza FFT na příchozích zvukových datech. Funkce nejprve zkontroluje, zda jsou ve frontě k dispozici vyrovnávací paměti pro audio a zkopíruje zvuková data z vyrovnávací paměti do vyrovnávací paměti monofonního zvuku. Poté jsou

zvuková data zpracována přes generátor dat FFT, který převádí data v časové oblasti do oblasti frekvenční. Výsledná FFT data jsou poté předána dalšímu objektu s názvem *pathProducer*, který na základě FFT dat a zadaných parametrů vygeneruje cestu. Nakonec je vygenerovaná cesta načtena a uložena do objektu *leftChannelFFTPath*.

4.3.16.3 PluginProcessor.h

```
template<typename T>
struct Fifo
{
    void prepare(int numChannels, int numSamples)
    {
        static_assert(std::is_same_v<T, juce::AudioBuffer<float>>,
            "prepare(numChannels, numSamples) should only be used when the Fifo is holding
juce::AudioBuffer<float>");

        for (auto& buffer : buffers)
        {
            buffer.setSize(numChannels, numSamples, false, true, true);
        }
    }

    void prepare(size_t numElements)
    {
        static_assert(std::is_same_v<T, std::vector<float>>,
            "prepare(numElements) should only be used when the Fifo is holding
std::vector<float>");
        for (auto& buffer : buffers)
        {
            buffer.clear();
            buffer.resize(numElements, 0);
        }

    }
```

Zdrojový kód 37 – Definování struktury Fifo, která připraví vyrovnávací paměť pro ukládání a načítání dat

V kódu výše se definuje struktura s názvem *Fifo*. Struktura obsahuje několik funkcí. První funkce *prepare* připraví vyrovnávací paměť pro uložení dat nastavením velikosti paměti dle počtu kanálů a vzorků. Druhá funkce *prepare* připraví vyrovnávací paměť pro uložení

vektorových dat vymazáním vyrovnávací paměti a změnou její velikosti na zadaný počet prvků. Funkce *push* zapíše do vyrovnávací paměti data a vrátí logickou hodnotu, která udává, zda byla operace úspěšná. Funkce *pull* vytáhne data z vyrovnávací paměti a vrátí logickou hodnotu udávající, zda byla operace úspěšná. Funkce *getNumAvailableForReading* vrací počet objektů typu T obsahující data, které jsou v paměti k dispozici pro čtení.

```
template<typename BlockType>
struct SingleChannelSampleFifo
{
    SingleChannelSampleFifo(Channel ch) : channelToUse(ch)
    {
        prepared.set(false);
    }

    void update(const BlockType& buffer)
    {
        jassert(prepared.get());
        jassert(buffer.getNumChannels() > channelToUse);
        auto* channelPtr = buffer.getReadPointer(channelToUse);

        for (int i = 0; i < buffer.getNumSamples(); ++i)
        {
            pushNextSampleIntoFifo(channelPtr[i]);
        }
    }
```

Zdrojový kód 38 – Část kódu odpovědného za správu vyrovnávací paměti

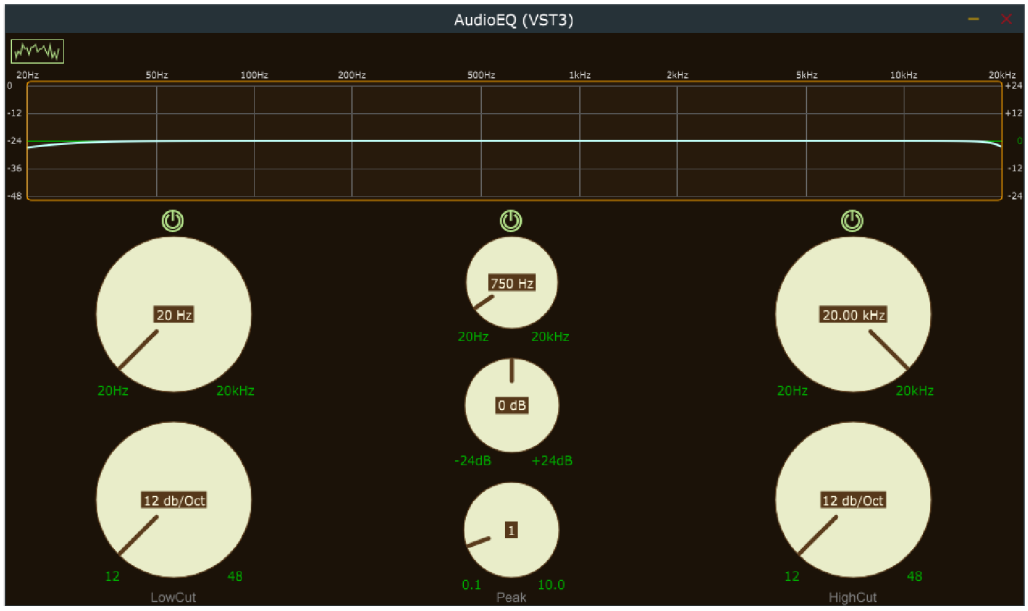
Struktura v tomto kódu obsahuje několik členských proměnných a funkcí pro správu zvukové vyrovnávací paměti FIFO se zvukovými daty z jednotlivých kanálů. Členská proměnná *channelToUse* uchovává kanál zvukových dat, který se má použít ze vstupní vyrovnávací paměti zvuku. Členská proměnná *audioBufferFifo* obsahuje objekt *FIFO*, který obsahuje vyrovnávací paměť audia.

Proměnná *bufferToFill* je vyrovnávací paměť, která dočasně uchovává vzorky vstupní zvukové vyrovnávací paměti před jejich přesunutím do *audioBufferFifo*. Členské proměnné *prepared* a *size* slouží k uložení stavu a velikosti vyrovnávací paměti. Funkce *prepare* nastavuje stav a velikost vyrovnávací paměti FIFO a funkce *getAudioBuffer* načítá data vyrovnávací paměti *audioBufferFifo*. Funkce *update* přesune vzorky vstupní vyrovnávací

paměti audia do *bufferToFill* a funkce *pushNextSampleIntoFifo* přesune naplněnou vyrovnávací paměť do objektu *FIFO*.

5 Výsledky a diskuse

Výsledkem práce je třípásmový ekvalizér. Disponuje parametry low-cut (vlevo), high-cut (vpravo) a parametrem bell (uprostřed). Parametry jsou schopny poupravit zvuk v rozmezí od 20 Hz do 20 kHz na ose X a vybraný rozsah frekvencí parametrem bell je možné zesílit nebo zeslabit v rozsahu od -24 dB do 24dB.



Obrázek 14 – Výsledná podoba ekvalizéru (vlastní zdroj)

5.1 Low-cut a high-cut

Parametry low-cut a high-cut mají na sebe navázaný parametr kvality sklonu křivky, který může mít hodnotu 12 dB, 24 dB, 36 dB nebo 48 dB. Tento parametr určuje sklon, respektive kvalitu sklonu křivky vyříznutí frekvencí. Čím větší je tato hodnota, tím strmější je křivka a zvuk tedy neobsahuje tolik vzdálených frekvencí od místa určeného parametrem low-cut nebo high-cut. Vybrané hodnoty parametrů se zobrazují na grafickém rozhraní parametru. Ty obsahují i tlačítko bypass, se kterým lze vypínat a zapínat provedené změny parametrů. Pomocí těchto parametrů lze do značné míry měnit charakter zvuku a efektivně jej modifikovat.

5.2 Parametr bell

Parametr bell má na sebe navázané dva parametry: první určuje míru zesílení nebo zeslabení parametru bell v decibelech, druhý určuje rozsah frekvencí parametru bell. Tato skupina parametrů disponuje přepínačem bypass pro vypínání a zapínání provedených změn. Tento parametr umožňuje odstranit rušivé a nepříjemné frekvence, zároveň s ním lze frekvence zesilovat.

5.3 Křivka frekvenčního spektra

Ekvalizér dále obsahuje tlačítko pro vypínání a zapínání křivky frekvenčního spektra. To se hodí pro zmenšení vlivu vizuálních prvků frekvencí zvuku na rozhodnutí člověka pracujícího s ekvalizérem, a díky tomu se může zvukový inženýr soustředit skutečně na to, jak zvuk po provedených změnách zní.

Při zapnutém spektru se v ekvalizéru v reálném čase vykresluje frekvenční spektrum zpracovaného zvuku. Ve spektru je barevně odlišen levý kanál, pravý kanál a stereo kanál.

5.4 Možná rozšíření ekvalizéru

Ekvalizér lze rozšířit o mnoho dalších užitečných funkcí, které se během hudební produkce a úpravě zvuku využívají. Implementace rozdělení parametrů na mono a stereo kanály by umožnila přesnější úpravu zvuku. Menu s několika předdefinovanými hodnotami nastavení ekvalizéru, které se v produkci často využívají, by ušetřilo čas a umožnilo rychleji experimentovat se zvukem. Do parametrů by mohl být zakomponován kompresor, díky čemuž by bylo možné nastavit, aby se frekvence v daném rozsahu modifikovaly pouze nad určitou úrovní hlasitosti.

6 Závěr

V rámci bakalářské práce se dosáhlo vytyčených cílů: vznikla první verze ekvalizéru, která disponuje základními vlastnostmi a přehledným grafickým rozhraním. Ekvalizér lze použít jako samostatnou aplikaci, nebo jako zásuvný modul typu .vst3, který se využívá v digitálních zvukových pracovních stanicích a je podporován na několika operačních systémech.

V teoretické části práce je čtenář seznámen se základními pojmy a fakty z oblasti programování v jazyce C++, z oblasti zvuku, kde je objasněna charakteristika zvuku, jeho vnímání a je vysvětleno, jak je zvuk zpracován počítačem. Taktéž je čtenář seznámen s ekvalizérem a tím, jak se ovládá.

V rámci vlastní práce je čtenář obeznámen se samotným postupem vývoje ekvalizéru. V této části se čtenář dozvídá, jak nainstalovat JUCE framework a jak s ním pracovat. Čtenář je seznámen se strukturou projektu a dílčími kroky vývoje. Nejprve je práce zaměřena na funkční část ekvalizéru, kde je uvedeno, jak JUCE framework přistupuje ke zpracování zvuku a jak zvuk zpracovává ekvalizér. Dále je vlastní práce zaměřena spíše na grafickou část ekvalizéru, kde je nastíněn vývoj grafických prvků ekvalizéru a toho, jak vypadá a působí. V závěru vlastní práce se pak tyto dvě odlišné části prolínají, jelikož spolu úzce souvisí.

Výsledný ekvalizér je přehledný, jednoduchý na používání a rychlý, což může hrát v hudební produkci zásadní roli ve výkonu počítače a následně pak rozhodnutí, zda takový plugin producent během své práce využije, nebo bude pracovat s jiným. Ekvalizér má přehledný kód a lze jej snadno rozšířit o další funkce.

7 Seznam použitých zdrojů

Alex. 2019. Pass Filter Explained: What Is A Low-Pass Filter And A High-Pass Filter? *Mixing Lessons.* [Online] 26. Říjen 2019. [Citace: 12. Prosinec 2022.] https://www.mixinglessons.com/pass-filter/.

Armada Music. EQ Explained: The Basics. *Armada University.* [Online] Armada Music. [Citace: 10. Prosinec 2022.] https://www.armadamusic.com/university/music-production-articles/eq-explained-the-basics.

DOSITS Team. Frequency. *Discovery of Sound in the Sea (DOSITS).* [Online] DOSITS. [Citace: 1. Prosinec 2022.] https://dosits.org/science/sound/characterize-sounds/frequency/.

—. How is hearing measured? *Discovery of Sound in the Sea (DOSITS).* [Online] DOSITS. [Citace: 1. Prosinec 2022.] https://dosits.org/science/measurement/how-is-hearing-measured/.

—. Intensity. *Discovery of Sound in the Sea (DOSITS).* [Online] DOSITS. [Citace: 1. Prosinec 2022.] https://dosits.org/science/sound/characterize-sounds/intensity/.

—. Wavelength. *Discovery of Sound in the Sea (DOSITS).* [Online] DOSITS. [Citace: 1. Prosinec 2022.] https://dosits.org/science/sound/characterize-sounds/wavelength/.

—. What is sound? *Discovery of Sound in the Sea (DOSITS).* [Online] DOSITS. [Citace: 1. Prosinec 2022.] https://dosits.org/science/sound/what-is-sound/.

Feist, Jonathan. 2021. Digital Audio Basics 1: What You Need to Know. *Waves.* [Online] Waves Audio Ltd., 9. Listopad 2021. [Citace: 5. Prosinec 2022.] https://www.waves.com/digital-audio-basics-1-what-you-need-to-know.

Frontiers for Young Minds. 2022. What Makes Human Hearing Special? *Frontiers for Young Minds.* [Online] Frontiers Media SA, 17. Leden 2022. [Citace: 3. Prosinec 2022.] https://kids.frontiersin.org/articles/10.3389/frym.2022.708921.

Jonathan. EQ Shelving : High Shelf And Low Shelf Filters Explained. *Reboot Recording.* [Online] [Citace: 12. Prosinec 2022.] https://rebootrecording.com/high-and-low-shelf/.

Laukkonen, Jeremy. 2021. What Are VST Plugins and What Do They Do? *Lifewire.* [Online] 9. Červen 2021. [Citace: 14. Prosinec 2022.] https://www.lifewire.com/what-are-vst-plugins-4177517.

Leffler, S. 2018. The Top Audio Plug-In Design Mistakes: Don't Fall for These! *SonicScoop.* [Online] 1. Únor 2018. [Citace: 28. Listopad 2022.] https://sonicscoop.com/2018/02/01/top-audio-plug-design-mistakes-dont-fall/.

Lippman, Stanley Bruce, Lajoie, Josée a Moo, Barbara E. 1998. *C++ Primer.* Reading : Addison-Wesley, 1998. 0201824701.

Mozilla Contributors. 2021. Object-oriented programming. *MDN Web Docs.* [Online] Mozilla Corporation, 23. Červenec 2021. [Citace: 28. Listopad 2022.] https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object-oriented_programming.

Pirkle, Will. 2019. *Designing Audio Effect Plugins in C++ For AAX, AU, and VST3 With DSP Theory.* New York : Routledge, 2019. str. 34. 978-1-138-59193-6.

—. 2013. *Designing Audio Effect Plug-Ins in C++ With Audio Signal Processing Theory.* Burlington : Focal Press, 2013. str. 501. 978-0-240-82515-1.

Robinson, Martin. 2013. *Getting Started with JUCE.* místo neznámé : Packt Publishing, 2013. 978-1783283316.

Rouse, Margaret. 2022. Sound Wave. *TechTarget.* [Online] TechTarget, Červen 2022. [Citace: 29. Listopad 2022.] https://www.techtarget.com/whatis/definition/sound-wave.

Schrag, A. 2018. How to use JUCE as a plugin host. *Medium.* [Online] 28. Září 2018. [Citace: 29. Listopad 2022.] https://medium.com/@AndrewSchrag/how-to-use-juce-as-a-plugin-host-75b2e2c9d3ab.

Smith, Steven W. 1999. *The Scientist and Engineer's Guide to Digital Signal Processing.* San Diego : California Technical Publishing, 1999. 0-9660176-6-8.

The Audacity Team. Digital Audio. *Audacity Manual.* [Online] Audacity Team. [Citace: 10. Prosinec 2022.] https://manual.audacityteam.org/man/digital_audio.html.

The Physics Classroom. Sound Waves and Music. *The Physics Classroom.* [Online] [Citace: 2. Prosinec 2022.] https://www.physicsclassroom.com/Class/sound/u11l2a.cfm.

The Sleep Foundation. 2021. Can Pink Noise Help You Sleep? *Sleep Foundation.* [Online] National Sleep Foundation, 25. Březen 2021. [Citace: 3. Prosinec 2022.] https://www.sleepfoundation.org/noise-and-sleep/pink-noise-sleep.

Vincent, Robin. 2022. What Is a VST? All Your Questions Answered. *CareersInMusic*. [Online] 2. Září 2022. [Citace: 13. Prosinec 2022.] https://www.careersinmusic.com/what-is-a-vst/.

Wilson, Alex. 2021. Shelf EQ vs. Bell EQ: When and How to Use Them Both. *Flypaper*. [Online] 13. Červenec 2021. [Citace: 12. Prosinec 2022.] https://flypaper.soundfly.com/produce/shelf-eq-vs-bell-eq-when-and-how-to-use-them-both/.

8 Seznam obrázků, tabulek a zkratek

8.1 Seznam obrázků

8.2 Seznam tabulek

8.3 Seznam zdrojových kódů

8.4 Seznam použitých zkratek

VST – Virtual Studio Technology

AU – Audio Unit

AAX – Avid Audio Extension

LV2 – Linux Audio Developer's Simple Plugin API v2

ASIO – Audio Stream Input/Output

WASAPI – Windows Audio Session API

ALSA – Advanced Linux Sound Architecture

MIDI – Musical Instrument Digital Interface

WAV – Waveform Audio File

MP3 – Moving Picture Experts Group Layer-3 Audio

FLAC – Free Lossless Audio Codec

RTAS – Real Time Audio Suite

VoIP – Voice over Internet Protocol

PCM – Pulse-Code Modulation

MPEG – Moving Picture Experts Group

CD – Compact Disc

DVD – Digital Video Disc

HD – High Definition

LPCM – Linear Pulse Code Modulation

VSTi – Virtual Studio Technology Instrument

IDE – Integrated Development Environment

GUI – Graphical User Inteface

DSP – Digital Signal Processing

API – Application Program Interface

IIR – Infinite Impulse Response

FFT – Fast Fourier Transformation

FIFO – First In, First Out

9 Přílohy

Soubory zdrojového kódu a samotného ekvalizéru ve formátu VST3 jsou dostupné v přiloženém souboru Ekvalizer.zip. Obsah přiloženého souboru netvoří funkční celek. Všechny soubory tvořící funkční celek nejsou přiloženy z důvodu kapacitních omezení úložiště fakulty. Ekvalizér je funkční v případě otevírání v hostiteli.