TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Arvutiteaduse instituut

# VST heliefektide ülesehitus ja arendamine „JUCE“ raamistikul

Bakalaureusetöö

Üliõpilane: Matis Oolup
Üliõpilaskood: 103692IAPB
Juhendaja: Jaagup Irve

Tallinn
2015

# Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

---------- ----------

(*kuupäev*) (*allkiri*)

# Annotatsioon

Bakalaureusetöö töö põhieesmärgiks on VST *plugina* loomine C++ programmeerimiskeeles, kasutades JUCE raamistikku. Lisaeesmärgina luuakse VST *plugin* ilma raamistikku kasutamata, et paremini mõista nende ehitust ja arendada autori programmeerimisoskusi.

Töö käigus püütakse aru saada digitaalse helisignaali töötluse põhimõtetest, mis on VST *pluginate* olemuseks, ning jõuda selguseni, kuidas neid kasutatakse helitöötluseks.

Tulemustes tõi autor välja VST *pluginate* arendamise keerukuse nende testimise, C++ keele ja ülesehituse tõttu. JUCE raamistikul *plugina* kirjutamise kohta on autor väitnud, et selle kasutamine lihtsustas oluliselt arendusprotsessi, kuna oli võimalik keskenduda olulisemale ehk DSP sidumisele kasutajaliidesega. Lisaks on mainitud, et ilma C++ juurde õppimiseta ei oleks arendatud *pluginate* valmimine võimalikuks saanud.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 31 leheküljel, 5 peatükki ja 12 joonist.

# Abstract

The main purpose of the thesis is to write a VST plug-in using C++ programming language and JUCE framework. Also to create a separate plug-in for understanding the underlying principles of VST plug-ins and to improve the author's programming skills.

During the course of writing this thesis, the author is trying to understand the principles of DSP, which are requried for writing VST plug-ins.

As the results, the author has mentioned that writing VST plug-ins is not an easy task because of the complicated testing involved, complexity of the C++ language and also the complex nature of VST plug-ins. About the development by using the JUCE framework, the author has said that the process is easier because JUCE lets you focus more on the important aspects of development, such as integrating the DSP algorithms to the user interface. It is also mentioned that without learning more about the C++ language, the development for neither of the plug-ins would not have been possible.

The thesis is in Estonian and contains 31 pages of text, 5 chapters and 12 figures.

# Lühendite ja mõistete sõnastik

**ADC** — ***Analog to Digital Converter***
Seade, mis muundab analoogsignaali digitaalseks.

**C++** — ***C++ Programming language***
C++ programmeerimiskeel.

**DAC** — ***Digital to Analog Converter***
Seade, mis muundab digitaalse signaali analoogseks.

**DAW** — ***Digital Audio Workstation***
Digitaalne heli salvestuse ja produtseerimise seade või tarkvara rakendus [1].

**DSP** — ***Digital Signal Processing***
Digitaalne signaali töötlemine.

**GUI** — ***Graphical User Interface***
Graafiline kasutajaliides.

**IDE** — ***Integrated Development Environment***
Tarkvara arenduskeskkond.

**JUCE** — ***Cross-platform C++ application framework***
Platvormiülene, C++ keelne tarkvaraarenduse raamistik [2].

**MIDI** — ***Musical Instrument Digital Interface***
Andmevahetusprotokoll muusikalise info jaoks [3].

**Plugin** — ***Plug-In***
Tarkvara rakendus, mis pakub mõnele teisele rakendusele lisafunktsionaalsust [4].

| | |
|---|---|
| **SDK** | ***Software Development Kit***<br>Tarkvara arenduse pakett kindlat tüüpi rakenduste programmeerimiseks. |
| **VST** | ***Virtual Studio Technology***<br>Tarkvara liides, mis seob omavahel tarkvaralised helisüntesaatorid ning heliefektid helisalvestussüsteemidega [5]. |

# Sisukord

# Sissejuhatus

Virtuaalsetes helistuudiotes muusikat produtseerides kasutatakse laialdaselt VST *pluginaid*, mis implementeerivad erinevaid heliefekte [6]. Töö autor kasutab muusika tegemiseks ise programmi Reaper, mis toetab VST formaadis olevaid *pluginaid*. Kuna tekkis vajadus helifiltri *plugina* järgi, mis ei sisaldaks peale filtri muud funktsionaalsust ning mille resonantsi väärtuste vahemik oleks suurem kui autori poolt siiani kasutatud filtritel, otsustas autor käesoleva töö käigus programmeerida vastava *plugina* ning seeläbi ka arendada enese C++ oskusi. Valmivat *pluginat* oleks võimalik kasutada autoril endal kui ka teistel VST kasutajatel.

Tulemi saamiseks on autor püstitanud järgmised eesmärgid:

- Tutvuda DSP tööpõhimõtetega, mis on vajalikud VST *pluginate* arendamiseks;
- Saada ülevaade C++ keeles kirjutatud VST *pluginate* ehitusest;
- Arendada *plugin*, mis oleks kasutatav muusika tegemisel virtuaalses helistuudios.

Valmivate *pluginate* kirjutamiseks otsustas autor kasutusele võtta programmeerimiskeele C++, kuna kirjutatavad *pluginad* töötaksid reaalajas, mispärast keele kiirus on oluliseks aspektiks ning C++ on tuntud eelkõige oma mälukasutuse poolest kiire programmeerimiskeelena.

*Pluginad* arendatakse Windows 7 platvormil, kasutades CodeBlocks ning Visual Studio IDE-sid.*Pluginaid* testitakse Reaper-i nimelisel DAW-il (*Digital Audio Workstation*).

Antud töö on jaotatud neljaks osaks. Esimeses ja teises osas tutvustab autor teooriat, mida VST *pluginad* endast kujutavad ning digitaalse signaalitöötluse põhimõtteid. Kolmandas ja neljandas osas käsitleb autor VST *pluginate* loomist kahel erineval viisil.

# 1. VST ülevaade

Järgnevate alapeatükkide käigus kirjeldatakse lähemalt, mida kujutab endast VST tehnoloogia.

## 1.1 VST Tutvustus

VST on tarkvara liides, mis integreerib tarkvaralised heli süntesaatorid ja efekti *pluginad* helitöötlus- ja salvestamissüsteemidega. VST ning teisedki sarnased tehnoloogiad kasutavad digitaalset helitöötlust simuleerimaks traditsioonilist salvestusstuudio riistvara tarkvara näol. *Pluginaid* eksisteerib nii kommerts- kui ka vabavarana ja suur osa helirakendustest toetavad VST-d tema looja (Steinberg) litsentsi alusel. [5].

### 1.1.1 VST *pluginad*

VST *pluginaid* on kolme tüüpi:

1. VST instrumendid – Genereerivad heli ja on kas virtuaalsed süntesaatorid või sämplerid. Paljud neist üritavad taasluua populaarsete riistvaraliste süntesaatorite heli ja välimust.

2. VST efektid – Üldiselt töötlevad, mitte heli genereerivad *pluginad.* Need täidavad samu funktsioone nagu riistvaralised heliprotsessorid, näiteks kajaefekt või ekvalaiser. On olemas ka monitooringu efektid, mis on suutelised andma visuaalset tagasisidet sisendsignaali kohta seda töötlemata.

3. VST MIDI efektid – Töötlevad MIDI sõnumeid ja suunavad MIDI andmed teistele VST instrumentidele või riistvara seadmetele. [5].

VST *plugina* lähtekood on platvormiülene, kuid omab platvormi arhitektuurile omast laiendit:

- Windowsi platvormil on selleks mitmelõimeline DLL formaat;

- OS X puhul on formaadiks Mach-O Bundle. [5].

### 1.1.2 DAW

VST juhtprogrammiks on üldiselt mõni DAW, mis kujutab endast virtuaalset helistuudiot. DAW esitab *plugina* kasutajaliidest ja juhib digitaalset heli ning MIDI sõnumeid nii VST

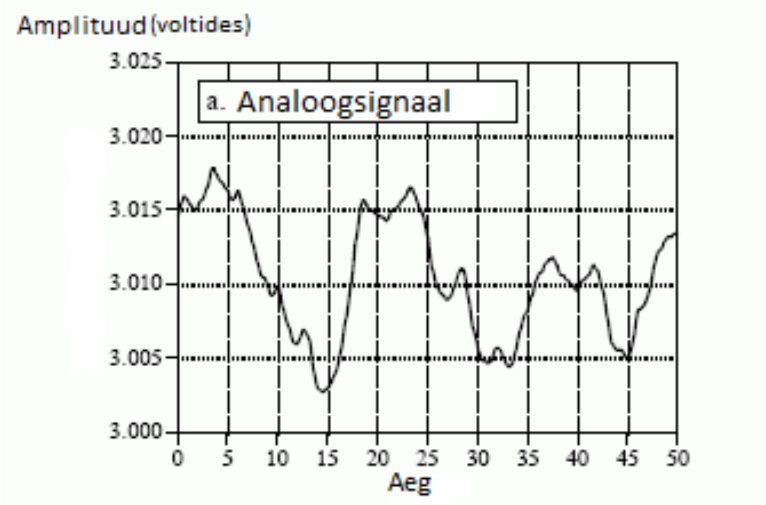
*pluginasse* kui ka sealt välja. [5]. Näiteid populaarsetest DAW-idest: Ableton Live, FL Studio ja Reaper.

# 2. Digitaalse helisignaali töötlus

VST *pluginad* töötlevad digitaalset heli ning selleks, et mõista nende ülesehitust, on vajalik uurida digitaalse helisignaali tööpõhimõtteid, mida järgnevalt ka tutvustatakse.

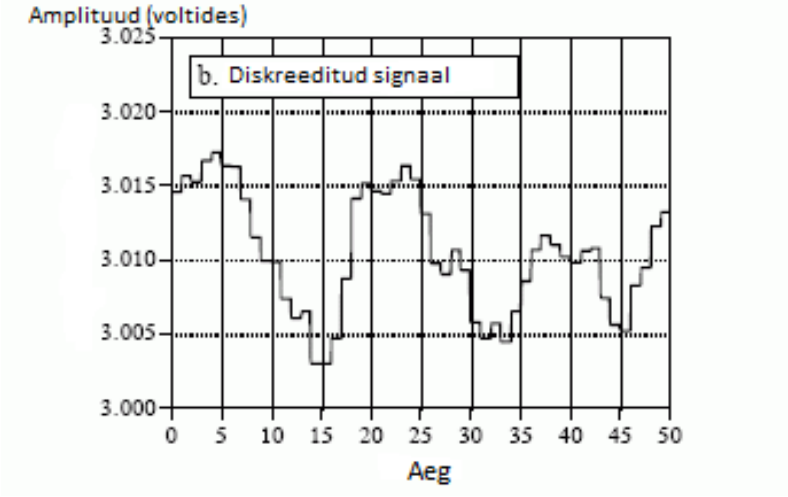
## 2.1 Digitaalse helisignaali tööpõhimõtted

Digitaalne diskreetimine on andmepunktide omandamine ajaliselt pidevalt analoogsignaalilt. Joonisel 1 on kujutatud pinge analoogsignaali, mis muutub ajas katkematult. [6].



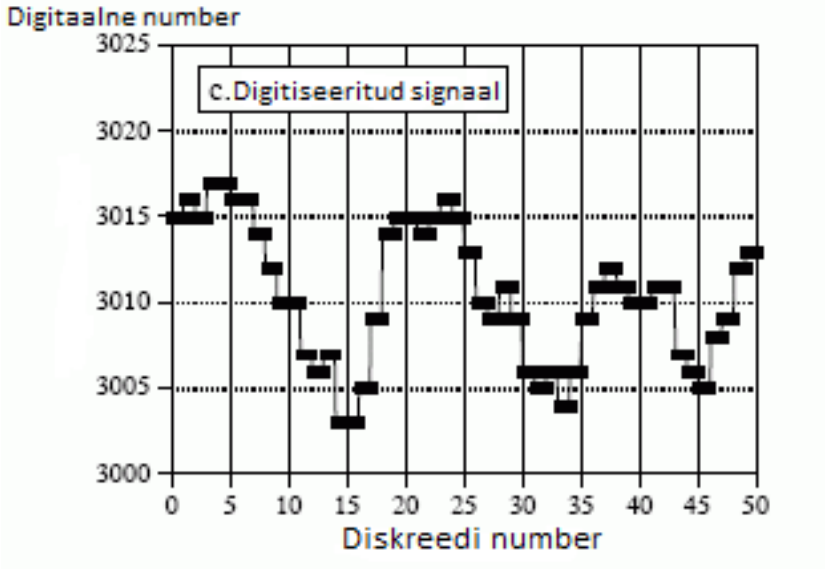

**Joonis 1 – Analoogsignaal [7]**

Joonisel 1 toodud analoogsignaali diskreetimisel saadakse signaal, mida on kujutatud joonisel 2 ning mis ei ole ajas enam pidev.




**Joonis 2 – Diskreeditud signaal [7]**

Andmepunkte diskreeditakse korrapärasel intervallil, mida kutsutakse diskreetimise intervalliks või perioodiks. Diskreetimisperioodiga pöördvõrdeline mõiste on diskreetimissagedus, mis määrab pideva signaali diskreetimisel diskreetide arvu sekundis. Näiteks CD-d kasutavad 44100 hertsilist diskreetimissagedust ehk toodetakse 44100 diskreeti kanali kohta sekundis ning diskreetimisintervall on umbes 22,7 mikrosekundit [6]. Joonisel 3 on võimalik täheldada digitiseeritud signaali, mis on saadud Joonisel 2 toodud signaali kvantiseerimisel (*quantization*). Jooniselt 3 on näha, et signaal koosneb nüüd kindlatest andmepunktidest ehk diskreetidest ning saadud on digitaalne signaal.




**Joonis 3 - Digitiseeritud signaal [7]**

Peale analoogsignaali muundamist digitaalseks, on oluline mõista diskreetimise teooriat, mis ütleb, et pidevat analoogsignaali on võimalik diskreetida diskreetseteks andmepunktideks ja seejärel rekonstrueerida tagasi esialgseks analoogsignaaliks ilma igasuguse informatsiooni kaota. Diskreeditud andmed võivad tunduda rikutud, kuid teoreem peab paika, kuni sisendsignaal on filtreeritud, nii et ta ei sisalda kõrgemaid sagedusi kui pool diskreetimise sagedust, mis on tuntud ka kui Nyquisti sagedus. Filtreerimise all peetakse silmas madalsagedusfiltri kasutamist. [6].

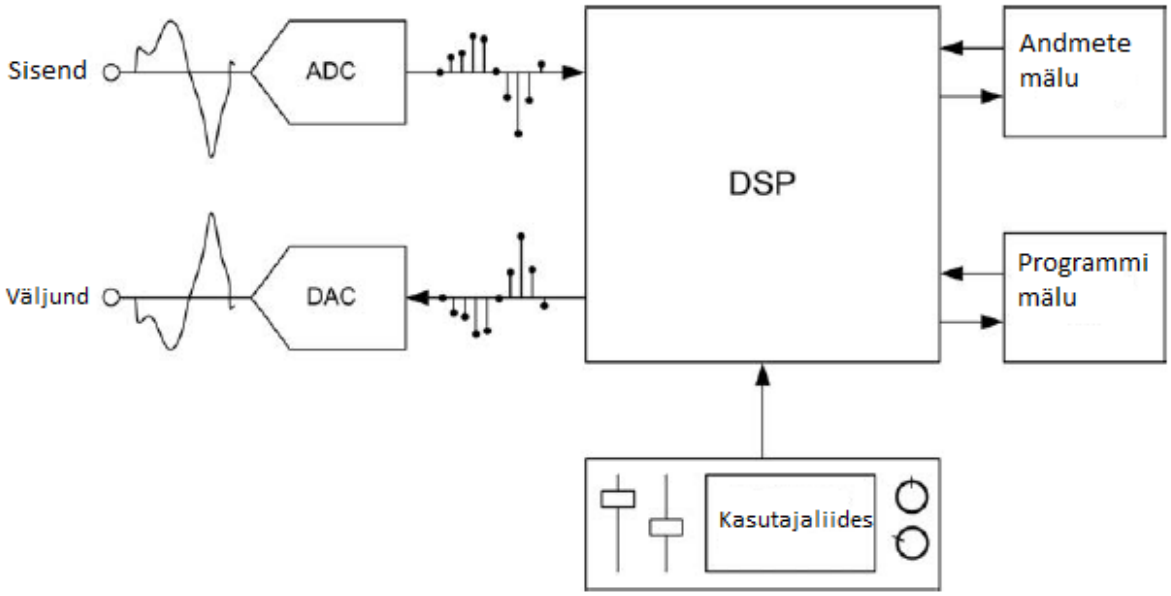
## 2.2 Digitaalsignaali töötlev süsteem

Signaali töötlemise süsteemid kombineerivad andmete omandamise seadmeid mikroprotsessoritega, mis jooksutavad matemaatilisi algoritme saadud heli andmetel. *Pluginad*

on järeltulijad eraldiseisvatele riistvaralistele efektiprotsessoritele, mis on disainitud peamiselt ainult DSP (*Digital Signal Processor*) algoritmide jooksutamiseks. DSP seadmed sisaldavad endas keskset protsessorit, mis on disainitud andmete paljundamiseks ja kogumiseks. Kuna see protsess on korduv ja fundamentaalne osa DSP algoritmidest, kasutavad modernsed DSP-d konveiermeetodit, et saada järgmised andmed ning samal ajal töödelda kehtivaid andmeid. Selline meetod parandab oluliselt süsteemi efektiivsust. [6].

Tüüpiline signaali töötlemissüsteem koosneb järgnevatest osadest:

- Andmete omandamise seade ( ADC ja DAC);
- DSP kiip;
- Mälu algoritmi programmi jaoks (programmi mälu);
- Mälu algoritmi andmete jaoks (andmete mälu) kasutajaliides. [6].




**Joonis 4 - Signaali töötlev süsteem [6]**

Joonisel 4 on näitena kujutatud lihtsat helitöötlussüsteemi, mis inverteerib signaali faasi (väljund on kujutatud vastupidiselt sisendsignaalile) [6].

### 2.2.1 Sünkronisatsioon ja katkestused

VST *pluginad*, mida vaadeldakse antud uurimistöö käigus, kasutavad asünkroonseid operatsioone, kus heli andmed ei ole sünkroonis DSP-ga. Teisisõnu, sisend-ja väljundbittide

vood ei pruugi tegutseda samal kella (*clock*) signaalil. Täielikult sünkroonne süsteem oleks töökindlam, kuid vähem paindlikum. Valmistavate *pluginate* süsteem on katkestuste põhine, kus protsessori katkestus töötab justkui uksekell. Kui mõnel seadmel, näiteks ADC-l on andmeid, mis on edastuseks valmis, paneb ADC need eeldisainitud puhverisse ja "annab kella". Protsessor teenindab seda katkestust teenindava funktsiooniga *(interrupt handler*), mis võtab andmed vastu ja läheb edasi andmete töötlemise koodiga. Teine allikas katkestustele on kasutajaliides, kus läbi kasutajaliidese tehtud muudatused, saadetakse DSP-le, mis kohandab oma töötlemise protsessi, vastamaks tehtud muudatustele. [6].

### 2.2.2 Digitaalsignaali töötlemise töövoog

Järgnevalt on välja toodud digitaalsignaali töötlemise töövoo osad:

- Ühekordne initsialiseerimise funktsioon, mis seab valmis protsessori esialgse töötluse oleku ja valmisoleku sissetulevate andmete katkestusteks;
- Lõputu *while*-tsükkel, mis ootab katkestuste toimumist;
- Katkestuste käsitleja, mis dekodeerib katkestused ja otsustab, kuidas töödelda või hoopis ignoreerida "uksekella";
- Andmete lugemise ja kirjutamise funktsioonid nii juhtimise kui ka andmete informatsiooniks;
- Töötlemise funktsioon heli väljundi manipuleerimiseks;
- Funktsioon, et valmis seada muutujad järgmiseks tsükli iteratsiooniks, muutes neid vastavalt kasutajaliidese kaudu tehtud muudatustele, kui vaja. [6].

Reaalajas töötlemiseks peab algoritm olema suuteline vastu võtma sisendandmed, töötlema neid ja teha väljundandmed kättesaadavaks enne kui uus sisend saabub. Kui töötlemine võtab liialt kaua aega võib väljundis tekkida heli katkeid, müra jne. [6].

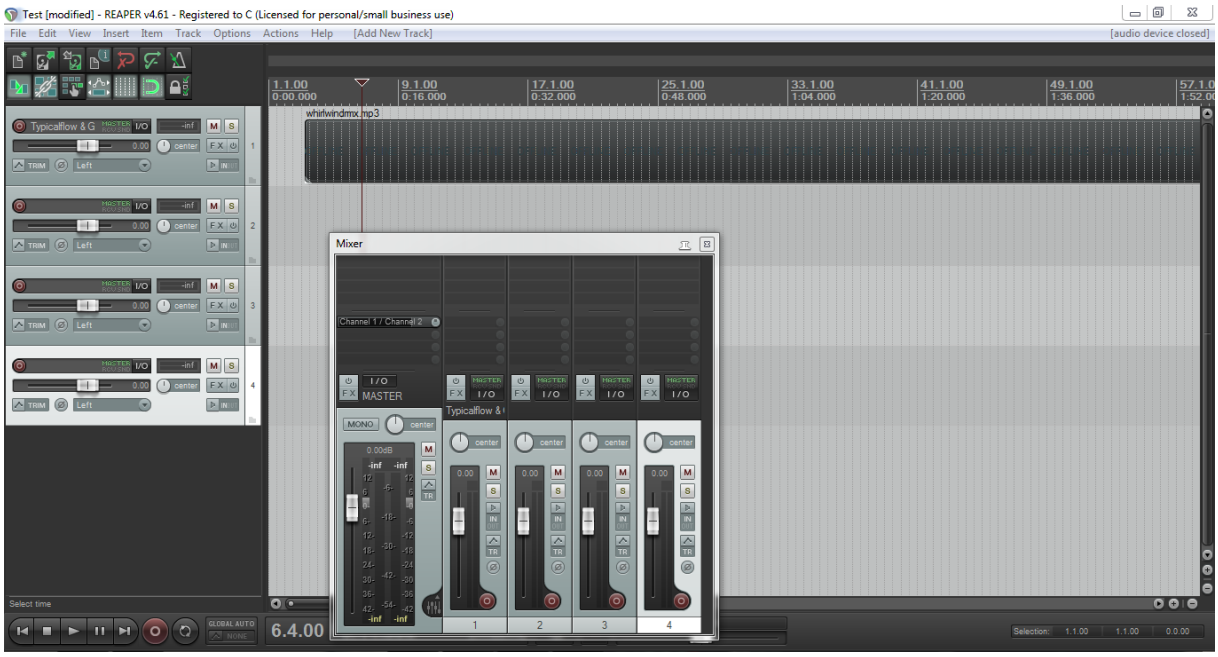
## 2.3 Programmeerimiskeele valik

Selleks, et vältida eelmainitud heli moondusi, on oluline kirjutada *pluginate* programmeerimisel võimalikult efektiivset koodi, et vältida ebaefektiivset mälu kasutust ja muid aspekte, mis võivad negatiivselt mõjutada *pluginate* töökiirust.

Seetõttu otsustasin kasutada C++ programmeerimiskeelt, mida peetakse tänu oma mälukasutusele üheks kiireimaks keeleks. Pidades kinni headest programmeerimise tavadest, ei tohiks *pluginad* oma jõudluses kannatada. C++ keeles programmeerimist peetakse üldiselt keerukamaks, kui näiteks Javas, millega on autor rohkem tuttav, kuid mis ei ole niivõrd soovitatav helitöötluseks [8].

# 3. *Plugin*a loomine SDK abil

Esimese *plugina* loomiseks kasutab autor ainult VST SDK-d ning „CodeBlocks“ nimelist IDE-d (*Integrated Development Environment*), et kirjutada C++ keeles madalsagedusfiltrit implementeeriv *plugin*. Kuna raamistike kasutamine võib arendaja eest ära peita osa tähtsast funktsionaalsusest, otsustas autor esialgu raamistikest loobuda, saamaks parem ülevaade VST *pluginate* ülesehitusest. Testimiseks kasutatakse „Reaper“ nimelist DAW-i.




**Joonis 5 - Reaper**

## 3.1 VST SDK valik

Selleks, et VST *plugina* kirjutamisega alustada, on vajalik hankida VST SDK (*Software Development Kit*), mille kõige uuem versioon (töö kirjutamise hetkel 3.6) on tasuta saadaval Steinbergi kodulehel. Kuna paljud DAW-id veel ei toeta VST III generatsiooni kasutamist, kaasa arvatud autori poolt muusika tegemiseks kasutatav „Reaper“, siis käesoleva töö käigus kasutatakse II generatsiooni ehk VST 2.4 kasutamist, mis ei ole enam Steinbergi poolt levitatav ega toetatav, kuid veebist siiski kättesaadav [9]. VST SDK koosneb C++ klasside kogumikust heli töötlemiseks ja töövoo kontrollimiseks ning lisaks on olemas hulk klasse GUI (*Graphical User Interface*) disainimiseks.

## 3.2 Heli mittetöötlev *plugin*

SDK-ga tuleb kaasa AGain nimelise näite *plugina* lähtekood, mida kompileerides saab heli amplituudi manipuleeriva VST *plugina*. Selle kasutajaliideses on implementeeritud üks liugur (*slider*), mille abil saab kasutaja vähendada väljundisse mineva heli valjust. Selle näite abil saab koostada esialgse versiooni filtrist, saates väljundisse sissetulev heli muutmata kujul.

### 3.2.1 Kõige olulisemad klassid

AGain koosneb kahest C++ keeles kirjutatud lähtefailist, esimene neist on päise- ning teine koodifail. Nad viitavad nii C++ standardteekidele, kui ka VST teekidele. Need klassid on olulised, kuna läbi nende defineeritakse *plugina* olemus ja töövoog. [10].

### 3.2.2 Päisefail „again.h“

Esimene neist on päisefail „again.h“, mis defineerib *plugina* objekti, et DAW teaks, kuidas sellega suhelda, samas teadmata, kuidas *plugin* enda siseselt ülesandeid täidab. Päisefail ütleb kompileerijale, et see lisaks programmi ka „AudioEffectX“ klassi, kuna vajalik on kasutada nimetataud klassis olevaid muutujaid ja funktsioone. Kindlasti ei tohiks seda klassi hakata ise muutma, kuna seeläbi on oht lõhkuda VST standard. VST SDK sisaldab endas dokumentatsiooni, kus on lühidalt kirjeldatud kõiki klasse, nende funktsioone ning parameetreid, kui on vaja uurida mõne klassi olemust. [10].

AGain klassi funktsioonide ja muutujate initsialiseerimine toimub läbi klassi konstruktori, mis kutsutakse välja DAW-i poolt. Konstruktoris initsialiseeritakse objekt „audiomaster“, mille tüübiks on audioMasterCallback ning see käitub parameetrina. See objekt saadetakse DAW poolt, läbi mille toimub programmide omavaheline suhtlus kui ka heli sämplite saatmine *pluginale*. Selleks, et vältida mälu lekkimist, on vajalik defineerida ka destruktor, kus enne *plugina* sulgemist saab vabastada mälu, mida see kasutas [10].

Edasi deklareeritakse avalikud (*public*) funktsioonid, mis on implementeeritud koodifailis ning on kirja pandud formaadis: siduvuse-tüüp*(binding type)* tagastus-andme-tüüp*(return-data-type)* funktsiooni-nimi (parameetrid). Kõik need funktsioonid on virtuaalse siduvuse tüübiga, et DAW saaks AGain klassi kaudu tuletada teisi klasse ning endiselt neid virtuaalseid funktsioone kasutada, kui enda omi. Need on funktsioonid nii helisämplite töötluseks, GUI kontrollimiseks, kui ka vajalike programmisiseste nimetuste defineerimiseks ehk kõik vajalik *plugina* töötamiseks.

Päisefailis on deklareeritud ka kaitstud (*protected*) muutujad, mida DAW saab lugeda, kuid mitte otse muuta. Kaitstud muutujate väärtuste uuendamiseks on võimalik kasutada GUI-t. [10]. Näites on kasutusel *float* tüüpi muutuja, millega saab reguleerida heli amplituudi, kuid hetkel võib selle muutuja eemaldada, et väljundisse saadetavad heli mitte muuta.

### 3.2.3 Koodifail „again.cpp“

AGain klassi koodifaili konstruktori parameetriteks on lisatud peale päisefailis deklareeritud „audiomaster“-i ka eeldefineeritud programmide ning kasutaja poolt muudetavate muutujate arvud [10]. Mõlemad arvud võib hetkel jätta nulliks, kuna eeldefineeritud programme *plugin* sisaldama ei hakka ning kasutaja poolt muudetavad parameetrid puuduvad.

Edasi defineeritakse konstruktoris sisend-ja väljundkanalite arvud, unikaalne programmi nimetus ning veel *plugina* käitumist mõjutavad funktsioonid, millest „canMono()“ ütleb, et *pluginat* saab kasutada ka mono signaali töötlemisel ja „canProcessReplacing()“, mis võimaldab *plugina send* efektina kasutamisel heli töötlemisel rakendada teistsugust algoritmi. [10].

Kuna hetkel ei ole heli töötlemiseks ühtegi objekti kasutusel, võib jätta destruktori tühjaks. Ühtlasi ei pea implementeerima ka ülejäänud funktsioone peale „process()“ funktsiooni, kuna kasutaja poolt muudetavad muutujad puuduvad.

„process()“ funktsioonis, mis võtab argumentideks sisend-ja väljundmassiivide *pointerid* ning *long* tüüpi muutuja „sampleFrames“, toimub *plugina* helitöötlus. Sisend-ja väljundmassiivid on helisämplite puhvrid, mille suurused määrab muutuja „sampleFrames“ ning mis aitab arvet pidada, millal helitöötlus lõpetada ehk millal on DAW-i poolt saadetud sämplid otsas. [10].

Antud funktsioonis luuakse kohalikud *pointer* muutujad, mida kasutatakse puhvrite läbikäimiseks vastavalt kanalite arvule ehk stereo heli puhul tehakse vasaku ja parema kanali jaoks eraldi muutujad [10]. Järgnevat *while* tsüklit täidetakse sämplite arv korda, kus väljundiks saab määrata sissetuleva signaali seda muutmata (vt Joonis 6).

```
110 //------------------------------------------------------------------------
111 void AGain::processReplacing (float** inputs, float** outputs, VstInt32 sampleFrames)
112 {
113     float* in1  =  inputs[0];
114     float* in2  =  inputs[1];
115     float* out1 = outputs[0];
116     float* out2 = outputs[1];
117
118     while (--sampleFrames >= 0)
119     {
120         (*out1++) = (*in1++);
121         (*out2++) = (*in2++);
122     }
123 }
```

**Joonis 6 - Sisendit mitte muutev funktsioon**

Seega on valminud *plugin*, mis ei töötle sissetulevat heli, vaid saadab selle DAW-ile tagasi muutmata kujul. Nüüd on võimalik hakata *pluginale* juurde lisama madalsagedusfiltri implementeerimiseks vaja minevaid osi.

## 3.3 Madalsagedusfiltri implementeerimine

Autor otsustas esimese *plugina* jaoks kasutada Nigel Redmoni poolt kirjutatud ühe poolusega madalsagedusfiltrit, mis oma omadustelt sobib pigem parameetrite jaoks, kui otse heli peal kasutamiseks, kuid tänu oma lihtsusele sobib väga hästi esimese *plugina* tegemiseks. Omaduste poolest on see filter üsna pehme ehk filtreerib suhteliselt vähesel määral helisagedusi välja. [11].

Implementeerimist saab alustada sama päisefaili muutmisest, mida eelnevalt kirjeldati. Selleks tuleb alla laadida filtri lähtekood järgnevalt veebiaadressilt: http://www.earlevel.com/main/2012/12/15/a-one-pole-filter/ [11].

## 3.4 *Plugina* päisefail

*Plugina* päisefaili tuleb lisada sisalduvuse (*include*) lause, alla laetud „OnePole.h“ faili kohta, et seda kasutada. Veel on vajalik lisada deklaratsioonid „OnePole“ filtri *pointer*-objektide kohta muutuja helisageduse määramiseks ning veel mõned abimuutujad. Seejärel saab asuda *plugina* koodifaili muutma.

## 3.5 *Plugina* koodifail

Koodifaili konstruktoris on vajalik initsialiseerida eelnevalt deklareeritud muutujad ja *pointerid*. *Pointerite* defineerimisel, tuleb neile argumendiks anda helisageduse väärtus normaliseeritud kujul ehk helisagedus hertsides tuleb läbi jagada DAW-i poolt kasutuses oleva sämplimissagedusega. Filtri objektide loomisel arvutatakse vastavalt määratud helisagedusele välja koefitsiendid, mille abil toimub hiljem helisämplite töötlemine. [11].

Selleks, et läbi kasutajaliidese oleks võimalik helisagedust muuta, tuleb vastavalt muudetava parameetri väärtusele kutsuda välja „OnePole.h“ failis olev „setFc()“ funktsioon, mis määrab läbi *pointeri* ligipääsetud objektile uued filtri koefitsiendid.

Selleks, et filtri objektid sissetulevat heli töötleksid on vaja neid kasutada „processReplacing“ funktsioonis vastavalt järgmisele joonisele.

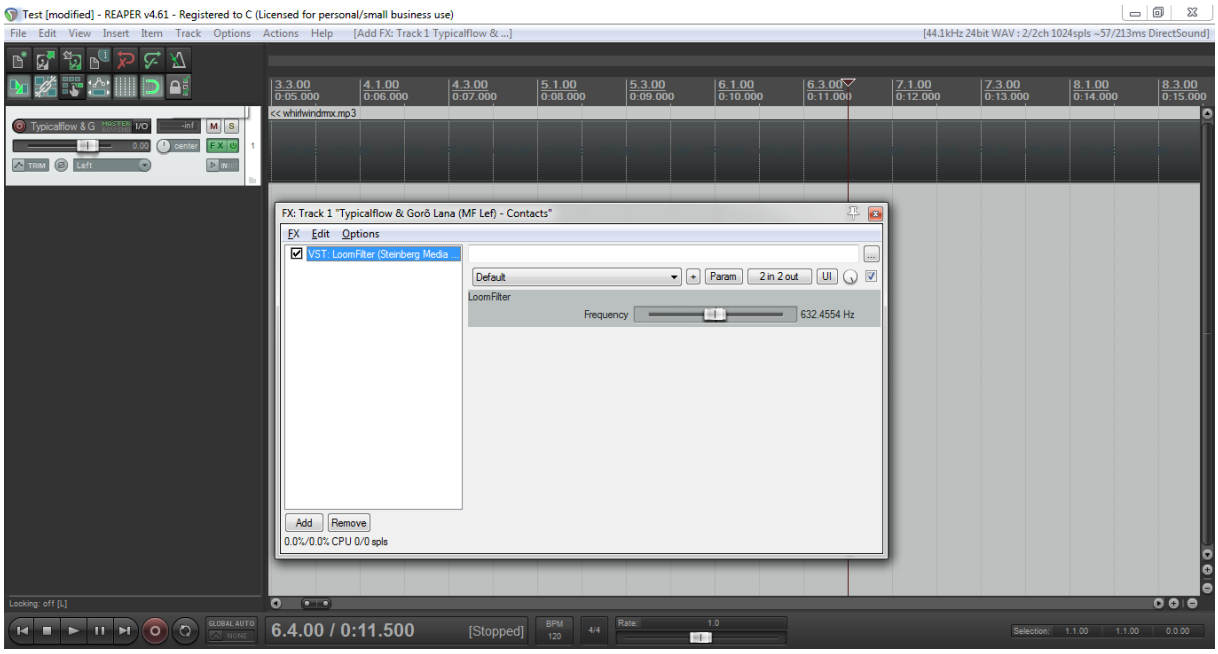
```
156  //-----------------------------------------------------------------------------------------
157  void LoomFilter::processReplacing (float** inputs, float** outputs, VstInt32 sampleFrames)
158  {
159      float* in1  =  inputs[0];
160      float* in2  =  inputs[1];
161      float* out1 = outputs[0];
162      float* out2 = outputs[1];
163
164      while (--sampleFrames >= 0)
165      {
166          (*out1++) = dcBlockerLpL->process((*in1++));
167          (*out2++) = dcBlockerLpR->process((*in2++));
168      }
169  }
170
```

**Joonis 7 - Madalsagedusfiltri "processReplacing()" funktsioon**

Jooniselt 7 on näha, kuidas kutsudes välja „process()“ funktsiooni läbi filtri *pointer*-objektide sisse tulevatele diskreetidele ja seejärel tagastatud andmed lisada väljundmassiivi, töötleb autori poolt valmistatud filter helisignaali.

Jooniselt 8 on vaadeldav, kuidas näeb välja äsja loodud madalsagedusfilter, mille jaoks ei ole loodud kohandatud GUI-t, mistõttu on *plugina* kasutajaliides implementeeritud vaikimisi DAW poolt. Jooniselt on näha liugurit, mille abil saab määrata filtri helisagedust hertsides.

Kuna antud *plugina* arenduse eesmärgiks oli aru saada eelkõige helitöötlusest, otsustas autor selle GUI-t edasi mitte arendama hakata.



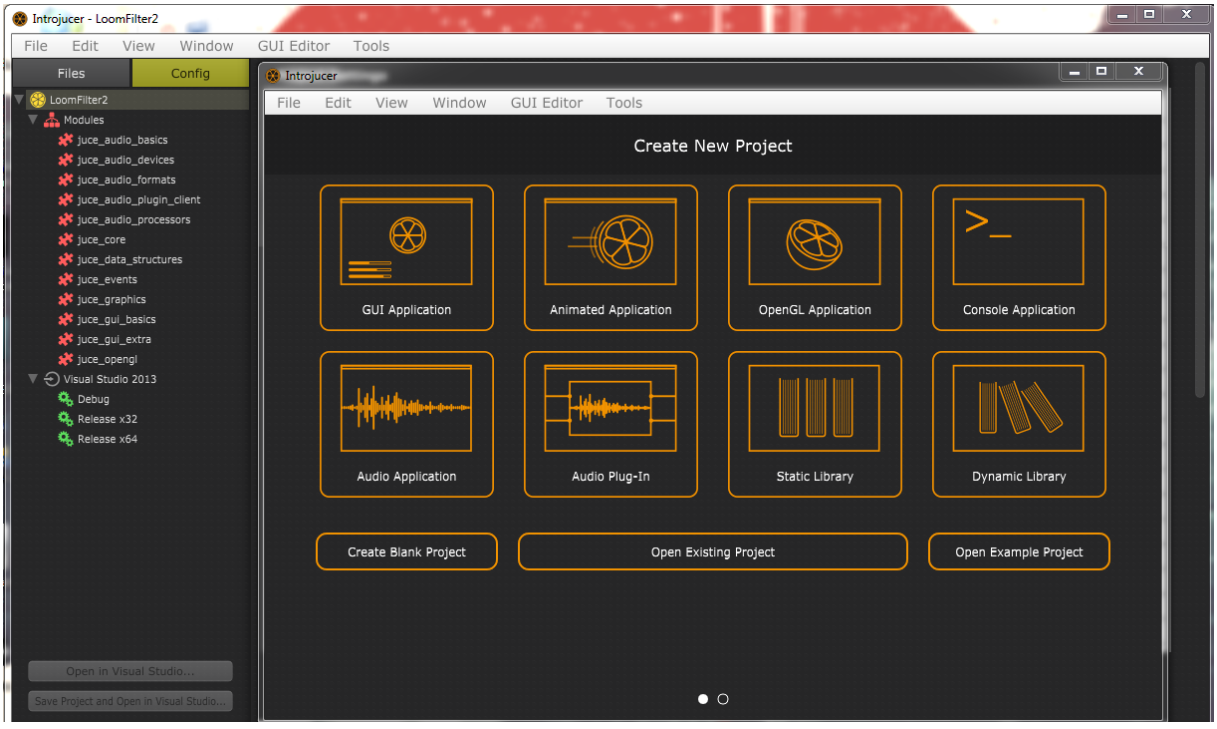

**Joonis 8 - Madalsagedusfilter**

# 4. *Plugin*a loomine JUCE raamistiku abil

Kui esimese *plugina* tegemisest selgus, et raamistikuta arendamine on aeganõudev ja tülikas protsess kogenematule VST arendajale siis loodab autor käesoleva raamistiku abil arendusprotsessi kiirendada. Selleks on valitud JUCE-nimeline raamistik, kus saab kasutada kaasatulevaid helitöötlusklasse ning GUI arendamiseks mõeldud abivahendeid. Ühtlasi pakub JUCE võimalust kompileerida VST-d korraga erinevate platvormide arhitektuuride jaoks, erinevates formaatides ning koostada projekte mitmete IDE-de jaoks. [12].

## 4.1 IntroJucer ja JUCE töövoog

JUCE saabub graafilist liidest sisaldava aplikatsiooniga IntroJucer, mida saab kasutada projektide loomiseks, koodi ja konfiguratsiooni haldamiseks, ühtlasi ka JUCE-i moodulite installimiseks/uuendamiseks/konfigureerimiseks [13].




**Joonis 9 – Introjucer**

IntroJucer-i abil saab esialgu paika panna *plugina* olulisemad sätted:

- Unikaalne nimetus;
- *Plugina* sisend-ja väljundkanalite arvu;

- Loodava *plugina* eri formaadid (VST 2, VST 3, AudioUnit jms);
- Platvormi arhitektuurid (32-bit või 64-bit), mille peal saaks *pluginat* jooksutada;
- Projekti failid erinevate IDE-de jaoks ja nende *build targetid* (kompileerimise sätete jaoks). [13]

Testimise lihtsustamise eesmärgil saab lisada ka *post-build* käsu, et kompileeritav *.dll* fail lisataks kasutaja VST *pluginate* kausta, mida DAW kasutab nende leidmiseks ja kasutamiseks. Vastasel juhul on vaja peale igat kompileerimist teha seda käsitsi, et DAW abil *pluginat* testida.

Kuna projekti jaoks loodi sätted IntroJucer-i kaudu, on vajalik edaspidi uued teegid ja failid samuti lisada läbi IntroJucer-i, et mitte olemasolevat konfiguratsiooni üle kirjutada ning mitte rikkuda *plugina* kirjutamise töövoogu. Sätteid võib muuta igal ajal peale nende esmast paika panemist, pidades kinni JUCE töövoost. [13].

Olulisemad klassid, mis „IntroJuceri“ poolt luuakse on PluginProcessor ja PluginEditor. PluginProcessor klassi abil saab juurde lisada vaja minevat DSP koodi ja PluginEditor on mõeldud GUI kontrollimiseks. Nendega ümberkäimisel tuleb olla ettevaatlik, sest koodi kirjutamine mitte selleks ettenähtud kohtadesse võib lõhkuda *plugina* ülesehituse ja JUCE töövoo. Selleks on automaatselt loodud klassides kommentaaridega kirjeldatud lubatud kohad kasutaja koodi lisamiseks. [13].

JUCE töövoo kirjeldamiseks on kolm tegevust:

1. Lähtekoodi failide lisamine ja projekti konfigureerimine (läbi „IntroJuceri“);
2. Kasutajaliidese disainimine (läbi „IntroJuceri“);
3. Kasutajaliidese sidumine ülejäänud lähtekoodiga (läbi IDE). [13].

## 4.2 Filtri algoritmi valik

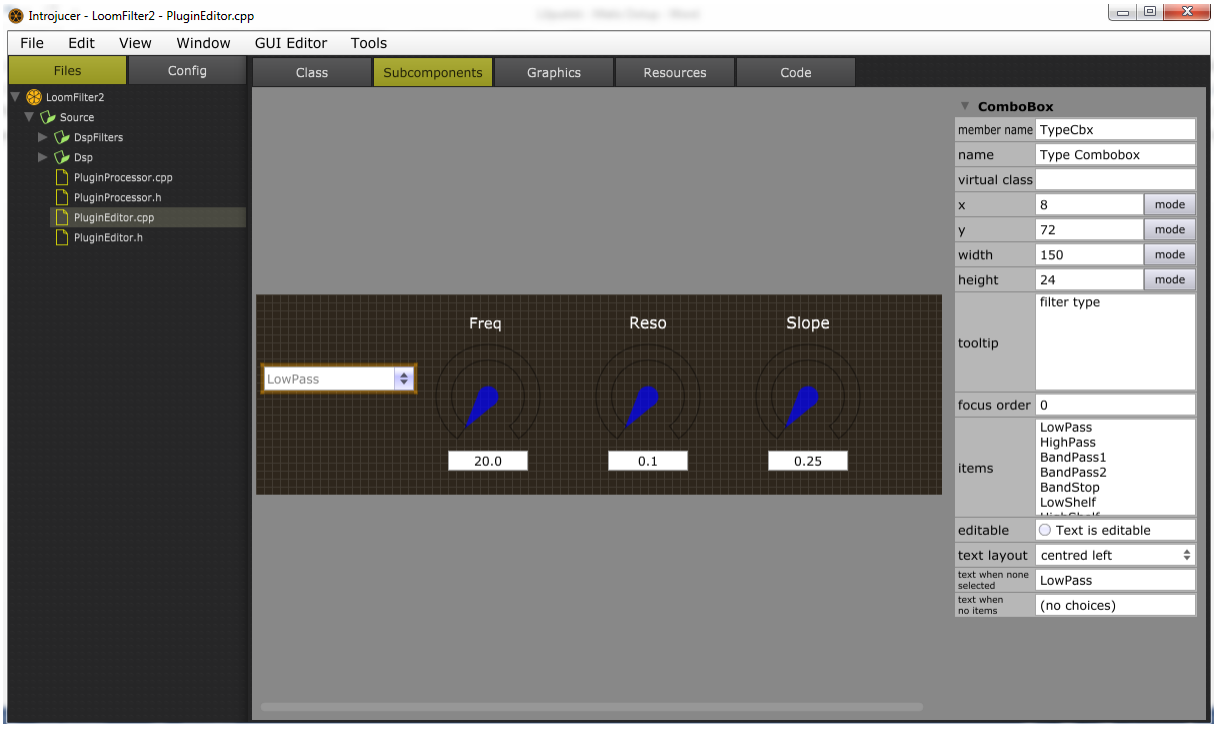
Autor on valinud filtri implementeerimiseks Vinnie Falco poolt MIT litsentsi alusel levitatava C++ teegi, mis sisaldab endas erinevaid DSP filtreid ja on kättesaadav järgnevalt veebiaadressilt: https://github.com/vinniefalco/DSPFilters/. [14].

Mainitud kollektsioonist sai välja valitud RBJ filter, mis on *biquad* tüüpi ehk 2. järku IIR filter. Antud filter on karakteristikute poolest piisavalt kõrge järguga, et seda iseseisvana kasutada ning piisavalt stabiilsete koefitsientidega. [14]. Ühtlasi leidis autor selle filtri musikaalsuse kõige paremini sobivat ülejäänud filtrite seast, katsetades kaasa tulnud rakendusega erinevaid filtrite tüüpe ja nende resonantsvahemikke.

## 4.3 *Plugina* implementeerimine

Esmalt on vajalik lisada alla laetud DSP filtrite failid läbi IntroJucer-i olemasolevasse projekti. Selleks, et filtrite lähtekoodi saaks kasutama hakata, on vajalik lisada veel sisalduvuse lause „Dsp.h“ faili kohta PluginProcessor klassi päisefaili, millest piisab ülejäänud koodile ligipääsuks [14].

Seejärel saab lisada IntroJucer-i kaudu vajalikud kasutajaliidese komponendid, mida on võimalik näha jooniselt 10. Lisatud parameetrite omadusi on samuti võimalik läbi kasutajaliidese muuta, määrates näiteks sagedusnupu väärtusvahemiku hertsides ja resonantsnupu väärtuste vahemiku. Selline metoodika lihtsustab oluliselt *plugina* arendust, kuna GUI taga olev kood genereeritakse IntroJucer-i poolt automaatselt ja arendaja ülesandeks jääb parameetrite sidumine PluginProcessor klassiga.




**Joonis 10 - GUI komponentide lisamine**

PluginProcessor klassi päisefailis on vaja deklareerida parameetrid, mille abil saab kontrollida kasutajaliideses tehtud muudatuste alusel plugina käitumist. Lisaks luuakse Filter tüüpi objekti kohta *pointer*, mis hiljem defineeritakse RBJ tüüpi filtriks, kui toimub plugina initsialiseerimine. [13].

„getParameter()“ ja „setParameter()“ funktsioonid on PluginProcessor klassis, et kontrollida läbi GUI tehtud muudatusi. „getParameter()“ võtab argumendiks muudetud parameetri indeksi, mis on deklareeritud varem PluginProcessor klassi päisefailis ja vastavalt sellele tagastab parameetri väärtuse, et näidata seda kasutajaliideses. „setParameter()“ võtab argumendiks sama indeksi ning teise argumendina juurde vastava parameetri uue väärtuse, mida spetsifitseeritakse läbi kasutaja liidese, näiteks filtri tüübi valimine *drop-down* menüüst. Seejärel vastavalt indeksi väärtusele, mis esindab *drop-down* menüüd, saab määrata uue filtri tüübi, mis on esindatud teise argumendina. [13].

Peale *plugina* laadimist DAW-i, käivitatakse enne heli taasesituse algust funktsioon „prepareToPlay()“, kus initsialiseeritakse filtri *pointer* suunama RBJ tüüpi filtrile ning ühtlasi antakse selle parameetritele vaikimisi väärtused. Kirjeldatud funktsioon on kujutatud alloleval joonisel 11.

```
//==============================================================================
void LoomFilter2AudioProcessor::prepareToPlay (double sampleRate, int samplesPerBlock)
{
    // Use this method as the place to do any pre-playback
    // initialisation that you need..
    f = new Dsp::SmoothedFilterDesign
        <Dsp::RBJ::Design::LowPass, 2>(1024);

    params[kSampleRate] = sampleRate; // sample rate
    params[kCutoff] = 10000; // cutoff frequency
    params[kResonance] = 1.25; // Q

    f->setParams(params);
}
```

**Joonis 11 – „prepareToPlay()“ funktsioon**

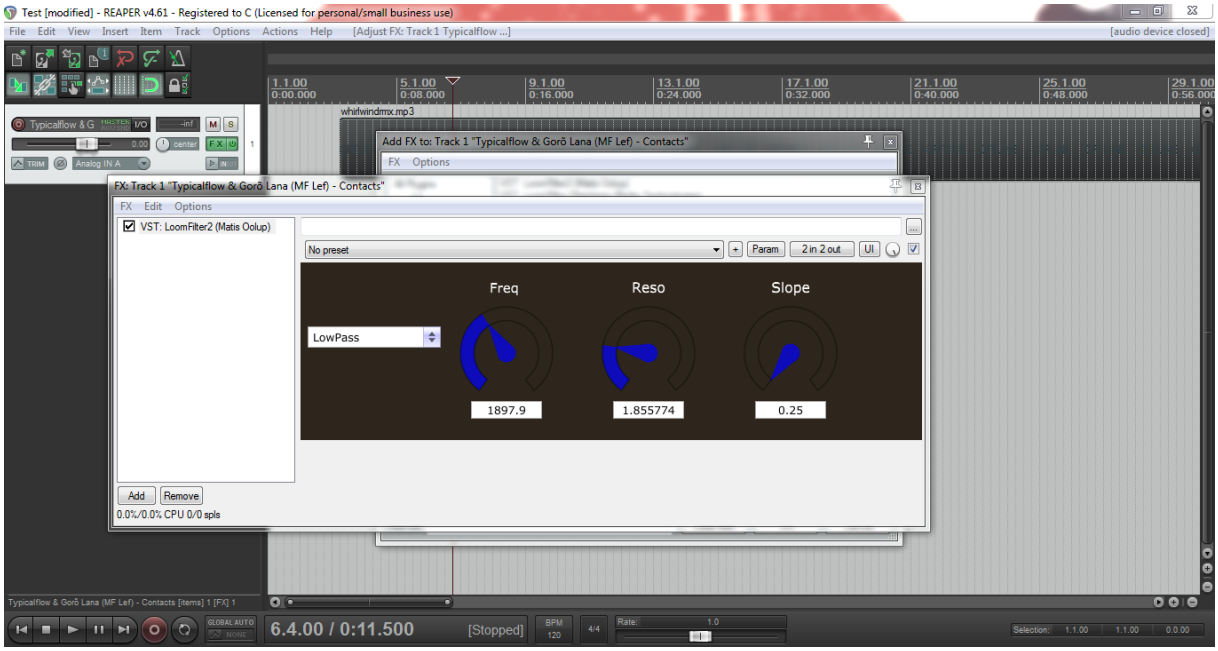
*Pointeri* väärtustamisel spetsifitseeritakse, et filter oleks „SmoothedFilterDesign“-iga, mis vähendab riski, et *plugina* parameetrite muutmisel ei tekiks heli rikkuvaid anomaaliaid nagu näiteks heli katkendlikkus. Edasi määratakse, et filter oleks RBJ disainiga ja tüübiks on madalsagedusfilter (*LowPass*). Argument „2“ määrab kanalite arvu ja „(1024)“ on diskreetide arv, mille alusel toimub parameetrite muutuste silumine (*smoothing*). [14].

Seejärel määratakse diskreetimissagedus, mille väärtus *sampleRate* saadetakse DAW poolt funktsiooni esimese argumendina. Teine filtri parameeter määrab helisageduse hertsides ning viimane parameeter on resonantsi kontrollimiseks. [14].

„processorBlock()“ funktsioon PluginProcessor klassis täidab sama eesmärki, nagu esimese *plugina* juures „processReplacing()“ funktsioon, ehk seda kasutatakse sissetulevate diskreetide töötlemiseks ja DAW-ile tagastamiseks. Selles funktsioonis saab sarnaselt esimesele *pluginale* läbi *pointeri* kutsuda välja filtri *process* meetod, mis töötleb sissetulevad diskreedid ja tagastab need muudetud kujul DAW-ile.

PluginEditor klassis tuleb spetsifitseerida *plugina* käitumine, kui läbi kasutajaliidese muudetakse mõnda parameetrit. Kui muudetakse näiteks resonantsi nupu abil selle väärtust siis PluginEditori koodifaili meetod „sliderValueChanged()“ võtab argumendiks *pointeri* muudetud kasutajaliidese objektile, pärib nupult äsja muudetud uue väärtuse ning seejärel kutsub välja PluginProcessor klassis oleva „setParameter()“ funktsiooni, mille läbi toimub filtri koefitsientide uuendamine vastavalt kasutaja määratud uuele väärtusele. [13].

Joonisel 12 on *plugin* kujutatud oma lõplikul kujul Reaperis, kus on näha kasutaja poolt muudetav rippmenüü, mille kaudu saab valida filtri tüübi ning kolm keeratavat nuppu sageduse, resonantsi ja sageduse kalde muutmiseks.




**Joonis 12 - Teine *plugin***

# 5. Järeldus

Esimese *plugina* arenduse käigus ilmnes, et *pluginate* testimine on raskendatud reaalajas töötamise tõttu, kuna ei ole võimalik lihtsalt koodi „läbi astuda“. Seetõttu oli vigade leidmine ja parandamine aeganõudev, kuna autoril puudusid varasemad kogemused VST programmeerimisega ning objektorienteeritusega C++-is. Seega tekkis programmeerimise käigus palju vigu. Esimese *plugina* arendus oli kindlasti esialgu aeganõudvam, kuna selle käigus tegi autor ühtlasi endale selgeks ka VST *pluginate* ülesehitust, millest ei olnud väga lihtne esialgu kinni haarata.

Kuna autoril on rohkem kogemusi varasemalt Java programmeerimiskeelega, mis erineb oluliselt C++-ist, eriti viimase mälukasutusvõimaluste tõttu, oli arenduse käigus keele selgeks õppimine samuti väljakutseks ning edasisel keele õppimisel oleks olnud võimalik *pluginat* veelgi optimiseerida.

JUCE raamistiku abil kirjutatava *plugina* testimine oli sarnaselt esimesele aeganõudev protsess ning seda eriti funktsionaalsuse suurenemisest tingituna. Teist *pluginat* tehes oli esimese *plugina* arendusest üldpilt selgeks saanud ning neid omavahel võrreldes on nad raamistikust hoolimata väga sarnase ülesehitusega. Kindlasti oli JUCE raamistiku abil mugavam arendada, kuna IntroJuceri abil oli võimalik vajalikud klassid ja *plugina* konfiguratsioon küllaltki lihtsalt üles seada. Raamistik võimaldas keskenduda rohkem olulisele ehk DSP algoritmide implementeerimisele ja nende sidumisele kasutajaliidesega.

# Kokkuvõte

Käesolevat tööd ajendas kirjutama autori huvi digitaalse muusika loomise vastu. Täpsemalt soovis autor luua ise VST *plugina*, millel oleks suurem resonantsi väärtuste vahemik, kui siiani autori poolt kasutatud filtritel.

Töö käigus loodud esimese *plugina* arendamisel suutis autor läbi digitaalse signaalitöötluse põhimõtete selgeks teha VST *pluginate* ülesehituse, mis aitas kaasa ka JUCE raamistikul tehtud *plugina* arendamisele. Digitaalne signaalitöötlus aitas mõista, et helitöötlus toimib lühidalt öeldes analoogsignaalilt saadud andmepunktide manipuleerimisel ning mis osadest tüüpiline signaali töötlev süsteem koosneb. Lisaks õppis autor kasutama objektorienteeritud C++ programmeerimiskeelt, ilma milleta poleks olnud võimalik *pluginaid* valmis saada. Töö lõpuks valmis VST *plugin*, millel puudub lisafunktsionaalsus peale heli filtreerimise ning mille resonantsi väärtuste vahemik on suurem kui siiani autori poolt kasutatud filtritel.

Järeldustes tõi autor välja VST *pluginate* arendamise keerulisemad osad, milleks on nende testimine, C++ kasutamise keerulisus ning JUCE raamistiku kasutamise positiivsed küljed, mis muudavad arenduse protsessi lihtsamaks.

Eelnimetatud teemasid uurides ja arendades seeläbi VST *pluginaid*, jõudis autor oma püstitatud probleemide lahenduseni. Käesoleva töö võimalikeks edasiarendusteks oleks kasutajaliidese täiendamine ja *plugina* arendus OS X platvormi jaoks, et seda laialdasemalt levitada.

# Summary

One of the causes for writing this thesis is the author's interest in creating digital music. More precisely, the author wanted to create a VST plug-in, which would have a bigger range in resonance settings than he has encountered so far.

During the creation of the first plug-in, the author learned the underlying principles in DSP, which helped to understand the construction of VST plug-ins and to help with writing a plug-in using the JUCE framework. Research about digital signal processing helped to understand that DSP is based on manipulating samples which are acquired from continuous analog signals. It also made clear of what parts a typical digital processing system consists of. By the end of this thesis a plug-in was made, which satisfies the set problem of having a filter which does not have any extra functionality and has a large range in value for resonance settings.

In the conclusion, the author brought out points about the complexity of developing VST plug-ins which are: complicated testing, complexity of the C++ programming language and also the positive effects of using the JUCE framework, which made the development process easier.

During the research of aforementioned goals and by developing VST plug-ins, the author had reached his goals. To further develope the thesis, it is possible to improve the plug-in written in JUCE framework by improving the user interface and to also make it available for OS X platform.

# Kasutatud kirjandus


[1] „Digital Audio Workstation, “ Wikipedia. [Internetiallikas].
http://en.wikipedia.org/wiki/Digital_audio_workstation. [Kasutatud 2015 3 24]

[2] „JUCE, “ JUCE. [Internetiallikas].
http://www.juce.com. [Kasutatud 2015 4 24]

[3] „Definition of: MIDI, “ PCMag. [Internetiallikas].
http://www.pcmag.com/encyclopedia/term/47014/midi. [Kasutatud 2015 5 20]

[4] „Definition of: plug-in, “ PCMag. [Internetiallikas].
http://www.pcmag.com/encyclopedia/term/49395/plug-in. [Kasutatud 2015 5 20]

[5] „Virtual Studio Technology, “ Wikipedia. [Internetiallikas].
http://en.wikipedia.org/wiki/Virtual_Studio_Technology. [Kasutatud 2015 3 24]

[6] Pirkle, W. Designing Audio Effect Plug-Ins in C++ : With Digital Audio Signal Processing Theory. Suurbritannia, Abingdon : Focal Press, 2013.

[7] Smith, S. W. PhD. The Scientist and Engineer Guide to Digital Signal Processing. Second Edition. USA, California : California Technical Publishing, 2011. [Internetiallikas]
http://www.dspguide.com/ch3/1.htm. [Kasutatud 2015 5 2]

[8] „What are the most important parts of C++ for coding plug-ins?, “ KVR. [Internetiallikas].
http://www.kvraudio.com/forum/viewtopic.php?f=33&t=52342. [Kasutatud 4 11]

[9] VST 2.4 SDK [Internetiallikas].
http://www.dith.it/listing/vst_stuff/. [Kasutatud 3 24]

[10] „Steinberg Example VST Plugin Walkthrough, “ Euphoria Audio. [Internetiallikas].
http://www.euphoriaaudio.com/tutorials/vstexample/vstexample1.php.
[Kasutatud 4 13].

[11] „A one-pole filter, “ EarLevel Engineering. [Internetiallikas].

http://www.earlevel.com/main/2012/12/15/a-one-pole-filter/. [Kasutatud 4 20].

[12] „The Introjucer, “ JUCE. [Internetiallikas].
http://www.juce.com/learn/introjucer. [Kasutatud 2015 4 24]

[13] „JUCE for VST Plugin Development, “ Redwood Audio. [Internetiallikas].
http://www.redwoodaudio.net/Tutorials/juce_for_vst_development__intro.html.
[Kasutatud 5 2].

[14] „A Collection of Useful C++ Classes for Digital Signal Processing, “ Vinnie Falco DSP Filters. [Internetiallikas]
https://github.com/vinniefalco/DSPFilters/. [Kasutatud 5 16].