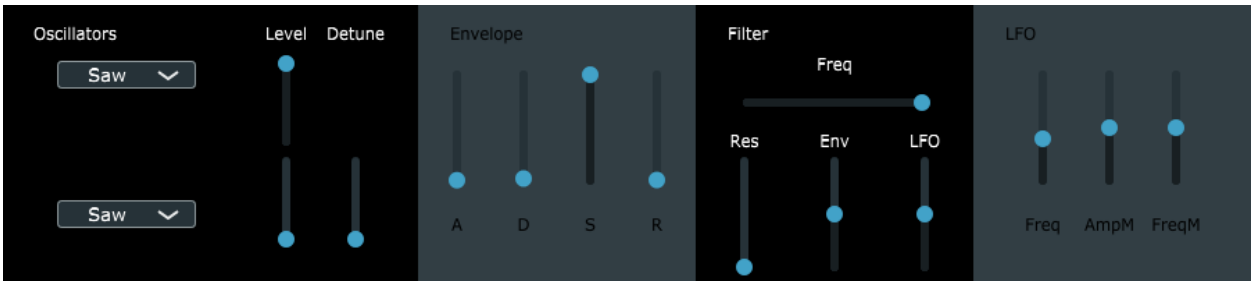
BaSynth Dokumentáció

Feladat leírása

A feladat egy analóg polifonikus szintetizátor megvalósítása volt, mely rendelkezik 2 oszcillátorral és egy negyedfokú rezonáns aluláteresztő szűrővel, amiknek a paramétereit (amplitúdó, vágási frekvencia, hangmagasság) egy adsr görbével és egy szinusz lfo-val modulálni lehet.

A programot Juce v6.0.4 környezetben írtam, Windows 10 operációs rendszer alatt, Visual Studio 16.7.7 fejlesztői környezetben. Emellett Juce beépített fordítóját használtam.

Felhasználói Felület



Az ábrán látható, hogy a szoftver 4 nagyobb blokkrészletből áll.

- Az első blokkban (Oscillators) a két oszcillátor típusát választhatjuk ki (4 féle),
    - Szinuszjel
    - Fűrészfogjel
    - Négyszögjel
    - Háromszögjel
    - Ezután a hangerősségüket állíthatjuk külön a két oszcillátornak.
    - És végül a második oszcillátort el lehet hangolni maximum 1 kisszekunddal (semitone)
- A második blokkban (Envelope) az adsr görbét lehet változtatni, ami alapbeállításoknál az amplitúdót változtatja.
- A harmadik blokk a szűrőé (Filter)
    - A 'Freq' paraméterrel a szűrő vágási frekvenciáját lehet változtatni. (10-19952 Hz)

- A 'Res' paraméterrel a rezonancia mennyiségét (0-1) a vágási frekvencián lehet változtatni.
- Az 'Env' paraméterrel (-1 és 1 között) azt lehet állítani, hogy hány százalékban modulálja a szűrő vágási frekvenciáját az adsr görbe.
- Az 'LFO' paraméterrel (szintén -1 és 1 között) azt lehet állítani, hogy hány százalékban modulálja a szűrő vágási frekvenciáját az alacsony frekvenciás oszcillátor (LFO).

- A negyedik blokkban (LFO) az alacsonyfrekvenciás oszcillátor paramétereit lehet állítani.
  - A 'Freq' paraméterrel az lfo frekvenciáját lehet változtatni, ami a DAW tempójával szinkronban van (BPM/2-től BPM*8-ig, a kettő között lineárisan változtatható).
  - Az 'AmpM' paraméterrel (-1 és 1 között) azt lehet állítani, hogy hány százalékban modulálja az jel amplitúdóját az lfo.
  - A 'FreqM' paraméterrel (-1 és 1 között) azt lehet állítani, hogy hány százalékban modulálja (1 kisszekundnyira maximum) az jel frekvenciáját az lfo.

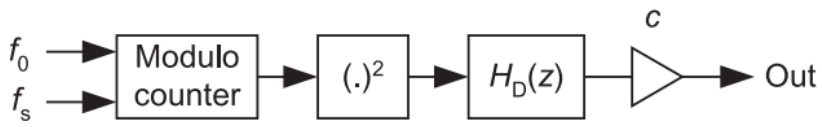
Oszcillátorok megvalósítása (Oscillators.h Osc osztály)

Szinuszjel megvalósítása:

Ezt nem szeretném részletezni, szerintem a kódrészlet magáért beszél.

```
double sine(double frequency)
	{
		output = sin(phase * (TWOPI));
		if (phase >= 1.0) phase -= 1.0;
		phase += (1. / (rate / (frequency)));
		return(output);
	};
```

Fűrészfogjel megvalósítása:



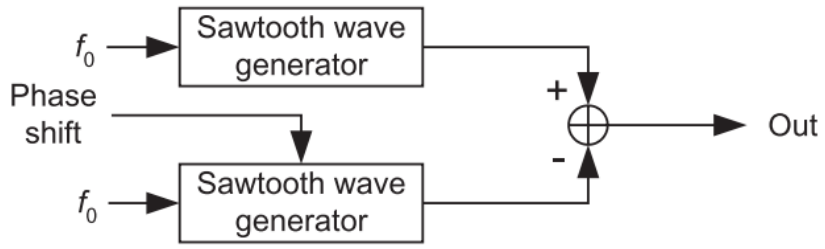
Az átlapolódás mértékét jelentősen lehet csökkenteni ezzel a módszerrel, miszerint egy ideális fűrészfog jelet négyzetre emelünk, így egy parabola jelet kapunk és ezt utána differenciáljuk és felszorozzuk.

A 'c' felszorzás értéke `fs/(f0*4)` ahol fs a mintavételi frekvencia és f0 az aktuális hang frekvenciája.

```
double saw(double frequency)
	{
		phase += (frequency / rate);
		if (phase >= 1.0)
			phase -= 1;

		bphase = 2 * phase - 1;
		sq = bphase*bphase;
		output = sq - z1;
		z1 = sq;
		return(output * rate/(frequency*4));
	};
```

négyszögjel megvalósítása:



Itt alapvetően, két fűrész jelet (a fent megvalósítottakhoz hasonlóan) vonunk ki egymásból egyiket a fázisával eltolva.

Háromszög jel megvalósítása:

Ezt a jelet véletlenül hoztam létre a négyszög jelből, az alábbi kódrészletben a kommentnél jelöltem, mit rontottam. Alapvetően a második fűrészfog jelet differenciáláskor véletlenül az első jel múltbeli értékét vontam ki belőle. Ehhez viszont a végén le kellett osztanom a jelet 20-szal mert különben túlveszérlődött volna.

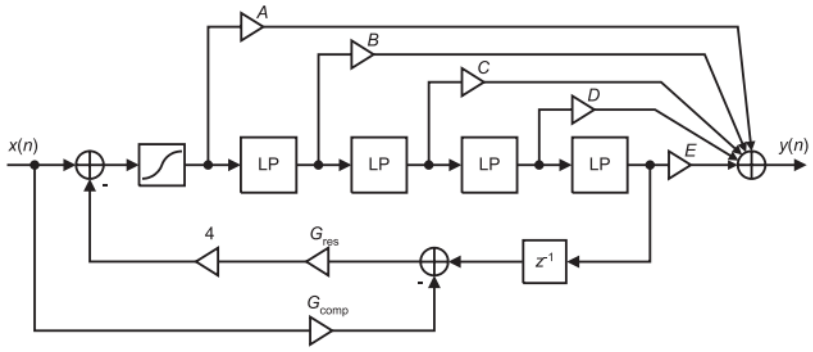
```
double triangle(double frequency)
	{
		phase += (frequency / rate);
		if (phase >= 1.0)
			phase -= 1;
		bphase = 2 * phase - 1;
		sq = bphase * bphase;
		output = sq - z1;
		z1 = sq;
		out1 = output * rate / (frequency * 8);

		phase2 = phase + 0.5;
		if (phase2 >= 1)
```

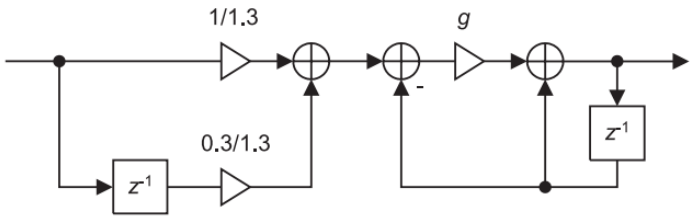
```
            phase2 -= 1;
        bphase = 2 * phase2 - 1;
        sq = bphase * bphase;
        output = sq - z1; // z1<-->z2
        z1 = sq;
        out2 = output * rate / (frequency * 8);

        return((out1 - out2)/20); // 20-szal való osztás
};
```

Szűrő megvalósítása (Filters.h Filters osztály)



A szűrőt 4 db elsőfokú aluláteresztő szűrővel valósítottam meg. Minden egyes LP blokk az alább látható rendszert valósítja meg.



Itt a $g = 0.9892wc - 0.4342wc^2 + 0.1381wc^3 - 0.0202wc^4$ ezt amiatt kell beiktatnunk mert különben magas frekvenciákon el fog térni a valós vágási frekvencia az elvitől.

Ahol $wc = 2\pi fc/fs$ (a vágási frekvencia radiánban a mintavételi frekvenciához képest)

Az A, B, C, D, E erősítések értékének változtatásával lehet a szűrő típusát változtatni. Ezt én nem valósítottam meg így úgy tekintettem az elemeket, hogy A=0, B=0, C=0, D=0 és E=1.

A végleges szűrő visszacsatolása a rezonancia miatt fontos, itt a

$$Gres = Cres\ (1.0029 + 0.0526wc - 0.0926wc^2 + 0.0218wc^3)$$

Ahol Cres E[0;1], a rezonancia mennyisége. Ezzel a függvénnyel 17 kHz alatt a torzítás kevesebb mint 1%-os.

A Gcomp amiatt kell, hogy az áteresztő tartomány a rezonancia változtatásával ne változzon, ezt én 0.5-re választottam mert igy egy kellemes hangerőt érhetünk el.

Modulálók megvalósítása (Modulators.h Mod osztály)

ADSR görbe: `double Mod::adsr(double input);`

Állapotokkal valósítottam meg. Az osztályban van egy változó (`int trigger`) ami igazából lehetett volna egy `bool` típusú is, ugyanis 0 értékűre állítottam, ha nincs lenyomva egy billennyű és 1-re ha igen. Igy alapvetően ameddig `trigger=1` addig elindult attack állapotból decay-en keresztül a sustain-ig ami után addig ott maradt amíg `trigger=0`-ba nem vált. Viszont akármikor, ha a trigger=0 lesz, azonnal a release állapotba kerül ahol az amplitúdó csökkentése mellett az állapotváltozókat vissza állítottam 0-ba.

Az időket az adsr értékeknek külön be lehet állítani ms-ban a megfelelő függvényekben.

LFO: `double Mod::sine(double BPM, double rate);`

Ez tulajdonképpen (mivel a nevében is benne van) ugyanaz, mint egy szinusz oszcillátor csak a fázis sokkal alacsonyabban 'forog'.

Ezt változtattam csak, az oszcillátorokhoz képest:

```
phase += (1. / (sampleRate / (rate * BPM / 480)));
```

Itt a rate a csúszka alapján beállított érték és a BPM pedig a külső tempó.

A 480 úgy jött ki, hogy a BPM et 60-nal le kell osztani így BPS-t kapunk, ami tulajdonképpen a Hz és ezt még le kell osztani 8-cal hogy kellően lassú oszcilláció legyen a minimum frekvencia érték. A slider (rate) értéke 1 és 16 közötti értékeken mozoghat.

Modulációk megvalósítása (SynthVoice.h SynthVoice osztály)

Mivel százalékosan lehet a modulálókat a paramétereken használni ezért normálni kellet ezeket az értékeket. Ezt a szűrő vágási frekvenciájának az adsr görbével való modulációján mutatom be.

Úgy gondolkodtam, hogy ha +100%-osan szeretnénk modulálni a vágási frekvenciát akkor azt értelmezzük úgy, mintha a Freq slider aktuális értéke és a maximális értéke közötti teljes utat bejárja a moduláció. Ez természetesen, ha maximum értéken van a Freq slider akkor nem modulál +1 irányba semmit (viszont, ha -1-re visszük a moduláció mennyiségét akkor fordítva a teljes intervallumon mozogni fog a vágási frekvencia valódi értéke).

```
if (env > 0)
	enva = double(*env) * (4.3 - filterfreq);
else
	enva = double(*env) * (filterfreq - 1);
```

Ezt a fentebb látható módon oldottam meg itt az 'env' érték az a slider lebegőpontos értéke és ezt normáltam. Ezek után az 'enva' változót használtam a program többi részén.

Látható még ez a 4.3 és 1-es érték a kódrészletben. Ez amiatt van mert a vágási frekvencia slider lineárisan változik, viszont ahogy halljuk a hangmagasságot az logaritmikus így azt csináltam, hogy a 'filterFreq' értékét 1 és 4.3 közé választottam és ha ezt átalakítjuk frekvenciákra akkor $10^1 = 10$ és $10^{4.3} = 19952$ Hz értékeket kapunk. Így a csúszka változtatásával a vágásifrekvenciát is lineárisan változtatja.

A többi értékek modulációjával is hasonlóan gondolkodtam, ezeket nem szeretném leírni, mert nagyon hosszú. A szűrő beállítását az alábbi kódrészletben mutatom meg:

```
fltr->setFreqAndQ(juce::Decibels::decibelsToGain((filterfreq + env1.adsr(enva) + filterLfo
* lfo1.sine(BPM, freqSet)) * 20), filterres);
```

A Juce környezet decibelsToGain() függvényét használtam a frekvenciák átalakításához, emiatt kellett a jelet még 20-szal beszoroznom mert ez alapvetően amplitúdóra van ez kitalálva.

Elhangolás

Ezt egy egyszerű függvénnyel oldottam meg, a cél az volt, hogy 1 kisszekunddal változzon a második oszcillátornak a frekvenciája.

```
double setDetune()
	{
		return frequency + (0.059463094*frequency) * detune + (0.059463094 *
frequency) * freqMod * lfo1.sine(BPM, freqSet); // 2^(1/12)-1 :: 1 semitone detune
	}
```

Ahogy a kommentezésben is látható 1 kisszekund hang frekvencia növekedése 2^(1/12)=1.059… (ennyivel kell beszorozni az alap frekvenciát hogy egy semitone-nal nagyobb hangot halljunk)

1-et azért vontam le belőle mert a frekvencia különbségük kellett nekem.

```
frequency + (0.059463094*frequency) * detune
```

igazából ez az alap koncepció, ezután az lfo-t is hozzáadom.

Észrevettem, hogy ezzel, ha elmozdítjuk a detune slidert és utána vissza 0-ba akkor már nem lesznek fázisban érthető okokból az oszcillátorok. Ezt a problémát nem oldottam meg. Bár úgy meg lehetett volna, ha kívülről is hozzátudok férni az oszcillátorok aktuális fázisához.

Grafikus felület

Minden olyan osztály és file, ami nevében szerepel a 'Menu' idetartozik. Emellett az 'AmpandFreq' (ami az első 3 sliderért felelős) és a 'PluginEditor' (ez fogja össze a különböző vizuális blokkokat) is.

A 'PluginProcessor' alapvetően felelős a grafikus felület és külső adatok (befele: midi, bpm és kifele: samplebufferek) és a valós számolások közötti kapcsolatért.

Megjegyzések

A mintavételi frekvencia 44.1 kHz mindenhol, és sajnos nem írtam teljesen modulárissá, tehát elhangolódik és más problémák is előjöhetnek, ha más mintavételi frekvenciával használjuk a programot.

A négyszögjel kitöltési tényezőjét sajnos nem lehet valós időben változtatni, csak a kódban lehet átírni.

Próbáltam minden konstanshoz egy viszonylag pontos közelítő double számot adni, ugyanis nem akartam, hogy ezzel valós időben számolnia kelljen még a programnak (pl: 2^(1/12) helyett 1.059463094-gyel számoltam) ezzel lecsökkentve a számolási igényét a programnak.

Polifóniát a Juce környezet beépített függvényeivel valósítottam meg az alább látható kódrészlettel:

```
mySynth.clearVoices();
for (int i = 0; i < 5; i++)
{
    mySynth.addVoice(new SynthVoice());
}
mySynth.clearSounds();
mySynth.addSound(new SynthSound());
```

Ahol, a mySynth egy Juce által definiált Synthesiser típusú osztály objektuma (juce::`Synthesiser mySynth;`). Emellett látható, hogy 5 hang generálására képes egyszerre a program. Ezt a kódrészletet az AudioProcessor konstruktorába írtam a megfelelő működéshez. (PluginProcessor.cpp)