Brukergrensesnitt for sanntidsbehandling av lyd
Mapping av sensordata til DSP-funksjoner

Arve Voldsund

Masteroppgave

Institutt for musikkvitenskap
Universitetet i Oslo

Vår 2007

Forord

Min fascinasjon for den elektroakustiske musikken startet med Arne Nordheim og platen *Electric*, som originalt ble gitt ut i 1974, og på nytt i 1998 av *Rune Grammofon*. Stykkene som er samlet i *Electric* tilbyr et lydlandskap som for meg fungerer som en ladestasjon for å fortsette letingen etter nye lyder, og det er et intenst ønske om å kunne spille denne typen lyd på konsert som danner grunnlaget for min masteroppgave.

Heldigvis er det mange rundt om i verden som arbeider med utvikling av bedre verktøy for kontroll av datamaskiner, og innfallsvinkelen til problemstillingen har mange fasetter. I denne oppgaven har jeg konsentrert meg om å finne ut noe om hvordan datamaskiner kan styres uten at man må gi dem noe særlig oppmerksomhet under fremføringen av musikken. Mange av oss er utøvere på et instrument som krever fingre fra to hender mer eller mindre hele tiden, og da er det ugunstig å være tvunget til å ta hendene bort fra instrumentet for å forberede datamaskinen på det den skal gjøre. Jeg tror at det er mulig å lage et enkelt system som i utgangspunktet ikke krever at man bygger om instrumentet, eller som tvinger utøveren til å endre en gjennomarbeidet spillestil. Interaksjonen med hovedinstrumentet bør være uanfektet så langt det lar seg gjøre, med mindre man ønsker å bruke ny teknologi som et verktøy i utviklingen av en spillestil som integrerer sensorer og datamaskiner til et felles instrument.

Den følgende teksten gir en generell gjennomgang av sensorer som musikalske instrumenter og teknologien som er nødvendig for å kunne bruke sensorer i en praktisk applikasjon. I tillegg presenterer jeg min nåværende løsning på problemstillingen, og håper med dette å kunne bidra til *NIME (New Interfaces for Musical Expression)*[1] og musikkteknologimiljøet i Norge.

Jeg ønsker å takke hovedveileder Rolf Inge Godøy for velvilje, tilbakemeldinger og for alle forslag til forbedringer som har kommet gjennom hele arbeidet. I tillegg setter jeg stor pris på å ha fått muligheten til å delta på mange interessante samlinger i regi av *Musical Gestures* prosjektet ved Institutt for Musikkvitenskap, UiO. Videre ønsker jeg å takke biveileder Alexander Refsum Jensenius som har vært en viktig samtalepartner i forhold til de tekniske aspektene ved oppgaven, og for å ha introdusert meg for programmeringens lunefulle verden.

1 *NIME* refererer i denne oppgaven til fagfeltet som organisasjonen *NIME* representerer. Organisasjonen *NIME* arrangerer en årlig konferanse hvor utviklere av nye grensesnitt for musikalske uttrykk kan presentere sine arbeider, og det er i denne sammenheng jeg ønsker å bidra.

I tillegg ønsker jeg å rette en takk til Arnfinn Lunde og Herman Lia ved Høgskolen i Vestfold avdeling realfag og ingeniør. Arnfinn for å lært meg å programmere i *C*, og hvordan mikro-kontrollere fungerer, mens Herman har vist meg rundt i fysikkens verden via samtaler om musikk, matematikk og elektronikk.

Innholdsfortegnelse

1. Introduksjon

Dette kapittelet gir en introduksjon til musikkteknologien som fagfelt, og en beskrivelse av de ulike delene av et datamaskinbasert musikalsk verktøy som vil bli diskutert i denne oppgaven.

1.1 Utgangspunkt for arbeidet

I utgangspunktet var denne oppgaven tenkt å være en beskrivelse av en praktisk tilnærming til programvare for sanntidsmanipulering av *timbre* i betydningen musikkinstrumenters klang-karakteristikk. *Timbre* er et mangfoldig emne, og i denne oppgaven omhandler manipulering av *timbre* teknikker som endrer lydens karakteristikk med tanke på frekvensspekterets balanse mellom høye og lave frekvenser: *brightness,* samt endringsmønster for en lyds overtoner og lydens attakk (Bregman, 1990, s.481). Jeg mener at forholdet mellom disse tre er et tilstrek-kelig utgangspunkt for å finne fornuftige metoder for manipulering av *timbre* ettersom det i denne oppgaven handler om lyd fra komplekse lydkilder, og at oppgaven ikke har til hensikt å gjøre rede for hva vi vet og ikke vet om *timbre* som fenomen. K.W Berger sin *confusion matrix* (Rossing, Moore og Wheeler, 2002, s.140) er med på å underbygge dette: Resultatene som Berger registrerte i *Confusion matrix* er basert på avspilling av instrumentlyder som har fått attakk og slutt fjernet (første og siste 0,5 s av instrumentlydens levetid), og viser hvordan 30 lyttere bommer når lydene de hører skal knyttes til instrumentets navn. Attakket har også stor innvirkning på en lyds overtonespekter og dermed forholdet mellom lave og lyse frekvenser i lydens frekvensspekter. Dette kan nesten alle erfare ved å bruke et plekter, og slå an en streng på en gitar gjentatte ganger mens man flytter anslagspunktet langs strengen. Kvifte (1988, s93) kaller dette analog tonefarge, og eksperimentet viser på en enkel måte endring i *timbre* og forholdet mellom attakk og overtonespekter/brightness.

I forordet nevnte jeg Arne Nordheims elektroakustiske musikk som en inspirasjonskilde, men generelt sett er det alle opplevde lydbilder som er fremmede i forhold til dem jeg er vant til fra tradisjonelle instrumenter som har bygget opp et ønske om å kunne kombinere det beste fra to tradisjoner: den akustiske og den elektriske. Med dette som utgangspunkt fant jeg det naturlig å konsentrere oppgaven om hva *timbre* er (se feks Godøy, 1999), hva som kreves av programvare for å manipulere de bestanddelene som finnes i *timbre* som fenomen, og til slutt omsette erfaringene fra kartleggingen til et praktisk verktøy for sanntid manipulering av

samplet[2] lyd. Etter hvert som et program kom til, oppdaget jeg at det mest interessante er hvordan man får kontroll over programvaren samtidig som man skal gjøre andre oppgaver. Derfor ble emnelisten utvidet til å inkludere maskinvare og programvare for menneske-maskin interaksjon: *"Human-computer interaction is a discipline concerned with the design, evaluation and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them"* (ACM SIGCHI[3]).

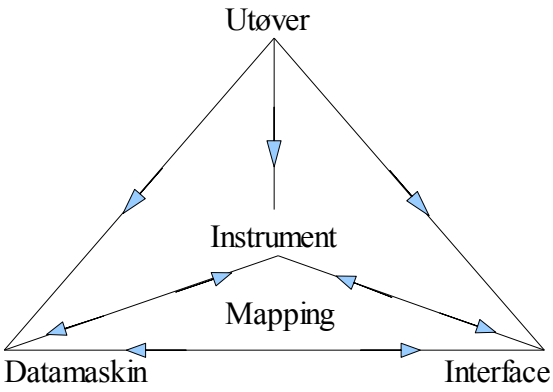
Denne typen interaksjon er uløselig knyttet til menneske på den ene siden og maskin på den andre, og det er innlysende at når mennesket behersker interaksjonsformatet, er det kvaliteten på linken mellom utøver og resultat som avgjør om resultatet av interaksjonen er god eller dårlig. På grunn av denne avhengigheten mellom delene i et komplett system, vurderer jeg valg av DSP-funksjoner (*Digital Signal Processing*), sensorer (sensorer er elektroniske komponenter som brukes til å kvantifisere et fysisk fenomen) og mappingteknikker (mapping-teknikker referer til hvordan vi bruker informasjon fra sensorer i forhold til det vi ønsker å styre) som likestilte. Med andre ord: Jeg tror ikke det er en smart metode å velge sensorer man kan tenke seg å bruke, for så å tvinge disse til å fungere godt sammen med DSP-funksjonene. Viktigheten av at forholdene mellom enhetene i sensorbaserte interaksjons-systemer er avstemt gjør seg like gjeldende her, som når pianoet, som instrumentgruppe, gang på gang blir utstyrt med tangenter i stedet for plekter i en pose ved siden av. Denne avhengigheten kommer til syne i denne oppgaven ved at DSP-funksjonene som er valgt, blir gjennomgått hver for seg for å gjøre det klart hvilke parametre som er tilgjengelige for kontinuerlige endringer. Og like viktig: hvilke parametre som må kontrolleres for å nå et definert auditivt mål.

1.2 **Tilnærmingsmetode**

I henhold til *ACM SIGCHI* sin definisjon (referert ovenfor), kan en tenke seg forholdene mellom utøver, mappingteknikker og maskinvare som angitt i figur 1.1, som utgangspunkt for interaksjonsmodellen i denne oppgaven.

2 **sample** [el. sæmple] *-et; -ing* gjøre et utvalg (av prøver som er representative for en helhet, f.eks. et statistisk materiale) Etym.: eng. (http://www.ordnett.no/ordbok.html?search=sample&publications=23). I en musikalsk sammenheng er ordet sample stort sett brukt om et utdrag av en lydfil, men kan også være en enkeltstående verdi: Ett sample i analog- til digitalkonvertering varer i 1/44100 sekund ved CD kvalitet.

3 The Association for Computing Machinery's Special Interest Group on Computer-Human Interaction.



Figur 1.1: Denne figuren illustrerer sammenhengen mellom utøver og et interaksjonssystem. Utøveren bestemmer i alle tilfeller hva og når noe skal gjøres, og datamaskinen er mappet (koplet) til utøveren på instrumentets premisser via et sensor- og brukergrensesnitt.

Figur 1.1 definerer et lukket system, ettersom en konfigurasjon av komponentene ikke kan endres uten å også endre en del av det totale systemet. Det kreves likevel at interaksjonen mellom utøver/instrument og datamaskin er fleksibel i forhold til valg av hensiktsmessige sensorer, og minst mulig forstyrrende integrasjon av sensorer for kvantifisering av menneske-instrument interaksjon. Dette legger føringer for systemdesigneren, men dersom det auditive målet er veldefinert, bør det i de aller fleste tilfeller være mulig å konstruere et system som kan hjelpe til med å løse problemet. En trend innenfor musikalsk interaksjonsdesign er å finne opp nye instrumenter, og det blir stadig presentert nye løsninger på det kommersielle markedet. Men for oss som ikke kan være uten det instrumentet vi allerede spiller, er løsningene ofte veldig individuelle og representerer naturlig nok ikke en stor grad av gjenbruk for ulike spilleteknikker. Derfor forsøker jeg i denne oppgaven å peke på generelle metoder for å kvantifisere menneske-instrument interaksjon, og se på hvordan dataene man får ut, kan brukes som kvalitative kontrollere for de aktuelle DSP-funksjonene.

DSP-funksjonene som er brukt i denne oppgaven, er ikke laget for å presentere nye DSP-teknikker, men er valgt ut i henhold til de ønsker jeg har for bearbeiding av *timbre*. Hensikten er å presentere tilnærmingsmåter til programvareutvikling med utgangspunkt i et definert auditivt mål. Siden det er elektroakustisk musikk som i utgangspunktet inspirerte meg til å skrive denne oppgaven, har jeg valgt å fokusere på granulering av samplet lyd. Granulering er en teknikk som er basert på å kutte opp lyd i små biter, *grains*, for så å behandle disse bitene individuelt (vi kommer tilbake til granulering i seksjon 4.5). Granuleringen er i denne oppgaven integrert med et typisk effektoppsett for instrumentalister: romklang, forsinkelse/

ekko, forvrenging og attakkmanipulering. Utfordringen i forhold til styring av effektene, er å definere en logisk sammenheng mellom de tilgjengelige parametrene, og finne en fornuftig og musikalsk metode for å gruppere disse, slik at ett sensorsignal kan styre mange parametre samtidig.

1.3 Oppgavens inndeling

På grunn av at mange av de akustiske instrumentene vi bruker regelmessig har utviklet seg over flere tusen år (se for eksempel Rossing, Wheeler og Moore, 2002), er det en logisk slutning at et instrument avhenger av en vellykket form og et vellykket betjeningspanel, for å bli tatt vare på og videreutviklet. Derfor er det muligens grunn til å tro at historien vil gjenta seg for programvarebaserte instrumenter. Forhåpentligvis vil forskning på koblingen mellom bevegelse og lyd gi oss varige og veldefinerte teknikker som gjør det mulig å ivareta de motoriske relasjonene mennesket har utviklet i forhold til lyd, når vi utvikler nye konsept for å kople bevegelser til lyd. Motoriske relasjoner til lyd er kulturbetinget ettersom instrument-bruk og instrumentdesign henger sammen: En tromme ment til bruk under rituell dans blir selvsagt ikke så stor og tung at den ikke kan bæres. Dersom vi fjerner de kulturbetingede faktorene, og bare ser på brukergrensesnitt for historiske akustiske instrumenter, er det ikke så vanskelig å tenke seg at man ved å se på en fremføring vet noe om hvordan en lyd frembringes. En harpe, en gitar og et piano har samme lydkilde, det vil si strenger, men spilles ulikt. Når vi kjenner igjen disse instrumentene bare ved å høre lyden av dem, kan vi trekke slutninger om hvordan lyden produseres. Dette synes jeg er ekstra interessant når det produseres lyder man ikke ventet å høre fra det lydproduserende instrumentet. Eksempel på dette er *preparerte* instrumenter som er et samlebegrep på at instrumentets lydkilde påvirkes ved fysisk inngripen. I *preparering* av en gitar er det vanlig å stikke gjenstander inn mellom strengene slik at de ikke får anledning til å svinge fritt. Pianister legger gjerne gjenstander oppå strengene som endrer strengenes svingninger, eller at strengene dempes med hånden for å fremheve instrumentets perkusive potensial. Uansett hvilke teknikker man bruker i en *preparering* er utøverens mål å utvide instrumentets lydpalett, og dermed får uerfarne lyttere utvidet sin oppfatning av hvilke lyder man kan forvente av for eksempel et piano.

Det interessante for meg er altså bevegelsen som forårsaker lyden, og det er tradisjonelle motoriske relasjoner til gitaren som er utgangspunktet for mappingteknikkene i denne oppgaven: Gitarister slår på strengene med en hånd, og bestemmer tonehøyde med den andre.

Jeg tror at på samme måte som man *preparerer* et instrument, kan vi i en overført betydning *preparere* motoriske vaner, og på den måten utvide et instruments auditive og tekniske bruksområde (med teknisk bruksområde menes den funksjonen et tradisjonelt grensesnitt kan ha overfor for eksempel et dataprogram).

For å finne ut mer om hvordan vi har forholdt oss til lydproduserende gester, etter at vi tok kontroll over lydkilden, er oppgaven organisert i fire hovedemne:

1. Historisk oversikt over elektroniske, musikalske grensesnitt
2. Utvikling av sensorgrensesnitt
3. Utvikling av DSP-programvaren
4. Mappingteknikker

Via disse fire delene ønsker jeg å se på utviklingen i kommersielt tilgjengelige elektroniske instrumenter, og gjennom utvikling av maskinvare og programvare vil jeg forhåpentligvis finne ut noe om hva jeg kan gjøre videre for å bidra til *NIME*. Det ligger i sakens natur at utvikling av nye grensesnitt fører til at man fjerner seg fra tradisjonelle musikkinstrumenters grensesnitt. Dette synes jeg er problemfritt, fordi en utvikling av nye grensesnitt ikke bør ansees som en trussel mot eksisterende instrumenter, men en berikelse. For min del synes jeg det er problematisk når et instrument, gammelt eller nytt, blir "forbedret" til det ugjenkjennelige i forhold til bruk, men beholder sin form: At instrumentet gir seg ut for å være noe det ikke lenger er. Jeg har stor tro på at de tradisjonelle instrumentene som er mest vanlige i dag har mye å tilføre designprosessen. Siden tradisjonelle instrumenter er foredlet gjennom hundrevis (klaveret), om ikke tusenvis av år (gitaren), og fremdeles er hovedinstrument i ulike musikksjangrer, finner jeg ingen grunn til å tro at disse instrumentgruppene vil opphøre selv om det kommer mange teknologiske nyvinninger. Jeg tror den viktigste grunnen til at tradisjonelle instrumenter vil bestå er den fysiske interaksjonen med lydkilden; noe man til en viss grad mangler i et programvaremiljø. For at disse to verdener - den akustiske og den elektroniske - skal møtes, tror jeg det er lurt at jeg begynner min utvikling av nye musikalske brukergrensesnitt på eksisterende instrumenter.

Siden oppgaven inkluderer utvikling av praktiske verktøy for musikalsk interaksjon vil en stor del av teksten være en forklaring av de valg som er gjort i forhold til verktøyene, samt en gjennomgang av utviklingsprosessen i forhold til DSP-funksjoner og sensorgrensesnitt.

Hovedkapitlene er:

1. Introduksjon, det vil si dette kapittelet
   Presentere oppgavens omfang og bakgrunn, samt plassere arbeidet i riktig kontekst.

2. Historisk overblikk
   Utviklingen av brukergrensesnitt for musikere er et like gammelt felt som instrumentene selv, og det er hensiktsmessig for oppgaven å gi et overblikk over nyere teknologi og allerede eksisterende grensesnitt for elektroniske/databaserte instrumenter.

3. Maskinvareutvikling
   For at det skal være enklere for andre studenter uten elektronikkbakgrunn å lage sitt eget sensorgrensesnitt, en enhet som kvantifiserer sensorsignaler, vil jeg gå gjennom de ulike aspektene ved utviklingen, og hva man kan gjøre for å forenkle prosessen.

4. Programvareutvikling
   DSP-programmet som er lagt ved oppgaven, er et supplement til kapittel fire og fem, og under dette punktet vil jeg beskrive teknikkene jeg har implementert, samt gi en begrunnelse for hvilke parametre jeg mener må kontrolleres.

5. Mappingteknikker
   Dette kapittelet vil gi et kort overblikk over nyere mappingteknikker. I tillegg vil jeg forsøke å lage et oppsett som fungerer på de fleste fysiske instrumenter, inkludert nye brukergrensesnitt for musikalsk interaksjon.

6. Oppsummering, konklusjon og et prospekt for videre arbeide
   Et av mine hovedmål for denne oppgaven er at jeg personlig skal ha lært så mye om moderne musikkteknologi at jeg kan bidra til *NIME*, samt at jeg til enhver tid skal kunne løse mange av de praktiske utfordringene jeg kommer til å møte på som komponist og utøver.

2. Historisk overblikk over programmerbare grensesnitt

Dette kapittelet gir en oversikt over utviklingen av programmerbare musikalske brukergrensesnitt siden 1950. Programmerbare grensesnitt er de grensesnittene som lar brukeren lagre innstillinger for bruk ved en senere anledning, eller som bruker programmer for å utføre operasjoner. Hensikten er å gi et bilde av de produktene som har ledet frem til dagens generelle grensesnitt hvor brukeren selv kan velge hvilke sensorer som skal benyttes.

2.1 **Introduksjon**

I løpet av det 20. århundre har verden sett et enormt antall nye elektroniske instrumenter komme til det kommersielle markedet. Den største delen av dette markedet er instrumenter i keyboardkategorien, men i denne sammenhengen kan vi ikke utelate de mange effekt-prosessorene som har spilt en stor rolle i den musikalske utviklingen. For å avgrense dette temaet, som kunne vært minst en oppgave alene, vil jeg begynne med Max Mathews forskning ved Bells laboratorium fra og med året 1957. Jeg er oppmerksom på den australske *CSIRAC* maskinen (Doornbush, 2006), men siden dette ikke er en diskusjon om hvem som var først eller en hyllest til en spesiell oppfinnelse, vil jeg konsentrere meg om nyvinningene som gjorde det mulig å ha en form for interaksjon med elektroniske instrumenter. Jeg kommer altså ikke til å gå inn på teknologien som gjorde det mulig å få lyd ut av datamaskiner.

Det er imidlertid to instrumenter som ikke kan programmeres, som ble funnet opp i henholds-vis 1919 og 1928 som må nevnes i en oppgave som denne: *Theremin* og *Ondes Martenot*.

Theremin ble laget av russeren Leo Theremin, og instrumentet produserer en sinustone med tonestyrke og tonehøyde styrt av bevegelser i forhold til to antenner. Teknikken som brukes for å kvantifisere *proximity* (nærhet) mellom hånd og antenne (Rossing, Moore og Wheeler, 2002, s.599) er lettest å skjønne ved å undersøke hvordan de elektroniske komponentene vi kaller *kondensatorer* fungerer. *Kondensatorer* kjennetegnes ved at de kan lagre et elektrisk energipotensiale og elektriske ladninger, og består i utgangspunktet av to *plater* av et ledende material separert med enten et isolerende material, vakum eller luft (Young og Freedman, 2004, s.908 og 909). Når vi spiller på en *Theremin* danner antennen den ene *platen,* og hånden den andre. *Platene* er isolert fra hverandre av luften mellom de, og ved å bevege hånden i

forhold til antennen endres størrelsen på det isolerende materialet (luften) seg, og *kondensatorens kapasitans* endres (*kapasitans* er angitt ved et tall, og uttrykker en *kondensator*s evne til å lagre energi). I en *Theremin* er hver av disse *kondensatorene* (hånd/objekt-antenne) en del av en større elektronisk krets: En *oscillatorkrets* (US Patent 1661058). En *oscillator* er en elektronisk krets, eller en programvarefunksjon, som produserer et eller flere signal med en gitt, eller variabel frekvens. I elektronikkbøker vil denne typen kretser stå omtalt under *L-C kretser*, og dersom det er ønskelig med mer informasjon om denne typen kretser, henviser jeg til for eksempel Young og Freedman (2004). Det viktigste i forhold til å forstå hvordan *Theremin* fungerer er koplingen mellom hånd/objekt og antennen. Kretsene som styres av denne koplingen gjør om signalet man får ved å endre *kapasitans* ved bevegelse, til en spenning som kan brukes til å bestemme henholdsvis frekvens og amplitude i de respektive kretsene. Det er også verdt å nevne at tonehøyden vi oppfatter ikke er direkte mappet til en enkelt signalgenerator, men *beat*-frekvensen som produseres av differansen mellom en *oscillator* med fiksert frekvens, og en *oscillator* som endrer frekvens som følge av bevegelse. *Beat*-frekvenser er frekvenser som oppstår når to rene signaler, for eksempel sinuser, ligger så nærme hverandre at man bare oppfatter en tone: *"In the case of two pure tones of slightly different frequency, linear superposition at our ears leads to a sensation of audible beats at the difference frequency* Δf*. These beats are heard as a pulsation in the loudness of the tone having the average frequency* $f = \frac{1}{2}(f_1+f_2)$*"* (Rossing, Moore og Wheeler (2002, s154).

Theremin har ingenting med datamaskiner å gjøre, men i forhold til musikalsk interaksjon representerer *Theremin* noe helt nytt; nemlig et musikkinstrument som ikke krever fysisk kontakt med utøveren for å produsere lyd, men som likevel er avhengig av en utøver for å klinge. Denne typen interaksjon har fått en renessanse de senere årene via videoanalyse av bevegelser, noe vi kommer tilbake til i den siste seksjonen i dette kapittelet.

Ondes Martenot, oppfunnet av Maurice Martenot, er en klassiker innen sjangeren elektroniske instrumenter. Den fungerer omtrent som *Theremin*, men i stedet for at tonene alltid var trinnløse har *Ondes Martenot* tangenter. I tillegg har den en ring festet til en akse (som bestemmer kapasitansen til en variabel kondensator) som utøveren kunne føre opp og ned langs tangentene, og på den måten produsere glissando. (Rossing, Moore og Wheeler, 2002). Siden *Ondes Martenot* var et monofont instrument, som *Theremin*, forholdt man seg til polyfonisk musikk slik man kan se i figur 2.1.

Figur 2.1: Bildet er hentet fra Simeone (2002), og illustrerer hvordan man fremførte polyfon musikk med Ondes Martenot.

Et annet viktig aspekt ved *Ondes Martenot* er at den har brytere som tillater brukeren å endre lydens *timbrale* kvaliteter mens man spiller. Disse bryterne lar brukeren justere dynamikk (tonestyrke) og kombinasjonen av frekvenser i tonens overtonespekter: "*The instrument's arrangement prefigured that of almost every keyboard synthesizer today: right hand for basic frequency (pitch), and left hand for providing articulation features that change the character of the sound"* (Paradiso, 1997).

Det er flere instrumenter som kunne vært omtalt, men siden dette kapittelet er tenkt som en rask gjennomgang av de viktigste instrumentene i utviklingen av det vi kjenner som programmerbare grensesnitt, er det et bevisst valg å utelate produkt som ikke har innført noen ny teknikk. Det er altså ikke tilstrekkelig å være en variasjon av et tidligere instrument, eller en ny utgave hvor forskjellen ligger i bedre ytelse/spesifikasjoner.

Max Mathews er regnet som computermusikkens far, og hans arbeide med musikk og datamaskiner startet i 1950årene. Derfor er resten av dette kapittelet organisert etter tiårene fra 1950 og frem til i dag. Kilde for dette kapittelet er Roads (1996) dersom ikke annet er nevnt.

2.2 **1950-årene**

Dette tiåret markerer starten på computermusikken og utviklingen av nye brukergrensesnitt for elektroniske instrumenter. I 1957 introduserte Max Mathews dataprogrammet *Music I,* som kunne produsere lyd etter å først ha mottatt informasjon fra brukeren om tonehøyde, notelengde og bølgeform. *Music I* var svært begrenset, men åpnet opp for en fortsettelse i arbeidet med utviklingen av programvare for musikk. Da *Music I* ble utviklet, var data-maskinene bygget med rør, og de var temmelig trege. I 1958 skrev Mathews programmet *Music II* som kunne spille fire stemmer samtidig, og i tillegg kunne brukeren velge mellom 16 forskjellige bølgeformer. Grunnen til at kapasiteten til *Music II* var så mye større enn for *Music I*, var at datamaskinene det ble utviklet for ble konstruert med transistorer som tillot raskere bearbeiding av data enn med rørbaserte datamaskiner.

2.3 **1960-årene**

Max Mathews fortsatte sitt imponerende arbeide, og hvert eneste program han lanserte utnyttet at teknologien ble bedre og tilbød mer regnekraft. I 1960-årene kom imidlertid en av hans viktigste oppfinnelser, nemlig konseptet med *Unit Generator (UG)*. *UG-konseptet* er lett å skjønne dersom vi tenker oss at hver *UG* er en programvarefunksjon som kan gjøre et definert sett av operasjoner uten at vi forandrer på programkoden. Et eksempel kan være en *UG* som utfører en matematisk funksjon hver gang den blir spurt om det. Da er det likegyldig hva man skal bruke svaret til, for *UG* gjør akkurat det den er programmert til å gjøre uavhengig av formålet med operasjonen. *UG* kan altså sees på som ferdige programvare-brikker som brukes i byggingen av et dataprogram (Mathews, 1963). *Music III* var det første programmet som utnytta konseptet med *UG*. Denne teknikken vil brukere av programvare som *Max/MSP/Jitter* og *PureData*, som i dag er to av de mest brukte verktøyene innen musikkteknologien, kjenne seg igjen i, og det kommer vi tilbake til senere. Mathews var også en pioner når det gjelder nye enheter for musikalsk interaksjon, og i 1969 ble programmet *GRIN* (*Graphical Input*) lansert (Mathews og Rosler, 1968). *GRIN* registrerer dataene fra *Light Pen*, som lot brukeren tegne lyd på en skjerm, som i sin tur ble oversatt til informasjon om envelope og bølgeform som *Music IV* kunne bruke til å produsere lyd.

Utover 1960-tallet kom det flere versjoner av *UG-baserte* programmer, og i takt med utvikling av teknologien ble også funksjonalitet og lydkvalitet forbedret ettersom AD-konvertere og datamaskiner ble raskere. På grunn av det store antallet programmer som har

blitt utviklet, vil jeg henvise til Roads (1996) for en generell oversikt over de ulike programmene.

2.4 **1970-årene**

Dette tiåret er preget av at elektronikk, og da spesielt mikrokontrollere, ble kommersielt tilgjengelig med en pris som gjorde det mulig å bruke disse mikrokontrollerene til å gjøre synthesizere programmerbare. Tidlige analoge synthesizere kunne ikke programmeres, men med denne nye funksjonaliteten ble repertoaret til disse instrumentene utvidet ved at utøvere og komponister kunne lagre *presets*. *Presets* er et begrep som brukes om forhåndslagrede innstillinger. Disse presetene kunne hentes frem igjen med et knappetrykk, og man kunne også definere hvor lang tid vandringen fra et preset til et annet skulle ta sånn at endringene i innstillingene ble utført over tid. Med andre ord: Interpolering mellom to sett av verdier.

Mathews og Moore lanserte i 1970 *Groove* (Mathews og Moore, 1970), et program skrevet for datamaskinen *DDP-224*. *Groove* gjorde det mulig å lagre, reprodusere og redigere funksjoner som kunne utføres over tid.

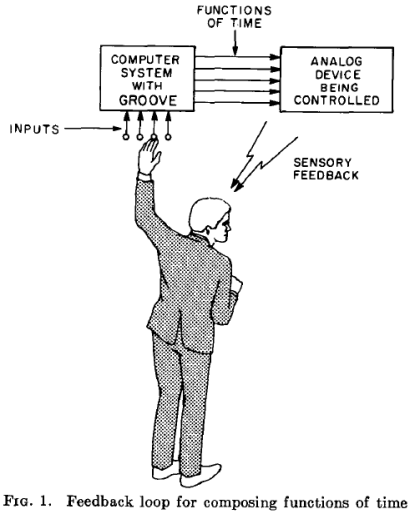


FIG. 1. Feedback loop for composing functions of time

Figur 2.2: Illustrasjonen er hentet fra Mathews, Moore (1970), og viser menneskets interaksjon med Groove samt dataflyten i systemet: "We have chosen to emphasize the human rather than the direct feedback. Systems without human links require more intelligent programs, or from a different viewpoint,with the addition of a human link, we can currently do more complex tasks. In addition, the authors, who wrote the program, enjoy not only being in the loop but retaining command" (Mathews, Moore, 1970).

Det viktigste med *Groove*, slik jeg ser det, var at mennesket ble inkludert som en viktig del av tilbakekoblingssystemet, som vist i figur 2.2, og kunne dermed ha kontinuerlig kontroll over *Groove.*

Mot slutten av 1970-tallet kunne *Groove*-funksjonalitet implementeres i mikrokontrollere. Digitalstyring av synthesizere kunne bygges inn i instrumentet, og det var ikke lenger nødvendig å ha digital kontroll separert fra den analoge lydgeneratoren som var tilfellet med *Groove*. Jon Appleton utviklet *Dartmouth Digital Synthesizer*, som etterhvert ble hetende *Synclavier* (se figur 2.3). *Dartmouth Digital Synthesizer* var det første produktet med sanntid digital kontroll innebygd, og produktet startet en nye era innen synthesizerdesign.

Figur 2.3: Synclavier (1975) var den første digitale synthesizer med sanntid interaksjon.[4]

De produktene som er nevnt i denne seksjonen er de jeg anser som mest interessante i løpet av 70-tallet, men ettersom sampling er en viktig del av DSP-programvaren i denne oppgaven, synes jeg det er verdt å nevne at Fairlight Computer Music Instrument lanserte verdens første kommersielle tangentbaserte digitale sampler, *CMI*, i 1979 med 8-bits oppløsning. Jeg ønsker imidlertid ikke å gå mer inn på samplere som musikkinstrumenter, siden de ikke introduserer noe nytt i forhold til utviklingen av programmerbare grensesnitt.

2.5 **1980-årene**

1980-tallet er *MIDI*-protokollens første tiår, og *MIDI* ble utviklet av Dave Smith, en av eierne av *Sequential Circuits*, sammen med *Oberheim* og *Roland*, for å konstruere en standardisert

4 Bildet er hentet fra: http://www.obsolete.com/120_years/machines/synclavier/index.html

kommunikasjonsprotokoll for kontroll av styringssignaler. Ettersom mikrokontrollere nå var blitt så billige at man kunne bygge de inn i musikkinstrumenter, førte det til at de ulike produsentene løste kommunikasjonen mellom mikrokontrollere, grensesnitt og styrings-signaler på sin egen måte: De utviklet hver sin protokoll. Dette gjorde kommunikasjon mellom instrumenter fra ulike produsenter vanskelig, og *MIDI* ble løsningen på problemet. *MIDI* ble allment tilgjengelig i 1983, og det var i tillegg en åpen standard for at alle som ville slutte seg til den, kunne bruke den uten å måtte betale lisens eller lignende avgifter. Firmaet *Sequential Circuits* laget verdens første midibaserte instrument, som er vist i figur 2.4.

Figur 2.4: Prophet 600 var verdens første midibaserte instrument. (Bildet er hentet fra: http://www.siliconbreakdown.com/prophet600.htm)

MIDI har i over 20 år vært den dominerende protokollen for kommunikasjon mellom digital-styrte instrumenter. Selv om mange nå arbeider for å få et gjennomslag for *Open Sound Control (OSC)*, en ny og moderne kommunikasjonsprotokoll (vi kommer tilbake til *OSC* i kapittel 3), er det ikke umulig at *MIDI*-protokollen fester grepet igjen i og med at *MIDI Manufacturers Organisation (MMA)* har innsett svakhetene med *MIDI* og startet utviklingen av *High-Definition MIDI*[5].

På grunn av den ekstreme tilgangen på nye instrumenter gjennom 1980-årene ser jeg ingen grunn til å liste opp ulike instrumenter i dette kapittelet. Jeg tror det er noenlunde bred enighet om at *MIDI* er det viktigste som skjedde musikkteknologien på 80-tallet. Likevel vil jeg trekke frem et instrument som skiller seg ut i mengden, nemlig *Radio Baton*, laget av Max Mathews i 1987 med patent innvilget i 1990[6]. *Radio Baton* registrerer posisjon i tre dimen-sjoner for hver stikke, og ut fra posisjonsdataene beregnes tidspunktet for når stikken vil treffe den tilhørende matten, som i tillegg til å huse elektronikken fungerer som slagflate.

5 Se http://www.midi.org for mer info om status på utviklingen av *High-Defintion MIDI*.
6 http://www.google.com/patents?vid=USPAT4980519&id=hRUcAAAAEBAJ&printsec=abstract&zoom=4&dq=radio+baton

2.6 **1990-årene**

Don Buchla[7] er ansvarlig for flere interessante musikkinstrumenter, og tre av dem ble utviklet i løpet av dette tiåret. *Thunder*[8] er Buchla sin første *MIDI*-kontroller (Buchla, 2005) og består av 26 touchsensorer, hvorav 14 også registrerer posisjon. *Thunder* introduserte et alternativ til tangentinstrumenter i form av at det manglet mekaniske komponenter, og at det er kontinuerlige målinger som styrer mottakeren av *MIDI*-dataene. Tangentmetaforen er derfor ufullstendig i og med at et tradisjonelt tangentinstrument ikke utfører kontinuerlige målinger, og at premissene for lyden legges i anslaget. *After touch*[9] kompenserer for dette i mange digitalstyrte tangentinstrumenter.

Buchla lanserte *Lightning II*[10] i 1994. *Lightning II* består av to trådløse stikker som hver sender ut infrarøde signaler, og en mottaker som registrerer signalene fra stikkene og regner ut velocity og akselerasjon. I tillegg utfører mottakeren en analyse av stikkenes bevegelse:

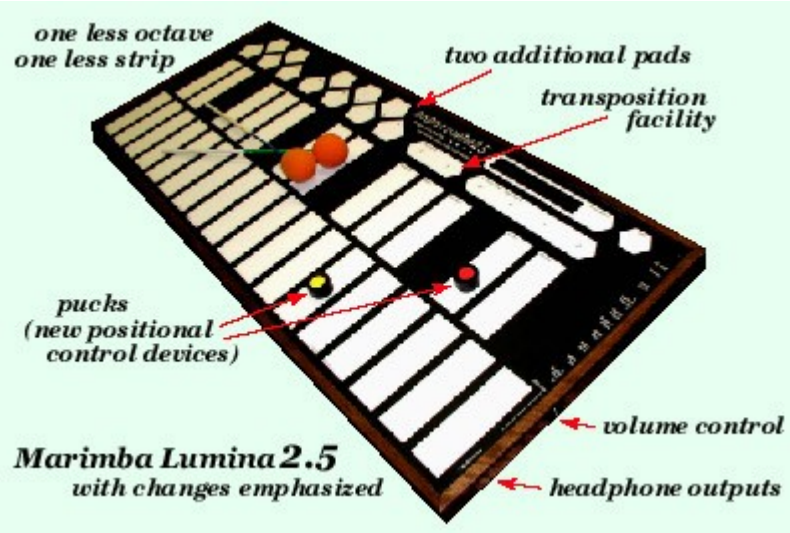
> "*LIGHTNING II features a conducting facility that can analyze a conductor's gestures, display deviations from a preset tempo, and signal errors such as missed beats. Simultaneously, LIGHTNING can transmit a synchronous MIDI clock for controlling external sequencers and output programmed note data to accompany specific beats within a measure*"
>
> (Lightning Description, se fotnote nr 10).

Lightning II hadde også en innebygget 32-stemmers synthesizer, så dette var et fullverdig lydproduserende instrument.

Neste og 1990-tallets siste Buchla innovasjon var *Marimba Lumina*[11], som ser ut som en flat marimba med 42 elektroniske tangenter som aktiviseres av stikkene man bruker. Stikkene er myke, man kan bruke inntil fire samtidig, og instrumentet vet hvilken stikke som treffer hvor. Slagflaten deles inn i soner slik at utøveren kan bruke overflaten som en *touch-pad*. *Marimba Lumina* tilbyr et ganske avansert brukergrensesnitt, og jeg henviser til nettsiden for instrumentet for mer informasjon: http://www.buchla.com/mlumina/features.html. Se også figur 2.5

7 http://www.buchla.com/
8 http://www.buchla.com/historical/thunder/index.html
9 *Aftertouch* registrerer endring i påført trykk etter at tangenten er slått an (Roads, 1996).
10 http://www.buchla.com/lightning/descript.html
11 http://www.buchla.com/mlumina/description.html

for en beskrivelse av instrumentets funksjoner.



Figur 2.5: Marimba Lumina (Bildet er hentet fra http://www.buchla.com/mlumina/press.html)

Det siste punktet under dette tiåret er *OSC*[12], som ble utviklet ved CNMAT og publisert i 1997. *OSC* er som nevnt tidligere en kommunikasjonsprotokoll, og den er med i oversikten over programmerbare grensesnitt fordi den åpner for at utøvere kan kjøpe ferdige *Single Board Computere (SBC)*, og lage sitt eget grensesnitt med kommunikasjon via *Ethernet.* Protokollen ble utviklet for å overvinne problemene med *MIDI*, og tiden som har gått siden *OSC* ble introdusert, har vist at den er veldefinert og brukbar for en rekke applikasjoner. *OSC*-protokollen blir gjennomgått i detalj i kapittel tre siden den henger direkte sammen med maskinvareutviklingen.

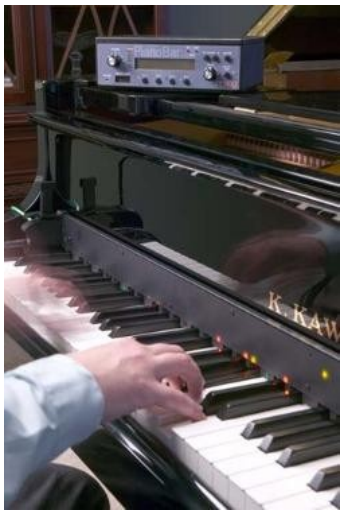
2.7 **2000-2007**

De siste syv årene har det vært en nærmest kontinuerlig utvikling i bruken av avanserte sensorteknikker innen design av musikkinstrumenter. Sett fra mitt ståsted er utviklingen av sensorgrensesnitt, som gir brukeren mulighet til å kople på sensorer ut fra egne preferanser, en av de viktigste tendensene de siste årene. Franske La Kitchen[13] tilbyr sensorgrensesnitt for et utvalg sensorer med sine *OSC*-kompatible *Kroonde Gamma* og *Toaster*. Andre produkter av

12 http://www.cnmat.berkeley.edu/OpenSoundControl/ og http://opensoundcontrol.org/
13 http://www.la-kitchen.fr/kitchenlab/kitchenlab.html

denne typen er *Eobody*[14] og *Ethersense*[15] fra Ircam, *Teabox* fra Electrotap[16] og så videre. Denne typen grensesnitt vil bli gjennomgått i kapittel 3. Det er imidlertid ett grensesnitt som ikke er åpent for ekstra sensorer, men som likevel representerer en liten revolusjon i inter-aksjonsdesign, og det er firmaet *Jazzmutant* sin *Lemur*: En berøringsskjerm som kan registrere mange punkter samtidig. Dette vil si at man kan bruke alle ti fingrer samtidig på en og samme skjerm, og via ferdiglagde grafiske objekter kan man konfigurere og bruke Lemur som et miksebord, en x/y kontroller, eller hva som helst annet som til enhver tid er tilgjengelig fra Jazzmutants grafikkobjektdatabase. Sensorteknikken i *Lemur* er basert på et rutenett av tynne, semitransparente tråder som registrerer posisjon for berøring[17]. En teknikk som denne kan forøvrig også være interessant i forhold til overflatebehandling på tradisjonelle instrumenter. Jeg har ikke funnet patentskrivet for *Lemur* ennå, så en diskusjon rundt hvorvidt det er mulig eller ikke, kan jeg ikke si noe om i denne omgang.

Et annet produkt som heller ikke er åpnet for tilkopling av sensorer, men som likevel bør tas med her, er Moog *Pianobar*. Moog *Pianobar*, som vist i figur 2.6, er utviklet av Bob Moog og Don Buchla[18], og er en enhet som settes direkte på et standard 88-tangenters klaviatur. *Pianobar* kvantifiserer tangentbevegelser og gjør tilgjengelig informasjon om tonehøyde og velocity på samme måte som man er vant til på en *MIDI-basert* digital synthesizer.



Figur 2.6: Bildet viser Moog Pianobar montert på et flygel, og kontrollerboksen står oppå flygelet (Bildet er fra http://www.moogmusic.com/detail.php?main_product_id=71).

14 http://www.ircam.fr/237.html?&L=1&tx_ircamboutique_pi1[showUid]=9&cHash=9a73c8e406
15 http://www.ircam.fr/237.html?&L=1&tx_ircamboutique_pi1[showUid]=12&cHash=0703d70a6e
16 http://electrotap.com/teabox/
17 De tekniske opplysningene er beskrevet på bakgrunn av visuell inspeksjon av Lemur, og samtale med en representant for Jazzmutant under Lyd og bilde messen i Oslo, 2007.
18 http://www.moogmusic.com/newsarch.php?cat_id=24

Moog *Pianobar* har sensorer som finner ut hvilken tangent som blir trykket ned på pianoet, og sender noteverdi og velocity til kontrollenheten. Kontrollenheten har innebygde lyder, som i en vanlig digital synthesizer, og bruker sensordataene fra *Pianobar* som *MIDI*-verdier (http://www.moogmusic.com/detail.php?main_product_id=71).

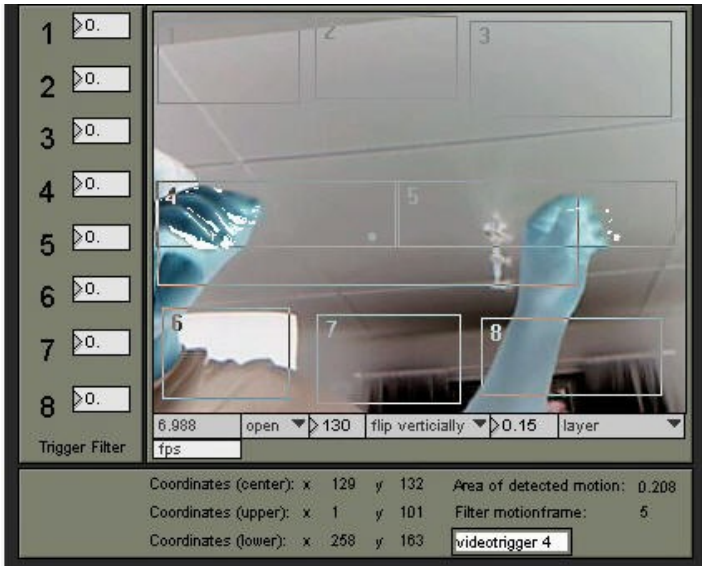
2.8 **Oppsummering kapittel 2**

I de foregående kapitlene har jeg forsøkt å plukke ut de produktene som jeg mener har satt en standard for musikalsk interaksjon og som fortsatt er viktige og utnyttbare. Under skrivingen av dette kapittelet har det slått meg, at det er en tendens til at mange produkter nå deler design med Max Mathews *Groove* system. *Groove* var i grove trekk basert på et program som lot komponisten skrive inn et partitur, og et digitalt grensesnitt for styring av analoge lydmoduler. Nå i de seneste tiårene har vi sett at databaserte musikkinstrumenter blir styrt med analoge/ digitale sensorer via avanserte sensorgrensesnitt, og jeg tror at dette er et systemdesign som vil holde seg lenge. Derfor vil neste kapittel handle om utviklingen av et sensorgrensesnitt for standard analoge sensorer[19].

Som nevnt i forbindelse med gjennomgangen av *Theremin,* er videoanalyse med på å videreutvikle menneske-maskin interaksjon uten fysisk berøring, og jeg har laget en *Max/ MSP/Jitter* patch, vist i figur 2.7, som demonstrerer en metode for videoanalyse. Patchen er en videreutvikling av et program jeg laget for *Drivhuset, Stiftelsen for musikalsk verksteds-arbeid*[20]. Jeg liker godt å eksperimentere med videoanalyse, men sliter litt med å finne gode løsninger for å kontrollere inn- og utgang av skjermbildet: Når man skal slå an en streng kan håndens hastighet mot strengen være så høy man klarer, men likevel reduseres til et minimum like før man treffer strengen. Med videoanalyse er det et problem at bevegelsene inn i, og ut av bildet, må enten være så raske eller sene at systemet ikke registrerer bevegelsen. Dette ønsker jeg å finne en løsning på, og den enkleste metoden jeg har kommet på, er å plassere en sensor under foten, eller mellom fingrene, som kan registrere et anslag for å bestemme når videoanalysen skal iverksettes. Ved å bruke en ekstra sensor på denne måten vil det være lett å stoppe analysen midt i en bevegelse, og på den måten gjøre videobildet til en fullt kontrollerbar matrise. Dersom man har en sensor per hånd vil det også være enkelt å slå av for eksempel venstre hånds bevegelse, men beholde høyre hånd.

19 Med standard analoge sensorer menes standard i forhold til hvilke typer analoge sensorer som til en hver tid er tilgjengelig.

20 Link til Drivhuset sine nettsider: http://drivhuset.musikkverksted.no/



Figur 2.7: Max/MSP/Jitter patch for videoanalyse. Patchen lar brukeren definere 8 triggere, og gir i tillegg koordinater og areal i forhold til størrelsen på registrert bevegelse som angitt i bildet.

Video 2.1 demonstrerer videopatchen vist i figur 2.7. I denne videoen er bildeoppdateringen veldig treg, på grunn av at det tas bilde av dataskjermen samtidig som bevegelsene blir visualisert. Jeg kommer ikke til å gå mer inn på videoanalyse i denne oppgaven, ettersom dette temaet alene kunne fylle mange masteroppgaver, men nevner det her fordi det er en viktig del av moderne interaksjonsformer. Videoene som det refereres til i oppgaven ligger vedlagt på cd, sammen med programvaren som gjennomgås i kapittel tre, fire og fem. Se filen *les meg* på cd-platen for informasjon om de ulike filene.

For mer informasjon om bruk av video i *Max/MSP/Jitter* henviser jeg til *Jitter* dokumentasjonen fra Cycling74, samt artikler på http://cycling74.com/section/indepth, spesielt *Jitter Recipes 1,2 og 3*. Det er disse artiklene samt dokumentasjonen for *Jitter* som er brukt for å programmere patchen i figur 2.7.

3. Sensorgrensesnitt

Dette kapittelet beskriver maskinvaren som er utviklet som en del av oppgaven. Hensikten er å begrunne de valgene som er gjort med utgangspunkt i hva som kreves av et sensor-grensesnitt fra et mappingperspektiv, og hvorfor det er nødvendig å oppgradere/bytte ut MIDI som standard kommunikasjonsprotokoll.

3.1 **Spesifikasjoner for sensordata**

Signalets oppløsning og overføringshastighet er de to viktigste parametrene når det gjelder sensordata til musikalsk bruk. Oppløsning i en analog til digitalkonvertering (AD-konvertering) definerer hvor detaljert det digitale signalet gjengir det analoge, og både antall bit og samplingfrekvens må tas med i beregningen. Overføringshastigheten definerer hvor fort vi får brukt informasjonen. Det finnes mange grensesnitt på markedet, og flere av disse har spesifikasjoner som er langt over behovet for mappingen i denne oppgaven. Men på grunn av at sensorgrensesnitt er en viktig del av mange interaksjonssystem, fant jeg det hensiktsmessig å trekke utviklingen av maskinvaren inn i oppgaven. Et annet viktig moment for å ta dette med i oppgaven, er at det produseres veldig mange like enheter som ikke er skalerbare i særlig grad (med skalerbar menes hvorvidt grensesnittet kan bygges ut og endres etter behov). Et kriterium som må være oppfylt før et sensorgrensesnitt er skalerbart er for det første at kapasiteten kan utvides (antall sensorinnganger), samt at det ideelt sett bør kunne konfigureres av sluttbrukeren slik at det kan passe til alle typer sensorer. Noen sensorer trenger 2,5 volt andre 3,3 eller 5 volt, og et kommersielt sensorgrensesnitt bør ha støtte for slike variasjoner. I denne oppgaven vil det ikke bli lagt vekt på produktdesign når det gjelder utforming og støtte-elektronikk for brukervennlighet, og tilpassing av spenning til hver enkelt sensor gjøres på prototypenivå. Det finnes riktignok mange åpne *MIDI*-enheter, men bakdelene i forhold til oppløsning og hastighet gjør *MIDI*-protokollen til et lite hensiktsmessig valg, samtidig som at *OSC* har etablert seg som en standard i de fleste musikkteknologiske forskningsmiljø.

Videre finner jeg det fornuftig å gjøre informasjon om konstruksjon av et sensorgrensesnitt av høy kvalitet tilgjengelig, slik at interesserte kan bygge sitt eget grensesnitt uten å måtte sette seg inn i elektronikk som fag. Det kan innvendes mot denne holdningen at kunnskap om elektronikk er nødvendig for å konstruere et sensorgrensesnitt, men personlig mener jeg det er likestilt med muligheten man har for å kjøpe en ny hals til en elektrisk gitar, og bytte den selv

uten å vite noe om instrumentbygging. Tilsvarende nivå innen elektronikken er en formfaktor (en formfaktor definerer en standard for hva et produkt innen den aktuelle formfaktoren kan gjøre, produktets størrelse og så videre) som går under fellesbetegnelsen *Single Board Computer (SBC)*, og referer til en komplett datamaskin med ekstern kommunikasjon bygget inn i en enhet. (Se seksjon 3.4 for mer om valg av elektronikk.).

3.2 **Kommunikasjonsprotokoll**

Kommunikasjonsprotokoller har det til felles at de skal gjøre interaksjon mellom to elektroniske enheter enklere, og ved å holde seg til en standard – noe det er enighet om i et miljø kalles en standard – blir det lettere for sluttbrukeren å kople sammen produkter fra ulike leverandører. Tom White, president og *CEO* for MMA, har et veldig godt argument for å fortsette å bruke *MIDI*, og sier dette om den nye protokollen de utvikler under navnet *High Definition MIDI:*

> *"MIDI has worked fantastically for more than 20 years, but with today's computers and embedded microprocessors we can do much more than MIDI was originally designed to", said MMA President and CEO Tom White. "But more important, the new protocol enables us to build upon MIDI without abandoning all of the great hardware and software that already exists"*
>
> (http://www.midi.org/newsviews/hdmidipr2.shtml).

Sitatet ovenfor kan være en indikasjon på at *MIDI* kommer til å bli værende lenge, og i forhold til *OSC* vil det kanskje bety at *OSC* ikke blir en vanlig kommersielt tilgjengelig protokoll ettersom *MIDI* er så utbredt. Men så lenge en bedre versjon av *MIDI* ikke er lansert, er det fornuftig å støtte de protokollene som er åpne og fleksible med tanke på oppløsning og overføringshastighet.

Hovedsaken for både *MIDI* og *OSC* er å transportere informasjon fra sensorer til en mottaker. Det som skiller *MIDI* og *OSC*, er at *MIDI* er strengt definert både når det gjelder hvor mange bit vi kan gjøre nytte av fra AD-konverteringen, og når det gjelder overføringshastighet av styringsignalet. *OSC* stiller ingen krav til oppløsning, og overføringshastigheten er styrt av tilgjengelig *Ethernetteknologi*. Et annet moment er at *MIDI* krever spesielle drivere for å kommunisere der *OSC* bruker standard *Ethernetprotokoller*. For sluttbrukeren betyr det at ingen installasjon av maskinvaredrivere er nødvendig, og at utstyret vil fungere på både

Linux, *Windows* og *Apple*. Disse faktorene gjør *OSC* mer fleksibelt enn *MIDI*, og det er enklere å tilpasse utstyret til kravene man til en hver tid har til bitoppløsning, samplefrekvens og overføringshastighet i forhold til både behov og økonomi. Jeg har laget flere sensor-grensesnitt etter dette som blir omtalt i denne oppgaven, og selv om jeg personlig foretrekker den varianten som blir beskrevet her, er det i mange tilfeller unødvendig å lage et så avansert grensesnitt. En mikrokontroller fra for eksempel *Atmel* med nødvendig støtteelektronikk koster ikke mange hundrelappene, og kan kjøre åtte AD-konvertere med en samplefrekvens på mange megahertz med 10-bit oppløsning. Dette er i mange tilfeller en god løsning dersom *Ethernetfunksjonalitet* ikke er nødvendig, og man klarer seg med overføringshastigheten som til en hver tid er tilgjengelig på *RS232*: En *Atmel Mega32*, som er en fin mikrokontroller for små grensesnitt, har overføringshastighet på maks 2,5Mbs når den kjøres på 20.0Mhz (Atmel, 2006). Ved å velge en løsning som denne, vil det være en enkel sak å lese datastrømmen fra en serieport, og konvertere dette til *OSC* i programvare på datamaskinen, slik at dataene kan distribueres videre til andre datamaskiner via *Ethernet*.

I forhold til overføringshastighet, har *MIDI*-spesifikasjonen vært uforandret i mer enn 20 år. Helt siden protokollen ble introdusert i 1983 har 7-bit oppløsning og en overføringshastighet på 31250 baud[21] vært standard. Litt matematikk viser da at 8 analoge sensorer overført som *MIDI CC*-beskjeder *(Control Change)* tar tid: *MIDI CC* består av tre 8-bits byte, hvilket betyr at det er 24-bit per analog sensor for hver beskjed. For 8 sensorer blir det: 24-bit*8sensorer som gir 192 bit. Tiden dette tar finner vi ved denne formelen: 1s/baudrate*antall bit=tid: (1s/31250^{bit}/s)*192bit=0,006144s. I utgangspunktet er 0,006s veldig lite og akseptabelt for de fleste applikasjoner, men dette er bare overføringstiden. Legger man sammen tiden det tar for mikrokontrolleren å konvertere det analoge signalet fra sensoren til en ferdig *MIDI*-pakke kan brukes av mottakeren, begynner forsinkelsen å bli merkbar. Electrotap[22] har gjort tilgjengelig en tabell, gjengitt i figur 3.1, som viser en utregning i forhold til deres eget produkt *Teabox*.

21 Baud referer som regel til overføringskapasitet i bit/s.
22 http://www.electrotap.com

	MIDI	Teabox
Time needed to aquire the data from all 24 sensors and an additional start flag and hardware version number.	23.04 ms (not accounting for jitter)	0.25 ms
Number of times data (all 24 sensors) is updated per second	43.47 Hz	4000 Hz
Latency for any given sensor (Teabox time is based on an I/O vector size of 64 and a sample-rate of 44.1 KHz. MIDI is again displayed with a theoretical number. It is common to have the MIDI latency be at least twice as high).	23.04 ms (not accounting for jitter)	1.71 ms

Figur 3.1 Sammenligning av kommunikasjon via OSC og MIDI.
(Kilde: http://electrotap.com/teabox/)

Hastigheten som er oppgitt for *Teabox*, er ikke representativ for hastigheten til sensor-grensesnittet som er utviklet for denne oppgaven, ettersom *Teabox* kommuniserer via *S/PDIF* og ikke *Ethernet*. Tallene i figur 3.1 er likevel en god indikasjon på hvor raske kommersielle *OSC*-grensesnitt er i forhold til grensesnitt som følger *MIDI-spesifikasjonen*. Tallene i figur 3.1 er selvforklarende, men det er verdt å merke seg at latenstiden – som er den faktoren som har størst praktisk betydning for utøveren – er vesentlig lavere for *OSC* enn *MIDI*.

3.3 **Elektronikk**

Sensorgrensesnitt er bygget opp rundt en *SBC* fra *Z-World* som vist i figur 3.2, og program-meres ved hjelp av utviklingsprogramvaren *Dynamic-C*. *SBCen* heter *Z-World BL 2600 Wolf*. Den ble valgt på grunn av gode spesifikasjoner både når det gjelder AD-konvertere og *Ethernetkontroller*.

Figur 3.2: Z-World BL 2600 Wolf (Bildet er hentet fra manualen)

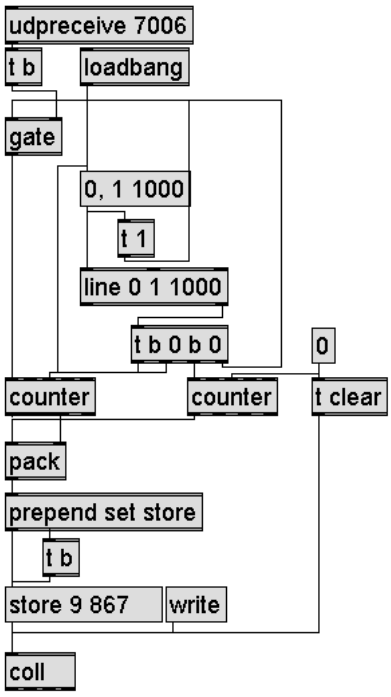
Spesifikasjonene for *Z-World BL 2600 Wolf* er presentert i appendix 4, men det har ingen

hensikt å gå gjennom alle punktene i tabellen ettersom mange av dem er selvforklarende. Men konfigurasjonen av maskinvaren spiller inn på ytelsen, og fokuset i denne oppgaven har vært å lage et så enkelt oppsett som mulig, for å finne ut hva som er minstekravet for at *BL2600 Wolf* skal fungere som et *OSC*-grensesnitt. Fremgangsmåten var å først lære programmerings-språket *C,* og finne ut hvordan en *OSC*-datapakke må være for at mottakeren skal skjønne innholdet i pakken. I appendix 1 er *OSC-spesifikasjonen*, som også er tilgjengelig fra http://www.opensoundcontol.org, presentert, og den inneholder alt man trenger å vite om utviklingen av *C-kode* for et ganske avansert *OSC*-system. Imidlertid er det ganske for-virrende lesing dersom man ikke er vant til *C.* Derfor var en viktig del av min strevsomme ferd gjennom *OSC*-spesifikasjonen å komme inn til kjernen, og se om kommunikasjonen ville fungere med et minimum av kode. Det gjorde den, og kildekoden for sensorgrensesnittet finnes i appendix 2. Det eneste man trenger å ta hensyn til i formateringen av pakken, er at den blir satt sammen riktig. Metoden man lager for å legge dataene inn i en *OSC-kompatibel* pakke er revnende likegyldig, men hver pakke må inneholde all informasjon som kreves for at mottakeren skal kunne hente ut dataene på riktig måte.

Det er flere mulige formateringer for *OSC*-pakker, og koden i appendix 2 pakker dataene som en *OSC-message*. Grunnen til det, er at jeg syntes *OSC-message* formatet så lettest ut ettersom den krever bare en *adresse*: En *OSC-message* er bygget opp av en *adresse* fulgt av *typetags,* og til sist verdiene fra sensorene. Hver pakke må inneholde et antall bit som kan deles på 32 (svaret må være et heltall). Adressestrengen begynner alltid med tegnet '/' også adressenavnet. Neste er *typetags* som alltid begynner med ',', og så en indikasjon på hva slags data som er i pakken. I dette tilfellet er det 8 heltall som angis med tegnet 'i'. Til slutt kommer *argumentene* som her er verdiene fra 8 AD-konvertere:

/avc	,iiiiiii	1	2	3	4	5	6	7	8

Siden hastighet er avgjørende for sanntidinteraksjon, laget jeg et *Max/MSP*-program, se figur 3.3, som teller antall registrerte pakker inne i *Max/MSP*. Jeg synes dette er en fornuftig måte å finne den reelle latenstiden, fordi med en oppdateringsfrekvens på 12000 ganger per sekund er definitivt ikke flaskehalsen i AD-konverteringen. Prosessoren arbeider på 44.2MHz, så det er heller ingen grunn til å tro at den lager store forsinkelser.



Figur 3.3: Max/MSP patch som teller pakker mottatt pr sekund.

Resultat fra programmet som vist i figur 3.3 er angitt i tabellen i figur 3.4:

Tabellen viser antall pakker pr. sekund i 50 sekund:				
0, 865; 1, 867; 2, 865; 3, 866; 4, 867; 5, 864; 6, 867; 7, 865; 8, 867; 9, 865;	10, 865; 11, 866; 12, 865; 13, 867; 14, 865; 15, 867; 16, 865; 17, 867; 18, 865; 19, 866;	20, 865; 21, 867; 22, 867; 23, 863; 24, 867; 25, 865; 26, 867; 27, 865; 28, 866; 29, 865;	30, 867; 31, 865; 32, 867; 33, 864; 34, 865; 35, 867; 36, 865; 37, 866; 38, 865; 39, 867;	40, 865; 41, 866; 42, 865; 43, 867; 44, 867; 45, 864; 46, 865; 47, 867; 48, 865; 49, 866;

Figur 3.4: Tabellen viser mottatte OSC-pakker fra programmet i figur 3.3 over en periode på 50 sekunder. Jeg gjør oppmerksom på at tallene kan variere dersom datamaskinen har mye å gjøre.

Av tabellen ser vi at antall pakker holder seg temmelig stabilt med et avvik på ±3 pakker. Dette tilsvarer: 1000ms/864pakker/s=1.15ms pr pakke. Merk at dette gjelder for 8 analoge innganger, og at dersom man ønsker å registrere flere sensorer vil det føre til flere AD-konverteringer, at pakken blir større, og at det dermed vil ta litt lenger tid fra sensoren endrer tilstand til endringen er registrert i mottakerprogrammet.

3.4 **Dynamic C**

Dynamic C er et komplett utviklingsverktøy fra *Z-World*[23], og består av følgende deler:

1. *Teksteditor*: Her skrives koden, og teksteditoren uthever ord som er definert som del av *Dynamic C*.
2. *Kompilator*: En kompilator gjør om tekstkoden til maskinkode.
3. *Linker*: En linkfunksjon i denne sammenhengen betyr at kompilatoren automatisk inkluderer bibliotek og andre filer som programkoden referer.
4. *Loader*: Å "loade" betyr å laste maskinkoden inn i datamaskinen.
5. *Debugger*: En debugger tester et program for feil.

Dynamic C er som navnet angir en "dynamisk" versjon av *C*. Hvor dynamisk den til syvende og sist er skal ikke diskuteres her, men for å bruke en *SBC* fra *Z-World* må *Dynamic C* brukes for utvikling av programkoden som kreves for at *SBCen* skal virke. Ifølge manualen er *Dynamic C* laget for å gjøre utviklingen av programkode for *embedded systems*[24] lettere for programmereren, og hovedforskjellen mellom *Dynamic C* og standard *C,* ligger i at først-nevnte har tillegg som eliminerer behovet for spesialtilpassing av standard *C,* slik at det fungerer for hver enkelt mikrokontrollerarkitektur (Z-World, 2003). På tross av at det er forskjeller, er det standard *C* som brukes når man programmerer, og det er altså unntakene heller en reglene som gjør at ordet *Dynamic* henges på. For mer informasjon om likheter og ulikheter referer jeg til manualen for *Dynamic C* som finnes på internet, og til spesifikasjonen for standard *C*[25]. Se for eksempel *Deitel (2004),* for litteratur om generell *C*-programmering.

3.5 **Gjør det selv**

For å lage et sensorgrensesnitt som dette, må en kjøpe et utviklingskit for *Z-World BL2600 Wolf*, lese manualen så man skjønner hvor strømmen skal koples på, og hvordan programmeringskabelen virker. Programmeringskabelen er en spesialkabel som koples mellom datamaskinen og *BL2600 Wolf,* slik at koden man utvikler kan lastes inn i minnet på *SBCen*. Etter det er det bare å ta koden i appendix 2 inn i *Dynamic C*, slå på strømmen på *BL2600*

23 http://www.zworld.com/ Zworld er en del av Rabbit Semiconductor som igjen er en del av Digi.

24 Embedded system referer til datamaskiner som er satt til å gjøre spesifikke oppgaver feks å kontrollere en installasjon. Skillet mellom embedded systems og vanlige personlige datamaskiner viskes stadig mer ut ved at teknologien blir raskere og mindre, og maskiner som kan bruke Linux eller Windows operativsystemer blir stadig mindre og dermed brukbare i kontrollinstallasjoner. (Se feks: http://en.wikipedia.org/wiki/Embedded_system

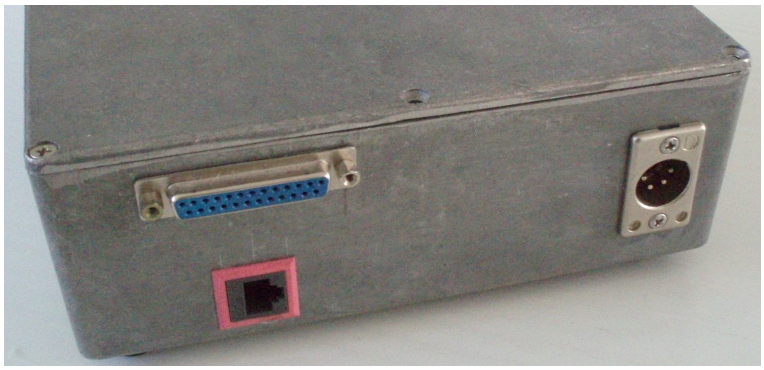
25 http://www.open-std.org/jtc1/sc22/wg14/www/standards.html#9899

Wolf og trykke F9 på tastaturet. Jeg minner om at koden som er vedlagt her, bare er egnet til å samle inn data fra 8 analoge innganger på *BL2600 Wolf*. Den er ikke direkte kompatibel med standard *C*. Dersom man velger å lage sitt eget grensesnitt er det ikke spesielt vanskelig å skjønne hvordan man enkelt utvider *C-koden* til å ta med digital IO i tillegg. For å gjøre det veldig enkelt kan vi se på et utdrag fra koden i appendix tre (koden ligger også på cd-platen):

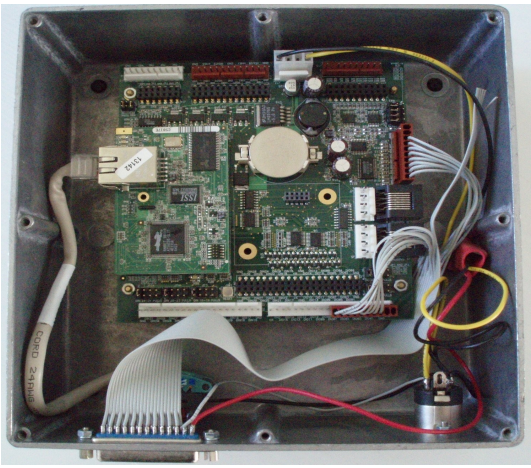
a0=anaIn(0,2); // Her leses det fra analoginngang 0, og verdien plasseres i variabelen a0.
a1=anaIn(1,2); // 2 tallet som står i alle parentesene definerer gainfaktoren for forsterkeren
//som sitter på hver inngang.

Dersom man ønsker å lese fra digitale innganger er de definert på samme måte: "*digIn (0);"* leser fra digital inngang 0, og det eneste man trenger å gjøre er å definere en variabel for å lagre verdien fra inngangen og legge denne til i *OSC-beskjeden.*

Den ferdige prototypen til denne oppgaven har tilkoplinger via vanlig *D-SUB* kontakt, men jeg anbefaler ikke den løsningen fordi alle sensorer da blir koplet til samme plugg, noe som gjør lodding og håndtering unødvendig tungvint. Det er en bedre løsning å bruke et standard en-til-en (en sensor per plugg) oppsett med feks *XLR*- eller *jackplugger*. Figur 3.5 og 3.6 viser den ferdige prototypen som er utviklet for denne oppgaven.



Figur 3.5: Tilkobnlinger.



Figur 3.6: BL2600 Wolf montert inni boksen

Jeg er fornøyd med funksjonaliteten til sensorgrensesnittet, og kommer til å bygge videre på det som er gjort i forbindelse med denne oppgaven. Grensesnittet har vært testet over tid for å se om det mister kontakten med datamaskinen, og hittil har jeg ikke observert noe som tyder på at det er en fare for at det skal skje. Det ble nevnt tidligere i dette kapittelet at produkt-design i forhold til brukervennlighet og utforming ikke ville bli prioritert, men i ettertid ser jeg at det hadde gjort testingen lettere dersom jeg for eksempel hadde valgt en plugg per sensor, og lagt opp til et fleksibelt brukergrensesnitt for strømstyring for sensorer med behov for ulik spenning.

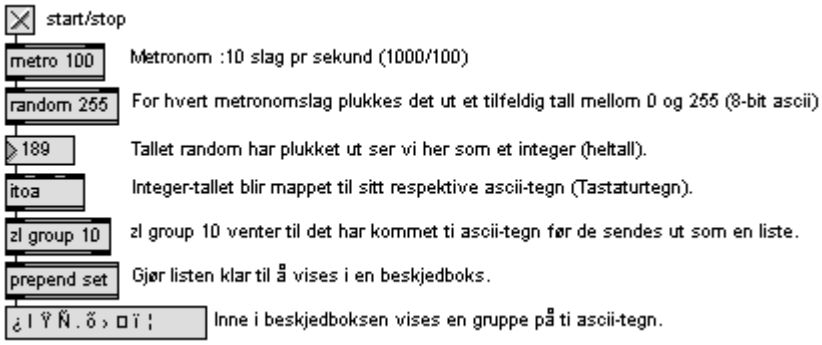
4. Lydbehandling

Dette kapittelet gir en introduksjon til programmeringsteknikker i Max/MSP/Jitter, og gir en gjennomgang av de valg og implementeringer som er gjort i forhold til manipulering av lyd i sanntid. Litteraturen jeg har brukt for programmeringsteknikker i Max/MSP Jitter, er dokumentasjonen og eksemplene som følger med programvaren. Se cd for programvare.

4.1 **Max/MSP/Jitter**

Max ble tilgjengelig ved Ircam[26] som et verktøy for å lage interaktiv musikk i 1986, men det var ikke før i 1991 da Opcode[27] tok over at *Max* ble kommersielt tilgjengelig. Miller Puckette som utviklet *Max* i utgangspunktet, fortsatte sammen med David Zicarelli å utvikle *Max* i Opcode. I 2000 tok Cycling74[28] over distribusjonen av programmet, og David Zicarelli, grunnleggeren av Cycling74, utvidet *Max* til også å kunne behandle lyd: *MSP*, *Max Signal Processing*. I 2003 ble også video/matrisedata lagt til i form av programvarepakken *Jitter* (Cycling74, 2006).

Utvikling av programmer i *Max/MSP/Jitter* gjøres ved at man plasserer grafiske objekter inne i en *patcher*. En *patcher* er et hvitt ark, og etter hvert som man fyller opp med objekter og knytter objektene sammen med tråder, blir det til slutt et program som vist i figur 4.1.



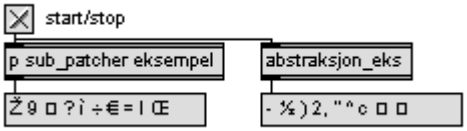
Figur 4.1: Max patcher eksempel.

Som vi ser av de grafiske objektene i figur 4.1 er noen av dem markert med et mørkt felt i ett eller flere hjørner. Disse feltene representerer innganger til objektene, og det er via disse inngangene, eller *inlets* som de heter i *Max/MSP/Jitter*, man sender beskjeder til objektene.

26 Institut de Recherche et Coordination Acoustique/Musique (www.ircam.fr)
27 Opcode eksisterer ikke lenger, men ble etablert i 1985, kjøpt av Gibson Guitar Corp. i 1998, og utviklingen av Opcode produkter stoppet i 1999. (Wikipedia: http://en.wikipedia.org/wiki/Opcode_Systems)
28 www.cycling74.com

Dataflyten i en *patcher* skjer alltid fra høyre mot venstre, som i praksis betyr at dersom et objekt har flere *inlets* er det *inlet 0*, altså det som er lengst til venstre, som gjør at objektet produserer data i et eller flere *outlets*. I figur 4.1 ser vi at alle objektene har et eller flere *inlets,* og dersom vi hadde sendt informasjon inn i *inlet 1,2,...[n],* ville den informasjonen blitt lagret, og tatt med i beregningen av data neste gang det kommer informasjon inn *inlet 0.* Programmet utføres altså fra høyre til venstre, og videre i den rekkefølge objektene er koplet sammen med tråder.

Objektene representerer programkode som opprinnelig er skrevet i programmeringsspråket *C* og som er kompilert til å være et *Maxobjekt*. Det går også fint an å utvikle sine egne objekter i *C* og kompilere disse som *Maxobjekter*. Dette hever fleksibiliteten til *Max/MSP/Jitter,* og gjør det til et komplett utviklingsmiljø for interaktive applikasjoner med tall, lyd og matriser/ video. En *patcher*-teknikk i *Max,* som går igjen ganske ofte i de programmene som ligger ved oppgaven, er *sub-patcher* og *abstraksjoner*. En *sub-patcher* er ganske enkelt en måte å rydde i *hovedpatcheren* på, og ved å legge et program inn i en *sub-patcher* kan mange objekter fremstå som ett. En *abstraksjon* er noe av det samme, men i motsetning til en *subpatche*r blir *abstraksjoner* lagret som selvstendige objekt, som kan hentes inn i en hvilket som helst patch bare ved å bruke navnet på *abstraksjonen*. Figur 4.2 viser hvordan en *abstraksjon* skiller seg fra en *subpatcher* i en *patcher*.



Figur 4.2: Patcheren fra figur 4.1 som subpatcher og abstraksjon. En subpatcher begynner alltid med bokstaven p, og en abstraksjon har et selvstendig navn.

For å skille mellom de ulike komponentene for tall, lyd og video/matriser har utviklerne valgt å legge dette til navnet på objektene:

Figur 4.3: De ulike objekttypene i Max/MSP/Jitter.

Som vi ser fra figur 4.3 har et *Maxobjekt* ingen ekstra attributt i tillegg til objekt navnet '*', mens *MSP* har tegnet ~ , og *Jitterobjekt* begynner alltid med *jit.[objektnavn]*. Dette gjør det

enklere for programmerere å skille mellom objektene, og det er til god hjelp for å forstå hvordan *Max/MSP/Jitter* lar data vandre fritt mellom de ulike domenene. *Maxobjekter* kan koples sammen med både *MSP*- og *Jitterobjekter*, og *Jitter* og *MSP* kan koples sammen: *"Because Max turns all control information into a simple stream of numbers, you can "patch" anything to anything else"* (Cycling74, 2006). For videre lesing om *Max/MSP/Jitter* anbefales dokumentasjonen som følger med programpakken, samt David Zicarellis artikkel *How I Learned to Love a Program That Does Nothing* (Zicarelli, 2002). I denne oppgaven er det mest *Max/MSP* som er brukt både til lydbehandling, behandling av sensordata og mapping, samt visualisering av lyd og matematiske funksjoner. *Jitter* brukes i videoanalyse-eksempelet i seksjon 2.8.

Min utvelgelse av DSP-funksjoner i denne oppgaven er et direkte resultat av personlige preferanser, men det er bare granulering av samplet lyd som skiller seg ut fra de mer vanlige gitareffektene. Med de vanlige effektene menes overstyring, forsinkelse/ekko og romklang. Jeg har ikke statistikk å vise til som beviser at disse tre er de mest vanlige gitareffektene, men det er god grunn til å tro at iallefall overstyring og romklang er riktig plassert i denne kategorien ettersom de har vært inkludert i gitarforsterkere i lange tider. Fjærklangen ble integrert i Leo Fenders *Twin Reverb* tidlig på 1960-tallet, og ble med det en industristandard[29]. Overstyring har vært så godt som en standard fra en gang på 60-tallet, og rockhistorien er, i tillegg til å være en beskrivelse av en epoke, et dokument som kunne beskrevet utviklingen av gitarlyd, og dermed overstyring som musikalsk effekt.

Prinsippet med forsinkelse som musikalsk effekt er som navnet sier: å forsinke lyd. I musikalske sammenhenger er det som regel en kombinasjon av det originale og en forsinket versjon av det originale signalet som kjennetegner bruk av forsinkelse. Store konsertsteder med lydanlegg bruker ofte forsinkelse for å kompensere for avstanden fra scene til publikum slik at den akustiske lyden ikke skaper interferens med den forsterkede lyden. Dette gjøres ved å sette høyttalere et stykke bak i konsertlokalet som sender ut forsinket lyd slik at det passer med den lyden som kommer direkte fra scenen eller større høyttalere lenger fremme.

En annen viktig del i bruken av forsinkelse som musikalsk effekt er at man kan ta opp over et definert tidsrom, for så å la opptaket avspilles i en loop (repetisjoner). Pierre Schaeffer og Pierre Henry var tidlig ute med denne teknikken i sitt arbeide med *Musique Concrete*, og de

29 http://www.accutronicsreverb.com/history.htm

laget loop ved å lime sammen enden av tapen med starten, og ved å bruke plater med lukkede spor: *Sillon fermé* (Godøy, 2006). Siden de arbeidet med tape var det også nærliggende å endre hastighet og retning på avspillingen, og med dette oppnå endring i tonehøyde (på grunn av endring i avspillingshastighet) samt baklengs avspilling av det innspilte materialet (Rossing, Moore og Wheeler, 2002). Senere ble det vanlig å inkludere tilbakekobling (*feedback*) av det forsinkede/repeterte signalet til inngangen av opptakeren. Ved å tilbake-koble signalet legges lyden oppå seg selv i flere lag, og forholdet mellom lagene defineres av tilbakekoplingens amplitude. Ved hjelp av denne teknikken kan små musikalske objekt bli til store kompliserte lydtepper.

4.2 **Implementering av romklang**

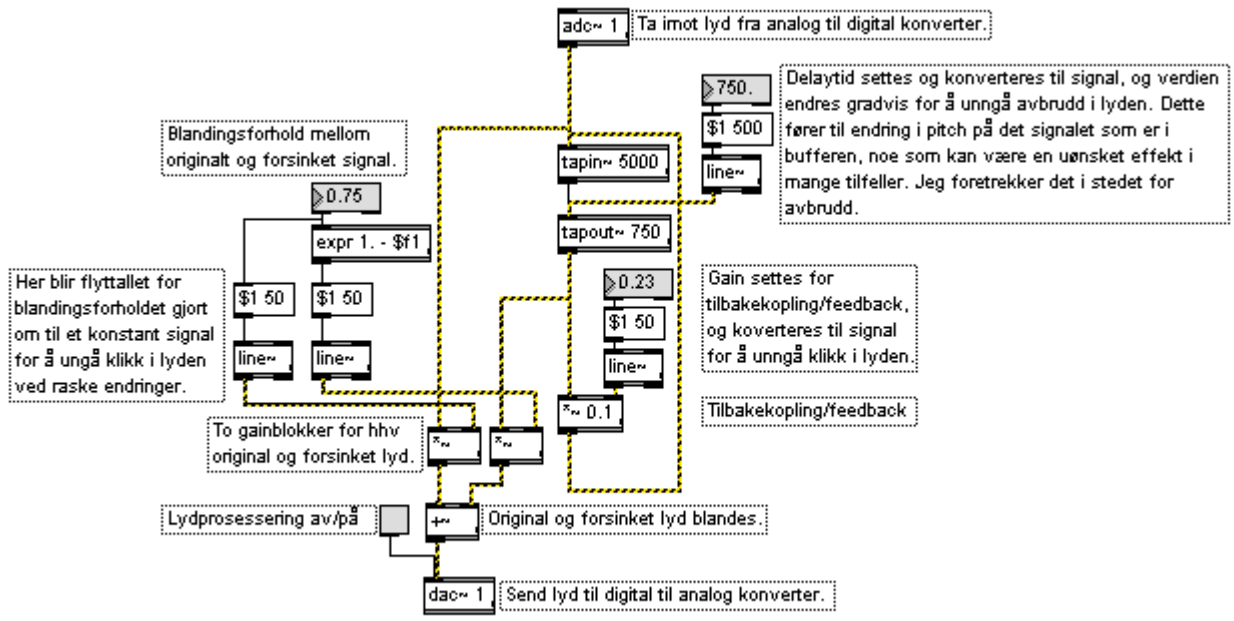
Romklangmodulen som er brukt i denne oppgaven, er hentet fra Olaf Matthes[30] *Max/MSP* objekt *monoverb~*, som er en monoversjon av *freeverb~* fra samme utvikler. Begge utgavene er basert på en romklangmodell kjent som *Schroeder/Moorer* algoritmen. Schroeder foreslo to måter å lage realistisk romklang på. Den første metoden var fem *allpassfilter* i serie, og den andre summerte signalet som kom ut fra fire parallelle *kamfilter*, for deretter å sende det videre inn i to *allpassfiltre* (Analog Devices,1998). Et *allpassfilter* har jevn frekvensrespons i det hørbare området, og virker ved at det forsinker ulike frekvensregioner med ulik tid: frekvensavhengig faseskift (Roads, 2000). Et *kamfilter* lager en serie topper og daler som er spredt med lik avstand over hele frekvensspekteret (Roads, 2000). Schroeder sine løsninger for å produsere kunstig romklang løser ikke alle problemer, og James A. Moorer[31] foreslo flere forbedringer i forhold til den originale *Schroederalgoritmen*. Et av problemene Moorer ville finne en løsning på, var at høye frekvenser hadde en tendens til å klinge lenger enn lave frekvenser. Hans løsning var å bruke *kamfilter* med et *lavpassfilter* i tilbakekoplingen for hvert romklangtrinn for å øke tettheten (*density*) i romklangen. Videre introduserte Moorer *preforsinkelse* for å simulere romklang i store haller: alt for å gjøre klangen mer realistisk. I denne oppgaven er det ikke simulering av rom som er det viktigste, men romklang brukes først og fremst for å skape en helhetsfølelse i lyden etter at den har blitt kuttet opp og reorganisert gjennom granulering. I tillegg er grensesnittet for innstilling av romklang lagt inn i sammen med innstillingene for vreng, fordi jeg liker å sende lyden gjennom vreng etter romklang. Denne metoden gjør at vrengen fremhever kompleksiteten i lyden som kommer ut

30 http://www.akustische-kunst.org/maxmsp/
31 http://www.jamminpower.com/jam.jsp

av romklangen, og etter min smak er dette et effektivt verktøy for å la originallyden nærme seg en slags støyestetikk. Se figur 4.11 for en oversikt over romklangens fire tilgjengelige parametre: romstørrelse, demping, våt og tørr.

4.3 **Implementering av forsinkelse**

Forsinkelse er et fundamentalt element i mange ulike DSP-prosesser, deriblant pitchskifter, ekko, chorus, flanger og forvrengning. På grunn av allsidigheten til forsinkelse som fysisk fenomen, finnes det brukt innenfor så og si alle musikalske sjangere som inkluderer elektroniske instrument. Ekko er den enkleste forklaringen på hva forsinkelse er, og i betydningen av at både ekko og forsinkelse returnerer en forsinket utgave av den originale lyden er det riktig, og ordene kan brukes om hverandre. I denne oppgaven brukes ordet forsinkelse på grunn av at ekko vanligvis brukes om naturlige fenomener, og at det i naturen alltid er en fysisk sammenheng mellom original lyd og tiden det tar før den returneres. Denne avhengigheten mellom original og returnert lyd er ikke tilstede i elektroniske forsinkelse-effekter selv om den naturligvis kan simuleres.

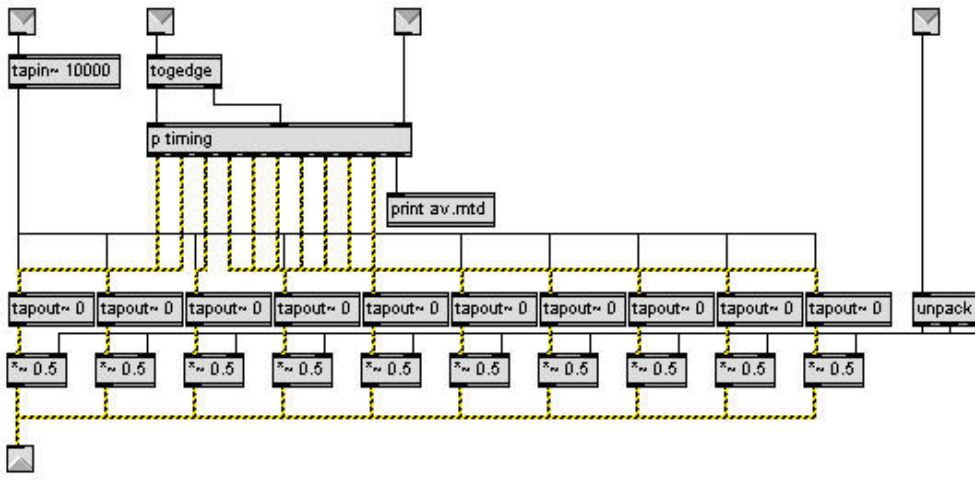


Figur 4.4: Max/MSP patch for forsinkelse av lyd med tilbakekopling og blanding mellom originalt og forsinket signal.

En digital forsinkelse er bygget rundt en såkalt sirkelbuffer, som er en buffer med gitt lengde der data blir lagret suksessivt fra begynnelse til slutt i en repeterende funksjon. En sirkel-

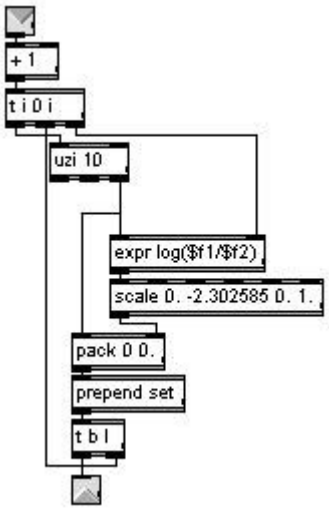
buffers lengde (antall sample det skal være plass til i bufferen) bestemmer hvor lang forsinkelsestiden er, og må regnes ut i forhold til samplefrekvensen: Lengde = samplefrekvens * sekund. Gainverdien for tilbakekoplingen av den forsinkede lyden settes med en multiplikasjonsfaktor mellom 0.0 og 1.0. Settes den høyere vil amplituden øke kontinuerlig mot uendelig. Dette bør generelt sett unngås, men dersom man ønsker å bruke overflyt (gainfaktor over 1.0 i tilbakekoplingen) som oppbyggingsteknikk av lydtepper må termineringen, eller en reduksjon av tilbakekoplingens gain, være underlagt brukerens kontroll, ellers kan man risikere skader på ører og utstyr. Et forsinket signal som multipliseres med seg selv med en faktor over 1.0 og tilbakekoples, blir veldig fort vondt å håndtere. Forsinkelse er brukt to steder i dette prosjektet: det er i granuleringen, og som simulert ekko i forhold til en tidskonstant definert av utøveren (tidskonstanten er som regel definert i et forhold til musikkens tempo). Figur 4.4 viser en *Max/MSP*-patch som er et eksempel på forsinkelse med tilbakekobling.

Jeg har lenge benyttet sirkelbuffer med tilbakekopling for forsinkelse ettersom det er den mest vanlige metoden. I mange applikasjoner fungerer den fint, men under arbeidet med denne oppgaven har det blitt et estetisk problem at den har svært begrensede kontrollmuligheter. Ettersom forsinkelse med tilbakekopling gir repetisjoner av lyden som reduseres med samme faktor hver gang den koples tilbake, får man ikke god kontroll over hvor mange repetisjoner forsinkelsen skal produsere. Repetisjonenes gain er heller ikke tilgjengelige som individuelle størrelser. For å løse dette problemet har jeg benyttet en annen metode som ikke har tilbake-kopling, men som simulerer det med ulike forsinkelse av samme buffer, og individuell kontroll over repetisjonenes gain. Denne metoden er vist i figur 4.5.



Figur 4.5: Abstraksjonen av.mtd som bruker ti forsinkelser av samme buffer for å simulere tilbakekopling, og fortsatt ha kontroll over lydens repetisjoner.

Ved å forsinke lyden som *patchen* i figur 4.5 viser, får man full kontroll over hver enkelt repetisjon, og man kan også prosessere forsinkelsene individuelt med ulike DSP-funksjoner, for eksempel med filtrering, for å simulere naturlig ekko. Jeg foretrekker en logaritmisk skalering av repetisjonenes styrke, og for å automatisere innstillingen av volum for hver forsinkelse laget jeg *patchen* i figur 4.6. Grunnen til at jeg tar skaleringen av styrke med i teksten, er at det viser hvordan skalering av verdier via matematiske funksjoner forenkler interaksjonen med programvare. Logaritmisk skalering er bare et eksempel på hvordan styrken til forsinkelsene kan skaleres, og hvilken funksjon som helst kan benyttes til dette formålet avhengig av hva slags effekter man ønsker å skape.



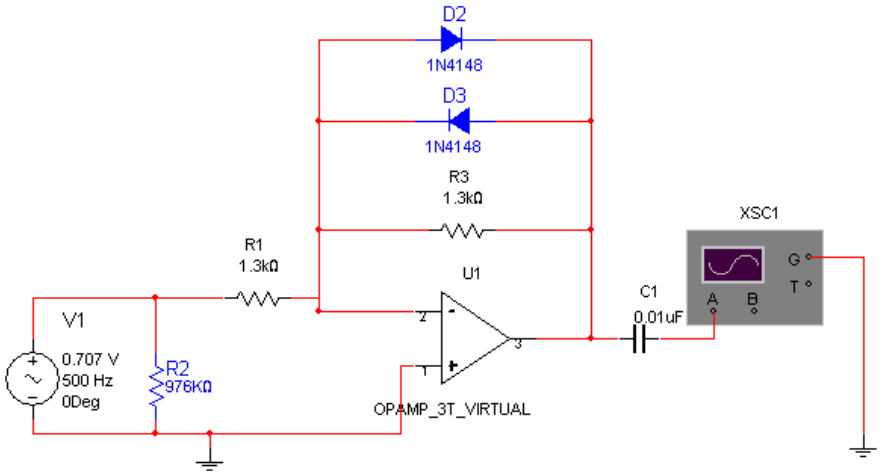
Figur 4.6: Denne patchen skalerer styrken til hver enkelt forsinkelse, slik at simuleringen av tilbakekopling får ønsket karakteristikk.

Et eksempel på hvor nyttig det kan være å ha individuell kontroll over repetisjonene i en forsinkelse er å sammenligne med en *stepsequencer*, som er en avspiller som beveger seg trinnvis i henhold til et satt tempo, eller en sampler: Ved å sette forsinkelsen slik at hver forsinkelse av originallyden er i henhold til musikkens tempo, kan man bruke forsinkelse med individuell kontroll over repetisjonene som en stepsequencer/sampler ved å sette antall repetisjoner og justere volumet for hver repetisjon.

4.4 **Forvrengning**

Figur 4.7 viser en typisk elektronisk krets for en diodeklipper som er en av de vanligste metodene for å lage vrengeffekter for elektrisk gitar. Det er for omfattende å gi en detaljert

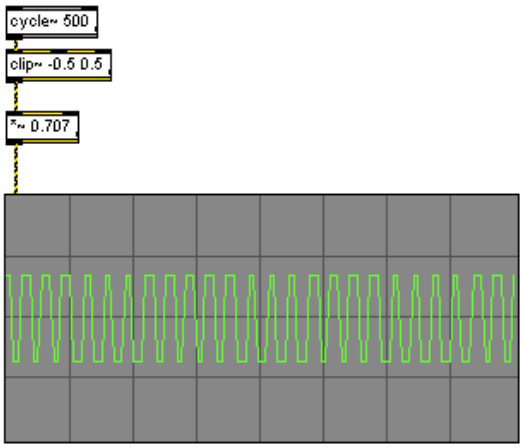
forklaring på hva en diode er i denne oppgaven, men dersom det er interessant for leseren vil jeg henvise til litteratur om *pn-junction,* for eksempel Bogart, Beasley og Rico (2004). En diode er resultatet av en *pn-junction*: positiv/negativ sammenkopling. Her nøyer jeg med meg å beskrive kretsen i figur 4.7. Som det fremgår av figuren er det to dioder, merket D2 og D3, som er koplet mellom en operasjonsforsterkers utgang og dens negative inngang. Diodene sørger for at spenningen som kommer ut av operasjonsforsterkeren ikke blir høyere enn en bestemt voltverdi. Grunnen til at det er to dioder, og at de står hver sin vei, er at det er nødvendig at en vreng klipper både den positive og den negative delen av lydsignalet: Strøm kan gå bare en vei gjennom en diode (Bogart, Beasley og Rico, 2004).



Figur 4.7: Elektronisk krets med diodeklipping. (Satt opp i National Instruments Multisim 9)[32]

Det er ingen fasit på hvordan en vreng skal bygges, men hensikten med å vise en analog krets her, er at diodeklipping er en oversiktlig krets som det er forholdsvis lett å modellere i *Max/MSP*. En *Max/MSP-patch* som gjør samme operasjon som den analoge kretsen i figur 4.7, er vist i figur 4.8. Figur 4.7 og 4.8 viser vreng i sin enkleste form, og selv om klippingen er tatt hånd om på en god måte både i det analoge og det digitale eksempelet, kreves det litt ekstra elektronikk eller programkode for at det skal låte bra når man kjører en et lydsignal gjennom kretsen/ programmet.

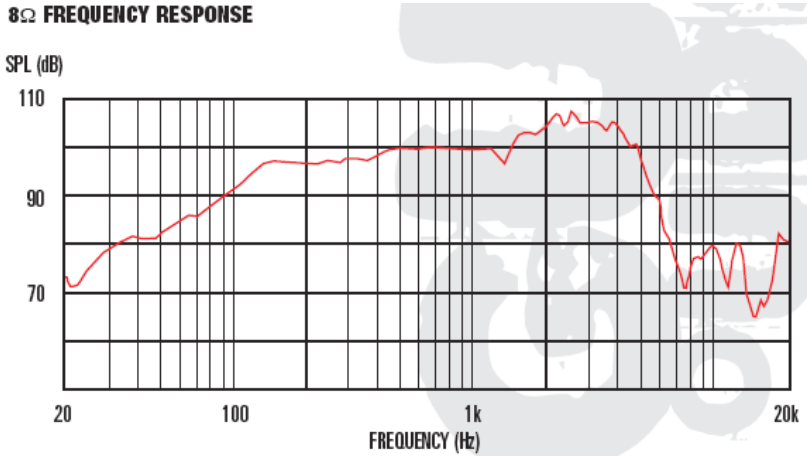
32 http://www.ni.com/academic/multisim/



Figur 4.8: Figuren viser en Max/MSP klippefunksjon.

Design av analoge effektbokser er et langt lerret, og det er for omfattende i denne sammenheng å omtale både en digital og en analog utgave. Siden utfordringen her har vært å lage en digital vreng, er det programvareversjonen som blir gjennomgått her. Den viktigste faktoren ved digital vreng, er at den har en frekvensrespons som samsvarer med det man er vant til fra signalet som skal forvrenges, som i dette tilfellet er en el.gitar. Det er ikke interessant hvilke frekvenser som kommer ut av gitaren, men hvilke frekvenser en høyttaler for elektriske gitarforsterkere er designet for å gjengi som setter premissene for frekvensområdet det klipte signalet bør ligge innenfor.

Forsterkeren som er brukt i denne forbindelse er en *Bogner Ecstacy 101B* med *Celestion Vintage30* høyttalerelementer som har en frekvensrespons som angitt i figur 4.9. Frekvensrespons er en kritisk del av en vellykket vreng, og siden vreng for alvor ble en musikalsk effekt via elektriske gitarer, synes jeg det er fornuftig å hente informasjon fra en av de mest brukte høyttalerne i gitarforsterkere på 1960-70 tallet. Høyttalere laget for å gjengi gitarlyd har en begrenset frekvensrespons, ettersom det ikke er pent å høre på de høye frekvensene fra en el.gitar. Det er ikke lett å vise dette med målinger og grafiske plot, men kan lett erfares ved å plugge en el.gitar inn i et vanlig stereoanlegg. Jeg tror det råder en konsensus om at min påstand er riktig, ettersom de aller fleste gitarister kjøper nettopp gitarforsterkere.

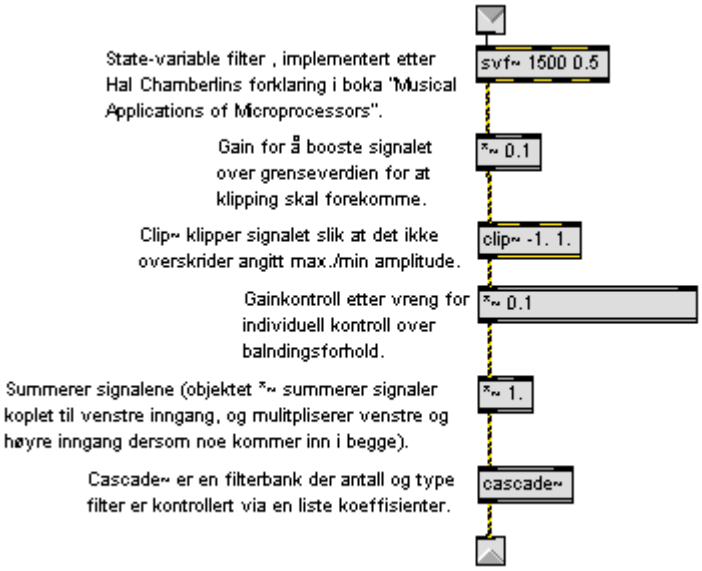


Figur 4.9: Figuren viser frekvensresponsen til høyttaleren Celestion Vintage 30.[33]

I databladet til *Celestion Vintage 30* finner vi at frekvensresponsen er oppgitt til 70-5000Hz[34], som kan verifiseres fra grafen i figur 4.9, og dersom frekvensresponsen til en digital vreng ikke blir tilpasset dette spekteret vil det rett og slett ikke låte bra gjennom høyttaleren. Den logiske løsningen på dette problemet er et lavpassfilter før utgangen. Ved å lese skjema for et hvilket som helst utvalg vrengbokser for gitar, ser en at det ofte finnes et lavpassfilter før utgangen som runder av signalet for å ta bort de overtonene som er høyere enn hva designeren synes er pent å høre på. En annen grunn til at lavpassfilteret er viktig er at en operasjons-forsterker, som er en mye brukt komponent i alle slags elektroniske kretser, ofte har lineær frekvensrespons til langt over audiobåndbredden, og at signalet derfor må vektes slik at høye overtoner som produseres av klippingen ikke dominerer det vrengte signalet. Vreng som lages ved å klippe et signal kalles for harmonisk vreng (Rossing, 2002).

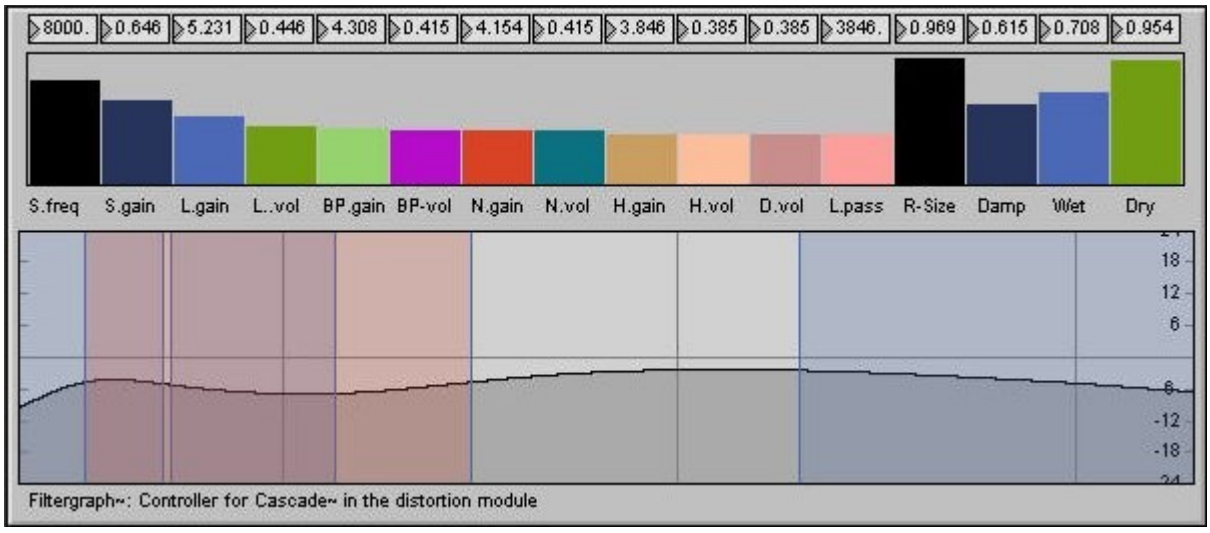
Utviklingen av den digitale vrengmodulen er basert på et ønske om å kunne kontrollere mengden av vreng på flere frekvensområder samtidig. Dette er realisert ved å bruke *Max/ MSP* objektet *svf~,* som er en implementering av Chamberlins *state-variable* filteralgoritme[35], som gjør både lavpass, båndpass, høypass og båndstop tilgjengelig på en gang. *Svf~* er veldig godt egnet til dette formålet fordi det tillater kontroll over splittfrekvens (i dette tilfellet frekvensen som markerer skillet mellom lavpass og høypass) og q-faktor.

33 http://professional.celestion.com/guitar/products/classic/detail.asp?ID=4
34 Celestion, http://professional.celestion.com/guitar/products/pdfs/Vintage%2030.pdf
35 Hal Chamberlin lagde en digital ekvivalent til et analogt state-variable filter.



Figur 4.10: MSP-objektet Svf~ med en av totalt fire klippblokker.

Figur 4.10 viser et utdrag av vrengmodulen, og vi ser objektet *svf~* koplet til *clip~* via *~ på filterets lavpassutgang, og filterbanken *cascade~* som står sist i signalkjeden. I mange tilfeller er vrengeffekter enkle enheter med noen få kontrollere, men implementasjonen i denne oppgaven har ganske mange parametre ettersom den tillater individuell kontroll over fire frekvensbånd. Figur 4.11 viser brukergrensesnittet for vrengmodulen i denne oppgaven.



Figur 4.11: Oversikt over parametrene som er tilgjengelig via brukergrensesnittet til vrengmodulen i denne oppgaven.

4.5 **Granulering av samplet lyd**

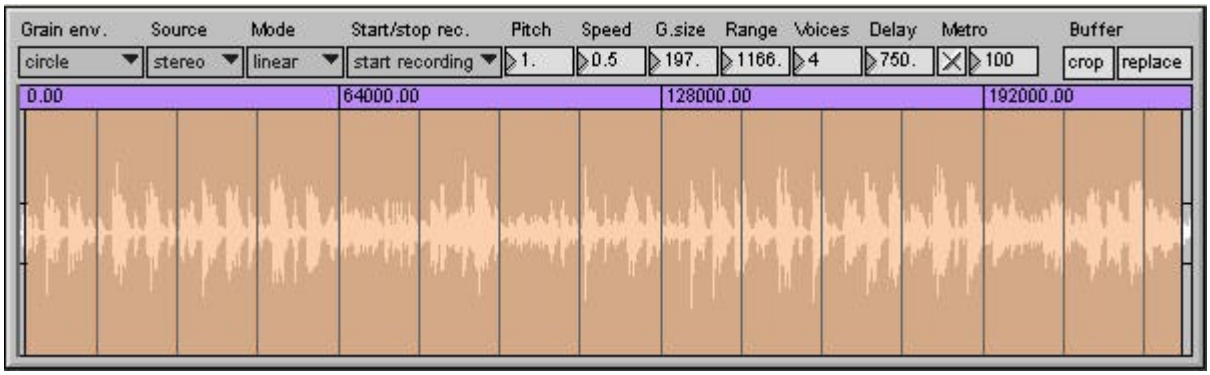
Granulering er hovedkomponenten i lydbehandlingen i dette prosjektet, og siden granulering kan brukes til å fremstille et mangfold av lyder vil jeg si litt om hvorfor jeg har lagt vekt på granulering i denne oppgaven. Min fascinasjon for granulering begynte med Rasmus Ekmans program *Granulab* (http://hem.passagen.se/rasmuse/Granny.htm), og mange timers eksperimentering med ulike lyder. Jeg husker godt da jeg begynte å eksperimentere med granulering, og opplevelsen av å laste inn en kjent lydfil, spille den av og endre innstillinger suksessivt til originallyden ble ugjenkjennelig. Dette kan selvsagt gjøres på mange måter, men granulering åpner for en kontrollert inn- og utfasing av originallyden slik at man på den måten kan beholde originalen som et utgangspunkt det er mulig å returnere til når som helst. Jeg har aldri vært spesielt glad i korte, harde *"glitschlyder"*, så i denne oppgaven er det de lange, massive, styggvakre lydene som er hovedfokus. I den grad korte grains blir brukt er det for å fremstille lydtepper i små fragment av gangen. Kort oppsummert kan jeg kanskje si at jeg har en hang til å bruke granulering på grunn av teknikkens mulighet for kontrollert nedbryting eller oppbygging av lyder. Implementasjonen av granulering i denne oppgaven er basert på Curtis Roads modell som er beskrevet i hans bok *Microsound* (2001):

- *Selection order-from the input stream: sequential (left to right), quasi-sequential, random (unordered)*
- *Pitch transposition of the grains*
- *Amplitude of the grains*
- *Spatial position of the grains*
- *Spatial trajectory of the grains (effective only on large grains)*
- *Grain duration*
- *Grain density-number pr second*
- *Grain envelope shape*
- *Temporal pattern-synchronus or asynchronus*
- *Signal processing effects applied on a grain-by-grain basis – filter, reverberators, etc.*

(Roads, 2001)

Granuleringen som følger med dette prosjektet, inkluderer ikke spatialiseringsteknikker på grunn av at de aller fleste instrument som fanges opp av en mikrofon ikke inneholder stereoinformasjon i utgangspunktet. Det er imidlertid ikke noen grunn til å utelukke spatialisering i det auditive resultatet, men i denne omgang har jeg valgt å ikke ta det med.

Programvaren for granulering er utviklet for å støtte både random og lineær avspilling av samplet lyd, ettersom avspillingsmetodene har et ulikt auditivt potensiale: Random avspilling reorganiserer den samplede lyden, og kan dermed brukes som en radikal lydgenerator basert på det signalet som til en hver tid kommer inn i programmet. En random avspillingsrutine kan endre lydens temporale mønster, og være med på å bygge opp nye lyder via sampling av et akustisk instrument. Lineær avspilling reorganiserer ikke lyden på samme måte som en random avspilling, men er en effektiv metode for å strekke og endre tonehøyden til den samplede lyden. Dette er en teknikk jeg bruker for å kunne simulere bruk av sustainpedal, og for å endre tonehøyde. Jeg bruker disse teknikkene om hverandre alt etter om jeg ønsker å endre lydens temporale mønster eller ikke. Figur 4.12 viser et eksempel på et grafisk bruker-grensesnitt for en granuleringspatch, og kjernen i denne patchen er abstraksjonen *av.grain* som gjennomgås i figur 4.14.



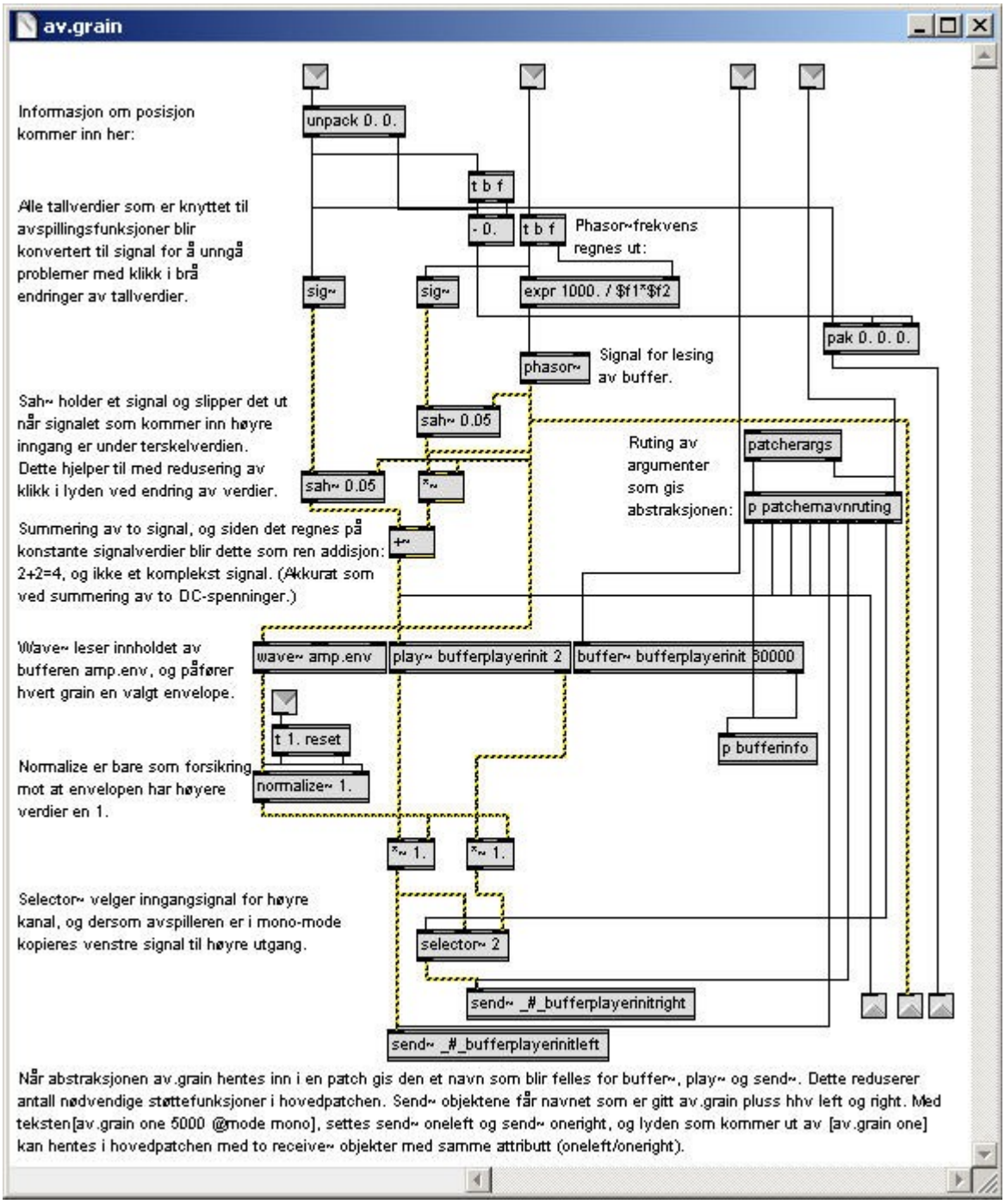
Figur 4.12: Max/MSP granuleringspatch med mulighet for flere stemmer pluss forsinkelse.

Navn	Forklaring
Grain env.	Menyen lar brukeren velge mellom tre ulike grainenveloper: *Gauss*, *Cosine Taper* og *Circle*. *Circle* er egentlig en sinus funksjon, men siden den ser ut som en del av en sirkel kalles den *Circle*. *Cosine Taper* kalles også *Quasi Gauss*, og brukes for å maksimere lengden på grainets tid med maks styrke. Gauss oppfatter jeg som mer perkusiv, og ikke så godt egnet til overlapping. *Circle* er altså en mellomting av *Gauss* og *Cosine Taper*.
Source	Velger om lydkilden er stereo eller mono. Dersom den er mono blir kanal en kopiert til kanal to.
Mode	Velger avspillingsmetode: Lineær, random eller lineær synkronisert med opptaksposisjon.
Start/stop rec.	Denne setter i gang eller stopper sampling av lyd til buffer.

Pitch	*Pitch* justerer hastigheten på avspillingen av grainene, og dersom man velger random avspilling fungerer denne kontrollen som en variabel avspillings-hastighet. I lineær avspilling fungerer denne som en pitchskifter.
Speed	Speed styrer hvor fort man leser gjennom bufferen, og fungerer bare ved lineær avspilling. I praksis strekkes eller forkortes lydens varighet, men pitch er uendret.
G.size	*G.size* setter størrelsen på grainene i millisekunder (ms)
Range	*Range* bestemmer området det skal lese fra i bufferen. I *random* avspilling velges grainenes posisjon vilkårlig, og i lineær avspilling setter *Range* start og stop-posisjon, og lesingen går i runder.
Voices	*Voices* bestemmer hvor mange grain som overlappes (maksimalt 32).
Delay	*Delay* angir lengste delaytid i millisekunder for syv forsinkelser. Tiden som settes her deles på 1, 3, 5, 7, 9, 11 og 13 (oddetallene er valgt for å minske risikoen for at to forsinkelser skal synkroniseres: feks dersom det deles på 2 og 4 vil de kunne gå i loop i forholdet 1:2, og det vil ikke hjelpe til å med å øke tettheten i lyden).
Metro	*Metro* setter tiden mellom hver gang det produseres et nytt randomtall for avspillingens startposisjon (gjelder bare i random avspilling).
Crop	*Crop* sletter innholdet i buffer som ikke er markert i bølgeformdisplayet.
Replace	*Replace* åpner en dialogboks som lar brukeren laste inn en ny lyd dersom det arbeides med lagrede lyder, og ikke sanntid sampling. Bufferen endrer størrelse så den er akkurat like stor som lydfila som blir lastet inn.

Figur 4.13: Tabellen viser funksjonaliteten til patchen i figur 4.12

Figur 4.13 gir et overblikk over hvilke parametre som trenger oppmerksomhet i granuleringen fra figur 4.12, og det er abstraksjonen *av.grain* som tar i mot informasjon om grainstørrelse, posisjon og hastighet på avspilling av hvert grain. Når avspillingen skjer ved random utvelgelse av posisjon, blir startposisjon bestemt av randomfunksjonen som velger nummer mellom 0 og 1000000, som deretter er skalert ned til verdier mellom valgt startposisjon og stopposisjon: $Pos_{grenseverdier} = startpos \rightarrow (stoppos - grainstørrelse)$.



Figur 4.14: Bildet viser abstraksjonen av.grain med forklaringer.

Jeg er fornøyd med utviklingen av granuleringen, og det fungerer som jeg ønsket. De øvrige DSP-funksjonen som er gjennomgått i dette kapittelet, er nødvendige støttefunksjoner for at granuleringen skulle behandle gitarlyden etter mine ønsker. Jeg har lenge hatt lyst på et system som lar meg bygge opp lydtepper bare ved å spille, og ikke måtte håndtere mange knapper for å få det til. DSP-utviklingen til denne oppgaven har hjulpet meg et godt stykke på veien. Neste kapittel handler om mapping, og foreslår metoder for å styre programvaren.

5. Mapping

Dette kapittelet tar for seg mapping i en musikalsk kontekst, og forklarer tilnærmingsmetoden brukt i denne oppgaven, samt en gjennomgang av en mulig teknisk løsning for å få informasjon om interaksjonen mellom utøver og instrument. Se cd for programvare.

5.1 **Tilnærmingsmetode**

Generelt sett er mapping alle koblinger mellom to enheter der koblingen er essensiell for funksjonaliteten til den ene eller begge enhetene. Mapping som kategori har et tilsynelatende uendelig omfang, og både profesjonelt og privat er mapping en del av hverdagen enten det handler om underholdning eller industri. En hvilken som helst musikalsk kontekst vil passe inn i denne generelle definisjonen av mapping, men musikalsk interaksjon setter ekstra store krav til at mappingens respons oppleves intuitiv i forhold til den funksjonaliteten mappingen kontrollerer. Så langt jeg kan se er det to nivåer en mapping kan ligge på. Den ene er som direkte link mellom interaksjon og lyd, med andre ord i de tilfeller der sensordata representerer et fysisk fenomen som er direkte mappet til en lydgenerator. Det andre nivået er de tilfeller hvor mapping er et redskap som utvider eller forbedrer et eksisterende, eller et nytt fysisk grensesnitt. Nivået mappingen skal fungere på bestemmer kravene til kvaliteten på dataene. I de tilfeller hvor mappingen er en direkte link mellom bevegelse og lyd, som for eksempel i koblingen lunger via munn til fløyte, stilles det større krav til dataene enn i de tilfeller hvor mappingen grovt sett tjener som styresignal som ikke er direkte knyttet til lydproduserende funksjoner. Eksempel på sistnevnte kan være de tilfeller hvor man ser etter en generell tendens, og resultatene brukes som en del av et større styresystem. Dette kan sees i sammenheng med komposisjon med databaserte instrumenter, og de krav dette stiller til komponisten:

> *"Contrary to the situation with conventional instruments, with the computer the composer himself is solely responsible for the sound. He has no conductor to interpret his composition. He himself must give careful consideration to even such a simple matter as the relative loudness of the instruments in a group."*
>
> (Mathews, 1963)

Da jeg begynte arbeidet med denne oppgaven var et av delmålene å finne en generell bruk av mappingteknikker for kontroll av fysiske modeller for syntese. Rovan med flere (2000)

skriver at i en fysisk modell er multidimensjonale aspekter ved interaksjonen integrert i modellen, i og med at en modell av et instrument i utgangspunktet krever samme interaksjon med brukeren som et akustisk instrument. Ettersom mitt problem er å kontrollere programvare samtidig som jeg spiller et akustisk instrument, flyttet jeg fokus fra fysiske modeller til parametre i utvalgte DSP-funksjoner.

Siden mappingen i denne oppgaven tar for seg metoder for å kontrollere datamaskinbaserte DSP-funksjoner samtidig som man spiller, er det nødvendig å ha en generell innfallsvinkel til tematikken for at det skal ha bruksverdi for ulike instrumentgrupper. For å ha en ramme rundt arbeidet med mapping, har jeg tatt utgangspunkt i tre ideer formulert i artikkelen *Control parameters for musical instruments: a foundation for new mappings of gesture to sound* (Levitin, McAdams og Adams, 2002). I artikkelen henvises det videre til andre kilder for de følgende tre punkter:

1. *Suitable mappings must be found between a musician's gesture and the control of various aspects of a musical tone.*
2. *Gestures are motions of the body that contain information.*
3. *Mappings are best when they are intuitive, and when they afford the maximum degree of expression with minimal cognitive load.*

Spesielt punkt nummer tre har vært viktig for min del ettersom denne oppgaven handler om å integrere datamaskin og et akustisk instrument til en enhet. For å ikke gå i fella og lage et kognitivt uhåndterlig system, har jeg valgt å ikke bruke sensorer som statiske brytere, men heller forsøke å finne kontinuerlige sensorer som hele tiden gir informasjon om endringer i bevegelsesmønster. Et problem man støter på med kontinuerlige sensorer er at de er ubrukelig til å sette faste verdier. Imidlertid tror jeg at behovet for å definere diskrete posisjoner er overdrevet på grunn av at mange programmerbare grensesnitt historisk sett har brukt sensorer som blir værende i den posisjonen man setter dem i. Jeg regner ikke musikk for å være en eksakt vitenskap, og finner ikke en logisk sammenheng mellom behovet for å definere innstillinger med desimaltall, og den gode opplevelsen av variasjon i to fremføringer. Det er ikke ofte jeg møter mennesker som liker improvisasjon, som begeistres over at musikerne improviserte helt likt som på forrige konsert. Etter mitt skjønn handler utvikling av nye musikalske grensesnitt like mye om å redefinere utøveres holdninger til grensesnittet, som det handler om innovasjon i forhold til bruk av sensorer. Når det gjelder kognitiv belastning tror

jeg ikke at en kontinuerlig måling av bevegelser vil innføre en større kompleksitet enn hva er tilfelle med brytere. Uansett så synes jeg det virker logisk at naturlige gester er bedre egnet til å endre innstillinger i programvare, enn endring via brytere som i tillegg må svitsjes med ganske nøyaktig timing. Jeg er heller ikke sikker på at det er så farlig å øke kompleksiteten for utøveren så lenge det gjøres med integrasjon av nye kvantifiseringspunkt: *"Modern neurological evidence also points to the plasticity of the brain: changes in cortical organization occur with changes in use or type of sensory stimulation."*(Cook, 1999 s233). Ettersom hjernen har denne evnen til å venne seg til nye dimensjoner ved et grensesnitt, er det fristende å strekke strikken, og mappingteknikkene som presenteres i denne oppgaven setter ingen grenser for hvor langt man kan gå. Formålet er å se hvordan sensordata kan skaleres og manipuleres via generelle teknikker for å utføre spesifikke oppgaver i henhold til brukerens ønsker, ikke å begrense mulighetene for individuell tilpasning.

I denne oppgaven er det lagt vekt på å ikke gjøre radikale endringer av instrumentets brukergrensesnitt, men å fokusere på hvordan fysisk interaksjon som allerede er tilstede mellom utøver og instrument, samt rytmiske og melodiske valg, kan integreres som kontrollere for programvare. Til sammen danner fire delmål bakgrunnen for mappingteknikkene som er brukt i denne oppgaven:

1. Et eksempel på dynamisk mapping av multiskalerte datastrømmer.
   Med dynamisk mapping mener jeg bruk av matematiske funksjoner for å gi datastrømmene fra sensorene en dynamisk karakter: Altså en bane som ikke er lineær.
   Med multiskalerte datastrømmer menes at et signal fra en sensor splittes til flere styresignaler som kan behandles individuelt.
2. Melodisk og rytmisk segmentering som markører for parameterstyring.
3. Bruke tonehøyde og tonestyrkeanalyse for styring av parameter vi finner i tradisjonelle effekter som for eksempel forsinkelse, romklang, vreng, lydstyrke, og så videre.
4. Utstyre instrumentets brukergrensesnitt med sensorer som gir informasjon om hvordan instrumentet brukes.

5.2 Kartlegging av dimensjonene i alminnelig gitarspill

Innfallsvinkelen til en eventuell utvidelse av et instruments grensesnitt bør gjøres på instrumentets premisser. Fra egne erfaringer vet jeg at fysisk interaksjon er sensitiv i forhold til

endringer i instrumentets fysikk. Ettersom jeg tar utgangspunkt i at hjernen klarer å venne seg til endringer i sensorisk stimulering, ser jeg det som fornuftig å gjøre disse endringene så små som mulig, og i samsvar med hvordan instrumentet brukes gjennom hva vi kan kalle alminnelig spill: "*The control organs consist of six strings and a fingerboard on the neck of the instrument. Each string is tuned using its tuning peg. The performer controls the six strings by plucking, usually with the right hand, and by shortening the strings using the left hand"* (Kvifte, 1988).

Dimensjonene i teknikker for gitarspill er varierte, og med el.gitarer og forsterkere åpner det seg muligheter for å fremheve nyanser som ved et akustisk instrument ikke er hørbare for tilhørere som sitter langt unna utøveren. Et eksempel på dette er kunstige overtoner, gnikking på strengene og noen flageoletter. Imidlertid er det fellestrekk som er knyttet til kontrollert produksjon av skalatoner, og Kvifte (1988) har summert dette i fem punkter, se figur 5.1, som dekker disse fellestrekkene.

Digital pitch (forstås som en diskret verdi)	Kontrollert av valg av streng, og fingrenes posisjon langs gripebrettet (venstre eller høyre hånd defineres av utøverens preferanse).
Analog pitch (forstås som en verdi som kan endres kontinuerlig)	Kontrolleres for eksempel ved å trekke eller skyve strengen sidelengs i kontaktpunktet mellom finger og gripebrett.
Analog loudness	Kontrolleres av hvor hardt strengen blir plukket.
Digital tonefarge	Kontrolleres av kvaliteten i anslaget.
Analog tonefarge	Kontrolleres av plukkeposisjon langs strengen.

Figur 5.1: Oversikt over kontrollert produksjon av skalatoner på gitar i henhold til Kvifte (1988).

Av de fem dimensjonene som er listet i tabellen i figur 5.1, er det åpenbart at det viktigste interaksjonspunktet mellom utøver og gitar er fingrenes kontakt med strengene. En katalogisering av instrumentets deler vil gjøre det klart hvilke andre interaksjonspunkt som er tilgjengelige for kvantifisering. Med kvantifisering menes her enhver bruk av sensorer som kan si noe om kontakten mellom utøver og instrument i et valgt interaksjonspunkt. Det presiseres at siden det forekommer store variasjoner i el.gitardesign, vil en liste over instrumentets deler ikke være fullstendig for alle el.gitarer. Gitaren som er brukt i denne oppgaven

er bygget over samme lest som en *Fender Stratocaster*[36], og derfor kan delelisten regnes som representativ for en stor gruppe el.gitarer. Et annet viktig moment er at kategoriseringen av instrumentets interaksjonspunkt tar utgangspunkt i de delene en typisk gitar er bygget opp av, og ikke i interaksjonspunkt som er et resultat av uvaner/vaner. Tabellen i figur 5.2 eksemplifiserer hvordan interaksjonspunkt kan velges ut, samt et forslag til hva slags sensorteknikk som kan brukes for å registrere grad av interaksjon. Figur 5.3 plasserer instrumentets deler på et bilde av en elektrisk gitar i henhold til tabellen i figur 5.2 .

Instrumentets deler	***Interaksjonspunkt***	***Sensor***
1. Gripebrett	fingre	Kapasitans
2. Hals (baksiden)	håndflate + tommel	FSR (trykksensitiv resistor)
3. Kropp, topp	overarm, fingre, palm	FSR
4. Kropp, bunn	torso	FSR
5. Kropp, sarg	lår, overarm, fingre	FSR
6. Elektronikk, potmeter, bryter	fingre	FSR, tilt, akselerometer
7. Mekanikk, stemmeskruer	fingre	FSR, akselerometer
8. Mekanikk, stol/tremolo bridge	fingre, palm	Flex[37], FSR
9. Hele instrumentet	gravitasjonskraft	Akselerometer

Figur 5.2: Oversikt over tilgjengelige deler i en alminnelig elektrisk gitar.

Figur 5.3: Typisk elektrisk gitar, og delene er nummerert i forhold til tabellen i figur 5.2.

I tabellen som er vist i figur 5.2 refereres det til sensortyper som kan komme i ulike former, derfor ønsker jeg å gå kort igjennom de ulike typene jeg har nevnt i tabellen:

36 Fender Stratocaster er et registrert varemerke fra Fender Musical Instruments Corp.
37 Flex-sensor referer til sensorer som endrer motstand ved bøyning. Eksempel finnes feks i sensorhansker.

1. Kapasitans referer ikke til en spesifikk sensor, men til at det er mulig å kvantifisere endringer i kapasitans. Se kapittel 2 om *Theremin* om kapasitans og måling av endring i elektrisk feltstyrke.
2. FSR (Force Sensing Resistor) sensorer gjør akkurat det navnet sier, og rapporterer hvor hardt man trykker på den. De som er brukt i denne oppgaven kommer fra Interlink Electronics, og er formet som en rektangulær eller sirkulær flate. De er cirka 1mm høye, og flatemålet er tilgjengelig i flere størrelser (se figur 5.4 for bilde av en FSR). FSR-sensorer er svært anvendelige ettersom de kan lamineres, og dermed integreres i instrumentet.
3. Flexsensorer endrer motstand når de blir bøyd, derav navnet flex, og spenningen som man sender inn i sensoren blir skalert i henhold til hvor mye sensoren bøyes.
4. Tiltsensorer refererer til sensorer som gir data om vertikal og horisontal orientering.
5. Akselerometer kvantifiserer bevegelse, og kan brukes til å sjekke kontinuerlige endringer i tilt. Et akselerometer forholder seg til gravitasjonskraften, og dersom det er tiltet vil gravitasjonen gjøre at sensoren gir ut et tall som sier noe om hvor mye den tilter. Ved å bruke akselerometer får vi vite noe om en bevegelses retning og fart (se figur 5.5 for blide av akselerometer).

I forbindelse med menneske-maskin interaksjon som verktøy for å kontrollere ett sett parametre, er det essensielt å vite hvilken del av lyden som skal påvirkes. I følge Levitin, McAdams og Adams (2002) er det fornuftig å beskrive en lyds livsløp over tre stadier: start, midt og slutt. Ved å generalisere lydens deler etter denne modellen blir det lettere å forholde seg til hvilke hendelser som er viktige i forhold til en lydproduserende gest, og dermed hvilke moment som er kritiske for en vellykket mapping. I denne oppgaven er det midt- og slutt-stadiet til lyden som regnes som viktigst i forhold til måling av interaksjon. Startposisjon kontrolleres av anslag, og DSP-prosessene som er valgt i denne oppgaven, vil påføre lyden karakteristikker som til en viss grad kan regnes som dynamiske i forhold til lydens startfase. I en granulering av kontinuerlig samplet lyd, vil for eksempel startfasen bli påvirket av de valg som blir gjort i midt- og sluttfasen av foregående lyd. En innvending her kan være at det er en kraftig inngripen i lydens sluttfase dersom samplet lyd tar over og forlenger lyden "kunstig", men så lenge det finnes et valg for å enten la granuleringen fortsette etter at lydkilden er stille, eller at den skal være aktiv bare så lenge det registreres lyd fra kilden, vil dette ikke være et problem.

5.3 **Parametre og sensordata**

I en musikalsk kontekst vet alle som har erfaringer fra bruk av programvare til komposisjon/ fremføring at det kan være en nærmest uhåndterlig mengde parametre som trenger oppmerksomhet når en ny lyd skal stilles inn. Løsningen på dette problemet er at man utfører endringer serielt: Det er ikke mulig å bruke datamus til å gjøre flere individuelle kontinuerlige operasjoner samtidig dersom programvaren ikke støtter datamus som en multiskalert sensor, og man må derfor gjøre operasjonene etter hverandre. For å sette seg i stand til å gjøre alle prosessene i riktig rekkefølge, brukes det mye tid på øving og repetisjon. Tilbakemeldingen på at kvaliteten i gjennomføringen er god nok, gis gjennom lytting og evaluering av lyden som produseres. Det samme er det med bruken av de sensorene som settes på et instrument: de krever oppmerksomhet, og brukeren blir ikke virtuos uten øving. Imidlertid ser det ut som om det er en tendens til at nye musikalske interaksjonsformer er avhengig av umiddelbar suksess for å bli en standard på samme måte som vanlige musikkinstrument har blitt standardiserte. Dette kan kanskje ha sammenheng med det Wanderley og Depalle (2004) forklarer som en idiosynkratisk tilnærming til grensesnittdesign. Idiosynkratisk utvikling referer til noe som har blitt laget for å dekke et personlig behov uten å ta høyde for om det har noen nytte for en bredere forsamling, og jeg mener det synliggjør viktigheten av å tilnærme seg mapping som fagfelt på generelt grunnlag. Gjennom egne erfaringer har jeg funnet ut at jeg er veldig rask med å legge skylden for umusikalske lyder på sensorer og lydprogram, men dette var før det slo meg at det trengs mye øving før system og utøver har utviklet en verdifull interaksjon. Kanskje kommer dette av at man som erfaren utøver tar med seg teknikkene fra hovedinstrumentet over i nye interaksjonsformer uten å sette seg inn i hva slags interaksjon som er nødvendig for å oppnå ønsket resultat. For å ha et overkommelig prosjekt, og for å ikke gape over for mye, og dermed risikere å miste en naturlig progresjon i videre utvikling av *NIME*, har jeg valgt å fokusere på dynamikken i sensorsignalenes amplitude, og vise hva jeg har gjort for å komme frem til de mappingteknikkene jeg har brukt.

5.4 **Dynamikk**

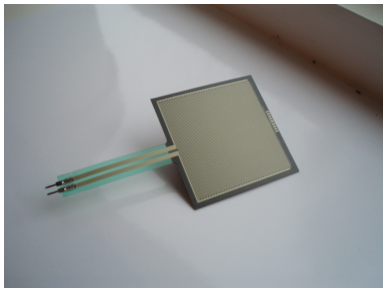
Dynamikk uttrykker forholdet mellom bevegelse og kreftene som forårsaker bevegelsen. Uavhengig av om man studerer et skips bevegelse gjennom vannet, kollisjonstester eller sammenhengen mellom lyd og bevegelse, er grunnprinsippene identiske. Et nyttig utgangspunkt for å begynne å tenke på mapping er Newtons lover som gjelder for dynamikk i fysikken. Dynamikk i musikk er stort sett tenkt på som variasjoner i lydstyrke, og selv om det

er en begrenset definisjon er det en fornuftig begynnelse. Problemene med å begrense definisjonen av dynamikk til lydstyrke kommer fort til overflaten idet man ønsker å se på interaksjon mellom utøver og instrument. Da kan vi ikke lenger bare snakke om variasjoner i lydstyrke som en separat hendelse, men hele systemet av bevegelse og interaksjon må tas i betrakting. På samme måte som at det er årsakssammenhenger som realiserer en tenkt klang-farge, er det ikke tilfeldige valg som gjør at lydstyrken endrer seg. I prosessen som skaper lyden man er ute etter, spiller bevegelser og instrumentets posisjon i forhold til utøver en betydelig rolle. De grunnleggende forutsetningene for å produsere lyd er at det blir påført kraft på et legeme som igjen setter luften rundt legemet i bevegelse, og Newtons tre lover eksemplifiserer dette:

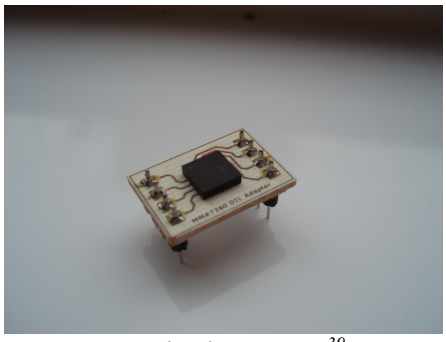
1. Når summen av kreftene på et legeme er lik 0, er legemet i ro.
2. Når summen av kreftene på et legeme ikke er like, vil legemet akselerere i henhold til formelen $\Sigma \vec{F} = m\vec{a}$ (F=kraft, m=masse, a=akselerasjon)
3. Når det virker en kraft **på** et legeme, virker det en like stor motsatt rettet kraft **fra** legemet: $\vec{F}_{A\,on\,B} = -\vec{F}_{B\,on\,A}$ (Kraft A som virker på B= - Kraft B som virker på A)

(Young og Freedman, 2004)

Sammenhengen mellom fysikken og musikken er åpenbar når vi leser Newtons lover, og sett i forhold til lydstyrke gir lov nummer to et godt bilde av virkeligheten. I et orkester vil summen av kreftene være summen av alle utøvernes kraft mot instrumentene, og oppfattet lydstyrke står i forhold til summen av de aktive kreftene. Lov nummer tre forklarer hvorfor lydstyrke øker i forhold til økt kraft på instrumentet. På denne måten blir det innlysende at lyd ikke er noe annet enn auditiv tilbakemelding som beskriver den fysiske responsen til et legeme i forhold til kraften det er utsatt for. Denne sammenhengen mellom bevegelse og kraft gir et godt utgangspunkt for å tenke seg noen mulige løsninger for kvantifisering av interaksjon mellom utøver og instrument. For meg har det mest åpenbare vært at gitaren henger inntil kroppen, berører armen og at en pianist trykker på tangentene mens vedkommende sitter på en stol med bena på gulvet. Uansett hvilket instrument man velger å fokusere på, er det en fellesnevner at alle trenger å bli påført en kraft før de kan produsere lyd. For å få ut data som beskriver kraft og retning for testing av mappingteknikker, er det to sensortyper som brukes i denne oppgaven: FSR, figur 5.4, og akselerometer, figur 5.5.



Figur 5.4: FSR[38]



Figur 5.5: Akselerometer[39]

Det er to viktige distinksjoner som må gjøres i forhold til bruken av sensorer, og det er forskjellen på signaler fra multidimensjonale og endimensjonale sensorer. En FSR-sensor responderer bare på hvor mye trykk den blir påført, og dermed vil vi ikke få noe informasjon om kvalitative aspekter ved bevegelsen som påfører trykket. Dersom vi sier at FSR-sensoren representerer et ytterpunkt, vil et tredimensjonalt akselerometer på mange måter være det andre ytterpunktet. Akselerometeret som er brukt i denne oppgaven kvantifiserer bevegelse horisontalt (x), vertikalt (y) og til/fra (z)[40] (Young og Freedman, 2004, s29). Ved å bruke akselerometer kan vi finne ut hvordan en bevegelse blir utført, og kvalitative analyser er forholdsvis lette å gjennomføre i de tilfellene det er et referansepunkt i systemet. Dersom vi holder et akselerometer i hånden og utfører bevegelser, kan vi beregne posisjon i forhold til kroppen isolert fra omgivelsene. Det ligger i navnet at vi får ut data om akselerasjon, og når vi først vet noe om akselerasjonen kan vi også beregne fart og finne posisjon. Farten finner vi ved integralet av akselerasjon i forhold til tid, og posisjon ved integralet av fart i forhold til tid (Young og Freedman, 2004, s63). I denne oppgaven er det akselerometerets posisjon i forhold til tyngdekraften som definerer endringer i programvare, og det regnes derfor ikke på fart og posisjon i forhold til forflytting fra et punkt til et annet. Verdiene fra multidimensjonale bevegelser blir altså brukt som separate enheter, og ikke som størrelser som definerer en bevegelsesbane.

5.5 **Bearbeidelse og tilordning av datastrømmer**

I et mappingsystem er bearbeidelse og tilordning av datastrømmer de viktigste punktene. Det hjelper ikke hvor gode resultater man får fra sensorer dersom tilordning og bearbeidelse

38 Force Sensing Resistor fra Interlink
39 Akselerometer fra Freescale seminconductor.
40 Siden jeg har sitert Young og Freedman 2004, forholder jeg meg til såkalt *right hand system* i det tredimensjonale planet.

svikter. Sensorer gir ofte et lineært signal som ikke stemmer med vår oppfattelse av verden, og da er det nødvendig å bearbeide datastrømmene slik at tilordningen til de funksjonene man ønsker å styre resulterer i et intuitivt brukergrensesnitt.

Uansett hvilke data man velger å fokusere på, enten det gjelder design av nye instrumenter eller kvantifisering av interaksjon mellom tradisjonelle instrumenter og utøver, er det i grove trekk tre mulige metoder:

1. Vi kan manipulere verdiene fra kvantifiseringen direkte
2. Vi kan generere nye verdier ut fra et sett regler som blir påvirket av kvantifiseringen (Wessel, 1978)
3. En kombinasjon av punkt en og to

Det enkleste eksempelet på punkt nummer en er å skalere lineære data logaritmisk. Et potensiometer kan være både lineært og logaritmisk, men bevegelsen som trengs for å nå fullt utslag fra null kan være identisk. I de tilfellene hvor styring av programvare er målet, vil det være relevant å undersøke om det er hensiktsmessig å ekstrahere mer informasjon fra en sensor enn bare tilstand. Newtons lover viser at det ikke er mulig for et objekt å endre tilstand uten at det blir påvirket av en eller flere dominerende krefter, og ved å finne endringsraten, akselerasjonen, vil vi få verdifull informasjon om kvaliteten i bevegelsen som forårsaker endringen i sensorens tilstand. Dette kan videreforedles ved å bruke informasjon om instrumentet som definerer miljøet sensoren er plassert i. En farbar vei for å finne ut noe om dette miljøet er å kvantifisere tonestyrke, tonehøyde og hastighet i spektral endring. Ved å kombinere kvantifiserte tonestyrke- og tonehøydeverdier kan det lages regelbasert interaksjon som for eksempel sier: Er tonehøyden over notenummer [n] og tonestyrke under [n] RMS så gjør X. Dersom et musikkstykke blir skrevet for et system som dette vil data fra sensorer og lydanalyse kunne brukes som en notefølger, og utføre endringer i innstillinger uten at utøveren trenger å gjøre alle endringene manuelt. Dersom det er avvik mellom intensjon og interpretasjon kan et system som dette også utvikles til å forsøke å tvinge fremføringen inn i rammer som kan defineres av komponisten, og dermed åpne for interaksjon mellom komponist og utøver uavhengig av om begge er tilstede.

Hastighet i spektral endring henspeiler på hvor ofte det observeres en endring i pitch. For å finne tonale attakk brukes det ofte transientdetektorer som sjekker et signal for ikkevarige

komponenter i frekvensspekteret. Jeg synes det er bedre å registrere endringer i tonehøyde, ettersom mange instrumentgrupper har egenskaper som fremelsker bruk av glissando, og i slike tilfeller kan man ikke alltid med sikkerhet si at årsaken til endringer i transientene er et resultat av et nytt attakk. Miller Puckettes *Pd*-objekt *fiddle~*, som også finnes for *Max/MSP*, er godt egnet til å kvantifisere endringer i tonehøyde, og sammen med *fiddle~* sin innebygde transientdetektor får vi et system som med stor grad av sikkerhet registrerer om det er et nytt attakk eller en glissando. I og med at hastigheten på spektral endring bare gir en indikasjon på den musikalske konteksten, er det hensiktsmessig å også kvantifisere amplitude, og sammen med endringsraten kan amplitudemålinger gi oss informasjon om:

1. Hvor fort man spiller.
2. Tonestyrke
3. Tonehøyde
4. Forholdet mellom attakkets amplitude og signalets RMS kan indikere om det er brukt visper, trommestikker, om det spilles med plekter/bue eller fingrer og så videre.

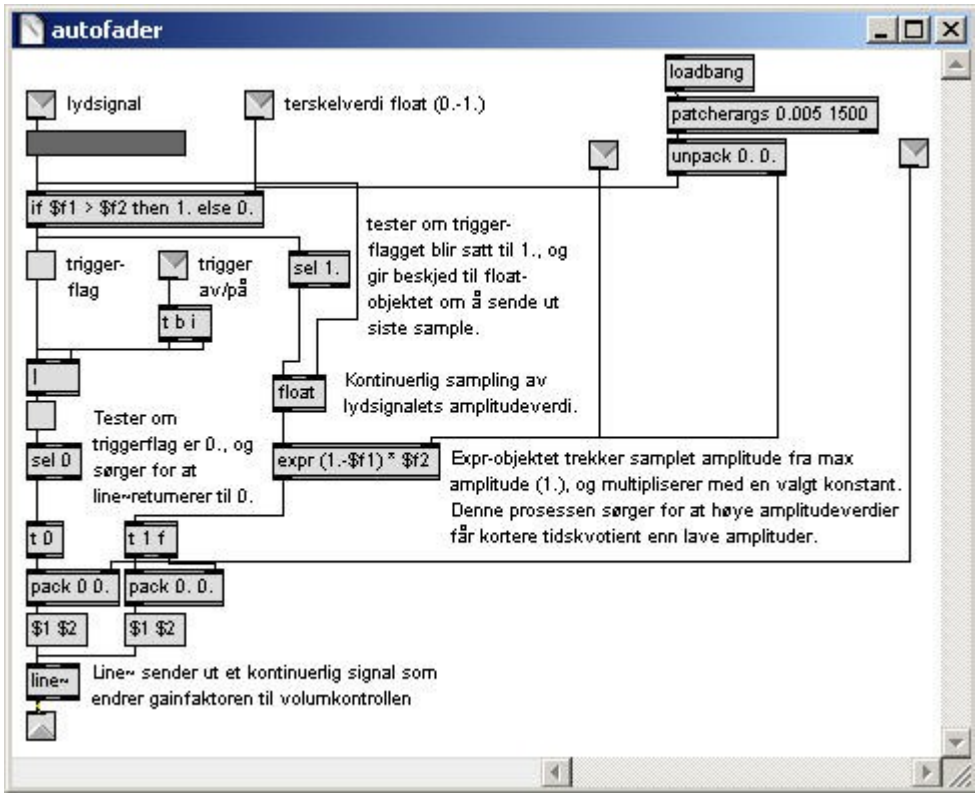
Det viktigste aspektet i forhold til hvilken mappingteknikk man velger, er selvsagt om den hever kvaliteten på interaksjonen. Se Van Nort og Wanderley (2006) for en gjennomgang av mappingens rolle i instrumentdesign.

5.6 **Dynamisk mapping**

Den siste tiden har det kommet flere musikalske grensesnitt som bruker uttrykket dynamisk mapping når det refereres til en automatikk i bindingene mellom sensorer (for eksempel potensiometer) og parametre i programvare. Dynamisk mapping burde, ettersom dynamikk er et musikalsk begrep, være et uttrykk for en kvalitet ved en mapping som hjelper det kunstneriske uttrykket, og ikke bare en automatisk funksjon som sparer utøveren for (riktignok ganske mange) knappetrykk. Bruken av uttrykket dynamisk mapping i denne oppgaven refererer, som nevnt tidligere, til karakteristikker som påføres koplingen mellom sensordata og signalprosesseringen, for på den måten å kunne endre retning, og skalering på datastrømmene som kommer fra sensorgrensesnittet.

Dynamisk mapping kan altså brukes til hva som helst, men hovedpoenget er at den bare bestemmer hvordan signalprosesseringen skal foregå, og ikke fungerer som en en-til-en

regelbasert parameterstyring. Et eksempel på dette kan være en automatisk volumkontroll hvor lydstyrke bestemmer om volumkontrollen skal være av eller på, og den dynamiske mappingen bestemmer om lyden skal fades inn logaritmisk, i henhold til en bølgeform og så videre. *Max*-patchen i figur 5.6 viser en mulig gjennomføring av en automatisk volumkontroll.

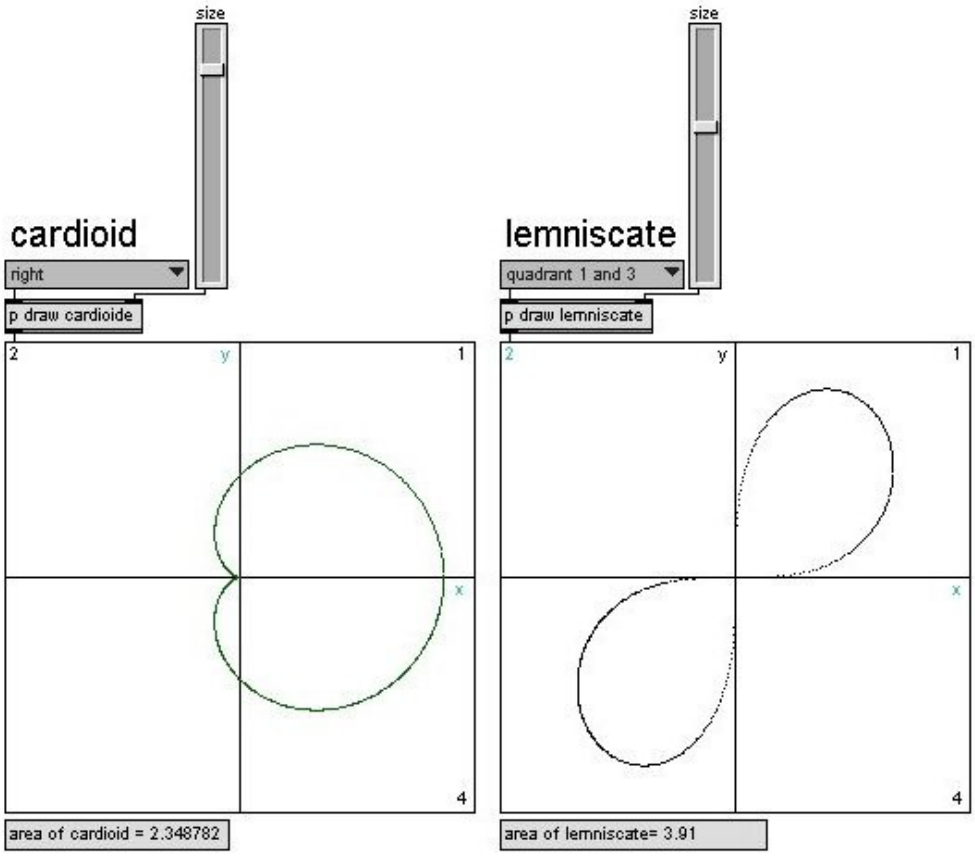


Figur 5.6: Max patch som viser en automatisk volumkontroll, hvor attakkstyrke bestemmer fadetid.

Patchen i figur 5.6 analyserer det innkommende signalet, og finner ut om det har en amplitude som er over terskelverdien for triggeren. Dersom det er sant at amplituden er over terskelverdien, blir flagget satt til 1, og *line~* får beskjed om at det skal gå fra forrige verdi til 1.0 (fullt volum) i løpet av et tidsrom som er omvendt proporsjonalt med attakkets styrke. Dette betyr i praksis at dersom man slår hardt på strengen vil volumet gå raskere til 1.0, enn dersom man slår svakt. Dette finner jeg nyttig når jeg ønsker å spille lydtepper som fades inn og blir hengende, for da blir gjerne strengene plukket av flere fingre slik at de slås an samtidig. Når jeg så veksler til å spille med plekter, eller hardere, blir ikke faden så lang. Hvis triggerflagget blir satt til 0, returnerer *line~* til 0.0 (volum satt til 0) i løpet av 50 ms.

5.7 **Gjennomføring**

For at en dynamisk mapping skal fungere, er vi avhengige av at den er lett å skjønne, og at den reagerer likt hver gang man velger en gitt skaleringsfunksjon. Dersom vi tenker oss et eksempel hvor en vandring fra venstre til høyre langs x-aksen i et kartesisk system bestemmer virkningsgraden representert ved y-aksen, kan vi lage en *patch* i *Max/MSP* som illustrerer teknikken:

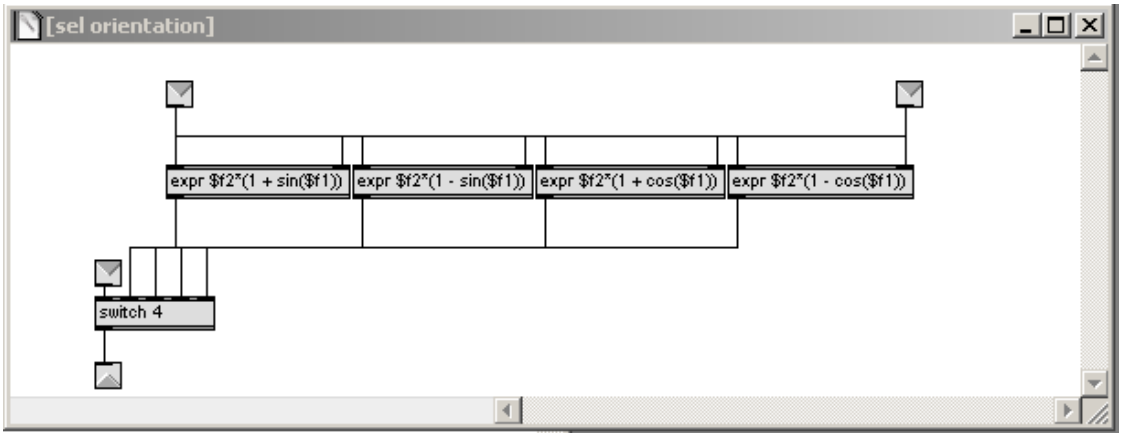


Figur 5.7: Eksempel på skalering og visualisering via matematiske funksjoner

Figur 5.7 viser to grafer som er plottet inn i systemene ved hjelp av samme sensordata, og som vi ser representerer de ulike formene forskjellig dynamikk. Holder vi fast ved at vandring langs x-aksen representerer verdiene fra sensoren og at y er virkningsgraden, er det lett å forestille seg at man bare ved å vurdere grafen visuelt kan velge riktig overføringsfunksjon for ønsket dynamikk. Det er også et viktig poeng at utøveren må kunne velge funksjoner som representerer samme virkning ved lav som ved høy påvirkning av sensoren.[41] Av figur 5.7 fremgår det også at grafene har en positiv og negativ del. Dette illustrerer at en invertering av en funksjon resulterer i at kurven snus opp ned i forhold til utgangspunktet. Signalet fra

41 Høy og lav påvirkning referer til kvantifisert spenning fra sensoren. Det er likegyldig hva slags sensor man bruker.

sensorene kan også foldes tilbake til grafens startpunkt langs sensorens vandring fra null til fullt utslag ved å bruke både den positive og den negative delen av funksjonen i samme skalering. Grafene over er bare ment som eksempel, og hvilke funksjoner som helst kan brukes i dette systemet. Det er bare tilnærmingsmåten til de matematiske funksjonene som er viktig her. Funksjonene som blir valgt til å styre et system, vil variere både som resultat av personlige preferanser, valg av interaksjonspunkt og ønsket virkning.

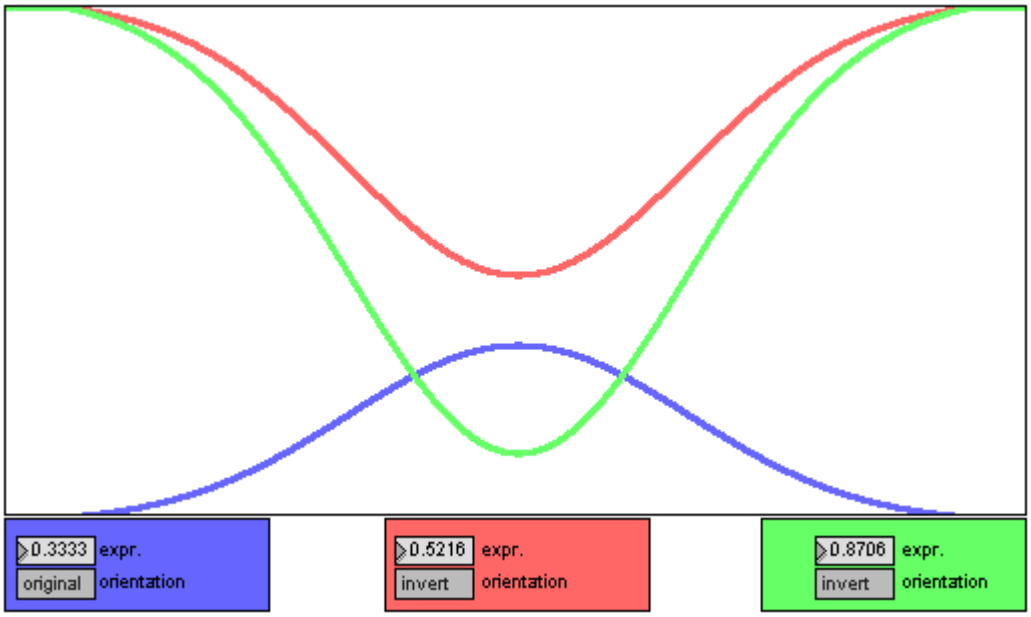


Figur 5.8: Bildet viser hvordan en funksjons graf kan roteres i et kartesisk system.

Figur 5.8 viser et eksempel på hvordan en datastrøm kan skaleres i henhold til en valgt funksjon. I dette tilfellet resulterer skaleringen i en kardiodegraf, og en av de fire formlene velges av brukeren i henhold til ønsket dynamikk. Hver formel representerer en retning på plottet i det grafiske systemet: mot venstre, høyre, opp eller ned.

Etter å ha brukt systemet lenge nok til å ha en mental oversikt over parametrene har jeg laget et praktisk eksempel på en mulig metode for styring av parametre i DSP-patchen. Det er veldig mange mulige konfigurasjoner i enhver mapping, og metodene som er vist her er bare ment som illustrasjoner til teksten. Det viktigste med dynamisk mapping i denne sammenhengen er at ved å tvinge sensordataene inn i en form (som vist i figur 5.7), enten det er en sirkel, karidoide eller normal distribusjon, som er brukt i *patchen* som produserer figur 5.9, får vi ut datastrømmer som forholder seg til et definert referansepunkt, grafens toppunkt, som kan kalkuleres kontinuerlig ut fra sensorens normalposisjon. Normalposisjon referer altså til sensorens posisjon når den ikke er påvirket av en kraft, som når en streng er i ro. Figur 5.9 viser et grafisk plot av skaleringsfunksjonene for akselerometerets tre akser, x, y og z. Grafen til en normalfordeling er lett gjenkjennelig, og grunnen til at jeg har valgt den her er at ved de to åpenbare formene, normal (blå graf) og invertert (rød og grønn graf), kan vi snu sensorenes

dynamiske respons til den retningen vi ønsker. Dette er etter mitt skjønn spesielt gunstig når det brukes sensorer som ikke har noen statiske kvaliteter utenom når de ved ytre påvirkning blir holdt i ro, for eksempel et akselerometer. Grunnen til at jeg har valgt å gjøre det på denne måten er ganske enkelt at strenger, tangenter og trommeskinn ikke flytter seg, ettersom de har en normalposisjon som utøveren må forholde seg til straks den er satt opp. En løsning som dette gjør at akselerometerets bevegelse mot aksenes ekstremtilstand (ytterpunktene som definerer maks utslag den ene eller andre veien) definerer et avvik fra normalposisjon akkurat som når en streng blir slått an og beveger seg mellom ytterpunktene som er definert av kraften i anslaget. Bevegelse mot aksenes ytterpunkt vil være et avvik enten vi påfører en kunstig normalposisjon eller ikke, men forskjellen ligger i at med et normalpunkt visualisert av en graf, blir dynamikken i systemet enklere å skjønne siden man leser dynamikken direkte fra grafens dynamiske utslag (y-aksen). Et system som dette er en samlet visuell tilbakemelding på hvordan bevegelse langs de tre aksene forholder seg til hverandre.

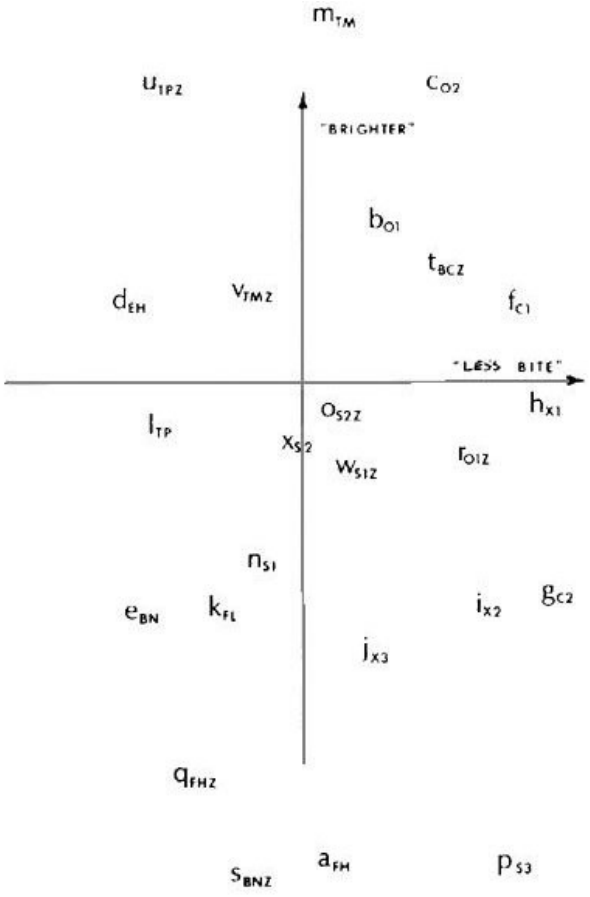


Figur 5.9: Bildet viser tre gaussgrafer, en for hver av akselerometerets akser.

Som nevnt tidligere kan hvilken som helst funksjon bli valgt, men om man ønsker å tilføre sensorene en dynamisk normalposisjon for skaleringen, er det naturlig å velge funksjoner som gir ønsket skalering på begge sider av normalposisjon. Dersom man velger funksjoner uten å ta hensyn til hva man ønsker å oppnå blir mappingen lite intuitiv og vanskelig å få tak på. Denne teknikken kan selvsagt brukes på alle sensorer, og grunnen til at jeg har valgt et akselerometer i eksempelet over, er at det returnerer fra et ytterpunkt til en normalposisjon som er relativ fordi den forholder seg til tyngdekraften. En FSR-sensor returnerer til et ytterpunkt så

fort den er sluppet, og i de tilfeller hvor vi ønsker å finne ut noe om krefter i en bevegelse i flere retninger er den ikke så gunstig. Like fullt har denne typen sensorer like stor bruksverdi i en skalering med normalposisjoner som et akselerometer, men siden det kommer tydelig frem i seksjon 5.2 at det viktigste kontaktpunktet mellom gitar og utøver er fingrenes møte med strengene, fant jeg det hensiktsmessig å bruke akselerometeret i eksempelet.

I David L. Wessels klassiker *Timbre Space as a Musical Control Structure* (1978) viser han hvordan ulike *timbrale* dimensjoner/egenskaper kan distribueres i et todimensjonalt system, og dermed gjøres tilgjengelige for et brukergrensesnitt. Dette systemet er vist i figur 5.10.



Figur 5.10: "Two-dimensional timbre space representation of 24 instrument-like sounds obtained from Grey".

Symbolene i figuren referer til timbrale egenskaper etter følgende regler: "Abbreviations for stimulus points : 01, 02 = oboes, FH = French horn, BN = bassoon, C1 = E-flat clarinet, C2 = bass clarinet, FL = flute, X1 X2, X3 =saxophones, TP = trumpet, EH = English horn, S1 = cello played sul ponticello, S2 = cello played normally, S3 = cello played muted sul tasto, FHZ = modified FH with spectral envelope, BNZ = modified BN with FH spectral envelope, S1Z = modified S1 with S2 spectral envelope, S2Z = modified S2 with S1 spectral envelope, TMZ = modified TM with TP spectral envelope, BCZ = modified C2 with 01 spectral envelope, 01Z modified 01 with C2 spectral envelope."

"The horizontal dimension is related to the quality of the "bite" in the attack." (Wessel, 1978)

Jeg synes det er nyttig å studere Wessels fremstilling fordi denne typen visualisering og plassering av dimensjoner/egenskaper kjenner ingen grenser for hva det kan brukes til. For eksempel kunne *preset* i programvare for sanntidprosessering av lyd organiseres i tilsvarende system som i figur 5.10, og gjøres tilgjengelige for brukeren via navigering med matematiske funksjoner. Dersom man setter seg inn i ulike matematiske funksjoners grafiske plot, tror jeg det vil være mer effektivt å bruke faste funksjoner for å navigere i systemet, enn å tegne udefinerte grafer for hånd. Grunnen er, som tidligere nevnt, at jeg ikke ser en idiosynkratisk utvikling av programvare som ideell. Jeg mener at enhver utvikling av mappingteknikker bør forholde seg til kjente matematiske funksjoner slik at bruksverdien for andre ivaretas.

Dersom vi tar utgangspunkt i grafen til normalfordeling som illustrert i figur 5.9, og tenker oss den som transportfunksjon for sensordata i det todimensjonale systemet, får vi en god indikasjon på om normalfordeling kan være riktig funksjon for å ha ønskede punkter i det todimensjonale systemet innenfor grafens rekkevidde. Ved å kombinere sensordata med musikalske data (tonehøyde, tonestyrke og så videre) kan det lages regler som igjen kan danne utgangspunkt for grafens plassering i systemet, og funksjonenes grafer kan roteres og flyttes langs x-aksen etter en gjennomført dynamisk modell. Dersom vi bruker et akselerometer til å kvantifisere bevegelse kan vi ved en og samme multidimensjonale bevegelse flytte markøren i et todimensjonalt system, og alle punktene er innen rekkevidde dersom reglene for grafenes orientering i planet blir laget utfra utøverens totale interaksjon med instrumentet. Se Momeni og Wessel (2003) for en tilsvarende metode basert på orientering i et tredimensjonalt landskap av geometriske modeller via et pekergrensesnitt (fortrinnsvis *Wacom* tegnebrett, eller lignende enheter som i tillegg til pekefunksjon også gir data om tilt og trykk).

Se video 5.7.1[42] for en presentasjon av hvordan kreftene som virker på akselerometeret kan skaleres ved hjelp av modellen som er presentert i denne seksjonen. Video 5.7.2 viser samme skaleringsmetode for en FSR, men hvor en datastrøm resulterer i tre skalerte signaler.

5.8 **Fuzzy regler**

Under improvisasjon med utgangspunkt i noterte musikalske retningslinjer, i betydningen notasjon som sier noe om ønsket musikalsk retning, er det ikke alltid like lett å formidle til de andre musikerne hva man har tenkt via vanlige noter. De gangene jeg har forsøkt å notere

42 Videoer finnes på vedlagt cd.

musikalske ideer til slike formål, har jeg ofte kommet til kort. Det er ikke spesielt vanskelig å notere notene og deres respektive verdier, men jeg mister kontrollen over tankene om *timbre* og timing. Heldigvis er jeg ikke alene om å ha det sånn, og i 1974 skrev Cornelius Cardew om to "sykdommer" han hadde funnet i holdninger til notasjon:

> *"So far I have identified two main diseases: first, the idea that each composition requires or deserves its own unique system of notation. Let's be more accurate: the composer doesn't conceive of a piece of music so much as a notation system, which musicians may then use as a basis for making music, or more likely (as I would evaluate it today), aimless manipulations of the system in terms of sound. Second, the idea that a musical score can have some kind of aesthetic identity of its own, quite apart from its realisation in sound, in other words that the score is a visual art work, the appreciation of which may depend on a consciousness of music and sound and the ways they have been notated, but with no certainty that the ideas of the composition can be transferred into and expressed through the world of sound. In my output I was preoccupied for several years with a large scale manifestation of this second disease, the graphic score Treatise, and it is to this work that I wish to apply some more detailed criticism."* (Cardew, 1974)

Jeg synes Cardew peker på elementer som fortsatt er både relevante og viktige i forholdet mellom standard notasjon og notasjon som er spesifikk for et stykke. Mitt syn på denne problematikken er den samme som for utvikling av programvare: Blir det spesielt for hver enkelt utøver/komponist, eller hvert enkelt stykke, reduseres nytteverdien i forhold til en felles utvikling av notasjon. Med dette som utgangspunkt har jeg forsøkt å finne andre metoder enn grafisk notasjon og laget et *logisk* system basert på, eller rettere sagt inspirert av *fuzzy sets* (Zadeh, 1965). Før jeg går igjennom dette systemet, er det hensiktsmessig å se litt nærmere på hva *Fuzzy sets* er.

Fuzzy sets ble introdusert av Lotfi Zadeh[43] i 1965 og er et system som tillater overlappende grupper av informasjonen som skal vurderes. Meningen med dette er at man da kan vurdere hvilken gruppe verdien hører mest hjemme i. Vanlige *boolske* operatorer som *AND*, *NOT* og *OR* (se for eksempel Floyd, 2005) tar ikke høyde for noe annet en veldefinerte tilstander, og dersom man trenger informasjon om mellomtilstander er det fornuftig å dele dataene inn i *fuzzy sets*. Et eksempel på dette kan være tallene mellom 0.0 og 3.0: Dersom vi skulle dele

43 Lotfi Zadeh: http://www.cs.berkeley.edu/~zadeh/

disse inn i to like store grupper ville det være naturlig å dele inn i en gruppe fra 0.0 til og med 1.5, og en gruppe fra 1.5 til og med 3.0. Vi har da to kategorier, og ofte er det tilstrekkelig, men i mange tilfeller er det ønskelig å vite noe om hvor man befinner seg innenfor hoved-kategoriene. Med *fuzzy set* kan vi for eksempel sette en kategori fra 0.0-1.75 og en fra 1.25-3.0, og tallfeste hvor mye dataene som testes hører hjemme i den ene eller andre kategorien. Nytten av å tallfeste tilhørighet via *fuzzy set* kommer til syne dersom vi tenker oss at vi sykler mot en bratt oppoverbakke. Fra tidligere erfaringer vet vi noe om hvor hardt man må tråkke på pedalene for å komme opp bakken, og valg av riktig gir gjøres med bakgrunn i disse erfaringene. Dersom bakken ikke er den bratteste man har syklet, vet vi at det antakelig er unødvendig å slite seg ut for å komme seg til toppen og vi kan spare litt krefter. Er bakken derimot det bratteste vi noensinne har sett, er det fornuftig å bruke krefter på å få opp farten før bakken begynner slik at det blir lettere å holde oppe farten helt til vi er på toppen. Dersom vi ikke registrerte at det er en oppoverbakke før hellingen har begynt, ville det selvsagt ikke være mulig å forberede seg før man allerede er på vei oppover.

Systemet jeg har laget er som sagt inspirert av *fuzzy sets*, og skiller seg ut ved at det ikke bruker en medlemskapfunksjon for å tallfeste tilhørighet. Grunnen til det er at jeg bare er interessert i mellomtilstandene som selvstendige grupper, og at jeg ønsker å fokusere på en visuell tilnærming til gruppeinndeling. Systemet er laget for at brukeren skal kunne definere gruppeinndeling ved hjelp av skalering via matematiske funksjoner, mens visuell tilbake-melding gis med funksjonenes grafer. I dette systemet kombineres *fuzzy sets* og logikk, og viser en elementær praktisk tilnærming til *fuzzy logic*: *"In a broad sense, fuzzy logic is almost synonymous with fuzzy set theory. Fuzzy set theory, as its name suggests, is basically a theory of classes with unsharp boundaries"* (Zadeh, 1994).

Jeg synes dette systemet resulterer i en enkel metode som kan brukes til å stille inn parametre basert på brukerens gruppeinndeling. I seksjon 5.9 går jeg gjennom et eksempel på bruk av dette systemet i forhold til tonehøyde. Dersom tonehøydeverdier blir mappet direkte til para-metre, kan det resultere i et rigid og umusikalsk system, og derfor synes jeg tonehøyde er godt egnet som et eksempel på hvilken nytte man kan ha av å dele inn et signal i *fuzzy* set. Et pro-blem med å bruke tonehøydeverdier som utgangspunkt for en mapping, er at verdiene kan variere veldig mye innenfor et kort tidsrom. Et annet problem er at dersom utøveren hele tiden må være oppmerksom på tonevalgets innvirkning på programvaren, vil det forstyrre den naturlige spillestilen like mye som ved bruk av eksterne kontrollenheter. Ettersom jeg startet

denne seksjonen med en henvisning til mine problemer med notasjon, kan det også innvendes at å løse problemer relatert til musikalsk notasjon med programvare stiller krav til at komponisten kan å programmere. For min del ser jeg på kunnskap om programmering som likestilt med kunnskap om musikalsk notasjon dersom programvaren er nødvendig for gjennomføringen av musikkstykket, og jeg velger derfor å ikke bekymre meg for denne problemstillingen.

5.9 Eksempel på bruk av *fuzzy logic*

For at man skal få et best mulig resultat av å analysere et signal ved hjelp av *fuzzy logic*, bør datastrømmene deles opp i størrelser som er relevante for signalet som skal analyseres. For et instrument uten overlappende tonehøyderegister, for eksempel et piano (alle tangentene representer en unik tone), kan en inndeling i overlappende oktaver være hensiktsmessig, mens instrumenter med overlappende frekvensbånd, for eksempel gitarer og strykeinstrument, kan deles inn i henhold til frekvensspekteret til hver enkelt streng.

Figur 5.11: Et standard 22-bånd gripebrett. Strengene er stemt som tabellen i figur 5.12 viser.

Streng	Laveste pitch	Høyeste pitch	*Fuzzy* set inndeling:		
E1	82,4	293,7	82,4	110,0	Unik
A	110,0	392,0	110,0	246,9	Overlapp
D	146,8	523,3	196,8	329,6	Overlapp
G	196,0	698,5	246,9	523,3	Overlapp
H	246,9	880,0	329,6	880,0	Overlapp
E2	329,6	1174,0	880,0	1174,0	Unik

Figur 5.12: Oversikt over strengenes inndeling i forhold til pitch.

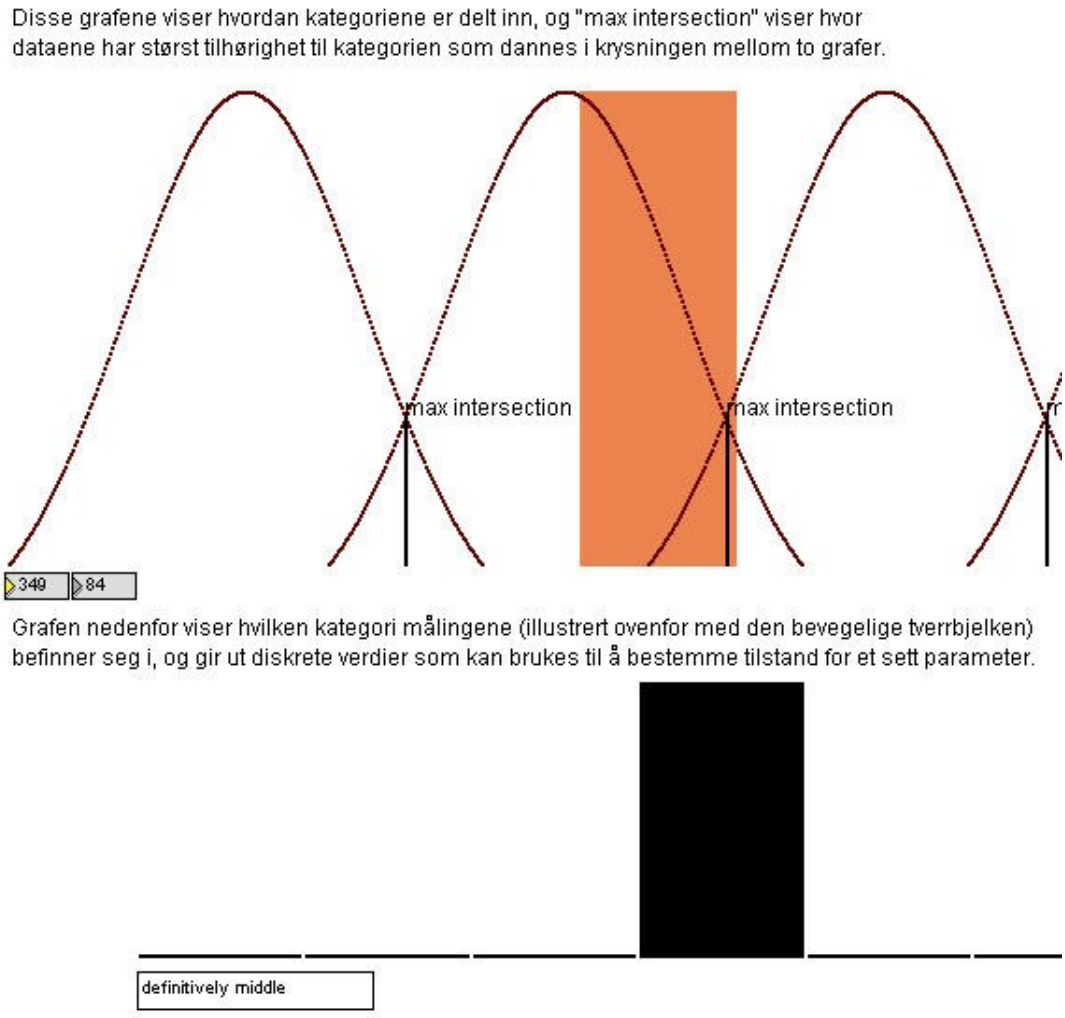
Tabell 5.12 viser hvordan en gitar med 22-bånd gripebrett, som vist i figur 5.11, vanligvis blir stemt. Fra tabellen i figur 5.12 ser vi at dersom analysen viser at tonehøyden er under 110,0 Hz eller over 880,0 Hz, kan vi si med sikkerhet at lyden blir produsert på henholdsvis E1 eller

E2. Det er riktignok en forutsetning at tonene defineres av gitarhalsens bånd, og ikke som flageoletter eller kunstige overtoner. Dette er det eneste sikre tilfellet, for alle andre toner kan komme fra minst to av de andre strengene. Derfor er det viktig å dele inn gitarens tonespekter *s*lik at man oppnår så mange *fuzzy sets* man behøver. De åpenbare inndelingene er etter strengenes frekvensområde, men ved å løse det på den måten vil man få mange toner som slekter til samme *fuzzy set*, og analyseresultatene vil være veldig generelle. Hvis vi deler strengene i to grupper, ser vi at dersom en frekvens er lavere en 196,0 Hz, må den komme fra de tre dypeste strengene, forutsatt at gitaren er stemt som i figur 5.12. På samme måte må en frekvens høyere en 523,3 Hz komme fra en av de tre øverste strengene.

Selv om det er fristende å sette opp kategorier som gir mest mulig eksakt informasjon om hvordan et instrument blir brukt, er det en fare for at systemet ikke blir *fuzzy* nok selv om inndelingene er i henhold til et *fuzzy* system. For min del synes bruksverdien av mapping-teknikker å være avhengig av at det er rom for små uregelmessigheter i interaksjonen. Dersom sensordataene blir mappet direkte til de parametrene som skal kontrolleres uten å først filtreres, blir systemet avhengig av at utøveren er ekstremt nøyaktig i gjennomføringen. Et eksempel på dette er kontroll av volum med en FSR-sensor. Dersom sensorens signal blir mappet direkte til volumkontrollen vil den minste trykkendring på sensoren føre til endring i volum. Effekten av dette er i beste fall en tremoloeffekt, og i verste fall uønsket "pumping" i lydstyrke. For å unngå dette kan vi filtrere sensordataene slik at signalet blir jevnet ut, og bare trykkendringer som varer over en viss tid får justere volumet. En innvending mot dette kan være at en filtrering vil gjøre systemet unøyaktig, men jeg mener at et system som skal respondere på interaksjon mellom utøver og instrument i mange tilfeller ikke trenger å være eksakt. Personlig finner jeg ikke en egen musikalsk verdi i å oppdage at det er minimalt avvik mellom to gjennomføringer av samme stykke, men jeg mener heller ikke å ta slurv og slendrian i forsvar. Jeg synes at små avvik som forekommer i en gjennomføring av et innstudert stykke, og på lik linje med variasjoner i improvisatoriske grep, er resultater av menneskelige faktorer som tilfører musikken kvaliteter og forlenger levetiden til stykket. Det er de samme menneskelige faktorene som gjør opplevelsen av samspill så verdifull for meg som musiker.

For å lage et *fuzzy logic* system som kategoriserer analyseresultater i henhold til noteverdier, kan det i mange tilfeller være tilstrekkelig å indikere datarekkevidden (for eksempel tone-høyderekkevidden til en gitar), antall logiske kategorier og grad av overlapping mellom

kategoriene. Med de angitte verdiene som utgangspunkt, vil da systemet klare å forvalte et sett regler ved å ganske enkelt gi informasjon om hvilken kategori det analyserte signalet til en hver tid tilhører. Figur 5.13 viser en inndeling av gitarens tonale rekkevidde inndelt i fem grupper: tre hovedgrupper og to overlappinger. I figuren er det et punkt i hver overlapping som kalles *max intersection*, og det indikerer at det er i dette punktet de kryssende hoved-gruppene har mest til felles, og feltet som ligger innenfor overlappingen er en *fuzzy* gruppe. For å finne ut om verdiene vi tester ligger i en overlapping, bruker jeg vanlige logiske operatorer: Dersom to grupper rapporteres som sanne, hører verdien til en overlapping.



Figur 5.13: Fuzzy logic inndeling. (Se video 5.9.1 for en visuell demonstrasjon av kategoriseringen.)

Sammen med *fuzzy logic* systemer som dette, kan vi lage linker til andre funksjoner/ algoritmer ved å for eksempel bruke *Markovrekker*. *Markovrekker* er en metode som er stadig mer brukt innen musikkteknologien, og ved en senere anledning vil jeg lage et system som

beskriver disse prosessene slik at det er enkelt å sette seg inn i funksjonaliteten. Grunnen til at jeg ikke har tatt med *Markovrekker* her, er at jeg forsøker å lage et skille mellom grunnleggende skaleringsfunksjoner og utnyttelsen av skaleringsresultatene. Dersom vi drar en parallell til utøving på et hovedinstrument, vil erfarne utøvere ha en rekke motoriske vaner som reduserer antall beslutninger som må tas før en handling iverksettes. På samme måte ønsker jeg å finne mappingteknikker som reduserer antall tilstander som registreres i *Markovrekker*, og som *probabilitet* beregnes ut fra. Det er ikke sikkert at det er en vellykket fremgangsmåte, men jeg tror at et system basert på *Markovrekker* og *probabilitet* i forhold til mapping er tjent med færrest mulig tilstander, og at resultatene fra et system som dette kan brukes til å utvikle nye dynamiske regler for mapping. Ettersom det tar noe tid å bli helt fortrolig med resultatene fra dette arbeidet, ønsker jeg ikke å begynne med *probabilitetsberegninger* før jeg er mer sikker på hvor det er mest fornuftig å redusere datamengden i forhold til skaleringsteknikkene.

5.10 **Metode for å skifte mellom ulike preset**

I de aller fleste digitale effekter og synthesizere er det vanlig sjargong å referere til banknummer [n], og programnummer [n]. Dette er en gunstig måte å katalogisere mange *preset* på slik at brukeren lett kan hente frem ulike innstillinger etter behov. Jeg ønsker å bringe dette videre til et system som gjør det mulig å spille et musikalsk motiv, og bruke dette motivet som en ekvivalent til et brett med knotter. Figur 5.14 viser et eksempel på et brukergrensesnitt for veksling mellom ulike programmer i et *MIDI-system*.

Figur 5.14 Standard midifotbrett for programbytte etc. Bildet er hentet fra http://www.nobels.com/en/products.htm#mc

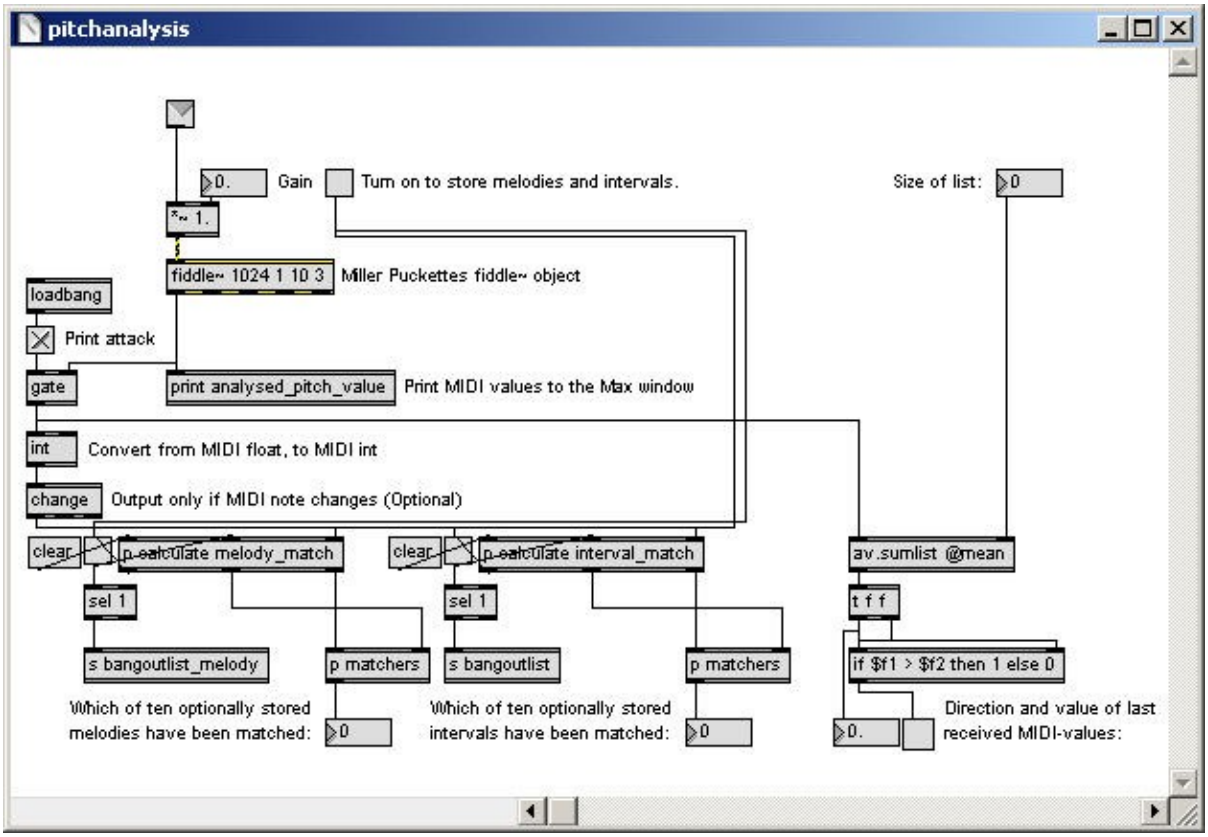
Mange musikere er helt avhengig av den typen enheter som er vist i figur 5.14, og til mange oppgaver fungerer de meget bra, men de har begrensninger i forhold til timing og posisjonering på scenen. Det har ikke vært et mål for meg å lage et system som eliminerer behovet

for vanlige kontrollere, men jeg ønsker å ha muligheten til å kommunisere med datamaskinen via musikalske kommandoer. I improvisasjon er det svært vanlig at musikerne har avtalte markører i musikken som indikerer at et skifte skal komme, og det er ingen grunn til at vi ikke skal kunne kommunisere på samme måte med datamaskiner. Det eneste som kreves er at vi lagrer små melodiske eller rytmiske motiv som programmet kan kjenne igjen. Teknikken bygger på prinsipper for segmentering, og for tiden arbeides det med å lage systemer som kan segmentere melodier i polyfonisk materiale (se for eksempel Paiva, Mendes og Cardoso, 2005). I denne oppgaven dreier segmenteringen seg om monofoniske motiv. Jeg stiller to krav til de melodiske motivene, nemlig at de skal være bygd opp av fire toner og at hvert motiv må ha minst en toneverdi som skiller det fra de andre motivene. For perkusjonsinstrument er det tilsvarende krav for rytmiske motiv: De må ha minst en rytmisk verdi som skiller de fra hverandre. I praksis betyr dette at tre tonehøyder, eller tre toneverdier kan være like:

1. a-b-c-d,
2. b-b-c-d,
3. a-b-c-a,
4. a-b-c-b.

I de fire motivene ovenfor er bokstavene a,b,c,d brukt som symboler for enten tonehøyde eller rytmisk verdi, og siden det er en variasjon i hvert av dem er de gyldige i henhold til kravene for gjenkjenning. Imidlertid synes jeg det er tryggest å unngå at like toneverdier følger etter hverandre, og i patchen i figur 5.15 har jeg lagt inn et *change* objekt som ikke slipper gjennom like tall etter hverandre.

Rytmiske og melodiske motiv kan også kombineres slik at systemet ikke kjenner igjen motivet som markør dersom det er avvik i enten rytme eller melodi. Foreløpig ser jeg det imidlertid som mest verdifullt at rytme er skilt fra melodi, slik at det bare er tonehøydeverdier som avgjør om et motiv er spilt eller ikke. Grunnen til dette valget er at kompleksiteten i et system som krever både riktig rytme og melodi for å utføre en kommando, blir veldig mye høyere enn et system som bare ser på melodi eller bare ser på rytme. I tillegg vil et melodisk motiv ha samme betydning for kontrollsystemet dersom det spilles fort eller sakte, og det legges dermed ingen føringer for timing i gjennomføringen av motivet, noe som er gunstig i en improvisasjon fristilt fra metrisk inndeling.



Figur 5.15: Max/MSP patch som viser et eksempel på en gjennomføring av melodi og intervallgjenkjenning, samt retning og verdi på sist mottatte tonehøydegruppe.

Gjenkjenningen av de musikalske motivene skjer ved analyse av alle tonehøydeverdier som kommer inn i systemet og kontroll av om de kommer i riktig rekkefølge i forhold til motivet de blir sammenlignet med. Det samme kan gjøres med rytmiske motiv ved å teste tiden mellom to anslag i stedet for tonehøydeverdier. Dersom det lagres et motiv som består av tonene 60,62,64,65 (*MIDI* noteverdier for c^1,d^1,e^1, og f^1), er det en vesentlig fare for at lignende fraser vil bli brukt flere ganger i en komposisjon. På grunn av denne risikoen er det nødvendig å legge inn et valg for når signalet analyseres for å finne motiv, og samtidig åpner det for at små melodiske motiv som finnes i komposisjonen kan brukes som triggere flere ganger bare ved å telle antall ganger et motiv har blitt gjenkjent. Dette er veldig nyttig i komposisjoner hvor melodiske motiv blir repetert mens *timbre* endres. Ved å telle motiv er det mye hjelp å få fra et system som ut fra denne informasjonen kan følge et skjema for planlagte oppgaver.

5.11 **Tonehøyde og tonestyrke**

Jeg liker godt å ha muligheten til å styre interpolering mellom *preset* via en kontinuerlig sensor, og ved å analysere lydsignalet og finne tonehøyde og tonestyrke håper jeg å kunne bruke analyseresultatene som en kontinuerlig sensor. For å finne en fornuftig løsning på disse ønskene har jeg laget et lite program, som vist til høyre inne i figur 5.15, som i dette tilfellet beregner middelverdien av en serie tonehøydeverdier, som kan brukes til å endre innstillinger gradvis i forhold til om middelverdien øker eller synker.

Hvordan man velger å bruke middelverdiens retning er selvsagt opp til brukeren, og jeg tror at en løsning som denne kan være god i de tilfeller hvor brukeren har mulighet til å definere hvilke endringer som skal utføres når middelverdien øker eller synker. Et enkelt eksempel er å lagre to *preset* og la middelverdien interpolere mellom de. Dersom man holder rede på hvor mange ganger middelverdien har økt, eller sunket, kan dette enkle systemet utvikles til et ganske avansert mappingsystem som sammen med for eksempel gjenkjenning av melodiske motiv kan endre hvilke *preset* som interpoleres, eller hvilke som helst andre oppgaver man måtte ønske å kontrollere.

Erfaringene fra testing av tonehøyde, segmentering og gjenkjenning av melodiske motiv synes jeg er positive. Dersom man ønsker å ha full kontroll over systemet, er det nødvendig å diskretisere instrumentets grensesnitt slik at man uavhengig av signalet vet hvor man trykker. Dette er en smal sak for digitale instrumenter, spesielt *MIDI*-baserte instrumenter da verdiene man trenger for analysen er definert før signalet produseres. Likevel synes jeg dette er et brukbart system for analoge instrumenter, og det er ikke noe problem å bruke gjenkjenning av melodiske motiv for å styre programvare.

5.12 Oppsummering kapittel 5

Gjennom kapittel fem har jeg forsøkt å gi et overblikk over en rekke teknikker som kan brukes til å påvirke datastrømmer fra sensorer, men listen er langt fra komplett. Konklusjonen etter å ha arbeidet med mappingteknikkene som er presentert i denne oppgaven, er at primærmålet for utvikling av mappingteknikker bør være å finne generelle metoder som kan brukes av en stor gruppe brukere. For å være noenlunde sikre på at vi utvikler generelle mappingteknikker er jeg trygg på at det er lurt å ta utgangspunkt i fysikken som relaterer seg til de endringene vi ønsker å utføre: Det er ikke naturlig å sparke på et potensiometer dersom det er

et ønske å ha kontroll over endringen. Så lenge de fysiske betingelsene, altså hvilke bevegelser som trengs for å utføre endringer, blir tatt med i planleggingen av programvare-designet tror jeg det er gode sjanser for å finne metoder for mapping som gjør interaksjon med datamaskiner enklere. De av oss som spiller akustiske instrumenter og som samtidig ønsker å bruke datamaskiner, vil ha stor nytte av alle mappingteknikker som lar oss styre programvare uten å ta hendene/føttene bort fra hovedinstrumentet. Personlig har jeg hatt stor glede av å arbeide med de ulike teknikkene som er presentert i kapittel fem, og vil ta utgangspunkt i det som er gjort her for videre arbeid med mapping. Imidlertid er det nødvendig å få mer praktisk erfaring ved bruk av disse teknikkene for å finne ut hva som fungerer og ikke fungerer over tid. I tillegg kan metodene, som er gjennomgått i denne oppgaven, settes sammen på mange ulike måter, og jeg anser det som nødvendig å eksperimentere med hvordan de bør settes sammen før jeg går videre. Som nevnt tidligere er *Markovrekker* og *probabilitet* interessante temaer for å utvide mappingsystemets funksjonalitet, og sammen med bruk av sensorer og analyse av lyd bør det være gode muligheter for å komme nærmere et generelt system, som kan brukes uavhengig av hvilket instrument/musikkstil man spiller. I mange tilfeller har jeg behov for at innstillinger i programvare følger lydsignalet som sendes inn i programmet, og etter å ha testet teknikken med gjenkjenning av melodiske motiv tror jeg at det kan utvikles til en god løsning for gjennomføring av stykker som forholder seg til noterte melodier, eller improvisasjoner hvor melodiske motiv kan brukes som en triggere. Det understrekes at alle mappingteknikkene som er gjennomgått i denne oppgaven strever etter å være generelle, og samtidig la det være opp til brukeren å definere hva de skal kontrollere.

6. Konklusjoner og tanker om videre arbeid

6.1 **Konklusjon**

Den viktigste erfaringen jeg tar med meg fra dette arbeidet, er at det er en sammenheng mellom hva man ønsker å høre og hva man velger å gjøre. Dette ser åpenbart banalt ut, men dersom det settes i sammenheng med noe av det vi vet om melodiske preferanser, er det lettere å forestille seg hvorfor det er nyttig å skrelle av lagene så vi står igjen med koblingene mellom motoriske forutsetninger og det auditive. Bregman (1990, s 462) sier at små intervall ser ut til å være foretrukket av mennesker rundt om på hele kloden, og mener grunnen er at det er lettere å synge små enn store intervaller. Videre sier han at de små intervallene kan spille en rolle i å binde melodien sammen slik at vi oppfatter den som en enhetlig strøm av toner. Denne tenkingen har etterhvert dannet grunnlaget for hvordan jeg tenker om organisering og valg av sensorer og mappingteknikker til programvare for sanntidbehandling av *timbre* og *tekstur*. Hvis vi ønsker å ha datamaskinen med i et musikalsk system som en medspiller heller enn en passiv ressurs, er det nødvendig å finne en metode som lar musikken danne premissene for endringsprosesser i programvare på samme måte som mellom to musikere. Noen ganger ønsker vi å fremheve dissonans eller rytmiske brudd, og dess mer markante disse bruddene er, dess større er avviket fra en normalisert strøm av *tekstur* og *timbre*. Jeg vet ikke om det er riktig å forvente at datamaskinen skal kunne forutse at det snart vil komme et brudd, men dersom vi kan sende den små impulser på forhånd (en analogi for blikkontakt mellom musikere) har vi gode sjanser for å få verdifull respons både for små endringer, men også når vi forbereder et markant brudd. Det er i denne sammenhengen sensorer kan hjelpe oss som ønsker å bruke datamaskinen som en utvidelse av hovedinstrumentet til å kontrollere programvare som om det er en fysisk del av et konkret instrument.

I kapittelet om mapping har jeg foreslått noen teknikker vi kan benytte for å bygge opp mappingsystemer for menneske-maskin interaksjon og bruke resultatene til å justere innstillinger. I den forbindelse har det vært nyttig for meg å bruke erfaringer fra både den utøvende og kompositoriske siden av musikklivet. Ettersom vi nå kan lage programvare som utfører oppgaver som for bare 10-15 år siden var forbeholdt en liten gruppe mennesker - med tanke på kostnadene ved innkjøp av utstyr - kjennes det som en stor utfordring å håndtere en

så rask utvikling. Jeg tenker ikke på utviklingen i datamaskiners regnekraft, men mulighetene som har åpnet seg innen instrumentdesign. Som nevnt tidligere har de aller fleste instrumenter som regnes for vanlige av folk flest, en historie på flere hundre, om ikke tusen år, og det er heldigvis en del erfaringer vi kan ta med oss fra denne historien inn i design av nye instrumenter. Så langt jeg kan se, etter å ha skrevet denne oppgaven, ligger utfordringen i å lage noe som varer og som kan brukes av kommende generasjoner. Det er viktig å ha i mente at de problemene vi har i dag med stykker som er skrevet med utgangspunkt i teknologi som ikke er i bruk, ikke lages eller ikke virker lenger, ikke føres videre. Når det gjelder programvare er det kanskje ikke så mye å gjøre ved at ting blir utdatert, men grensesnitt for menneskelig interaksjon bør kunne styres inn i en utvikling som ligner på den vi kjenner fra den historiske utviklingen av vanlige instrumenter. Hvordan dette bør gjøres er ikke så lett å definere ettersom mange av de vanlige instrumentene har vært gjennom flere hundre års utvikling. Men det kan vel sies med noenlunde sikkerhet, basert på *common sense,* at instrumenter med en historisk utvikling i ryggen ikke har blitt endret for å bli vanskeligere å håndtere, og at disse instrumentene ikke har endret form i det vesentlige: Strykeinstrumenter og gitarer har fortsatt en hals, et gripebrett og en kropp. Jeg tror denne utviklingen handler om å finne kjernen i interaksjonen for å gjøre bruken av instrumentet lettere, og jeg vil foreslå at kjernen i en hver musikalsk interaksjon er bevegelsene som får instrumenter til å produsere lyd. I forbindelse med bruken av akselerometer har jeg har lest mye av ingeniørenes fysikkpensum, og det slår meg at det i fysikken veldig ofte dreier seg om krefter i en eller annen tilstand, og jeg tror ikke at dette er noe annerledes for oss som arbeider med musikkteknologi.

Praktisk har jeg via denne oppgaven lært meg å programmere i *C* (via fag jeg har fulgt på Høgskolen i Vestfold, avdeling for Realfag og Ingeniør) samt at jeg har opparbeidet god kunnskap om programmering i *Max/MSP/Jitter*. Dette er erfaringer som har hatt, og kommer til å ha, stor betydning for gjennomføringen av mine musikalske ideer, og dette er nok det viktigste for meg personlig. Hele tiden under arbeidet med oppgaven har målet vært å nøste opp i de fagfeltene som legger hindringer i veien for oss som ikke har en historie som inkluderer elektronikk og programmering. Jeg synes at gjennomgangen av utviklingen av sensorgrensesnitt i kapittel tre forenkler omstendighetene rundt det å lage sitt eget grensesnitt tilstrekkelig til at andre enn meg selv kan ha nytte av det. Det krever riktignok at man ikke er redd for å sette seg inn i generelle termer om elektronikkens funksjon, og at man enten har kunnskap om *C* programmering fra før, eller har lyst til å lære det. Mange av dem jeg snakker med som ikke har erfaring med *C*, trekker seg bort fra det fordi det virker vanskelig. Da finner

jeg det på sin plass å minne om at jeg ikke hadde skrevet en millimeter *C*-kode før denne masteren, og at det ikke er spesielt vanskelig å lære det – i alle fall ikke for dem som har erfaring fra *Max/MSP/Jitter* programmering.

I utviklingen av programvare for lydbehandling har jeg valgt DSP-teknikker etter egne musikalske preferanser for å ha noe jeg liker og er kjent med som testmaterial for mapping-teknikkene. Jeg har ikke valgt metoder ut fra vitenskapelige kriterier, men siden granulering er stadig mer brukt kommersielt sett, har jeg lagt vekt på den delen av utviklingen. Etter å ha laget mange ulike programmer og varianter av granulering, ender jeg opp med samme konklusjon gang etter gang: Jeg foretrekker de metodene som knytter grainene mest mulig sammen, og dermed skaper koherens i *timbre* og *tekstur*. Dette er i samsvar med hva jeg vet om persepsjon av lyd gjennom å ha lest *Auditory Scene Analysis* (Bregman, 1990), og er derfor ingen overraskelse. Jeg finner det litt paradoksalt at jeg er så begeistret for en teknikk som kutter opp lyd i små biter fordi den samtidig setter bitene så fint sammen, og i den forbindelse vil jeg fortelle litt om planene fremover.

6.2 **Fremover**

Som nevnt ovenfor er granulering en teknikk jeg setter stor pris på fordi det hjelper meg både i komposisjon og sanntidbehandling av *timbre* og *tekstur*. Sammen med byggingen av sensor-grensesnittet, og de praktiske erfaringene i arbeidet med å bruke sensorene samtidig som jeg spiller, ønsker jeg å bidra til utviklingen av *NIME* via lydpersepsjon som fag og med utgangs-punkt i ulike granuleringsteknikker. For å få en vellykket kopling mellom granulering, grense-snitt og persepsjon er det åpenbart at vi trenger gode resultater fra sensorer og analyser for å få ut de lydene vi ønsker. Det er to hovedgrunner til at jeg tror granulering er en god innfalls-vinkel til design av *NIME*:

1. Granulering har i sin natur en evne til å omstrukturere hvilken som helst kompleks lyd. Dermed åpner granulering for en friere tenking i DSP-design enn hva som er tilfelle ved å forsøke å være tro mot et kjent instruments auditive egenskaper ved bruk av fysiske modeller.
2. Granulering er godt egnet som utgangspunkt for design av *NIME* ettersom teknikken i seg selv åpner for endringer i kompleksiteten i det auditive resultatet ut fra resultatene av sensorinteraksjon og analyse.

Punkt nummer to bringer oss tilbake til Newtons lover om krefter som virker på og i forhold til legemer, og jeg tror at de opplysningene vi har fra kjente akustiske instrumenter, kan hjelpe oss til å skjønne hvordan vi forholder oss til lydproduserende gester. Det finnes ikke et eneste instrument som lager lyd uten at det virker krefter direkte på eller nær grensesnittet (*Theremin*), og i mitt hode er dette faktumet premiss nummer en for et vellykket design. Instrumenter i alle kategorier bør ha et reaksjonsmønster som fremelsker konkrete bevegelser, og ikke et design som gjør utøveren usikker på hva som vil skje under og etter neste gest. Forutsigbarhet i koplingen mellom grensesnitt og lydkilde er den faktoren jeg holder for å være den viktigste. Dette danner bakgrunnen for de teknikkene jeg har gått gjennom under seksjonen om dynamisk mapping, og jeg ønsker å videreutvikle tankene om å skalere og visualisere sensordataene med definerte matematiske funksjoner som både kan erstatte og utvide kontaktpunktene mellom for eksempel strenger, tangenter, trommer og fingre i *NIME*.

Litteraturliste

Analog Devices, 1998, *Using The Low Cost, High Performance ADSP-21065L Digital Signal Processor For Digital Audio Applications*, Revision 1.0 – 12/4/98 tilgjengelig fra: http://www.analog.com/UploadedFiles/Application_Notes/7056820721065L_Audio_Tutorial.pdf#xml=http://search.analog.com/search/pdfPainter.aspx?url=http://www.analog.com/UploadedFiles/Application_Notes/7056820721065L_Audio_Tutorial.pdf&fterm=reverberation&fte

Atmel, 2006, *ATmega32(L), revision J, updated 10/06, tilgjengelig fra http://atmel.com/dyn/products/product_card.asp?part_id=2014*

Bogart, Thedore F, Beasley, Jeffrey S, og Rico, Guillermo. 2004 *Electronic Devices and Circuits, 6th. Edition*, Prentice Hall , 2004

Bregman, Albert S. 1990, *Auditory Scene Analysis. The Perceptual Organization of Sound,* Cambridge, Massachussets: The MIT Press

Buchla, Don. 2005, *A History of Buchla's Musical Instruments,*Proceedings of the 2005 International Conference on New Interfaces for Musical Expression (NIME05), Vancouver, BC, Canada

Cardew, Cornelius. 1974, *Stockhausen serves imperialism.* Tilgjengelig fra Jstor.

Celestion, *http://professional.celestion.com/guitar/products/pdfs/Vintage%2030.pdf*

Cycling74, 2006, *Getting Started*, Version 4.6, tilgjengelig fra http://cycling74.com/twiki/bin/view/ProductDocumentation

Deitel, Harvey M ,Deitel, Paul J, 2004, *C How to program, fourth edition*, Prentice-Hall, Inc. New Jersey, Pearson Education International.

Doornbush, Paul. 2005, *Computer Sound Synthesis in 1951: The Music of CSIRAC,* Computer Music Journal, 28:1, pp. 10–25, Spring 2004, Massachusetts Institute of Technology.

Floyd, Thomas L. 2005, *Digital Fundamentals, International Edition, 9th edition,* Pearson Education.

Godøy, Rolf Inge. 1999. *Shapes and Spaces in Musical Thinking*. Universitetet i Oslo, Institutt for musikkvitenskap.

Godøy, Rolf Inge. 2006. *Gestural-Sonorous Objects: embodied extensions of Schaeffer's conceptual apparatus*. Organised Sound 2006, 11(2):149-157

Levitin, D. J., McAdams, S., and Adams, R. L. 2002. *Control parameters for musical instruments: a foundation for new mappings of gesture to sound.* Org. Sound 7, 2 (Aug. 2002), 171-189.

Kvifte, Tellef, 1988, *Instruments and the Electronic Age*, Solum Forlag

Mathews, Max V., Moore, 1970, F.R. *GROOVE—a program to compose, store, and edit functions of time,* Communications of the ACM archive Volume 13 , Issue 12 (December 1970) Pages: 715 - 721 Year of Publication:1970 ISSN:0001-0782

Mathews, Max V. 1963, *The Digital Computer as a Musical Instrument*, Science, New Series, Vol. 142, No. 3592. (Nov. 1, 1963), pp. 553-557.

Mathews, Max V. ; L. Rosler. 1968, *Graphical Language for the Scores of Computer-Generated Sounds*, Perspectives of New Music, Vol. 6, No. 2. (Spring - Summer, 1968), pp. 92-118.

Momeni, Ali og Wessel, David, 2003, *Characterizing and Controlling Musical Material Intuitively with Geometric Models,* Proceedings of the 2003 Conference on New Interfaces for Musical Expression (NIME-03), Montreal, Canada

Paiva, Rui Pedro., Mendes, Teresa. og Cardoso, Amilcar, 2005, *Segmentation of Pitch Tracks for Melody Detection in Polyphonic Audio*, Proceedings of the 13th European Signal Processing Conference, EUSIPCO'2005, Antalya, Turkey, September 2005

Paradiso, Joseph A. 1997, *Electronic music:New ways to play,* IEEE Spectrum, IEEE

Roads, Curtis. 2002, *Microsound,* The MIT Press; Har/Com edition

Roads, Curtis. 1996 , *The Computer Music Tutorial*, The MIT Press February 27, 1996, fifth edition 2000

Rossing, Thomas D., Wheeler, Paul. og Moore, Richard F. 2002, *The science of sound,* Pearson Education, Inc

Rovan, Joseph Butch, Wanderley, Marcelo M., Dubnov, Shlomo. Depalle, Phillippe. 2000, *Instrumental Gestural Mapping Strategies as Expressivity Determinants in Computer Music Performance,* Analysis-Synthesis Team/Real-Time Systems Group, Ircam

Simeone, Nigel. 2002, *Music at the 1937 Paris Exposition: The Science of Enchantment,* The Musical Times, Vol. 143, No. 1878. (Spring, 2002), pp. 9-17. Musical Times Publications Ltd.

Van Nort, Doug og Wanderley, Marcelo M. 2006, *Exploring the effect of mapping trajectories on musical performance,* Proceedings of the 2006 Sound and Music Computing Conference (SMC06), Marseille, France, 2006

Wanderley, Marcel M. og Depalle, Philippe.2004, *Gestural Control of Sound Synthesis,* Proceedings of the IEEE, Vol.92, NO. 4, april 2004.

Wessel, David L. 1978, *Timbre space as musical controller structure,* Rapport Ircam 12/78, 1978, oppdatert versjon:1999

Young, Hugh D. og Freedman, Roger A. 2004, *University Physics,* Pearson Education, Inc

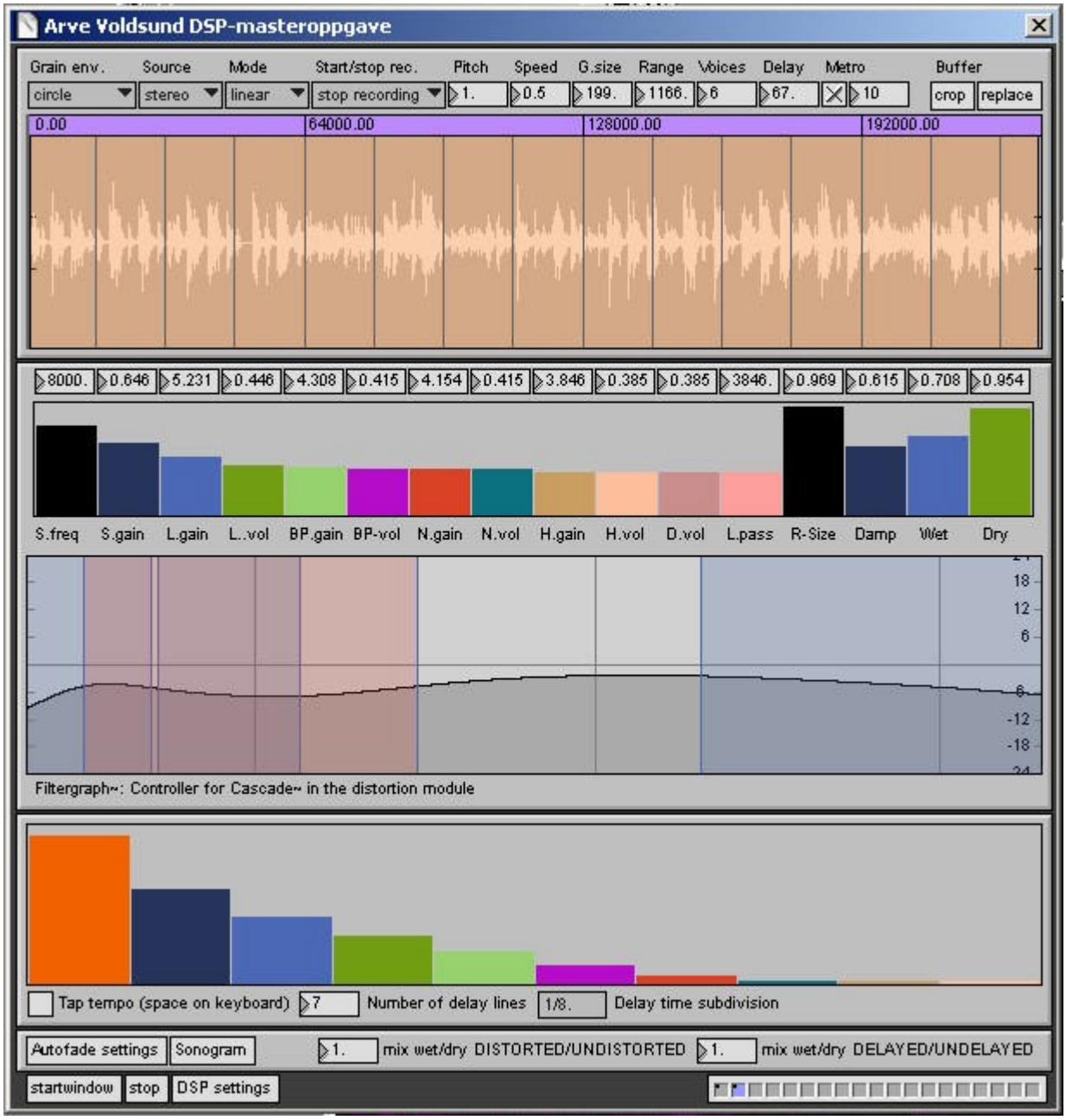
Z-World, Inc. 2004, *Wolf (BL2600) User's Manual,*Z-World Inc. Tilgjengelig fra http://www.rabbitsemiconductor.com/products/dc/index.shtml

Zadeh, Lotfi, A. 1965, *Fuzzy Sets,* Department of Electrical Engineering and Electronics Research Laboratory, University of California, Berkeley, California. Tilgjengelig fra: http://www-bisc.cs.berkeley.edu/Zadeh-1965.pdf

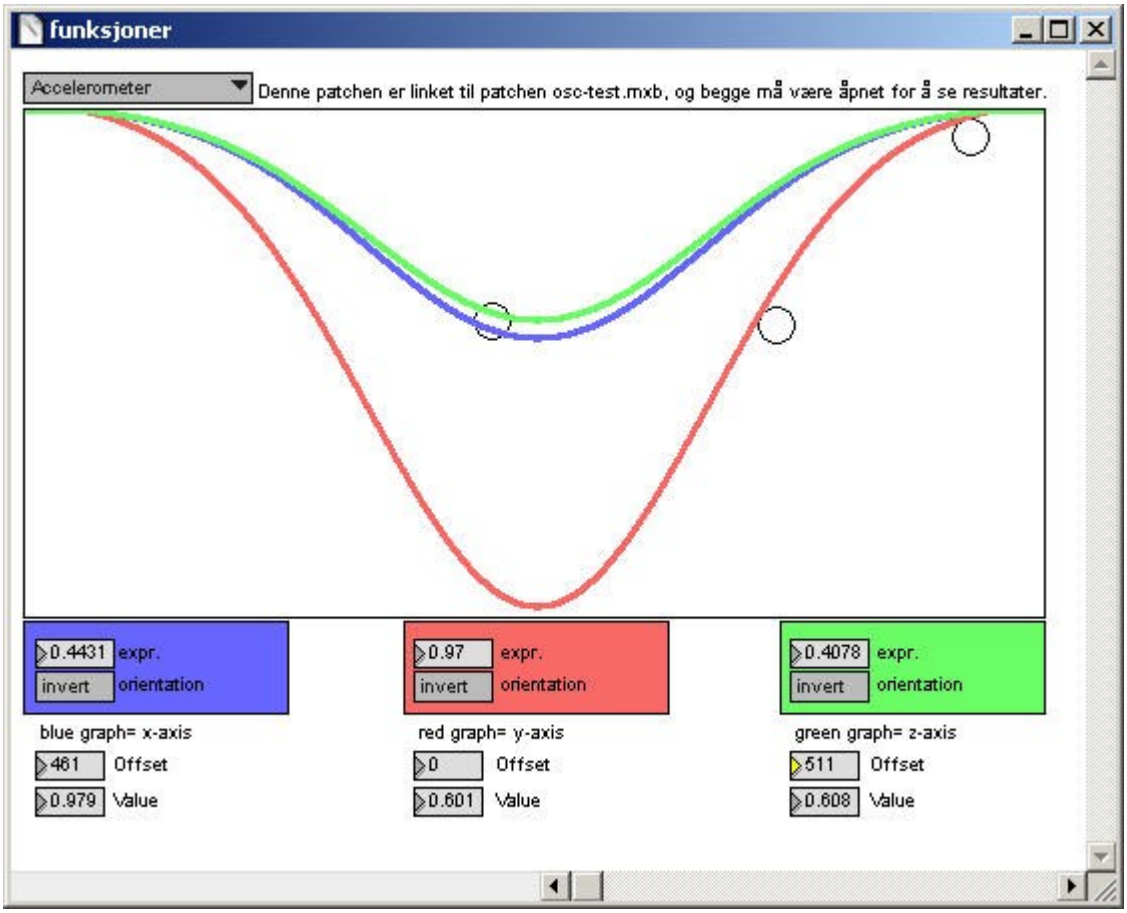
Zadeh, Lotfi A. 1994, *Fuzzy Logic, Neural Networks, and Soft Computing*,Communications of the ACM, March 1994/vol. 37, no 3. Tilgjengelig fra http://www-bisc.cs.berkeley.edu/zadeh/papers/Fuzzy%20Logic,%20Neural%20Networks,%20and%20Soft%20Computing-1994.pdf

Zicarelli, David. 2002, *How I Learned to Love a Program That Does Nothing,* Computer Music Journal, 26:4, pp. 44–51, Winter 2002, The MIT Press.

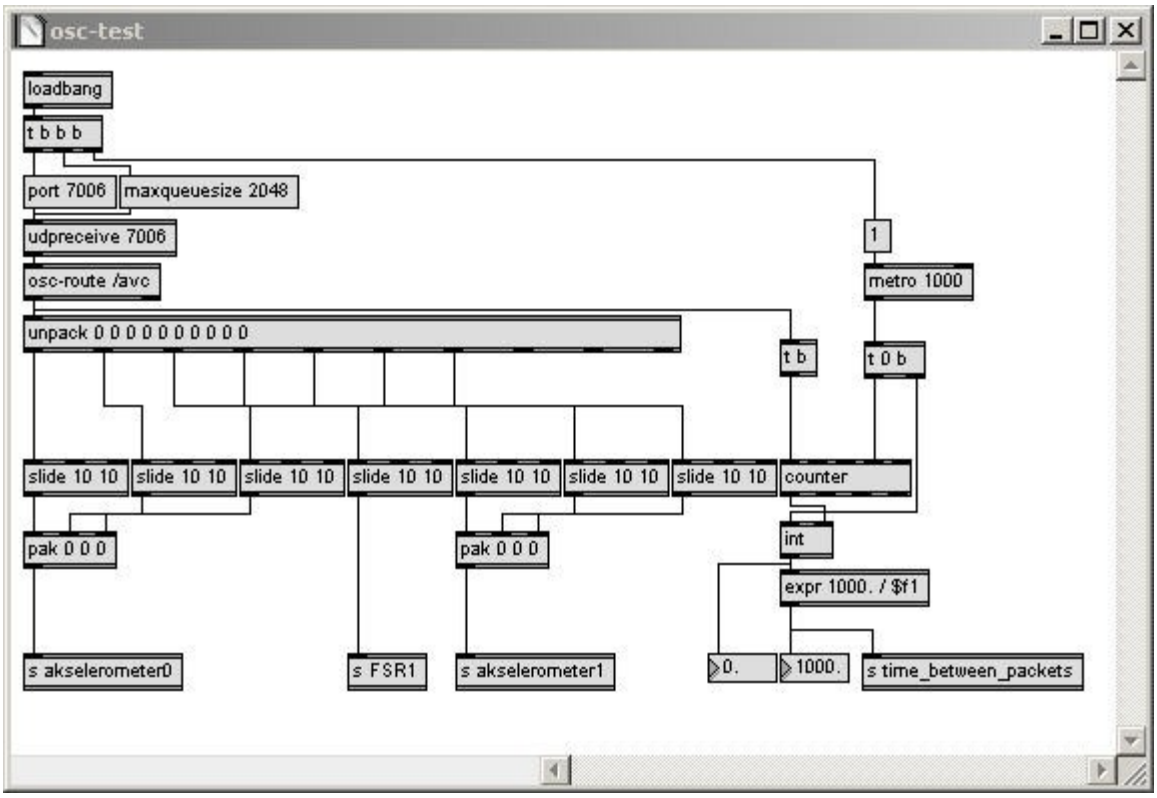
APPENDIX 1: Oversikt over programvare utviklet for denne oppgaven



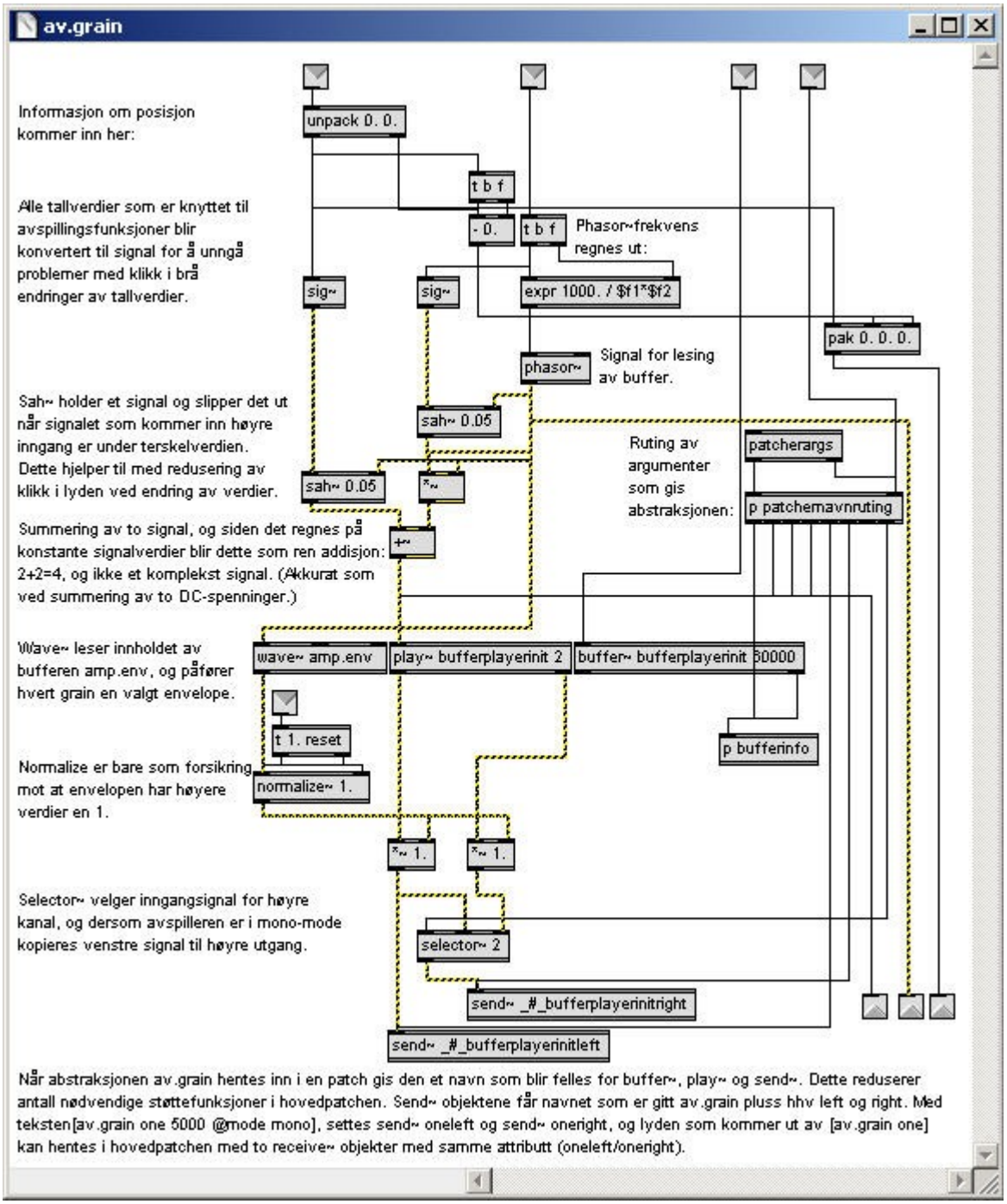
Figur appendix 1.1: Patchen hvor alle de utvalgt DSP-funksjonene er samlet. Denne kan brukes både ved å sample fra lydkortet, men også ved å laste inn en lydfil. All bruk er for egen risiko.



Figur appendix 1.2: bilde av patchen som er brukt som eksempel for skalering via matematiske funksjoner for visuell tilbakemelding om banene som det skalerte signalet følger.



gFigur appendix 1.3: Mottak av sensordata fra sensorgrensesnitt som er utviklet for denne oppgaven. Alle data fra grensesnittet sendes ut til ip-adresse 192.168.2.103 på port 7006.



Figur appendix 1.4: av.grain – kjernen i granuleringspatchen (hovedelement i DSP patchen)

APPENDIX 2: OSC-spesifikasjon:

version 1.0, March 26 2002, Matt Wright

Introduction

OpenSound Control (OSC) is an open, transport-independent, message-based protocol developed for communication among computers, sound synthesizers, and other multimedia devices.

OSC Syntax

This section defines the syntax of OSC data.

Atomic Data Types

All OSC data is composed of the following fundamental data types:

int32

32-bit big-endian two's complement integer

OSC-timetag

64-bit big-endian fixed-point time tag, semantics defined below

float32

32-bit big-endian IEEE 754 floating point number

OSC-string

A sequence of non-null ASCII characters followed by a null, followed by 0-3 additional null characters to make the total number of bits a multiple of 32. (OSC-string examples) In this document, example OSC-strings will be written without the null characters, surrounded by double quotes.

OSC-blob

An int32 size count, followed by that many 8-bit bytes of arbitrary binary data, followed by 0-3 additional zero bytes to make the total number of bits a multiple of 32.

The size of every atomic data type in OSC is a multiple of 32 bits. This guarantees that if the beginning of a block of OSC data is 32-bit aligned, every number in the OSC data will be 32-bit aligned.

OSC Packets

The unit of transmission of OSC is an *OSC Packet*. Any application that sends OSC Packets is an *OSC Client*; any application that receives OSC Packets is an *OSC Server*.

An OSC packet consists of its *contents*, a contiguous block of binary data, and its *size*, the number of 8-bit bytes that comprise the contents. The size of an OSC packet is always a multiple of 4.

The underlying network that delivers an OSC packet is responsible for delivering both the contents and the size to the OSC application. An OSC packet can be naturally represented by a datagram by a network protocol such as UDP. In a stream-based protocol such as TCP, the stream should begin with an int32 giving the size of the first packet, followed by the contents of the first packet, followed by the size of the second packet, etc.

The contents of an OSC packet must be either an *OSC Message* or an *OSC Bundle*. The first byte of the packet's contents unambiguously distinguishes between these two alternatives.

OSC Messages

An OSC message consists of an *OSC Address Pattern* followed by an *OSC Type Tag String* followed by zero or more *OSC Arguments*.

Note: some older implementations of OSC may omit the OSC Type Tag string. Until all such implementations are updated, OSC implementations should be robust in the case of a missing OSC Type Tag String.

OSC Address Patterns

An OSC Address Pattern is an OSC-string beginning with the character '/' (forward slash).

OSC Type Tag String

An OSC Type Tag String is an OSC-string beginning with the character ',' (comma) followed by a sequence of characters corresponding exactly to the sequence of OSC Arguments in the given message. Each character after the comma is called an *OSC Type Tag* and represents the type of the corresponding OSC Argument. (The requirement for OSC Type Tag Strings to start with a comma makes it easier for the recipient of an OSC Message to determine whether that OSC Message is lacking an OSC Type Tag String.)

This table lists the correspondance between each OSC Type Tag and the type of its corresponding OSC Argument:

OSC Type Tag	Type of corresponding argument
i	int32
f	float32
s	OSC-string
b	OSC-blob

The meaning of each OSC Type Tag

Some OSC applications communicate among instances of themselves with additional, nonstandard argument types beyond those specified above. OSC applications are not required to recognize these types; an OSC application should discard any message whose OSC Type Tag String contains any unrecognized OSC Type Tags. An application that does use any additional argument types must encode them with the OSC Type Tags in this table:

OSC Type Tag	Type of corresponding argument
h	64 bit big-endian two's complement integer
t	OSC-timetag
d	64 bit ("double") IEEE 754 floating point number
S	Alternate type represented as an OSC-string (for example, for systems that differentiate "symbols" from "strings")
c	an ascii character, sent as 32 bits
r	32 bit RGBA color
m	4 byte MIDI message. Bytes from MSB to LSB are: port id, status byte, data1, data2
T	True. No bytes are allocated in the argument data.
F	False. No bytes are allocated in the argument data.
N	Nil. No bytes are allocated in the argument data.
I	Infinitum. No bytes are allocated in the argument data.
[	Indicates the beginning of an array. The tags following are for data in the Array until a close brace tag is reached.
]	Indicates the end of an array.

OSC Type Tags that must be used for certain nonstandard argument types

OSC Type Tag String examples.

OSC Arguments

A sequence of OSC Arguments is represented by a contiguous sequence of the binary representations of each argument.

OSC Bundles

An OSC Bundle consists of the OSC-string "#bundle" followed by an *OSC Time Tag*, followed by zero or more *OSC Bundle Elements*. The OSC-timetag is a 64-bit fixed point time tag whose semantics are described below.

An OSC Bundle Element consists of its *size* and its *contents*. The size is an int32 representing the number of 8-bit bytes in the contents, and will always be a multiple of 4. The contents are either an OSC Message or an OSC Bundle.

Note this recursive definition: bundle may contain bundles.

This table shows the parts of a two-or-more-element OSC Bundle and the size (in 8-bit bytes) of each part.

Data	**Size**	**Purpose**
OSC-string "#bundle"	8 bytes	How to know that this data is a bundle
OSC-timetag	8 bytes	Time tag that applies to the entire bundle
Size of first bundle element	int32 = 4 bytes	First bundle element
First bundle element's contents	As many bytes as given by "size of first bundle element"	
Size of second bundle element	int32 = 4 bytes	Second bundle element
Second bundle element's contents	As many bytes as given by "size of second bundle element"	
etc.		Addtional bundle elements

Parts of an OSC Bundle

OSC Semantics

This section defines the semantics of OSC data.

OSC Address Spaces and OSC Addresses

Every OSC server has a set of *OSC Methods*. OSC Methods are the potential destinations of OSC messages received by the OSC server and correspond to each of the points of control that the application makes available. "Invoking" an OSC method is analogous to a procedure call; it means supplying the method with arguments and causing the method's effect to take place.

An OSC Server's OSC Methods are arranged in a tree strcuture called an *OSC Address Space*. The leaves of this tree are the OSC Methods and the branch nodes are called *OSC Containers*. An OSC Server's OSC Address Space can be dynamic; that is, its contents and shape can change over time.

Each OSC Method and each OSC Container other than the root of the tree has a symbolic name, an ASCII string consiting of printable characters other than the following:

character	name	ASCII code (decimal)
' '	space	32
#	number sign	35
*	asterisk	42
,	comma	44
/	forward slash	47
?	question mark	63
[	open bracket	91
]	close bracket	93
{	open curly brace	123
}	close curly brace	125

Printable ASCII characters not allowed in names of OSC Methods or OSC Containers

The *OSC Address* of an OSC Method is a symbolic name giving the full path to the OSC Method in the OSC Address Space, starting from the root of the tree. An OSC Method's OSC Address begins with the character '/' (forward slash), followed by the names of all the containers, in order, along the path from the root of the tree to the OSC Method, separated by forward slash characters, followed by the name of the OSC Method. The syntax of OSC Addresses was chosen to match the syntax of URLs. (OSC Address Examples)

OSC Message Dispatching and Pattern Matching

When an OSC server receives an OSC Message, it must invoke the appropriate OSC Methods

in its OSC Address Space based on the OSC Message's OSC Address Pattern. This process is called *dispatching* the OSC Message to the OSC Methods that *match* its OSC Address Pattern. All the matching OSC Methods are invoked with the same argument data, namely, the OSC Arguments in the OSC Message.

The *parts* of an OSC Address or an OSC Address Pattern are the substrings between adjacent pairs of forward slash characters and the substring after the last forward slash character. (examples)

A received OSC Message must be disptched to every OSC method in the current OSC Address Space whose OSC Address matches the OSC Message's OSC Address Pattern. An OSC Address Pattern matches an OSC Address if

1. The OSC Address and the OSC Address Pattern contain the same number of parts; and
2. Each part of the OSC Address Pattern matches the corresponding part of the OSC Address.

A part of an OSC Address Pattern matches a part of an OSC Address if every consecutive character in the OSC Address Pattern matches the next consecutive substring of the OSC Address and every character in the OSC Address is matched by something in the OSC Address Pattern. These are the matching rules for characters in the OSC Address Pattern:

1. '?' in the OSC Address Pattern matches any single character
2. '*' in the OSC Address Pattern matches any sequence of zero or more characters
3. A string of characters in square brackets (e.g., "[string]") in the OSC Address Pattern matches any character in the string. Inside square brackets, the minus sign (-) and exclamation point (!) have special meanings:
    - two characters separated by a minus sign indicate the range of characters between the given two in ASCII collating sequence. (A minus sign at the end of the string has no special meaning.)
4. An exclamation point at the beginning of a bracketed string negates the sense of the list, meaning that the list matches any character not in the list. (An exclamation point anywhere besides the first character after the open bracket has no special meaning.)
5. A comma-separated list of strings enclosed in curly braces (e.g., "{foo,bar}") in the OSC Address Pattern matches any of the strings in the list.
6. Any other character in an OSC Address Pattern can match only the same character.

Temporal Semantics and OSC Time Tags

An OSC server must have access to a representation of the correct current absolute time. OSC does not provide any mechanism for clock synchronization.

When a received OSC Packet contains only a single OSC Message, the OSC Server should invoke the correponding OSC Methods immediately, i.e., as soon as possible after receipt of the packet. Otherwise a received OSC Packet contains an OSC Bundle, in which case the OSC Bundle's OSC Time Tag determines when the OSC Bundle's OSC Messages' corresponding OSC Methods should be invoked. If the time represented by the OSC Time Tag is before or equal to the current time, the OSC Server should invoke the methods immediately (unless the user has configured the OSC Server to discard messages that arrive too late). Otherwise the OSC Time Tag represents a time in the future, and the OSC server must store the OSC Bundle until the specified time and then invoke the appropriate OSC Methods.

Time tags are represented by a 64 bit fixed point number. The first 32 bits specify the number of seconds since midnight on January 1, 1900, and the last 32 bits specify fractional parts of a second to a precision of about 200 picoseconds. This is the representation used by Internet NTP timestamps.The time tag value consisting of 63 zero bits followed by a one in the least signifigant bit is a special case meaning "immediately."

OSC Messages in the same OSC Bundle are *atomic*; their corresponding OSC Methods should be invoked in immediate succession as if no other processing took place between the OSC Method invocations.

When an OSC Address Pattern is dispatched to multiple OSC Methods, the order in which the matching OSC Methods are invoked is unspecified. When an OSC Bundle contains multiple OSC Messages, the sets of OSC Methods corresponding to the OSC Messages must be invoked in the same order as the OSC Messages appear in the packet. (example)

When bundles contain other bundles, the OSC Time Tag of the enclosed bundle must be greater than or equal to the OSC Time Tag of the enclosing bundle. The atomicity requirement for OSC Messages in the same OSC Bundle does not apply to OSC Bundles within an OSC Bundle.

APPENDIX 3: OSC-kildekode for Z-World BL2600 Wolf

```
#class auto                                          //lokale variabler lagres på stack
#define MAX_UDP_SOCKET_BUFFERS 1                    //definerer maks 1 udp sokkel buffer.
#define HEARTBEAT_RATE 0.005                        //Definerer en klokke for sending (5ms)
#define LOCAL_PORT   1234                           //Sender fra port 1234
#define REMOTE_IP              "192.168.2.103"      //Sender til denne IP-adressen
#define MY_IP_ADDRESS    "192.168.2.127"            //Sensorgrensesnittets IP-adresse
#define MY_NETMASK       "255.255.255.0"            //Netmaske
#define MY_GATEWAY       "255.255.255.0"            //Gateway
#define MY_NAMESERVER  "192.168.2.128"              //Nameserver
#define REMOTE_PORT      7006                       //sender til denne porten
#define int4byte long                               //int er 4 byte
#memmap xmem                                        //Se Dynamic C manual.
#use "dcrtcp.lib"                                   //Henter bibliotek for tcp funksjoner
#use "BL26XX.LIB"                                   //Bibliotek for BL26XX maskinvare
//Konfigurering ferdig---Globale variabler defineres:-----------------------------------------------

udp_Socket sock;                                    //udp sokkel kalles sock
char buf[100];                                      //Bufferen buf opprettes som et array
  const int length=100;                             //int length 100 for udp_send()
  int i,j;                                          //oppretter lagringsplass for tellere
  int a0,a1,a2,a3,a4,a5,a6,a7;                      //oppretter lagringsplasser,heltall(ADC)
  int retval;                                       //lagringsplass for udp_send returnverdi

//Globale variabler ferdig definert-----------------------------------------------------------------
//Her starter oppretting av funksjoner som trengs for OSC-støtte-----------------------------------

void pad0()                                         //Funksjonen pad0 opprettes og setter
   {int i;                                          //hele bufferen buf lik 0 for å sikre
   for(i=0;i<100;i++) buf[i]=0;                     //at bufferen er "tom".
   }
void adressABC()                                    //Funksjonen  adressABC opprettes.
   {
   buf[0]='/';                                      //Obligatorisk / før OSC-adressens navn
   buf[1]='a';buf[2]='v';buf[3]='c';                //adressen til OSC-pakken
                                                    //settes inn i bufferen buf
   }
```

```
void typetagS()                                          //Funksjonen typetagS opprettes
   {
  buf[8]=',';                                            //Obligatorisk komma først i typetags
  buf[9]='i'; buf[10]='i';                               //Typetags settes til int, og
  buf[11]='i'; buf[12]='i';                              //plasseres i buffer.
  buf[13]='i'; buf[14]='i';
  buf[15]='i'; buf[16]='i'; buf[17]='i'; buf[18]='i'; buf[19]='i';
   }

void argInt(int t,int q)                                 //Funksjonen argInt opprettes, og mottar
                                                         //verdien fra ADC og lagrer som heltall
                                                         //i t, og mottar buf.posisjon og lagrer
                                                         //i q. Takk til Arnfinn Lunde for hjelp
                                                         //med denne funksjonen.
    {
     char *p1, *p2;                                      //Funksjonen argInt får to peker-
                                                         //variabler av typen char (karakter).
     union utag{                                         //Unionen utag defineres.
     int ival;                                           //Utag får en lagringsplass for et
                                                         //heltall, og lagrinsplassen heter ival.
     char v[4];                                          //Utag får en lagringsplass for et
                                                         //char (karakter) array med 4 byte.
     }w;                                                 //w blir en variabel av typen utag.
     w.ival=t;                                           //t blir lagret i w sin medlem ival
     p2=&w.v[0];                                         //p2=adressen til w sin medlem v[0].
     p1=&buf[q+7];                                       //p1=adressen til buf [q+7]
     for(i=0;i<2;i++) *p1-- = *p2++ ;                    //*p2 flyttes 1 plass høyere i v[], og
                                                         //verdien lagres der p1 peker - 1 plass.
                                                         //OSC spesifiserer int til 4 byte, og
                                                         //siden ad-konverteringen er 11-bit
                                                         //(2-byte) må tallet plasseres riktig i
                                                         //buffer.
}
```

```
int send_packet(void)                                   //Denne funksjonen er stort sett hentet
                                                        //fra Dynamic C eksempler.
{
  retval = udp_send(&sock, buf, length);                //udp_send er Dynamic C funksjon.
  if (retval < 0) {
    printf("Aapner sokkel...\n");                       //debuginfo
    sock_close(&sock);
    if(!udp_open(&sock, LOCAL_PORT, resolve(REMOTE_IP), REMOTE_PORT, NULL)) {
      printf("Beklager det gikk ikke!\n");              //debug info
      exit(0);
    }
  }
  tcp_tick(NULL);                                       //Se Dynamic C TCP/IP manual
  return 1;                                             //Returnerer 1 dersom det gikk bra å
                                                        //sende pakken.
}

//Slutt på oppretting av funksjoner-------------------------------------------------------------------------

//Main begynner her: Main styrer hele programmet, og henter inn funksjonene i den //rekkefølge de settes
opp.

void main()
{
  brdInit();                                            //brdInit er i biblioteket til DynamicC
  sock_init();                                          //sock_init er i biblioteket til DynamicC
  pad0();                                               //pad0 er en OSC-relatert funksjon (se
                                                        //ovenfor)
  adressABC();                                          //adressABC er en OSC-relatert
                                                        //funksjon (se ovenfor)
  typetagS();                                           //typetagS er en OSC-relatert funksjon
                                                        //(se ovenfor)
{
  anaInConfig(0, SE0_MODE);                             //Konfigurer de analoge inngange
  anaInConfig(1, SE0_MODE);                             //parvis, og setter de i Single
  anaInConfig(2, SE0_MODE);                             //Ended Mode.
  anaInConfig(3, SE0_MODE);
}
```

```
for(;;)                                          //denne for () loopen lagrer verdiene fra ADC i
                                                 //programmets variabler, og kaller funksjonen argInt
                                                 //som plasserer verdiene på riktig sted i bufferen buf.
  {
                a0=anaIn(0,2); a1=anaIn(1,2);
                a2=anaIn(2,2); a3=anaIn(3,2);
                a4=anaIn(4,2); a5=anaIn(5,2);
                 a6=anaIn(6,2); a7=anaIn(7,2);
    {
                argInt(a0,20); argInt(a1,24);
                argInt(a2,28); argInt(a3,32);
                argInt(a4,36); argInt(a5,40);
                argInt(a6,44); argInt(a7,48);

     costate{                                                    //Dynamic C funksjonalitet
                waitfor(IntervalSec(HEARTBEAT_RATE));    //Venter på intern klokke
                waitfor(send_packet());                  //Venter til send_packet er ferdig

            }
     }
  }
}


//Programmet er slutt-----------------------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------------------------------------
```

APPENDIX 4: Spesifikasjoner for Z-World BL2600 Wolf

Egenskap:	***Spesifikasjon:***
Microprocessor	Rabbit 3000 at 44.2 MHz
Ethernet Port	10/100Base-T, 3 LEDs
Flash Memory	Flash Memory
Program Execution SRAM	512K
Data SRAM	256K
Configurable I/O	16: Individually software configurable @ ±36 V DC, 1.5 V switching threshold, or sinking digital outputs up to 40 V, 200 mA each
Digital Inputs	16: Hardware-configurable pull-up or pull-down, ±36 V DC, switching threshold 1.4 V typ.
High-Current Digital Outputs	4: Individually software configurable, +40 V DC, 2 A max. per channel, sinking or sourcing
Analog Inputs	8 channels with 11-bit resolution, software selectable ranges Unipolar: 1, 2, 2.5, 5, 10, 20 V DC; Bipolar: ± 1, ± 2, ± 5, ± 10 V DC; Four of the eight channels are hardware-configurable for 4 - 20 mA , 12 kHz update rate
Analog Outputs	4 channels, 12-bit resolution, buffered 0 - 10 V DC, ± 10 V DC, and 4 - 20 mA, 12 kHz update rate
Serial Ports	Up to 5 serial ports: • 1 RS-485 or 1 RS-232 • 2 RS-232 or one RS-232 (with CTS/RTS) • 1 clocked serial port multiplexed to 2 RS-422 SPI master ports • 1 CMOS compatible serial port for programming/debug
Serial Rate	Max. async = CLK/8, Max. sync = CLK/2
Real-Time Clock	Yes
Timers	Ten 8-bit timers (6 cascadable from the first) and one 10-bit timer with 2 match registers
Watchdog/Supervisor	Yes
Power	9 – 36 V DC, 12 W
Connectors	• One Ethernet and two RabbitnetT RJ-45 connectors • Two polarized, 9-position with 0.1" pitch friction-lock connectors • Three 4-position power terminals with 0.156" pitch friction-lock connectors • Two 20-position terminals with 0.1" pitch (and 2 x 20 IDC headers) friction-lock connectors • One 13-position terminal with 0.1" pitch (and 2 x 13 IDC header) friction-lock connector • One 10-position terminal with 0.1" pitch (and 2 x 7 IDC header) friction-lock connector • One 2 x 5 IDC, 1.27 mm pitch (BL2600) programming port • One 2 x 5 IDC, 2 mm pitch (BL2610) programming port
Board Size	123 × 126 × 25 mm

Appendix 4: Tekniske spesifikasjoner for grensesnittet gjengitt fra brukermanualen til BL2600Wolf (Z-World, 2004).