KTH Computer Science
and Communication

A Synth in its Prime

En implementering och evaluering av en nittontonssynt.

SIMON KLEIN
WARFVINGES VÄG 13
112 18 STOCKHOLM
070-862 15 96
SKLEIN@KTH.SE

,

ANDERS HAGWARD
ÄRVINGEVÄGEN 14
164 46 KISTA
073-505 89 56
HAGWARD@KTH.SE

Examenrapport vid CSC
Handledare: Henrik Eriksson
Examinator: Mårten Björkman

Referat

I denna uppsats undersöks och beskrivs hur man kan implementera en synt med 19 toner. Detta i kontrast till vår normala musikskala som har 12 toner. Relevanta delar ur musikteorin gås igenom för att ge läsaren möjlighet att förstå de resonemang som senare kommer att föras när de två skalorna jämförs och den tekniska implementationen beskrivs. Detta är inte bara en studie i musikskalor, utan också en studie i hur man implementerar en synt. Framförallt det förhållandevis nya programmeringsspråket ChucK tas upp i samband med ljudlogiken för synten och blir utsatt för analys och diskussion, men även hur man skapar ett grafiskt gränssnitt i Java och får den delen att samarbeta med ett program skrivet i ett vilt främmande språk klargörs.

Abstract

A synth in its Prime

In this essay it is described how to implement a 19 tone digital synthesizer. This in contrast to our normal musical scale which consists of 12 tones. Relevant pieces of music theory is reviewed to give the reader a chance to grasp the reasoning that is used when the two musical scales are compared and the technical implementation is explained. This is not only a study of musical scales, but also a study in how to implement a digital synthesizer. In particular the relatively new programming language ChucK is examined in order to construct the audio logic for the synthesizer and is consequently liable for analysis and discussion, but also the procedure of creating a graphical interface in Java and making it cooperate with an application written in an extraneous programming language is made clear.

Innehåll

Kapitel 1

Inledning

I den västerländska världen har vi sedan länge vant oss vid en musikskala med tolv stycken toner. Faktum är dock att uppdelningen i tolv toner på intet sätt är given av naturen. Inom arabisk musik är det till exempel vanligt förekommande med 24 toner[1]. Antalet toner är alltså godtyckligt, men varje uppdelning är ingalunda lika bra som den andra. Att ha tolv toner fungerar uppenbarligen bra och de flesta instrument är anpassade efter dessa, men vore det inte intressant att få prova ett instrument med ett annat antal toner?

1.1 Problemformulering

Det finns anledning att tro att en uppdelning av oktaven i 19 olika toner skulle fungera väl. Tyvärr finns det inte en uppsjö av instrument stämda efter en skala med 19 toner, och kanske skulle det till och med vara rent ut sagt besvärligt att åstadkomma detta, men den digitala tidsåldern med datorer och programmeringsspråk erbjuder nya möjligheter. Därför kan man med fördel försöka att skapa en mjukvarubaserad, digital synt med utgångspunkt i en musikskala med 19 toner. Vill man sedan kunna jämföra 19-tonsskalan med den vanliga 12-tonsskalan bör synten även ha stöd för den senare.

Med siktet inställt på en sådan synt, kan man ställa följande två frågor.

- Hur förhåller sig en skala med 19 tonsteg till den vanliga med tolv; kan man utan förbehåll ersätta den tolvindelade skalan med den nittonindelade eller är en skala med 19 toner bara ett intressant komplement?
- Är det lätt att i dagens alltmer digitaliserade samhälle skapa ett instrument med en egen musikskala och hur kan en sådan procedur gå till?

En intressant detalj är att till skillnad från 12, är 19 ett primtal. Detta fascinerande faktum skulle kunna undersökas, men ryms tyvärr inte här. Uppsatsen handlar dock om en synt som har ett till antalet primtal toner och detta har fått ge dokumentet namnet “A Synth in its Prime”, vilket vitsigt kan översättas till “En prima synt”.

1.2 För läsaren

Innan man börjar att läsa ett relativt omfattande dokument som detta kan det vara bra att veta vad som förväntas av en som läsare och vilka avsnitt som kan vara läsvärda. Läsaren av dokumentet förväntas inte ha några särskilda förkunskaper inom musikalisk teori. Följaktligen gås de grundläggande och relevanta delarna ur denna igenom i avsnitt *2 Musikteori*. Detta dokument behandlar dock inte endast 19-tonsskalan i förhållande till 12-tonsskalan, ty det skulle inte ha särskilt stor koppling till datalogi, utan också processen för att implementera en digital synt med 19 toner. För att till fullo kunna tillgodogöra sig texten under *3 Utförande* bör läsaren ha kunskap inom programmering och talang för att förstå främmande programmeringsspråk, eftersom stor del av förförandet handlar om interaktionen mellan språken Java och ChucK medelst OSC. I synnerhet Java förväntas läsaren vara bekant med, medan de mest grundläggande konstruktionerna i ChucK förklaras i *3.2.3 Grunder*.

Detta dokument kräver dock ingen programmerare för att kunna läsas, utan kan till stor del även uppskattas av personer med ett intresse som främst ligger inom den musikaliska sfären. För den senare typen av läsare rekommenderas avsnitten *2 Musikteori*, *3.1.1 Kontrollmedel och layout*, *3.4 En utvärdering av 19-TET*, *4 Resultat* och *5 Diskussion*, medan de tekniska bitarna under *3 Utförande* förmodligen kan utelämnas utan större förlust av insikt i projektets slutsatser. En å andra sidan datatekniskt intresserad läsare bör uppskatta i princip hela avsnittet *3 Utförande*, medan avsnittet *2 Musikteori* förmodligen kan skummas igenom utan att gå miste om insikter i den tekniska implementationen. Avsnitten *3.1.3 Att använda paint()-metoden*, *3.1.4 Lyssnare* och *3.1.5 Dynamisk storlek* presenterar förhållandevis allmängiltiga tekniker och tricks som använts i konstruktionen av det grafiska gränssnittet, vilka då får ses som intressant kuriosa eller fördjupning för en läsare enbart intresserad av strikt syntrelaterade detaljer.

1.3 Arbetsfördelning

Detta dokument är skrivet i samband med kursen DD143X Examensarbete inom datalogi (grundnivå) på KTH. Eftersom det är en kurs med individuell betygsättning ska det framgå vem som har gjort vad. Arbetet kan delas upp i tre kategorier; sådant som båda har gjort, Anders har gjort och Simon har gjort.

Båda:

- Läst på om musikteori.
- Lärt sig ChucK.
- Skrivit avsnittet *1 Inledning*, samt sammanfattning.

- Korrekturläst hela dokumentet.

Anders:

- Skrivit samtliga avsnitt under *2 Musikteori*, *3.4 En utvärdering av 19-TET*, *4.2 12-TET vs. 19-TET* och *5.3 19-TET i praktiken*.
- Fördjupat sig i musikteori.
- Assisterat i konstruktionen av det grafiska gränssnittet och ljudlogiken.
- Ansvarat för formateringen av dokumentet.
- Utfört utvärderingen av 19-TET.

Simon:

- Skrivit samtliga avsnitt under *3 Utförande* förutom *3.4 En utvärdering av 19-TET*, *4.1 Den digitala synten*, *5.1 Att skapa en synt*, *5.2 Förbättringar av synt* och *5.4 Slutord*.
- Programmerat det grafiska gränssnittet och ljudlogiken.
- Ritat samtliga bilder i dokumentet.

Kapitel 2

Musikteori

För att kunna genomföra detta projekt har en inte helt oansenlig mängd musikteori behövt studeras, inte minst om olika stämningar och tempereringar av musikinstrument liksom olika toners förhållanden, som spelar en betydande roll för vad vi musikaliskt sett uppfattar som rent eller falskt. I syfte att förklara varför den liksväviga tempereringen i tolv toner fått så starkt fäste i dagens västerländska musik och vad som skiljer de olika tempereringarna åt tas musikteoretisk historia såväl som ren fakta upp. Sporadiska ickebeskrivna musikteoretiska termer som dyker upp i det här kapitlet finns att slå upp i *Bilaga A: Ordlista*.

2.1 Toner och skalor

En elementär byggsten inom musikteorin, och i den musikaliska världen överhuvudtaget, är *tonerna*. En ton är ett ljud som har en hörbar frekvens, det vill säga att en människa kan uppfatta att en ton låter "högre" eller "lägre" än en annan ton och således placera ut tonerna relativt varandra på en musikskala. En musikskala, eller i folkmun enbart "skala", är med andra ord en stigande eller sjunkande sekvens av toner och är ett vanligt förekommande begrepp för den som sysslar med musik. Skalbegreppet är förvisso inte avgörande för att lära sig hur ett musikinstrument kan stämmas, men kan vara till nytta i syfte att förstå varför ett piano ser ut som det gör. I det här kapitlet tas främst en skala upp, nämligen den diatoniska skalan, men även den kromatiska skalan nämns vid enstaka tillfälle.

2.1.1 Musiknotation

Den västerländska musiknotationen bygger på den diatoniska musikskalan, som är en skala bestående av sju toner. På ett piano motsvarar de vita tangenterna en diatonisk skala, nämligen C-durskalan, och dess toner, även kallade *stamtoner*, har beteckningarna C, D, E, F, G, A och B. Dessa toner kan sedan sänkas eller höjas med en halvton för att få notation för de övriga fem tonerna. Vid en sänkning läggs ett $\flat$ till efter bokstaven och vid en höjning tecknet $\sharp$, och tonerna får uttalssuffixen

"ss" respektive "iss". En höjning och en sänkning av tonen A skulle således skrivas som $A\sharp$ respektive $A\flat$ och uttalas som "aiss" och "ass". De höjda och sänkta tonerna motsvarar på ett piano de svarta tangenterna.

Om var och en av de sju stamtonerna går att höja eller sänka betyder det att det totalt går att skriva notation för musik med $3 * 7 = 21$ olika toner, men i vår västerländska standardiserade stämning existerar endast tolv olika toner. Hur går detta ihop? Jo, vissa av tonerna är så kallat *enharmoniska*, vilket innebär att vissa olika beteckningar syftar på samma ton. För att ta ett exempel är $C\sharp$ och $D\flat$ faktiskt samma ton, men vilken beteckning som används för varje ton i ett stycke beror på styckets tonart. Detta innebär att det praktiskt nog är möjligt att avläsa vilken tonart ett stycke är i genom att avläsa vilka kombinationer av höjningar och sänkningar som används.

Att vårt notationssystem stödjer fler toner än vad som normalt används är mycket fördelaktigt då det tillåter notation för musik i andra stämningar än den med 12 toner, så länge antalet toner inte överstiger 21. Det går alltså att använda samma notationssystem för ett stycke skrivet för en stämning i 19 toner. Skillnaden är att vissa av tonerna ej längre är enharmoniska – $C\sharp$ och $D\flat$ är då exempelvis inte samma ton. De toner som förlorar sin enharmoni är: $C\sharp/D\flat$, $D\sharp/E\flat$, $F\sharp/G\flat$, $G\sharp/A\flat$ och $A\sharp/B\flat$, medan de nya toner som tillkommer är: $B\sharp/C\flat$ och $E\sharp/F\flat$.

2.2 Harmonier och intervall

Hur kommer det sig att vissa toner klingar fint tillsammans medan andra matchar varandra uselt? Det är förstås till viss del subjektivt vad som låter bra och inte, men redan omkring år 500 f. Kr. insåg den grekiske filosofen och matematikern Pythagoras att det fanns speciella matematiska förhållanden mellan toners ljudfrekvenser som i större utsträckning var mer angenäma att lyssna på än andra. Han upptäckte att toner vars förhållanden kan skrivas som en kvot av två små heltal generellt sett låter bra om de spelas samtidigt, en upptäckt som varit av stor vikt för hur kommande musikinstrument har utformats[2].

2.2.1 Konsonans och dissonans

De musikteoretiska termerna för att toner passar eller inte passar ihop är att de tillsammans kan låta *konsonanta* eller *dissonanta*[3]. Sett till det totala antalet existerande toner, som är oändligt då frekvensstegen i verkligheten inte är diskreta utan kontinuerliga, existerar det långt många fler dissonanser än konsonanser. Det är i själva verket inte särskilt märkligt eftersom en konsonant harmoni av toner som tidigare beskrivits har ett frekvensförhållande som kan skrivas som en kvot av två små heltal, och antalet sådana är relativt få.

De intervall som existerar i vår tolvtonsbaserade stämning omfattas av den kro-

matiska skalan. Till skillnad från den diatoniska skalan är i den kromatiska skalan varje skalstegs intervall detsamma, nämligen en halvton. Intervallen går, från primen som är förhållandet 1:1 till oktaven som är 2:1, under benämningarna:

1:1 prim

16:15 liten sekund

9:8 stor sekund

6:5 liten ters

5:4 stor ters

4:3 kvart

45:32/64:45 tritonus

3:2 kvint

8:5 liten sext

5:3 stor sext

9:5 liten septim

15:8 stor septim

2:1 oktav

Vilka av dessa intervall som är konsonansintervall är som sagt en subjektiv upplevelse, men av de intervall som är strikt större än primen och mindre eller lika med oktaven anses följande sju vara konsonanta av många[4]:

2:1 oktav

3:2 perfekt kvint

4:3 perfekt kvart

5:3 stor sext

5:4 stor ters

6:5 liten ters

8:5 liten sext

Ett intervall som i sammanhanget är lite speciellt är oktaven. En oktav är, som listan ovan antyder, intervallet mellan två toner vars frekvensförhållande är 2:1, det vill säga att den ena tonen har dubbelt så hög frekvens som den andra. Det visar sig att vi människor upplever två toner som förhåller sig en oktav ifrån varandra som lika, så om en ton har frekvensen 440 Hz och en annan ton har frekvensen 880 Hz kommer dessa två uppfattas som samma ton. Även en tredje ton på 1760 Hz skulle upplevas likt de övriga två. Det är alltså inte den absoluta skillnaden i frekvens som är av intresse för människors perception av ljud, utan den relativa skillnaden. Av denna anledning är det ointressant att studera förhållanden och toner som ligger utanför oktaven.

2.3 Temperering

För att ett musikinstrument ska låta bra när man spelar på det behöver det justeras. Man talar om att man *stämmer* instrumentet och det kan röra sig om att ställa in spänningen hos en gitarr- eller pianosträng eller tublängden hos ett blåsinstrument. Den praxis som tillämpas vid stämning kallas temperering och definierar vilka ljudfrekvenser ett instrument ska kunna frambringa när man spelar på det. Det finns dock många olika tempereringar och vilken som ska användas har inte alltid varit självklart. Vi har idag åtminstone i västvärlden till största del valt att hålla oss till en temperering kallad *liksvävig temperering*, men på vägen dit har ett flertal olika tempereringar provats.

Innan den liksväviga tempereringen infördes på 1400-talet stämdes huvuddelen av musikinstrumenten efter Pythagoras tonförhållanden för att låta så konsonanta som möjligt, en stämning som lämpligtvis fått namnet *ren stämning*. Detta sätt att stämma ett instrument medför att instrumentet måste stämmas i en bestämd tonart – försöker man spela på instrumentet i en annan tonart är tonerna plötsligt inte alls konsonanta. Detta är särskilt framträdande på instrument där intonationen är fast, såsom piano och gitarr. På instrument med fri intonation, exempelvis en violin, kan en musiker i realtid höja eller sänka en ton i valfritt intervall för att kompensera för tonartsbytets bieffekter.

2.3.1 Liksvävig temperering

Kruxet, och anledningen till att ett instrument stämt i ren stämning låter mer eller mindre konsonant beroende på aktuell tonart, är att tonernas förhållanden ändras beroende på vilken funktion de har i ett stycke[5]. Exempelvis är tonen G kvint om grundtonen är C, men om grundtonen är E är den istället stor ters. Med ren stämning ska den ha olika frekvens beroende på om den ska agera kvint eller ters. Man förstod att det skulle vara ypperligt om oktaven kunde delas upp i ett antal lika stora delar så att förhållandet mellan varje halvton skulle vara detsamma. Det skulle innebära att tonarten inte skulle spela någon roll. Men var detta möjligt? Intervallen skulle i så fall inte låta helt konsonanta utan vara en kompromiss mellan

konsonans och funktionalitet.

Det visar sig att det är fullt möjligt och är faktiskt vad vi i västvärlden idag är vana vid att höra. I den liksväviga tempereringen delas oktaven upp i en geometrisk serie av multiplikationer, vilket betyder att det minsta intervallet är förhållandet:

$$r^n = p$$
$$r = \sqrt[n]{p}$$

Där n är antalet toner oktaven är uppdelad i. Eftersom det är oktaven, med förhållandet 2:1, som delas upp i lika delar är $p = 2$. Liksvävig temperering i n toner skrivs vanligtvis kort som *"n-TET"* och för enkelhets skull kommer den liksväviga 12- och 19-tonstempereringen, som är de två varianterna av intresse i det här projektet, fortsättningsvis gå under benämningarna *"12-TET"* respektive *"19-TET"*.

Men hur många toner bör oktaven delas upp i? Kan man välja en helt godtycklig siffra? Svaret är nej; i hur stor utsträckning de olika intervallen låter konsonanta beror helt och hållet på hur många toner man väljer att dela upp oktaven i. Den uppdelning som idag kan anses som standard är den i tolv toner, som är den minsta uppdelning som innehåller samtliga sju konsonansintervall listade i avsnitt *2.2.1 Konsonans och dissonans* där den största avvikelsen från det rena intervallet är mindre än 1%. Den minsta intervallkvoten är i 12-TET $r = \sqrt[12]{2} = 1,059$.

Vanligtvis mäts dock intervall i så kallade *cent*, vilket är en enhet som delar upp oktaven i 1200 lika intervall där varje intervall kallas för en cent. För att beräkna storleken på ett intervall mellan två ljudfrekvenser a och b används formeln:

$$n = 1200 * log_2(b/a)$$

Med denna formel kan de olika vanliga intervallens "korrekthet" i förhållande till den rena stämningen jämföras mellan 12-TET och 19-TET för att få en uppfattning om hur väl de står sig mot varandra.

Namn på intervall	Storlek, 12-TET (cent)	Storlek, 19-TET (cent)	Rent intervall (kvot)	Rent intervall (cent)	Fel, 12-TET (cent)	Fel, 19-TET (cent)
Prim	0	0	1/1	0	0	0
Liten sekund	100	126,32	16/15	111,73	-11,73	14,59
Stor sekund	200	189,47	9/8	203,91	-3,91	-14,44
Liten ters	300	315,79	6/5	315,64	-15,64	+0,15
Stor ters	400	378,95	5/4	386,31	+13,69	-7,36
Kvart	500	505,26	4/3	498,04	+1,96	+7,22
Tritonus	600	568,42	7/5	582,52	+17,49	-14,09
Kvint	700	694,74	3/2	701,96	-1,96	-7,22
Liten sext	800	821,05	8/5	813,69	-13,69	+7,36
Stor sext	900	884,21	5/3	884,36	+15,64	-0.15
Liten septim	1000	947,37	7/4	968,83	+31,17	-21,46
Stor septim	1100	1073,68	15/8	1088,27	+11,73	-14.59
Oktav	1200	1200	2/1	1200	0	0

Tabell 2.1. En jämförelse av de olika intervallen i 12-TET med motsvarande i 19-TET och ren stämning.

Som synes i *Tabell 2.1* är både intervallen i 12-TET och 19-TET relativt nära den rena stämningen. Kvinten i 12-TET ligger nära den rena kvinten med en avvikelse på endast $-1,96$ cent. I 19-TET är den lilla tersen närmast ren och har ett fel på blygsamma $+0,15$ cent. Båda tempereringar täcker dock i det stora hela in de sju konsonansintervallen med relativt stor träffsäkerhet vilket gör 19-TET såväl som 12-TET till lämpliga uppdelningar av oktaven.

2.4 Pianots utseende

En intressant fråga som uppstår när det nya instrumentet ska designas är varför en synt, eller ett piano, egentligen ser ut som det gör. Som många känner till är det uppbyggt av ett antal vita och svarta tangenter, där de svarta kommer i grupper om varannan två tangenter och varannan tre. Varför är de placerade på detta vis och vilka förändringar kommer en oktav med 19 toner att innebära?

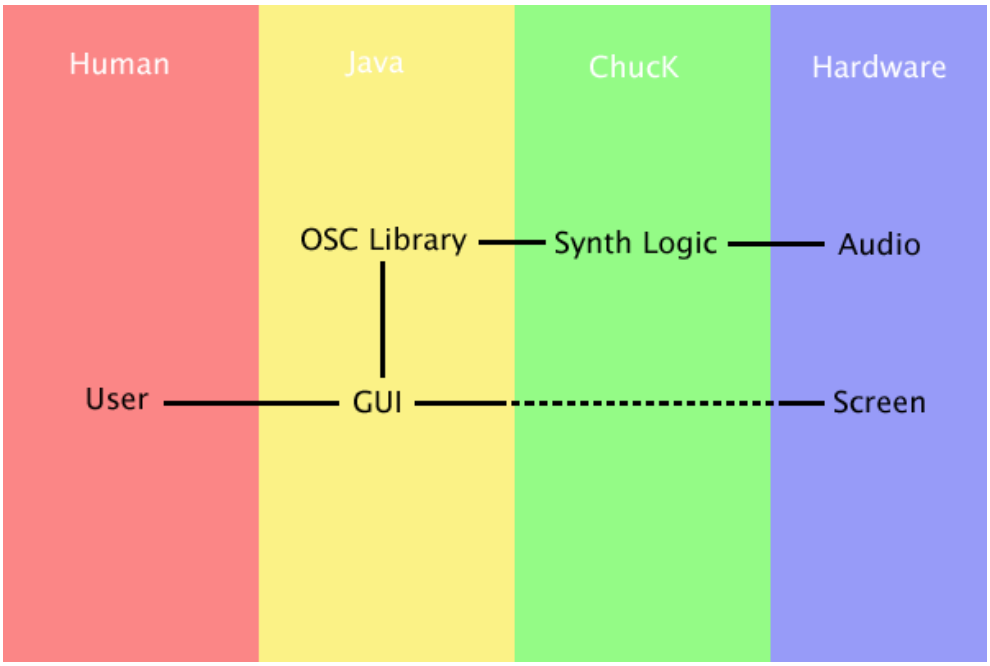
Som nämns i avsnitt *2.1.1 Musiknotation* motsvarar de vita tangenterna på ett piano den diatoniska C-durskalan. I en diatonisk skala är inte intervallen mellan varje ton detsamma utan består av fem hela toner och två halvtoner i samma mönster som de svarta och vita pianotangenterna utgör: T-T-S-T-T-T-S, där T betecknar en hel ton och S en halvton. Intervallet mellan två vita tangenter som har en mellanliggande svart tangent är således en hel ton (lika med två halvtoner) och de som saknar en svart tangent en halvton.

Eftersom ett piano vanligtvis är stämt i och anpassat för 12-TET bör någon form av justering av dess layout göras för att det ska vara praktiskt att spela på i 19-TET. Den senare tempereringen har sju toner fler än den föregående och således bör ett piano för 19-TET ha sju extra tangenter per oktav jämfört med ett vanligt piano. Hur de nya tangenterna bör placeras in är ej helt självklart men ett förslag på hur detta kan göras kan läsas i avsnitt *3.1.1 Kontrollmedel och layout*.

Kapitel 3

Utförande

Tidigt var planen att använda programmeringsspråken Java och ChucK för att bygga synten. Det upptäcktes att man kunde kommunicera språken sinsemellan medelst OSC. Ett försök att från Java anropa funktionalitet i ChucK med hjälp av OSC gjordes, i syfte att få bekräftat att ett grafiskt gränssnitt i Java skulle vara möjligt samtidigt som ljudlogiken för synten är skriven i ChucK. Detta lyckades och instrumentets utformning kunde börja designas. *Figur 3.1* visar ur ett fågelperspektiv hur synten har strukturerats.



Figur 3.1. Visar hur synten är logiskt strukturerad. En användare interagerar med det grafiska gränssnittet som visas på datorns skärm och det grafiska gränssnittet kommunicerar med ljudlogiken med hjälp av ett OSC-bibliotek. Ljudlogiken använder sig i sin tur av datorns högtalarsystem för att kunna spela upp ljud.

3.1 Det grafiska gränssnittet

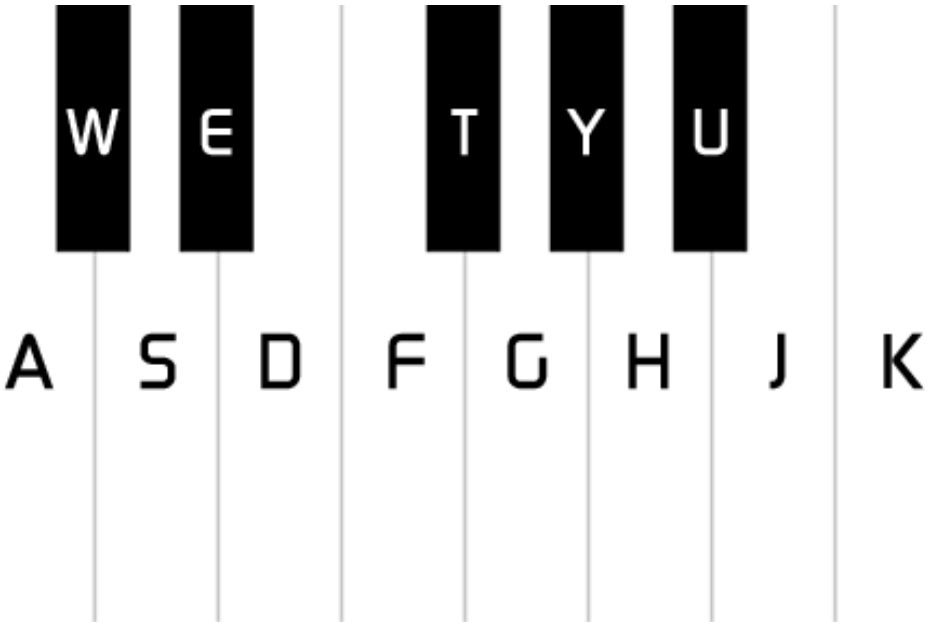
Gränssnittet är byggt i Java med hjälp av Swingbiblioteket. Anledningen till att dessa två valdes är för att Java är ett språk som behärskas väl och erfarenhet av att jobba med Swing var befintlig. På så sätt kunde en prototyp för det grafiska gränssnittet snabbt och effektivt tas fram.

Fort nåddes insikten att en användare behövde mer än bara ljudlig återkoppling när en tangent trycktes ned, varför det beslutades att en tangent ska lysas upp samtidigt som den är nedtryckt. En grafisk återkoppling för användaren hjälper även henne att lära sig syntens layout i förhållande till kontrollmedlet. Dessutom måste möjlighet till att växla mellan en musikskala med 12 toner och en med 19 toner erbjudas, eftersom det annars blir svårt att jämföra de två skalorna. Detta kräver förstås att utseendet för synten förändras, men hur bör synten se ut i vardera läge och hur bör synten kontrolleras?

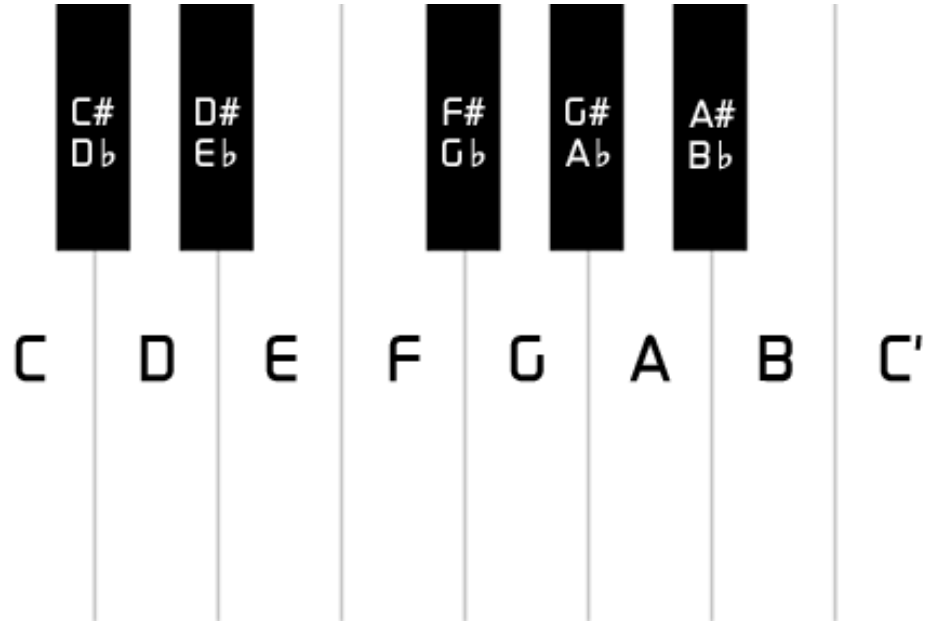
3.1.1 Kontrollmedel och layout

I ett tidigt skede valdes datorns tangentbord som främsta medel och musen som sekundärt medel för att interagera med synten, eftersom interaktionen då blir lätt att programmera tack vare inbyggt stöd i Java. Därtill fanns ingen tillgång till extern enhet att koppla till datorn för att kontrollera synten. Även om en sådan hade funnits tillgänglig, skulle den förmodligen inte vara anpassad efter en skala med 19 toner, så några alternativ fanns det egentligen aldrig.

Det knappa antalet tangenter på tangentbordet och dess placeringar inför dock begränsningar för hur användargränssnittet kan designas. Till exempel blir det omöjligt att försöka imitera ett komplett piano, då antalet tangenter i en följd på en rad på tangentbordet inte är tillräckligt många. Vad man kan göra är att blott representera en oktav och sedan ge användaren möjlighet att höja respektive sänka oktav. Nackdelen med denna lösning är att det blir svårt för användaren att spela två eller fler toner ur olika oktaver samtidigt, men någon alternativ lösning som samtidigt kan ge en naturlig och lättanvänd placering av syntens tangenter på tangentbordet finns inte. Användbarhet i en oktav har alltså premierats framför möjligheten att spela i flera oktaver samtidigt. *Figur 3.2* och *Figur 3.3* visar vilka tangenter som motsvarar vilka toner.



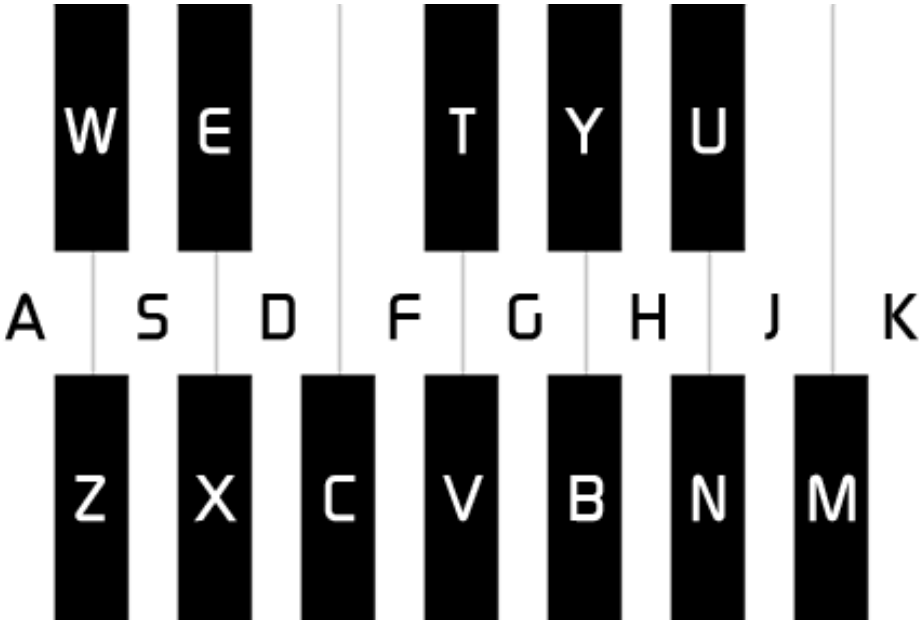
Figur 3.2. Visar hur tangenter på tangentbordet svarar mot tangenter på en 12-tonssynt.



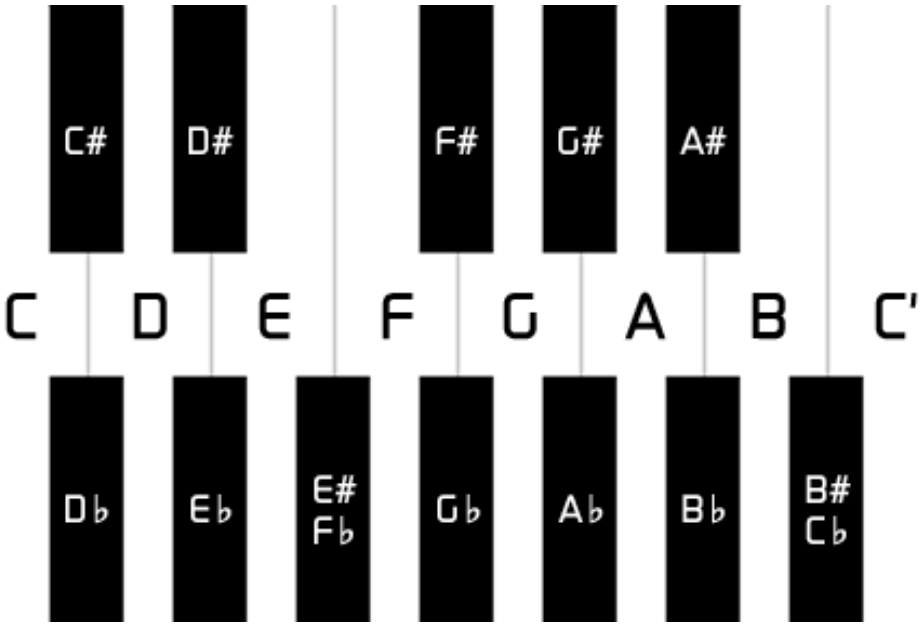
Figur 3.3. Visar namnet på tonen varje tangent på 12-tonssynten svarar mot. C' betyder tonen C i nästa oktav.

Hur tangenterna är placerade på ett vanligt piano känner de flesta till och det finns ingen anledning att avvika från denna norm när synten befinner sig i 12-

tonsläge, men hur bör tangenterna placeras när synten befinner sig i 19-tonsläge? Sebastian Sjögren, som har gjort ett arbete inom samma område (en nittontonssynt) fast med större fokus på hur ett lämpligt GUI bör designas, kom fram till att en lämplig layout är att även placera svarta tangenter längs syntens underkant som synes i *Figur 3.4* och *Figur 3.5*[6]. Ett sådant utseende skulle givetvis vara olämpligt för ett verkligt piano, men för en synt med tangentbordet som gränssnitt blir layouten lyckad då det blir lätt att koppla en tangent på tangentbordet till en tangent på synten. Den ter sig även logisk i avseendet att stamtonerna svarar mot vita tangenter, höjningar av stamtonerna (♯-toner) svarar mot svarta tangenter längs överkanten och sänkningar av stamtonerna (♭-toner) svarar mot svarta tangenter längs underkanten. Ett ytterliggare plus för layouten är att en person som är ovan med en 19-tonssynt kan ignorera de undre svarta tangenterna och spela på synten som om det vore en vanlig 12-tonssynt, om än med ett lite annorlunda ljud.



Figur 3.4. Visar hur tangenter på tangentbordet motsvarar tangenter på en 19-tonssynt.



Figur 3.5. Visar namnet på tonen varje tangent på 19-tonssynten svarar mot. C' betyder tonen C i en oktav högre.

3.1.2 Att strukturera ett GUI med Swing

Det grafiska gränssnittet är strukturerat med fem klasser utöver vad som erbjuds av Javas standardbibliotek, klasserna är följande:

SynthFrame En klass som ärver *JFrame* och representerar ett fönster. En instans av klassen innehåller främst en menyrad, samt en area med en synt.

Synth En klass som ärver *JPanel* och representerar en synt. En instans av klassen ansvarar för den grafiska utritningen av synten och interaktionen med denna, samt kommunikationen med den logiska ljudenheten skriven i ChucK. Detta innefattar bland annat uppgiften att hålla reda på samtliga tangenter.

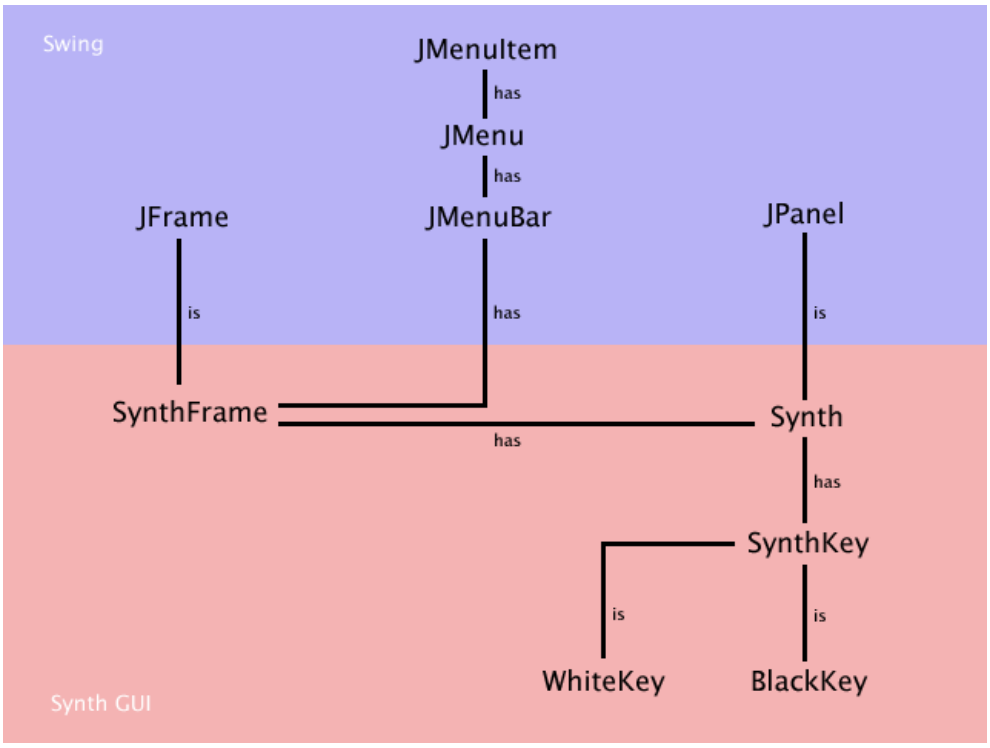
SynthKey En klass som representerar en tangent på synten. En instans av klassen ansvarar för sin egen utritning och håller reda på själv om tangenten är nedtryckt eller ej.

WhiteKey En klass som ärver *SynthKey* och representerar en vit tangent på synten. Klassen erbjuder ingen ny funktionalitet utöver den som ges av *SynthKey*, men är behändig i avseendet att den innehåller färdiga värden för att få en vit tangent.

BlackKey En klass som ärver *SynthKey* och representerar en svart tangent på synten. Klassen erbjuder ingen ny funktionalitet utöver den som ges av *SynthKey*,

men är behändig i avseendet att den innehåller färdiga värden för att få en svart tangent.

En överblick av dessa klasser och hur de använder sig av Javas Swingkomponenter visualiseras i *Figur 3.6*.



Figur 3.6. Visar hur Javaklasserna förhåller sig till varandra.

3.1.3 Att använda paint()-metoden

Anledningen till att klassen *Synth* ärver *JPanel* är för att det är förhållandevis smidigt att rita i en *JPanel*. Allt man behöver göra är att överlagra metoden *paint()* och när man sedan vill uppdatera skärmen anropar man *repaint()*. En annan fördel med att rita i en Swingkomponent såsom *JPanel* är att den har inbyggd dubbelbuffring, till skillnad från lite äldre och mer traditionella komponenter för utritning såsom *Canvas*, vilket dock inte är av vikt i detta projekt där skärmen sällan uppdateras mer än en ett fåtal gånger per sekund.

Strukturen för hur *paint()*-metoden är överlagrad i *Synth* är i stort följande:

```
public void paint(Graphics g) {
    Sudda ritytan, ekvivalent med att rita en rektangel av ritytans
        storlek med bakgrundsfärgen.
```

```
    Rita ut synten och dess tangenter.
}
```

För att rita kan metoder som erbjuds av klassen *Graphics* användas. Vad som kanske inte är lika uppenbart är att *Graphics*-objektet som *paint()*-metoden erhåller faktiskt är en instans av subklassen *Graphics2D*, vilket gör att man får tillgång till ett större utbud av funktionalitet för utritning än vad man först kan tro.

3.1.4 Lyssnare

Ett grafiskt gränssnitt är givetvis trevligt, men det är inte mycket värt om man som användare inte kan interagera med det. För att kunna erbjuda interaktion måste man ha ett sätt för att upptäcka när användaren gör något som är tänkt att ge gensvar från applikationen. Detta kan göras med hjälp av en lyssnare.

Det vi är intresserade av i klassen *Synth* är att upptäcka tangenttryck och musklick (vilka kan vara avsedda av användaren för att trycka ned en tangent), för detta behöver vi således använda en tangentlyssnare och en muslyssnare, vilka i Java heter *KeyListener* och *MouseListener* och är bägge interface som *Synth* då bör implementera. Detta innebär att klassen *Synth* måste deklarera en rad metoder, vanligtvis har man en metod per händelse, till exempel en metod för när en tangent trycks ned och en annan metod för när en tangent släpps upp. Metoden för att upptäcka att en tangent trycks ned ser i stort ut så här:

```
public void keyPressed(KeyEvent ke) {
    Ta reda på vilken tangent som tryckts ned.
    Om tangenten svarar mot en tangent på synten, tala om för tangenten
        på synten att den ska bete sig som nedtryckt och uppdatera (rita
        om) skärmen.
}
```

Där objektet av klassen *KeyEvent* som fås som inparameter bland annat innehåller information om vilken tangent som tryckts ned.

När interfacet implementerats och metoderna deklarerats, måste vi slutligen tala om vilken komponent lyssnaren ska lyssna efter handlingar på, vilket Swingkomponenter erbjuder metoder för. En nackdel med interface, i synnerhet sådana som gäller för lyssnare, är att man ofta kan behöva deklarera massa metoder man inte behöver. För mushandlingar är vi till exempel bara intresserade av musnedtryck och musuppsläpp, men *MouseListener* innehåller en mängd andra metoder som vi är ointresserade av. Vad man kan använda då istället är en Adapter, vilket är en abstrakt klass som implementerar ett lyssnarinterface och deklarerar tomma metodkroppar för samtliga metoder i motsvarande lyssnare. Så istället för en *MouseListener* kan vi använda en *MouseAdapter* och bara specificera de metoder vi är intresserade av. Detta kan med fördel göras i samma veva som lyssnaren (adaptern) läggs till med en så kallad *anonym klass*.

```
addMouseListener(new MouseAdapter() {
    Deklarera och specificera de metoder du är intresserad av i
        MouseListener.
});
```

Slutligen måste vi också fånga upp när användaren väljer menyval i *SynthFrame*. Detta kan göras med en *ActionListener*, men vi går inte djupare in på den komponenten ty det är analogt med vad vi tidigare har redogjort för. Vad som däremot kan vara värt att nämna är att *SynthFrame* ibland kan stjäla fokus från *Synth*, vilket innebär att alla knapptryckningar och musklick användaren gör fångas upp av *SynthFrame* och triggar således inte motsvarande lyssnarmetod i *Synth*, med konsekvensen att programmet upplevs som trasigt. Detta kan man råda bot på genom att anropa metoden *requestFocus()* för instansen av *Synth* i slutet av konstruktorn för *SynthFrame*. Att göra instansen av *Synth* fokuserbar kan också vara nödvändigt ifall fokus skulle senare förloras. Det vill säga:

```
SynthFrame() {
    ...
    synth.setFocusable(true);
    synth.requestFocus();
}
```

Där synth är en instans av klassen *Synth* som används i programmet.

3.1.5 Dynamisk storlek

En funktion som många grafiska program erbjuder är att kunna ändra storleken för fönstret. Vad man som användare då förväntar sig är att gränssnittet skalar därefter, vilket i detta fall gäller synten. Trots att det är en relativt allestädes funktionalitet är det inte alla programmerare som vet hur man åstadkommer detta i Java på ett lämpligt sätt. Följande kodstycke illustrerar lösningen som används i programmet:

```
addComponentListener(new ComponentAdapter() {
    public void componentResized(ComponentEvent e) {
        Insets is = getInsets();
        synth.setSize(getWidth() - is.left - is.right, getHeight() -
            menu.getHeight() - is.top-is.bottom);
    }
});
```

Koden är placerad i *SynthFrame*, så genom att fråga efter fönstrets storlek när det ändrats och storleken för dess menyrad samt indrag, kan man beräkna storleken för arean där själva synten huserar. En fråga man måste ta ställning till när man ändrar storleken för en grafisk komponent och dess subkomponenter är huruvida man ska förändra den logiska storleken eller bara skala om utritningen. Den förstnämnda lösningen är en aning omständlig då man i viss mån blir tvungen att förändra det

logiska beteendet, vilket förstås är betingat med sidoeffekter. Därför har en omskalning av utritningen valts.

Omskalning kan göras överst i *paint()*-metoden med följande rad kod:

```
((Graphics2D) g).scale(scaleX, scaleY);
```

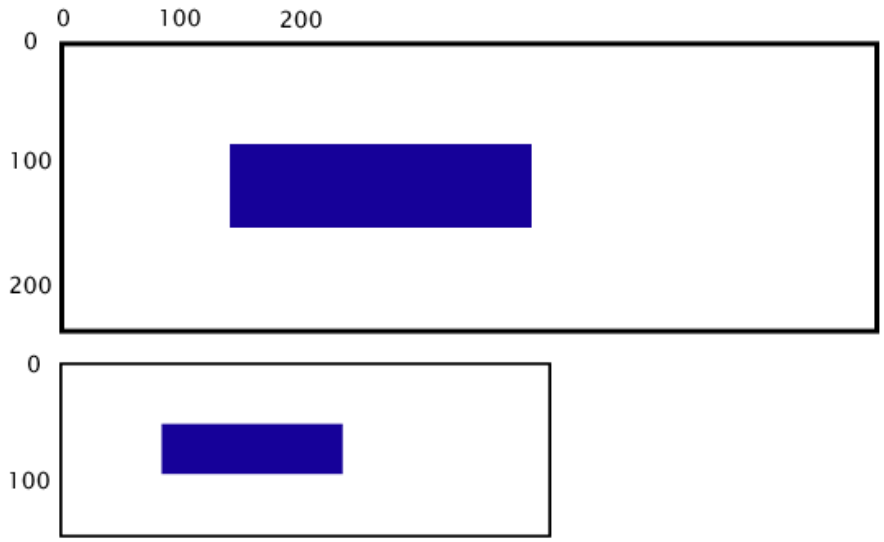
Där flyttalen `scaleX` och `scaleY` anger hur mycket utritningen ska skalas i sidrespektive höjdled.

Att skala om utritningen får dock konsekvenser för hur man bör tolka koordinater. Låt säga att en användare har klickat med musen, detta genererar ett *event* som fångas av programmet ur vilket koordinaterna för musklicket kan utläsas. Dessa koordinater ges absolut mätt från fönstrets övre vänstra hörna. Om vi nu har förminskat fönstret och således skalat om utritningen, men samtidigt behåller den logiska storleken och positionen (så som den var i ursprungsläget), måste vi skala om koordinaterna för att veta om musklicket till exempel träffar en tangent på synten.

Omskalningen av musklickets koordinater kan göras på följande sätt:

```
public void mousePressed(MouseEvent me) {
    final int x = (int) Math.round(me.getX() / scaleX), y = (int)
        Math.round(me.getY() / scaleY);
}
```

Figur 3.7 illusterar problematiken, där den övre rektangeln kan betraktas som fönstret i normal storlek (den logiska storleken) och den undre som fönstret i förminskad form (vad användaren för tillfället ser). Den blå rektangeln representerar synten.



Figur 3.7. Ett musklick kring position (105,60) i det undre fönstret torde träffa den blå rektangeln, men logiskt sett (övre fönstret) ligger inte punkten (105,60) innanför rektangeln, varför den behövs skalas om för att stämma överens med användarens verklighetsuppfattning.

3.2 Ljudprogrammering med ChucK

I detta kapitel beskrivs hur syntens ljudlogik konstruerats. Detta har valts att göra i ett programmeringsspråk som heter ChucK. Ljudlogiken har delats upp i två ChucK-program lagrade på varsin fil: en för en musikskala med 12 toner och en för 19 toner. De två programmen är nästintill identiska kodmässigt och fungerar på samma sätt. Av den anledningen går vi bara igenom hur vi konstruerat ljudlogiken för synten ur ett av programmens synvinkel.

3.2.1 Vad är ChucK?

ChucK är ett statiskt och starkt typat imperativt programmeringsspråk framtaget av Ge Wang och Perry R. Cook under 2000-talet för ljudsyntes. Ett program kan skapas med valfri textredigerare och kan sparas med valfri filändelse, men filändelsen ".ck" anses vara standard. Sparade program kan sedan köras av en virtuell maskin som först kommer att kompilera koden till bytekod och sedan exekvera den.

Tack vare den virtuella maskinen stödjer ChucK även så kallad "on-the-fly programming", vilket innebär att man dynamiskt kan starta och avsluta ett eller flera program som körs av den virtuella maskinen. Detta betyder alltså att man kan köra flera ChucK-program samtidigt, vilket vi kommer att se utnyttjas i syntens implementation.

För att köra ett ChucK-program behöver man ha en fil innehållandes källkoden för själva programmet samt ChucKs virtuella maskin installerad på datorn. Programmet kan sedan köras igång via terminalen (kommandotolken i Windows) på följande sätt. `> chuck mittProgram.ck`

3.2.2 Varför ChucK?

Anledningen till att ChucK valdes som programmeringsspråk för ljudlogiken var till stor del på grund av att det rekommenderades av högre makter (Henrik Eriksson), men också för att det är ett språk som är konstruerat för just ljudsyntes och således bör lämpa sig för ändamålet. Efterforskningar gjordes och det insågs att ChucK till stor del påminde om Java och C i syntax, vilket talade för att det inte skulle bli alltför svårt att lära sig. Det verifierades även att ChucK kunde erbjuda den funktionalitet som var nödvändig för att konstruera en synt med 19 toner.

För att skapa en synt med 19 toner är det givetvis bra att använda ett språk med smidiga konstruktioner för att spela upp ljud, men språket måste också kunna erbjuda operationer på relativt låg nivå, som till exempel att bestämma frekvensen på en ljudvåg, eftersom frekvensen för tonerna i en nittontonstempererad skala skiljer sig från den vanliga med tolv. ChucK erbjuder möjlighet till detta med hjälp av så kallade oscillatorer som finns tillgängliga i standardbiblioteket. De mest intressanta oscillatorerna som undersöktes heter:

- SinOsc
- TriOsc
- SqrOsc
- PulseOsc
- SawOsc

De agerar alla som en abstraktion för en generator av en viss typ av ljudvåg vars frekvens kan ställas in. Att de alla konstruerar olika typer av ljudvågor gör att de alla ger ett annorlunda ljud gentemot varandra, men tonen förblir densamma för en viss frekvens vilket betyder att man godtyckligt kan välja en efter egen preferens för att konstruera synten. I den slutgiltiga implementationen av synten har TriOsc valts, just för att den gav den trevligaste ljudbilden.

Oscillatorerna är dock bara en del av vad man i ChucK kallar för *"Unit Generators"* eller *"UGens"*, hädanefter kallat för generatorer, vilket är en samling klasser för ljudkontroll. Tillsammans med ett lagom omfattande standardbibliotek gör de ChucK till ett ypperligt val för ljudlogiken till en synt.

3.2.3 Grunder

Syntaxen i ChucK har som sagt likheter med den i språk som C och Java. De aritmetiska, logiska och bitvisa operatorerna är i stort desamma. Kontrollstrukturer som if/else-satser såväl som while- och for-loopar finns att tillgå. Primitiva datatyper för heltal och flyttal används i stor utsträckning och man kan skapa klasser och objekt. Dessa välkända programspråkskomponenter gör ChucK relativt enkelt att komma igång med för en programmerare som tidigare programmerat i imperativa programmeringsspråk. En sak som dock skiljer sig markant är sättet man gör tilldelningar på.

Tilldelningar görs i ChucK med den så kallade ChucK-operatorn (=>) eller med at-ChucK-operatorn (@=>). Skillnaden är att den förras funktion till stor del kan bero på sammanhanget (en överlagrad operator), medan den senare är en explicit tilldelningsoperator. Vill vi deklarera en int med värdet 1337, en vektor med värdena 1,5 och 2,7, samt en vektor med plats för sju heltal, skriver vi på följande sätt:

```
1337 => int elit;
[1.5, 2.7] @=> float arr[];
int v[7];
```

Vektorer i ChucK har likheten med Java att de initialiseras automatiskt. I exemplet ovan blir till exempel vektorn `v` fylld med nollor. ChucK tar dock detta ett steg längre och låter det även gälla för objekt, där objektet får vissa standardvärden.

ChucK-operatorn används dock inte bara för att göra tilldelningar, utan också för att koppla samman ljudenheter och avancera tid. Ett exempel som visar på ChucK-operatorns mångfasetterade användning är att spela upp ett ljud med frekvens 440 Hz i en sekund, vilket kan göras så här:

```
SinOsc osc => dac;
440 => osc.freq;
1::second => now;
```

Det vi börjar med att göra är att deklarera (och skapa) en variabel `osc` av typen SinOsc. Sedan kopplas den till `dac` (**d**igital to **a**nalog **c**onverter), vilket man kan säga är språkets "Standard Audio Out", det vill säga en abstraktion för datorns högtalarsystem. Vi sätter frekvensen för sinusoscillatorn till 440 Hz och låter sedan tiden avancera med en sekund, under vilken ljud genereras av samtliga enheter anslutna till `dac`[7].

Ljud spelas endast upp i ChucK genom att man explicit låter tiden avancera i programkoden. Att avancera tiden görs som vi redan har sett med ChucK-operatorn. Det finns dock tre olika sätt man kan avancera tiden på; man kan vänta en viss tid, vänta på tidpunkt eller vänta på ett *event*. Att avancera tiden spelar inte bara upp ljud, utan låter också andra trådar (vilka vi ska återkomma till) exekvera.

Vi avslutar detta avsnitt om syntaxen i ChucK genom att visa hur man deklarerar en funktion. Passande nog låter vi det vara en funktion för att spela upp ljud av en viss frekvens, vilket kan ses som en första ansats till en lämplig funktion för synten att använda.

```
fun void playTone(int freq) {
    SinOsc s => dac;
    freq => s.freq;
    256::ms => dac;
    s =< dac;
}
```

Funktionen kan sedan anropas med raden `playTone(440);` om man vill spela upp tonen A i 256 millisekunder. En nyhet i denna funktion är dock sista raden, vilken innehåller den så kallade UnChucK-operatorn (=<), som i detta fall används för att koppla bort sinusoscillatorn som blivit ansluten till högtalarsystemet. Skulle detta inte göras, skulle generatorn fortsätta att generera ljud nästa gång vi låter tiden avancera, vilket inte är önskvärt.

3.2.4 Trådar

Att spela upp ett ljud som beskrivet i föregående avsnitt fungerar galant. Låt oss anta att det är just den funktionen som exekveras när användaren trycker på en tangent på synten. Vad händer då om vi trycker på två tangenter samtidigt? Jo, då spelas först den ena tonen upp och sedan den andra. Alla som har spelat på ett piano och använt ett sådant med fördel förväntar sig nog att två toner ska ljuda samtidigt som två tangenter trycks ned. För att åstadkomma en upplevelse av simultant klingande ljud kan trådar användas, vilka i ChucK kallas för *shreds*.

En shred är egentligen en process snarare än en tråd om man ska vara petig, men vi kommer att i fortsättningen referera till shreds som trådar. Värt att veta är att en tråd i ChucK endast kan bli avbruten (och låta andra trådar få köra) genom att avancera tiden, så man behöver som programmerare inte oroa sig över att göra instruktioner atomära eller dylikt. Skulle man dock vilja låta andra trådar få köra, samtidigt som man inte vill avancera tiden, kan man anropa `me.yield()`, där `me` refererar till tråden som för tillfället exekverar.

ChucK har ett föredömligt enkelt sätt att hantera trådar på. Vill man skapa en ny tråd från ett program kan man använda nyckelordet `spork` tillsammans med en funktion på följande sätt:

```
spork ~ myFunction();
```

Detta kommer att exekvera funktionen `myFunction()` i en ny tråd som blir schemalagd för körning omedelbart. Vill vi då spela upp flera toner samtidigt behöver vi

bara exekvera funktionen för att spela upp en viss ton i en ny tråd. Nedanstående kod kommer att spela upp ljud av frekvens 440 Hz (ton A) och 330 Hz (ton E) simultant, det vill säga en perfekt kvart:

```
spork ~ playTone(440);
spork ~ playTone(330);
```

3.2.5 Ljudstyrka

Testar man koden i avsnittet innan kommer dock resultatet att vara allt annat än en fröjd för örat, då den ger upphov till högt brus och oregelbundna missljud. Detta var länge ett mysterium, men av en ren slump upptäcktes att den sammanlagda ljudstyrkan av samtliga oscillatorer kopplade till `dac` aldrig får överstiga 1,0 förutsatt att man inte vill uppleva regelrätt oväsen. Anledningen till detta är att ljudvågornas amplitud adderas på varandra och endast svängningar inom intervallet [-1,1] tas hänsyn till, allt utanför intervallet klipps bort[7].

Ljudstyrka kan i ChucK regleras via parametern `gain`, vilken alla generatorer har och således även alla oscillatorer. Så om vi har kopplat två oscillatorer till `dac` måste vi sätta deras respektive `gain` till 0,5 (eller två andra icke-negativa tal vars summa är mindre än eller lika med 1,0). Att hålla reda på alla oscillatorer kopplade till `dac` och göra individuella inställningar för samtliga kan dock bli ganska besvärligt. Tack och lov finns det en smidigare lösning.

Gain är en standardgenerator i ChucK vars enda uppgift är att förstärka eller försvaga ljud och till vilken andra generatorer förstås kan kopplas. Så istället för att skala om ljudstyrkan för varje enskild generator kan vi koppla samtliga till en *Gain*, justera dess ljudstyrka (som vi hädanefter kommer att referera till som den globala ljudstyrkan) och låta den i sin tur vara kopplad till `dac`.

```
Gain g => dac;
0.5 => g.gain;
SinOsc a => g;
SinOsc b => g;
```

Nu kan vi återigen spela upp ljud på harmoniskt manér och därtill två olika ljud samtidigt. Men vad bör den globala ljudstyrkan sättas till? 0,5 fungerar bra när vi har blott två oscillatorer kopplade till `dac`, men med en synt är det inte helt ovanligt att man vill spela upp tre eller ännu fler toner samtidigt, vilket då skulle innebära åtminstone tre oscillatorer. En lösning som länge användes var att skala om ljudstyrkan efter antalet oscillatorer för tillfället anslutna till `dac` på följande sätt, där variabeln `cnt` håller reda på antalet till `dac` (indirekt) anslutna oscillatorer:

```
fun void playTone(int freq) {
    cnt++;
```

```
    1.0/cnt => g.gain;

    Spela upp ljud likt tidigare.

    cnt--;
    if (cnt > 0) 1.0/cnt => g.gain;
}
```

Ovanstående lösning skalar onekligen till ett godtyckligt antal samtidigt ljudande toner och fungerar i jämförelse med alternativet oväsen föredömligt. Den har dock en icke önskvärd sidoeffekt, nämligen att man upplever ljudet som svagare ju fler toner man spelar samtidigt. Förmodligen finns det ingen annan lösning som klarar av ett godtyckligt antal toner simultant, men låt oss för en stund vara realistiska och se till hur en användare faktiskt kommer att använda synten. En någorlunda seriös användare av synten kommer knappast att frenetiskt fläka sig över hela tangentbordet och på så sätt spela upp dussintals av ljud samtidigt. Så man kan med gott samvete sätta den globala ljudstyrkan till ett värde 1/n där n är det maximala antalet toner man förväntar sig att en vettig användare någonsin kommer att vilja spela på en gång.

Begränsar man synten till en oktav och programmerar mekanismer som förhindrar användaren från att spela exakt samma ton i flera trådar samtidigt (vilket skulle kunna inträffa om man kör igång en ny tråd för varje tangenttryck användaren gör), skulle en global ljudstyrka på $\frac{1}{\textit{antalet tangenter på synten}}$ vara ett säkert val. Att sätta värdet så lågt gör dock att man måste höja volymen avsevärt för att kunna uppfatta ljud från synten, vilket får till följd att ljud från andra ljudkällor på datorn kommer att upplevas som väldigt högljudda. Det är förstås inte heller önskvärt, så en kompromiss måste göras. Man får helt enkelt ta risken att en oseriös användare kan uppleva industriellt gnissel till förmån för en trevligare upplevelse för den gemene användaren. I syntens implementation har värdet 1/8 valts som global ljudstyrka eftersom det, utöver att ha en tvåpotens[1] i nämnaren, är lagom stort samtidigt som det är betryggande litet. Faktum är att de flesta tangentbord inte klarar av att hantera fler än på sin höjd sex simultana tangenttryck[8], så 1/8 är i många fall valt med marginal.

3.2.6 Kontinuerligt ljud

Ett designval man tidigt ställs inför när man konstruerar en synt är vad som ska ske när användaren håller en tangent nedtryckt. Ska synten ljuda kontinuerligt eller likt ett piano klinga av? Till en början var visionen att efterlikna ett piano; lyfts fingret från tangenten slutar tonen ljuda abrupt medan den klingar av om tangenten hålls nedtryckt. Detta visade sig dock vara svårt att åstadkomma på grund av att tangentbord genererar *events* kontinuerligt under perioden en tangent är ned-

[1] Dataloger gillar tvåpotenser.

tryckt, vilket gick stick i stäv med vår tro att ett event genererades när tangenten trycktes ned och ett när den släpptes upp. Detta tillkrånglas ytterligare av att olika tangentbord därtill genererar events på olika sätt. Ett tangentbord på en av KTH:s Ubuntudatorer genererade keypress- och keyrelease-event kontinuerligt, medan en PC med Windows blott genererade keypress-event kontinuerligt och endast ett keyrelease-event när tangenten släpptes på riktigt.

Följande två resultat visar hur keypress- och keyrelease-event genereras när man håller en tangent nedtryckt ett litet tag, samt tiden från det att tangenten började hållas nedtryckt, på en Ubuntudator respektive en PC med Windows.

Ubuntu:

```
KeyPress, 0 ms
KeyRelease, 497 ms
KeyPress, 497 ms
KeyRelease, 529 ms
KeyPress, 530 ms
KeyRelease, 562 ms
```

Windows:

```
KeyPress, 0 ms
KeyPress, 488 ms
KeyPress, 520 ms
KeyPress, 550 ms
KeyPress, 581 ms
KeyRelease, 595 ms
```

Dessa två beteenden kan givetvis lösas om man vill åstadkomma en pianoliknande effekt, till exempel genom att ha en tråd som kontrollerar när en tangent senast trycktes ned respektive släpptes och utifrån det försöka avgöra om den verkligen har tryckts ned eller släppts. Det kändes dock som en osäker och omständlig lösning som förmodligen skulle vara betingad med implementationssvårigheter. Följaktligen gavs visionen om en pianoimitation upp och en kontinuerligt ljudande ton valdes.

För att åstadkomma upplevelsen av att en ton ljuder kontinuerligt medan motsvarande tangent på synten är nedtryckt och förhindra att exakt samma ton spelas upp av två olika trådar, användes följande idé: Istället för att starta en ny tråd varje gång en ton ska spelas upp, skapa alla trådar från början, en för varje ton. Låt trådarna ligga och vänta tills huvudtråden som lyssnar på kommandon från GUI:t säger åt dem att sätta igång att spela upp ett ljud, vilket kan göras genom att använda standardklassen *Event*.

Vidare kan man sätta en variabel som får indikera att en ton ska fortsättas att

spelas upp, som ansvarig tråd kommer att titta på med jämna mellanrum. För enkelhetens skull låter vi också tråden återställa variabeln i samma veva, istället för att explicit vänta på att ett sådant kommando anlänt från GUI:t. På så sätt måste GUI:t kontinuerligt säga åt ljudlogiken att fortsätta spela upp en ton, vilket vid första anblick kan tyckas vara ett korkat designbeslut, men som fungerar väl tillsammans med hur tangentbord i allmänhet beter sig. Det hela kan då få följande utseende:

```
int freq[20]; // en array med frekvenser
Initialisera freq.

int isPressed[20];
Event e[20];

for (0 => int i; i < 20; i++)
    spork ~ play(i);

while (true) {
    Vänta och ta emot kommando för en ton i.
    1 => isPressed[i];
    e[i].signal();
}

fun void play(int i) {
    TriOsc s => g;
    0 => s.freq;

    while (true) {
        e[i] => now;
        freq[i] => s.freq;
        while(isPressed[i] > 0) {
            0 => isPressed[i];
            128::ms => now;
        }
        0 => s.freq;
    }
}
```

Det fina med funktionen signal är att den bara signalerar väntande Eventobjekt, så inga signaleringar kommer att läggas i en kö för Eventobjektet att behandla och vi undviker på så sätt att spela upp tonen en gång för varje keypress-event som användaren genererar (vilket är många om en tangent hålls inne).

Ett knep som används för att få tyst på oscillatorn och se till att den inte bygger på ljudstyrkan medan Eventobjektet väntar, är att sätta frekvensen till 0. Detta

fungerar dock inte för alla oscillatorer fullt ut; de är alla förvisso tysta vid frekvens 0, men vissa kommer fortsätta att bygga på ljudstyrkan, så om man vill vara på den säkra sidan bör man även sätta oscillatorns `gain` till 0.

3.2.7 Avklingning

Att man inte låter ljudet klinga av när tangenten hålls inne hindrar dock inte en generell avklingning av en tons ljud. Lägger vi till detta efter att en ton har spelats upp klart upplever användaren att tonen klingar av när tangenten släpps, vilket förhöjer syntens helhetsintryck.

En avklingning i ChucK kan man åstadkomma genom att gradvis sänka ljudstyrkan, vilket kan göras på följande sätt:

```
1.0 => float tmp;
while(tmp > 0) {
    tmp - 0.0078125 => tmp => s.gain;
    2::ms => now;
}
```

Där värdet man sänker ljudstyrkan med för varje iteration lämpligtvis är resultatet av en division av 1 med ett heltal, eftersom ljudstyrkan då slutar på 0. I detta fall är $1/128 = 0,0078125$, vilket gör att avklingningen varar i 256 millisekunder.

Man får dock inte glömma att återställa ljudstyrkan för generatorn s till 1,0 innan man ska spela upp en ton på nytt, annars kommer inte något ljud höras. Detta kan göras antingen innan ljudet ska spelas upp eller efter avklingningen gjorts. Till en början placerades raden `1.0 => s.gain` innan ljudet skulle spelas upp.

Avklingningen fungerar fint, men nu har ett subtilt problem uppstått: varje gång man spelar upp en ton hörs ett svagt knaster. Är det det avklingningen som orsakar detta? Det skulle man kunna tro, men följande två observationer gav en annan insikt:

- Första gången tonen spelas upp hörs inget knaster.
- Hålls en tangent nedtryckt och ljudet spelas upp kontinuerligt, hörs knastret endast ögonblicket tangenten trycks ned, inte när den släpps.

Först kunde man misstänka att det var ChucK som hade vissa instabila tendenser, men sedan upptäcktes att om man ändrar ljudstyrkan drastiskt, i detta fall från 0 till 1 direkt, uppstår ett litet knaster. Anledningen till att det inte hörs första gången är att generatorns ljudstyrka är 1 från början, så första gången sker ingen förändring. Att knastret hördes när man tryckte ned tangenten berodde helt enkelt på att det var då ljudstyrkan sattes. För att råda bot på detta måste ljudstyrkan höjas gradvis, vilket kan göras på följande sätt:

```
s.gain() => float tmp;
while (tmp < 1) {
    tmp + 0.03125 => tmp => s.gain;
    1::samp => now;
}
```

Frekvensen för generatorn s bör givetvis vara satt till 0 under den gradvisa förändringen, så att inget ljud hörs under tiden höjningen görs. Observera att vi måste låta tid passera inuti loopen för att ChucK ska beräkna om ljudet. Denna tid bör dock vara så pass liten att användaren inte märker av loopen. Ett lämpligt värde att flytta fram tiden med är en samp, ty det är den minsta tidsenhet ChucK beräknar om ljudet på, eftersom det motsvarar tiden av en sample. Värdet som ljudstyrkan ökar med bör återigen vara ett resultat av en division av 1 med ett heltal, i detta fall är $0,03125 = 1/32$, så loopen kommer att orsaka en fördröjning på 32 samp. Enligt gjorda undersökningar[2] går det cirka 44 samp på en millisekund, så fördröjningen är i användarens öron omärkbar.

Ett problem är dock att användaren kan trycka på tangenten för att spela upp en ton medan tråden som ansvarar för den tonen befinner sig i avklingningsfasen. Detta är förvisso osannolikt att inträffa, men kan ske om användaren gör två snabba tangenttryck direkt efter varandra. Sker detta kommer användaren att uppleva det andra tangenttrycket som ignorerat, vilket naturligtvis inte är bra då synten gärna ska upplevas som responsiv. Detta kan avhjälpas genom att avbryta avklingningen om `isPressed[i]` blir satt till 1 (sant), öka ljudstyrkan (snabbt) till 1 igen och börja spela om ljudet på nytt. Koden för detta utelämnas här och nu, men kan återfinnas i *Bilaga B: Källkod* för synth12.ck såväl som synth19.ck.

3.2.8 Toner

Slutligen bör det redogöras för hur systemet avgör vilken frekvens som ska spelas upp. Det hela är inte alls komplicerat: De två ChucK-programmen har en vektor med tjugo respektive tretton flyttal som svarar mot frekvenser, en för varje ton och en sista för första tonen i nästa oktav. Positionen för en frekvens i vektorn är densamma som för motsvarande tangent i GUI:ts interna vektor för att lagra tangenter, så när en ton ska spelas upp behöver GUI:t bara skicka indexet för den vidrörda tangenten för att ljudlogiken ska kunna spela upp rätt ljud.

Frekvenserna i vektorn är anpassade på så sätt att tonen A får frekvens 440 Hz, det vill säga att synten är stämd efter normalton. För att ta fram frekvenserna som behövdes för 19- respektive 12-tonssynten användes Haskell och två kommandon som kördes i GHCi:

```
Prelude> map (*440) [2**(fromIntegral i / 19) | i <- [-19..19]]
```

[2] 1::ms/1::samp ger resultatet 44,1.

```
[220.0,228.17410977816226,236.65192896844013,245.44474190753962,
254.56425220219836,264.0225983071827,273.83236968208513,
284.0066235484291,294.5589022693839,305.50325137522435,
316.85423825852695,328.62697156398684,340.8371212986643,
353.50093968942826,366.6352828153596,380.2576330439065,
394.38612230065655,409.0395562036988,424.23743909469783,440.0,
456.3482195563244,473.30385793688026,490.8894838150792,
509.1285044043967,528.0451966143654,547.6647393641703,
568.0132470968582,589.1178045387678,611.0065027504487,
633.7084765170539,657.2539431279737,681.6742425973284,
707.0018793788565,733.2705656307191,760.515266087813,
788.7722446013131,818.0791124073976,848.4748781893957,880.0]

Prelude> map (*440) [2**(fromIntegral i / 12) | i <- [-12..12]]
[220.0,233.08188075904496,246.94165062806206,261.6255653005986,
277.1826309768721,293.6647679174076,311.1269837220809,
329.6275569128699,349.2282314330039,369.9944227116344,
391.99543598174927,415.3046975799451,440.0,
466.1637615180899,493.8833012561241,523.2511306011972,
554.3652619537442,587.3295358348151,622.2539674441618,
659.2551138257398,698.4564628660078,739.9888454232688,
783.9908719634985,830.6093951598903,880.0]
```

Ur vilka de markerade intervallen [264.022..., 528.045...] och [261.625.., 523.251...] som svarar mot en oktav i 19- respektive 12-tonsskalan med C som första ton kunde extraheras. Dessa frekvenser fick sedan kastas om för att matcha ordningen av tangenterna på synten.

Vill man sedan höja tonen med en oktav behöver man bara gå igenom vektorn och multiplicera varje frekvens med två. Att sänka en oktav görs på samma sätt, fast man dividerar varje frekvens med två.

3.3 Att kommunicera med OSC

För att förmedla de instruktioner som Javagränssnittet snappar upp från användaren till ljudlogiken skriven i ChucK används OSC. Kommunikationen är enkelriktad och ser ut på så sätt att gränssnittet helt sonika säger åt ljudlogiken vad för ljud som ska spelas upp eller om en oktav ska höjas eller sänkas. Vad som då krävs av Javadelen (det grafiska gränssnittet) är att kunna skicka OSC-meddelanden, medan ChucK-delen (ljudlogiken) bara behöver kunna ta emot OSC-meddelanden.

För att få tillgång till OSC-funktionalitet i Java kan man ladda ned biblioteket JavaOSC av Illposed. I biblioteket finns det framförallt två klasser som är av stort

intresse, nämligen OSCPortOut och OSCMessage. Den senare används för att skapa ett meddelande och den förra för att skicka ett. Hur klasserna används kan illustreras med ett kodstycke som är plockat ur programmet, vars syfte är att förmedla en uppmaning till ljudlogiken att spela en ton:

```
OSCPortOut sender = new OSCPortOut(InetAddress.getLocalHost(), 6449);

private void sendOSCMsg(int tone) {
    OSCMessage msg = new OSCMessage("/synth", new Object[]{tone});
    sender.send(msg);
}
```

Där man först måste specificera var meddelandet ska skickas, vilket är den egna datorn och en i förväg utvald port som ljudlogiken lyssnar på. Sedan måste man specificera namnet på meddelandet och eventuell medföljande data. Slutligen kan meddelandet skickas.

För att lyssna efter OSC-meddelanden i ChucK kan de inbyggda klasserna OscRecv och OscEvent användas. Hur dessa används är inte helt trivialt, så detta visas med ett kommenterat stycke kod, vars syfte är att fånga upp meddelanden som skickats med Javakoden ovan.

```
// skapa en OSC-mottagare
OscRecv orec;
// koppla mottagaren till port 6449
6449 => orec.port;
// börja lyssna (en tråd körs igång i bakgrunden)
orec.listen();

// specificera ett event för ett meddelande med namn "/synth" och en
// medföljande int som data
orec.event("/synth, i") @=> OscEvent event;
while (true) {
    event => now; // Vänta på meddelanden.

    // behandla de meddelanden som kommit
    while (event.nextMsg() != 0) {
        // plocka ut indata ur meddelandet
        event.getInt() => int tone;
        <<< "Tog emot ", tone >>>;
    }
}
```

Den kodrad som framförallt inte är helt intuitiv är den innehållandes `orec.event("/synth, i")`. Det raden gör är att skapa ett event som ska genereras när ett meddelande av ett visst format anländ. Formatet specificeras av en sträng

och är som sådant att först namnet anges och sedan inparametrarna separerade med kommatecken. Typen för en inparameter specificeras med en bokstav, där i betyder int, f betyder float och s betyder string.

Nu är det ju dock som så att vi har en ljudlogik för 12 toner och en för 19 toner som båda körs samtidigt. Hur gör vi då för att särskilja på kommandon till de två? En lösning vore att använda olika portar, men en smidigare lösning är att låta dem lyssna på meddelanden med olika namn. Den ena kan få meddelanden med namn "/synth12" och den andra meddelanden med namn "/synth19". Sedan behöver bara Javadelen av programmet ändra namnet på meddelandena den skickar beroende på om användaren för tillfället har valt en 12- eller 19-tonslayout.

3.4 En utvärdering av 19-TET

När synten är klar och fungerar kan den användas för att evaluera 19-TET och jämföra det med det väl beprövade 12-TET. I undersökningen deltog 11 personer av varierande musikalisk bakgrund; såväl hobbymusiker som personer som aldrig spelat på ett musikinstrument fick delta för att få svar från flera infallsvinklar och variation på resultatet. De fick lyssna på en grupp väl valda konsonansintervall samt ackord från synten i 12- respektive 19-tonsläge och, för varje intervall och ackord, berätta vilken temperering de föredrog. För att minimera risk för att svaren skulle bli influerade av att de visste vilken temperering som var vilken gjordes testet i blindo, med andra ord fick de inte veta vilket läge synten befann sig i när ett visst intervall eller ackord spelades upp.

Deltagarna gavs för varje intervall och ackord alternativen att svara "12-TET", "19-TET" eller "ej avgörbart" och skiftandet mellan syntens två lägen försöktes göras med så liten fördröjning som möjligt för att skillnaden mellan de två tempereringarna skulle uppfattas tydligare. Vidare gjordes undersökningen i syntens standardoktavläge $C4$.

Kapitel 4

Resultat

Resultatet kan delas upp i två delar; dels det färdiga programmet – synten – och dels vad våra testpersoner tyckte om konsonantintervallen i 19-TET i jämförelse med motsvarande intervall i 12-TET.

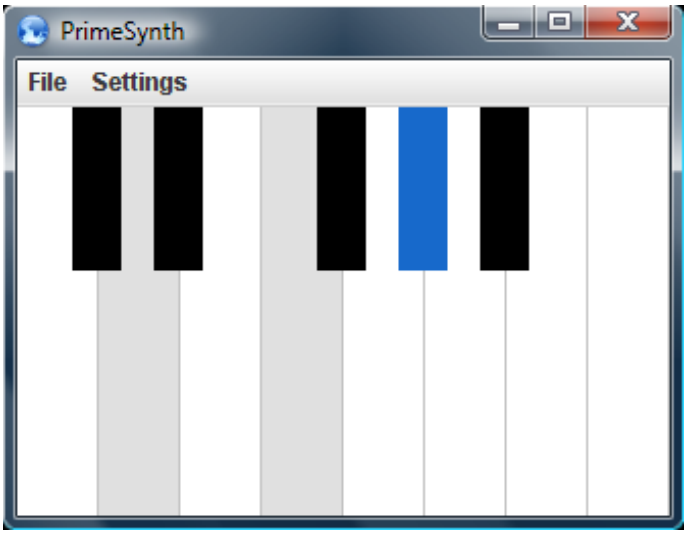
4.1 Den digitala synten

För att kunna köra synten behöver användaren utöver själva programmet ha installerat ChucKs såväl som Javas virtuella maskin på sin dator. ChucKs virtuella maskin finns tyvärr bara tillgänglig som ett färdigt körbart program till Windows, använder man Linux eller Mac måste man kompilera programmet själv. Ett alternativ som har testats för att köra synten på Ubuntu är att köra programmet i Wine.

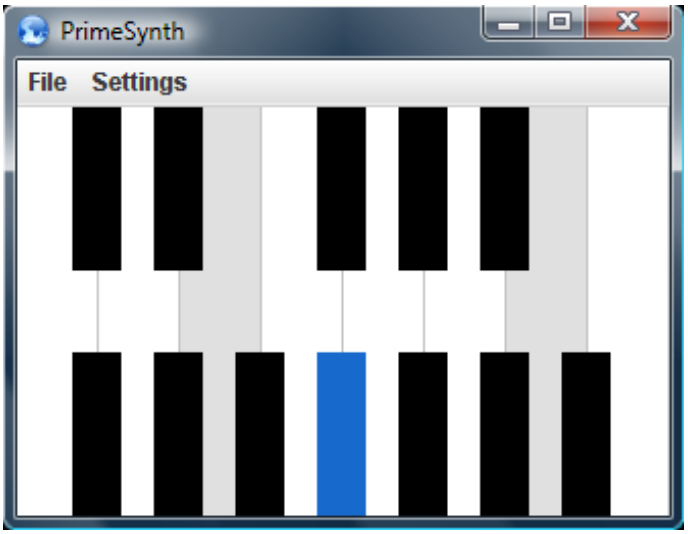
Med alla virtuella maskiner installerade behöver man bara ställa sig i rätt katalog/mapp med terminalen/kommandotolken och köra följande två kommandon för att starta ljudlogiken respektive det grafiska gränssnittet:

```
> chuck synth12.ck synth19.ck
> java SynthFrame
```

Hur synten ser ut när den körts igång kan ses i *Figur 4.1* och *Figur 4.2*. Utöver redan nämnd funktionalitet så som interaktion med tangentbord och mus, kan man under "Settings" slå på/av helskärmsläge, byta bakgrundsfärgen (vilken är den som utgör färgen på strecken som separerar de vita tangenterna), samt växla mellan tolv och nitton toner. För att höja respektive sänka oktav används tangenterna uppåtpil och nedåtpil.



Figur 4.1. Visar hur synten ser ut i 12-tonsläge samtidigt som någon spelar upp tonerna D, F och $G\sharp/A\flat$.



Figur 4.2. Visar hur synten ser ut i 19-tonsläge samtidigt som någon spelar upp tonerna E, $G\flat$ och B.

4.2 12-TET vs. 19-TET

Resultatet av testpersonernas uppfattning av 19-TET i jämförelse med 12-TET presenteras i *Tabell 4.1* och *Tabell 4.2*.

Intervall	12-TET	19-TET	Ej avgörbart
Liten ters	3	**7**	1
Stor ters	3	**4**	**4**
Kvint	**6**	3	2
Liten septim	**5**	**5**	1
Stor septim	**5**	4	2

Tabell 4.1. Visar för varje konsonantintervall i testet hur många personer som favoriserade det i 12-TET respektive 19-TET. Det största värdet för varje intervall är markerat i fetstil.

Ackord	12-TET	19-TET	Ej avgörbart
C	2	**6**	3
Dm	**5**	2	4
Dm7	3	**6**	2
Esus4	**6**	3	2

Tabell 4.2. Visar för varje ackord i testet hur många personer som favoriserade det i 12-TET respektive 19-TET. Det största värdet för varje ackord är markerat i fetstil.

Kapitel 5

Diskussion

5.1 Att skapa en synt

ChucK har fungerat mycket bra för projektets ändamål och samarbetet med Java genom OSC har flutit på fläckfritt. Trådhanteringen i ChucK måste sägas vara föredömlig och konstruktionen att enkelt kunna ange funktionen man vill köra i en ny tråd är något flera språk borde ta lärdom utav. Överlag är språket enkelt att komma igång med och det tar inte mer än 10 minuter innan man har lyckats klämma ut ett ljud ur högtalarna. Blandningen av högnivå med möjlighet till lågnivå är mycket lyckad.

En nackdel med ChucK är att användaren måste installera en virtuell maskin för att kunna köra program skrivna i språket. Detta kan dock sägas om alla språk som förlitar sig på en virtuell maskin, men skillnaden mellan Java och ChucK är att nästan alla har Java installerat på sin dator medan väldigt få har installerat ChucK. Så om vi skulle vilja nå ut med vår synt och sprida nittontonsskalan i världen, skulle vi nog få behöva tänka om. För syftet att jämföra nitton- och tolvtonsskalan samt undersöka möjligheterna för en programmerare att skapa sin egen synt, är det dock inte av större relevans.

Vad som skulle kunna förbättras angående ChucK är framförallt dokumentationen och då i synnerhet i samband med referenslistan för olika generatorer. Hade detta varit fallet hade problem med ljudstyrkan kunna ha undvikits och eventuellt skulle smidigare lösningar till vissa delar av ljudlogiken kunna tas fram. Nu är det dock inte så och förhoppningsvis kommer framtida generationer att kunna undvika dessa problem tack vare de insikter och upptäckter som har redogjorts för i uppsatsen.

5.2 Förbättringar av synt

Skulle möjligheten att bygga om eller förbättra synten ges, skulle ljudlogiken och det grafiska gränssnittet försökas göras mer autonoma och oberoende av varandra.

För i dagsläget är ljudlogiken anpassad efter hur keyevents genereras och hur GUI:t hanterar dessa, samt på vilken position i den interna datastrukturen tangenterna är lagrade. En bättre och mer modulär lösning vore att låta GUI:t hantera problematiken med keyevents och konstruera ljudlogiken på så sätt att den tar emot kommandon för att starta, klinga av och avsluta uppspelningen av en viss ton. Sättet man anger en ton på bör heller inte bero på GUI:ts interna struktur, bättre vore att låta GUI:t skicka en frekvens som ljudlogiken ska spela upp. Tillsammans med denna frekvens, skulle man kunna skicka ett unikt id för att hålla reda på den och på så sätt enkelt kunna identifiera den vid olika kommandon. Detta öppnar även upp för möjligheten att spela upp två ljud av samma frekvens samtidigt, vilket man eventuellt skulle kunna vilja göra för att uppnå en högre ljudstyrka. Dessutom skulle en sådan lösning göra ljudlogiken oberoende av vilken temperering synten är skapad utefter.

Vad gäller funktionalitet att utöka synten med skulle först och främst någon form av grafisk återkoppling behövas när användaren växlar oktav, men även någon form av indikation på vilken oktav man befinner sig i skulle vara användbar. Därtill skulle det även vara trevligt om användaren själv kunde ställa in hur tangenterna på tangentbordet ska svara mot tangenter på synten, då olika användare kan ha olika preferenser.

5.3 19-TET i praktiken

Av resultaten presenterade i *Tabell 4.1* och *Tabell 4.2* att döma blev det så gott som oavgjort mellan de två tempereringarna. 19-TET vann den lilla tersen med marginal medan 12-TET vann kvinten solklart, vilket var ett väntat resultat när man tittar på de två tempereringarnas avvikelser från de rena intervallen i *Tabell 2.1*. Övriga intervall hade inte lika självklara vinnare, men resultaten uppvisade dock tendenser att stämma överens med den musikaliska teorin.

Vilka resultat som förväntades i undersökningen av ackord är något oklart; varje ackord innehåller åtminstone ett intervall som låter bäst i vardera temperering. Till exempel finner man i ackordet C både en stor ters och en kvint. Det förefaller från dessa resultat att inte någon av de två tempereringarna generellt sett är bättre än den andra, men det finns tydliga tendenser till en vinnare i respektive ackord.

Det blotta antalet av elva deltagare, där majoriteten saknade musikalisk bakgrund, ger dock inte resultaten någon större statistisk tyngd. Att uppfatta renhet när blott två toner ljuder kan vara svårt och en person utan musikalisk bakgrund är förmodligen inte bekant med att försöka uppfatta de karaktäristiska svängningar som kan uppstå. Därför kan man misstänka att vissa deltagare upplevde det svårt att höra någon större skillnad på de två tempereringarna, varpå de valde en favorit på måfå. Detta försöktes förvisso förebyggas genom att klargöra för testpersonerna att det

var fullt tillåtet att svara att de inte kunde avgöra vilken som lät bäst. Vidare kan det också vara så att testpersonerna är vana med att höra intervallen i 12-TET och således även vant sig med de bitvis höga felmarginalerna.

Resultatets felkällor till trots visar det tydliga tendenser som därtill stämmer relativt bra överens med teorin, varför det i allra högsta grad kan tas i beaktning. 19-TET bör således betraktas som ett intressant alternativ, om än inte en ersättare, till 12-TET. Därtill finns det andra faktorer än de som tas hänsyn till i undersökningen att beakta:

- Majoriteten av dagens musiker är idag invanda med 12-TET och instrumenten är konstruerade därefter. Ett plötsligt byte av antal toner per oktav skulle innebära att många instrument, såsom pianot, skulle behöva modifieras och troligen bli svårare att spela på.
- Att 19-TET har fler toner, vilket givetvis ger större möjligheter till precision.
- Det är lättare att av misstag spela ett dissonant intervall på ett instrument stämt i 19-TET då det har fler toner och därmed fler potentiella dissonansintervall.

12-TET är förmodligen på det stora hela något bättre än 19-TET, men inget musikstycke är det andra likt och i vissa lägen kan den senare kanske vara det bättre valet. Spelar den lilla tersen en central roll i musikstycket samtidigt som kvinten inte är lika framträdande bör 19-TET förmodligen vara att föredra. Det finns ingen temperering som alltid är bäst, utan det beror på situationen.

5.4 Slutord

Valet att använda ChucK för ljudlogiken ångrar vi inte. Skulle vi börja om och försöka åstadkomma samma ljudlogik i Java, skulle det med stor sannolikhet ta betydligt längre tid än den tid det har tagit att lära sig ChucK och knåpa ihop de få rader som utgör logiken. Förmodligen skulle vi inte ens kunna uppnå ett lika bra resultat inom en överskådlig framtid. Därur följer vår viktigaste lärdom under detta projekt, nämligen att det kan löna sig att programmera delar och komponenter av ett program i ett annat språk än sitt modersmål. Visst Java är mer allsidigt och generellt sett bättre än ChucK på det mesta, men det finns områden där ChucK är att föredra. På samma sätt är 12-tonsskalan i de flesta fall bättre än 19-tonsskalan, men i vissa speciella musikstycken, exempelvis sådana med många tersar, kan den senare erbjuda en angenämare musikupplevelse. ChucK är inte här för att ersätta Java och 19-tonsskalan bör inte ersätta 12-tonsskalan, men de bör bägge tas hänsyn till och användas som seriösa komplement, för ibland kan det vara nödvändigt att nischa sig om man vill uppnå högsta kvalitet.

Litteraturförteckning

[1] Wikipedia. *Arab Tone System* [webbsida på internet]. 2005 [uppdaterad 21 januari 2012; hämtad 7 mars 2012]. Tillgänglig på `http://en.wikipedia.org/wiki/24-TET`

[2] Peter A. Frazer. *Ancient Greek Origins of the Western Musical Scale* [webbsida på internet]. 2010 [hämtad 7 mars 2012]. Tillgänglig på `http://www.midicode.com/tunings/greek.shtml`

[3] Wikipedia. *Consonance and dissonance* [webbsida på internet]. 2005 [uppdaterad 31 mars 2012; hämtad 12 april 2012]. Tillgänglig på `http://en.wikipedia.org/wiki/Consonance\_and\_dissonance`

[4] Keith Enevoldsen. *Twelve-Tone Musical Scale* [webbsida på internet]. 2010 [hämtad 7 mars 2012]. Tillgänglig på `http://thinkzone.wlonk.com/Music/12Tone.htm`

[5] Kyle Gann. *Just Intonation Explained* [webbsida på internet]. 1997 [hämtad 7 mars 2012]. Tillgänglig på `http://www.kylegann.com/tuning.html`

[6] Sebastian Sjögren. *A nineteen tone scale synthesizer* [webbsida på internet]. 2011 [hämtad 12 april 2012]. Tillgänglig på `http://www.csc.kth.se/utbildning/kth/kurser/DD143X/dkand11/Group7Henrik/sebastian.sjogren.report.pdf`

[7] Ge Wang och Perry Cook. *The Chuck Manual*, sida 17-19 [webbsida på internet]. 2007 [hämtad 12 april 2012]. Tillgänglig på `http://chuck.cs.princeton.edu/release/files/chuck\_manual.pdf`

[8] MaxFPS. *Tips inför val av tangentbord* [webbsida på internet]. 2002 [hämtad 12 april 2012]. Tillgänglig på `http://www.maxfps.se/guider/tangentbord.aspx`

Bilaga A

Ordlista

12-TET Den liksväviga tempereringen med tolv toner.

19-TET Den liksväviga tempereringen med 19 toner.

C4 Den fjärde tonen C på ett vanligt piano med 88 tangenter. C4 = 261,63 Hz.

ChucK Ett programmeringsspråk specialiserat på ljudsyntes. dac (digital to analog converter): Abstraktion i ChucK för datorns högtalarsystem.

Event En programmeringskonstruktion för att upptäcka och ta hand om ur ett programs synvinkel yttre händelser.

Frekvens Antalet svängningar per sekund hos en ljudvåg. gain: Parameter som bestämmer en generators ljudstyrka.

Generator (Unit Generator, UGen) Verktyg för ljudkontroll i ChucK.

GHC (Glascow Haskell Compiler) Kompilator och plattform för Haskell.

GHCi Interaktiv miljö för GHC.

Grundton Första tonen i en skala.

GUI (Graphical User Interface) Förkortning för ett grafiskt användargränssnitt.

Halvton Ett intervall av storlek 100 cent. Det minsta intervallet i 12-TET.

Harmoni Användandet av flera toner eller ackord samtidigt.

Haskell Ett funktionellt programmeringsspråk.

Interface Programkonstruktion i Java för att påtvinga en viss struktur hos en komponent.

Intonation Att sätta en ton i rätt tonhöjd i förhållande till andra toner.

Java Ett allsidigt programmeringsspråk.

Keyevent Event som är associerat med ett tangenttryck.

Keypress-event Event som genereras när användaren trycker ned en tangent.

Keyrelease-event Event som genereras när användaren släpper upp en tangent.

Normalton En universellt bestämd frekvens för stämning av musikinstrument, vanligtvis 440 Hz.

OSC (Open Sound Control) Protokoll som används för kommunikation mellan datorer, syntar och andra multimediaenheter.

Oscillator Verktyg för att generera ljudvågor i ChucK.

Renhet Hur nära motsvarande intervall i ren stämning ett intervall ligger.

Sample Ett ljudvärde som datorn använder sig av för att approximera en kontinuerlig ljudvåg under ett (kort) intervall. Flera samples används för att approximera den kontinuerliga ljudvågen i sin helhet med en diskret ljudvåg.

Skalsteg Avståndet mellan två i en skala efterföljande toner.

Stämning Vilka toner ett musikinstrument justerats för att kunna frambringa.

Swing Standardbibliotek i Java för grafiska komponenter.

Temperering Ett sätt att stämma ett musikinstrument där tonernas intervall är kompromissade gentemot dem i den rena stämningen.

Tonart Är i själva verket ett potentiellt komplicerat begrepp men syftar i denna rapport användandet av en viss skala med specificerad grundton.

Vektor Översättning av engelskans “array”. En typ av datastruktur med sekventiell lagring.

Bilaga B

Källkod

B.1 synth12.ck

```
Gain g => dac;
1.0/8 => g.gain;

OscRecv orec;
6449 => orec.port;
orec.listen();

[
261.6255653005986,  // c
293.6647679174076,  // d
329.6275569128699,  // e
349.2282314330039,  // f
391.99543598174927, // g
440.0,              // a
493.8833012561241,  // b
523.2511306011972,  // c
277.1826309768721,  // c#/db
311.1269837220809,  // d#/eb
369.9944227116344,  // f#/gb
415.3046975799451,  // g#/ab
466.1637615180899   // a#/bb
] @=> float freq[];

Event e[freq.cap()];
int isPressed[freq.cap()];

for (0 => int i; i < freq.cap(); i++) spork ~ play(i);

orec.event("/synth12, i") @=> OscEvent event;
while(true) {
    event => now;

    while(event.nextMsg() != 0) {
        event.getInt() => int i;

```

```
36          if (i == -1) {
37              for (0 => int j; j < freq.cap(); j++) 2 *=> freq[j];
38          } else if (i == -2) {
39              for (0 => int j; j < freq.cap(); j++) 2 /=> freq[j];
40          } else {
41              1 => isPressed[i];
42              e[i].signal();
43          }
44      }
45  }
46
47  fun void play(int i) {
48      TriOsc s => g;
49      0 => s.freq => s.gain;
50
51      while (true) {
52          if (!isPressed[i]) e[i] => now;
53
54          //Fast fade-in, avoid crackle. If crackle use lower increment
                .
55          s.gain() => float tmp;
56          while (tmp < 1) {
57              tmp + 0.03125 => tmp => s.gain;
58              1::samp => now;
59          }
60
61          freq[i] => s.freq;
62
63          while (isPressed[i]) {
64              0 => isPressed[i];
65              128::ms => now;
66          }
67
68          //Fade-out.
69          1.0 => tmp;
70          while (tmp > 0 && !isPressed[i]) {
71              tmp - 0.0078125 => tmp => s.gain;
72              2::ms => now;
73          }
74
75          0 => s.freq;
76      }
77  }
```

B.2 synth19.ck

```
1  Gain g => dac;
2  1.0/8 => g.gain;
3
4  OscRecv orec;
5  6449 => orec.port;
6  orec.listen();
7
```

```
[
264.0225983071827,  // c
294.5589022693839,  // d
328.62697156398684, // e
353.50093968942826, // f
394.38612230065655, // g
440.0,              // a
490.8894838150792,  // b
528.0451966143654,  // c
273.83236968208513, // c#
305.50325137522435, // d#
366.6352828153596,  // f#
409.0395562036988,  // g#
456.3482195563244,  // a#
284.0066235484291,  // db
316.85423825852695, // eb
340.8371212986643,  // e#/fb
380.2576330439065,  // gb
424.23743909469783, // ab
473.30385793688026, // bb
509.1285044043967   // b#/cb
] @=> float freq[];

Event e[freq.cap()];
int isPressed[freq.cap()];

for (0 => int i; i < freq.cap(); i++) spork ~ play(i);

orec.event("/synth19, i") @=> OscEvent event;
while (true) {
    event => now;

    while (event.nextMsg() != 0) {
        event.getInt() => int i;

        if (i == -1) {
            for (0 => int j; j < freq.cap(); j++) 2 *=> freq[j];
        } else if (i == -2) {
            for (0 => int j; j < freq.cap(); j++) 2 /=> freq[j];
        } else {
            1 => isPressed[i];
            e[i].signal();
            <<< "Synt 19:", i, freq[i] >>>;
        }
    }
}

fun void play(int i) {
    TriOsc s => g;
    0 => s.freq => s.gain;

    while (true) {
        if (!isPressed[i]) e[i] => now;

```

```
62          //Fast fade-in, avoid crackle. If crackle use lower increment
                .
63          s.gain() => float tmp;
64          while (tmp < 1) {
65              tmp + 0.03125 => tmp => s.gain;
66              1::samp => now;
67          }
68
69          freq[i] => s.freq;
70
71          while (isPressed[i]) {
72              0 => isPressed[i];
73              128::ms => now;
74          }
75
76          //Fade-out.
77          1.0 => tmp;
78          while (tmp > 0 && !isPressed[i]) {
79              tmp - 0.0078125 => tmp => s.gain;
80              2::ms => now;
81          }
82
83          0 => s.freq;
84      }
85  }
```

B.3 SynthFrame.java

```
1  import java.awt.BorderLayout;
2  import java.awt.Color;
3  import java.awt.Insets;
4  import java.awt.event.ActionEvent;
5  import java.awt.event.ActionListener;
6  import java.awt.event.ComponentAdapter;
7  import java.awt.event.ComponentEvent;
8  import java.awt.event.KeyEvent;
9
10 import javax.swing.AbstractButton;
11 import javax.swing.ButtonGroup;
12 import javax.swing.ImageIcon;
13 import javax.swing.JCheckBoxMenuItem;
14 import javax.swing.JColorChooser;
15 import javax.swing.JFrame;
16 import javax.swing.JMenu;
17 import javax.swing.JMenuBar;
18 import javax.swing.JMenuItem;
19 import javax.swing.JRadioButtonMenuItem;
20 import javax.swing.KeyStroke;
21
22 public class SynthFrame extends JFrame implements ActionListener {
23     private static final long serialVersionUID = -8132073014402103985
          L;
24
```

```
25      public static void main(String[] args) {
26          new SynthFrame("PrimeSynth");
27      }
28
29      private Synth synth;
30      private JMenuBar menu;
31      private JMenu fileMenu, settingsMenu, bgMenu, screenMenu;
32      private JMenuItem quitItem;
33      private ButtonGroup colorGroup, layoutGroup;
34      private JRadioButtonMenuItem stdRadioItem, redRadioItem,
            greenRadioItem,
35              blueRadioItem, customRadioItem, normalRadioItem,
                    nineteenRadioItem;
36
37      private JCheckBoxMenuItem fullscreenCheckItem;
38
39      public SynthFrame(String title) {
40          synth = new Synth();
41
42          menu = new JMenuBar();
43          fileMenu = new JMenu("File");
44          settingsMenu = new JMenu("Settings");
45          bgMenu = new JMenu("Background color");
46          screenMenu = new JMenu("Screen");
47          quitItem = new JMenuItem("Quit");
48          colorGroup = new ButtonGroup();
49          layoutGroup = new ButtonGroup();
50          stdRadioItem = new JRadioButtonMenuItem("Default", true);
51          redRadioItem = new JRadioButtonMenuItem("Red");
52          greenRadioItem = new JRadioButtonMenuItem("Green");
53          blueRadioItem = new JRadioButtonMenuItem("Blue");
54          customRadioItem = new JRadioButtonMenuItem("Choose...");
55          fullscreenCheckItem = new JCheckBoxMenuItem("Full screen");
56          normalRadioItem = new JRadioButtonMenuItem("Normal layout",
                true);
57          nineteenRadioItem = new JRadioButtonMenuItem("Nineteen tone
                layout");
58
59          add(synth, BorderLayout.CENTER);
60          setJMenuBar(menu);
61          menu.add(fileMenu);
62          menu.add(settingsMenu);
63          fileMenu.add(quitItem);
64          settingsMenu.add(bgMenu);
65          settingsMenu.add(screenMenu);
66          settingsMenu.addSeparator();
67          settingsMenu.add(normalRadioItem);
68          settingsMenu.add(nineteenRadioItem);
69          bgMenu.add(stdRadioItem);
70          bgMenu.add(redRadioItem);
71          bgMenu.add(greenRadioItem);
72          bgMenu.add(blueRadioItem);
73          bgMenu.addSeparator();
74          bgMenu.add(customRadioItem);
```

```
75          screenMenu.add(fullscreenCheckItem);
76
77          colorGroup.add(stdRadioItem);
78          colorGroup.add(redRadioItem);
79          colorGroup.add(greenRadioItem);
80          colorGroup.add(blueRadioItem);
81          colorGroup.add(customRadioItem);
82
83          layoutGroup.add(normalRadioItem);
84          layoutGroup.add(nineteenRadioItem);
85
86          addActionListeners(quitItem, stdRadioItem, redRadioItem,
87                  greenRadioItem, blueRadioItem, customRadioItem,
88                  fullscreenCheckItem, normalRadioItem,
                        nineteenRadioItem);
89
90          // listener for changes of window size
91          addComponentListener(new ComponentAdapter() {
92              @Override
93              public void componentResized(ComponentEvent e) {
94                  Insets is = getInsets();
95                  synth.setSize(getWidth() - is.left - is.right,
                            getHeight()
96                                  - menu.getHeight() - is.top - is.bottom);
97              }
98          });
99
100         quitItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_Q,
101                 ActionEvent.CTRL_MASK));
102         fullscreenCheckItem.setAccelerator(KeyStroke.getKeyStroke(
                KeyEvent.VK_F11, 0));
103
104         setTitle(title);
105         setIconImage(new ImageIcon("icon.png").getImage());
106         setDefaultCloseOperation(EXIT_ON_CLOSE);
107         pack();
108
109         setLocationRelativeTo(null);
110         setVisible(true);
111
112         synth.requestFocus();
113     }
114
115     @Override
116     public void actionPerformed(ActionEvent e) {
117         Object src = e.getSource();
118
119         if (src == quitItem) {
120             System.exit(0);
121         } else if (src == fullscreenCheckItem) {
122             if (fullscreenCheckItem.getState())
123                 goFullScreen();
124             else
125                 resetScreen();
```

```
126         } else if (src == normalRadioItem) {
127             if (synth.getKeyLayout() == 19)
128                 synth.switchLayout();
129         } else if (src == nineteenRadioItem) {
130             if (synth.getKeyLayout() == 12)
131                 synth.switchLayout();
132         } else {
133             Color c;
134 
135             if (src == redRadioItem)
136                 c = Color.red;
137             else if (src == greenRadioItem)
138                 c = Color.green;
139             else if (src == blueRadioItem)
140                 c = Color.blue;
141             else if (src == stdRadioItem)
142                 c = Color.lightGray;
143             else
144                 c = JColorChooser.showDialog(this, "Choose color",
                        Color.black);
145 
146             synth.setBackground(c);
147         }
148     }
149 
150     private void addActionListeners(AbstractButton... v) {
151         for (AbstractButton ab : v)
152             ab.addActionListener(this);
153     }
154 
155     private void goFullScreen() {
156         dispose();
157         setUndecorated(true);
158         setExtendedState(MAXIMIZED_BOTH);
159         synth.setSize(getWidth(), getHeight() - menu.getHeight());
160         setVisible(true);
161     }
162 
163     private void resetScreen() {
164         dispose();
165         setUndecorated(false);
166         synth.resetSize();
167         pack();
168         setLocationRelativeTo(null);
169         setVisible(true);
170     }
171 }
```

B.4 Synth.java

```
1 import java.awt.Color;
2 import java.awt.Dimension;
3 import java.awt.Graphics;
```

```
4  import java.awt.Graphics2D;
5  import java.awt.event.KeyEvent;
6  import java.awt.event.KeyListener;
7  import java.awt.event.MouseAdapter;
8  import java.awt.event.MouseEvent;
9  import java.net.InetAddress;
10 import java.util.Arrays;
11 import java.util.LinkedList;
12
13 import javax.swing.JPanel;
14
15 import com.illposed.osc.OSCMessage;
16 import com.illposed.osc.OSCPortOut;
17
18 public class Synth extends JPanel implements KeyListener {
19     private static final long serialVersionUID = -2762139888541781696
          L;
20
21     // window size
22     private final int width = 320;
23     private final int height = 200;
24
25     // scaling factor
26     private double scaleX = 1, scaleY = 1;
27
28     private Color bgColor;
29     private SynthKey[] keys12, keys19, keys;
30
31     private static final int[] keymap12 = new int[256], keymap19 =
          new int[256];
32     private static int[] keymap;
33     static {
34         // -1 means that it is unmapped
35         Arrays.fill(keymap12, -1);
36         System.arraycopy(keymap12, 0, keymap19, 0, 256);
37
38         // white keys
39         keymap12['A'] = keymap12['a'] = keymap19['A'] = keymap19['a']
               = 0;
40         keymap12['S'] = keymap12['s'] = keymap19['S'] = keymap19['s']
               = 1;
41         keymap12['D'] = keymap12['d'] = keymap19['D'] = keymap19['d']
               = 2;
42         keymap12['F'] = keymap12['f'] = keymap19['F'] = keymap19['f']
               = 3;
43         keymap12['G'] = keymap12['g'] = keymap19['G'] = keymap19['g']
               = 4;
44         keymap12['H'] = keymap12['h'] = keymap19['H'] = keymap19['h']
               = 5;
45         keymap12['J'] = keymap12['j'] = keymap19['J'] = keymap19['j']
               = 6;
46         keymap12['K'] = keymap12['k'] = keymap19['K'] = keymap19['k']
               = 7;
47
```

```
48          // black sharp keys (top)
49          keymap12['W'] = keymap12['w'] = keymap19['W'] = keymap19['w']
                = 8;
50          keymap12['E'] = keymap12['e'] = keymap19['E'] = keymap19['e']
                = 9;
51
52          keymap12['T'] = keymap12['t'] = keymap19['T'] = keymap19['t']
                = 10;
53          keymap12['Y'] = keymap12['y'] = keymap19['Y'] = keymap19['y']
                = 11;
54          keymap12['U'] = keymap12['u'] = keymap19['U'] = keymap19['u']
                = 12;
55
56          // black flat keys (bottom)
57          keymap19['Z'] = keymap19['z'] = 13;
58          keymap19['X'] = keymap19['x'] = 14;
59          keymap19['C'] = keymap19['c'] = 15;
60          keymap19['V'] = keymap19['v'] = 16;
61          keymap19['B'] = keymap19['b'] = 17;
62          keymap19['N'] = keymap19['n'] = 18;
63          keymap19['M'] = keymap19['m'] = 19;
64      }
65
66      private OSCPortOut sender;
67
68      // used for remembering the latest pressed, using mouse, key(s)
69      // so they can be properly released (depressed, haha) later
70      private final LinkedList<SynthKey> pushStack = new LinkedList<
            SynthKey>();
71
72      private final MouseAdapter mouseAdapter = new MouseAdapter() {
73          @Override
74          public void mousePressed(MouseEvent e) {
75              final int x = (int) Math.round(e.getX() / scaleX), y = (
                    int) Math
76                      .round(e.getY() / scaleY);
77
78              // the black keys need to be checked first because they
                    are
79              // placed on top of the white ones, therefore the key
                    array is
80              // iterated backwards
81              for (int i = keys.length - 1; i >= 0; i--) {
82                  if (keys[i].contains(x, y)) {
83                      pushKey(i);
84                      pushStack.addFirst(keys[i]);
85                      break;
86                  }
87              }
88
89              repaint();
90          }
91
92          @Override
```

```
93          public void mouseReleased(MouseEvent e) {
94              if (pushStack.isEmpty())
95                  return; // optimization
96
97              for (SynthKey key : pushStack)
98                  key.release();
99
100             repaint();
101         }
102     };
103
104     public Synth() {
105         try {
106             sender = new OSCPortOut(InetAddress.getLocalHost(), 6449)
                    ;
107         } catch (Exception e) {
108             e.printStackTrace();
109         }
110
111         bgColor = Color.lightGray;
112
113         initBoard();
114
115         // needed for pack() in PianoFrame
116         setPreferredSize(new Dimension(width, height));
117
118         // for registering key presses
119         addKeyListener(this);
120
121         // for registering mouse clicks
122         addMouseListener(mouseAdapter);
123
124         setFocusable(true);
125     }
126
127     private void initBoard() {
128         // y coordinate for keys
129         final int y = 0;
130
131         keys12 = new SynthKey[13];
132         keys19 = new SynthKey[20];
133
134         for (int i = 0; i < 8; i++) {
135             keys12[i] = new WhiteKey(i * 40, y);
136             keys19[i] = new WhiteKey(i * 40, y);
137         }
138
139         // x is starting x coordinate, k is number of keys and i
                groups
140         for (int i = 0, x = 27, k = 8; i < 2; i++, x += 40) {
141             for (int j = 0; j < 2 + (i & 1); j++, k++, x += 40) {
142                 keys12[k] = new BlackKey(x, y);
143                 keys19[k] = new BlackKey(x, y);
144             }
```

```
145         }
146
147         for (int i = 13, x = 27; i < 20; i++, x += 40)
148             keys19[i] = new BlackKey(x, keys19[0].getY() + keys19[0].
                    getHeight() - keys19[i - 1].getHeight());
149
150         keys = keys12;
151         keymap = keymap12;
152     }
153
154     public void octaveUp() {
155         sendOSCMsg(-1);
156     }
157
158     public void octaveDown() {
159         sendOSCMsg(-2);
160     }
161
162     public void switchLayout() {
163         if (keys == keys12) {
164             keys = keys19;
165             keymap = keymap19;
166         } else {
167             keys = keys12;
168             keymap = keymap12;
169         }
170         repaint();
171     }
172
173     public int getKeyLayout() {
174         if (keys == keys12)
175             return 12;
176         return 19;
177     }
178
179     @Override
180     public void keyPressed(KeyEvent e) {
181         if (e.getKeyCode() == KeyEvent.VK_UP) { octaveUp(); return; }
182         else if (e.getKeyCode() == KeyEvent.VK_DOWN) { octaveDown();
                return; }
183
184         int c = e.getKeyChar();
185         if (c >= keymap.length)
186             return;
187
188         int key = keymap[c];
189         if (key >= 0)
190             pushKey(key);
191     }
192
193     @Override
194     public void keyReleased(KeyEvent e) {
195         int c = e.getKeyChar();
196         if (c >= keymap.length)
```

```
197             return;
198
199         int key = keymap[c];
200         if (key >= 0)
201             releaseKey(key);
202     }
203
204     @Override
205     public void keyTyped(KeyEvent e) { }
206
207     /**
208      * drawing to the frame, called via repaint()
209      */
210     @Override
211     public void paint(Graphics g) {
212         ((Graphics2D) g).scale(scaleX, scaleY);
213
214         // clear
215         g.setColor(bgColor);
216         g.fillRect(0, 0, width, height);
217
218         for (int i = 0; i < keys.length; i++)
219             keys[i].draw(g);
220     }
221
222     private void pushKey(int push) {
223         keys[push].push();
224         sendOSCMsg(push);
225         repaint();
226     }
227
228     private void releaseKey(int push) {
229         keys[push].release();
230         repaint();
231     }
232
233     public void resetSize() {
234         scaleX = 1;
235         scaleY = 1;
236         repaint();
237     }
238
239     private void sendOSCMsg(int tone) {
240         try {
241             OSCMessage msg = new OSCMessage("/synth" + getKeyLayout()
                    , new Object[] { tone });
242             sender.send(msg);
243         } catch (Exception e) {
244             e.printStackTrace();
245         }
246     }
247
248     @Override
249     public void setBackground(Color color) {
```

```
250         this.bgColor = color;
251         repaint();
252     }
253 
254     /**
255      * scales the drawing area to the given size
256      */
257     @Override
258     public void setSize(int w, int h) {
259         scaleX = (double) w / width;
260         scaleY = (double) h / height;
261         repaint();
262     }
263 }
```

B.5 SynthKey.java

```
1  import java.awt.*;
2  
3  public class SynthKey {
4      private int x, y, width, height;
5      private Color color, pushColor;
6      private boolean isPushed;
7  
8      public SynthKey(int x, int y, int width, int height, Color color,
9              Color pushColor) {
10         setX(x);
11         setY(y);
12         setWidth(width);
13         setHeight(height);
14         setColor(color);
15         setPushColor(pushColor);
16     }
17 
18     public boolean contains(int x, int y) {
19         return x >= this.x && x <= this.x + width && y >= this.y && y
               <= this.y + height;
20     }
21 
22     public void draw(Graphics g) {
23         if (isPushed)
24             g.setColor(pushColor);
25         else
26             g.setColor(color);
27 
28         g.fillRect(x, y, width - 1, height);
29     }
30 
31     public Color getColor() {
32         return color;
33     }
34 
35     public int getHeight() {
```

```
36          return height;
37      }
38
39      public Color getPushColor() {
40          return pushColor;
41      }
42
43      public int getWidth() {
44          return width;
45      }
46
47      public int getX() {
48          return x;
49      }
50
51      public int getY() {
52          return y;
53      }
54
55      public boolean isPushed() {
56          return isPushed;
57      }
58
59      public void push() {
60          isPushed = true;
61      }
62
63      public void release() {
64          isPushed = false;
65      }
66
67      public void setColor(Color c) {
68          this.color = c;
69      }
70
71      public void setHeight(int h) {
72          this.height = h;
73      }
74
75      public void setPushColor(Color pc) {
76          this.pushColor = pc;
77      }
78
79      public void setWidth(int w) {
80          this.width = w;
81      }
82
83      public void setX(int x) {
84          this.x = x;
85      }
86
87      public void setY(int y) {
88          this.y = y;
89      }
```

```
90 }
```

B.6 WhiteKey.java

```
1  import java.awt.Color;
2
3  public class WhiteKey extends SynthKey {
4      public WhiteKey() {
5          this(0, 0);
6      }
7
8      public WhiteKey(int x, int y) {
9          super(x, y, 40, 200, Color.white, new Color(222, 222, 222));
10     }
11 }
```

B.7 BlackKey.java

```
1  import java.awt.Color;
2
3  public class BlackKey extends SynthKey {
4      public BlackKey() {
5          this(0, 0);
6      }
7
8      public BlackKey(int x, int y) {
9          super(x, y, 25, 80, Color.black, new Color(20, 100, 200));
10     }
11 }
```