A 19-Tone Scale Synthesizer

CHRISTIAN LINDEBORG
and CHRISTOFFER SANDBERG

KTH Computer Science
and Communication

Bachelor of Science Thesis
Stockholm, Sweden 2011

A 19-Tone Scale Synthesizer

C H R I S T I A N L I N D E B O R G
a n d C H R I S T O F F E R S A N D B E R G

Bachelor's Thesis in Computer Science (15 ECTS credits)
at the School of Computer Science and Engineering
Royal Institute of Technology year 2011
Supervisor at CSC was Henrik Eriksson
Examiner was Mads Dam

URL: www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2011/
lindeborg_christian_OCH_sandberg_christoffer_K11085.pdf

Kungliga tekniska högskolan
Skolan för datavetenskap och kommunikation

KTH CSC
100 44 Stockholm

URL: www.kth.se/csc

Abstract

The subject of this report is the theory behind a *19-tone equal temperament musical scale* (as opposed to the regular *12-tone equal temperament scale*) and the implementation of a synthesizer using such a scale made using an audio programming language. After a brief introduction to musical theory and the construction of scales we delve into the theory behind the 19-tone scale, and show why it is an interesting experiment. We also discuss the alternative ways of implementing the synthesizer and have documented our efforts to create one in our chosen language, *SuperCollider*.

Sammanfattning

Denna rapport behandlar teorin bakom en *19-tonsskala av liksvävande temperatur* (i motsats till den vanliga *12-tonsskalan av liksvävande temperatur*) och implementationen av en *synthesizer* som använder den skalan i ett språk avsett för ljudprogrammering. Efter en kort introduktion till musikteori och matematiken bakom skalor fördjupar vi oss i teorin bakom 19-tonsskalan, och visar varför det är ett intressant experiment. Vi diskuterar också alternativa sätt att implementera synthesizern och har dokumenterat våra försök att skapa en i det valda språket, *SuperCollider*.

Contents

1 Introduction

The goal of this project is to implement a nineteen tone scale synthesizer in a programming language suited for audio programming. We explore the musical theory behind the concept and the different ways of performing the implementation. For clarity's sake we point out that when we say *synthesizer* in this report we mean a *software synthesizer*, not a physical instrument.

1.1 Background

We have chosen this project because of both its programming- and musical aspects. We are both interested in music and its production and performance, and learning a new programming language especially tailored for this seemed interesting. We have a basic grasp of musical theory and hope to use this project to enhance our understanding of it.

1.2 Statement of Collaboration

While much of the work was done in close cooperation, there was a division of labour between the theory- and coding parts. Christian was responsible for the theory part and wrote most of its text. Christoffer was responsible for the coding of the application and the text concerning its implementation.

1.3 The Problem Statement

We aim to explore the theory of the 19 tone equal temperament scale, compare it to the 12 tone equal temperament scale, find out what makes it interesting and look into the new possibilities that the scale offers. In order to do so many, more fundamental, concepts in musical theory will have to be explained.

To be able to experiment with the scale we will develop a computer program in the form of a synthesizer programmed in a so called audio programming language, that we will have to learn to use. There are a number of different languages that might be suitable. These options will be explored and we will attempt to choose the one best fit for our purposes. The synthesizer will feature some sort of graphical user interface. We will make two separate versions of the synthesizer, one featuring a simple keyboard (this also serves the purpose of evaluating what our chosen language can do) producing sustained notes, and another more advanced with a simple looping sequencer, for making melodic patterns.

2 Theory

2.1 Music Theory

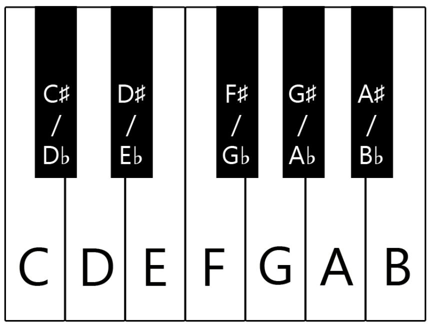

Figure 1: An octave on a standard piano keyboard.

In modern western music the twelve tone scale is ubiquitous and the basis of almost all music, but it is by no means the only scale possible. The nineteen tone scale has many interesting qualities that we will attempt to cover here, while introducing music theory concepts that might not be familiar to the reader. A glossary of musical terms used in this report can be found in appendix A.

A *tone* is a vibration of air at a certain frequency audible to our ears and a *scale* is a series of these tones. An important concept when working with scales is the *interval*, the ratio between the frequency of two tones. For example the octave, the most fundamental interval, is a doubling of frequency. So in the octave interval the ratio between frequencies is 2:1. Most scales are defined as subdivisions of this fundamental interval. Some other fundamental intervals are the perfect fifth, with the ratio 3:2 and the perfect fourth with the ratio 4:3.

The words tone and note are sometimes used interchangeably in our report. Tone really means a sound at a certain pitch and a note is the visual representation of a tone playing for a certain amount of time in musical notation. What term should be used where can sometimes be hard to decide. We have tried to use tone for the most part.

In earlier music, preserving these pure ratios of intervals was seen as very important and musicians used a system called *Just Intonation*, where the ratios of the intervals were kept as close to their correct ratios as possible. This kind of tuning made some intervals sound pure and pleasing while others had to adapt and not come as close to their just intervals. This uneven distribution led to the situation where an interval could represent different frequency ratios on different parts of the keyboard.

As music evolved musicians wanted to be able to move between different keys more freely. Just intonation was not well suited for this, and in the beginning of the 18th century the idea of *Equal Temperament* was introduced [2, p.7]. This meant that the octave was simply split into equally large intervals and music could be *transposed* between different keys without altering the relationships between the notes. Today we mostly use the *Twelve-Tone Equal Temperament*, often abbreviated as 12-TET (which we will use from now on), but others are possible. All equal temperaments involve compromise and in most cases all intervals (except the octave) will be approximations of their ideal ratios.

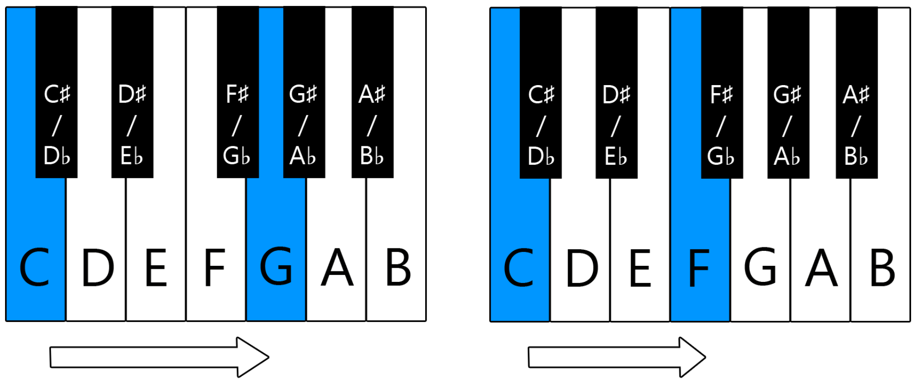

Figure 2: The perfect fifth and perfect fourth mapped to 12-TET

In order to create an equally tempered scale, the general idea is to keep the octaves *just interval* and divide the remaining space of frequencies into equally large intervals. Since an interval is the relation and not the difference between two frequencies it is not enough to just divide it into equally large parts. To divide the octave into n equally large intervals we start with what we know. If we have some tone with frequency a the octave up from that tone has the frequency $2a$. To get the factor r and ratio between the frequencies of two tones of the smallest interval we can make the equation $a \cdot r^n = 2a$ which gives us $r = \sqrt[n]{2}$ independent of frequency.

$$r = \sqrt[n]{2} \tag{1}$$

One more thing that should be mentioned is the different usages of the word scale. Even those who have no real experience of music might have heard of the *C Major Scale* for instance. This is a set of seven notes from the 12-TET scale, called a *diatonic scale.* These types of scales are what most musical works are *written in.* This means that the composition mostly uses the notes from this scale. The 12-TET scale on the other hand is a *chromatic scale.* These chromatic scales are seldom used as a basis for a musical composition, but rather contain the building blocks to a range of smaller scales.

2.2 19-TET and 12-TET

I order to show how well different equal temperaments (in our case 12-TET and 19-TET) approximate the just intonation, we use a unit called the *cent.* The cent is modeled after 12-TET where every *semitone* consists of 100 cents, and thus the octave consists of 1200 cents.

To calculate the number of cents between two frequencies (meaning it does not matter which temperament we are in) we use the formula: [1]

$$n = 1200 \cdot log_2 \left(\frac{b}{a} \right) \approx 3986 \cdot log_{10} \left(\frac{b}{a} \right) \tag{2}$$

The following is a list of intervals in 12-TET and how well they correspond to the just intervals. The intervals presented were chosen either because they are commonly used, important, or deviate very little or very much from their just interval counterparts. *Steps* refers to the number of semitones you would increment with on the keyboard to reach traverse the interval.

Cents are used all over music theory, not just when speaking of the 12-TET scale (although this is where they make the most sense). So in 19-TET every step in the scale is 1200/19=63.158 cents.

Name	**Steps**	**Exact value in 12-TET**	**Cents**	**Just Ratio**	**Cents (just)**	**Error**
Minor third	3	$2^{3/12} = \sqrt[4]{2}$	300	6:5	315.64	-15.64
Major third	4	$2^{4/12} = \sqrt[3]{2}$	400	5:4	386.31	+13.69
Perfect Fourth	5	$2^{5/12} = \sqrt[12]{32}$	500	4:3	498.04	+1.96
Perfect Fifth	7	$2^{7/12} = \sqrt[12]{128}$	700	3:2	701.96	-1.96
Minor Seventh	10	$2^{10/12} = \sqrt[6]{32}$	1000	7:4	968.83	+31.17

We see here that some intervals benefit from this partitioning, especially the perfect fifth and the perfect fourth, while some others suffer, the most prominent example being the minor seventh. If we now attempt to perform the partitioning into nineteen equally large parts in the same way we get another set of intervals:

Name	**Steps**	**Exact value in 19-TET**	**Cents**	**Just Ratio**	**Cents (just)**	**Error**
Minor third	5	$2^{5/19}$	315.79	6:5	315.64	+0.15
Major third	6	$2^{6/19}$	378.95	5:4	386.31	-7.36
Perfect Fourth	8	$2^{8/19}$	505.26	4:3	498.04	+7.22
Perfect Fifth	11	$2^{11/19}$	694.74	3:2	701.96	-7.22
Minor Seventh	15	$2^{15/19}$	947.37	7:4	968.83	-21.46

Here the set of errors in the rightmost column looks quite different. The perfect fifth is less accurate but the minor- and major third are more accurate and no tone deviates as much from its correct ratio as the minor seventh does in 12-TET.

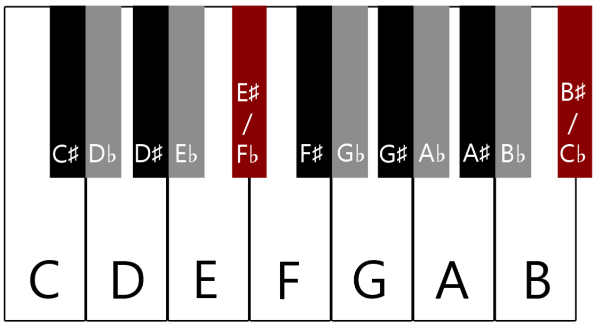

Figure 3: What a 19-TET keyboard might look like.

The picture above is one way of constructing a 19-TET keyboard. Compared to the regular 12-TET keyboard the black keys have simply been split in two, and new keys added between those white keys which were immediately adjacent in 12-TET. New colors were introduced to differentiate the keys. As we saw in Figure 1, and as musicians know, all the black keys have two names. C sharp and D flat for example refer to the same note, they are *enharmonic*. In 19-TET all these notes (except for E sharp/F flat and B sharp/C flat) refer to distinct notes. This is one of 19-TETs strengths, compared to other unorthodox scales (31-TET for example), we can use the traditional notation system, even though the meaning of the notes is a bit different. One could in theory take any piece of music written in regular 12-TET and interpret it as if it was written in 19-TET without any modifications to the sheet music. This would probably sound quite strange, but it is possible!

12-TET is designed with the idea in mind that fifths on top of each other will cycle through all the notes in the scale. This is known by musicians as the *Circle of Fifths*, and is a very useful tool for constructing chords and modulating between different keys, an important feature indeed in modern music.

In 19-TET this behaviour is extended to work for all intervals. This is because 19 is a prime number. Any interval repeated 19 times will cycle through all the tones of the scale.

19-TET is thus interesting for several reasons. It approximates some fundamental just intervals better than 12-TET, it can use existing 12-TET note terminology and it has the interesting property of every interval being able to cycle through the notes of the scale. This is also accomplished with a relatively few number of notes compared to other unorthodox tunings like 31-TET or 22-TET which are more cumbersome.

2.3 Technical

There are several programming languages specially tailored for working with audio. They are good examples of domain-specific languages, that is, programming languages designed to ease development in a relatively narrow field. The term *Audio Synthesis Environment* is also often used, since they often consist of both a language (that can sometimes be graphical) and an environment to run the language in. Some have been around since the 80s, like *CSound* and *Max/MSP* while others are more recent like for example *ChucK*. *SuperCollider* is somewhere in between on the timescale.

3 Execution

3.1 Choosing a Language

We chose to look closer at three different audio programming languages, *Max/MSP*, *ChucK* and *SuperCollider*. There are many others but we cannot within the scope of this project thoroughly investigate all the alternatives. We have had no prior experience with any of the candidate languages, and have attempted to use our knowledge of other more general languages to assess the suitability of the choices. We take into account our own ability to learn the language, and therefore we will place preference on a language that are somewhat similar to what we already know(object-oriented languages such as *Java*).

Max/MSP is a largely graphical language that has been around since the 80s, where visible modules are connected and different parameters are set in a graphical user interface, and is used by many artists and others who don't have a lot of technical knowledge[7]. We will not be using Max/MSP for this project since we would rather work with and learn a real programming language, and Max/MSP is also one of the few alternatives which is not free.

ChucK is a new language in this category and emphasizes readability and flexibility, while sacrificing some performance [6]. It lacks some of the features that would make it a real programming language, such as good support for string operations[6]. ChucK is a young language, and there is a risk it is not mature enough not to cause unnecessary trouble.

SuperCollider is a fully fledged programming language that's aimed towards designing sounds and instruments, and makes the construction of GUIs relatively easy[5]. We have chosen to use SuperCollider, since it facilitates easy construction of GUIs yet it is still a mature all-purpose programming language, which makes it more interesting to learn. It also has a thriving community built around it and a nice built-in documentation system.

3.2 SuperCollider details

SuperCollider uses a client-server module architecture that can communicate using a protocol called *Open Sound Control*, or *OSC*. The server generates the sounds and the programmer generally uses only the client to program, and sends commands to the server. Since OSC, which is implemented by many other audio applications and environments, is used in communication between the two components, there is nothing preventing the use of another client than the default one.

The language is object oriented but many things are done "in reverse" from the perspective of a Java programmer. For instance the *if*-statement and the *for*-loop.

```
(2>1).if({
    "2 is larger than 1".postln;
});
```

Here we see that the if-statement and its condition are in reversed positions compared to most other languages. The reverse phenomenon is probably derived from the traditional class-methods where the object reference is passed as the first argument to the method.

```
19.do({
    "Hello world!".postln;
});
```

The preceding code prints the string Hello world! 19 times to the post window, followed by a 19 (SuperCollider always prints out the value of the last line of code).

3.3 Constructing the Synthesizer and learning SuperCollider

First off we will attempt to construct a GUI featuring a simple musical keyboard suited for 19-TET, with keys added for the new notes. The user should be able to simply click on the keys and a tone will sound for a period of time. One should be able to click several notes and hear how they sound together.

In order to be able to experiment and try out new things in 19-TET we will also design a small and simple looping *sequencer*. A sequencer is what is often used in digital music production to describe a progression of notes, much like a piano roll on an old self-playing piano would look, except sideways (in most modern cases). The sequencer should be able to make use of the sound generating code from the keyboard implementation. It will look like a simple grid of 19 rows (one for each note in the scale) and 16 columns, where every cell can be active or inactive. Every note in the same column is played at the same time, and all the columns are played in order, over and over again. This looping behaviour lets the user try out chords and melodies in an experimental way.

3.3.1 Construction of the keyboard application

To make the application there are a few obstacles that needs to be overcome using SuperCollider.

- Playing a tone of a certain frequency.
- Turning the tone on/off.
- Creating a GUI with a button for each tone.

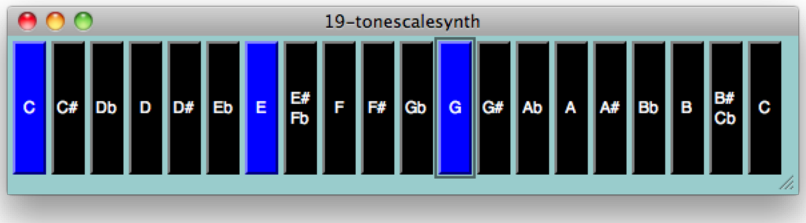

Figure 4: The synthesizer playing a C Major chord.

To play a sound in SuperCollider a *Synth*-object is used. A Synth-object is created referencing a *SynthDef*-object which act as a template for the Synth and is sent to the audioserver in advance. The Synth can then be told to play or pause as needed. For this application a SynthDef was created with a variable parameter, the frequency *freq*.

```
SynthDef("synth_template", {
   |freq|
   Out.ar([0, 1],
       SyncSaw.ar(freq, 150, 0.2)
   );
}).send(s);
```

As output a sawtooth wave synced to the frequency *freq* is generated and played at channels 0 and 1 (right and left speaker). After creation the SynthDef is immediately sent to the audioserver and can be used for creating a Synth. As the plan is to have one whole octave on the keyboard there will be twenty different tones and thus twenty Synths are needed. These are put into an array for easy access in the future.

```
synths = Array.new(20);
r = 1.03715504;

20.do({ |i|
   var calcFreq = baseFreq * (r**(i));
   a = Synth.new("synth_template", [\freq, calcFreq]);
   a.run(false);
   synths.add(a);
});
```

For each tone the frequency is calculated from the base-frequency (261.626 Hz in this case for a standard C) and stepping up i steps of the smallest interval by multiplying the base-frequency by r, i times. The variable r is the ratio between two tones of the smallest interval in the scale and calculated to be approximately 1.03715504 by formula (1). A Synth is then created using the previously created SynthDef and given the calculated frequency as parameter. The Synth is by default started in playing mode so its run method is called with the parameter set to false to prevent just that.

The window is created by simply making an instance of the Window-class that takes parameters like title and position.

```
w = Window( "19-tonescalesynth", Rect( 450, 64, 640, 160 ));
```

The buttons are then created in a similar fashion with the window as parent. They also need two states, playing and non-playing, and some functionality when clicked.

```
20.do({ |i|
   b = Button( w, Rect( 0, 0, 25, 100 ));
   b.states = [[ toneMap.at(i), Color.white, Color.black ],
               [ toneMap.at(i), Color.white, Color.blue ]];

   b.addAction({
       |button|
       (button.value == 1).if({
           synths[i].run(true);
       }, {
           synths[i].run(false);
       });
   });
});
```

The states are set by the attribute with the same name. The same button text is set to the two states but different colors. The text is taken from the Dictionary toneMap which just maps the numbers 0-19 to a string with the name of the tone. The addAction method of the Button-class sets the function to run when the button is clicked. The called function is given one argument which is a reference to the clicked button, so depending on the state of that button the corresponding Synth is set to play or pause. The resulting program source code can be found in appendix B.

Construction of a simple graphical keyboard and an elementary sound-generation component was accomplished in SuperCollider with around 70 lines of code, showcasing the rapid development of these sort of applications that SuperCollider enables. The simple application consists of 20 buttons (the first note, C, was duplicated in order to provide us with the octave interval) representing tones that can be set to on or off. The tone from each button continues to sound for as long as it is set to on.

3.3.2 Construction of the sequencer

As predicted, large parts of the first application can be re-used in the construction of the sequencer. The GUI will need to be a grid of two-state buttons that are used to indicate if a tone should be played at the specific time or not. Each button-reference is stored in an array for later checking of state. The bigger change is that a timed loop is handling the playing of tones, and that tones end after a period of time rather than when a button is clicked. To be able to use the delaying function in the timed loop it has to happen in a Task. A Task is done in a separate process and created by instantiating the class and giving it a function to run.

```
Task.new({
   loop {
       // Play relevant tones.
       0.1.wait;
   }
}).play(AppClock);
```

This will create a new process in the form of a Task and start it. In it runs a loop indefinitely waiting a tenth of a second between every iteration. Before the delay the tones corresponding to pressed buttons in the current column needs to be played. As before this is done with one Synth per tone. The difference here is that each tone is supposed to end after a certain time and make use of what is called an envelope. That is, the sound should change over time, and emulate the sound of a key being struck or a string being plucked, rather than a constantly sounding tone. This is done with a slightly modified SynthDef.

```
SynthDef("synth_template", {
   |freq|
   var env = Env.perc(0.005, 1.5, 1, -4);
   var env_gen = EnvGen.kr(env, doneAction: 2);
   Out.ar([0, 1],
       SyncSaw.ar(freq, 150, 0.2, mul:env_gen)
   );
}).send(s);
```

As stated above the difference is the envelope, an EnvGen-object is given as argument when creating the wave which will give the amplitude the characteristics of the defined envelope. When creating the EnvGen there is also a special argument given, the doneAction, which when set to 2 effectively cleans up after the Synth when it has finished playing. In this example the envelope was set to a percussive shape by the *perc* method. So when creating a Synth from this SynthDef, it will follow the envelope and eventually

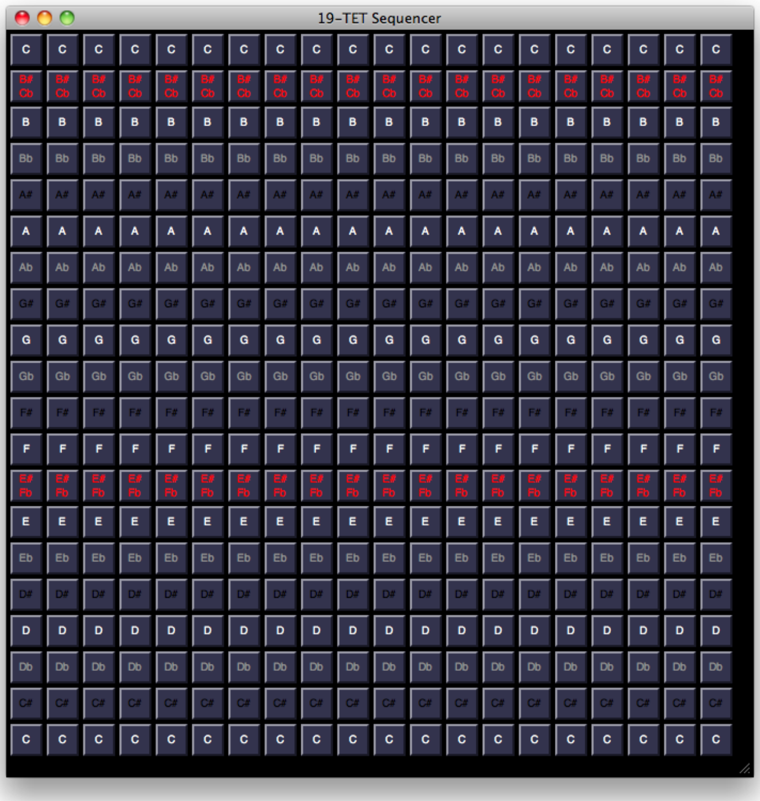

Figure 5: The sequencer.

die out and self-destruct. All that is left to do then is to actually create Synths corresponding to the frequency of the selected buttons in the current column in the iteration. The resulting program source code can be found in appendix C.

3.4 Sound experimentation

After construction of the sequencer was complete we made a 12-TET variation of the same application to make comparisons to. The first few notes of *Twinkle Twinkle Little Star* were used to perform very unscientific blind-tests. When listened to in rapid succession the two tunings were noticeably different. The perfect fifth in particular stood out as different. In 19-TET it is about 5 cents lower in pitch, which could make it sound like it "does not quite get there". However when listened to on its own without recent exposure to 12-TET it is hard to be sure. In this case we are of course only using intervals that appear in both scales, to be able to compare them.

When using those intervals from 19-TET that do not appear in 12-TET a completely different and “out of tune” sound is achieved.

The new intervals we have at our disposal enable the construction of new kinds of chords. One could for example take the regular minor chord (which consists of a base tone, a minor third and a fifth) and lower the minor third one one step, to a position that was never available in 12-TET. This produces a sound that could be interpreted as “even more sad” (approaching tragic) than the usual minor chord.

4 Conclusions

4.1 Results

The process for creating a 19-TET scale (generalized to any number of tones) has been explained together with the fundamentals of music theory needed to understand it. Some of the intervals from the 19-TET scale were calculated in cents. Comparing these to their just interval counterparts produced a different set of errors compared to those from 12-TET. Certain intervals like the minor and major third produced smaller errors while the error of the perfect fifth grew.

The audio programming language SuperCollider was chosen and successfully used to create two simple synthesizers with graphical user interfaces, using less code than we initially anticipated. Our rather modest goals for the synthesizers were quite easily fulfilled.

Minor and highly unscientific experimentation was made with the help of the produced sequencer program. Results of this are highly subjective and our thoughts can be read about in the previous section.

4.2 Discussion

Musical theory is a surprisingly big field and the exploration of 19-TET and what possibilities it entails could be taken much further, but not without spending a lot of time familiarizing oneself with a lot more general musical theory, which this report does not attempt. We feel however that we have made clear both that 12-TET is not the only alternative and that 19-TET is an interesting one, at least from a mathematical and theoretical point-of-view. What sounds good and what does not is as always highly subjective but in this case we think most people would agree that music in 19-TET sounds strange, even when only the intervals we are used to from 12-TET are used (with the new approximations of 19-TET). It is hard not to ask oneself questions like "what is music?" and "why does *this* sound nice while *that* does not?" We have been subjected to the sounds of 12-TET all our life, and that is hard for our ears to put aside. There have been attempts to make music in 19-TET but they are few and far between. So even though we doubt 19-TET will make an appearance on the charts anytime soon it is very interesting to think about.

SuperCollider is quite different from other languages we are used to and required some research and quite a bit of trial-and-error, even though the total amount of code required to do what we wanted was quite small. Finding information when we encountered problems was not always as easy as we wanted. The online documentation was hard to perform searches in compared to what we were used to from other languages, and the built-in documentation was not always enough. We never used any of the other audio programming languages to do any real work and cannot compare SuperCollider with them when it comes to practical usage, but can say from our experience with other more general-purpose languages that development was greatly enhanced by using a domain-specific language. Writing a similar program in Java would have required many times the amount of code and would be much more complex to write and understand.

If we were to develop the application further we would add ways to modify the sound on the fly through the use of sliders controlling things like the envelope parameters or playback speed. Another interesting path would be to enable testing of scales of arbitrary size, and new ways of comparing them side-by-side in some fashion to be able to hear the more subtle differences.

References

[1] *Cent article on Wikipedia.* `http://en.wikipedia.org/wiki/Cent_(music)` (Accessed 2011-04-13) 2011.

[2] Per Brolinson, *Kompendium i Musikteori.* Stockholms Universitet, 1996.

[3] Ivor Darreg, *A Case For Nineteen.* `http://sonic-arts.org/darreg/case.htm` (Accessed 2011-03-09) 1979.

[4] Hubert S. Howe, Jr., *19-Tone Theory and Applications.* `http://qcpages.qc.edu/~howe/articles/19-ToneTheory.html` (Accessed 2011-03-09) 1993.

[5] *SuperCollider article on Wikipedia.* `http://en.wikipedia.org/wiki/SuperCollider` (Accessed 2011-04-10) 2011.

[6] *ChucK article on Wikipedia.* `http://en.wikipedia.org/wiki/ChucK` (Accessed 2011-04-10) 2011.

[7] *Max/MSP article on Wikipedia.* `http://en.wikipedia.org/wiki/Max/MSP` (Accessed 2011-04-10) 2011.

A List of Terms

A.1 Music Terms

- *12-TET:* The twelve-tone equal temperament scale.
- *19-TET:* The nineteen-tone equal temperament scale.
- *Tone:* A sound at a specific pitch
- *Note:* A written representation of a tone and how long it should be played etc.
- *Semitone:* Also known as a half-tone. This is the smallest step in a scale, the distance between two adjacent notes.
- *Interval:* The relationship between two notes.
- *Just interval:* A perfect interval, corresponding to a fraction of small numbers. For example a perfect fifth is (3/2) times the frequency of its keynote.
- *Scale:* A sequence of intervals acting as a palette to choose from, covering one octave.
- *Temperament:* A system of tuning where the just intervals are compromised.
- *Equal temperament:* In an equal temperament scale the step between two adjacent notes is always the same size, no matter where on the keyboard.
- *Key:* A scale having its origin at a specific note.
- *Transposition:* To move a note or group of notes up or down in pitch by a certain amount.

A.2 Technical Terms

- *GUI:* A Graphical user interface
- *Synthesizer:* An instrument that produces sound by electronic means. Can refer to a physical instrument with real keys and knobs, but can also be implemented in software and controlled in many ways. Throughout this report the latter is what is implied.
- *Sequencer:* Some way of electronically put notes in sequence in a specific way. Can be likened to the piano rolls of old self-playing pianos. Sequencers can differ greatly in appearance.

B nineteentonescale.scd

```
var synths, nrKeys, baseFreq, toneMap, calcFreq;

// Number of keys to play on (the toneMap needs to be synced with this)
nrKeys = 20;

// Starting freqency for the first tone in toneMap (C)
baseFreq = 261.626;

// toneMap maps the numbers 0-19 to the 20 tonenames in the octave
toneMap = Dictionary[
  0 -> "C",
  1 -> "C#",
  2 -> "Db",
  3 -> "D",
  4 -> "D#",
  5 -> "Eb",
  6 -> "E",
  7 -> "E#\nFb",
  8 -> "F",
  9 -> "F#",
  10 -> "Gb",
  11 -> "G",
  12 -> "G#",
  13 -> "Ab",
  14 -> "A",
  15 -> "A#",
  16 -> "Bb",
  17 -> "B",
  18 -> "B#\nCb",
  19 -> "C"
];

// Define the Synth-template for the tones with variable frequency
SynthDef("synth_template", {
  |freq|
  Out.ar([0, 1],
    SyncSaw.ar(freq, 150, 0.2)
  );
}).send(s);

// Create an array to fill with Synth-objects
synths = Array.new(nrKeys);
```

```
// The "magic number" that specifies the ratio between
// two tones of the smallest interval in the scale.
r = 1.03715504;

// For each key
nrKeys.do({ |i|

  // Calculate the frequency for the tone
  calcFreq = baseFreq * (r**(i));

  // Create Synth-object with the frequency
  a = Synth.new("synth_template", [\freq, calcFreq]);

  // Turn it off to begin with and put it in the array
  a.run(false);
  synths.add(a);
});

w = Window( "19-tonescalesynth", Rect( 450, 64, 640, 160 ));
w.view.decorator = FlowLayout( w.view.bounds );
w.view.background = Color( 0.6, 0.8, 0.8 );

// Generate the button-keyboard
nrKeys.do({ |i|
  b = Button( w, Rect( 0, 0, 25, 100 ));
  b.states = [[ toneMap.at(i), Color.white, Color.black ],
  [ toneMap.at(i), Color.white, Color.blue ]];

  b.addAction({
    |button|
    (button.value == 1).if({
      synths[i].run(true);
    }, {
      synths[i].run(false);
    });
  });
});
w.front;
```

C nineteentonesequencer.scd

```
var nrTones, nrTimes, baseFreq, countFreq, toneMap, buttons, toneIndex, colorMap;

// Number of keys to play on (the toneMap needs to be synced with this)
nrTones = 20;
// Number of "time units" in the sequence
nrTimes = 20;

// Starting freqency for the first tone in toneMap (C)
baseFreq = 261.626;

// toneMap maps the numbers 0-19 to the 20 tonenames in the octave
toneMap = Dictionary[
  0 -> "C",
  1 -> "C#",
  2 -> "Db",
  3 -> "D",
  4 -> "D#",
  5 -> "Eb",
  6 -> "E",
  7 -> "E#\nFb",
  8 -> "F",
  9 -> "F#",
  10 -> "Gb",
  11 -> "G",
  12 -> "G#",
  13 -> "Ab",
  14 -> "A",
  15 -> "A#",
  16 -> "Bb",
  17 -> "B",
  18 -> "B#\nCb",
  19 -> "C"
];

// Dictionary to control button-color
colorMap = Dictionary[
  0 -> Color.white,
  1 -> Color.black,
  2 -> Color.gray,
  3 -> Color.white,
  4 -> Color.black,
  5 -> Color.gray,
  6 -> Color.white,
  7 -> Color.red,
  8 -> Color.white,
  9 -> Color.black,
  10 -> Color.gray,
  11 -> Color.white,
  12 -> Color.black,
  13 -> Color.gray,
  14 -> Color.white,
  15 -> Color.black,
  16 -> Color.gray,
  17 -> Color.white,
  18 -> Color.red,
  19 -> Color.white
];

// Define the Synth-template for the tones with
// variable frequency and an envelope
SynthDef("synth_template", {
  |freq|
  //var env = Env.triangle(1, 0.2);
  var env= Env.perc(0.005, 1.5, 1, -4);
  var env_gen = EnvGen.kr(env, doneAction: 2);
  Out.ar([0, 1],
    SyncSaw.ar(freq, 150, 0.2, mul:env_gen)
  );
}).send(s);

w = Window( "19-TET Sequencer", Rect( 400, 64, 35*nrTimes, 35*nrTones ));
w.view.decorator = FlowLayout( w.view.bounds );
w.view.background = Color( 0.0, 0.0, 0.0 );

buttons = Array2D.new(nrTones, nrTimes);

// Generate "the grid" of buttons
nrTones.do({ |j|
  toneIndex = nrTones - j - 1;

  nrTimes.do({ |i|
    b = Button( w, Rect( 0, 0, 30, 30 ));
    b.states = [
      [ toneMap.at(toneIndex), colorMap.at(toneIndex), Color.new(0.2, 0.2, 0.3) ],
      [ toneMap.at(toneIndex), colorMap.at(toneIndex), Color.new(0.4, 0.4, 0.4) ]
    ];
    b.font_(Font("Helvetica", 10));

    buttons[toneIndex, i] = b;
  });
});
w.front;

// Create and start the loopsequence
Task({
  j = 0;
  loop {
    nrTones.do({
      |i|
      toneIndex = nrTones - i - 1;
      // Check if the button is pressed and if so play the tone
      (buttons[toneIndex,j].value == 1).if({
        countFreq = baseFreq * (1.03715504**(toneIndex));
        Synth("synth_template", ["freq", countFreq]);
      });
    });
  j = j + 1;
  (j == nrTimes).if({ j = 0; });
  0.1.wait;
  };
}).play(AppClock);
```