MODERN API DESIGN AND PHYSICAL COMPUTING TECHNIQUES IN JUST INTONATION PERFORMANCE PRACTICE

Mark Sonnabaum

Thesis Prepared for the Degree of

MASTER OF ARTS

UNIVERSITY OF NORTH TEXAS

May 2013

APPROVED:

Sonnabaum, Mark. Modern API Design and Physical Computing Techniques in Just Intonation Performance Practice. Master of Arts (Music), May 2013, 48 pp., 16 figures, references, 18 titles.

Music that uses just intonation has been historically difficult to perform. Using modern instruments and custom built instruments both have different sets of advantages and disadvantages. This paper explores how the problem has been approached previously by both Harry Partch and Ben Johnston, and proposes the decoupling of interface and sound production as a way forward. The design and implementation of a software instrument and a hardware prototype are described, both using a simple API for variable tuning instruments. The hardware prototype uses physical computing techniques to control the tuning of a string with a servo motor, while the software instrument exists entirely in a web browser. Finally, potential algorithms for clients of the API are presented, and the effectiveness of the hardware prototype is evaluated by measuring its pitch accuracy.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The performance of music using just intonation is problematic for a number of reasons, not least of which is the fact that modern instruments are often designed for equal temperament and performers are trained to fight against the justly intonated characteristics their instrument may have. Most well known composers who have attempted to solve this problem have done so in very different ways. Because issues lie with both performers and the use of modern instruments, a common approach has been to focus largely on one area or the other, which has mixed results.

In this paper, I discuss historical precedents in the area of just intonation performance, specifically focusing on the work of Harry Partch and Ben Johnston. Both composers have approached this problem differently due to skill sets or available technologies, so I highlight the most effective methods in each case and how we can learn from them.

My approach to this topic involves the use of a simple application programming interface to facilitate the use of both software modeling and physical computing techniques to achieve the level of pitch accuracy necessary for just intonation music. It is my belief that developing standard interfaces and tools that address these technical challenges will allow a greater number of composers and musicians to explore the rich musical possibilities of extended just intonation.

CHAPTER 2

JUST INTONATION PERFORMANCE

Unlike many musical trends in the twentieth century, composing using just intonation is neither a way to break free of tonality nor an attempt to return to it. It is inherently free of the western European musical associations that so many composers try to exorcise from their musical consciousness, simply because it's neither a genre, style, nor a school, all of which are ephemeral, but rather a musical foundation. Its practice is as ancient as music itself because it is based on the two musical phenomena, which are absolutely unchangeable: an object's ability to produce sound as it vibrates, and the human ear's ability to perceive it.[1] As a musical system, its rationale is irrefutable, although a modern composer's implementation of this system in his or her music tends to require a bit more conviction, if not downright evangelism. Because each composer who either experiments with or dedicates himself to using just intonation will likely use it differently in his or her music, most of this music will get broadly classified as "microtonal." Even the most educated performers will likely find this method of developing scales and harmonies by adding and subtracting whole number ratios to be confusing. The fact that there is no standard set of pitches, which can change with each composer (or piece), makes it even more difficult to understand as a whole. This alone is likely to intimidate most performers, who will not want to commit to learning the theory behind your system of tuning, becoming accustomed to the notation, and practicing in this unfamiliar system, only to sound "out-of-tune" to a lay audience. All composers who make the decision to compose in just, or extended just intonation deal with these issues in one way or another. The following section will explore,

[1] Partch, *Genesis of a Music: An Account of a Creative Work, Its Roots, and Its Fulfillments, Second Edition*.

compare, and contrast two very different approaches to composing and performing using just intonation through two of its most dedicated composers: Harry Partch and Ben Johnston.

Partch was initially inspired to develop a system of tuning based on just intonation to closer approximate the human voice. About his early work, Partch writes, “I came to the realization that the spoken word was the distinctive expression my constitutional makeup was best fitted for, and that I needed other scales and other instruments.”[2] He then decided to build a new instrument, which would allow him to realize these pitches more easily. He constructed a viola with an elongated neck that he initially called the “monophone,” but later renamed it the “adapted viola.” With the pitches clearly marked on the neck and the longer fingerboard, allowing for more widely spaced intervals, he was able to write and perform music using this instrument, which would provide a monophonic vocal accompaniment.

Partch eventually settled on a 43-tone-to-the-octave scale that he would build his instruments around and which would become the basis of all of his compositions. This system was developed using just ratios up to the 11th partial. The 11th partial is more or less an arbitrary stopping point for Partch, but necessary considering that he would be building instruments around this tuning, and the system needed to be somewhat fixed for his musicians to be able to play the instruments and understand what they were playing. In a way, the 11th partial is a rather modest stopping point since it could be considered the first step into “extended” just intonation. Although the natural harmonic 7th is still far from what we hear used in 12-tone equal temperament, it still naturally occurs in a cappella vocal groups, and is still heard as a “dominant seventh.” The 11th partial, however, occurs

[2]Partch, *Genesis of a Music: An Account of a Creative Work, Its Roots, and Its Fulfillments, Second Edition*.

almost exactly between a perfect fourth and a tritone—so if anything, it is heard as a very flat tritone, though I think most would agree that it has its own unique harmonic quality. Partch refers to his pitches as either “otonal” or “utonal,” meaning they are derived either directly from the overtone series, or they are the inverse of an overtone. These terms can roughly be considered synonymous with “major” and “minor” since the major triad that occurs in the 4th, 5th, and 6th partials (1:1, 5:4. 3:2) invert into a minor triad (1:1, 8:5, 4:3).

After writing mostly accompanied speech music for a number of years, Partch was criticized for focusing solely on pitch, resulting in a lack of rhythmic interest. He then began conceiving and building percussion instruments to add to his growing ensemble of custom built instruments. Being a very competent carpenter, he naturally gravitated towards building marimba-like instruments. As obsessive as Partch was with tuning, he was equally passionate about the idea of corporeality in music. The design of his instruments was heavily influenced by this idea, putting great stress on how the performer would play the instrument, and how he or she would look playing it. One example of this is the diamond marimba. The basic principals of the instrument are identical to a traditional marimba, but the keys are laid out in a diamond shape rather than the typical linear keyboard. This makes it possible to play just ratios in a logical way, since the bars are arranged according to Partch’s tonality diamond, where diagonal lines from left to right are utonal and lines from right to left are otonal.

Partch’s early attempts at developing a notation for his music were not terribly successful. He did not want to simply specify how the notes were different from 12-tone equal temperament, but rather write each note as its true whole number ratio. Although this may have seemed intuitive for Partch, it was not the case for those who tried to perform his music. He finally developed different notations for each instrument that were based on the layout and how the instrument was played. Still, Partch was present at each performance to coach players on how to correctly play the instruments, read the notation, and interpret the music.

In 1947, Partch published his book *Genesis of a Music*, where he explains in great detail his view that twelve-tone equal temperament was a terrible mistake and how he developed his own tuning system to correct this mistake. While still in school, Ben Johnston was given Partch's book by one of his professors who knew that he had an interest in just tuning and the harmonic series. Soon after reading Partch's book, Johnston wrote to him, and they arranged for Johnston to come to California where he could live and work with Partch. Although "studying" with him was surely Johnston's original intention, Partch insisted that he was not a teacher; Johnston would be more of an apprentice than a student. Johnston recalls, "He told me that if I or anyone else ever claimed to have been a student of his, he'd cheerfully strangle us; and I think he meant that very literally."[3] Partch was adamantly against the idea of influencing someone else's art, so he would never comment on Johnston's music. Partch simply had work that needed to be done, so in exchange for labor, he would answer Johnston's questions. Because Johnston did not have a talent for carpentry (which did not take long for Partch to notice), he was given mostly errands and chores around the property. One of Johnston's more musical tasks was tuning the instruments everyday. He had a very good ear to begin with and Partch took notice of this; soon Johnston was able to easily distinguish all forty-three tones in Partch's system. Although he was not teaching Johnston to train his ear in a formal way, it

[3]Johnston, "The Corporealism of Harry Partch."

is likely that Partch saw that this was his strong suit, and gave him this task so that he could develop his ear further. Through answering some of Johnston's questions and refining his ability to discern just intervals, Partch left him with a strong foundation in just intonation theory, while not directly interfering with how he composed. And although he was against the idea of influencing Johnston's music, being as opinionated as Partch was, he would still give Johnston pieces of advice like, "Never write a fugue or a sonata or anything else unless you have something to add to the tradition that nobody has added before you."[4]

While working with Partch in northern California, Johnston met Darius Milhaud, who immediately took an interest in him. Knowing that he had absorbed what he could from Partch, Johnston decided it was time to move on and went to study with Milhaud. Here Milhaud taught him about compositional process, but did not teach him how to compose. He felt that Johnston was talented enough to find his path with some guidance rather than shaping his music as he might have done with a weaker student. Before this period, Johnston still was writing very intuitively and had never been taught traditional methods of composing.

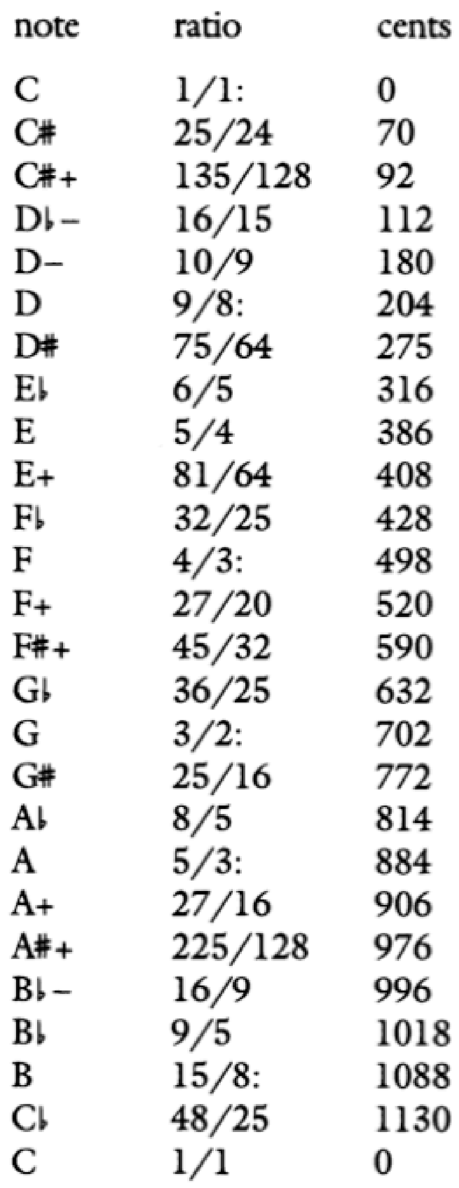
In 1951, Johnston joined the faculty of the University of Illinois as a professor of theory and composition. After attending a lecture given by John Cage at his university, they quickly became friends and Cage invited him to come to New York and study with him that summer. This, in a way was similar to his experience with Partch, as Cage did not play the typical role of a teacher, but rather he exposed Johnston to the New York avant-garde scene — which, incidentally, Johnston never identified with, nor felt that he belonged to. He returned to New York a few years later on a Guggenheim fellowship to

[4]Duckworth, *Talking Music: Conversations With John Cage, Philip Glass, Laurie Anderson, And 5 Generations Of American Experimental Composers*.

work in the electronic music studios at Columbia/Princeton. Knowing his limitations as a carpenter and that he would not be able to build his own instruments, Johnston thought that electronic music would be an ideal way to write music using just intonation, finally applying the skills he learned from Partch. He soon discovered that the equipment was not sophisticated enough to do what he wanted, and that he wasn't particularly good at making electronic music either. During this period, Cage invited him back to study with him in a more formal composition lesson once a month. Here Cage would critique his pieces, and they would also work on serial techniques, which Johnston was experimenting with at the time; in this regard, Cage would be a great help, having studied with Schoenberg himself.

Johnston took what he could from all three composers because he wanted a very "full" background. Unlike Partch, who detested music education as a whole, Johnston prepared himself for a very typical career as a well-rounded composer and professor of composition and theory. His works were largely composed in a neo-classical in style for around 10 years, with a brief period where he would explore serial techniques.

Johnston eventually decided that if he wasn't able to build his own instruments and the current electronic instruments were not sophisticated enough to do what he wanted, he would have to write music in extended just intonation for traditional instruments. He also decided to use traditional notation, but instead of treating the notes on the page as if they were equally tempered with accidentals showing how the just tones differ, his music made the assumption that the notes on the staff were a 5-limit C just intonation scale. Then, to get the pitches he needed to represent just ratios, he developed a system of accidentals that would raise or lower a pitch by a comma based on a particular prime number. Figure 2.1 shows each pitch with the corresponding ratio and cents offset from equal temperament.



note	ratio	cents
C	1/1:	0
C#	25/24	70
C#+	135/128	92
D♭–	16/15	112
D–	10/9	180
D	9/8:	204
D#	75/64	275
E♭	6/5	316
E	5/4	386
E+	81/64	408
F♭	32/25	428
F	4/3:	498
F+	27/20	520
F#+	45/32	590
G♭	36/25	632
G	3/2:	702
G#	25/16	772
A♭	8/5	814
A	5/3:	884
A+	27/16	906
A#+	225/128	976
B♭–	16/9	996
B♭	9/5	1018
B	15/8:	1088
C♭	48/25	1130
C	1/1	0

Figure 2.1. Johnson's tuning system.

The advantages and disadvantages of Partch's approach to the performance of extended just intonation seems to tend toward the extreme on either side. The obvious disadvantage is that a musician can't play his music without first being trained on the particular instrument. This puts great limitations on his music's complexity since even musicians who were willing to learn the instrument along with the music would require parts that could be easily learned in the amount of rehearsal time available for each performance. Additionally, the performer would have to learn a new form of notation to play the instrument. Considering these two rather large obstacles, Partch's method is not likely to attract virtuoso performers, who would be uncomfortable playing such foreign instruments. To Partch, these seemingly deal-breaking disadvantages would not be problematic at all: on the contrary, these obstacles keep those performers conditioned by western music and twelve-tone equal temperament far away from his music. As a result, Partch was not concerned about western habits creeping into his work, which adds to the approachability and accessibility of his music to the lay audience.

For Partch, the decision to build instruments was not a difficult one to make, since he already had basic carpentry skills that he developed in his youth. He did, however, have a very difficult time gathering tools and quality materials to build with since he never had a steady income to live on, let alone to buy supplies. This was very influential in the design of his instruments, because if it was not made of wood, it was usually made from found objects (such as the "zymo-xyl," which is made from whisky bottles).

Using traditional instruments with flexible intonation has clear advantages since the only effort required is either practicing yourself or convincing a performer to adjust his or her technique to play your music. With instruments like those in the traditional string family, it's almost as natural to play in just intonation as it would be to play in equal temperament (arguably more so). In the example of a violinist, the instrument is in no way an obstacle in the performance of extended just intonation. However, a classically trained violinist who has been trained to play in equal temperament may find it difficult to adjust his or her fingerings to just intonation. To complicate matters further, most just intonation composers are likely to require their music to be played with little or no vibrato to accentuate the natural beauty of the just intervals they have worked so hard to create. Some would argue, as I believe Partch did, that the cost of overcoming these obstacles is far greater than that of teaching someone to play a new instrument.

Modified traditional notations have similar drawbacks. While Johnston assumes that even unmodified notes on the staff are derived from just intervals, this requires the performer to be comfortable playing these pitches first before they can attempt to raise or lower the pitch according to specialized accidentals. While this makes Johnston's music very approachable on the surface, and considerably easier to understand from the point of someone who would analyze his work, it is still enormously difficult to perform with even a modest level of accuracy. Performing this music assumes knowledge of just ratios that very few performers have and even fewer would be willing to learn in order to perform a piece of music. It is my belief that the vast majority of trained performers will play the music from a "this far from equal temperament" approach rather than reevaluating everything they know and have learned about where their pitches lie.

Although it would be difficult to compare the success (monetary or otherwise) of the two composers explored in this paper, because such a thing is wholly subjective, I attempt to do so here for the sake of this argument. If at any time in his life, financial success (or even stability) appeared to be within reach, Partch surely would have sabotaged the opportunity with either his unapologetic stubbornness concerning his musical aesthetic, or his sheer brashness alone. An inability (or unwillingness) to maintain steady employment aside, Partch detested the idea of music in higher education, which has been considered the most common, and more importantly, sensible career path for generations of composers, including his former apprentice, Ben Johnston. It is of little use to speculate about what Partch could have accomplished for himself, his music, and the practice of composing and performing using extended just intonation had he taken a more traditional approach to new music; but to imagine Partch doing this is to not imagine Partch at all. With every new school and trend that arose in the twentieth century came the impression of a new found musical freedom: a release from the oppression of tonality. However, when you look at this from Partch's perspective, it starts to look more like a prison than ever before. It is his complete stubbornness to uphold his aesthetic, no matter what the cost, that made it possible for him to accomplish what he did in his lifetime. No school in his day (or perhaps even today) would allow him this kind of artistic freedom without significant compromises, which Partch was simply unwilling to make.

Johnston on the other hand was very quick to join the faculty of the University of Illinois early in his compositional career, where he would be a professor of theory and composition for over thirty years. It was never his intention to completely reinvent music the way Partch did; Johnston felt very much a part of the western tradition, and was happy to contribute to it. This is made evident by his studying with other composers such as Milhaud and Cage, as he wanted very much to be a "well-rounded" composer. He made his commitment to using just intonation not because he was unsatisfied by contemporary music itself, but rather because he "thought it was out of tune."[5] Although his use of just intonation would go well beyond 5-limit (which could be used to "tune" equally tempered

[5]Duckworth, *Talking Music: Conversations With John Cage, Philip Glass, Laurie Anderson, And 5 Generations Of American Experimental Composers*.

music), he approached it as a natural progression of European music just as he viewed twelve-tone equal temperament as a necessary mistake to correct. This allowed him to be quite successful in academia without compromising his own aesthetic.

Although Johnston's music doesn't receive nearly as many performances as many other experimental or avant-garde composers do, it has attracted many highly accomplished performers that are willing to make a commitment to performing his music accurately. The flutist/composer John Fonville has not only recorded Johnston's *Twelve Partials for Flute and Microtonal Piano*, but he has also written an excellent article on the performance of extended just intonation, with a clear explanation of Johnston's notation from a performance perspective. More recently is the founding of the Kepler Quartet, whose sole purpose is to perform and record the complete set of Johnston's string quartets, which are his most widely known works. [6]

Because of the difficulty in putting together performances of Partch's music, it seemed logical that recordings would be an important part of preserving his music. Unfortunately, most all of his later music is highly theatrical in nature, so the performances are poorly represented on audio recording. His most grand theatrical work, *Delusion of the Fury*, is an example of this: the film version of the work represents the theatrical elements poorly, much to Partch's dissatisfaction.

Partch's music is still quite difficult to perform today, regardless of the performer's dedication to the music. His instruments are currently housed in the Newbandinstrumentarium at Montclair State University. Composer and former Partch musician, Dean Drummond, currently directs Newband, the microtonal ensemble that is still performing Partch's

[6]Formed in 2002 by Sharan Leventhal, Eric Segnitz, Brek Renzelman and Karl Lavine. http://www.keplerquartet.com/

music on a mix of original and replica instruments. There have been transcriptions of his music for traditional instruments, the most famous of which was a transcription of *Barstow*, done by Johnston, which was recorded by the Kronos quartet.[7] As noble an effort as this was, those who worked closest with Partch later in his life agree that he would have most likely hated it.

Johnston's notational system has served as a valuable tool to the next generation of composers using just intonation. David Doty uses the Johnston notation for all of the examples in his book, *The Just Intonation Primer*, which often serves as an introduction to young composers interested in the topic. Some of Johnston's students have also continued to use his notation system, such as Toby Twining, who's choral work *Chrysalid Requiem* uses 13-limit extended just intonation. Another former student of Johnston, Kyle Gann, has written extensively on the topic of just intonation, and has composed several works using the Johnston notation for both traditional and electronic instruments. While Partch continues to inspire and provide guidance to composers interested in just intonation through his book, *A Genesis of a Music*, this is generally the extent of direction one can receive from his legacy.

[7]Daugherty et al., *Howl U.S.A.*.

CHAPTER 3

INSTRUMENT DESIGN

Perhaps the most problematic part of performing justly intonated music is the way in which a performer interfaces with an instrument. While both designing new instruments with unfamiliar interfaces and working with familiar instruments have had varying degrees of success, neither is approachable enough to be widely practiced. One limitation historically has been the tight coupling of interface and sound production. For example, the design of a piano is ideal for using alternate tunings. Each pitch has its own string and can be tuned independently. However, the keyboard itself is immutable, so the use of extended just intonation with pianos is conceptually limited by its interface. A violin on the other hand is an example of an interface well suited for varying tunings, but puts all of the responsibility for accuracy on the performer.

With the human interface decoupled from the sound production mechanism, the design of an instrument with variable tuning capabilities can be approached from a different perspective. As seen in Partch's instruments, the interface is usually tied to a particular system of tuning, thus limiting the versatility of the instrument. When the interface is abstracted, the problems of accuracy and playability can be approached separately. The tuning systems used and the theory behind the tuning system will likely change with each composer and piece. On the other hand, an instrument with varying tuning capabilities could be used in any situation where non-standard tunings are required. It therefore makes sense to treat the interface as variable and instead build instruments that can be dynamically tuned with a high degree of accuracy with only an application programming interface (API) to interact with them.

The use of robotics in musical instruments has increased considerably in the last few decades. Instruments like Yamaha's Disklavier, a modern MIDI controlled player piano, have been embraced by composers who can use software to create piano music beyond the capability of a human performer. While very capable, these instruments are quite expensive and their flexibility is limited to what is necessary to emulate a human performance.

As the price and size of microcontrollers and components has reduced, building small robotics projects has become much more accessible. The term "physical computing" has been used to describe this trend of programmers and artists controlling hardware with software. As described in *Physical Computing*, "Physical computing is about creating a conversation between the physical world and the virtual world of the computer."[8]

A new generation of musical instrument makers have emerged, building custom, inexpensive instruments using physical computing techniques. In 2000, musician/engineer Eric Singer founded the group LEMUR,[9] where they build robotic acoustic instruments and put on performances. The first instrument they built, the "GuitarBot",[10] consists of four guitar strings, each having its own metal body, with a bridge that moves up and down the string on a track to change the pitch. The separating of each string allows for a more modular design as well as more room for electronic components, which could not fit within a typical guitar string spacing. This design choice proved to be influential to my own work. And although the pitch can be controlled with a high degree of accuracy using the movable bridge, this mechanism works more like a guitar slide, making large leaps by shortening

[8]Dan O'Sullivan and Igoe, *Physical Computing: Sensing and Controlling the Physical World with Computers*.
[9]*LEMUR: Purveyors of Fine Musical Robots Since 2000*.
[10]Singer et al., "LEMUR GuitarBot: MIDI Robotic String Instrument."

the length of the string vibrating over the magnetic pickup, as opposed to a tremolo system on a guitar that would change the string's tension.

Dynamically Tunable Bridge

Very few instruments are designed to be tuned dynamically without the performer being responsible for accuracy. The pedal steel guitar is a unique exception in that it uses pedals and knee levers to adjust the tuning of the strings up to a full step, with a very high degree of accuracy. Often seen in country music, the pedal steel is played with a slide bar, usually made of glass or steel, placed perpendicularly across the strings. Compared to a traditional guitar, it has more strings and is tuned in smaller steps, usually a half step to a fourth apart. Because the left hand holding the bar cannot easily change the pitch of one string independently of the rest, tunable knee levers and pedals are used to raise or lower individual strings. A series of rods underneath the body of the instrument are attached to the pedals and levers and they can travel a certain distance based on how they are tuned.

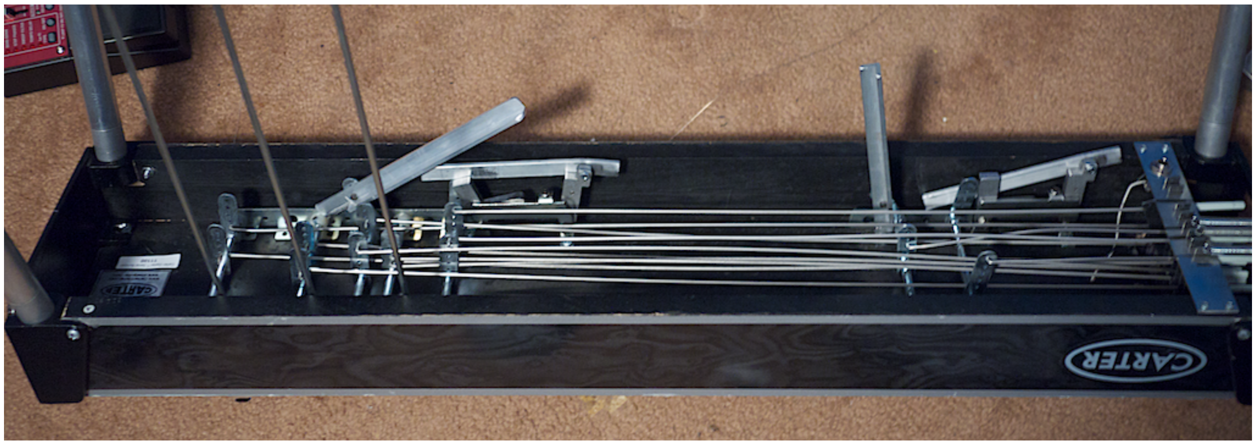


Figure 3.1. Steel guitar rods.

Each rod is connected to what is called the "changer", which also acts as a bridge for each string. The top of the changer, called the "finger", is rounded so that the string sits naturally as the rod pulls it forward or backward, changing the tuning of the string.

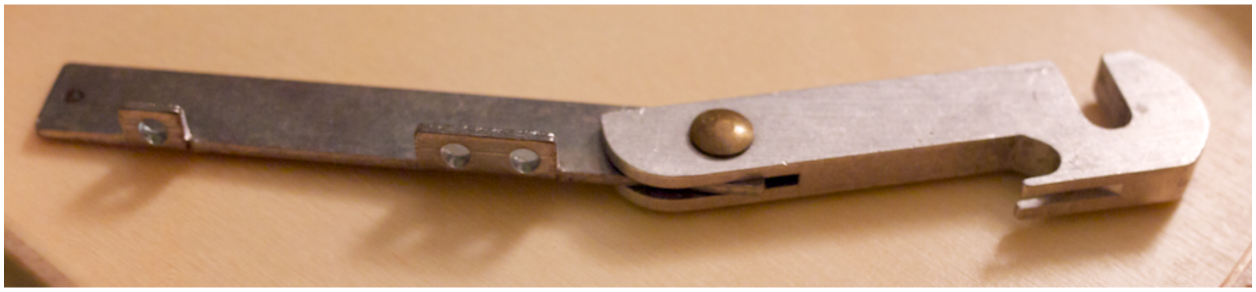


Figure 3.2. Steel guitar changer.

The design of the changer is quite elegant and an ideal source of inspiration for a dynamically tunable instrument. If the rods are removed and instead the changer is moved by a stepper motor or servo, the tuning of the string could be changed in much smaller intervals, with a high degree of accuracy.

OSC as an Interface

Abstracting the interface from sound production is not a new idea. The MIDI messaging protocol has allowed for MIDI controllers to control any type of sound production device that accepts these messages, usually in the form of a synthesizer or sampler. Wendy Carlos took advantage of this in her piece, “That’s Just It”, using two keyboard controllers.[11] One is played traditionally using a JI tuning and the second controls what pitch the tuning is based on by switching to a different table of tunings.[12] This is a very sensible approach given the technology available at the time. It allowed composers like Carlos to explore JI tuning systems while completely avoiding issues related to instrument building and performers of traditional acoustic instruments. While it is arguable whether or not it is an advantage, the consistent timbre produced by synthesizers of this era would make it easier for a listener to hear the sonic purity of a JI tuning system.

[11] Carlos, “Beauty in the Beast.”

[12] Milano, “A Many-Colored Jungle of Exotic Tunings.”

When viewed from an API perspective, MIDI still works well for many applications. And although improvements have been made recently to allow MIDI over IP networks and higher resolutions than the original 7-bits, it still lacks the flexibility and ease of use that more modern communication protocols provide.

The Center for New Music and Audio Technology (CNMAT) at UC Berkeley developed Open Sound Control (OSC) as an alternative to MIDI. While it addresses some of the limitations of MIDI, it was also designed with the modern computer networking stack in mind. Because low latency is often more important than reliability and message ordering in real time music applications, OSC is often transmitted over the UDP networking protocol, although TCP can also be used when it is preferred.

Another advantage of OSC over MIDI is its concept of "bundles." Concerning MIDI's limitations in this area, Director of CNMAT, David Wessel, wrote, "MIDI provides no mechanism for atomic updates. Chords are always arpeggios and even when MIDI events are time tagged at the input of a synthesizer they arrive as a sequence."[13] An OSC bundle allows you to group messages together and send them as a single unit rather than several messages in succession. This more closely represents how music is communicated given that chords are best represented as an array of note messages, and would ideally be sent as a single message.

Because of these advantages, OSC is the ideal choice when designing a generic API for musical instruments. It is well supported in most all of the popular programming environments and can be easily interfaced with hardware by using a micro-controller like the Arduino[14].

[13]Wessel and Wright, "Problems and Prospects for Intimate Musical Control of Computers."

[14]A small, inexpensive, Atmel AVR based micro-controller. http://arduino.cc/

CHAPTER 4

IMPLEMENTATION

To demonstrate the ideas I have described previously, I designed a simple API for dynamically tunable instruments, and both a software instrument that implements the API and a hardware prototype capable of using the same API.

API

The API aims to be a RESTful[15] design on top of OSC. The only two resources are notes and pitch classes, as indicated below:

Notes

Resource	Description
/inst/:id/note/:pitch/play :velocity :duration	Plays a note when the duration is known.
/inst/:id/note/:pitch/on :velocity	Turns a note on.
/inst/:id/note/:pitch/off	Turns a note off.
/inst/:id/note/:pitch/tuning/offset/cents	Returns a pitch's current tuning offset in cents.
/inst/:id/note/:pitch/tuning/offset/cents :offset	Sets a pitch's tuning offset in cents.
/inst/:id/note/:pitch/tuning/offset/frequency	Returns a pitch's current tuning offset as a frequency.
/inst/:id/note/:pitch/tuning/offset/frequency :offset	Sets a pitch's tuning offset as a frequency.

Pitch class

Resource	Description
/inst/:id/pitch-class/:pitch-class/tuning/offset/cents	Get the current offset in cents for a pitch class.
/inst/:id/pitch-class/:pitch-class/tuning/offset/cents :offset	Set an offset in cents for a pitch class.
/inst/:id/pitch-class/:pitch-class/tuning/ratio	Get the current ratio for a pitch class.
/inst/:id/pitch-class/:pitch-class/tuning/ratio :ratio	Set a ratio for a pitch class.
/inst/:id/pitch-class/tuning-base :pitch-class	Set a pitch class to calculate ratios from.

[15] Fielding, "Architectural styles and the design of network-based software architectures."

The note resource can be used to sound a pitch at a given velocity with a predefined duration, as well as a more MIDI-like on/off interface. The tuning of a particular pitch is both obtained and set as an offset of equal temperament, either in cents or as a frequency when using the note resource.

Specifying pure, just intervals by using offsets of equal temperament may seem like an odd choice. From the perspective of a composer that uses just intonation, equal temperament introduces complexity that does not otherwise exist. Calculating scales using just intervals, although unfamiliar to most, is rather simple. Given a base frequency, all other frequencies can be derived by multiplication of their ratio and the base. Calculating offsets of equal temperament is considerably more convoluted, as illustrated by the ruby example in Figure 4.1:

```
def midi_to_et_frequency(pitch)
  pitch_class = pitch % 12
  440 * (2**((pitch - 69) / 12.0))
end

def ratio_to_cents_offset(pitch, ratio,
                          base_pitch_class = 0, tones_per_octave = 12)
  pitch_class = pitch.to_i % tones_per_octave

  f = ratio[0].to_f / ratio[1].to_f
  et_frequency = midi_to_et_frequency pitch
  ji_frequency = f * et_frequency
  pitch_cents_offset = (3986.3 *
             ((Math.log(ji_frequency) / Math.log(10)) -
              (Math.log(et_frequency) / Math.log(10))))

  if pitch_cents_offset != 0
    pitch_cents_offset = pitch_cents_offset -
      (((pitch_class - base_pitch_class) % tones_per_octave) * 100)
  end

  pitch_cents_offset
end
```

Figure 4.1. MIDI note numbers to frequency offset.

Regardless of how a composer may feel about equal temperament, it is considered a standard. Many musicians are accustomed to thinking about tuning in cents, so this can provide a familiar interface when appropriate. Also, this calculation can be easily abstracted at the API level, so that the composer would never have to think in terms of equal temperament offset, as this would be the responsibility of the API client application.

The pitch class resource however, does provide a way to set a particular pitch class to a ratio, and then set a base pitch class from which to derive each ratio. For example, to setup a typical five-limit just intonation tuning based on G, you would send the messages in Figure 4.2.

```
/inst/0/pitch-class/tuning-base 7
/inst/0/pitch-class/1/tuning/ratio 25:24
/inst/0/pitch-class/2/tuning/ratio 9:8
/inst/0/pitch-class/3/tuning/ratio 6:5
/inst/0/pitch-class/6/tuning/ratio 45:32
/inst/0/pitch-class/7/tuning/ratio 3:2
/inst/0/pitch-class/8/tuning/ratio 8:5
/inst/0/pitch-class/9/tuning/ratio 5:3
/inst/0/pitch-class/10/tuning/ratio 9:5
/inst/0/pitch-class/11/tuning/ratio 15:8
```

Figure 4.2. Five-limit API example.

Software Instrument

For simplicity and easy availability, the software implementation is developed using a number of open source tools. Due to the asynchronous nature of musical events, I also needed something that would provide non-blocking I/O so that one OSC message would not be delayed by another which had not yet responded. Given these requirements, I chose to build this using javascript and HTML, with a server-side component in ruby. The browser is an ideal tool to build user interfaces using just HTML, CSS, and javascript. However, to receive OSC messages, they must be passed through a separate server since the browser doesn't have direct access to the TCP/IP layer, but it does support a new protocol called WebSockets, which allows persistent connections to remote servers.

EventMachine[16] is a ruby library that provides event-driven I/O using the reactor design pattern,[17] which allows for network communication in ruby that doesn't block the main thread while waiting for a response. To be able to send OSC messages to the browser, a small OSC server is needed that can take incoming messages and relay them to the browser via WebSockets. I implemented this server using two EventMachine-based libraries, em-websockets and em_server from the osc-ruby package. Both the WebSockets and OSC servers run in their own thread and forward messages as they are received [18].

When a user opens the web page that contains the software instrument, a WebSocket connection is made to the EventMachine server. The server can then send the translated OSC messages to the browser to turn notes on or off, or alter the tuning of a particular pitch.

I chose to use a piano keyboard for the user interface, and although the keyboard is an inherently limited interface, in that it implies twelve tones to the octave, I don't necessarily consider this a disadvantage. With the decoupling of the interface, a key's response is no longer fixed. This provides tuning flexibility while still retaining a familiar interface. In cases where many small intervals are needed that cannot be represented by offsets of equal temperament, the base pitches of each key can be completely remapped, similar to Partch's Chromelodeon [19].

To build the user interface, I used a lightweight JavaScript MVC framework called

[16]Cianfrocca and Gupta, *eventmachine (0.12.10)*.

[17]Schmidt, *Reactor – An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching*.

[18]See Appendix

[19]The Chromelodeon is a standard reed organ that Partch modified to use his 43-tone scale.

Spine.[20] This allowed me to easily decouple user interface elements from sound production while still allowing communication between these layers using event messages. The interface consists of HTML elements that are styled using CSS to resemble a piano keyboard. The piano keys can be triggered by clicking on them, pressing a particular key on the computer keyboard, or by messages sent through WebSockets.

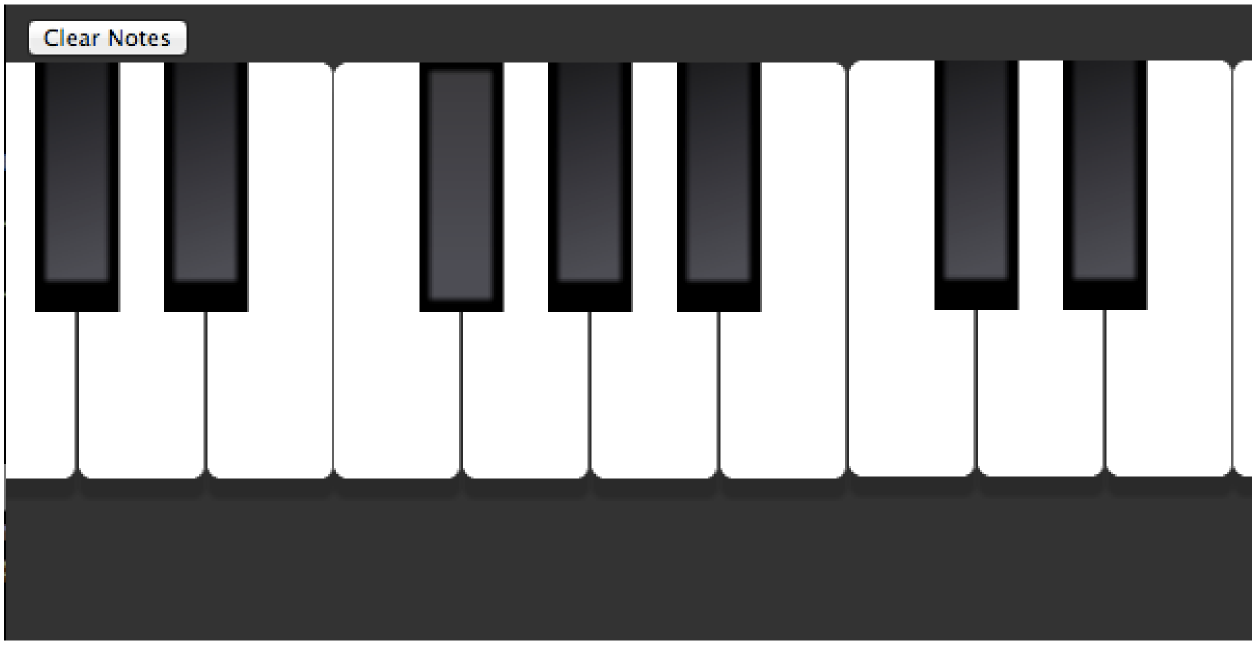


Figure 4.3. Software instrument user interface

For generating sound in a web browser, there are a few options. This has previously been done using java applets, midi files, flash, or the more recent HTML audio element[21] which can play wav files. Of these options, the HTML audio element is the best choice, considering that web development is moving away from plugins and towards native APIs. However, implementing an instrument using this technique would require either loading individual audio files for each pitch and tuning offset, or using one file per pitch and controlling the tuning with playback speed. Neither of these options are ideal, as the

[20]MacCaw, *Spine (1.0.5)*.

[21]*The audio element — HTML5*.

former would require generating a prohibitively large number of files while the latter would have an undesirable affect on the quality of the sound.

A newer development in web browser technologies is the Web Audio API, “a high-level JavaScript API for processing and synthesizing audio in web applications.”[22] While the quality of sound achieved with synthesis in a web browser is currently limited by available CPU and the efficiency of the APIs themselves, this option provides the most flexibility for building an instrument with variable tunings.

Using this API, I built a simple synthesizer that encapsulates the sound production in a “Synth” object, which has a settable frequency. These objects receive messages from “Note” objects, where both the pitch and the tuning offset can be set and the pitch/tuning to frequency calculation is abstracted within this object’s methods.

```
class Note extends Spine.Model
  # @param {Number} pitch A midi note number.
  constructor: (@pitch) ->
    @etfreq = 440 * Math.pow(2, ((@pitch - 69)/12))
    @freq = @etfreq
    @state = off

  on: ->
    @off() if @state
    @state = on
    # Use etfreq to prevent hanging notes who□s frequency has changed.
    Spine.trigger ”synth:on”, @etfreq

  off: ->
    @state = off
    Spine.trigger ”synth:off”, @etfreq

  # Apply a tuning offset.
  #
  # @param {Number} offset An offset value in cents.
  offsetCents: (offset) ->
    @freq = @etfreq * Math.pow((Math.pow(2, 1/1200)), offset)
    Spine.trigger ”synth:setfrequency”, @etfreq, @freq
```

Figure 4.4. Note class

This allows for a more modular design as the Synth object’s only responsibility is to

[22] *Web Audio API*.

generate a waveform at a given frequency. Changes to how Notes objects are used within the application or how tuning offsets are communicated should not require changes to the Synth class. The Note class is also decoupled from any interface elements or methods of sound production, making it similarly flexible within the design.

While this software instrument is rather basic, it is an example of how the API can be used to quickly test client software. This is useful for both educational purposes as well as aiding in the development of a piece where the performance will interface with a hardware instrument.

Hardware Instrument

The design and craftsmanship of acoustic instruments is a complex topic that is beyond the scope of this project. Because of this, I chose to identify areas that could be simplified without compromising the overall goals of the project: namely, pitch accuracy and flexibility. Since the piano is one of the best examples of pitch accuracy, I chose to start with it's design, and then remove or alter components to make it simpler and less expensive to build.

Fortunately, the quality of the sound produced acoustically, which would ordinarily be one of the most difficult goals to achieve in the building of an instrument, is not a high priority with the current project. Because of this, I decided to borrow from the design of the electric guitar and use a solid hardwood body, relying on magnetic pickups for amplification. Also, since guitar strings come in many different sizes and lengths, it is possible to use common and inexpensive guitar parts, such as tuning keys.

I decided to build this prototype in two pieces. The body consists of two one-inch-by-three-inch pieces of oak held together by two large bolts, both at the opposite end of the bridge. This allowed me to adjust the distance of the two pieces as necessary to accommodate changes in the design of the tuning mechanism. Although it would have provided more support for the body, I avoided placing the second bolt on the side of the tuning bar in order to keep the area clear until the design was finalized.

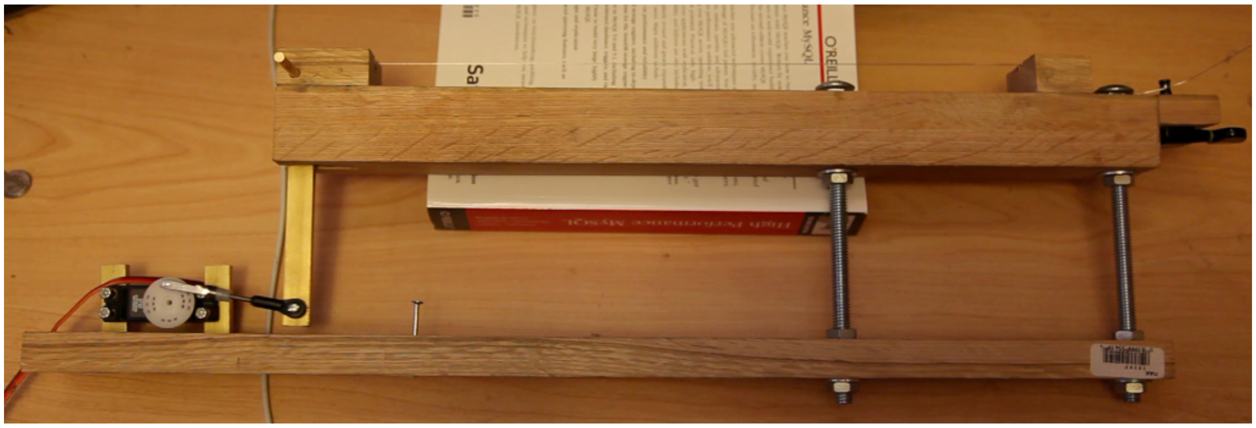


Figure 4.5. Prototype body

Given that guitar tuning keys are designed to be turned by hand, they have a very low gear ratio. Using this as the means to alter the tuning is not ideal because it would require a too much physical motion to get to the desired tuning, which would be a very demanding requirement for a servo. Instead, I designed a simplified version of the pedal steel bridge and changer, consisting of a rectangular brass bar with a slot filed out at the top for the string to rest on, a hole just below the top so that a rod can hold it in place while allowing it to freely rotate, and a hole at the bottom of the bar so that it can be moved back and forth to change the tuning.

For the method of sound production, I experimented with several different designs that combined the hammer and damper into one mechanism. The benefit of having the two functions in one is that it only required one physical action to operate, similar to a piano key. If the hammer was separated from the damper, it would have required multiple servos or solenoids, with code to keep them synchronized.

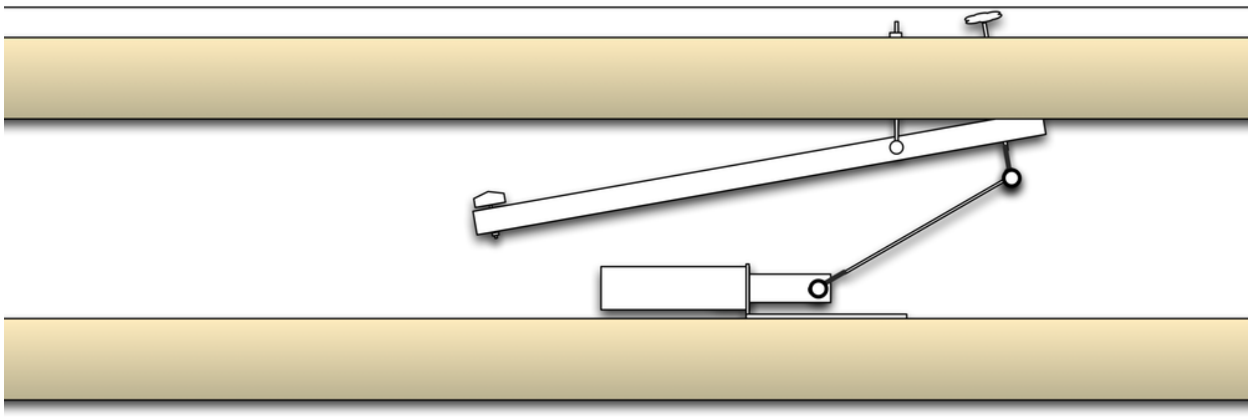


Figure 4.6. Hammer/damper diagram

However, this turned out to be more difficult than I had planned. The best version of this mechanism that I had built still had the problem of limited dynamics and speed.

Because the method of sound production is not the focus of this project and due to the difficulties descibed previously, I chose to simplify this area of the prototype. Instead of a hammer-type motion, I replaced it with a plucking mechanism using a guitar pick attached to the arm of a servo. This design does not have the ability to dampen the sound gracefully, but can stop the string from vibrating rather abruptly by moving the servo's arm to where the pick is touching the string, but not far enough for it to move past, causing the string to sound. Dynamics are also limited with this design, but can still be controlled by the speed of the servo's arm and the starting distance from the string before the arm is moved. Given these comprimises, the sound produced is more than adequate for testing the tuning mechanism, and could even be appropriate in some cases for performance.

CHAPTER 5

TESTING

Hardware Testing

As simple and effective as the tuning bar design was, I discovered several flaws that introduced unwanted variance into the tuning.

The servo arm is connected to the tuning bar using a rod with a ball link end, usually found on remote control helicopters, to connect the arm to the bar (Figure 5.1). This allowed for easier prototyping since the tuning bar and the servo arm don't have to be perfectly aligned to function, and hobby servos don't typically mount on their sides.

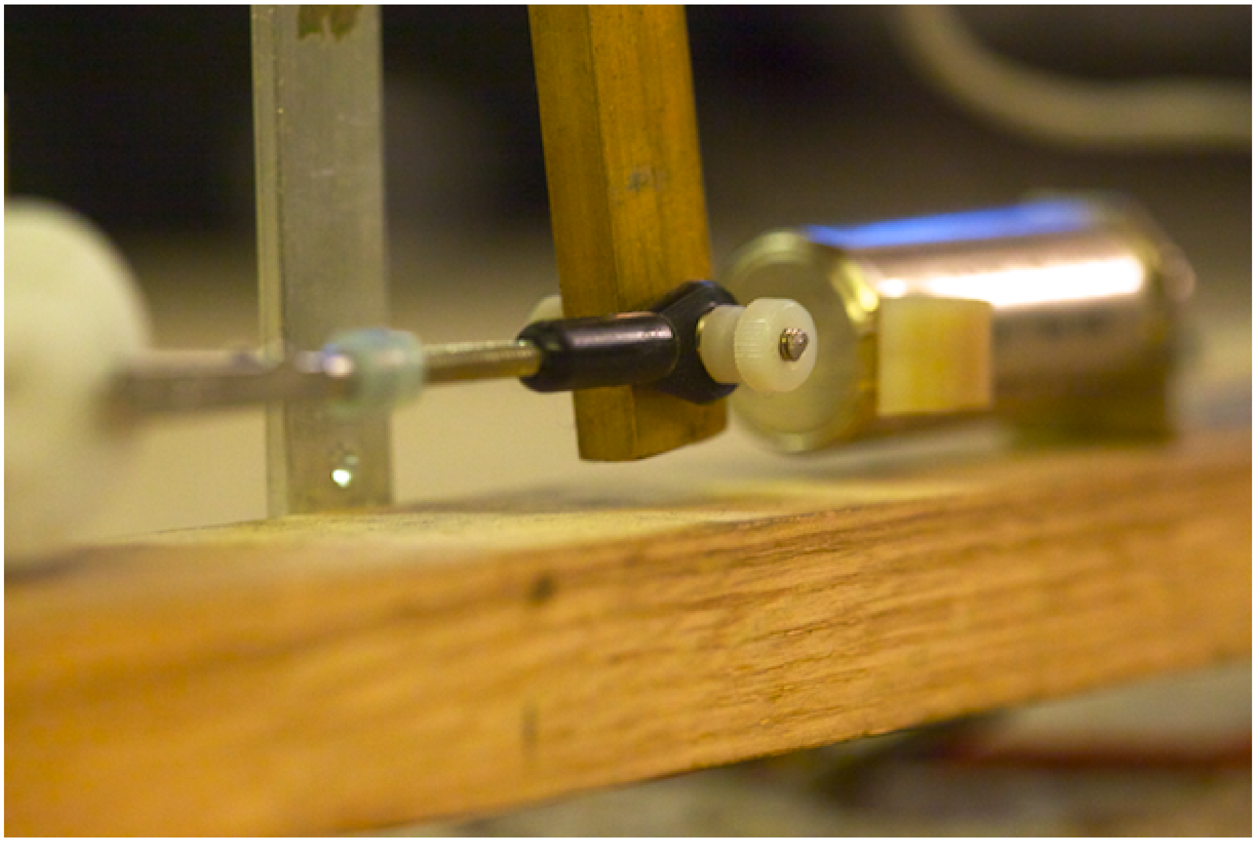


Figure 5.1. Ball link end attached to tuning bar

While this could potentially work well enough on it's own, there was too much space where the bar could move on either side of where the string is held. This caused the servo to pull the bar back and then slightly to the side, so that at higher tensions, the bar moving side to side would cause the tuning to fluctuate. To minimize this behavior, I removed enough space on either side of the bar so that the bar could be held tightly in place with two washers on either side (Figure 5.2).

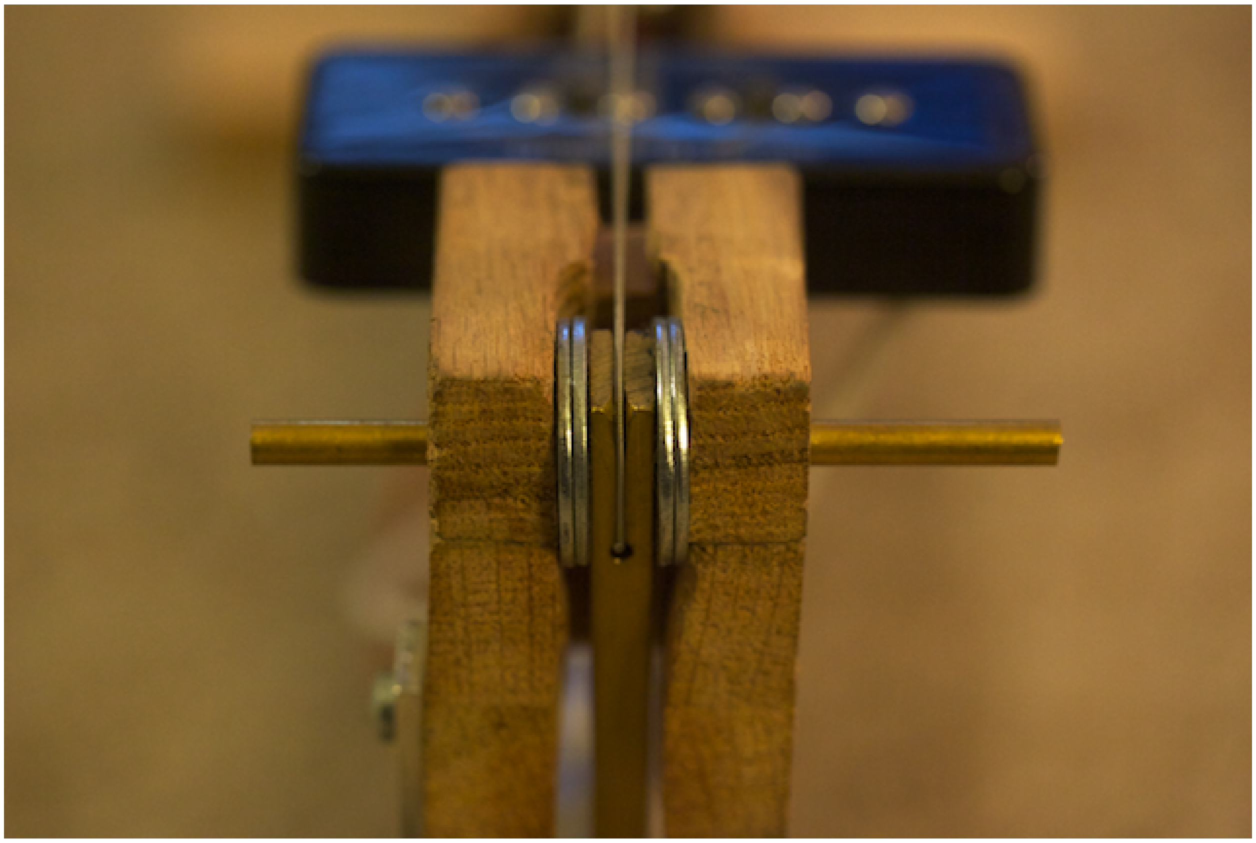


Figure 5.2. Tuning bridge

Although the two-piece body was very convienient to prototype with, as the servo pushed the tuning bar forward and increased the string tension, the top half of the body would bend slightly downward, which introduced small variations in the tuning. To remedy this issue quickly, I added an aluminum bar to one side of the end needing support. Another issue I found was that when making small tuning changes, the string could not move as freely as necessary across the wooden bridge. This issue is much more prominent when using a wrapped string, like the sixth, fifth, and fourth string of a guitar. To fix this problem, I replaced the wooden bridge with a small brass wheel, similar to a “roller bridge” commonly found on electric guitars with tremolo systems (Figure 5.3). This allowed the string to move freely as the tension changes.

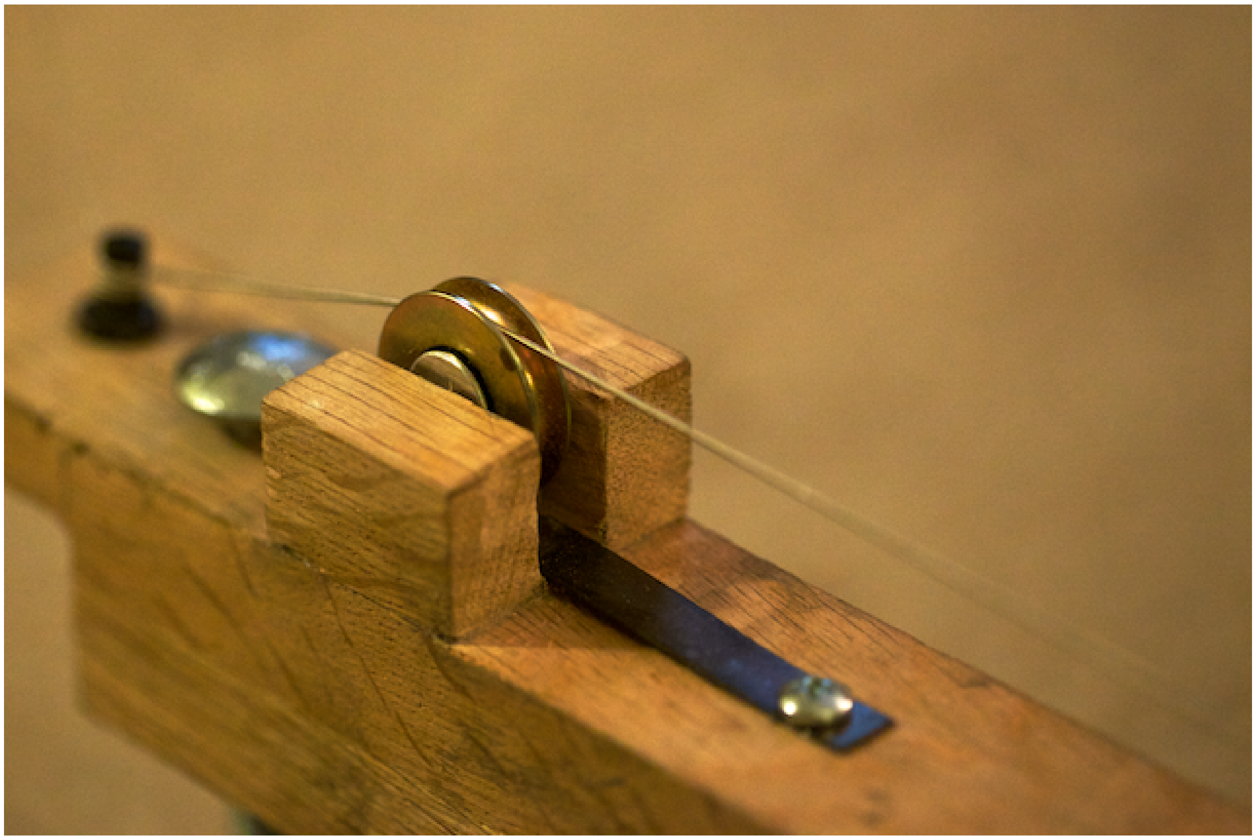


Figure 5.3. Roller bridge

After finishing these improvements, I decided to run a test to measure the accuracy and consistency of pitch when moving the tuning bar with the servo. The test script starts at a central location, and then moves 500 steps in both directions (in 10-step increments), plucking the strings after each change. While testing the instrument beforehand, I observed that small tuning changes were often very accurate, but inconsistencies would be introduced after making larger leaps. For example, if I brought the tuning bar from the center to the top of it’s range, then from the top of the range to the bottom, and back to the center, the tuning of the center point could be slightly off from the original starting point. This is likely because the large change in string tension caused parts of the body to shift slightly. To take this flaw into account, I had the test script return to the center point after each tuning change.

```
#!/usr/bin/env ruby

require "logger"
require "./tuner"
require "./plucker"

tuner = Tuner.new
plucker = Plucker.new
logger = Logger.new("logfile.log")
middle = 6800

def tuning_bar_test(servo_step)
  tuner.send middle
  sleep 0.5

  plucker.pluck
  sleep 1

  tuner.send servo_step
  plucker.pluck
  sleep 2

  logger.info servo_step
end

tuner.send middle
sleep 1

# Test upper range
(0..500).step(10).each do |n|
  n = middle - n
  tuning_bar_test n
end

# Test lower range
(0..500).step(10).each do |n|
  n += middle
  tuning_bar_test n
end
```

Figure 5.4. Tuning bar test script

While running the test, I recorded the output of the pickup so that I could run a pitch analysis and compare the results against the log file from the script. To perform the pitch analysis, I used an audio processing library called Marsyas.[23] This gave me a file with the fundamental frequency of the string for every 1024 samples. I then plotted each log entry against these pitches to produce the plot in 5.5.

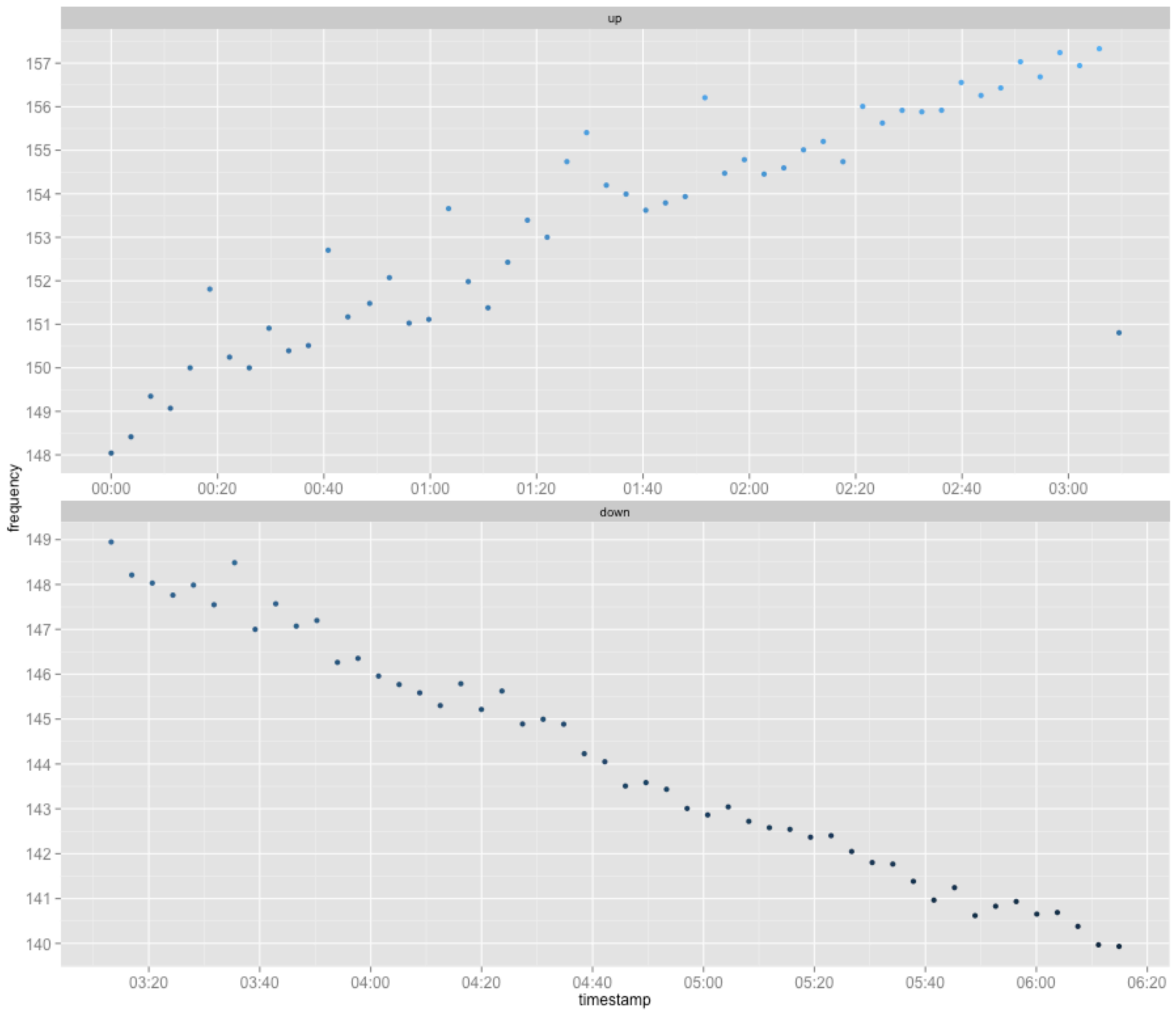


Figure 5.5. Tuning bar test pitch analysis

There is too much inconsistency in pitch changes, although this is clearly related to string tension. As the pitches go up and the tension increases, it is causing components to shift slightly, which unfortunately has a strong impact on the pitch. However, in the run where the pitch is lowered, it is considerably more consistent.

[23]Tzanetakis, *Marsyas (0.2)*.

The amount each step is off from the expected pitch in the descending run is mostly below 0.5Hz. While it may be difficult to perceive these inconsistencies, considering that a syntonic comma in this range is between 1.75-1.97Hz, it still requires some improvement before it becomes reliable enough for performance.

Client Software Application

One example of how this API could be used is to analyze a stream of pitches being sent and dynamically send tuning changes based on the analysis. If we wanted to solve the problem of not being able to modulate when using a basic five-limit just intonation, we would need to store the current key, identify potential key changes, and then send out tuning offsets based on those key changes. It would be difficult to implement this in a way that would work in all styles of music, considering rapid key changes and tonicizations would need to be accounted for, but we can use a much simpler algorithm for the sake of this example.

If we have a client application that is sending out pitch messages, we can have it hold a buffer of the last n notes that it sent out for analysis. It would then call a "detect_key" method as each new note is received to see if a key change can be detected. A very simple way to do this is to compare the buffered notes with each major scale, and then consider the scale with the highest intersection to be the current key. The example in Figure 5.6 shows how this could be done in a ruby client.

This is also an example of how tuning changes can be handled by "middleware," which can be inserted between the interface that is sending out pitch messages and the OSC listener producing sound. It would be ideal to design client software where middle-ware could be plugged-in, as changes to the tuning algorithm can be made independently of the rest of the application.

```ruby
def initialize
  # Start with a major scale based on pitch class 0.
  @scales = [[0, 2, 4, 5, 7, 9, 11]]
  # Iterate on this scale to generate the rest of the keys.
  10.times {|i| @scales << @scales[0].map {|n| (n + i) % 12} }
  @base = 0 # Start with a tuning base of 0.
end

def detect_key(notes)
  results = {}
  # Convert the notes to pitch classes
  pclasses = notes.map {|n| n % 12}
  # Compare our array of pitch classes against each major scale, and
  # record the size of the resulting intersection.
  @scales.each_with_index do |scale, i|
    results[i] = (pclasses & scale).size
  end
  # Determine the nearest key by taking the scale with the
  # largest intersection.
  closest_matches = results.max_by {|k,v| v}
  # Take the first result and ignore keys tied for a match.
  change_base closest_matches[0]
end
```

Figure 5.6. Key detection example.

CHAPTER 6

FURTHER IDEAS AND POTENTIAL IMPROVEMENTS

Hardware

The ideas and designs of the hardware instrument prototype could be expanded and improved to build larger, more versatile instruments. A multi-string version is the natural next step, either a large instrument that resembled a small piano in range or multiple one-to-three octave instruments. Smaller multi-string instruments would be more practical considering that the space needed for the tuning servo makes the distance between each string greater than would be afforded on a piano or a guitar.

One challenge that would arise with multi-string instruments is calibration. Both the string gauge and the scale length would introduce variance in the degree the servo has to move to tune the string to a particular offset. The application controlling the servos would then need to be aware of these positions for each string. Since this process would likely become laborious given more than a few strings, it could be greatly simplified with automation. One could write a calibration program to move the servo's tuning arm by a small number of steps incrementally, striking the string at each step, while analyzing the audio signal and recording the frequency. The program could then generate a configuration file to be used during a performance that is tuned specifically for that environment.

Software

The example algorithm for making live tuning changes based on pitch analysis could be improved in several ways depending on the needs of the composer. The simplest change that would result in a higher degree of accuracy would be to limit the number of scales that the algorithm can test for. In the case where a piece does not modulate into all twelve keys, simply removing those it does not use would result in fewer false positives. The available scales could also change dynamically based on the current key, if the key transition patterns are known. This could provide enough information to correctly identify the modulations in a piece while retaining enough flexibility to not require explicit key changes or score following.

If common practice tonality can be assumed, the algorithm could be greatly improved by applying a weight to each available key based on the current key. Consider the example in Figure 6.1:

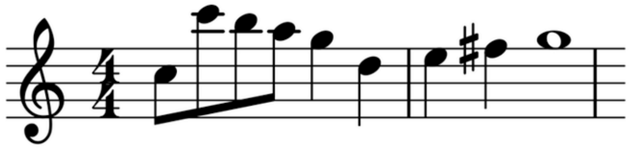


Figure 6.1. Tonality example

When the last five notes are being analyzed, there are only two potential candidates for the key change: G and D. When the F#4 is viewed as a leading-tone to G, which would be harmonized with V/V, this would indicate a modulation to G that is prepared with a secondary dominant. If this were a modulation to D, it would be abrupt, which is certainly possible, but less likely. We could also apply a slightly higher weight to G based on key proximity, as it is more common to modulate to closely related keys.

Each of these examples of how the algorithm can be improved can also be viewed as evidence that the tuning API is designed at the appropriate level of abstraction. The requirements of a dynamically tunable application will vary considerably based on the needs of the piece and the composer, so the API is limited in the assumptions it can make.

The previous examples show a wide range of needs that can all be easily accommodated with the current design. An argument could be made that because so few parameters are assumed, the usefulness of the API is limited. While not entirely untrue, the same can be said about most low-level APIs, which provide a base to build higher level abstractions that can make decisions and assumptions appropriate for a specific use case.

Composers who build software are often required to work at both high and very low levels to implement their ideas. Because many are not trained software engineers and are not familiar with object oriented design principles such as the Single responsibility principle,[24] the resulting applications are tightly coupled and often contain areas that work at distant levels of abstraction. This severely limits code reuse, which leads to composers wasting time solving problems that have been solved many times before, as well as fewer composers who are able to implement their ideas in software.

It is my intention that the API and the implementation of ideas presented here can serve as an example of how we can create reusable components for music composition software. Designing for OSC APIs will allow us to write single purpose, language agnostic software that can be shared among composers working in similar areas. The more we as composers embrace the design principles and development practices that have been established in the world of traditional software engineering, the easier it will be for new composers to realize their musical ideas. In an environment where we are already limited by the skills and willingness of performers, it is increasingly important that with technology we are able to extend and build upon each other's work, so that we can continue to push the bounds of new music.

[24]Martin, *Clean code: a handbook of agile software craftsmanship*.

APPENDIX CODE

SAMPLES

osc2websockets.rb

```
#!/usr/bin/env ruby

require □em-websocket□
require □osc-ruby□
require □osc-ruby/em_server□
require □json□

@sockets = []

Thread.abort_on_exception=true

Thread.new do
  EventMachine.run do
    EventMachine::WebSocket.start(:host => □0.0.0.0□, :port => 8081) do |ws|
      ws.onopen { @sockets << ws }
      ws.onclose { puts ”Connection closed” }
      ws.onclose { @sockets.delete ws }
    end
  end
end

Thread.new do
  @server = OSC::EMServer.new(3333)
  @server.add_method □/**□ do |mess|
    message = { □args□ => mess.to_a, □route□ => mess.address}
    @sockets.each {|s| s.send(message.to_json)}
  end

  @server.run
end

sleep
```

plot_pitches.R

```
library(ggplot2)
library(scales)
library(zoo)

# Add since it was removed from scales
to_time <- function(x)    structure(x, class = c("POSIXt", "POSIXct"))

parse.log4r.log <- function(logfile) {
  logstrings <- sapply(readLines(logfile)[-1:-2], function (line) {
     regex <- "I, \\[([^\\s]+) [^\\]]+\\]\\s+INFO -- : (\\w+)"
     str <- gsub(regex, "\\1,\\2", line, perl = TRUE)
  }, USE.NAMES = FALSE)

  logfile.df <- as.data.frame(
    do.call("rbind", strsplit(logstrings, ","))
  ), stringsAsFactors = FALSE)
  names(logfile.df) <- c("log.timestamp", "servo.degree")

  # Remove the "T" from the timestamp to make it easier to parse
  regex <- "(.*\\d{4}-\\d{2}-\\d{2})(T)(\\d{2}:\\d{2}.*)"
  logfile.df$log.timestamp <- gsub(regex, "\\1 \\3", logfile.df$log.timestamp)
  logfile.df$log.timestamp <- as.POSIXct(logfile.df$log.timestamp)
  logfile.df
}

# processes pitch.txt files generated with
# pitchextract -t 1 -v -l 48 -u 52 -p audiofile.wav
marsyas.pitchextract.to.data.frame <- function (pitch.file) {
  pitch.data <- read.csv(pitch.file)
  names(pitch.data) <- c("frequency")
  pitch.data$id <- seq(1, length(pitch.data[[1]]))

  sample.rate <- 44100
  buffer.size <- 1024
  samples <- length(pitch.data[[1]]) * buffer.size
  samples.in.ms <- buffer.size / (sample.rate / 1000)

  max.bin <- samples.in.ms * length(pitch.data[[1]])
  bin.range <- seq(samples.in.ms, max.bin, by=samples.in.ms)
  pitch.data$ms <- as.POSIXct(bin.range/1000, origin="1970-01-01")
  pitch.data
}

plot_pitches <- function(skip.seconds) {
  logfile.df <- parse.log4r.log("logfile.log")

  # Convert the timestamps to start at 0 to match our data below
  logfile.df$log.timestamp <- as.POSIXct(
    as.numeric(logfile.df$log.timestamp - min(logfile.df$log.timestamp)),
    origin="1970-01-01")

  pitch.data <- marsyas.pitchextract.to.data.frame(
      "./pitch_limited_noterange.txt"
  )

  # Need to add 21600 because that□s what as.numeric returns for
  # 1970-01-01 00:00:00 in seconds.
  ms.numeric <- as.numeric(pitch.data$ms)
  pitch.data <- pitch.data[ms.numeric > (21600.02 + skip.seconds),]
  pitch.data$ms <- pitch.data$ms - seconds(skip.seconds)

  # Throw out frequencies below C3 and above E4
  pitch.data <-
    pitch.data[pitch.data$frequency > 130 & pitch.data$frequency < 164,]

  # Join the two data frames by date
  pitch.data.zoo <- zoo(pitch.data$frequency, pitch.data$ms)

  pitch.data.zoo.agg <- aggregate(
    pitch.data.zoo,
    time(pitch.data.zoo) - as.numeric(time(pitch.data.zoo)) %% 0.3,
    mean
  )

  logfile.df.zoo <- zoo(logfile.df, logfile.df$log.timestamp)
  merged <- merge(pitch.data.zoo.agg, logfile.df.zoo)
  # Change column name back from pitch.data.zoo.agg
  names(merged)[1] <- "frequency"
  merged$log.entry <- !is.na(merged$servo.degree)

  merged.df <- data.frame(timestamp=index(merged), coredata(merged))
  merged.df$log.timestamp <- NULL
  merged.df$ms <- NULL
  merged.df$id <- NULL
  merged.df$frequency <- as.numeric(na.locf(merged$frequency))
  merged.df <- merged.df[!is.na(merged.df$servo.degree), ]
  merged.df$run <- sapply(
    as.numeric(as.character(merged.df$servo.degree)),
    function (degree) {
      ifelse(degree > 6800, "down", "up")
    }
  )

  merged.df
}

df <- plot_pitches(6.5)
pitches.up <- df[df$run == "up",]
pitches.down <- df[df$run == "down",]
lm.up <- lm(pitches.up$frequency ~ pitches.up$timestamp)
lm.down <- lm(pitches.down$frequency ~ pitches.down$timestamp)

# Reverse the order of the run factor so facet_wrap outputs in the right order.
df$run <- factor(df$run, levels = c("up", "down"))

ggplot(df[df$frequency > 0, ], aes(timestamp, frequency, colour=frequency)) +
  geom_point() +
  scale_y_continuous(breaks=seq(130, 167)) +
  scale_x_datetime(breaks=date_breaks("20 secs"), labels=date_format("%M:%S")) +
  facet_wrap(~run, scale="free", ncol=1) +
  theme(axis.text.x=element_text(size=12)) +
  theme(axis.text.y=element_text(size=12))
```

api_client.rb

```
#!/usr/bin/env ruby

require 'osc-ruby'
require ”json”
require 'yaml'
require 'tuning_tools'
require 'thread'

class Replay
  attr_accessor :tempo_multiplier
  def initialize (midifile = nil)
    @midifile = midifile
    @last_event = []
    @last_time = []
    @events = {}
    @running_tracks = {}
    @tempo_multiplier = 1
  end

  def load_from_json (file)
    data = File.read file
    @midi_data = JSON.parse data
    curtime = (Time.now.to_f * 1000).to_i
    @midi_data['tracks'].each do |i|
      @last_event.push 0
      @last_time.push curtime
    end
  end

  def register_event(event, &callback)
    @events[event] = callback
  end

  def send_notes(note)
    @events[note['event']].call note
  end

  def check_notes
    curtime = (Time.now.to_f * 1000).to_i
    elapsed = curtime - @start_time

    @running_tracks.each do |track, track_state|
      next unless track_state
      notes = @midi_data['tracks'][track]
      last_event = @last_event[track]

      note = notes[@last_event[track]]
      begin
        note['elapsed'] = elapsed
        if (note['time_from_start'] * @tempo_multiplier) <= elapsed
          @queue << note
          @last_event[track] += 1
        end
      rescue NoMethodError
        if @midi_data['tracks'][track].length <= @last_event[track]
          @running_tracks[track] = false
        end
      end
    end
  end

  def play
    @start_time = Time.now.to_f * 1000
    @queue = Queue.new
    checker = Thread.new do
      @midi_data['tracks'].each_index {|i| @running_tracks[i] = true }
      loop do
        check_notes
        unless @running_tracks.values.include? true
          # Add false to the queue so we can detect it in the other thread and
          # know to stop.
          @queue << false
          break
        end
        sleep 0.001
      end
    end
    sender = Thread.new do
      loop do
        note = @queue.pop
        break if note == false
        send_notes note
      end
    end
    checker.join
    sender.join
  end
end

class TuningApiClient
  include TuningTools

  def initialize (opts)
    @debug = opts[:debug] || false
    @osc_client = OSC::Client.new('localhost', 3333)
    @latest_notes = Queue.new
    @notes_buffer = []

    # Start with a major scale based on pitch class 0.
    @scales = [[0, 2, 4, 5, 7, 9, 11]]

    # Iterate on this scale to generate the rest of the keys.
    10.times {|i| @scales << @scales[0].map {|n| (n + i) % 12} }

    # Start with a tuning base of 0.
    @base = 0
  end

  def publish (channel, mess)
    @osc_client.send(OSC::Message.new(channel, mess.join(” ”)))
  end

  def init_messages
    tuning_file = ARGV[0]
    if not tuning_file.nil? and tuning_file[/ya?ml/]
      tuning = YAML.load_file('./partch_tuning.yml')
      send_tuning tuning[□base□], tuning[□ratios□]
    else
      send_five_limit
      play_midi
    end
  end

  def send_five_limit(base = 0)
    five_limit = {
      0 => [1,1],
      1 => [25,24],
      2 => [9,8],
      3 => [6,5],
      4 => [5,4],
      5 => [4,3],
      6 => [45,32],
      7 => [3,2],
      8 => [8,5],
      9 => [5,3],
      10 => [9,5],
      11 => [15,8]
    }

    scale = Hash[five_limit.map {|k, v| [(k + base) % 12, v] }]

    offsets = {}
    scale.each do |pitch, ratio|
      offsets[pitch] = ratio_to_cents_offset(pitch, ratio, base)
    end

    offsets.each do |pitch_class, cents_offset|
      (1..12).each do |octave|
        pitch_class = pitch_class.to_i
        pitch = pitch_class + (12 * octave)
        publish "/inst/0/note/#{pitch}/tuning/offset/cents", [cents_offset]
        # Sleep for a bit so we don□t flood the queue.
        sleep 0.01
      end
    end
  end

  def send_tuning(base = 0, ratios)
    num_tones = ratios.size

    # Add the base on to the scale
    scale = Hash[ratios.map {|k, v| [(k + base) % num_tones, v] }]

    # Find the key with the value [1,1] to get the new base
    new_base = scale.invert[[1,1]]

    offsets = {}

    scale.each do |pitch, ratio|
      offsets[pitch] = ratio_to_cents_offset(pitch, ratio, new_base, num_tones)
    end

    offsets.map do |pitch_class, cents_offset|
      (0..8).each do |octave|
        pitch_class = pitch_class.to_i
        pitch = pitch_class + (num_tones * octave)

        puts "/inst/0/note/#{pitch}/tuning/offset/cents #{[cents_offset]}"
        publish "/inst/0/note/#{pitch}/tuning/offset/cents", [cents_offset]
        # Sleep for a bit so we don□t flood the queue.
        sleep 0.01
      end
    end
  end

  def play_midi(file = nil)
    file = ARGV[0] || file
    puts "Starting replay"
    replay = Replay.new
    replay.tempo_multiplier = 1
    replay.load_from_json file

    replay.register_event "NoteOn" do |params|
      @latest_notes << params[□note□]
      note_on params[□note□], params[□velocity□]
    end

    replay.register_event "NoteOff" do |params|
      note_off params[□note□], params[□velocity□]
    end

    player = Thread.new { replay.play }
    checker = Thread.new do
      loop do
        detect_key
        sleep 0.001
      end
    end
    player.join
    checker.join
  end

  def note_on(pitch, velocity)
    publish "/inst/0/note/#{pitch}/on", [velocity]
  end

  def note_off(pitch, velocity)
    publish "/inst/0/note/#{pitch}/off", [velocity]
  end

  def detect_key(notes = nil)
    @notes_buffer.unshift @latest_notes.pop
    notes = @notes_buffer[0..15]

    # Convert the notes to pitch classes
    pclasses = notes.map {|n| n % 12}
    results = {}
    @scales.each_with_index do |scale, i|
      results[i] = (pclasses & scale).size
    end

    closest_matches = results.max_by {|k,v| v}
    # Take the first result and ignore keys tied for a match.
```

```
    change_base closest_matches[0]
  end

  def change_base(pitch_class)
    unless @base == pitch_class
      puts "CHANGING BASE #{pitch_class}"
      send_five_limit pitch_class
      @base = pitch_class
    end
  end
end

client = TuningApiClient.new({:debug => true})
client.init_messages
```

REFERENCES

Carlos, W. (2000). Beauty in the beast.

Cianfrocca, F. & Gupta, A. (n.d.). eventmachine (0.12.10). Retrieved from http : / / rubyeventmachine.com/

Dan O'Sullivan, T. & Igoe, T. (2004). *Physical computing: sensing and controlling the physical world with computers*. Course Technology Ptr.

Daugherty, M., Partch, H., Johnson, S. [, & Hyla, L. (1996, June). Howl U.S.A. Nonesuch.

Duckworth, W. (1999). *Talking music: conversations with john cage, philip glass, laurie anderson, and 5 generations of american experimental composers*. Da Capo Press.

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures* (Doctoral dissertation). AAI9980887.

Johnston, B. (1975). The Corporealism of Harry Partch. *Perspectives of New Music*, *13*(2).

MacCaw, A. (n.d.). Spine (1.0.5). Retrieved from http://spinejs.com/

Martin, R. (2008). *Clean code: a handbook of agile software craftsmanship*. Prentice Hall.

Milano, D. (1986). A many-colored jungle of exotic tunings. *Keyboard Magazine, November*.

LEMUR: Purveyors of Fine Musical Robots Since 2000. (n.d.). Retrieved from http:// lemurbots.org/

The audio element — HTML5. (n.d.). Retrieved from http://dev.w3.org/html5/spec/the-audio-element.html

Web Audio API. (n.d.). Retrieved from https://dvcs.w3.org/hg/audio/raw-file/tip/webaudio/specification.html

Partch, H. (1979). *Genesis of a music: an account of a creative work, its roots, and its fulfillments, second edition*. A Da Capo Paperback. Da Capo Press.

Schmidt, D. C. (1995). Reactor – an object behavioral pattern for concurrent event demultiplexing and event handler dispatching.

Singer, E., Larke, K., & Bianciardi, D. (2005, July 15). Lemur guitarbot: midi robotic string instrument. In F. Thibault (Ed.), *Nime* (pp. 188–191). Faculty of Music, McGill University. Retrieved from http://dblp.uni-trier.de/db/conf/nime/nime2003.html#SingerLB03

Tzanetakis, G. (2003). Marsyas (0.2).

Wessel, D. & Wright, M. (2002). Problems and prospects for intimate musical control of computers. *Computer Music Journal*, *26*, 11–22.