# Dies Wiirae

David Andrews, Steven Das, Adam Lederer
Carnegie Mellon University
{dta,scdas,alederer}@andrew.cmu.edu

Group 3, Fall 2009

December 7, 2009

# 1 Introduction

## 1.1 What is this project and why is it necessary, useful, or cool?

Most humans have a natural desire to express themselves through music. It manifests itself in the wide-spread drive to pick up instruments, sing along with the radio, and dance to a beat. There is one very fulfilling and powerful mode of musical expression in particular, though, that very few of us have the opportunity to experience: conducting an orchestra. The conductor has the unique pleasure of directing the full power and flexibility of an ensemble, rather than just a single instrument. Recognizing this, academia and industry over the past several years have capitalized upon the development of practical consumer accelerometer technology to devise several different simulated conducting experiences, ranging from video games for home use[1] to larger-scale touring exhibits at concert halls[2].

Our project represents another take on the "virtual conductor" concept, though with a purpose richer than that of a mere simulator for entertainment purposes. It's an expressive interface - an instrument that allows the player to minutely control the musical thrust of a whole ensemble at once in real-time. Real orchestra players, even when they are giving the conductor their best attention, react to the conductor's gestures only after a certain reflex latency, and make assumptions about the conductor's intention based on their own understanding of the music. Our system is "infinitely attentive", and allows the conductor to truly control the music as his own instrument, rather than just to guide it.

## 1.2 Who can use it?

While it's a goal of ours for even the musically untrained to be able to express themselves with our project, our priority is to afford the highest level of musical expression possible. To accomplish this, we restrict the gestures to certain kinds of expected motions, since that way the subtle variations in the gestures are easier to detect. The most logical way to do this is to expect the gestures to be similar to those of a trained conductor. The lion's share of the expression and tempo data is expressed in similar ways between the untrained and the trained.

## 1.3 A brief overview of conducting and associated music theory

Conducting is a sophisticated art of communication with and leadership of an ensemble. However, we are looking solely at the way the gestures of a conductor are formed and what they tell us about the conductor's intentions. A brief and very general description of these gestures follows:

[1] `http://www.wiimusic.com/`

The conductor waves a baton in a specific series of motions, or *conducting pattern* - certain points along that pattern are time markers that indicate *beats* (regular metric markers, i.e. the thing to which you might tap your foot when listening to a song on the radio) in the music. The part of a conducting pattern which marks a beat is called an *ictus*, and it occurs when the baton is at its lowest point (a local minimum in the vertical direction). Thus, the faster the conductor gestures, the faster these "icti" are reached, and the faster the implied *tempo*, or speed of the music (usually quantified as a number of beats per minute).

The conducting pattern itself varies in shape depending on the number of beats in a *measure*, or regular grouping of beats within a piece of music. Non-musicians will find this concept easiest to grasp with the aid of examples: marches like "The Stars and Stripes Forever" are commonly two beats to a measure, minuets or waltzes like Johann Strauss' "Blue Danube Waltz" (featured famously in the space dock scene and end credits of Stanley Kubrick's *2001: A Space Odyssey*) employ three-beat measure, while everything from "Mary Had a Little Lamb" to Duke Ellington's "Take the A Train" and Elvis Presley's "Jailhouse Rock" makes use of four-beat measures. In Western culture, songs written with four-beat measures are most common. Each iteration of the pattern corresponds to the passing of one measure; therefore, the pattern contains exactly the number of icti found within a given *bar* (the same thing as a measure). Shown in figure 1 are the standard conducting patterns for three and four-beat measures, respectively; the dots represent the locations of the icti. For the sake of simplicity, our system only analyzes and processes conducting patterns modeled after these forms. The bizarre, interpretive dance-like patterns, which some top orchestral conductors employ are not recognized. Usually only orchestras of equal preeminence can follow them without suffering in performance quality, so we feel justified in creating a system which does not tolerate such kinesthetic "creativity".

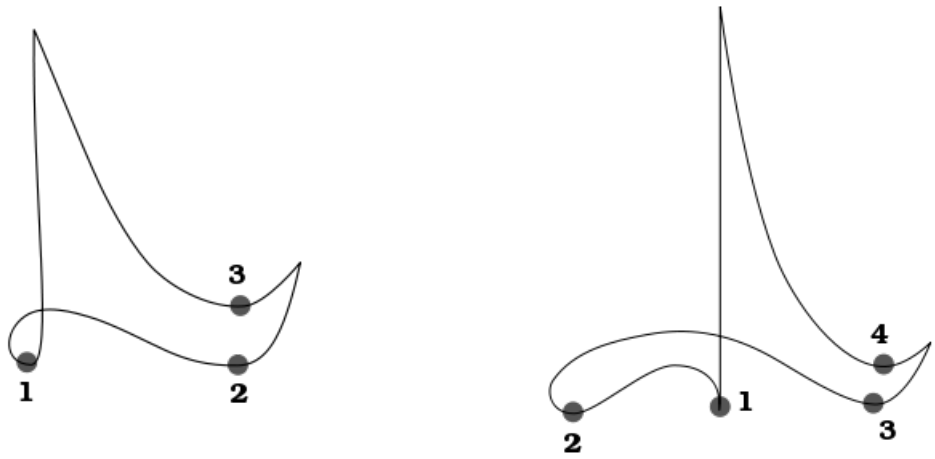


Figure 1: 3-beat and 4-beat conducting patterns

To begin, the conductor holds the baton at the ready in the "1" position; when he desires to start the ensemble, he lifts and lowers the baton in one even motion along the vertical

path indicated over the "1" ictus of each pattern, returning to the "1" position in time for the *downbeat* (first beat of each measure, or of a piece of music at large). Without stopping, the baton continues travelling along the path above toward "2", "3," etc., always reaching the subsequent ictus in time with the next beat of the music. The pattern is fully cyclical: having passed through the last ictus of each pattern, the conductor returns to "1" via the aforementioned "downbeat" motion (albeit without stopping this time), enabling the process to continue for the duration of the entire piece of music.

Each beat may contain several notes. In our system, we chose to divide a beat into four notes of equal duration, called *sixteenth-notes*. Although there are many different ways to divide a beat of music, this choice is representative of much of the music one hears today.

In addition to tempo, a conductor's beat pattern may also convey a wealth of other expressive information to an ensemble, most notably in the areas of *dynamics* (the loudness or softness of the music, also referred to as *volume* or *intensity*) and *articulation* (qualitative measure of how smooth or choppy, connected or separated, light or heavy, etc. music is played). Size is a conductor's best tool in manipulating ensemble dynamics; by keeping the baton in the same tempo but varying the overall height and breadth of the conducting pattern, the conductor may alternately tell his ensemble to play louder (*forte*, in musical terms) or softer (*piano*). Gestural modifications to indicate articulation are as diverse as the types of articulation themselves. An emphatically pronounced ictus indicates an *accent* (a note in which the onset is very pronounced), which makes for a heavier feel to the music. On the other hand, a smooth, even, and flowing conducting pattern signifies the ensemble to play *legato*, an (unsurprisingly) smooth and connected style of playing. An infinite variety of more and subtler styles of articulation than just these exists in the musical world, but we focus mainly on the degree with which a note is accented or legato (the term "articulation" from this point on will only refer to this specific continuum) the intensity of a note.

# 2 The Goal

Our goal was to create a system that would play audio based on conducting gestures. We used Nintendo's Wii Remote[2] to obtain information about the user's conducting gestures. This hardware is elongated in shape, a feature which makes it feel somewhat similar to a traditional conductor's baton, and contains 3 accelerometers which can be used to analyze a conductor's motion. In order to allow the conductor the greatest amount of expressive freedom possible, we decided upon producing synthesized music. We chose to synthesize music rather than time-stretch prerecorded music because (among other reasons) this would give us greater control over expressive qualities of each individual note and the music as a whole. The expressive elements on which we chose to focus were tempo, dynamics, and articulation.

[2] `http://www.nintendo.com/wii/what/controllers#remote`

## 2.1 Music Playback

The point of the music playback subsystem is to generate the tones necessary for the music, to respond optimally to the expressive controls, and to sound decent. The first component is the *sequencer*, which reads notes from the musical score data and interfaces with the synthesizer component to have it produce those notes. The musical score data was all entered by hand as numeric literals by one of our musically talented team members, although it in essence serves the same purpose as the standard MIDI file format.

A notable challenge is that we required our synthesizer to be at least 16-voice polyphonic (that is, be able to produce 16 distinct pitches at a time), since that's a very common amount of complexity to find in certain music, while still operating in real-time.

## 2.2 Tempo

The issue of tempo interpretation manifests itself in two ways. The first is the detection of the points in time at which a user intends a beat to occur. The system should synthesize beats to occur as temporally proximal to the user's intention as possible, reducing latency as much as possible.

The second issue in tempo interpretation occurs because in music, notes don't only fall on the beats; they also fall at certain spaced intervals between beats. Playing these notes at the correct time is trivial if the user never alters his tempo. However, since tempo is measured by the amount of time between beats, a note which occurs between beats is defined by the previous and next beats. In order to synthesize such a note at the correct time, the system must know ahead of time when the user's next beat will fall before it has actually occurred.

## 2.3 Dynamics

A user should have intuitive control over the volume of the synthesized music. Like an actual conductor, we expect the user to express intensity via the total size of the gesture - a very small wave of the baton indicates low intensity, while a very large one indicates that the conductor expects loud notes.

## 2.4 Articulation

We also wish a user to be able to control the articulation of the synthesized music. Since articulation is related to the emphasis with which a conductor approaches a beat, it is very related to the jerk (third derivative of position) of the conducting pattern. The system should produce a more accented note with greater jerk and a more legato note with less jerk.

# 3 The Implementation

## 3.1 Design of the System Architecture

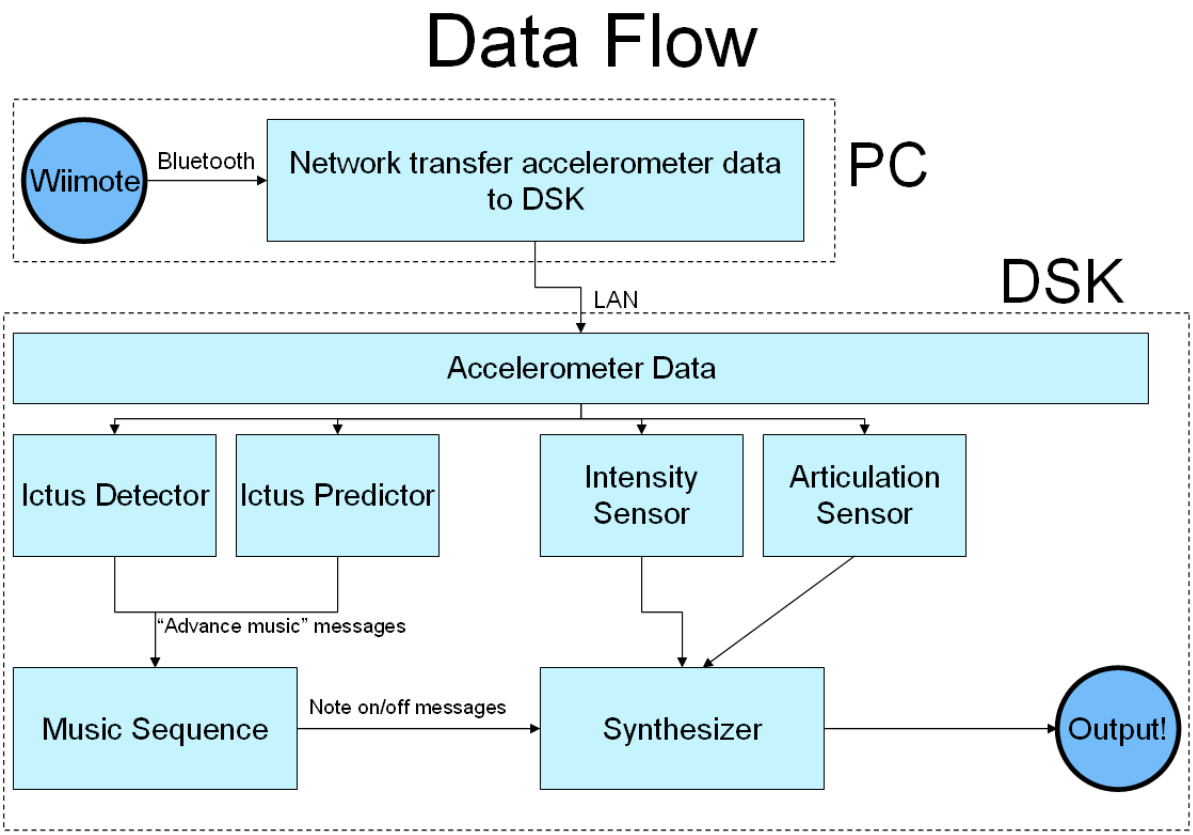


Figure 2: Design of the System Architecture

The majority of the system is implemented on a TI C6711 DSP Starter Kit (referred to as the DSK), which is a 225MHz digital signal processor with 8 logic units. The Wii Remote transmits its accelerometer data via Bluetooth to a PC, whose only role is to then immediately forward that data to the DSK, where the rest of the processing takes place. Once on the DSK, the accelerometer data is examined in real-time by 4 gesture analysis algorithms: ictus detection, ictus prediction, intensity detection, and articulation detection.

In particular, the ictus prediction and ictus detection subsystems control the way the music score sequencer plays back notes. In turn, these note on/off commands control what pitches the synthesizer is playing at a given moment. The synthesizer also responds to intensity and articulation control from those subsystems, which update asynchronously, and itself outputs audio at 32KHz.

## 3.2 Obtaining Data from the Nintendo Wiimote

Since there is no Bluetooth daughter board for the DSK, we made a Bluetooth connection between the Wii Remote and the PC, and forwarded the relevant information to the DSK via the network, without doing any processing on the PC. Software which interprets the Wii Remote's Bluetooth packets was obtained from Brian Peek[3].

The Wii Remote returns three-axis accelerometer information. The z axis refers to the vertical axis - up and down. The y axis refers to the direction in which the user is facing - front and back. The x axis refers to side-to-side motion with respect to the user - left and right. This data is transferred from the Wii Remote sampled at a rate of 100Hz.

The acceleration information returned from the Wii Remote consists of three integers between 0 and 255, one for each axis. In general, these values will be around 127 when the device is in freefall. Consequently, when at rest, the Wii Remote does not produce predictable values since it is accelerating with respect to freefall. Because we wish to consider the device's acceleration to be zero in all directions when at rest, we subtract the cumulative average from the raw accelerator data and use the resulting value for all calculations which use acceleration information.

## 3.3 Ictus Detection

Because an ictus occurs at the lowest point in a conducting pattern, it will occur at the point when velocity is zero. In order to determine when this occurs, we integrate the returned acceleration information and then perform zero-crossing detection.

We integrate by simply accumulating acceleration values. We do not scale the result to standard units because our algorithms only require zero-crossing detection and proportionality of velocity data. From the accumulation we subtract the moving average as a kind of high pass filter to eliminate integration error. After finding velocity, we test it for zero-crossings. When we detect a zero-crossing in the positive direction, we immediately notify the synthesizer that a new beat has occurred.

We chose only to consider the vertical axis for the purpose of detecting icti. We were initially concerned that the smaller vertical motions of beats 3 or 4 in a measure might also require analysis of x-axis motion, but after early analysis, we determined that z-axis information would be sufficient to detect icti. However, we still did calculate velocity for all three axes because we found this information to be useful for our intensity detection algorithm described in section 3.5.

[3] `http://www.wiimotelib.org`

## 3.4 Ictus Prediction

Ictus prediction is useful for calculating the timing of notes which occur between beats. We synthesize four sixteenth-notes of equal duration in each beat, which are spaced between the previous and next beat. Ictus prediction gives us an estimate of the timing of the next beat, which has not yet occurred, and allows us to correctly time these notes.

The most naive algorithm we could use is to simply assume that the time between the most recent ictus and the next will be the same as the time between the most recent ictus and the one before it. This of course completely rules out any notion of reasonably conforming to the user's tempo changes, as all changes will not only be abrupt, but will be one beat delayed. Such a solution is not acceptable by our standards of conformation to user intention.

Our solution was to use acceleration and velocity information obtained between beats to help us update our estimation of when the next beat would occur. In particular, we note four checkpoints within a beat. In order, these are positive velocity zero-crossing, negative acceleration zero-crossing, negative velocity zero-crossing, and positive acceleration zero-crossing. As discussed previously, the positive velocity zero-crossing indicates a beat. The other three checkpoints do not correspond directly to the user's intentions. However, they are consistent across beats in that they occur at similar times within each beat. We use this last insight as a basis for our ictus prediction algorithm.

At the beginning of a beat, we expect that the next beat will be of the same duration as the previous beat. As we reach each checkpoint, we update our estimate of the next beat based on information we have gained at that checkpoint. Our insight tells us that a checkpoint falls the same portion of the way through each beat, and so we can update our estimated beat length as follows:

$$D_{curr} = T_{ck-curr}/T_{ck-prev} * D_{prev} \tag{1}$$

where $D_{curr}$ is the predicted duration of the current beat, $D_{prev}$ is the duration of the previous beat, $T_{ck-prev}$ is the amount of time into the previous beat at which the checkpoint occurred, and $T_{ck-curr}$ is the amount of time into the current beat at which the checkpoint in question has just occurred.

Because the checkpoints occur toward the beginning of each beat, we usually obtain an updated beat prediction before the second sixteenth-note of a beat is played. The implication of this is that the first note in a beat is played with an ictus and no other note is played before an update has been made to the prediction. This causes the output of the synthesizer to follow a user's expectation very closely upon a change in tempo.

However, some interesting (sometimes negative) delays are produced by the ictus prediction. These occur because in practice, the checkpoints do not always lie exactly proportionally within beats. For example, if a checkpoint occurs early within a beat relative to the norm, it may cause an underestimate of the timing of the next beat and therefore the early

triggering of a note. In practice, these delays are small under normal circumstances. They are more pronounced with more eccentric conducting gestures, but to the average user, they are not very noticeable.

## 3.5 Gesture Intensity Detection

Because the dynamic intensity of an ensemble's playing is directly related to the size of the conductor's gestures, the ideal intensity detection algorithm would measure the physical distance between each ictus and calculate the intensity of the resulting note accordingly. However, we decided that we could make a very good approximation to this by relating each note's intensity to the baton's velocity rather than its position, working under the rationale that for a given tempo, a larger (louder) gesture must travel a greater distance to reach the next ictus in the same amount of time as a smaller (softer) gesture.

The algorithm itself is quite simple; the intensity of a given beat is proportional to the the root-mean-square average of the magnitude of the baton's velocity since the previous ictus:

$$Intensity = K_{intens} * |Velocity_{rms}| \tag{2}$$

where $K_{intens}$ is an arbitrary calibration coefficient. The cyclical pattern of the Wiimote conducting pattern in the xz-plane made the magnitude of the velocity, rather than the value of any one of its components, the obvious quantity for divining overall gesture size, and the root-mean-square average velocity proved a better fit for the quasi-sinusoidal velocity patterns between icti than a standard linear moving average. Using root mean squared velocity instead of mean velocity actually saved us computation because to find average magnitude velocity, we must take a square root for each sample, but to find the root mean squared velocity, we must take only one square root per beat.

At this point, the astute reader may observe a conceptual flaw in the above algorithm: certainly if the user maintains a steady tempo while resizing his gesture he ought to hear a smooth decrease or increase in the dynamic intensity of the synthesized music. However, if he speeds up the tempo while preserving the size of his gesture, the algorithm will produce an increased volume. Keeping this edge case in mind, our original algorithm multiplied the RMS average velocity by the predicted time between the last ictus detected and the upcoming ictus to prevent faster tempi, which naturally require faster baton motion at every dynamic level than slower tempi, from producing consistently louder dynamics for the same-size conducting patterns as slower tempi.

This algorithm, as we discovered during testing, indeed succeeded in normalizing intensity results for equally-sized conducting patterns across all tempi, but at the cost of introducing two undesirable side effects into the final aural output. One such problem resulted from the edge case in which an exorbitantly long period of time passed between two icti (i.e. if a

jokester might sit the Wiimote down for a few minutes before picking it up again to proceed to the next beat): in this case, the predicted time until the next beat inflated so dramatically that the predicted intensity of the note also increased without bound, causing the audible output to "explode" with painful loudness and distortion (a result of saturation). The irony here is tremendous: the algorithm originally meant to eliminate the effects of tempo on dynamic intensity actually turned out to worsen the problem at extremely slow tempi!

More significantly, however, we decided that the normalized algorithm didn't respond intuitively to our gestures. In spite of its undesired link between user tempo and dynamic intensity, algorithm (2) reliably provided us with real-time, natural-sounding dynamics and rejected the edge cases afflicting the other algorithm; in light of the immensely more satisfying conducting experience which the three of us felt this delivered, we considered it an acceptable engineering design tradeoff to include some otherwise undesirable tempo dependency in our intensity algorithm in exchange for an aesthetically pleasing intensity calculation method overall.

## 3.6 Gesture Articulation Detection

Jerk, the first derivative of acceleration and third derivative of position, can be one of the most informative metrics of the smoothness or sharpness of a given motion. Differentiating the raw accelerometer data from the Wiimote enabled us to approximate the the baton's position at any point along its path as shown below in Figure 3:

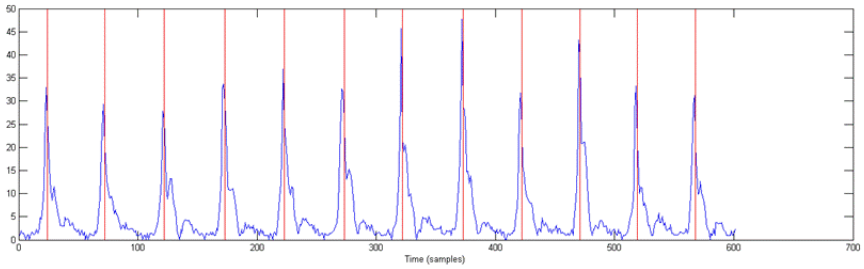


Figure 3: Magnitude Jerk (blue) and Note Onsets (red) Versus Time

Figure 3 illustrates an important observation about the calculated jerk at each point along the Wiimote's path: because the peaks of jerk fell in almost all cases before the ictus itself, we realized that we would be able to use these peaks and their magnitude to determine the strength of the accents on each note.

The final algorithm we devised for linking jerk to the articulation of a given note is as follows:

$$Articulation = K_{artic} * |jerk_{max}| \tag{3}$$

Like the aforementioned algorithm for calculating intensity from velocity, this algorithm too includes an arbitrary constant for experimental "tweaking" and deals with the magnitude of the quantity in question rather than its components to account for the multidirectional nature of the typical conducting pattern. Unlike the previous algorithm, this new scheme looks at the maximum jerk which occurs between icti rather than the average; our philosophy here was that the only significant jerk ought to occur near the icti anyway, so calculating some sort of RMS or moving average would only deaden the responsiveness of the system and add unnecessary computational overhead to the algorithm.

## 3.7 Music Score Sequencer

Our stored music scores consisted primarily of large arrays of notes (represented as bytes which correspond to musical pitches). Two special notes, 0 and 1, were given meanings of "no change" and "stop note" respectively - each of the other possible values refers to the same pitch as referred to by the MIDI note of the same value (60 for middle C). Each index corresponds to a sixteenth-note. Since we allow up to 16 voices to play at once, we allow for 16 voices times 1000 sixteenth-notes worth of note data.

While this dense representation is not nearly the most space-efficient or flexible data structure, this makes playback of the sequence extremely simple: when it's time to play the next quarter of a beat, we simply look at all the notes in the array at the next index, and trigger notes on or off with the synthesizer.

## 3.8 Synthesizer

### 3.8.1 Synthesis Method

We had four main goals for the synthesizer:

1. The basic task that it clearly and accurately convey the musical tones.
2. That it respond to intensity and articulation control in separate and intuitive ways.
3. That it produce a pleasant and satisfying timbre, in order to enhance the experience.
4. That it be at least 16-voice polyphonic (able to produce 16 distinct pitches at a time).

Time-stretching a pre-existing recording to fit the incoming beats (as in Nakra's Virtual Maestro kiosk [2]) was attractive because it would have been the closest simulation to a live performance, but it presented a steep performance obstacle to run it in real-time on our platform (especially with precedent of failure from 18-551 group 2 in Fall 2008 [3]). In order to avoid time-stretching, we investigated solutions where we would continually synthesize the tones in real time.

Using recorded one-note samples of real instruments and resampling them (which is a method employed very successfully in commercial sampling synthesis, albeit in a more complex form) also seemed initially advantageous. The main concern with that option was that, in the attempt to emulate a real instrument, the lack of realism would be distracting to the user. Our ultimate decision was that a neutral synthetic timbre (a sawtooth wave, in particular) would have enough harmonic richness to make the music interesting without risking the above concerns, all while simplifying our implementation.

### 3.8.2 Synthesizer Overview

Our synthesizer is polyphonic, and based on a very simple model: an *oscillator* algorithm generates a sawtooth waveform for each of 16 voices. Each voice has its own *envelope*, which is a time-varying function that scales the voice's volume as the note is held. The output of the synthesizer is simply the sum of all of these voices.

Calculating the output samples one by one as they are required would add a great deal of overhead, be a huge waste of cache and loop pipelining potential, and riskily extend the length of audio output interrupts. As such, part of our scheme for real-time output is to operate the synthesizer in blocks of 64 samples (roughly 2ms). We double-buffer the output: while one output buffer is being read through in the audio output interrupt, the other is being filled by the synthesizer. As soon as all of the buffer being read has been output, we switch, and the synthesizer fills the one that was previously being read from. This necessarily adds a latency of 2ms, since no input can have any effect on the output until the next time the synthesizer fills the buffer. Larger block sizes would provide a greater efficiency boost, but minimizing latency was one of our major priorities.

Since we require that the synthesizer be 16-voice polyphonic, the oscillator algorithm becomes the critical step in meeting that goal, since it must be run for each of those voices without using too many cycles. For performance and for other reasons stated below, we chose to use what's called a *wavetable oscillator*.

### 3.8.3 Wavetable Oscillator

The key to a good oscillator algorithm is avoiding aliasing, since numerically generating a waveform in the discrete-time domain can easily introduce it. A sawtooth waveform can handily be generated (in what we call the *naive* algorithm) by simply outputting a linearly increasing value that overflows from high to low at a certain point. However, this will introduce aliasing, causing the higher frequency notes to evidence a horribly grating noisy inharmonic sound.

The "correct" way to generate a *band-limited* waveform (a waveform that doesn't have any aliasing partials) is by using its Fourier series representation - simply summing over only the frequencies which are below the nyquist. However, this is extremely expensive, because

not only does it involve trigonometric functions to generate the partials, but it also requires you to loop over each sample of the output a non-trivial number of extra times, for each of the partials above the fundamental.

In concept, all that a wavetable oscillator is doing is creating a lookup table for the result of the Fourier series summation for our waveform, then upsampling that lookup table circularly into the output buffer. Because it only upsamples (rather than downsamples), it doesn't introduce aliasing, and for our application, relatively inexpensive linear interpolation produces an acceptably low level of table lookup noise. The upsampling is done by maintaining a phase variable which represents the current integer and fractional "index" into the lookup table, and for each sample to be output, reading the lookup table at the two indices our phase variable is between and linearly interpolating them. As we move on to the next output sample, we add to our phase a fractional value corresponding to our "speed" in reading through the table (for example, if we're outputting a tone at half the frequency of the tone in the lookup table, we would increase the phase by 0.5), then wrap it around the size of the table because the waveform is periodic.

As an initial step upon startup of the system, we generate the wavetables with the Fourier series method noted above. We could generate a wavetable for each note and avoid upsampling, but that would consume an unnecessarily large amount of space, and we would still need to use the same algorithm rather than just directly mem-copying the wavetables since the phase of most notes is fractional, so we generated wavetables for one note in each octave (a practice similar to one often used in computer graphics, known as mip-mapping). The reason we need at least one wavetable per octave is that while upsampling ensures the absence of aliasing, it also means that the wave in the table doesn't have enough high frequency partials to adequately fill the upper spectrum of notes of sufficiently lower frequency than it.

As an optimization to system startup time, we actually use the naive algorithm to generate waveforms for relatively low frequency notes, since aliasing is significantly less evident in lower notes and the Fourier series contains exponentially more numerous non-aliasing partials. The wavetable oscillator was the focus of our main optimization effort, and we were quite successful in not only making it inexpensive enough to meet our polyphony requirements, but to significantly exceed them, which will be detailed further on in this document.

### 3.8.4 Envelope

The envelope function that affects a voice's volume is a three-stage piecewise linear function. It has an "attack" phase, where it is linearly increasing, a "decay" phase, where it decreases from that peak, and a "sustain" phase where it holds its value. Each voice maintains the time since its current note was triggered, and this is used as the input to its envelope function. There are various methods of producing such a function, but our approach was to simply use conditionals to determine the stage and update the envelope value on a sample-by-sample basis, given the non-performance-critical nature of this particular process.

#### 3.8.5 Intensity and Articulation Response

The total volume of a given voice is the sum of the global intensity and the voice's envelope function weighted by the global articulation value. This allows us to have a loud but un-articulated sound when the user is making broad but smooth motions, or even a very quiet but highly articulated sound when the user makes small but jerky motions.

# 4 Speed and Memory Constraints and Optimization

## 4.1 Real-Time Limits

The most intensive subsystem is the synthesizer, since it runs in real-time at audio rate (32KHz). If at any point the synthesizer were not able to fill its output buffer in time before the buffers swap, the result would be audible glitching. In contrast, the gesture analysis algorithms are less intensive in nature and run at the rate of accelerometer input, which is only 100Hz. Within the synthesizer subsystem, the most critical algorithm is the wavetable oscillator.

#### 4.1.1 Wavetable Oscillator

With the DSK's clock cycle of 225MHz, and our audio sampling rate set at 32KHz, our 64 sample block size yields us 2ms, which is a mere 450,000 cycles, in order to synthesize our full 16 voices of audio output. And clearly the wavetable oscillator is not the only algorithm performing any computation during this time - all of the gesture analysis and network transfer processes take amortized portions of this time, although much less significant since only the synthesizer runs at full audio rate. But even assuming all non-wavetable oscillator computations to take 0 cycles, the wavetable oscillator only has $450,000/16 = 28,125$ to compute each voice. Profiling our first pass of the wavetable oscillator yielded that it took around 30,000 cycles, meaning it without doubt required optimization.

What followed was a quite successful process of optimization, by which we reduced the oscillator to a running time of 8,200 cycles, which we tested to meet our 16 voice polyphony requirement. We will only describe the process briefly, because later on we threw away most of these optimizations in favor of a drastically more efficient partially fixed-point implementation that achieved a runtime of 3,600 cycles.

**Floating-point Optimizations** Here is the pseudo-code of our first-pass implementation:

```
for(i = 0; i < buffer_size; ++i )
{
    phase1 = (size_t)phase % table_size;
    phase2 = (size_t)ceilf( phase ) % table_size;
    sample1 = wavetable[ phase1 ];
    sample2 = wavetable[ phase2 ];

    t = phase - phase1;
    output[i] = sample1 * (1 - t) + sample2 * t;
    phase = fmod(phase + delta_phase, table_size);
}
```

The issues mainly involved the use of fmod and several floating point operations that were implemented on the DSK as function calls, since they weren't part of the instruction set - particularly the conversion from floating point to fixed, as well as integer modulus. These calls are expensive, and they disqualify the loop from being pipelined by the compiler. fmod can conveniently be eliminated by utilizing the fact that the phase will never in one timestep increase to more than `table_size` above `table_size`, so we switch that line to

```
phase += delta_phase;
phase = phase>table_size ?  phase - table_size :  phase;
```

With this we save a cool 10,000 cycles, although the conditional still disqualifies the loop from pipelining. Additional savings came from ensuring that our wavetables all have sizes that are powers of 2, which allowed us to eliminate integer modulus in favor of much cheaper bit operations - this gave us another 5,000 cycles. Further optimizations involved eliminating floating point divides by caching inverses, and in general going to whatever lengths necessary to eliminate floating point operations - these miscellanea brought us down to 8,600 cycles, which is a 71% savings. While this allowed us to meet our polyphony goals, we later came up with a fixed-point version of our algorithm that was too good not to implement.

**Fixed-point Optimizations** By using a fixed-point phase variable rather than a floating point, we were able to secure a number of advantages. It obviously eliminated the expensive call for the conversion from floating point to fixed, although it introduced some much less expensive fixed to floating point operations which *are* in the instruction set.

A very nice phenomenon is that because our wavetables all have sizes which are powers of two, setting the fixed point properly allows us to take advantage of overflow to perform the phase modulus for free. Incidentally, our fixed point implementation is strictly greater than or equal to the precision of the floating point implementation. With these advantages, the compiler now reported that it could pipeline the loop by a factor of 2.

We set up the fixed-point phase variable so that a certain number of its most significant bits now exactly comprise the needed index into the lookup table, while the rest of its bits correspond to the fractional "distance" between one index and the next. It involves a few bit-shifts here and there and has made the code less maintainable, but it brought down the running time to a lovely 3,600 cycles. More precisely, 2,300 of those cycles in the wavetable oscillator algorithm are outside of the loop that we optimized, meaning that our optimization brought a 27,700 cycle loop down to 1300 cycles, which is a 95% savings.

**Running Time Estimation** To estimate the running time of our final (mostly-)fixed-point wavetable oscillator algorithm, we'll divide the algorithm into 3 phases, each of which is data-dependant on the last, and so occur strictly in sequence:

The first phase consists of calculating the lookup indices and retrieving the samples from the wavetable. This consists of 3 sequential bit-ops, then 2 on-chip reads that can happen in parallel, for a total of 4 cycles.

```
phase1 = phase >> index_shift;
phase2 = (phase1 + 1) & wavetable_index_mask;
sample1 = wavetable[ phase1 ];
sample2 = wavetable[ phase2 ];
```

The next phase consists of 1 bit-op and then a conversion from int to float (a 4-cycle operation) and a floating multiplication (another 4-cycle operation), all sequentially for a total of 9 cycles.

```
t = (phase & fraction_mask) * shifted_one_inverse;
```

The third and final phase is the linear interpolation, which requires a floating subtraction, 2 floating multiplications (which may or may not be paralellizable, depending on the status of the cross-path which connects the two logic unit sets on the DSK, so we'll assume they can't be), a floating point addition, and then finally another floating multiplication, with the on-chip write paralellizable at some point later on in the loop. This is a total of 20 cycles.

```
output[i] = scale * (sample1*(1 - t) + sample2*t);
```

Altogether, this is 33 cycles - counting these 33 cycles for all 64 output samples, we have 2112 cycles for the whole loop. Since the compiler reported that it was able to pipeline this loop by 2, we will optimistically assume that it was able to keep the logic units twice as busy, for a final estimation of 1061 cycles. This is a bit lower than the 1300 cycles we found the loop actually takes. The discrepancy is probably due to extra operations in setting up the pipelined loop (including the compare/branching and disabling/enabling interrupts) and imperfect pipelining by the compiler which may be creating non-trivial bubbles in the pipeline, particularly since certain floating point operations introduce hard-to-manage logic unit "cooldown times".

## 4.2 Latency

### 4.2.1 Synthesizer

Because each block is synthesized while the previous block is playing back, a triggered note will not be synthesized until playback of the next block begins. Consequently, the new note will begin to sound one block after the block in which it is triggered finishes playing. This causes a delay of 64 to 128 samples (2 to 4ms), depending on where in a block the note is triggered. A delay of this length is barely noticeable.

### 4.2.2 Note Triggering

A new note is triggered immediately upon detection of an ictus, or upon progressing an increment of 25% through the predicted length of a beat. Since accelerometer data is provided at a rate of 100Hz, these events will be delayed between 0 and 10ms, and delays caused by calculations used in ictus detection are far overshadowed by this number.

### 4.2.3 Overall Latency

By examining the individual subsystem latencies, we can determine that the system has a latency between 2 and 14ms for notes that fall on icti. Notes that fall between icti are subject to the same latency (2 to 14ms) from the time which the theoretical ictus prediction algorithm places them.

## 4.3 Memory Usage

We were able to fit all of our code and data into on-chip L2 memory ($256KB$ on the DSK), which allows us to not have to worry about paging any inputs or structuring our code around expensive calls to on-board memory.

The wavetables for the synthesizer are floating point, and we have 6, starting with 256 floats and then dividing by 2 with each subsequent one, so $4 * (256 + 128 + 64 + ...) = 2KB$.

Our sequences consist mainly of $16 \times 1000$ notes, which are just a byte large. We had 3 such sequences, for roughly $3 * 16KB = 48KB$.

Our code size was $67KB$, bringing the total storage requirement to $117KB$, well below the limit for on-chip memory.

# 5 The Demo and Results

On December 7, 2009, we gave a demo of the system to an audience consisting of Prof. David Casasent, his TAs from 18-551, and Mike Kessler (a representative from DRS Signal Solutions) at the Carnegie Mellon signal processing capstone lab. After a demonstration of the various features of the system through a rendition of "American Pie," and a rousing dramatic performance of "Dies Irae," we turned the system over for its true test: each of the audience members were eager to give our instrument a try. What we found was that our project had admirably succeeded in meeting our goals, and was an enjoyable experience for untrained users.

In terms of usability, everyone who tried it needed no more than a few seconds to become comfortable with the motions and their results. Even more rewardingly, everyone in the lab (which was crowded with students frantically putting the finishing touches on their capstone projects) wanted to try it. We saw users intuitively exercising every aspect of the system, naturally engaging the intensity and articulation controls in their physical portrayal of the music. The ictus prediction algorithm provided smooth tempo changes despite the new and sometimes unexpected gestures the users employed.

With the low latency architecture and robust detection algorithms we implemented, we achieved a responsive instrument that even non-musical people unfamiliar with the system were able to quickly adapt to and explore as a playground of musical expression.

# 6 Future Work

## 6.1 Gesture Analysis

Although the ictus prediction algorithm worked well enough to synthesize four notes per beat, attempting to synthesize eight notes per beat (which requires the second note to be synthesized earlier) may not work as smoothly. This problem will occur when the user adjusts tempo and the second note in the beat is synthesized before the first checkpoint is reached and the ictus prediction is updated. A more sophisticated version of the ictus prediction algorithm could be produced to consider the entire curves of the acceleration and velocity curves, rather than just their zero-crossings and minima/maxima.

On the subject of expression detection, the most obvious improvement which future work could yield would be to develop an algorithm capable of distinguishing between smooth legato playing and *staccato* playing, or an execution which is light and separated (though not necessarily as forceful as accented playing). Exactly what metric would be used to implement such an algorithm is still under consideration: jerk, also used to place accents, remains a likely candidate, although the new algorithm would need to display increased sensitivity to smaller gestures if it were to capture motions as subtle as that of a staccato

conducting pattern at a soft dynamic level. An improved sequencer to accompany this new algorithm would be able to adjust the length of the notes it outputs to respond to the user's articulation style accordingly.

### 6.2 Synthesizer

The wavetable oscillator we've established could easily be extended to support playback of longer recorded samples rather than single-cycle synthetic waveforms. The framework is here for extending the synthesizer to incorporate the advanced multi-sampling techniques that can produce realistic instrument sounds, potentially making the experience more compelling.

In the other direction, many opportunities present themselves for going further with the synthesizer using synthetic timbres. Even something as simple as implementing *vibrato* (a technique of varying note pitch with time, which human performers use to great effect) could bring more life to sensitive legato passages. Adding a filter to make the synthesizer subtractive, or adding frequency modulation, or even just adding a second oscillator to each voice would exponentially increase the possibilities for timbre creation. With just a sawtooth wave, every voice sounds the same; with the possibility of distinct timbres for each voice, the sky's the limit in terms of arrangements and ensembles.

The big win, of course, would be to bring the core gesture analysis system off of the DSK and onto the desktop PC, where it could interface with the incredibly rich ecosystem of synthesizers that already exists on that platform.

## 7 Hardware Purchases

**Wiimote** The Wiimote is an input device which contains accelerometers. We use this hardware as the interface to our project, which will send data (as acceleration information) provided by users to the DSK to process.

**Bluetooth Receiver** The Wiimote communicates with a computer via a Bluetooth connection. In order to use a Wiimote in our project the computer we use must have Bluetooth capability.

**Wii Music** Wii Music is a commercial product which performs a function very similar to that of our project. We bought it to test its responsiveness, and to see whether it achieves our goals and how we can improve on it.

**Batteries** The Wiimote runs on batteries. Batteries are necessary for the development and demonstration of our project.

| Item | Cost | Vendor |
|---|---|---|
| Wiimote | $34.96 | Amazon.com |
| Bluetooth Receiver | $17.95 | Amazon.com |
| Wii Music | $19.99 | Amazon.com |
| Batteries | $13.62 | Amazon.com |
| Total Cost | $86.52 | |

Table 1: Cost Analysis

# 8 Novelty and Related Work

The Fall 2008 18-551 Group 5 used Wii Remote technology in their project, Wii Want to Write[1]. Their project involved gesture recognition to identify letters "written" in the air by the user. While this group used similar hardware in their project, the main goals of their project are very different from ours.

Another 18-551 group, Fall 2008 Group 2, created a project called Ultimate Dance Party[3]. Their project was centered around the idea of enabling songs played at parties to flow more naturally together. The project used beat synchronization and tempo modification techniques to modify and combine the music. Our project is different from theirs in that ours synthesizes its output music whereas their output is a modification of prerecorded music. As such, our project focuses more on allowing expression of the user and less on time-warping an existing signal.

A team at The College of New Jersey led by Prof. Teresa Nakra[2] has created an exhibit which performs a function very similar to that of Ultimate Dance Party. This exhibit synchronizes the audio and video of a recorded orchestral performance with gestures supplied by a user with a Wiimote. This again uses prerecorded music whereas we will be using synthesized music, and does not allow for interpretation of articulation from the conductor. For more information on this project, see `http://www.youtube.com/watch?v=rX-bQR4bkqY`.

There is also a commercial product on the market which is somewhat similar to our project. It is a video game, available for the Nintendo Wii, entitled Wii Music. One of its modes involves allowing the user to conduct a virtual orchestra. However, this product only gives the user control over tempo, and it doesn't map icti precisely to beats (in general it will introduce a certain amount of tempo smoothing if the user changes tempo rapidly). This lack of precision makes it unviable as a musical instrument. Our project is focused on more consistent, predictable, and precise tempo specification, as well as expressive control of dynamics and articulation. For an example of a video of someone conducting on Wii Music, see `http://www.youtube.com/watch?v=lKggSTSPUHg`.

# 9 Schedule

The following chart displays our group and individual accomplishments for the entire semester:

| Week | Adam | Dave | Steven |
|---|---|---|---|
| 10/19/09 - 10/25/09 | | | • Ordered hardware |
| 10/26/09 - 11/1/09 | • Established real-time synthesizer architecture with naive oscillator | • Send data to DSK | • Interface Wii with PC |
| 11/2/09 - 11/8/09 | • Implemented music score sequencer<br>• Sequenced "Wizards and Warriors" theme | • Prototyped ictus detection in MATLAB and C# | • Prototyped articulation detection in MATLAB and C# |
| 11/9/09 - 11/15/09 | • Implemented band-limited wavetable oscillator | • Prototyped ictus prediction in MATLAB and C#<br>• Established DSK-side real-time gesture analysis architecture | • Prototyped intensity detection in MATLAB and C# |
| 11/16/09 - 11/22/09 | • Optimized wavetable oscillator<br>• Sequenced "Dies Irae" | • Implemented numerical integration<br>• Implemented DSK-side ictus detection | • Refined intensity detection prototype<br>• Refined articulation detection prototype |
| 11/23/09 - 11/29/09 | • Optimized wavetable oscillator (fixed-point)<br>• Sequenced "American Pie"<br>• User testing and algorithm tuning | • Implemented DSK-side ictus prediction<br>• User testing and algorithm tuning | • Implemented DSK-side intensity detection<br>• Implemented DSK-side articulation detection<br>• User testing and algorithm tuning |
| 11/30/09 - 12/6/09 | • Final report | • Final report | • Final report |

# References

[1] Jeffrey Lai, Tian Seng Leong, and Jeffrey Panza. Wii want to write. Previous 18-551 Project, Group 5, Fall 2008.

[2] Teresa Marrin Nakra, Yuri Ivanov, Paris Smaragdis, and Chris Ault. The *UBS Virtual Maestro*: an interactive conducting system. *NIME 2009 International Conference of New Interfaces for Musical Expression*, June 2009.

[3] Yang Zhou, Tina Milo, and David Tuzman. Ultimate dance party. Previous 18-551 Project, Group 2, Fall 2008.