# iDiMP: iPhone Distributed Music Performance

# <http://code.google.com/p/idimp/>

# Michelle Daniels and John Kooker

# CSE237B: Fall 2008

# Background/Motivation

Computer music is a broad field incorporating the diverse disciplines of music, computer science, and electrical engineering. In its early days, computer music primarily involved non-real-time audio synthesis and composition. A composition a few minutes long could take hours or even days to be rendered by a computer, and real-time interaction between musicians and computers was, for the most part, impossible. Today, with ever more powerful computers, computer musicians have become increasingly focused on real-time interaction between performers and technology. Such real-time interaction can be somewhat simplistically divided into two categories: real-time control of audio synthesis parameters and real-time signal processing of live recorded audio. In reality, these two categories are often connected, with real-time computer interaction modifying parameters that in turn control the real-time processing contributing to a live performance.

Because the traditional computer keyboard and mouse are so limited in their creative and expressive abilities in comparison to traditional musical instruments, a significant amount of research has been done in the area of human-computer interaction for musical applications [1]. It is not unusual these days to see musicians incorporating custom-built controllers employing technology such as motion sensing, haptic feedback, etc. alongside more familiar musical interfaces like piano-style keyboards. At the same time, increasingly inexpensive devices such as Wii video game remotes with built-in accelerometers are making experimentation with these kinds of controllers even more accessible to people who may not have the necessary hardware skills to build a custom device from scratch.

The iPhone[[1]](#footnote-0) is especially exciting from this perspective, since it provides at least two potentially very expressive forms of interaction: the touch screen with multi-touch capability and the accelerometer. Computer musicians caught on to this quite quickly and at this time freely available software exists [2] which will communicate control data such as position/orientation and touch locations from an iPhone to host computers that can then use the acquired data as parameters for musical processes.

However, just making another accelerometer-based controller is not all that exciting and new – a host computer is still required to do the actual synthesis and signal processing, so for this project we set out to see if we could instead use the iPhone itself as the synthesis and processing host machine, thus giving us a completely mobile computer music system. Because the iPhone is designed for multimedia applications, we were hopeful that it would have the processing power, audio quality, and SDK functionality that such an app would require.

At the same time, being aware of the iPhone’s inherently networked nature, we decided to attempt to incorporate another interesting area of research in computer music. This research involves networked musical performances where musicians in geographically separate locations are able to collaborate and play together by streaming each site’s audio over networks between the different locations [3]. The possibility of real-time interactive iPhone “jam sessions” between two people at different locations was an interesting challenge, so we were curious to see if we could not only use iPhones for audio synthesis and processing but also for simultaneous networked audio streaming.

# Goals

Our project, iDiMP (iPhone Distributed Music Performance), attempts to address the ideals described above of both mobile and distributed computer music. At the start of this project, we had no idea how much of our proposed iDiMP system would be possible or practical using the iPhone, so we divided the project into several main goals:

* Synthesize and play audio on the iPhone in real time.
* Record uncompressed PCM audio from the iPhone microphone and play it back in real time mixed with the synthesized audio.
* Stream the uncompressed synthesized and recorded audio over the network to a paired iPhone and simultaneously stream uncompressed audio from that paired iPhone to be mixed and played back in real time with the locally synthesized and recorded audio.
* Optionally apply signal processing effects to each audio source (synthesized, recorded, and input from the network).

The biggest unknowns included:

* How much synthesis and processing could the iPhone CPU handle in real time?
* What would be the minimum achievable latency for playback of recorded or network-streamed audio, and would it be acceptable for musical purposes?
* Would it be possible to stream uncompressed audio over Wi-Fi in real time without significant packet loss?
* How long would the battery last while running such a CPU-intensive application?
* How would we control all of the desired musical parameters with the iPhone’s limited screen size?

# Method

Broadly speaking, our design for iDiMP divides the functionality into three distinct areas. One is the underlying audio functionality, including recording, playback, and synthesis of audio. The second is networking, including the discovery and connection of two devices and the bi-directional streaming of audio data between them. The third is the user interface, which includes control of musical parameters along with general audio and networking configuration. Because the audio and networking functionality could be implemented largely independently of each other, they provided a clean point for division of labor. Michelle implemented the audio side and John implemented the networking side, while both of us contributed to the UI.

#### Architecture and Design Principles

##### Audio

The Core Audio Framework [4] is the C interface to Mac OS X’s fairly powerful audio libraries. In addition to providing clean ways for applications to use audio resources without interfering with the iPhone’s need to use the microphone and audio playback for phone calls, the iPhone OS adopts a subset of the OS X Core Audio Framework to provide applications basic audio I/O and processing within the limits of the iPhone’s hardware.

### Requirements and Design

The most well-documented ways to play audio on the iPhone involve playback of audio files from the file system and recording of audio from the microphone to the file system. Apple’s SpeakHere sample iPhone application [5] encapsulates exactly that functionality. However, iDiMP has advanced audio requirements for which this basic I/O support proved insufficient, including the following. Audio for iDiMP must meet the following requirements:

* Support synthesis of audio for playback in real-time.
* Support recording from a microphone while simultaneously playing back the recorded audio through the iPhone headphones or speaker. This requires very low latency between recording and playback because the user hears both the sound when it is initially recorded (directly from whatever made the sound) and when it is played back (through headphones or speakers). Significant delay between these two events is quite noticeable and annoying.
* Support real-time mixing of synthesized and recorded audio along with streaming audio received via the network.
* Support real-time signal processing of synthesized, recorded, networked, and/or mixed audio.

In addition to providing low-latency audio recording and playback, AudioEngine is also designed to support a flexible plug-in style architecture allowing any audio effects (represented by the AudioEffect class) to be applied at almost any stage of processing. This includes allowing AudioEffects that apply only to a specific input (such as recorded audio) or to the master mixed output. While the user interface currently supports only a hard-coded set of AudioEffects and associated parameters, AudioEngine itself is blind to the nature of the specific AudioEffects it applies. This architecture enables the development and deployment of new AudioEffects without any changes to AudioEngine itself.

Similarly, the AudioEffect class is designed with an interface that provides a standard way of describing AudioEffects and their parameters. A user interface could take advantage of this design to provide dynamically generated controls for whatever AudioEffects happened to be enabled at a given time. This design decision was motivated by the flexible design of Apple’s Audio Unit plug-ins for OS X, where plug-in developers specify what parameters they wish to make available to users along with some information about the type of parameter, and the user interface for the plug-in is rendered at run-time by the Audio Unit host application.

### Implementation

To meet these requirements, the audio for iDiMP is driven by a single, finely-tuned engine (the AudioEngine class) controlling both recording and playback of audio. To interface with the iPhone’s audio hardware, AudioEngine uses the RemoteIO Audio Unit from Core Audio for the iPhone. The Audio Queue Services available for iPhone are much more commonly used and thoroughly documented than RemoteIO, so they were our first choice of interfaces. However, while AudioQueue provides acceptable latency for playback of synthesized audio, it is completely inadequate for playback of recorded audio. AudioQueue works with a system of registering callbacks for handling recorded audio input and audio output for playback, and for some unknown reason, while the AudioQueue playback callback occurs every 1024 samples on the iPhone, the recording callback occurred only every 16384 samples. This much latency on the recording side (0.37 seconds at the standard sampling rate of 44.1kHz) is completely unacceptable for any kind of real-time audio application.

Fortunately, using RemoteIO resulted in playback and recording callbacks being called back-to-back at the same rates (every 1024 samples), enabling much lower and more acceptable latency (0.023 seconds at a 44.1kHz sampling rate). On the other hand, RemoteIO was almost completely undocumented, and without help from the limited internet resources available [6], figuring out how to use it would have been near impossible.

In addition to basic audio I/O, AudioEngine is also responsible for driving all synthesis and processing of audio for iDiMP. Each time the AudioEngine playback callback is called, AudioEngine must perform the following tasks to fill the output buffer of samples it was given:

1. Trigger synthesis of one block of audio, including applying any synthesis-specific effects.
2. Copy samples of recorded audio from the buffer where they were stored after the recording callback was called and apply any recording-specific effects.
3. Obtain samples of streamed audio input from the network, if available, and apply any network-specific effects
4. Mix these three audio inputs to obtain an output mix and apply any master effects. The processed output mix will be sent to the DAC.
5. Mix synthesized and recorded output only and apply any master effects. The processed result will be sent to the network for streaming output to any paired device.

The resulting flow of data is shown below in Figure 1.

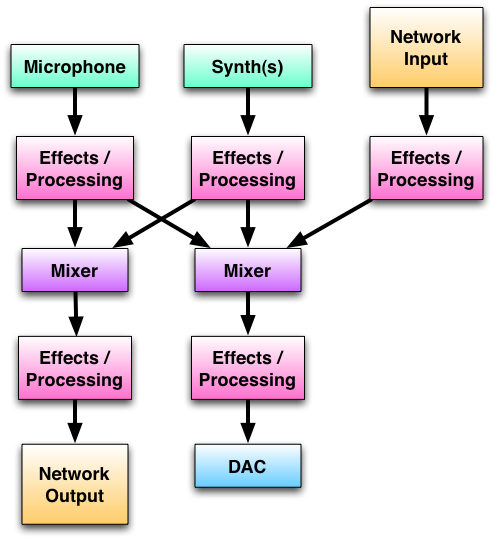


Figure . The flow of audio data through AudioEngine

To simplify computations, all audio processing and synthesis is done with floating point values in the range [-1.0, 1.0]. Because recorded input, network I/O, and playback all require 16-bit signed shorts, this required conversions from signed shorts to floats and back to signed shorts. This conversion seemed worth the effort given the convenience of avoiding fixed-point arithmetic and associated numerical precision issues. However, if it is deemed to be too computationally expensive, the floating point conversions could be avoided and all signal processing could be done as fixed point instead.

### Optimizations

AudioEngine is optimized to reduce CPU usage (and therefore battery consumption) whenever practical. Given the fact that the Core Audio library interfaces are in C, the decision was made early on to write all audio-related code in C++, which seems to offer less overhead for this kind of computing compared to using Objective-C while still allowing object-oriented design. Aside from this fundamental design choice along with common-sense practices such as reusing allocated memory rather than repeatedly allocating new large blocks, the most important optimization in AudioEngine is the ability to mute any combination of audio inputs and output. As long as an input is not muted, any effects registered for that input will be applied at each iteration of the playback callback. No processing or effects are applied to muted audio, resulting in savings in CPU usage.

Similarly, even when audio is not muted, certain kinds of effects can be optimized to minimize necessary computations under the most common use cases. For example, amplitude scaling is applied at various stages of audio processing to provide flexible balances between various inputs and to control the global audio output level. When an amplitude scaling factor is set to 1.0 (the default), multiplication by the scaling factor simply returns the same output audio that was input to the effect, so many multiplication operations can be avoided. When the scaling factor is 0.0, the resulting audio will always be zero, so again expensive multiplications can be avoided.

Unfortunately, some aspects of the AudioEngine could not be optimized in any practical way. For example, even when no audio was being synthesized and no audio was being recorded by the microphone, unless the user explicitly muted all audio, the underlying system playback and recording callbacks were still called periodically. That means that if a user were to leave the app running without actively using it, CPU usage would still be continuing in the background while only silence was output from the DAC, wasting valuable battery life. We could not think of any ways to eliminate this CPU usage without significantly limiting the way the user interacted with the application, but we continue to look for some kind of solution to this dilemma.

##### Networking

### Design and Implementation

The greatest challenge in the networking aspect of our application was finding a way to symmetrically send and receive audio streams. We decided early on that this project would not merely send control parameters, but would push the limit of networked audio streaming.

We elected against sending compressed streams for two reasons. The first was a matter of taste: we preferred the high quality of uncompressed audio for a musical application. The second was more concrete: any kind of audio compression would introduce latency due to encoding and decoding algorithmic delay, assuming that the iPhone could even handle encoding audio in a format with suitably high quality for musical needs in real-time. In order to minimize the latency of the entire network system, we were therefore left with the problem of finding a way to transmit and receive uncompressed audio in the most reliable way possible.

The iPhone OS supports BSD sockets, as Darwin is based on BSD UNIX. Given the real-time constraints of our networked audio transmission and playback, we chose UDP over TCP. TCP guarantees that all packets will arrive and that they will arrive in order, but the handshaking and retransmission used to guarantee this result would cause greater latency than we were willing to accept (for real-time audio playback it is preferable to drop the occasional packet every now and then rather than have large gaps of silence delaying the entire playback process while waiting for a lost packet to be resent). Also, because TCP is designed to be a “good neighbor” on a network, it does not take full advantage of available bandwidth. UDP’s focus on sending a constant stream of data without checking for each packet’s receipt, therefore, fit our needs best.

However, this choice had an associated cost. With UDP’s focus on streaming data transmission, there is a chance for packet loss and disordering between source and destination. Given our goal of glitch-free networked audio, we needed to protect against packet loss. In testing the iPhone’s UDP support, we found that lost packets were commonplace even in ideal, quiet network environments. We also found that packet disordering was rare, but possible. These experiments informed our network controller design decisions.

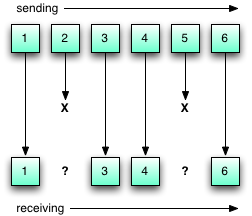


Figure 2. UDP packet loss leading to gaps in received data

Figure 2 illustrates the problem we discovered with UDP’s propensity for packet loss. Since our tests revealed that most dropped packets were isolated, we designed a simple redundancy scheme, illustrated in Figure 3.

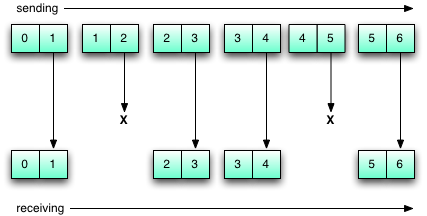


Figure 3. UDP packet loss overcome by data redundancy

Each device sends out audio buffers in pairs, sending each one twice. That way, the neighboring packets can be used to fill in any gaps from lost packets.

On the receive side, these buffers needed to be cached in a manner consistent with our low-latency goals. We designed a buffering scheme similar to circular buffering, but also employing a simple hash to keep data in relative order. Figure 4 illustrates the scheme.

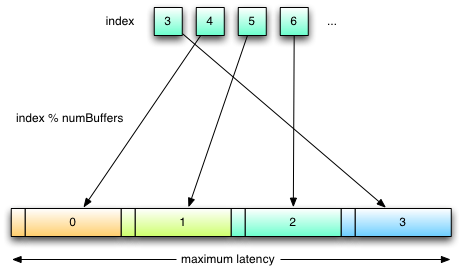


Figure 4. A 4-buffer cache being filled by incoming network data

The AudioEngine calls into the NetworkController to fill its buffer for playback. As this happens, the NetworkController iterates through the cache, copying data out for playback. The result is that the buffers always have the newest data, and the maximum latency is determined by the size of the cache. The current NetworkController implementation uses an 8-buffer cache, which computes to approximately 0.19 seconds of latency. This setting can be tuned according to the user’s preferences. Also noteworthy is the system’s combination of memory frugality and resistance to dropouts. The cache is statically allocated, and extraneous memory copying operations are avoided by including one byte of metadata about each cache buffer’s contents. Should clock or sample drift occur, the nature of the buffering allows for recovery within our design tolerances.

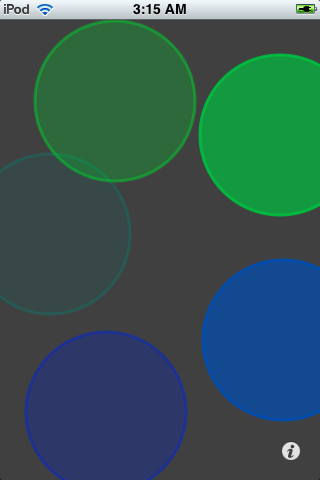
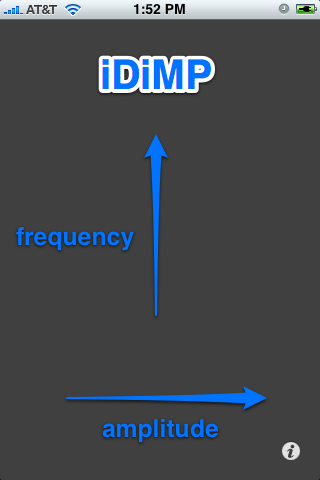
Another important networking feature was the inclusion of Bonjour, Apple’s Zeroconf networking feature. Using Bonjour, iDiMP is able to advertise itself on a local network as an iDiMP UDP peer, and simultaneously browse the local network for other iDiMP devices. As a result, iPhones running iDiMP can discover each other and automatically obtain each other’s IP address and port number. This saves the user the headache of configuring and debugging network settings on their own.

##### User Interface

### Design

When working on a platform like the iPhone, a User Interface designer has a very high bar of usability to meet. It is expected that any user can download and install an application and instantly recognize how to use it. It was with this in mind that we set out designing the main iDiMP screen.

Figure 5 shows the Default.png file, which is displayed on application launch. Apple’s own applications use this image to give the impression of speedy launch, putting an empty screenshot on the screen until the real user interface is ready for interaction. This design would not do for our application; the user must know exactly when iDiMP is ready to use. So, we used the placeholder for a different purpose: it provides the only usage instructions in the entire application.



Figures 5 and 6. iDiMP Default.png, and MainView with multitouch

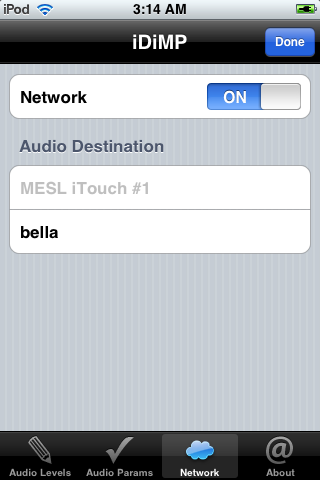
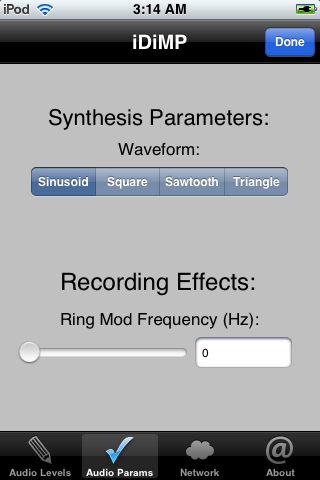
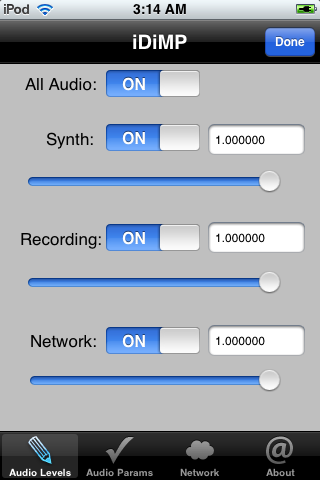
This shows the user what they could find out by just running their finger across the screen: the coordinates of each touch are converted into frequency and amplitude controls for individual synthesizer voices. The color of the touch highlights changes (appropriately enough) with frequency, while the opacity reflects the amplitude value.

iDiMP’s synthesis engine supports more than one waveform: it can produce sinusoids, square waves, sawtooth waves, and triangle waves. The touch highlights change shape to match the currently selected waveform (Figure 7), and the active waveform is toggled with a shake gesture. Additionally, we chose to tie the main accelerometer angle to a master volume control, granting players access to this control in real time.



Figure 7. Square waves indicated by square touches

Much like the Weather and Stocks iPhone applications, we placed our configuration screens on the “flip side” of the main view. This allowed us to expose far more control parameters than the live player’s UI could support, with the tradeoff that these controls are intended to be more-or-less constant. Figures 8 and 9 show the two tabs devoted to controlling these extra parameters.



Figures 8-10. Configuration tab views

Figure 10 shows the Network tab, which is designed to remind the user of the WiFi Networks settings page in Apple’s Settings application. To send or receive networked audio data, the user only needs to switch on the Network slider, and then select an available iDiMP device (discovered via Bonjour, as described above). This procedure should be familiar to users, so we are intentionally leveraging that.

Altogether, the iDiMP user interface is designed to be a balance: intuitive and powerful. We think we achieved this well, providing the most commonly used controls on the main screen, and adding important functionality in the configuration tabs.

### Implementation

iPhone development in Objective-C encourages the use of the Model-View-Controller data pattern. We followed this guideline in our user interface design, providing controllers for each of our views. It is the controllers that provide the interface between the AudioEngine and NetworkController models, and the Main and Flipside (or, configuration) views. Choosing which controller ought to handle which action proved to be a tricky task: We needed to handle accelerometer data for detecting shake gestures as well as providing constant master amplitude control data, and had to decide which active views should allow which parts of that functionality. Though sometimes the pattern was contrary to our regular habits, it helped us think through our design decisions. [11]

#### Workflow

To organize this work, we created an iDiMP project on Google Code, a SourceForge-like central location for open source coding projects. Google Code provided us with several valuable development tools. The first was a Subversion repository we could access anywhere and anytime to track and share code revisions. Apple’s Xcode development environment contains Subversion support and integration by default, so this was a very convenient and necessary addition to our tool chain.

The second was an issue-tracking system we could use to record information about bugs we found and features we wanted to implement. While we didn’t use this issue-tracking system until near the end of our project, we found it to quite useful. However, a system with more features, such as the abilities to give time estimates to tasks and assign tasks to particular developers would have been even more valuable.

The third resource Google Code provided us with was a wiki we could use to share resources and document complicated procedures we might later need to repeat. We used it early in the project to share knowledge with our classmates, as getting started in iPhone development proved to be tricky.

Lastly, Google Code provided our project with a permanent home where anyone who wants to continue our work can have full access to our source code and development process. For interested parties, the URL for iDiMP on Google Code is <http://code.google.com/p/idimp/>.

With these supporting tools in place, all implementation was then done using Apple’s Xcode development environment (version 3.1.2) and iPhone SDK (iPhone OS versions 2.1 and 2.2), including the use of Interface Builder for UI design and Instruments for application profiling.

# Results

The original goals of iDiMP were to work within the constraints of the iPhone platform in order to implement real-time synthesis, recording, processing, playback, and bi-directional networking streaming of audio with minimal latency while providing an expressive user interface for creative control of audio parameters. All of these goals were accomplished to varying extents.

Basic audio synthesis was a fairly straightforward feature to implement. Using wavetable synthesis minimized the computational load, and synthesizing enough “voices” of audio to simultaneously represent multiple touches on the touch screen was not a problem.

Once the RemoteIO Audio Unit interface was well understood, real-time audio recording with simultaneous playback similarly became a very practical task, also well within the computational limits of the iPhone. However, the iPhone’s CPU limits were pushed when real-time signal processing was introduced at several points in the audio playback chain. Because we had somewhat ambitiously designed iDiMP to be as flexible as possible, allowing as much control over processing each audio input as possible, this was somewhat expected, and fortunately some careful optimizations reduced the computational load required in the default operating mode to the point where synthesis, recording, and playback could all happen glitch-free the vast majority of the time.

The addition of networking, however, took us above and beyond the iPhone’s available CPU cycles, and also proved challenging in general WiFi networks. There was more packet loss than we had anticipated in non-ideal networks, forcing us to modify our approach to allow sending of redundant data as a preventative measure. Also, the processing power required to stream audio data over the network while simultaneously receiving audio data from the network used more CPU cycles than were available while the audio functionality described above was also happening. This resulted in audible glitches in the networked audio signal as well as a loss of UI responsiveness, but we see this as an opportunity for future optimization. In ideal conditions, we have seen these networking issues almost disappear.

Though we did not have the time to run a comprehensive power test suite on the iPhone running our application, we were able to get some valuable CPU usage information from Apple’s Instruments tool. In the idle state, just after starting the application, iDiMP runs at approximately 25% CPU utilization. This consists mostly of audio recording and playback, without synthesis. If we add to this a maximum number of synth voices, as seen in Figure 11, the CPU usage reaches its maximum. Thus, we seem to be achieving our goal of pushing the limits of iPhone processing power, while being a good citizen when not in active use.

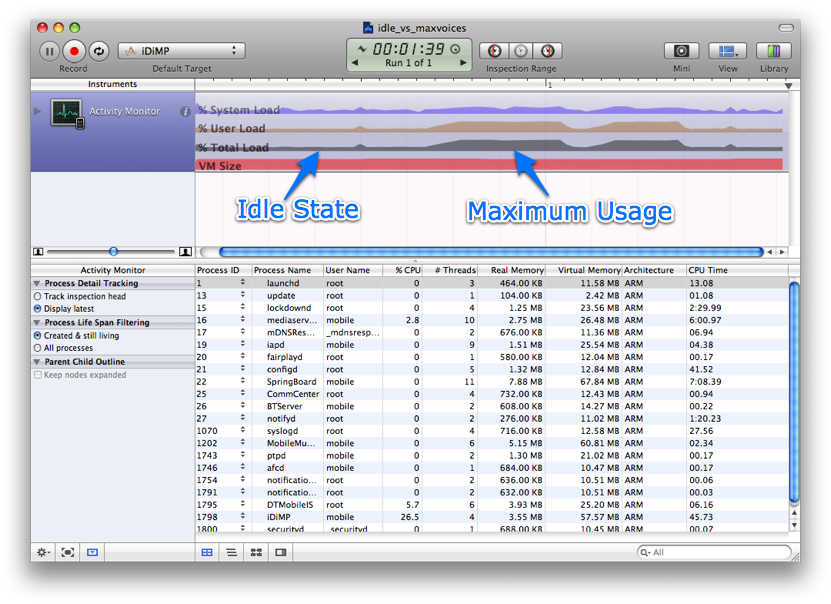


Figure 11. Instruments data for idle and maximum usage states.

# Problems Encountered

The development process using Apple’s iPhone SDK was far from smooth. Between the mysterious quirks of Apple’s Xcode IDE and the lack of comprehensive documentation for much of the iPhone SDK, we spent a large number of hours simply learning the subtleties of iPhone development, beginning from the most basic task of installing a user-built application on the iPhone.

In order to protect the iPhone platform, Apple has implemented rigorous security measures around the signing of executables. This results in some complicated and obtuse setup procedures that must be completed before an iPhone application can be tested on a real device. All testing devices must be registered on the Apple Developer Connection website, which generates “provisioning profiles” to be installed on every development machine, and every target device. Then there are settings that must be correctly typed in several places for each software project, none of which are clearly documented in any central location.

As could be expected with such a new SDK, there were subtle issues that cropped up at inconvenient times, like discovering that the iPhone Simulator (a generally wonderful tool) provides a different Device Name than the networking frameworks do. The simulator also lacks accelerometer support and real multi-touch capabilities, making it of somewhat limited use for thoroughly testing our application, which relied heavily on all three of these features. Fortunately, unlike many mobile phone development environments, Apple provides easy access to debugging console information from an iPhone while it is tethered to a development machine, making development on the iPhone itself less cumbersome than it otherwise might have been.

Other issues with the iPhone SDK included vastly differing behavior between different OS versions (things that worked with no problems on one version would crash or not run at all on others), and the repeated need to reinstall the OS on one of our devices in order to continue development.

# Conclusions and Future Work

In terms of audio capabilities, while the iDiMP project has demonstrated that the iPhone has the both the processing power and SDK support to handle fairly sophisticated audio tasks, the limitations of the UI on such a small device make it difficult to imagine it replacing laptop or desktop computers in their role as computer music “host” machine any time in the near future. Most computer music applications simply have too many parameters and controls to be represented on such a small screen in an efficient way.

The touch-screen and accelerometer certainly provide uniquely expressive ways to control audio synthesis and processing parameters, but each is limited to a small number of dimensions of simultaneous control. For example, the accelerometer can control at most three parameters in a practical way using its three axes of rotation, while motion on the touch screen is still limited to two-dimensional x-y coordinate locations. Such limitations lead us to believe that for now using the iPhone only as a controller within a larger integrated computer music system may be the best way to take advantage of its expressive value.

Despite that conclusion, however, there are a number of ways in which the iDiMP UI could be improved to provide a more powerful interface for a musician. Possibilities include having the app respond differently to double taps on the touch screen in comparison to single taps. At the same time, the iDiMP configuration pages could be designed in a more dynamic and flexible way to allow access to a larger number of non-real-time parameters aside from the real-time parameters controlled by the accelerometer and multi-touch screen interaction.

Another factor limiting the iPhone’s usefulness as a serious computer music device is the inability to obtain high-quality recordings using the cheap headset microphones, which are currently available. Even higher-end headsets are optimized for voice communications, so they do not work as well on more complex musical signals. Similarly, in terms of networking capabilities, the iPhone falls very short of the ideal tool for serious musicians to create distributed performances. The Wi-Fi internet connection lacks any kind of quality-of-service guarantee to support uninterrupted audio streaming, and streaming uncompressed audio suffers greatly under these conditions. Also, while the iPhone CPU may be powerful enough to perform the basic iDiMP audio processing without glitches, the added overhead of network communications can exceed the limits of the iPhone’s abilities.

Going forward, the iDiMP project is certainly worthy of more experimentation, and while it may not meet the quality and flexibility demands of serious performing computer musicians, the unique functionality that it does provide has the potential to grow into, at the very least, a fun toy. As the iPhone platform matures, along with hardware improvements, we foresee an increased viability of projects like our own. Combined with opportunities for significant new optimizations throughout the codebase, the future of iDiMP could be very bright indeed.

# References

1. New Interfaces for Musical Expression, <<http://www.nime.org/>>
2. OSCemote, <http://lux.vu/blog/oscemote/>
3. SoundWIRE Research Group at CCRMA, Stanford University, <<http://ccrma.stanford.edu/groups/soundwire/>>
4. Core Audio Overview, <http://developer.apple.com/iphone/library/documentation/MusicAudio/Conceptual/CoreAudioOverview/>
5. SpeakHere, <<http://developer.apple.com/iphone/library/samplecode/SpeakHere/index.html>>
6. Using RemoteIO Audio Unit, <http://michael.tyson.id.au/2008/11/04/using-remoteio-audio-unit/>
7. Roads, Curtis. The computer music tutorial. Cambridge, MA: The MIT Press, 1996.
8. Stevens, W. Richard, Bill Fenner, and Andrew M. Rudoff. Unix Network Programming: The Sockets Networking API. 3rd ed. Boston, MA: Addison-Wesley, 2004.
9. Dalrymple, Mark, and Hillegass, Aaron. Advanced Mac OS X Programming. 2nd ed. Big Nerd Ranch, 2005
10. Gruber, John. “Daring Fireball: iPhone-Likeness.” <http://daringfireball.net/2008/11/iphone_likeness>
11. Gallagher, Matt. “Singletons, AppDelegates and top-level data.” http://cocoawithlove.com/2008/11/singletons-appdelegates-and-top-level.html

1. For our purposes, the iPhone and 2nd generation iPod Touch are essentially equivalent, so references to “iPhone” should be interpreted as “iPhone and 2nd generation iPod Touch” unless otherwise specified. [↑](#footnote-ref-0)