A Mobile Music Environment Using a PD Compiler and Wireless Sensors

Robert Jacobs, Mark Feldmeier, Joseph A. Paradiso
Responsive Environments Group
MIT Media Lab
20 Ames St., Cambridge MA 02139, USA
rnjacobs@alum.mit.edu, carboxyl@mit.edu, joep@media.mit.edu

ABSTRACT

We describe a framework for a wireless sensor-based mobile music environment. Most prior work in this area has not been truly portable, or has been limited to simple tempo modification or selection of pre-recorded songs. The exceptions generally focused on external data rather than dynamic properties and states of the listener. Our system exploits a short-range wireless sensor network (using the ZigBee protocol and inertial sensors) and a compiler for PureData, a graphical music processing language. We demonstrate the system in an interactive exercise application running on a Nokia 800.

Keywords

PureData; PD compiler; ZigBee; interactive exercise; N800

1. INTRODUCTION

Many people with portable music players use them to generate soundtracks to their lives[5]. What would be more appropriate than being able to generate these soundtracks based on the activity being done at the time? For example, if a user went jogging, a system could synchronize the music to the rate at which he/she were running. Alternately, it could encourage the user to speed up or slow down as part of a exercise program[10]. This responsive environment would be able to synchronize to any periodic motion, such as their warm up or cool down period[16]. Dynamic leading or lagging can regulate the user's activity, encouraging exercise at the currently optimal pace. Going further, an array of simple body-worn sensors could immerse the jogger in music, with any motion producing appropriate sound. This paper describes a compiler for PureData and a wireless sensor system that enables efficient implementation of interactive music on a commodity handheld computer. Such systems promise to revolutionize the portable music player experience, ushering interactive compositions that never sound the same and need to be physically explored to be heard.

PureData[12] is a versatile language which was primarily designed for audio processing, although various extensions allow it to handle OpenGL-based graphics[1] or random-access full-motion video[2]. However, the fundamental design of PureData's runtime allows certain actions to be calculated in bulk (so called "signals", typically carrying audio data) while others are much more computationally intensive due to the amount of overhead from PureData's interpreter ("messages" – short lists of words and numbers).

In the past, many wearable music systems – either fully wired (like some of the very first systems), wired networks transmitting via wireless[14] or fully wireless[11, 4, 13] fed information to a central computer that analyzed the data and generated audio. This is fine for performance or other applications with small active areas, but does not allow for an experience that follows the user, particularly in cases where the user cannot accommodate heavy equipment like a laptop. Other projects exploring interactive music for exercise [16, 6] focused on only modifying the tempo and/or selection of pre-recorded songs to synchronize with the user, avoiding direct synthesis of real-time interactive music.

Gaye, Mazé, & Holmquist[7] and Vawter[15]invent a world very similar to what is envisioned for our project, involving deciding on a context from sensor data, and emphasizing natural rhythms heard by the wearer. However, their emphasis is more on integrating into the listener's city experience, and less on exercise.

Previous work in this direction has included PureData Anywhere (PDa)[8] – a port to embedded systems. Due to the common absence of floating point hardware on such processors, PDa made a compromise: floating point math would continue to be used for message logic, but 13.19 fixed point math would be used for the audio signals. This requires a dramatic difference in the function of two PD objects (`tabread~` and `tabwrite~`, which read and write audio data to look-up tables), and doesn't help for control-heavy applications, such as those with frequent sensor polling, because the use of floating point emulation for control signals is very slow.

2. COMPILER

Our PD compiler presents a middle ground between PureData and a lower-level language like C, with much of the ease of use of PD and much of the speed of C. By using a highly optimizing C compiler, many of the inefficiencies due to mechanical translation are further eliminated. For example, many objects in PD patches have exactly one incoming connection. A good C compiler, if told to optimize sufficiently, will take these objects and put them inside their callers. Further optimizations would then go and find redundant checks on the data type of the incoming message and discard the second set of checks.

Since we are converting between one format to another of text, we have written the compiler in Perl. Perl excels at parsing text, especially rigidly defined text like PD's save format. After parsing the entire file into a structure in mem-

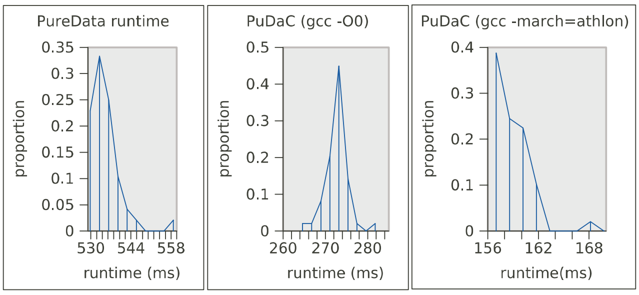

Figure 1: Histogram of time to multiply 2^{20} floating point numbers. Left to right: Time to run the patch in PureData, time to run the patch when compiled in GCC using the "no optimizations" flag (-O0), time to run the patch when compiled in GCC using the maximal optimizations (-O3)

ory, we execute the generators for all the objects that produce the C code for each.

The compiler takes in a plain text PureData patch file and produces C output. This allows us to take advantage of the large amount of work that other people have put into optimizing compilers without having to implement it ourselves.

The PureData Compiler (PuDaC) replaces each object with a uniquely-named subroutine (and possibly some uniquely-named globals). Each "wire" connecting objects is replaced with directly calling the connected object.

The compiler runs in two passes (three if you include the additional stage of running gcc). First it parses the input file, loading all the objects into an associative array with a UID for the object as the key. The value of each entry is an another associative array with several predefined entries specifying the object's arguments, the C representation of the objects attached to the inlets and outlets of the object, the C-generating perl code for this object, and a redirection specifying that the object has another name (like `sel` / `select`). Then it prints a prologue, executes the C-generating perl code for all objects, and prints the main function. This model makes debugging tremendously easier, although it is probably significantly less efficient than is possible. More details are presented in [9].

Still, a simple test patch (doing one million floating point multiplies) shows a significant performance increase over the plain interpreter: on an Athlon (Thunderbird core) running at 1066 MHz, PureData takes an average of 533ms, compared to as little as 158ms for the output of the optimized compiler, about 30%. (Histograms of trials are in Figure 1. The variance was consistently $\frac{1}{20}$th of the mean, probably due to the way the Linux scheduler works)

3. PHYSICAL INSTANTIATION

Nokia's N800 (figure 2) is a small portable computer (measuring 2.95 x 5.66 x 0.51 inches, weighing 7.26 ounces). It includes bluetooth, 802.11g, an ARM11 processor, a dedicated DSP, and runs the Linux-based Maemo internet tablet software suite. It shares general features with similar hand-held computing devices, so the exact choice of platform is flexible.

There are a variety of small single board computers that are capable of running Linux – for example, Gumstix makes a suite of Intel XScale based boards that range from slightly less powerful than the N800 to dramatically faster[3]. However, the Gumstix computers do not have their own battery, and the N800 has a floating point unit and agile user inter-

Figure 2: The Nokia N800 (right), with serial port exposed in an open back to show our custom adapter (left)

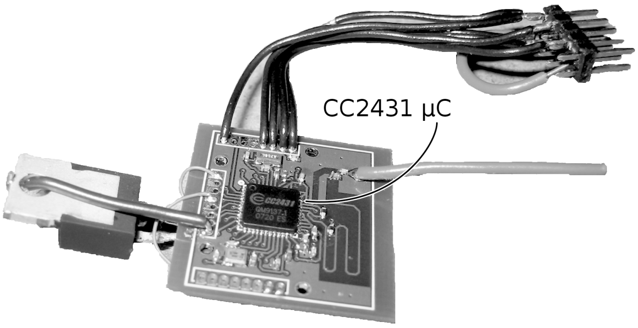

Figure 3: Front and back of the RF microcontroller to be attached to N800

face, making the amount of initial investment to get a useful system from an N800 much lower.

Our present requirements are sufficiently nongraphical to accommodate any battery-powered Linux machine with an FPU and a serial port (the N800's is in figure 2). That said, future incarnations of the system will benefit from the N800's interface in customizing the user's experience, much in the way parameters are set and adjusted on conventional music players. Additionally, we've minimized platform dependencies (beyond network in and audio out), so it should be straightforward to adapt this work to any other platform regardless of the overall requirements.

The sensor system is very straightforward. Each sensor should use a microcontroller, a simple radio, and appropriate sensors (e.g., multi-axis accelerometer, gyro, etc.). When we were introduced to the CC2431, it looked like an ideal solution, because it had almost everything needed already in it. It's tolerant of a wide range of voltages and so could be run directly off a battery, contains an ADC, and contains its own integrated ZigBee radio. All we had to add were the sensors. We use one CC2431 as a coordinator to initialize and gather the sensor data from all the others.

The wireless protocol has a few constraints: many transmitters, one receiver, no real need for collision detection, low bandwidth per transmitter (less than two kilobits per second), moderate-to-low aggregate bandwidth (less than a megabit per second), low latency, and permissibility of dropped data. Each sensor needs a coin cell, a low-power microcontroller, a transmitter and the sensor for that system. TI's ZigBee system was chosen.

TI provides, without charge (although with rather restrictive licensing terms) a software suite they call the Z-Stack, which provides a complete implementation of the ZigBee 2006 protocol, versatile enough for any use. However, the system is sufficiently large (since the standard is so complex) that it easily fills the vast majority of the 128kB of program space available to the program.

ZigBee is an RF standard designed for a variety of low-bandwidth applications. Multihop mesh routing is a pri-

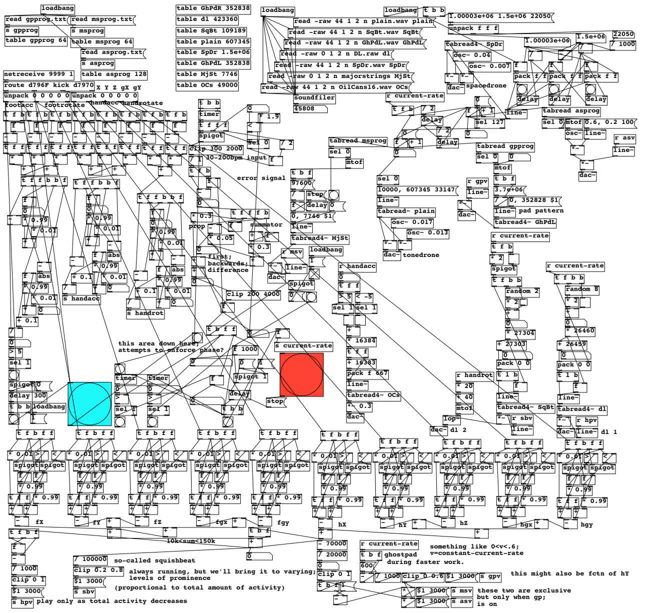

Figure 4: The final mapping used – several sounds are synchronized and queue from the user's motion

mary feature, although completely useless in this specific application since it imposes a significant latency cost and wearable sensors are within 1-hop proximity. Furthermore, unlike a simpler off-the-shelf design (such as a 9600 bps wireless modem), this is much more extensible to a large number of sensors and has better guarantees of data transmission.

The 5-axis IMU (Inertial Measurement Unit) board from SparkFun Electronics contains an Analog Devices ADXL330M and an InverSense IDG300. The ADXL330M measures ± 3.6 g on all 3 axes, and is configured to give a 50Hz bandwidth. The IDG300 measures $\pm 500°/s$ about the two axes not normal to the chip, and has a 140Hz bandwidth.

Inertial sensors provide the only way to determine how the user is moving without an external fixed reference. This IMU measures the user's footsteps, since landing on the ground is a large change in acceleration; however, the system also responds to fast foot swings and works relatively well on other parts of the user's body, such as the wrists.

The bridge between the N800 and our RF network (pictured in figure 3) contains a low-power microcontroller (to which bridge-mounted sensors can also be attached) and a ZigBee transceiver. More interesting mechanically than electrically, this involved attaching wires to the test pads inside the N800, using a LM1086 to regulate the LiIon battery voltage to 3.3V for the CC2431, and bending things into a nice flat shape to fit nicely behind the N800.

The N800 contains a serial port, apparently included for debugging the system in the factory. By using the flasher tool to enable the serial port, and disabling the getty normally running on the serial port, we can use the serial port to receive arbitrary low-bitrate data (115200 bps, 8 bits per byte, no parity, one stop bit – this is restricted, and appears to be an intentional crippling of the driver in the Linux source tree). Transmission is also possible, although the kernel and applications on the system send large amounts of debugging messages to the serial port and would have to be silenced first. A 33kΩ pulldown resistor is needed because the N800's serial port picks up ambient noise on the receive line, in turn causing it to reset.

The hardware serial port is very convenient since both it

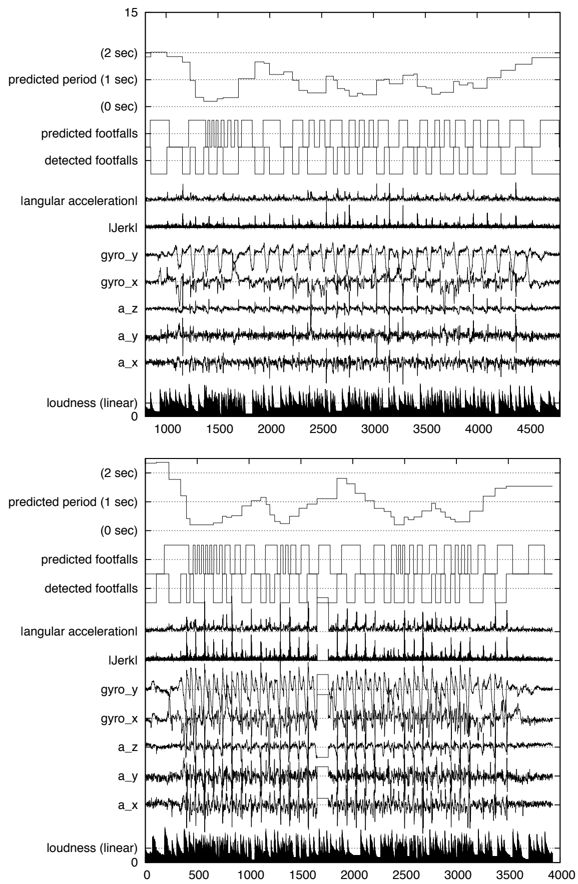

Figure 5: Graphs of results for walking (top) and jogging (bottom)

and the CC2431 support 3.3V logic-level serial. The N800 also supports switching its USB port to run in host mode, but using the internal serial port is simpler and smaller.

The microcontroller's reports are sent as a 16-bit sensor address followed by a series of 16-bit words. This small program converts these into the PureData-style `[addresssymbol] [space separated array of decimal numbers];`, handles the logical connection to the serial port and sends the resulting PureData-encoded datagram to a UDP destination of the user's choice, presumably to PureData or the compiled patch. PureData does not easily support binary data, so an external converter simplifies things tremendously.

4. EXAMPLE MAPPING

A working patch is shown in figure 4. The acceleration and angular velocity from the IMUs are converted to jerk and net angular rates. We then compute the ratio of local means to local average deviations, note when they exceed a given threshold, output this into a delay with holdoff. We also compute the local maxima and minima along each axis, and subtract to get a local dynamic range. The delay with holdoff drives a phase-locked loop (PLL) that attempts to match the phase and frequency of the input, which is attached to simple logic that runs several audio patterns. The dynamic range, rate of change of dynamic range, and current beat rate are used to select which patterns are played.

An example showing how well it works is seen in figure 5. The Z axis of the accelerometer (labeled a_z) was oriented normal to the leg, in the direction of stride (because no rotation was expected about this axis). The X axis (a_x) was parallel to the leg. |Jerk| and |angular acceleration|

were calculated as the magnitude of the first backwards differences on available axes. The detected and predicted footfalls plots indicate a detection or prediction when they change from low to high or vice versa. The period ranges from 0 to 2 seconds. Both plots are approximately 40 seconds (4000 centiseconds) long. The gap seen from 16-17 seconds in the bottom graph is because the data dumping program momentarily paused.

For jogging data, both gyro and accelerometer data provided good impulse sources, and the detected footfalls plot shows this. Unfortunately, as can be seen in both plots, the predicted period oscillates with a period of 10 steps, never really getting to the true pace (about 46/min when walking, 67/min when jogging). This is solely the fault of the PID (proportional integral derivative) controller inside the phase-locked loop. Later tuning should make this react better. The predicted footfalls trigger the beats of a 4-measure long drum loop and algorithmically launch other sequenced patterns according to the actions of the user.

5. FUTURE WORK

The compiler is distinctly to the point where it produces useful results, but it needs more effort to bring it to the point where it could be used in a commercial setting. It has a number of rough edges, as noted below, but can now be used on an experimental basis for further development.

Currently, there is no support for external patches or subpatches. Since PureData starts counting from 0 in each subpatch, and the current implementation doesn't recurse or handle multiple instances, the patch has to be flattened by hand before it can be compiled. Fortunately, flattening is a relatively easy, if tedious, task, only needs to be done once for each patch, and (if the patch is known to be destined for PuDaC) can be avoided altogether by not using subpatches. Furthermore, implementing subpatches in PuDaC is not difficult but was further down on the authors' list of priorities.

For efficiency and putting similar code together, we should aggregate similar objects. In PD, all the so-called "bin(ary)op(eration)s" inherit from a common class, so that each has only small fragments of C to specify the minor differences between them.

We have implemented a significant subset (approximately 70) of the total number of built-in objects in PD. There are at least another 70 to go to just implement the core functionality available in PD. We have, however, implemented enough objects to enable many control patches without further work, and adding additional built-ins onto the compiler should be easy and done on an as-needed basis.

Because the N800 has a floating point unit, we did not look into implementing automatic fixed point casting. As such, the compiler is not yet very useful on integer hardware such as cell phones or similar devices, but the small size of the N800 and functional equivalents compares favorably with current portable audio players. Additionally, several PD functions (such as `cos~` and `osc~`) are implemented using the FPU; for machines that lack a FPU, a lookup-table based solution will be necessary.

Because the current implementation was oriented to optimize control patterns, the DSP engine is suboptimal and behaves somewhat worse than PD's engine. This is fairly straightforward to fix, and then static analysis could be used to choose optimal fixed point representations throughout the DSP (and control) chain.

We can now make the sensor peripheral small enough to be a feasible device: the microcontroller is a mere 7mm^2 and the accelerometer is 5mm^2. The largest part is the battery, and a CR2032 lithium coin cell, containing 700mWHrs, could run the system for multiple hours. Furthermore, the single-item costs of this radio and accelerometer are $10 each, and the accelerometer already has lower cost versions. It seems likely that the microcontroller's bulk cost will drop even more within the next few years. This will result in a system that is both tiny and affordable for the end user that can run on a mobile phone with ZigBee. This work hints at a new art form going well below the simple music playback of the Nike iPod and essentially nonmobile Nintendo Wii, where music must be physically explored to be heard. When famous artists compose for this medium, the general public will have a strong motivation to exercise.

6. REFERENCES

[1] `http://gem.iem.at/`. Retrieved on 19 August 2007.

[2] `http://zwizwa.fartit.com/pd/pdp/`. Retrieved on 19 August 2007.

[3] `http://gumstix.com`. Retrieved on 24 May 2007.

[4] R. Aylward and J. A. Paradiso. A compact, high-speed, wearable sensor network for biomotion capture and interactive media. In *Proc. of IPSN '07*, pages 380–389. ACM Press, 2007.

[5] M. Bull. No dead air! the iPod and the culture of mobile listening. *Leisure Studies*, 24(4):343–355, 2005.

[6] G. T. Elliott and B. Tomlinson. PersonalSoundtrack: context-aware playlists that adapt to user pace. In *Proc. of CHI '06 - Extended Abstracts*, pages 736–741, 2006.

[7] L. Gaye, R. Mazé, and L.-E. Holmquist. Sonic city: The urban environment as a musical interface. In *Proceedings of NIME-03*, pages 109–115, 2003.

[8] G. Geiger. PDa: Real time signal processing and sound generation on handheld devices. In *Proc. of ICMC-03*, 2003.

[9] R. Jacobs. A wireless sensor-based mobile music environment compiled from a graphical language. Master's thesis, MIT EECS Department, Cambridge, Mass., September 2007.

[10] J. Nawyn, S. Intille, and K. Larson. Embedding behavior modification strategies into consumer electronic devices: A case study. In *Proc. of Ubicomp-06*, pages 297–314, 2006.

[11] J. A. Paradiso, K.-Y. Hsiao, A. Y. Benbasat, and Z. Teegarden. Design and implementation of expressive footwear. *IBM Syst. J.*, 39(3-4):511–529, 2000.

[12] M. Puckette. Pure data. In *Proceedings of ICMC-96*, pages 269–272, San Francisco, 1996.

[13] W. Siegel and J. Jacobsen. The challenges of interactive dance: An overview and case study. *Computer Music Journal*, 22(4):29–43, 1998.

[14] D. Topper and P. Swendsen. Wireless dance control: PAIR and WISEAR. In *NIME-05*, pages 76–79, 2005.

[15] N. Vawter. Ambient addition: How to turn urban noise into music. Master's thesis, MIT Media Lab, Cambridge, Mass., May 2007.

[16] G. Wijnalda, S. Pauws, F. Vignoli, and H. Stuckenschmidt. A personalized music system for motivation in sport performance. *IEEE Pervasive Computing*, 4(3):26–32, 2005.