PourOver: A Sensor-Driven Generative Music Platform

Kevin Schlei
University of Wisconsin-Milwaukee
3223 N. Downer Ave.
Milwaukee, WI 53211
kdschlei@uwm.edu

Christopher Burns
University of Michigan
1100 Baits Drive
Ann Arbor, MI 48109-2085
burnscl@umich.edu

Aidan Menuge
University of Wisconsin-Milwaukee
3223 N. Downer Ave.
Milwaukee, WI 53211
ajmenuge@uwm.edu

ABSTRACT

The PourOver Sensor Framework is an open iOS framework designed to connect iOS control sources (hardware sensors, user input, custom algorithms) to an audio graph's parameters. The design of the framework, motivation, and use cases are discussed. The framework is demonstrated in an end-user friendly iOS app PourOver, in which users can run Pd patches with easy access to hardware sensors and iOS APIs.

Author Keywords

generative music, mobile music, sensor mapping

ACM Classification

[Applied computing] Sound and music computing, [Human-centered computing] User interface design, [Software and its engineering] Software design engineering

1. INTRODUCTION

Generative music can be difficult to distribute in a way that preserves its generative qualities. Fixed recordings can capture a particular performance, but may be interpreted as a canonic representation by the listener, especially upon multiple listenings.

Generative pieces can use sensors or external input sources to add an interactive dimension to performance. Adding hardware requirements to the performance of a piece further complicates its distribution.

PourOver is an iOS app that provides a platform for creation, playback, and distribution of sensor-driven algorithmic / generative pieces. It aims to be a music player app built for generative music.

The underlying controller management is handled by PSFramework, a new open-source framework for iOS. PSFramework is built as a generic and extendable controller and API management system.

2. RELATED WORK

The iOS platform has provided tools for distributing generative, interactive, or multimedia-rich music apps. 'Album apps' such as Björk's Biophilia [2] and Brian Eno's Bloom [6] push the concept of 'open works' that invite user input to guide playback [5].

The role of the mobile device user as both audience member and participant is explored in echobo [10] and Nexus [1]. In echobo, audience members join a 'master musician' via a network, who controls high level musical structure. The NEXUS system provides a web-based distributed performance interface. Each blurs the performer / audience barrier through individual contributions to a larger collaborative performance. PourOver looks to apply this model to an individual listening experience by following many ambient, environmental, motion, and location changes.

The MoMu framework demonstrated how quickly accessible device parameters, paired with an included audio engine, could speed development of sensor-rich audio applications [3]. The PSFramework aims to solve similar design problems, but via a more generic protocol in which custom objects can easily register as controller sources, and the targeted destination is not a particular audio engine.

There are a number of projects that provide on-device GUI interactions to an underlying audio engine. The NexusUI JavaScript framework enables rapid GUI prototyping for instrument interfaces [11]. These pieces and instruments expose their interactivity through the portal of the device screen.

The urMus system provides a number of solutions for mobile music creation: sensor streams, value scaling, controller mapping, and networked live-coding all integrated with the built-in audio engine [7][8]. Through the urMus interface, musicians can design and perform pieces that leverage the device's multitude of interfaces. The system also allows for networked live coding where a desktop user can push code to another user's mobile device [9]. This design solution, where some coding was split from the device, was a significant inspiration for this research.

RjDj[1] was an iOS app that filtered external audio to create a reactive listening experience. Later, a desktop app, RJC1000, allowed users to customize their own RjDj 'scenes.' The PourOver app and framework also aims to give listeners a reactive listening experience. However, using Pd to create pieces affords a more general, open, and customizable content creation system.

Sensors2PD is a framework designed to quickly link Android device sensor data to Pd patches, and is the most similar in scope to PourOver [4]. The system sends controller values to numbered receivers (sensor1v0, sensor1v1, etc.). The system also supports touch input (sensorTIDvx, sensorTIDvy) and WiFi router information (sensorW-ID). PourOver looks to extend this kind of connectivity with a generic protocol design and expandability.

NIME'16, July 11-15, 2016, Griffith University, Brisbane, Australia.
.

[1]http://rjdj.me/

3. PSFRAMEWORK

The PourOver Sensor Framework (PSFramework) was developed by the authors to provide simple connectivity from iOS data streams to an audio engine's parameters, while automatically handling instantiating and clean up of source APIs. A variety of included control generating sources provide quick access to common sensor data.

The framework is open and extendable, allowing for new hardware sensors or custom controller streams to be added as they appear. Programmers can engage with the iOS side of an app by creating and registering their own controller generating objects with the framework.

PSFramework will be available on Github and installable via CocoaPods.

3.1 PSController

The framework is built around managing PSController objects. A PSController defines a data stream by name and transmits values in a normalized range.

For example, a PSController can be created to handle pitch attitude data from the gyroscope sensor. The controller would be initialized with the following properties:

```
PSController(
  name:"CMMotionManager.deviceMotion.attitude.pitch",
  min:-M_PI / 2.0,
  max:M_PI / 2.0
)
```

Controller names should follow a DNS-like, 'member of' naming convention. In this instance, UIKit's `CMMotionManager` class has a `deviceMotion` property, which has an `attitude` property, which is a struct that contains `pitch`. While this naming convention is verbose, it creates a strong bridge between the PSController name and the API it is referencing.

The gyroscope pitch sensor outputs values from $-\pi/2$ to $\pi/2$. In this case, the sensor range is a fixed limit. In others, such as `CMPedometer.speed`, a best estimate can be used for expected controller output ranges. PSController requires a value range because it normalizes its output. More information on controller ranges is discussed in 3.4.

PSControllers are designed to minimize excessive updating. They store a `sentValue` property, which is checked for equality on subsequent updates. They also have an `active` property to indicate whether a receiver exists for that controller. PSControllers will avoid sending superfluous messages based on these checks. This functionality can be overridden in cases where the behavior is undesired, for example when sending a single value repeatedly to act as a trigger.

3.2 Generators

Objects that produce control data and manage PSControllers are called 'generators.' Generators can implement hardware sensor APIs, handle user input, or create custom data streams (e.g. physics simulations).

Generators adhere to the PSControllerUpdating protocol. The protocol requires a static dictionary of PSControllers and method implementations for controller updating. The controller updating methods handle API instantiation, updating, and clean up. There is an option for generators to use a global timer for controller updating, indicated by a `requiresTimer` property.

Generators are instantiated automatically by the framework. When an outside object requests a controller, the framework will look for a generator class that contains that controller name. If it finds a generator class, it will load an instance of that class, or use the existing instance. This setup avoids unnecessary instantiation of objects, excess battery drain, and memory pressure.

Once a generator is up and running, it handles the pulling of data streams from its model. It then calls the PSControllerUpdating method `updateValue:forControllerNamed:` to send an updated data value through its corresponding PSController and on to its final destination.

3.3 PSControllerDelegate

To install PSFramework, instantiate a PSControllerCoordinator object and assign its `controllerDelegate` property. The PSControllerDelegate handles the reception of all PSController value changes. From there it can pass the updated value to the desired target.

3.4 Activity Modes and Controller Ranges

Sensor-driven applications normally need to scale sensor values to an appropriate range, since sensors can produce wildly different values depending on the use case. Presenting the complete range of a sensor in all cases only passes the issue to the user, who then must test for a practical range and scale their data accordingly.

PSFramework attempts to solve this issue by supporting four different activity modes during playback: walking, running, cycling, and automotive. These modes were chosen to match the output of `CMMotionActivityManager`, an object that analyzes the current activity of the device.

`CMMotionActivityManager` supplies real-time updates of the device activity along with an estimated `confidence` property. In tests, the ability to predict whether the device was stationary or in motion worked fairly well, but was skewed towards moving. Using the `confidence` property to filter out less confident reports fixed the issue.

A PSController can store separate value ranges for each of the four activity modes. When an activity mode is reported, the PSController object checks for a range for that mode and adjusts its minimum and maximum values appropriately. PSControllers that do not use different ranges will ignore the change.

This system allows pieces to produce a similar outcome regardless of whether the user is walking or driving. Conversely, a piece could be designed to follow mode changes and alter the music as they occur.

4. POUROVER APPLICATION

The iOS application PourOver has two major goals: provide a platform to play sensor-driven algorithmic / generative music, and give Pd coders a mobile, sensor rich platform for their patches.

PourOver prioritizes indirect user interactions over direct ones. There are no on-screen GUI controls that connect to audio parameters. This 'zero interface' approach generally suggests that pieces will make use of ambient, environmental, motion, and location changes rather than in-app user interaction.

4.1 Playback Interface

The PourOver interface similar to a music player. Users choose pieces to play and have basic playback control: play, stop, and timeline scrubbing. The app comes with a number of presets, and users can sync their own files through iTunes file sharing.

In addition to the standard playback controls, the app provides a 'freeze' button. Freezing a piece suppresses controller messages but does not stop any activity in the Pd file or cause the audio to mute. In a frozen state, controllers do not alter the state of the piece. The user can use this

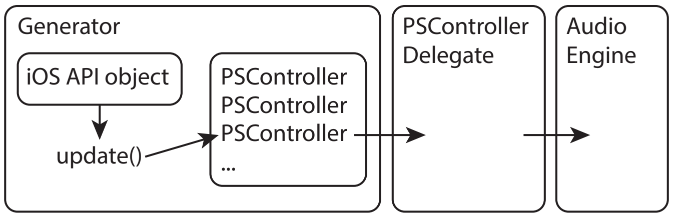

Figure 1: An API updated value patched from generator to audio engine.

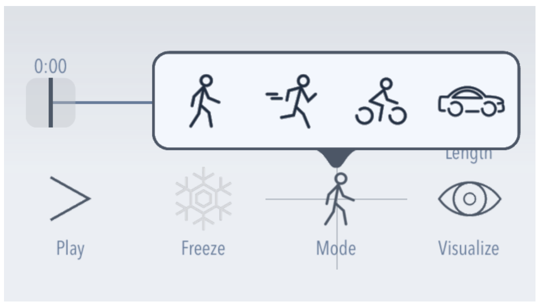

Figure 2: In the PourOver app, the user can override the detected activity mode.

button to hold on to a particular set of parameters and let the music continue in that state. When the user is ready to continue, they can press the 'thaw' button, and controller messaging resumes.

The timeline that displays the current time of the piece is a PSController. Scrubbing through the timeline does not move through an audio file, but rather communicates to the graph the current playback percentage. The piece can use this data to switch between formal sections of the music.

Finally, piece length is flexible and can be changed by the user from 1 minute to 24 hours. This may be useful in matching a particular excursion's duration.

4.2 Editing Platform

Creating content on mobile devices has had many approaches, including live coding, networked ensembles, on-the-fly mapping, GUI design, and distribution systems.

The PourOver app interface was designed for content playback rather than content editing. There are no patch editing capabilities[2], and no GUI interactions beyond the playback controls.

To create a piece for PourOver, a standard Pd patch (plus associated files) is copied into the iTunes File Sharing Documents directory of the PourOver app. This makes patch creation a desktop activity rather than an on-device activity.

PourOver will scan Pd files to build a list of pieces to load. Since Pd patches can make use of file abstractions, a method of determining a top-level patch was needed. To indicate a top-level patch, the following comments are required somewhere in the file: `//PSDECRIPTION:` and `//PSDEFAULTLENGTH:`

```
//PSDESCRIPTION: A really 'driving' piece.
//PSDEFAULTLENGTH: 180
```

The text after `//PSDECRIPTION:` appears in the app as the piece comments (the file name is used for the piece title). The `//PSDEFAULTLENGTH:` value sets the default piece length in seconds when the file is loaded.

An obvious area for improvement is the method of synchronizing Pd / asset files between the edited desktop versions and the device. The drag-and-drop iTunes file sharing method is tedious and often error-prone. It also requires the device to be physically connected to the computer, which is not ideal when the app testing often occurs in motion. While an ideal solution might look like the live code updating found in urMus, it is also possible that less direct, but more industrial solution such as Dropbox integration might be enough.

4.3 Pd Implementation

Some setup is required to link a target (in this case, libpd [?]) to PSFramework for message reception.

The target audio engine is responsible for requesting the controllers it needs. A solution was found that simplifies this process and minimizes error during development.

The libpd framework uses Pd's `[send]` and `[receive]` objects to communicate with the iOS side of the app. Rather than request a controller using `[send]`, then catch the incoming data with a `[receive]`, a new receiver object was created to handle controller requesting, data reception, and value mapping.

The `[psr]` object acts as a normal `[receive]` object, catching incoming messages matching its name argument. To request a controller data stream, it passes its controller name to the PSFramework when it finishes loading. The PSFramework then loads the required generator, and begins streaming data for the `[psr]` to receive.

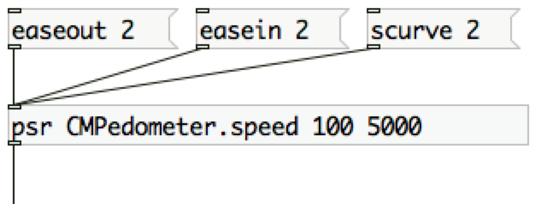

Figure 3: The [psr] object handles controller requesting and data mapping.

The `[psr]` object maps its output range to user supplied minimum and maximum value arguments. This removes the need to look up sensor data ranges. The object also provides data sloping functionality. By removing the need for redundant in-graph range adjustments and sloping, a significant amount of Pd message pressure can be reduced in controller heavy patches.

Multiple instances of `[psr]` can request the same controller data without causing duplicate generator instantiation. This provides some flexibility in programming, since each unique `[psr]` can map its output to a different range.

[2]Early design mock-ups contained the ability to map parameter controllers on-the-fly. This feature was abandoned in part to make clear the separation of desktop content creation and device playback.

While [psr] is available as a Pd external, its installation is not required for Pd users to write patches for PourOver. A dummy object with one inlet and outlet can stand in for [psr] during editing.

5. EXAMPLES AND TESTING

5.1 'Rose'

'Rose' makes use of heading data to control several variables of a generative composition with a pulse-oriented, minimalist aesthetic articulated by waveshaped oscillators. The piece cycles repeatedly through twelve sparse rhythmic patterns. Each is associated with a distinct pitch, and arrayed in a virtual circle around the listener. As the listener turns, new patterns gradually fade in and pan across the stereo field (opposing the listener's rotation, in simulation of a fixed spatial position), while previously sounding patterns gradually fade out and exit. If the listener pauses their rotation, the patterns gradually accumulate additional rhythmic elements, so that the piece tends to move from low to high rhythmic density over time.

5.2 'Floors'

'Floors' detects when a user moves between floors of a building and crossfades between sample playback states. Two methods of floor detection were attempted: the pedometer's `floorsAscended` and `floorsDescended` properties, and the altimeter's `altitude` property.

While the pedometer did detect changes, the `floorsAscended` and `floorsDescended` properties only detect when a user walks between floors. Elevator travel did not cause the controller to register a change.

The altimeter `altitude` property performed better. It detected both stairwell movement and elevator movement, and seemed to update more frequently and accurately.

5.3 'Virgo'

'Virgo' was designed to change based on movement direction, motion activity, and playback timer percentage. It follows the heading detected by `CLLocationManager` and detects whether the user is stationary. Changes in heading, for example rounding a corner or driving down a curved street, produce clear shifts in texture.

The piece structures changes around the physical stopping and starting of the listener. Layers enter and exit based on the direction pointed when stopped. A strong textural shift is programmed to begin at 70% piece duration. However, the shift waits for the next change in motion activity in order to pair the formal change with a moment of physical transition.

6. FUTURE WORK

The ability to easily share finished pieces between users, using an in-app interface, is a top priority for future versions of PourOver.

The editing / testing cycle would greatly benefit from a file synchronization solution, so the user can edit and update files on the desktop without the need to manually manage file copying to the device.

Similarly, testing sensor data could benefit from a sensor recording / playback system. During development, playback of a captured sensor performance could allow testing to occur at the desktop. As a distribution feature, users could store a particular 'performance' of a piece.

7. CONCLUSIONS

The PourOver Sensor Framework provides an opportunity to explore new combinations of generativity and interactivity, with a strong emphasis on indirect interaction involving movement, location and environmental factors. The open, expandable framework can grow with new device hardware or API additions.

The PourOver app has the opportunity to make generative playback experiences available to a large audience of iOS users, and gives composers of generative pieces a mobile performance platform.

8. ACKNOWLEDGMENTS

The authors would like to thank the Office of Research at the University of Wisconsin-Milwaukee for their funding and support for this project. Special thanks to Rebecca Yoshikane for her assistance.

9. REFERENCES

[1] J. Allison, Y. Oh, and B. Taylor. Nexus: Collaborative performance for the masses, handling instrument interface distribution through the web. In *Proceedings of the New Interfaces for Musical Expression conference*, 2013.

[2] Björk, L. One Little Indian, and L. Well Hart. Biophilia.

[3] P. Brinkmann, P. Kirn, R. Lawler, C. McCormick, M. Roth, and H.-C. Steiner. Embedding Pure Data with libpd. In *Proceedings of the Pure Data Convention*, volume 291. Citeseer, 2011.

[4] N. J. Bryan, J. Herrera, J. Oh, and G. Wang. Momu: A Mobile Music Toolkit. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, Sydney, Australia, 2010.

[5] A. D. de Carvalho Jr. Sensors2PD: Mobile sensors and WiFi information as input for Pure Data. In *Proc. of the International Computer Music | Sound and Music Computing Conference*, Athens, Greece, 2014.

[6] F. S. Dias. Album Apps: A New Musical Album Format and the Influence of Open Works. *Leonardo Music Journal*, 24:25–27, 2014.

[7] B. Eno and P. Chilvers. Bloom.

[8] G. Essl. *UrMus-an environment for mobile instrument design and performance.* Ann Arbor, MI: MPublishing, University of Michigan Library, 2010.

[9] G. Essl and A. Müller. Designing Mobile Musical Instruments and Environments with UrMus. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 76–81, Ann Arbor, MI, USA, 2010.

[10] S. W. Lee and G. Essl. Live Coding the Mobile Music Instrument. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, volume 1001, pages 493–498, Ann Arbor, MI, USA, 2013.

[11] S. W. Lee and J. Freeman. echobo: A Mobile Music Instrument Designed for Audience to Play. *Ann Arbor*, 1001:48109–2121, 2013.

[12] B. Taylor, J. Allison, W. Conlin, Y. Oh, and D. Holmes. Simplified Expressive Mobile Development with NexusUI, NexusUp and NexusDrop. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 257–262, 2014.