**Angelo Fraietta,* Oliver Bown,†
Sam Ferguson,§ Sam Gillespie,**
and Liam Bray‡**

*University of New South Wales
Faculty of Art and Design
PO Box 859
Hamilton NSW 2292, Australia
†University of New South Wales
Faculty of Art and Design
Corner of Greens Road and Oxford Street
Paddington, New South Wales 2021, Australia
§Faculty of Engineering and Information
Technology
University of Technology Sydney
PO Box 123, Ultimo, New South Wales
2207, Australia
**1/1a Monomeeth Street, Bexley, New
South Wales 2007, Australia
‡The University of Sydney
School of Architecture, Design and Planning
Wilkinson Building (G04)
148 City Road, Darlington NSW 2008, Australia
angelo@smartcontroller.com.au,
o.bown@unsw.edu.au,
samuel.ferguson@uts.edu.au,
sam@samgillespie.com,
liam.bray@sydney.edu.au

Rapid Composition for Networked Devices: HappyBrackets

Abstract: This article introduces an open-source Java-based programming environment for creative coding of agglomerative systems using Internet-of-Things (IoT) technologies. Our software originally focused on digital signal processing of audio—including synthesis, sampling, granular sample playback, and a suite of basic effects—but composers now use it to interface with sensors and peripherals through general-purpose input/output and external networked systems. This article examines and addresses the strategies required to integrate novel embedded musical interfaces and creative coding paradigms through an IoT infrastructure. These include: the use of advanced tooling features of a professional integrated development environment as a composition or performance interface rather than just as a compiler; techniques to create media works using features such as autodetection of sensors; seamless and serverless communication among devices on the network; and uploading, updating, and running of new compositions to the device without interruption.

Furthermore, we examined the difficulties many novice programmers experience when learning to write code, and we developed strategies to address these difficulties without restricting the potential available in the coding environment. We also examined and developed methods to monitor and debug devices over the network, allowing artists and programmers to set and retrieve current variable values to or from these devices during the performance and composition stages. Finally, we describe three types of art work that demonstrate how the software, called HappyBrackets, is being used in live-coding and dance performances, in interactive sound installations, and as an advanced composition and performance tool for multimedia works.

Background and Context

Many composers and performers today use algorithmic compositional techniques that map physical gestures and environmental changes to musical

parameters (Miranda and Wanderley 2006). Developers no longer require exhaustive electronics knowledge to design powerful interactive systems. The advent and evolution of microcontrollers and single-board computers has made the development of mobile music devices with built-in sensors readily accessible to enthusiasts (Cook 2017). Also, devices such as mobile phones are now being used by composers, utilizing the audience's personal devices as a source of both sensor input and sonic output (e.g., Rottondi et al. 2016).

Text-based music composition and performance is still highly popular in computer music circles, with languages including CSound, ChucK, SuperCollider, Perl, Ruby, Gibber, and Nyquist in common use (Collins et al. 2003; Keislar 2009; Roberts and Kuchera-Morin 2012). Moreover, creative coding has progressed from enthusiast or after-school club participation into the standard school curriculum (Thompson 2017), making text-based environments such as Arduino, Processing, and JavaScript familiar to many creative coders (Reas and Fry 2006; Margolis 2011). In extreme cases, live-coding performers may choose to start with a clean sheet and build their compositions from scratch, where the composition is an improvization that evolves over time (Magnusson 2011). The open-source ethos, online manuals and tutorials, the availability of free compilers, and the advancement of development suites are key components in the creative-coding movement (Bergstrom and Lotto 2015).

Creative media art is entering a new era powered by Internet of Things (IoT) technology, where sensors, wireless networks, cloud computing, and embedded systems are becoming seamlessly integrated into everyday objects and environments—such as public security, smart buildings, health management, shopping, business, and traffic monitoring. The IoT is regarded as the "third wave of information technology after Internet and mobile communication networks" (Zhu et al. 2010, p. 347). With the IoT impacting nearly every area of life, artists are utilizing this technology in their creative work. We have noted the rise of "media multiplicities" (Bown and Ferguson 2018b), whereby artists are able to exploit the capability afforded by smart devices to interact and coordinate with one other.

This facilitates the creation of media experiences that not only are spatial and portable, but also allow devices to send sensor information to other devices in the network. Moreover, these networks become scalable in that devices can be added or removed from the network (Bown and Ferguson 2018b). This effectively allows multiple devices to work together as a single, predefined artistic entity—similar to the concept of a modular synthesizer, where each unit provides specific functionality to the entire system.

When attempting to integrate these three disciplines—embedded system design, creative coding, and IoT—one needs to examine not only the features each discipline provides and how each discipline contributes to the whole system, but also what services are lacking, preventing the system from functioning as a unified entity. We contemplated future art that could exploit the attributes that these disciplines offer (Cook 2017) and considered what features would be required to combine them into a flexible yet robust unified system. First, devices should be capable of automatically connecting to the network and to one another without requiring that the user reconfigure them. Furthermore, it should be as easy to set up one hundred devices as it is to set up one. Also, each of the devices should be able to share the global resources as though they all existed on the same machine. Devices need to be adaptive, with automatic discovery of sensors, providing appropriate alerts and fallbacks when required. Serious coders require the highest-quality tools, not only for debugging code on individual devices, but also for building, developing, and debugging each part of the system from conception to realization. This means that each device needs to be immediately identifiable on, and accessible remotely through, the network, so that coders can see at a glance the state of the system. Also, coders must be able to use terse code through libraries and APIs that reduce verbosity and wrap common tasks in appropriate modules. Finally, coders should be able to expand and modify the system without having to rely on the system developers.

Although there are systems that address various aspects of these functional requirements, we were unable to find a system that successfully integrated all three of these disciplines. For example, some

systems facilitate embedded system design coupled with creative coding, such as Sonic Pi (Aaron, Blackwell, and Burnard 2016) and Bela (McPherson and Zappi 2015); however, they do not lend themselves to an IoT infrastructure. Similarly, novel instruments, such as the Sensus Smart Guitar (Turchet, McPherson, and Fischione 2016), integrate IoT with embedded design; but they do not facilitate creative coding. Also, although environments such as Processing make coding more accessible to the media artist (Reas and Fry 2006), the facility to develop advanced libraries while remaining inside the Processing environment is currently not possible, nor is it envisioned by the developers (Scheidl 2014). Consequently, creative coders must temporarily leave the Processing environment if their craft progression entails library development, as would have been in the case in the planetarium performance example detailed later in this article. We developed HappyBrackets to enable artists to write their compositions and send them to embedded devices for performance over an IoT infrastructure, facilitating creative coding practice without restricting the potential available in the workspace environment. Although HappyBrackets is focused primarily on music, it can be used on embedded devices and general-purpose computers for a wide range of contexts. In this article we describe HappyBrackets and discuss its use and design in light of the disciplines of embedded system design, creative coding, and for IoT.

Background to Research

Network technologies are increasingly enabling agglomerative media art and network music performances that involve multiple devices interacting over a network, both responding to commands and potentially sending state information (Bown and Ferguson 2018b). Ferguson and Bown (2017) discuss different design frameworks for working with large-scale "media multiplicities," noting that creators must often constrain their designs based on choices about whether computation will be distributed between the devices or mainly managed on a central server, with only basic instructions sent out to the device. Bandwidth and computation limitations conflict to make this a difficult design space in which to work creatively, as much planning and experimentation needs to go into any design.

Media multiplicities are increasingly being used in architecture and design. Building facades and internal areas are being shrouded in lights that need to be programmed to generate custom patterns, given the building geometry, aesthetics, and other requirements such as data visualization (Haeusler 2009). These facades are often recognizable as screens, no different from a computer monitor except in scale, resolution, and context. There are also many other media architecture configurations that demand different types of design, however, from radically low-resolution screens to unusual geometries. Similarly, we are finding networked media systems embedded into objects at a range of scales and with different affordances. For example, the Siftables project examined the use of small, tactile building-block devices with screens that could be used to build tactile digital experiences making use of the sensor networks that emerge when multiple sensing devices operate together (Merrill, Kalanithi, and Maes 2007). Hiroshi Ishii's work was an early examination of a world of tactile interaction with multiple interacting devices, this time in the creation of tangible moving tabletop surfaces (Ishii and Ullmer 1997). Other examples of moving media conglomerates include the Pixelbots (Digumarti et al. 2016) and Spaxels (Hörtner et al. 2012) projects. Similar to the world of images, the introduction of massive speaker arrays is increasingly common in performance spaces, although it is less common to see these as a feature of architectural integration into a space. Participatory, interactive conglomerate media experiences are also increasingly common in the world of art. A leading example is the work of Rafael Lozano-Hemmer, who creates participatory experiences in which something is added to the work by the audience members (Fernández 2007). For example, in his installation Pulse Room, sensors detect the heartbeats of individual audience members and render each measured pulse as a looped pattern on a light bulb in a room filled with hundreds of bulbs. The effect is aesthetically pleasing and stimulates emotional engagement through the sense of participation.

In music performance, a common theme has been the use of user devices—mainly smartphones—to create participatory musical experiences. The earliest work in this area worked within extreme limitations. An early example is *DialTones* by Levin et al. (2001), which literally involved calling people's cell numbers and utilizing their ringtones. By contrast, contemporary Web technology, notably the HTML5 media capabilities such as Web Audio and Web GL, enables rich control over audio and video processes directly through a browser on a smartphone. Web technologies have the advantage of being extensively supported across platforms, running in the virtual-machine environment of the browser. An obvious advantage of such a performance approach is that many people own smartphones and the performer can utilize the technology already in the hands of participants. Recent examples of multidevice performances include the work of Garth Paine, Andrew Bluff, and Ben Houge (Rottondi et al. 2016).

Distributed Systems

There is a plethora of embedded devices available for musical composition and performance. Programmable interfaces are generally one of two types: microcontrollers or single-board computers. A very popular microcontroller environment in use today is Arduino (Margolis 2011). Arduino is an extremely adaptable and accessible environment for development of interactive mobile devices. The code for the Arduino, known as a *sketch*, is written by the user in a simplified version of C++ and then compiled and uploaded to the Arduino device, running as a single program inside a loop function. Although it is possible to write the code to run many tasks, all the tasks must be coded and compiled together before being sent as a single executable code fragment into the target device. A change to the code requires a program restart, which is not conducive to live coding. In the live-coding paradigm, the artist no longer depends solely upon preexisting algorithms to create the art—the programming is the art (Bergstrom and Lotto 2015), and this can manifest itself as sound changing over time in

response to code that is changing or layering on the device. Our goal, therefore, was to treat multiple concurrent compositions running on the device as a standard, "out-of-the-box" feature that does not necessarily constitute an exceptional case when writing compositions.

Similarly, the use of single-board, small-form-factor computers as the basis for musical instrument design has enabled many artists to create complex novel musical interfaces. Various board form factors began to appear that allowed access to the CPU with general-purpose input/output (GPIO). This has enabled users to easily connect to the physical world similarly to the way developers could with microcontrollers. Furthermore, the availability of the Linux operating system for these single-board computers enabled them to seamlessly connect to wireless networks, with minimal effort required from the programmer. The most notable of these single-board computers are the Raspberry Pi and BeagleBone.

The Raspberry Pi is a single-board computer about the size of a credit card. It has an enormous user base, and it supports a significant number of plug-in sensors (Monk 2016) as well as a 128-GB SD memory card that can be used to store more than 200 hours of high-quality audio. Peripherals stack on top of one another, with many prebuilt sensors readily available, often making soldering unnecessary. For example, the Sense HAT add-on board plugs straight into the Raspberry Pi and has the following suite of sensors: triple-axis gyroscope, triple-axis magnetometer, triple-axis accelerometer, temperature, barometric pressure, and humidity (Monk 2016). The Raspberry Pi runs a derivative of the Linux distribution Debian known as Raspbian (Monk 2016). Raspbian's inclusion of compilers, its support for multiple coding languages, and its ability to concurrently run multiple programs provide the flexibility that enables a system to expand as an interactive platform as newer technologies become available. The Raspberry Pi ships with a platform for creative coding called Sonic Pi. Sonic Pi uses the SuperCollider sound-synthesis engine and is programmed using a creative-coding platform based on the Ruby language (Aaron, Blackwell, and Burnard 2016). Although it is simple for novice users to create

musical algorithms on the system, Sonic Pi does not lend itself natively to interdevice communication or implementation of custom sensors or physical outputs, the reason being that Sonic Pi is targeted primarily as an education tool (Aaron, Blackwell, and Burnard 2016).

The BeagleBone has been frequently used as a base for a multitude of professional and hobbyist projects and, like Raspberry Pi, feature extension boards can be stacked onto one another (Barrett and Kridner 2016). One particularly noteworthy system for the BeagleBone is Bela (McPherson and Zappi 2015). Bela is a BeagleBone-based platform that focuses primarily on digital signal processing. An in-browser editor enables users to program in C++, Faust, and Pd, as well as through the SuperCollider interface. New compositions are uploaded onto the device as source code, compiled on the device into machine code, and then run on the device. The Bela system relies on specific access to the programmable real-time unit of the BeagleBone Black (Barrett and Kridner 2016, p. 285), so the system is not easily portable to other platforms.

In recent years, systems have been developed that work with deployment to multiple devices. A leading example is the Particle.io system, which allows automated deployment of new code updates from a server (Vestergaard, Fernandes, and Presser 2017). Similarly, Digistump uses a pull-based method by which each device on a network regularly polls a server, as well as at boot time, to see if a new executable exists. The developer can upload a compiled executable to the server and can then either wait for the device to download the new code or restart the device. Each device automatically restarts once it has downloaded the new executable (Ferguson and Bown 2017; Ferguson et al. 2017). Although this method was utilized in *Bloom*, "a multiplicitous media artwork . . . consisting of 1,000 Wi-Fi networked computational devices" (Bown and Ferguson 2018a, p. 53), it does not function as a real-time update.

Although both the Sonic Pi and Bela facilitate live-coding platforms for embedded systems, they do not lend themselves easily to conglomeration or to functioning as IoT systems, because devices running these systems are not expected to communicate with one another in an IoT infrastructure (Turchet, Fischione, and Barthet 2017). Our goal was to address this gap by exploiting the opportunities afforded by these devices through their agile networking capability, powerful operating systems, small form factors, and low cost. Composers should not have to concern themselves with device and network configurations to develop an ensemble, because each computer should be able to control any number of devices simultaneously, and likewise, any number of computers should be able to control the same embedded device. Moreover, any number of devices should communicate with and respond to any number of other devices within the IoT domain.

Initial System Development and Compositions

Many text-based compositions use new languages that run through an interpreter (Magnusson 2014). Rather than require someone to learn a music-specific programming language, we set out to develop a system that used a common programming language that was easy to learn and portable across many embedded and desktop platforms. Moreover, we wanted the language to be sufficiently flexible and powerful to engage the expert programmer. We also wanted to provide access to the highest-quality professional development tools that would facilitate modularization and terse code. Finally, we required a language that had significant support for exchange of data over networks and was endorsed within the information technology community. We chose the Java programming language because it met all these requirements (Horstmann and Cornell 2002). Similarly, the Processing programming environment, which was specifically developed for the media arts community, uses Java as its core language (Reas and Fry 2006). Also, Java is currently used by a significant number of universities for first-year computer science students (Tabanao, Rodrigo, and Jadud 2011), ratifying its suitability for both the media artist and the information technology community.

The HappyBrackets project started as "A Java-based remote live-coding system for controlling multiple Raspberry Pi units" (Bown, Young, and Johnson 2013), where a master controller computer

sent precompiled Java classes to selected Raspberry Pi devices on a network. Unlike the Arduino sketch, the HappyBrackets composition is not a standalone executable program. The HappyBrackets core has a thread that listens for incoming bytecode classes, and after receiving a class, executes the new class's functionality through a Java interface. This allows for multiple concurrent compositions that can be easily created or updated during composition or live coding (Ferguson and Bown 2017). One of the pieces written for the system was *ChatterBox*, by Miriama Young, where ten Pi modules were placed in impermanent housings and distributed discretely throughout the audience. In a second context, the devices were suspended over a large staircase with a sensor used to detect movement within the space.

This research was extended with the development of the Distributed Interactive Audio Device (DIAD, cf. Bown et al. 2015), which contained an inertial measurement unit (IMU) consisting of an accelerometer, a gyroscope, and a compass. The devices were handled by the audience and incorporated into the environment. The DIADs not only responded to user manipulation, they also responded to one another. Furthermore, DIADs were configured to automatically connect to the wireless network, and once a DIAD came into range of the network, it became a part of the DIAD agglomeration. The main focus of this project was the development of a reusable platform that allowed creators to develop interactive audio and easily deploy it to other devices. This was effected through the use of a client-server architecture in Java. Later, the DIADs were incorporated into a musical game of bowls and exhibited in *Musify+Gamify* in the 2015 Sydney Vivid festival (Bown and Ferguson 2016).

System Overview

Our intent was to develop a platform where the composer or developer could enter the IoT domain with minimal effort to diagnose, develop, or interact with the distributed system. We wanted to be able to add or remove devices without having to reconfigure the system, particularly during a live performance. The HappyBrackets system runs with one or more controller computers for composing and for configuring and controlling devices, one or more embedded devices running compositions, and a local area network. Composers write the code as Java classes and send compiled Java bytecode to the devices for performance. The devices can respond to sensor input, and, depending upon the algorithm in the composition, generate sound or some other output. Additionally, devices are able to communicate with one another and respond according to the logic inside the compositions. Compiled compositions can be stored on the device's SD card if they are required to run on startup—a feature particularly useful for installations or as a preconfigured instrument. The system is designed to run on a local area network—Internet access is unnecessary (extending the performance to a wide area network using an Internet connection is trivial, however).

Device

The core HappyBrackets application is installed on the device's SD card as an executable Java archive that is launched when the device boots and then executes any Java compositions sent to it. Although HappyBrackets runs on many embedded platforms, the main research has been with the Raspberry Pi, primarily because of the availability and low cost of the devices. Composers are not limited to controlling only Java code, but can run other programs or scripts written in different languages from within a HappyBrackets composition. We present examples later in this article where HappyBrackets was used to develop a virtual spacecraft with a sonic poi, and a composition based on planetarium software utilizing JavaScript. A poi is a device used in a performance art related to juggling, known as *poi spinning*, "where weights on the ends of short chains are swung to make interesting patterns" (Farrington 2015, p. 173).

HappyBrackets includes a convenient sensor library that autodetects certain accelerometer, gyroscope, temperature, and magnetometer devices. Moreover, people are able to write their own sensor or output device drivers through access to the

GPIO without having to develop kernel-specific modules or modify the underlying HappyBrackets Java archive file. One of the authors is currently using this approach to interface with hall-effect rotary encoders in an installation using robotic, musical roller skates.

Controller

The controller is a host computer that the composer uses to write code, configure devices, and deploy compositions to one or more devices. The HappyBrackets developer kit contains a purpose-built plug-in that customizes a professional integrated development environment (IDE), IntelliJ IDEA, for seamless interfacing and communicating with the devices. Composers write and compile Java code on a controller computer, then send the code through the plug-in to the device for performance. The controller displays each device in a list, enabling the user to start sending commands or compositions to one or more devices.

Design Methodology

The system was created using an iterative development methodology, in which each iteration of the project provided "valuable feedback . . . as an input to direct or influence the next iteration of the project" (Fraietta 2006, p. 21). Although logic and software choices were generally made scientifically—such as measuring the amount of time certain operations took to complete, or the number of errors certain transmission types produced—the human–computer interface development was guided by general observations rather than strict evaluation metrics. Additionally, two of the authors developed a series of online lectures (with associated assessments), which were presented as an on-demand, massive open online course, in partnership with the online learning organization Kadenze. Example code, demonstrations, and "how-to" documents were distributed via GitHub. And for those users needing to install the compiled code onto an SD card to configure Raspberry Pi, we provided a prebuilt image. Demonstrations within the online lectures contextualized and explained each of the required software and hardware elements. These include attaching a simple sensor board, setting up a Wi-Fi network to control the Raspberry Pi, debugging with an external device, and working through a number of examples provided.

Three areas of feedback were from users from the Kadenze course, from workshops, and from the authors' own use of the software to develop compositions. Feedback from the Kadenze course was useful for identifying bugs and general issues; however, it was difficult to determine whether users had difficulties with the software, because many could work out issues in their own time. Furthermore, we could not determine whether a user had completely disengaged and dropped out because they thought the system was too difficult to use. We found observations from workshops to be the most useful feedback for elementary human–computer interface refinement. We found, however, that developing our own compositions was invaluable for determining not only what a frequent user of the system would require for complex compositions, but also for identifying features we had previously added that actually hindered work flow.

Workshop

A workshop was presented at the 2017 ACM SIGCHI Conference on Creativity and Cognition (Ferguson and Bown 2017), where participants were given an opportunity to explore some of the possibilities of HappyBrackets. Next, a workshop for twelve people was conducted at the University of New South Wales in January 2018, whose primary focus was not only to introduce HappyBrackets to other interactive media artists, but also to determine what features were beneficial to creativity, and what hindered creativity or caused frustration (Bown et al. 2019). A preworkshop survey was completed by the participants to determine their skill levels. All the participants were experienced in computer-generated music, had previously programmed with physical sensors, and understood the concepts of samples, oscillators, waveforms, and gain. Although most of the participants

Figure 1. Original code for using an accelerometer sensor.

```
Accelerometer sensor = {Accelerometer}hb.getSensor(Accelerometer.class);
if (sensor != null) {
    // Add the listener
    sensor.addListener(new SensorUpdateListener() {
        @Override
        public void sensorUpdated() {
            float x_val = (float) sensor.getAccelerometerX();
        }
    });
}
```

had experience programming in a text-based environment, the majority identified themselves as having little or no experience with Java.

Composing using a text-based paradigm is effectively writing a program, and "learning to write a program is a difficult task" (Tabanao, Rodrigo, and Jadud 2011). We observed during the workshop that there was some participant frustration, which appeared mainly due to the amount of manual typing required to generate simple functionality. For example, adding an accelerometer sensor and reading the *x*-axis required the user to manually type the entire code fragment shown in Figure 1. We found the users constantly had to look at the slides that showed the code example, which in some cases were quite lengthy. Furthermore, it was extremely common for the participants to type code in the wrong place, which in turn produced compile errors. Once users had generated code that compiled, they had to send the composition to their Raspberry Pi in order to hear it. We found that once participants had reached that stage, it was preferable for them to copy their Java class files rather than generate a new class owing to the amount of typing required to make the most basic of compositions.

After the workshop, we invited the participants to complete an online evaluation survey. Based on the evaluation responses and our observations during the workshop, our assessment was that HappyBrackets was too difficult to use in that state. Investigation into the difficulties that previous researchers had discovered directed our strategy to refine the users' compositional or programming experience for HappyBrackets, specifically addressing many of the problems we witnessed.

Reducing Programming Difficulties

Many studies indicate that difficulties in programming may be caused by a lack of a mental model of what was actually happening in the program (Milne and Rowe 2002), misconception of programming constructs, lack of programming strategies, and deficient debugging strategies (Tabanao, Rodrigo, and Jadud 2011). We do not intend to make programming in Java "easy," per se; its challenge is analogous to that of learning to play the violin. There are, however, techniques that teachers use to help students overcome steep learning curves. Although one might consider it nonsensical to put frets on a violin, teachers often place tape on the fingerboard to help the student learn to develop their intonation. Likewise, a sophisticated IDE facilitates learning the fundamentals of coding, and can later become a part of the composition or performance activity.

One study into programming problems for novice Java programs revealed that 59 percent of compilation events by students resulted in errors (Tabanao, Rodrigo, and Jadud 2011). Additional research indicates that access to automatic code-generation functionality and debugging tools capable of inserting breakpoints significantly improves the quality of the programs generated (Dyke 2011).

Although the Processing environment addressed many of these issues and helped make programming less intimidating to the beginner (partly because of its large community base of creative coders), the Processing language was "specifically designed for generating and modifying images" (Reas and Fry 2006, p. 527). This graphical or visual bias is

reinforced through the setup() and draw() functions, each of which, like Arduino, create a single program that runs inside a loop function. Also, the "processing development environment (PDE) is built with the sole purpose of creating sketches . . . that have a few files at most" (Scheidl 2014). Research indicates that creative coders "need the ability to mix and match different libraries directly in their code" (Vestergaard, Fernandes, and Presser 2017, p. 3). At the same time, the introduction of creative coding in the school curriculum is contributing to the production of a generation of artists who are becoming increasingly digitally literate (Tuomi et al. 2018). Competent users wanting to develop a complex library would be required to switch between the Processing environment and a separate IDE (Scheidl 2014). The facility to develop a library in the same environment as the artist's normal coding reduces the creative feedback time, producing greater creative flux (Mitchell and Bown 2013). An example is provided later in the article where a library to control planetarium software and access online astronomical catalogues was developed in the context of composing a work, without requiring the composer to switch between music-composition and library-development mindsets. We concluded that the restrictions placed on the coder using the Processing environment outweighed the advantages provided for the novice. In particular, we wanted to avoid requiring users to change programming environments if their craft extended in this direction.

We found that IntelliJ IDEA (hereafter IntelliJ) was able to help us address and mitigate these issues, in that it provided advanced features to facilitate learning Java without imposing future boundaries caused by a confined IDE.

InteliJ IDEA

IntelliJ is a Java IDE developed by JetBrains. In a review in *InfoWorld* (22 September 2010, "Top Java Programming Tools"), Andrew Binstock reports that IntelliJ received the highest score out of the four favorite Java programming IDEs, the other three being Eclipse, NetBeans, and JDeveloper. IntelliJ provides powerful features that facilitate

code and environment development. These include intuitive and context-sensitive suggestion of variable names, custom live templates, and advanced plug-in creation and development. Although IntelliJ is distributed as proprietary commercial software, HappyBrackets requires only the free Community Edition, which is distributed with the Apache 2.0 free-software license. A key IntelliJ feature that we utilized is its ability to develop a custom plug-in that allows a seamless interface between the coding environment and the distributed devices.

Prevention of Syntax Errors

Research has indicated that 26 percent of errors produced by users were that the compiler could not find a variable or method (Tabanao, Rodrigo, and Jadud 2011). Although many IDEs have a feature whereby the editing software will suggest the name of the variable or method to use, IntelliJ enhances this feature, such that it will only offer valid suggestions based on the context. The result is that users will only be offered suggestions that will actually compile.

Another significant error was caused by missing parentheses, brackets, or braces, accounting for 10 percent of the errors. This can be very complex, as Java enables nested anonymous classes and interface realization (Horstmann and Cornell 2002). When IntelliJ detects that an anonymous class requires realization through context-sensitive suggestion, the editor generates the entire section of code required to produce valid code that will compile.

Another powerful feature of IntelliJ is the ability to automate code generation using Live Templates. For example, if a composer wants to add an accelerometer, which requires a block of code several lines long, it would suffice to simply type "accelerometerSensor." After typing only the first few letters, the IDE would suggest completing the rest of the text (see Figure 2).

When the user presses the tab key, the entire code fragment shown in Figure 3 is automatically inserted into the composition.

Comparing the code in Figures 1 and 3 reveals the difference between the manually typed code

Figure 2. Live Template autosuggesting accelerometer sensor code.

Figure 3. Live Template generated accelerometer sensor code.

```
@Override
public void action(HB hb) {
    hb.reset(); //Clears any running code on the device
    //Write your sketch below
    acc
    accelerometerSensor    Inserts an accelerometer sensor.
Press ^. to choose the selected (or first) suggestion and insert a dot afterwards >>
}
```

Figure 2

```
/** Type accelerometerSensor to create this. Values typically range from -1 to +1 **/
new AccelerometerListener(hb) {
    @Override
    public void sensorUpdated(float x_val, float y_val, float z_val) {
        /* Write your code below this line */

        /* Write your code above this line */
    }
};  /* End accelerometerSensor */
```

Figure 3

required in the original workshop and the code now generated using the live template. In addition to simply automating the process of sensor creation with live templates and eliminating incorrectly typed code, we significantly simplified the API by returning the three accelerometer axis values through the sensorUpdated() method. Coupled with this, we generate comments indicating to the user where to add code needed to provide the behavior desired, an idea inspired by the code generation output produced by the Rational Rose computer-aided software engineering tool (Quatrani 2002; Fraietta 2006).

Debugging

Debugging code that is running on an external device is a complex task. This is exacerbated when the instrument has no viewable output. Syntax errors in the source code would have already been captured by the Java compiler, so the issue addressed

here is a compiled composition that does not run as desired. This is separated into two categories: the case where a composition that does not operate as expected, such as when an accelerometer output is not producing what the composer believes it should; and the case where the composer is searching for the right sound or functionality.

Visualization of Values

Milne and Rowe (2002) conducted research into the difficulties in learning to program and came to the conclusion that being able to visualize what is happening to the parameters in the program's memory facilitates learning and produces better results. Composers may not know the exact "numbers" to use in their algorithm to obtain the exact sound or effect they desire. Keislar (2009) suggests that manual manipulation of virtual objects with graphical widgets in combination with real-time sound synthesis or processing simulates a more natural experience for the musician. Similarly, using

Figure 4. Tuning synthesis parameters with DynamicControls.
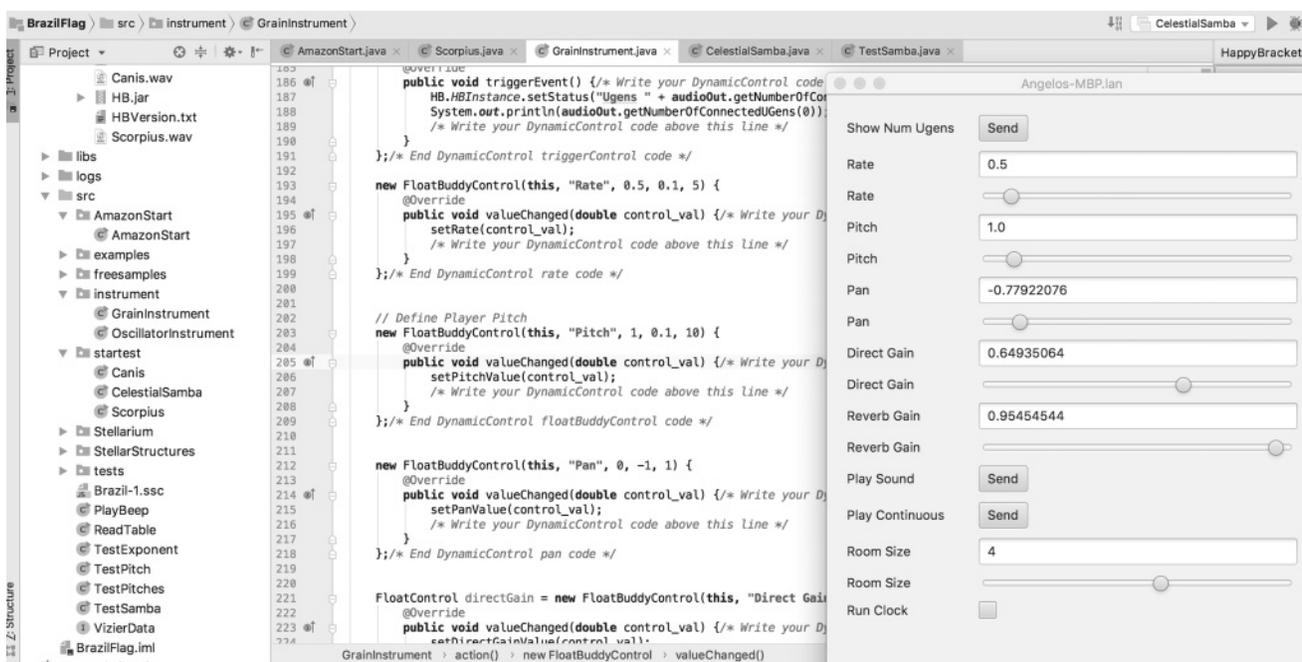
GUI objects to set and retrieve the current musical parameters facilitates rapid composition, because it allows composers to hunt for the sound without having to send a new composition.

Retrieving values from a remote device or instrument is known as *telemetry* (Fraietta 2005). The concept of visualizing parameters from musical instruments is not new, having been used in synthesizer patch editors for many decades through MIDI system-exclusive messages (Huber 2012). Although HappyBrackets is a text-based paradigm for creative coding, composers need to be given the option to control values using controls like a slider or checkbox, if desired.

Dynamic Controls

We implemented an object called a Dynamic-Control, which, although originally envisaged to transmit variable values to a graphical display in IntelliJ, became the mechanism whereby data could be easily communicated among compositions on the same device and to other devices on the network. Message types communicated are similar to those used in other programs: floating point, integer, Boolean, string, and trigger. In some cases, floats or integers presented as a slider control will better serve the composer, because of the ease of hunting for a value. A slider can also be paired with an editable text box that displays the slider's value and vice versa, producing a "buddy" pair. Figure 4 shows examples of trigger controls as buttons, a Boolean control as a check box, and integer and float controls as buddy pairs. This example was used to test various synthesis values for an instrument during a composition described later in this article.

Pitch, for example, can be changed by either typing a value into a text box or by manipulating a slider. The name of the control does not need to be unique; however, the name determines the interconnection with other DynamicControls having the same name and type, depending upon its ControlScope. The name is displayed next to the value on the GUI in IntelliJ. Any time the value of the object changes, typically from within the composition, the GUI object will display the new value.

Fraietta et al.   **99**

Although similar to the send and receive objects in Max, in that the name and type parameter of the DynamicControl determines message interconnection, DynamicControls also have an attribute called ControlScope, which dictates how far (in a topological sense) the object can "reach" to communicate with other DynamicControls, as will be explained shortly. DynamicControls can be bound to different objects, the default being the class that instantiated the DynamicControl.

For example, consider the DynamicControl called Pitch in Figure 4. The default scope of DynamicControls is SKETCH, which in this case means that only other FloatControl objects called Pitch within the same instance of GrainInstrument will communicate with one another. The FloatBuddyControl class is a specialized type of FloatControl that displays both a text box and a slider. If the composition is loaded again, a separate FloatBuddyControl will be created; however, the values between the two controls will be separate.

If the ControlScope of the Pitch object were set to CLASS, adjusting the control on either instance of the GrainInstrument would affect the other. This could be effective for controlling a global setting for a particular type of instrument. If the ControlScope were set to DEVICE, all FloatControl objects called Pitch with a ControlScope of DEVICE would receive this value. This type of control scope effectively facilitates global control on a device, for example, a reference pitch for all instruments on a device. If the ControlScope were set to TARGET, it would be possible to transmit the message to one or more selected devices on the network.

The final type of ControlScope, GLOBAL, allows the control to transmit the value to all devices, whether they are within the same composition, class, or device, or on another device on the network. The user does not need to think about IP addresses, device names, or ports—only the name, type of variable, and the desired reach of the control into the network (i.e., a global scope, or a more limited one). Although we currently use Open Sound Control (OSC) for transmitting global values across the network, the message is not encoded to OSC unless the value needs to go across the network. This significantly reduces the possibility of side

effects caused by the inefficiencies of OSC (Fraietta 2008; Fléty and Maestracci 2011; McCurry 2018; Turchet et al. 2018), as messages that remain on the device are always passed by reference (Gosling, Joy, and Steele 2000).

Running Code without a Physical Device

Tabanao, Rodrigo, and Jadud (2011) described three types of coders as "movers, stoppers, and extreme movers." *Movers* are those who use feedback and thinking to refine and develop their algorithms; *stoppers* are those who become frustrated and give up; and *extreme movers* are those who just make random changes in their code until they come up with something that works. Our aim was to help the user remain in the mover category by allowing faster and intuitive feedback. Apart from the difficulties just in writing code, the problem with having to interface with a separate hardware device has challenges, particularly in the time taken between making a change in code and hearing it. The longer it takes to hear or see the result of a change, the more frustrating it can be for the composer. Several factors determine how long it can take between making the code change and hearing it for feedback.

One of the factors that determined how fast a student was able to develop a program was the amount of time it took to compile the code and run it. Research by Gregory Dyke (2011) indicated that the ability for people to work in short, fast sessions improved the outcome of the work. Composers may have an idea and just want to test it without having to power their device up. The ability for composers to just open their laptop, work for five minutes to test an idea, and then close the laptop would be particularly conducive to rapid composition development. Also, it may not be practicable at times for composers to use a hardware device—for example, while travelling on public transport, if the hardware was unavailable, or if the artist is unwilling to commit to hardware purchase at that stage of the project. Running without the device allows artists to evaluate their compositions to a significant degree and to rapidly make and listen to changes. The two methods used to address running code without a physical device were running the

virtual HappyBrackets device, and running the composition within the context of the IDE.

Simulating a hardware device on a development computer is not novel, having been used in emulators in engineering circles for many years (Berger 2001). Rather than emulating a physical CPU, the development computer has identical code running as a separate process. For all functional purposes, the virtual device appears and behaves the same from the point of view of the controller plug-in as though it were a physical device. Although the virtual machine does not have physical sensors—such as accelerometers, gyroscopes, and GPIOs—sliders, text boxes and check boxes are provided to simulate the sensors, enabling composers to simulate the sensor without real hardware being available at the time (Fraietta 2005).

In some cases, a simulated value may be better suited than a real sensor. If, for example, a sound required a change over a minute based on a value of a gyroscope, the composer would need to physically turn the gyroscope for that time at the required rate. Simulating this value can make it physically easier, as setting a slider or typing a text value removes the requirement of physical rotation during this stage of composition.

A facility has also been added whereby a composition can run as a standalone program in the IDE. The advantage of running the composition inside the IDE is twofold. First, the composer does not need to reset the previous patch and send a new version. The composition can be immediately stopped and rerun in a fraction of the time, allowing faster iterations between composition modification and evaluation. Second, the user can set breakpoints on a particular line of code and then step through the program to evaluate variable values, which in turn helps the composer maintain a mental model of what is actually happening inside the composition (Milne and Rowe 2002). Moreover, IntelliJ facilitates advanced debugging by decompiling third-party Java libraries during runtime, allowing the debugger to step into them without requiring the composer to switch programming environments. Running the code from within the IDE was found to be far superior to the virtual device for standard debugging; however, the virtual device was better suited

to testing interaction between the controller and device.

Online Resources

HappyBrackets is available as a GitHub repository at https://github.com/orsjb/HappyBrackets, with instructions on how to install it available at www.happybrackets.net/doc/Getting-Started. Videos are available on YouTube with tutorials that demonstrate:

1. environment setup www.happybrackets.net/doc/setup
2. fundamental compositions www.happybrackets.net/doc/fundamental
3. advanced configuration www.happybrackets.net/doc/config, and
4. art works and performances www.happybrackets.net/doc/artworks

and more.

Art Examples Using HappyBrackets

We present three different types of art projects that used HappyBrackets significantly. The first is an improvisational performance using two networked, interactive sonic balls coupled with live coding. The second is an interactive installation for 25 networked Raspberry Pis, player piano, and robotic percussion. The third work is a composition for planetarium software, electronics, and percussion ensemble.

The first two narratives will describe briefly how HappyBrackets was used within the works. The third description, however, will also include the detailed creative process used by the composer to illustrate how HappyBrackets facilitates rapid composition from inception to realization.

Live Coding with Interactive and Responsive Sonic Balls

Two compositions for improvisational dance and live coding were realized using two DIADs in the

Figure 5. So Predictable!?
performed at Now Now
festival in Sydney in 2018.

Figure 6. Spiral
installation at the
Powerhouse Museum.
(Photo: Ryan Hernandez,
reproduced courtesy of the
Museum of Applied Arts
and Sciences.)

form of interactive sonic balls. Similar to the musical game of bowls (Bown and Ferguson 2016), each ball contained a Raspberry Pi, a loudspeaker, and an IMU encased in a spherical case (Loke et al. 2018). The first work, *So Predictable!?*, focused on merging movement improvisation with indeterminate sonic balls, where neither the performer or audience are able to predict the sound the balls will make. Teetering on the edge between music and noise, the audience faces an enigma: Who leads? who follows? and will the dancer's next move create harmony or chaos? The work was performed at the Now Now festival in Sydney, Australia, shown in Figure 5, and then at the New Interfaces for Musical Expression conference in Blacksburg, Virginia, USA. Both performances took place in 2018, a recording of the NIME performance is online at https://youtu.be/_wZrVDUnoVE.

The second composition built on this work as a semi-improvised sound game, where each player was invited to explore the relationship of movement to sound through play (Loke et al. 2018). The networked configuration of two DIADs enabled collective modes of interaction and behavior, effectively sharing each player's movement features between the two balls. Although players generated their own movements to manipulate the sound, they were also required to simultaneously listen and attend to the emergent sonic composition created by the pair of DIADs. When either or both players entered into a more stable periodic movement, the sound from both balls became responsive and transformed their behavior. Steady movements produced a rich,

harmonious, regular pattern as both balls began to synchronize with each other. The periodicity of the balls was not directly controlled by the players, but was influenced by them, with the resulting sound and behavior of each ball dependent on the combined state of both balls. The work was presented at ACM CHI PLAY conference in Melbourne, Australia, in October 2018.

Interactive Installation

Spiral is a mechanical and distributed performance that captured the performance style of the improvisational music ensemble Tangents, shown at the Museum of Applied Arts and Sciences in Sydney, Australia, in March 2019 (https://sydneydesign .com.au/2019/event/spiral). The installation is inspired by the concept of the traditional player piano that mechanically plays back a prepunched piano-roll recording though a real piano. Similarly, the simulated ensemble's piano, virtual bass, drums, and virtual guitar perform autonomously through a Disklavier, Ableton Live, MIDI-controlled electromechanical percussion, and 25 Raspberry Pis hanging in a spiral formation from the ceiling. The system, shown in Figure 6, is controlled through Ableton Live, which coordinates all the instruments through MIDI and OSC messages. Also, the audience is able to interact with the installation through a smartphone application.

HappyBrackets is used in the installation to allow the 25 Raspberry Pi units to function as tightly synchronized sample players to render the guitar. The devices are networked and tightly synchronized, allowing the sample playback to follow the spiral formation of the devices. The HappyBrackets environment facilitated rapid iterative refinement of the composition, as all the devices could be monitored and updated simultaneously.

Planetarium Software Performance

The final work we present is a multimedia sky- and soundscape based on the flag of Brazil. The work uses network-controlled planetarium software combined with an online astronomical database to obtain scientific data about the stars being displayed. This astronomical data is mapped to musical, political, geographical, and ethnological parameters, creating a virtual tour of Brazil. (A recent realization of the work as an installation, shown at the Convergence Conference at De Montfort University, Leicester, UK, in September 2019, can be seen at https://vimeo.com/355694234.). The composition and performance of the work was made almost completely within the HappyBrackets environment. Moreover, the creation and development of libraries to interface the planetarium and online astronomical database was undertaken completely within the HappyBrackets environment without requiring the composer to use other development tools. This section details the creative process used within HappyBrackets to rapidly develop, compose, and realize the work.

The concept of controlling remote planetarium software for a composition was inspired by a paradigm where astronomical data obtained from online astronomical databases was used as a stimulus for live musicians in an earlier performance, and as input to a musical automaton on the field night of an astronomical society (Fraietta 2014). The celestial positions of stars were determined by sensors on high-power binoculars that the audience participants used during the performance. One significant limitation of that earlier work was that the success of a performance was subject to the weather conditions, in particular, whether cloud obscured the night sky. One of the authors decided that a performance using a planetarium display instead of binoculars as the visual and control stimulus could be extremely effective for live performance.

The first stage in testing and implementing a control for the Stellarium planetarium software was by developing a library in HappyBrackets. This was accomplished by simply writing a HappyBrackets composition to connect to the Stellarium planetarium software through its representational state transfer interface on the local-area network and sending control messages via DynamicControls inside the composition (Fraietta and Bown 2019). When this proved successful, another HappyBrackets composition was created with an accelerometer and a gyroscope, using DynamicControls with the DEVICE ControlScope to send IMU values to the first composition. After simulating the IMU sensors, the composition was loaded into a HappyBrackets-powered sonic poi, allowing control of the planetarium by physically manipulating the poi. This composition was effectively the beginning of a library for controlling Stellarium (Fraietta 2019b). The library was iteratively developed inside a HappyBrackets composition because it took less than one second to send and run new versions to the device. Having to switch between different IDEs would have significantly increased the amount of time to develop the library.

The composer then developed an interactive spacecraft game for users at a university open day. The goal of the game was for players to choose an astronomical object on the planetarium display—for example, a planet or star—and virtually fly towards that object with the sonic poi. Players were able to zoom in and out by using the gyroscope, and to lift or lower their virtual head (i.e., the field of vision) by using the accelerometer. This enabled players to navigate to planets, moons, stars, and galaxies on the display. The sonic poi–generated sound that was indicative of a player's field of view and provided audible feedback when the player zoomed in or out. By observing the player's actions and comparing the actual display and audio to what we expected, we were able to easily note problems,

Figure 7. Planetarium display showing scientific and political information.
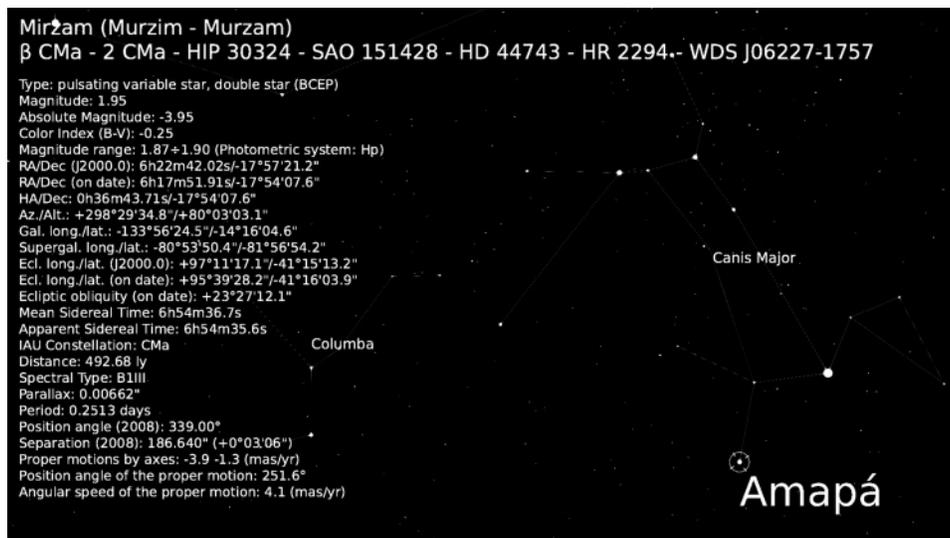
make modifications where necessary, and upload the updated code instantaneously (Fraietta and Bown 2019).

The research was further extended through the development of a standalone library for controlling Stellarium, coupled with a facility to retrieve astronomical data from online catalogs (Fraietta 2019b). This data would be used as input for compositional stimulus and for live performance. Fraietta also composed a work with a subject that would celebrate the theme and location of the conference where the research would be presented. He composed a multimedia, virtual astronomical tour of Brazil by abstracting the skyscape represented on the nation's flag (Fraietta 2019a). The flag of Brazil displays 27 stars from nine constellations as they would have appeared in the sky of Rio de Janeiro at 8:30 a.m. on 15 November 1889, with each star representing a specific state (Duarte 2010). The performer sequentially selects a different star on the flag using DynamicControls within HappyBrackets, which displays both scientific information and the name of the Brazilian state represented by the star, as shown in Figure 7. The work is in three movements, with HappyBrackets used differently in each movement.

The first movement, "Amerindia," primarily uses audio samples sourced from ethnomusicolog-ical recordings of the indigenous peoples of Brazil, coupled with spoken voices in indigenous languages (Fraietta 2019a). The movement is a sound-scape (Schafer 1993) that contains keynote sounds, created from climate sounds and indigenous music, to create an atmosphere of precolonial Brazil. Signal sounds were created by significantly manipulating indigenous music using granular synthesis and mapping the instrument pitch, playback rate, reverberation, and parameters for granular synthesis, all from astronomical parameters of the stars on the display. Testing various synthesis values during the composition phase was done using buddy DynamicControls, as shown in Figure 4. "Sound marks," which were spoken indigenous voices geographically related to the Brazilian state selected, were triggered when the performer clicked a DynamicControl button to index a table, which played the voice sample, moved Stellarium to the next star, and loaded the astronomical tables used as input to the signal sounds algorithm.

The second movement, "Cães Celestes," is composed of 64 sine-wave oscillators whose values were determined by the azimuth and altitude of the stars on display, mapped to frequency, rhythmic position, and panning. The altitude of each star was scaled between the minimum and maximum

Computer Music Journal

Figure 8. DynamicControls used to select from the indexed table of dances.
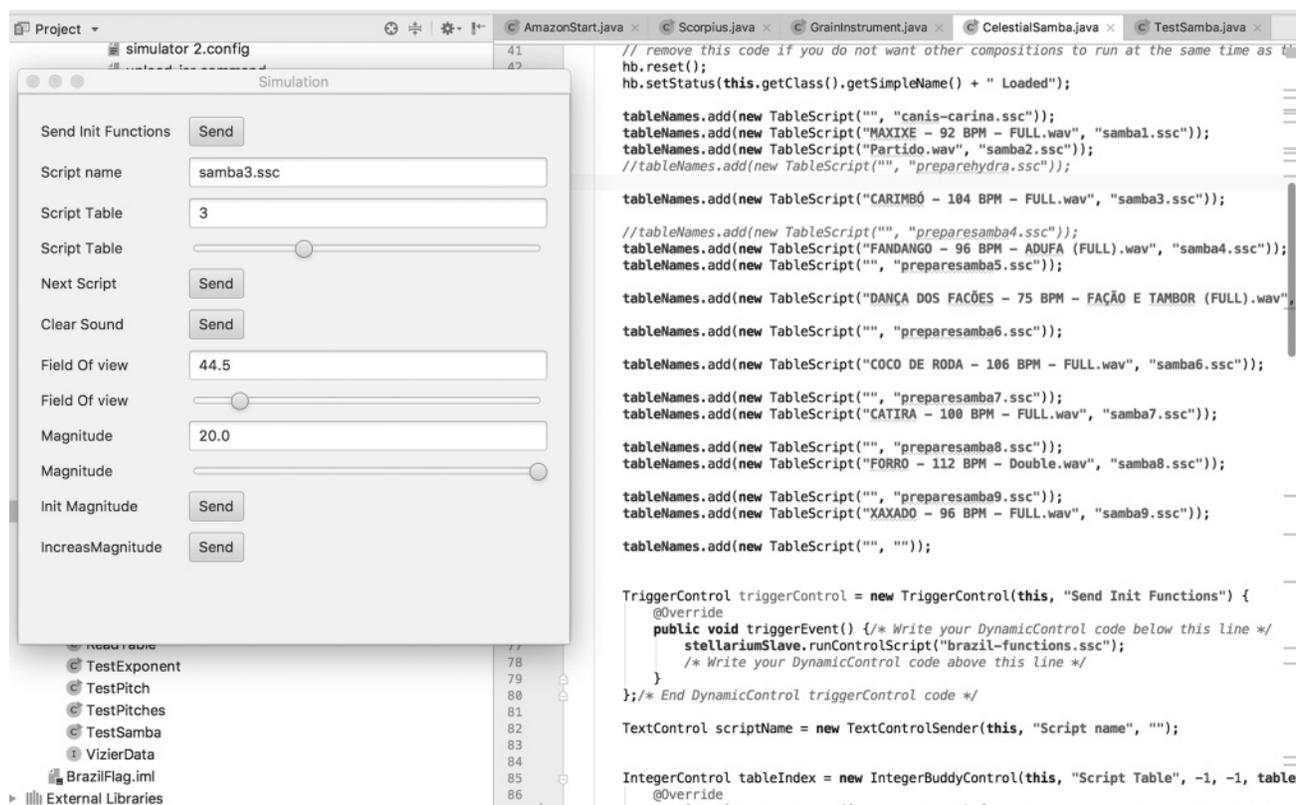
boundary pitches, and then quantized so the pitch slotted into the desired harmony. A significant amount of development of the Stellar Command library was performed at this time, requiring a large number of modifications, enhancements, and testing inside the composition.

The final movement, "Celestial Samba," was composed to include a live percussion ensemble for an intended performance in Brazil at the 2019 International Conference on New Interfaces for Musical Expression. Nine traditional Brazilian dances were supplied to the composer as prerecorded audio. A planetarium cadenza was composed in which the planetarium is controlled manually using DynamicControls in the HappyBrackets environment, through gyroscope and accelerometer controls in Raspberry Pi, and by triggering Stellarium to run precoded scripts (Zotti, Schaukowitsch, and Wimmer 2017). The scripts, written in JavaScript, control Stellarium internally through a series

of objects that represent Stellarium's internal application components Zotti and Wolf (2019). During composition, HappyBrackets was used as an indexed-playlist sample player that could be used to step through each section of the movement. Each index, as shown in Figure 8, contained the name of a dance file and the name of a Stellarium script to run. Using a DynamicControl to select an index caused a dance sample to commence playing while the associated script was simultaneously started, causing the name of the loaded script to be displayed in a text DynamicControl as feedback. These features facilitated rapid development of the Stellarium scripts as it was easy to start and stop any dance during the development of each Stellarium script.

HappyBrackets facilitated rapid composition in all sections of the work's development, as it allowed the composer to modify sonic parameters, build and refine a complex astronomical library, and develop

Fraietta et al.

JavaScript astronomy scripts without ever having to switch to another environment.

Conclusion

We developed the HappyBrackets system to integrate creative coding for embedded systems and general-purpose computers within an IoT domain. We leveraged the extensibility of features available with the Raspberry Pi and a wide range of sensors. Moreover, we exploited the Linux operating system by providing not only the ability to run third-party programs and execute scripts in other languages on the system, but also the facility to edit and update these scripts without leaving the HappyBrackets environment. Furthermore, the use of the Java language enabled loading of dynamic code on distributed devices, enabling code changes on remote devices without interrupting running code—a basic requirement for live coding. Moreover, the standard networking capabilities provided through Java and Linux enabled us to develop a system where the multiplicity of devices acting together in an installation was standard functionality rather than an extreme condition. We developed strategies that enabled users not only to write terse code and minimize coding errors, but also to obtain immediate feedback when developing compositions and extended libraries offline by running the code (either inside the IDE's debugger or in a simulator running virtual devices). We enabled developers to debug remote devices through the use of graphic DynamicControls that doubled as message interfaces between compositions and devices. All of these features were made available without ever requiring composers to switch between programming environments from conception to performance and diagnosis.

Finally, we presented art projects that demonstrated how HappyBrackets was used in improvisational dance with live coding, in an interactive distributed system containing 25 Raspberry Pis, and in a composition for live planetarium and percussion ensemble whereby complex IoT communication between planetarium software, online databases, and embedded systems was effected from conception to realization within the HappyBrackets environment.

References

Aaron, S., A. F. Blackwell, and P. Burnard. 2016. "The Development of Sonic Pi and Its Use in Educational Partnerships: Co-Creating Pedagogies for Learning Computer Programming." *Journal of Music, Technology and Education* 9(1):75–94.

Barrett, S. F., and J. Kridner. 2016. *Bad to the Bone: Crafting Electronic Systems with BeagleBone Black*. 2nd ed. San Rafael, California: Morgan and Claypool.

Berger, A. S. 2001. *Embedded Systems Design: An Introduction to Processes, Tools, and Techniques*. Boca Raton, Florida: CRC.

Bergstrom, I., and R. B. Lotto. 2015. "Code Bending: A New Creative Coding Practice." *Leonardo* 48(1):25–31.

Bown, O., and S. Ferguson. 2016. "A Musical Game of Bowls Using the DIADs." In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pp. 371–372.

Bown, O., and S. Ferguson. 2018a. "Creative Media + the Internet of Things = Media Multiplicities." *Leonardo* 51(1):53–54.

Bown, O., and S. Ferguson. 2018b. "Understanding Media Multiplicities." *Entertainment Computing* 25:62–70.

Bown, O., M. Young, and S. Johnson. 2013. "A Java-Based Remote Live Coding System for Controlling Multiple Raspberry Pi Units." In *Proceedings of the International Computer Music Conference*, pp. 31–38.

Bown, O., et al. 2015. "Distributed Interactive Audio Devices: Creative Strategies and Audience Responses to

Novel Musical Interaction Scenarios." In *Proceedings of the International Symposium on Electronic Art*, pp. 604–607.

Bown, O., et al. 2019. "Facilitating Creative Exploratory Search with Multiple Networked Audio Devices Using HappyBrackets." In *International Conference on New Interfaces for Musical Expression*, pp. 286–291.

Collins, N., et al. 2003. "Live Coding in Laptop Performance." *Organised Sound* 8(3):321–330.

Cook, P. 2017. "2001: Principles for Designing Computer Music Controllers." In *A NIME Reader*. Berlin: Springer, pp. 1–13.

Digumarti, T., et al. 2016. "Pixelbots 2014." In *Proceedings of the ACM SIGGRAPH 2016 Art Gallery*, pp. 366–367.

Duarte, P. A. 2010. "Astronomia na Bandeira Brasileira." https://web.archive.org/web/20080502120005/http://www.cfh.ufsc.br/~planetar/textos/astroban.htm. Accessed 21 December 2018.

Dyke, G. 2011. "Which Aspects of Novice Programmers' Usage of an IDE Predict Learning Outcomes?" In *Proceedings of the ACM Technical Symposium on Computer Science Education*, pp. 505–510.

Farrington, E. 2015. "Parametric Equations at the Circus: Trochoids and Poi Flowers." *College Mathematics Journal* 46(3):173–177.

Ferguson, S., and O. Bown. 2017. "Creative Coding for the Raspberry Pi Using the HappyBrackets Platform." In *Proceedings of the ACM SIGCHI Conference on Creativity and Cognition*, pp. 551–553.

Ferguson, S., et al. 2017. "Sound Design for a System of 1,000 Distributed Independent Audio-Visual Devices." In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pp. 245–250.

Fernández, M. 2007. "Illuminating Embodiment: Rafael Lozano-Hemmer's Relational Architectures." *Architectural Design* 77(4):78–87.

Fléty, E., and C. Maestracci. 2011. "Latency Improvement in Sensor Wireless Transmission Using IEEE 802.15.4." In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pp. 409–412.

Fraietta, A. 2005. "The Smart Controller Workbench." In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pp. 46–49.

Fraietta, A. 2006. "The Smart Controller: An Integrated Electronic Instrument for Real-Time Performance Using Programmable Logic Control." PhD dissertation, Western Sydney University.

Fraietta, A. 2008. "Open Sound Control: Constraints and Limitations." In *Proceedings of the International Conference on New Interfaces for Musical Expressions*, pp. 19–23.

Fraietta, A. 2014. "Musical Composition with Naked Eye and Binocular Astronomy." In *Proceedings of the Australasian Computer Music Conference*, p. 47.

Fraietta, A. 2019a. "Creating Order and Progress." In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pp. 83–88.

Fraietta, A. 2019b. "Stellar Command: A Planetarium-Based Cosmic Performance Interface." In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pp. 387–392.

Fraietta, A., and O. Bown. 2019. "Creating a Sonified Spacecraft Game Using HappyBrackets and Stellarium." In *Proceedings of the Linux Audio Conference*, pp. 1–7.

Gosling, J., B. Joy, and G. Steele. 2000. *The Java Language Specification*. Boston: Addison-Wesley.

Haeusler, M. 2009. *Media Facades: History, Technology, Content*. Stuttgart: Avedition.

Horstmann, C. S., and G. Cornell. 2002. *Core Java 2: Volume I, Fundamentals*. London: Pearson Education.

Hörtner, H., et al. 2012. "Spaxels, Pixels in Space: A Novel Mode of Spatial Display." In *Proceedings of the International Conference on Signal Processing and Multimedia Applications and Wireless Information Networks and Systems*, pp. 19–24.

Huber, D. M. 2012. *The MIDI Manual: A Practical Guide to MIDI in the Project Studio*. Waltham, MA: Focal.

Ishii, H., and B. Ullmer. 1997. "Tangible Bits: Towards Seamless Interfaces between People, Bits and Atoms." In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, pp. 234–241.

Keislar, D. 2009. "A Historical View of Computer Music Technology." In *The Oxford Handbook of Computer Music*. Oxford: Oxford University Press, pp. 11–43.

Levin, G., et al. 2001. "Dialtones (a Telesymphony): Final Report." Project report. Available online at www.flong.com/storage/pdf/reports/dialtones_report.pdf. Accessed October 2019.

Loke, L., et al. 2018. "Your Move Sounds So Predictable!" In *Proceedings of the Annual Symposium on Computer-Human Interaction in Play: Companion Extended Abstracts*, pp. 121–125.

Magnusson, T. 2011. "Algorithms as Scores: Coding Live Music." *Leonardo Music Journal* 21:19–23.

Magnusson, T. 2014. "Scoring with Code: Composing with Algorithmic Notation." *Organised Sound* 19(3):268–275.

Margolis, M. 2011. *Arduino Cookbook: Recipes to Begin, Expand, and Enhance Your Projects*. Sebastopol, California: O'Reilly.

McCurry, M. 2018. "RTOSC: Realtime Safe Open Sound Control Messaging." In *Proceedings of the Linux Audio Conference*, pp. 51–70.

McPherson, A., and V. Zappi. 2015. "An Environment for Submillisecond-Latency Audio and Sensor Processing on BeagleBone Black." In *Proceedings of the 138th Audio Engineering Society Convention*. Available online at www.aes.org/e-lib/browse.cfm?elib=17755 (subscription required). Accessed October 2019.

Merrill, D., J. Kalanithi, and P. Maes. 2007. "Siftables: Towards Sensor Network User Interfaces." In *Proceedings of the International Conference on Tangible and Embedded Interaction*, pp. 75–78.

Milne, I., and G. Rowe. 2002. "Difficulties in Learning and Teaching Programming: Views of Students and Tutors." *Education and Information Technologies* 7(1):55–66.

Miranda, E. R., and M. M. Wanderley. 2006. *New Digital Musical Instruments: Control and Interaction beyond the Keyboard*. Middleton, Wisconsin: A-R Editions.

Mitchell, M. C., and O. Bown. 2013. "Towards a Creativity Support Tool in Processing: Understanding the Needs of Creative Coders." In *Proceedings of the Australian Computer-Human Interaction Conference*, pp. 143–146.

Monk, S. 2016. *Raspberry Pi Cookbook: Software and Hardware Problems and Solutions*. Sebastopol, California: O'Reilly.

Quatrani, T. 2002. *Visual Modeling with Rational Rose 2002 and UML*. Chicago: Addison-Wesley.

Reas, C., and B. Fry. 2006. "Processing: Programming for the Media Arts." *AI and Society* 20(4):526–538.

Roberts, C., and J. Kuchera-Morin. 2012. "Gibber: Live Coding Audio in the Browser." In *Proceedings of the International Computer Music Conference*, pp. 64–69.

Rottondi, C., et al. 2016. "An Overview on Networked Music Performance Technologies." *IEEE Access* 4:8823–8843.

Schafer, R. M. 1993. *The Soundscape: Our Sonic Environment and the Tuning of the World*. Rochester, Vermont: Destiny.vadjust

Scheidl, H. 2014. "Library Overview." Available online at github.com/processing/processing/wiki/Library-Overview. Accessed 3 March 2019.

Tabanao, E. S., M. M. T. Rodrigo, and M. C. Jadud. 2011. "Predicting At-Risk Novice Java Programmers through the Analysis of Online Protocols." In *Proceedings of the International Workshop on Computing Education Research*, pp. 85–92.

Thompson, G. 2017. "Coding Comes of Age." *THE Journal* 44(1):28.

Tuomi, P., et al. 2018. "Coding Skills as a Success Factor for a Society." *Education and Information Technologies* 23(1):419–434.

Turchet, L., C. Fischione, and M. Barthet. 2017. "Towards the Internet of Musical Things." In *Proceedings of the Sound and Music Computing Conference*, pp. 13–20.

Turchet, L., A. McPherson, and C. Fischione. 2016. "Smart Instruments: Towards an Ecosystem of Interoperable Devices Connecting Performers and Audiences." In *Proceedings of the Sound and Music Computing Conference*, pp. 498–505.

Turchet, L., et al. 2018. "Internet of Musical Things: Vision and Challenges." *IEEE Access* 6:61994–62017.

Vestergaard, L. S., J. Fernandes, and M. Presser. 2017. "Creative Coding within the Internet of Things." In *Proceedings of the Global Internet of Things Summit*. Available online at ieeexplore.ieee.org/document/8016223 (subscription required). Accessed October 2019.

Zhu, Q., et al. 2010. "IoT Gateway: Bridging Wireless Sensor Networks into Internet of Things." In *Proceedings of the IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, pp. 347–352.

Zotti, G., F. Schaukowitsch, and M. Wimmer. 2017. "The Skyscape Planetarium." *Culture and Cosmos* 21(1):269–281.

Zotti, G., and A. Wolf. 2019. "Stellarium 0.19.0 User Guide." Technical report. Available online at github.com/Stellarium/stellarium/releases/download/v0.19.0/stellarium_user_guide-0.19.0-1.pdf. Accessed May 2019.