

Adding Music to App Inventor for Android

Thesis
Submitted in Partial Fulfillment
of the Requirements for

Degree of
Master of the Arts in Interdisciplinary Computer Science

Mills College
Fall 2012

By Trevor Adams

Advisor:
Prof. Ellen Spertus

Abstract

App Inventor for Android is a development environment allowing non-programmers to create their own applications for devices using Google's Android operating system. Tools are provided for creating many kinds of applications, but it is currently difficult to create interactive musical apps. I chose to address this issue by adding a component that would enable users to include limited real-time sound synthesis in their apps. Specifically, I designed an interface for playing notes by specifying pitch, volume and duration. I also added controls for adding reverb, envelopes and changing instruments. To support my component, I devised a system for managing libraries and assets, which allows native code to be packaged with an app, and ensures that any additional files are loaded only if necessary. This paper will discuss the motivation for my changes, the process of their design and the benefits they should provide to users and developers.

Table of Contents

[Title Page](#)

[Abstract](#)

[Table of Contents](#)

[I. Introduction](#)

[The Problem](#)

[Solution](#)

[Background](#)

[II. Instrument Component](#)

[Objective](#)

[The Component](#)

[Choice of Implementation](#)

[Design](#)

[Process](#)

[Closing](#)

[III. SuperCollider](#)

[Background](#)

[SuperCollider Core](#)

[SuperCollider-Android](#)

[App Inventor Integration](#)

[Process](#)

[Closing](#)

[IV. Selectively Loaded Files](#)

[Motivation](#)

[Functionality and Implementation](#)

[Process](#)

[Closing](#)

[V. Conclusion](#)

[Possible Extensions](#)

[Reflection and Suggestions](#)

[Final Thoughts](#)

[Bibliography](#)

I. Introduction

App Inventor for Android (AIA) is a web based tool that allows users without programming or development experience to create applications for Android mobile devices. Inheriting some qualities of older educational languages like Scratch and StarLogo TNG¹, App Inventor uses a “what-you-see-is-what-you-get” (WYSIWYG) interface that is extremely helpful to beginners.² Users use the App Inventor Designer (Figure 1) to precisely lay out the visual appearance of their app, then use the App Inventor Blocks Editor (Figure 2), a jigsaw-puzzle-type interface, to give the program functionality.

My first reaction to App Inventor was a mixture of admiration and annoyance. As electronic devices and software become more and more important to everyday life, App Inventor addresses an important concern for today’s society: the gradual opening of the world of programming to a wider populace. AIA allows people of all levels of experience to learn how to create software. I consider spreading knowledge of computing as one of the most important things we can do as programmers.

The inherent issue with tools like App Inventor is that users are limited to the features the tool designers have programmed for them. Certain types of apps may be impossible to create in AIA simply because the necessary components do not exist. Because of my background as a musician, I was struck with the difficulty that creating a music app in App Inventor would pose as it currently stood.

¹ Gibbs, Mark. “App Inventor, Scratch and Simple Programming.” *Network World*. 28 July 2010. Web. 2 Nov. 2012. <<http://www.networkworld.com/community/toolshed/app-inventor-scratch-and-simple-programmi>>.

² Siegler, MG. “Is Google App Inventor A Gateway Drug Or A Doomsday Device For Android?” *TechCrunch*. 11 July 2010. Web. 2 Nov. 2012. <<http://techcrunch.com/2010/07/11/google-app-inventor/>>.

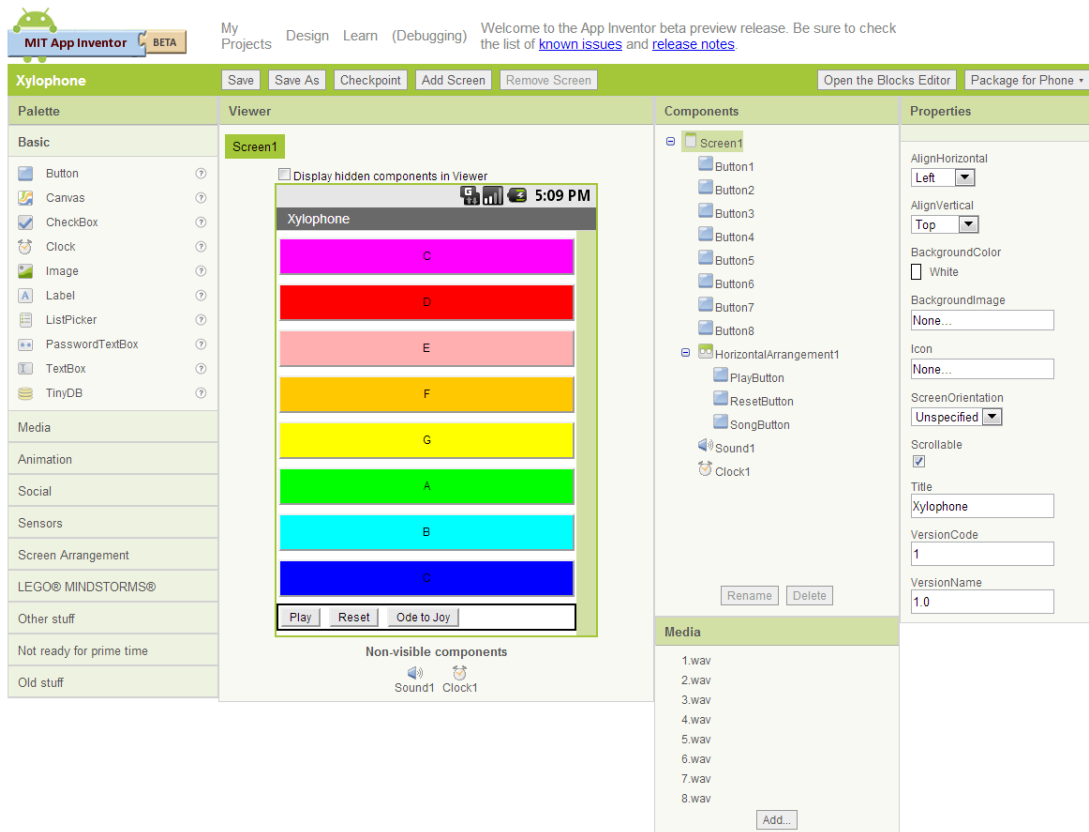


Figure 1 - App Inventor Designer

The Problem

App Inventor possesses some sound and music abilities already, but features that would be particularly useful in interactive music applications are lacking. All App Inventor's current musical abilities involve playing sound files the creator has uploaded to the phone. Control is limited to starting and stopping files; traits like volume and pitch are unmodifiable. AIA includes a timer which, with some math, can be used to play back sounds in rhythm (though it is notoriously inaccurate).³

³ "Issue 798 - MIDI command for playing internal MIDI sounds." *App Inventor for Android*. Google Code. 7 Nov. 2011. Web. 23 Nov. 2012. <<http://code.google.com/p/app-inventor-for-android/issues/detail?id=798&q=sound&sort=-stars&colspec=ID%20Status%20Summary%20Owner%20Reporter%20Stars>>.

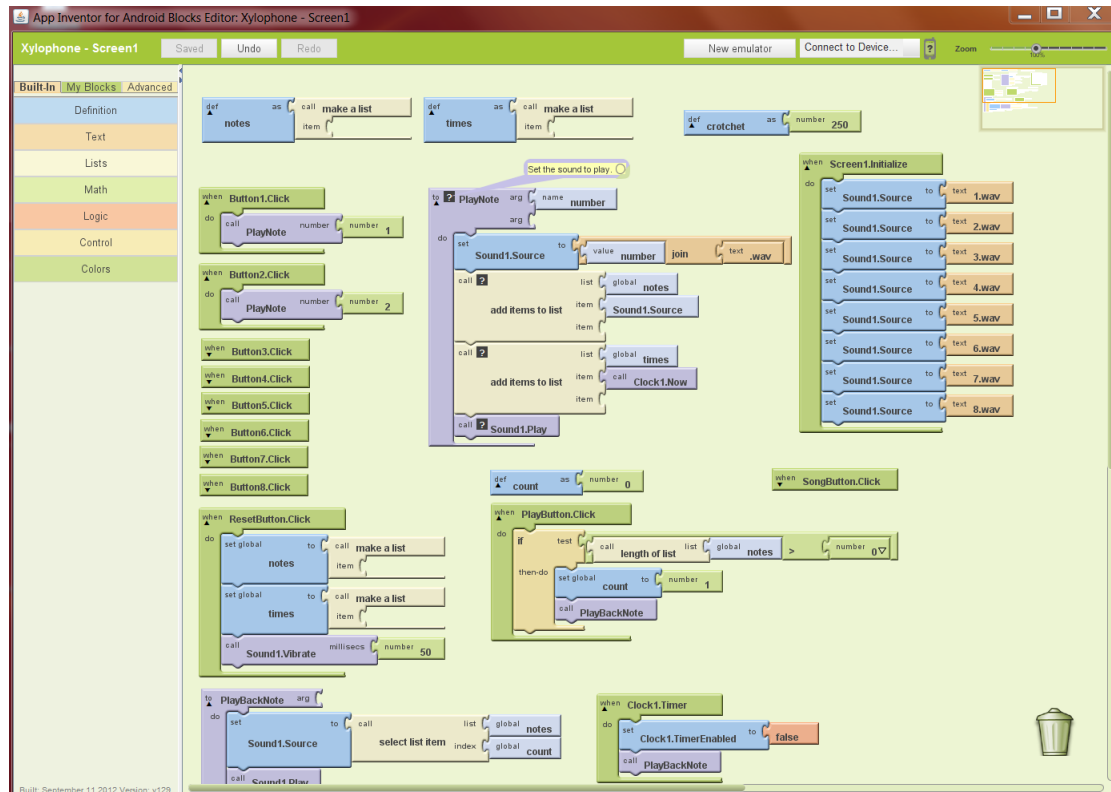


Figure 2 - App Inventor Blocks Editor

A great example of App Inventor's current sound capabilities (prior to my additions) is the "Xylophone" tutorial from *App Inventor: Create Your Own Android Apps*.⁴ A column of buttons represents the bars of a xylophone, and pressing a button will play a note (an octave of a C major scale is presented). The tutorial also shows how to record the notes played and their timings, so that a song or phrase can be played back.

The example app reveals some of the limitations of the current capabilities, the first of which is the requisite dependence on pre-recorded sound. Each note requires an individual sound file, which is acceptable (barely) for the 8 note xylophone app, but makes an 88-key keyboard unrealizable. Besides the awkwardness of procuring and uploading that many sounds, the file count would cause large application file sizes and long loading times, possibly preventing

⁴ Wolber, David, Hal Abelson, Ellen Spertus, and Liz Looney. *App Inventor: Create Your Own Android Apps*. O'Reilly Media, 2011. University of San Francisco Department of Computer Science. David Wolber. 23 Nov. 2012. <<http://cs.usfca.edu/~wolber/appinventor/bookSplits/ch9Xylophone.pdf>>.

the app from even running.

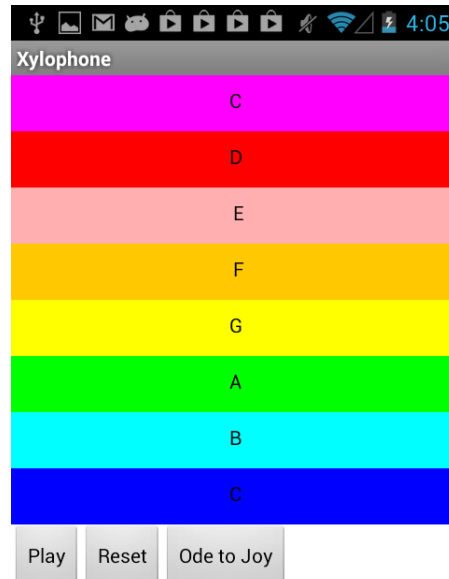


Figure 3 - Xylophone App

This implementation is also highly disruptive to musicality. A user wanting to change the velocity, length or timbre of a note would need to upload an entirely separate sound file (possibly having to record it themselves). Real music-making requires the flexibility to adjust these musical characteristics easily. Based on the small but persistent demand for similar additions on the AIA issues list,⁵ I believe the significant obstacles to producing musical sounds effectively eliminates music apps as a possibility for the normal user.

Solution

After considering several options, I decided to create a component that enabled musical synthesis. I wanted a simple, direct interface where users could name a note and have it played back. Each of the previous shortcomings (need for sound files, lack of flexibility) would

⁵ "App Inventor Issues List - Search for 'sound'." *App Inventor for Android*. Google Code. Various dates. Web. 23 Nov. 2012. <<http://code.google.com/p/app-inventor-for-android/issues/list?can=2&q=sound&sort=-stars&colspec=ID%20Status%20Summary%20Owner%20Reporter%20Stars>>. Issues 798, 1260, 1289, 780.

disappear if users had a synthesis engine at their fingertips. The next chapter will deal with the design and creation of this component.

There are also ways the project could benefit the App Inventor developer community besides the component itself. Because I include a complete synthesis library (SuperCollider) with my component, future developers will be able to easily make their own musical components. In addition, the synthesis library is the first library to be added to App Inventor that is written in the C programming language. My code will make adding future native libraries to AIA simple and convenient. Chapter Three will discuss SuperCollider and its inclusion.

The component also gave me reason to fix a long-standing issue with App Inventor. Previously, all libraries and additional resources were being loaded into every designed app's APK (Application Package Files, the file containing an Android application), regardless of whether they were needed in that particular app. This was already occurring with the Twitter library. This would also include SuperCollider, which is fairly large. My component by definition is fairly specialized, and would only be needed in specific kinds of apps; as such, its large size might have made it less likely to be added to the main App Inventor branch. It was in my best interests to find a way to load only files or libraries that were needed by an app's components. Implementing this feature will also benefit the App Inventor community at large by reducing the size of current apps and making library additions by future components more practical. The library part of my project will be explored in Chapter Four, followed by a reflection on the whole project.

Background

The primary building blocks of every App Inventor application are Components. Creators use the Designer view to place components in their app's layout. Some components are straightforward visible objects (buttons or images), while others function behind the scenes

(Sound). Each component has Properties, which adjust their behavior and appearance (the color and text of a button). The actions of components (such as making a button play a sound) are configured separately in the Block Editor. Because App Inventor is intended for beginners, intuitiveness and ease of use are important considerations for both editors.

Sound synthesis is the method of creating sound using electronics, in contrast with normal acoustic instruments. There are many techniques for generating the sound to be produced, from samples (recording short audio clips to be played back on cue) to combining basic sound waves (sine, sawtooth, square, and others). A synthesizer (“synth”) is an instrument which uses one of these techniques to make sounds.

SuperCollider is “an environment and programming language for real time audio synthesis and algorithmic composition” first developed in 1996. It has since gained a following from scientists and musicians as one of the most versatile tools for music-making of all kinds of electronic music making.⁶ It is licensed under the GPL (GNU General Public License), and has been partially ported to Android. The library is the basis of all sound synthesis in the component.

⁶ “SuperCollider - About.” *SuperCollider.Sourceforge*. n.p., n.d. Web. 10 Nov. 2012.
<<http://supercollider.sourceforge.net/>>.

II. Instrument Component

As mentioned in the introduction, the current sound capability of App Inventor for Android is inadequate for most music applications. I set out to determine a way to introduce music to AIA within the scope of what a single person could accomplish. The addition would need to be easy to use, but be robust and flexible enough to enable as wide a variety of apps as possible. My solution was the Instrument component, a simple interface for specifying particular notes to be played. Ideally, users could use Instrument to create applications ranging from interactive instruments to precomposed songs.

Objective

The goal for my component was to broadly enable as many previously unrealizable music features as possible. The most obvious app someone might want to make would be an interactive instrument, like the Xylophone app shown previously. Inputs other than buttons could be used (the position of a touch on the screen, the accelerometer or orientation sensor...), but they all would produce sounds based on that input. Another possible application would be a sequencer for playing back songs. The application designer could write a song to play back in the app, or provide an interface so the app user could write their own.

These possible applications suggested many possible improvements (an improved timer for syncing playback, or specialized inputs), but producing notes was the feature common to all of them. As a starting point, I thought back to what frustrated me with my initial experiments with App Inventor and music; the aforementioned issues with the xylophone app provided concrete problems to solve. Users would have to be able to produce musical notes without having to supply individual sound files for each. I wanted to reduce the effort required to get a music app

up and running; I would also have to make the interface as intuitive as possible. My two main concerns were now getting notes out of an app and making it easy to do so.

The Component

The Instrument component was my solution for how to add an easy-to-use note-playing interface. Its purpose is to allow users to play musical notes by specifying pitch, volume and duration. Like the “Sound” media component, Instrument is invisible in the Designer and is used primarily from the Blocks Editor. The user sets a number of passive properties of the instrument, such as the sound source, and then calls the component’s Play() method. Play() takes a list as an argument containing the values for pitch, volume and duration. The note is then sent to the SuperCollider server to be played.

Instrument

A musical instrument component that will play notes of the specified pitch, duration and volume. Duration is given in milliseconds and volume is 0 to 100. Pitch can be given as either the frequency (in Hertz), or as the letter and accidental (# or b) of the note, followed by a number specifying the octave.

Properties

Attack

The attack: how long it takes for the instrument to reach peak volume, in milliseconds. -1 uses the instrument's default.

Decay

The decay: how long it takes for the instrument to fall from peak volume, in milliseconds. -1 uses the instrument's default.

Release

The release: how long it takes for the instrument to go silent after finishing playing, in milliseconds. -1 uses the instrument's default.

Reverb

The wetness of the reverb effect, as a percentage.

Source

The name of the type of instrument to play. The available options are "sine", "pulse", "saw", "triangle", and "noise".

Sustain

The sustain: how loud the sound will be after the decay, as a percentage of the peak volume. -1 uses the instrument's default.

Events

none

Methods

Play(list list)

Plays a note of the specified pitch, duration and volume.

Figure 4 - Instrument component reference documentation

The core of the component lies in the Source property and the Play() method. After setting the Source to the name of the type of instrument to play (“sine” or “trumpet”; a drop down menu lists the options in the Designer), calling Play() plays a note on the instrument. The current number of instrument types is small, limited to simple electronic sounds, but the infrastructure to add more is in place. Like making colors, Play() takes a list as an argument, which specifies the pitch, length and volume of the note. As a result, a song can be created and stored as a list of lists.

The component has six properties that are all settable and gettable in the Blocks Editor. Normally, all properties would be visible in the Designer as well, but most of the properties are intended for advanced users, and are hidden to avoid confusion. Source is the most important property, and is the only one to appear in the Designer.

The remaining properties of Instrument are non-essential characteristics that give users with electronic music experience more control. Reverb is a special effect that simulates the echoes of a physical space. The four “ADSR” properties (attack, decay, sustain and release) explicitly define the shape of the note’s volume.

The advantages of the Instrument component over previous techniques can be shown by re-examining the Xylophone app. The sound files have been replaced with calls to Instrument. New features such as changing the instrument type and octave can now be added easily. Further such enhancements to volume and duration are also possible. This is in stark contrast to the inflexible previous capabilities.

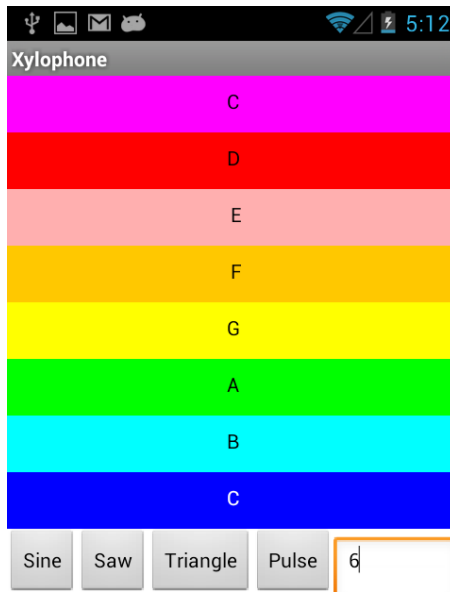


Figure 5 - New Xylophone App using Instrument component. The buttons change the instrument and the text field controls the octave of the note.

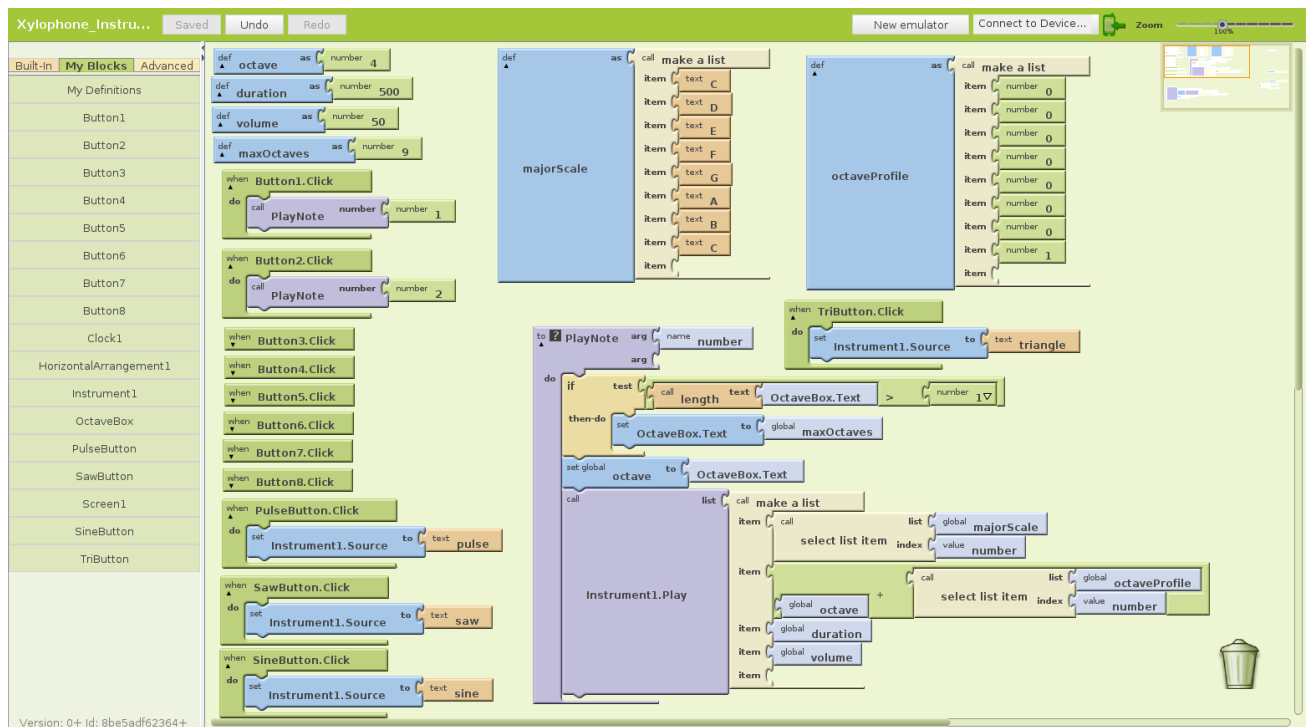


Figure 6 - Code for the new Xylophone App using the Instrument component

Choice of Implementation

After choosing my objectives for the component, I had to decide the best way to meet those goals. This involved both locking in the particular features I would support and choosing how to implement those features; there were several options for both choices.

MIDI

The first and most obvious route to me was to add MIDI capabilities to App Inventor. The MIDI (Musical Instrument Digital Interface) protocol is the current standard for issuing messages intended to be translated into musical sounds. Furthermore, the General MIDI (GM) specification provides a strict set of descriptions of instrument types and settings, and allows music to be created consistently across GM compliant devices (including Android phones). This includes both sending note messages to be played on the fly and creating .mid files containing notes and timings to play back later. MIDI is almost thirty years old, and other protocols have tried to replace it, but MIDI is still ubiquitous.

Android supports the playback of MIDI, so it initially looked like a promising choice. Unfortunately, there is no way to play individual MIDI notes in Android. The interface for doing so is not exposed to end users. MIDI information can only be played back from the non-interactive files (not counting JetPlayer, which features extremely limited playtime changes designed for games). There are various workarounds,⁷ but I worried that my options would be limited. The Android platform also has inherent sound latency issues which would likely hinder the potential fixes.⁸

⁷ peterdk. "The state of MIDI support on Android." *Umito.nl*. Umito, 7 May 2010. Web. 23 Nov. 2012.
<<http://blog.umito.nl/2010/05/07/midi-on-android.html>>

⁸ "Support for real-time low latency audio; synchronous play and record [UPDATED]." *Android Annoyances*. Wordpress, n.d. Web. 26 Nov. 2012.
<<http://www.androidannoyances.com/post/tag/low-latency-audio>>.

SoundPool and Audiotrack

Precomposed MIDI playback was still a possibility, but I considered the loss of any interactive ability too great; an alternate route was to extend the existing Sound component, which was based on Android's SoundPool class. In theory, App Inventor users would use the Sound component as before, but would find additional music related properties and features with it. For instance, a sound could have its playback speed altered at runtime, changing its pitch. The xylophone app could be made by uploading only a single note and then pitch-shifting it to create the others. Additional properties could be made for adding effects like reverb or distortion to sounds.

SoundPool supports real-time playback speed modulation, and looked like a good choice for giving users more musical flexibility. There were a few shortcomings: SoundPool could not increase or decrease speed by more than double, and it was unclear whether to present speed change musically or factually ("double speed of clip" vs. "raise clip by an octave"). As a solution, it was still somewhat brute force, rather than true music integration. Most importantly, besides looping, playback speed was the only SoundPool feature that remained unexplored. We would not be able to add any of the other features I desired (reverb, filters).

Alternatively, the AudioTrack class *did* support audio effects, but was not without its own complications. First and foremost was that its use would require the Sound component to be rewritten to use AudioTrack instead of SoundPool. AudioTrack is the lowest level sound API on the Android; the additional features reputedly add a good deal of complexity. I was under the impression that AudioTrack could not alter playback rate, though a check of the Android API shows this to be false.⁹

⁹ For more information about the Android audio API's: <http://www.wiseandroid.com/post/2010/07/13/Intro-to-the-three-Android-Audio-APIs.aspx>

Libraries and Pre-Existing Code

Unsatisfied with the inflexibility of the previously mentioned approaches, I began looking for an Android audio library capable of executing my desired features. My main goal was direct music production, and as I was no longer trying to adapt a pre-existing system to my needs, I particularly looked for synthesis libraries. There was not a large selection, however. Because of the issues hampering music apps on Android, few music libraries exist. One of the more notable Android apps is *Ethereal Dialpad*, an interactive synth with visual effects, which was programmed entirely from scratch due to the lack of appropriate libraries.¹⁰ While the app was open source (and I investigated the code), a library would (in theory) significantly increase my enhancement options, and give me more time to spend on adding features.

After doing a comprehensive search of various Android applicable sound libraries, and consulting my music advisor Prof. Chris Brown, I decided to use the SuperCollider audio engine to produce our sound. Earlier investigation had targeted a Java library named JSyn, developed by Phil Burk, a Mills College alumnus. Unfortunately, JSyn has not yet been adapted to Android. Professor Brown uses SuperCollider for most of his own work, and encouraged its use after I mentioned the sound engine had been ported to Android.

Though we would lack some of the finer features of the SuperCollider programming language, the SuperCollider server is a full powered synthesis tool which would allow more options than I could conceivably fit into a single component. Through discussion we divided my ideas into two distinct concepts for a component: an advanced component with patching capabilities resembling a modular synthesizer, and a simpler, user friendly component with an emphasis on playing individual notes.

¹⁰ Kirn, Peter. "Ethereal Dialpad Touch App, Development Experience on Android and Beyond." *Create Digital Music*. Create Digital Music, 5 May 2010. Web. 23 Nov. 2012. <<http://createdigitalmusic.com/2010/05/ethereal-dialpad-touch-app-development-experience-on-android-and-beyond/>>.

Because of my electronic music background and personal experiences, my mind had immediately gone towards modular synthesis for the capabilities to add. This involves connecting specialized modules such as sound sources and effects generators in order to construct a “patch” (a specific, reusable synthesizer configuration). One of my first electronic music experiences was exploring the combinations of sounds I could create on an ARP 2600 analog synthesizer. In my mind, this sort of exploratory music making was perfect for Android and App Inventor. Using fairly basic building blocks, users would be able to construct complex and interesting instruments themselves. Chris Brown particularly encouraged this idea.

My advisor, Prof. Ellen Spertus thought a simpler component would be more in keeping with the spirit of App Inventor. The concept was a bit complex and difficult to explain to those not already familiar with synthesis. Most users might not touch the more complex features, and the modular design would make their desired actions (simply playing an individual note) more difficult. The suggested simple component would have some secondary properties that could be set, but would primarily operate by a play method accepting pitch, duration and volume. In addition to being simpler, this component idea better fulfilled my original goal for App Inventor: making music-making accessible to all App Inventor users.

I eventually decided that the simpler component would be a better first addition to App Inventor. I began designing the component. I made sure, however, that the proposal stated that it was one of two planned components. I would be working on the “Instrument” component, but was designing it with the intention that a similar, more advanced component, “Synthesizer”, could be made to complement Instrument.¹¹ I would add Synthesizer myself if implementation turned out to be simple (it did not).

Design

¹¹ See Chapter V for more about the Synthesizer component.

After deciding on the implementation of the component itself, much of its interface fell into place. The main concern was making the component accessible not only to non-programmers, but to those without extensive musical knowledge as well. Spertus and I decided that the core functionality of the component would be contained in the Play() method. To play a note, a user needs only to select an instrument (the Source property) and issue a play command with the note's pitch (and duration and volume, optionally). The other component properties provided more control for intrepid users. I first wanted all properties to be visible from the Designer (users might not realize they were there), but was eventually convinced it would intimidate casual users.

Deciding what inputs the properties would accept seemed minor but was surprisingly important. Spertus' first suggestion was to model the Play() method on the "make color" library block, taking a list of arguments rather than taking the values themselves. This allows notes to be stored as lists of lists, facilitating playback of particular tunes. It also provides a novel solution to another interface issue, how to specify the pitch of a note.

I first described the pitch of a note being specified by its frequency, in Hz, but soon

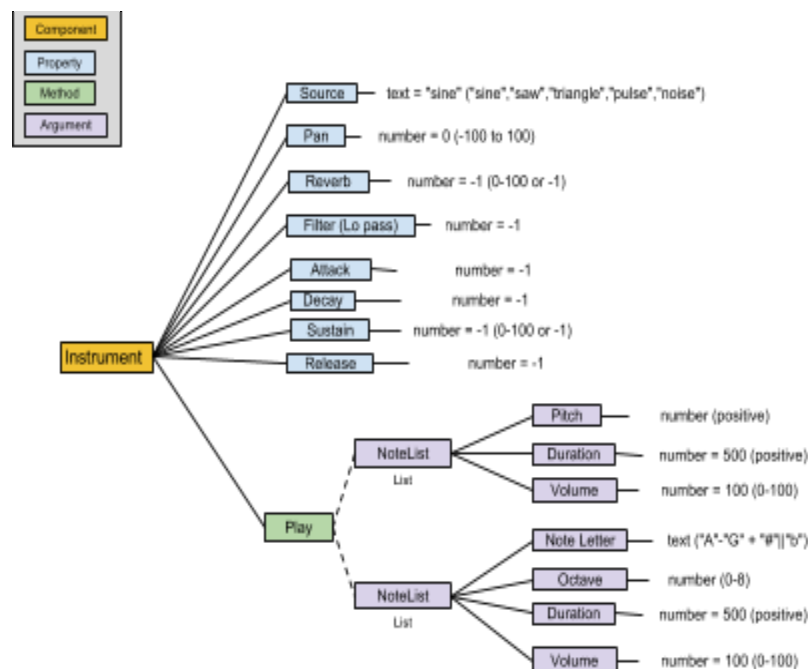


Figure 7 - Design diagram for the Instrument component, from my proposal

realized that this would severely limit its musical usefulness. Users needed to be able to specify standard notes on a piano keyboard (preferably by their note letter and accidental), though I didn't want to lose the ability to specify frequency. I imagined the feature being used for sound effects, tuning apps or other specialized contexts; it seemed a waste to hobble users' access to SuperCollider over such a minor issue. In retrospect, it might have been better to put this feature in a future update due to its niche appeal (the API would be the same). Implementing the feature was not particularly challenging, but gave rise to several significant bugs which needed to be fixed. For example, apps would crash when setting the frequency from a textbox, as the textbox content is a String, and it assumed it was receiving a note-letter format list.

Presentation of the other properties' values also raised a minor issue; there was no precedent for what units Volume, Pan and other settings should use (Sound has no volume property). Though it seemed trivial to be spending thought on, I recalled an anecdote Spertus told about how they considered their use of milliseconds instead of seconds as a design mistake. An inexperienced user is likely to assume seconds are used, and end up with infinitesimal time periods. The mistake could not be corrected because it would break existing apps.¹² In this case, I received comments from other developers that the choice of units was not an important decision. I ended up deciding on a 0 to 100 scale, as advanced users would likely accept it and it might be more intuitive to beginners.

Process

The primary concept of the component is one that evolved over a long period of time, between numerous technical setbacks and difficulties. We initially decided that I should start with a small modification to App Inventor in order to become more familiar with its workings.

¹² Spertus, Ellen. "How to Add a Component - Internal." *Google Docs*. 16 Oct. 2012. Web. 23 Nov. 2012. <https://docs.google.com/document/d/1I9qLSllxFgootg-IYSwEEqp3yjpoN_u0yxPAhq0aB94/edit#heading=h.gairsbwyrml>

Spertus, suggested tackling a minor bug in which sound files specified programmatically (using the Blocks Editor) would not be loaded the first time an application attempted to play them.

Getting started was less trivial than I had expected. I immediately found myself unable to build the (unmodified) App Inventor code on my 64 bit Windows 7 laptop. I would receive two different Ant errors depending on circumstances, both of which were ambiguous as to cause. After trying different versions of Java, I received word from the other developers that building was not supported on Windows. MIT undergraduate Weihua Li helpfully walked me through the process of setting up an Ubuntu virtual machine using VMWare Player, with a number of suggestions as to versions of different programs to install (Eclipse, OpenJDK, etc.). I then found myself having to choose between installing 32 or 64 bit Ubuntu. Not knowing any reasons to pick one or the other, I picked 64 bit.

After reinstalling the tools I needed in the new environment, I was surprised to find that App Inventor still would not build (though the error message had changed). Now, a temporary executable file generated by the buildserver could not be found, halting the build after ten minutes. The long build time seemed to be caused by sharing the code folder between my host and guest operating systems, but the error itself was harder to track down. After a good deal of searching and asking the list, I tracked down a forum post¹³ implying it might have something to do with a lack of proper 32 bit installation libraries (as it was a 64 bit OS). Sure enough, installing these libraries instantly solved the problem. Three weeks after beginning the project, I was finally able to build the core code of App Inventor.

I proceeded to work on the starter project (preloading sounds), hoping to get through it quickly and begin work on the component proper. To accomplish this, however, I had to run the local buildserver, which (of course) gave an error. Of several error messages I received, the

¹³ "Error: bash ./install: No such file or directory when installing Installation Manager on Ubuntu 64 10.04 LTS (64-bit)." *IBM Support*. IBM, 25 June 2010. Web. 23 Nov. 2012.
<<https://www-304.ibm.com/support/docview.wss?uid=swg21438771>>.

most problematic was “java.lang.RuntimeException: Unable to restore the previous TimeZone,” having no easily discernible cause or solution. The internet provided an easy solution for the error,¹⁴ but it did not work for me. On a whim, I eventually tried moving the code to a 32-bit Ubuntu VM, after which it worked perfectly. Despite having made some progress on my starter project (ensuring sounds load), I decided (because of the delays) to move on in order to begin on my goal project.

Synthdefs

An aspect of the component’s implementation that deserves special mention is the synth definitions. SuperCollider stores the blueprints for particular synthesizers (electronic instruments) in files called synth definitions (“synthdefs”) which can later be instantiated with varying arguments. This functionality allowed me to create synths with the better interface of my computer, then upload them to the Android device with SuperCollider.

Including synthdefs with App Inventor was not particularly difficult, but it did require a number of decisions to be made about their execution. Most important and visible of these was the matter of what synths to include. Ideally, we would like to have a large number of the instrument types a casual user would think of: standard orchestral instruments, keyboard and guitar, some interesting synthesized sounds and perhaps some sound effects. Because I was working with additive/subtractive synthesis (as opposed to sample-based), SuperCollider needed only the miniscule synthdef (less than 1 kb) to create each instrument. The total number of instruments was limited more by user overload than by size.

I would have to develop the synths myself, though I had no previous experience with *sclang* (SuperCollider’s programming language). I expected SuperCollider synthdefs and code

¹⁴ “TimeZone.class.getDeclaredField(“defaultZoneTL”) does not exist in JDK 6.” *Google App Engine Issues*. Google, 9 May 2012. Web. 23 Nov. 2012.
<<http://code.google.com/p/googleappengine/issues/detail?id=6928>>.

examples to be available online, and while they were, the volume of entries was less than I expected, and most snippets would require adaptation to work in App Inventor. Also, many of the extensions made to SuperCollider appear to be in the form of “plugins” in C rather than supplemental synthdefs. Such plugins could not be added without altering the SC-A library.

In the end, I decided to start with the basic oscillators used in electronic music (sine, saw, pulse and white noise). It was more important to take on the immediate challenges of the component and worry about the fairly trivial (not to mention more fun) job of adding extra synthdefs later.

Synthdefs also required some other minor implementation details to be worked out. Intending to preserve the possibility of a second SuperCollider component (Synthesizer), I anticipated problems might arise when the two components both required synthdefs named “sine”. I decided that all synthdefs would have a suffix attached to their filenames: “-inst” for Instrument defs and “-synth” for Synthesizer. I don’t believe I consulted the community or my committee on this decision, and I now consider it to have been a mistake.

At the time of my decision, all synthdef files (.scsyndef) were placed in the Assets folder of the application, making differentiation necessary. However, as Spertus later pointed out, this posed a risk of overwriting user assets (though the chance of a user uploading a .scsyndef file seemed small), and I ended up developing a method for placing assets in subfolders. The same method could easily be used to separate the synthdef files. Instead, I had unnecessarily introduced a convoluted naming convention. Furthermore, though I still support the concept of Synthesizer, the magnitude to which this project has grown precludes my proceeding with that work personally in the near future.

Closing

Looking back, I was amazed at how much work was needed to complete the

component's design; programming certainly isn't the only responsibility engineers have. The actual Java code governing Instrument was one of the simpler stages in bringing my component to life. Actually creating sound on an Android device proved to be perhaps the most challenging step on the way.

III. SuperCollider

As described in the previous chapter, I eventually decided to use the native library SuperCollider for the audio creation in the Instrument component. I reached this decision partially because of the lack of any Java synthesis libraries for Android. A relatively small number of Java libraries exist, but none of them worked on Android. A native code library seemed our best option, though including one with App Inventor would first require certain challenges to be met.

Background

The term “native library” refers to libraries written in languages that compile directly to platform-specific machine code (contrasting with languages compiled to portable byte code, such as Java). Though higher-level languages are preferable for a number of reasons, there are many situations that call for direct control of hardware. In particular, sound production is often associated with low level languages due to its reliance on hardware (speakers) and the need for precise timing. Higher level languages may include an interface for communicating with native code in order to access these options, or to gain access to existing native libraries. In Java, this is the Java Native Interface (JNI).

The JNI is used to call native code (C, C++, Assembly...) from Java code running on a Java Virtual Machine (JVM). The process signifies a significant jump in complexity; it is not a technique for writing a Java application in another language.¹⁵

Native code on the Android is executed through the JNI. An Android toolset called the

¹⁵ “Android NDK.” *Android Developers*. Google, Inc., n.d. Web. 8 Nov. 2012.
<<http://developer.android.com/tools/sdk/ndk/index.html>>.

Native Developer's Kit (NDK) is designed to help this process. Developers use the NDK to compile native code such that it will run on the Dalvik virtual machine (the Android analogue to JVM's). The NDK normally handles embedding the compiled files in a packaged application (.apk file) as well.¹⁶

SuperCollider Core

As stated in the Introduction, SuperCollider is a platform for synthesizing audio. SuperCollider consists of two main parts: the sound engine, *scsynth*, and the client side programming language, *sclang*. These can optionally be extended with OS-specific gui elements, but these were useful to me only during synthdef testing on the PC. While I could not use the SuperCollider language on the Android device itself (it was not ported), I found it useful when composing the synths that would be used, as well as for testing purposes.

The server, while intended to be used with *sclang*, was able to perform most essential functions via direct control. Tasks on the server can be carried out by sending it OSC messages (Open Sound Control, a proposed successor to MIDI). Two of the main tools for using the server are synths and ugens. Synths are the SuperCollider representation of a sound-producing node. They are instantiations of Synthdefs,¹⁷ templates written in *sclang* that define the sound produced. Ugens (short for Unit Generators) are “the basic building blocks of synths on the server, and are used to generate or process audio or control signals.”¹⁸ They signify shorthands for important audio functions like oscillators or audio effects, and make it easy to produce synthesized sounds without mathematically defining the audio data to be sent out.

¹⁶ “What is the NDK?” *Android Developers*. <<http://developer.android.com/tools/sdk/ndk/overview.html>>.

¹⁷ For more information see danielnouri.org/docs/SuperColliderHelp/ServerArchitecture/SynthDef.html.

¹⁸ “UGen.” *SuperCollider 3.2 Help Files*. SuperCollider, 28 Jan. 2011. Web. 8 Nov. 2012.
<danielnouri.org/docs/SuperColliderHelp/UGens/UGen.html>.

SuperCollider-Android

Professor Brown suggested investigating whether SuperCollider had been ported to Android; I was fortunate that it had been in 2010. Importantly, it appeared that work on the project stopped informally after the major milestone (getting the server running on a device), and not because all the project goals were complete. Several open issues are still listed on SCAndroid's Github page¹⁹, though no development has occurred for over two years. As a result, some features remain incomplete, though the essentials are there. I was concerned over the lack of activity in the SuperCollider Android Google Group forum²⁰ (one message every several months), but found that Dan Stowell, co-creator of the project, still regularly responds to questions posted there. He was instrumental in solving a number of my implementation issues.



Figure 8 - SC-Android explanatory diagram²¹

¹⁹ SuperCollider-Android. Github Inc, 12 Feb. 2011. Web. 28 Nov. 2012.

<<https://github.com/glastonbridge/SuperCollider-Android/wiki/>>.

²⁰ SuperCollider Android Developers. Google Groups. Web. 28 Nov. 2012.

<<https://groups.google.com/forum/#!forum/supercollider-android-developers>>.

²¹ SuperCollider-Android. <<https://github.com/glastonbridge/SuperCollider-Android/wiki/>>.

Besides the extensive features of SuperCollider itself, the biggest advantage to using the SCAndroid library was not having to call the JNI ourselves. SC-A has endeavored to handle all interfacing with the native code, allowing developers to call Java methods which will handle the JNI for them. This means one less thing I had to learn to get Instrument up and running. Because of the project's end, not all desirable features existed or worked correctly, but because they provided a method for sending arbitrary OSC messages to the server, most operations were still possible.

Some of the missing features in SCAndroid did actually limit the possibilities of what could be done in Instrument. The most significant alteration was the dropping of the Pan property. Pan refers to how sound is output on stereo speakers (left, right, or centered), and is a standard element of electronic music. Though it can be used to place different sounds in sonic space, its effect on small sound systems (like those on a smartphone) is minimal. I still imagined it might be audible, or that headphone users could take advantage of it, and so left it in the design.

Towards the end of the major coding on the component, Pan was one of the last features to be tested and fixed. While pan did not correctly work on apps, I knew that both Android and SuperCollider supported multichannel output, and so was confident there would be an easy fix. I traced the sound producing code into the library, and was surprised to discover that the number of output channels was hardcoded to one. Dan Stowell confirmed that the only way to produce stereo output would be to modify the library itself.

I decided that Pan was too minor a feature to justify using a non-standard version of SC-A. Besides obstructing possible future updates to the library, the library had never been tested for stereo output, and there was no guarantee it would work. It's difficult to say how much time I wasted on Pan: most of the work on it was merely setting up the property (like any other property), and was minimal. The revelation occurred so late in the project, though, that a

significant amount of time may have been spent thinking about it cumulatively.

Library limitations also meant we had access only to core SuperCollider features. The github project site mentions that most but not all core Ugens are supported in SC-A (though I never found any that were not, and no list is provided). More important were extensibility issues. As mentioned before, enhancements to SuperCollider usually occur in the form of C plugins, rather than additional synthdefs. As SC-A has no way of adding plugins, I could not use any internet examples that employed these plugins. This was a minor obstruction, of course, and mostly arose when looking for additional synths and effects to add to Instrument.

Other library quirks were primarily inconvenient, and not particularly limiting. SC-A expects callers to supply the data directory of their Android device, and the SC-A example project handles this with a hardcoded value (“/data/data/”). I did eventually find how to locate the data directory dynamically (a method had been added to the Android API since SC-A development had ceased), but the solution would not work with the way AIA currently builds.

Additionally, a bug prevents the arguments to a synth from being sent when it is created, forcing me to manually set their controls in separate messages. This forces me to temporarily track the ID’s of the synths (they are intended to be “fire and forget”), which was slightly more work to implement (though it did not affect performance).

App Inventor Integration

The problem of how to integrate SuperCollider-Android (a Java library interface to a native library) consisted of several stages. I first needed to learn how to make the library work with a normal Android application. After that, I would need to add the Java library to the libraries packaged with app inventor applications. Finally, I would have to learn how to do the same with the native library. Precedent existed for adding Java libraries, but I would be on my own adding the native code. The actual sending of messages from the Instrument component required

some attention as well. A parallel task was to program the synths to use in *sclang* and find a way to get them packaged with App Inventor applications.

Using SuperCollider in an application was a simple matter of learning to use the Android NDK. Native code is placed in an Android project directory sub folder (“jni”) with a number of make files (“.mk”) containing information on how to build the code (including what Android platforms to build it for). Running the “ndk-build” command on the project will compile the C code and place the resulting .so files in a “libs” folder of the project. Libs contains a subfolder for each architecture the code is targeted to. By default, there are two: ARM and ARM-NEON. At the time of writing, the NDK also supports x86 and MIPS devices. When the entire project is compiled, the .so files are automatically copied into a “lib” directory in the final .apk (note the lack of “s”). This is where Android will look by default for the native code when it is referenced in Java. So, running ndk-build was all I needed to do (for now) in order to get the normal Android App to use SuperCollider.

Attaching the Java libraries was also quite simple, and merely required finding the right instructions to follow. The hard work had been done for me, and Anshul Bhagi’s document on external jars²² was essential to adding the Java part of the library. App Inventor has a “libs” directory for storing external library files. Jars to be added must be placed in a subfolder of this directory, then copied to the runtime directory by the Ant build files. The class Compiler.java must then be modified to convert the library from Java .class files (for running on a JVM) to Dalvik compatible .dex files (using the dx tool).

Attaching the native libraries to the package was a much more difficult problem. Native code cannot simply be dexed like the Java files; I had to reproduce the process by which a conventional app packaged the code. I determined from the NDK documentation and inspection

²² Anshul Bhagi, “Adding App Inventor components that use External JARs.” *Google Docs*. Google Inc., 10 Nov. 2012. Web. 29 Nov. 2012. <<https://docs.google.com/document/d/1NyHU8pmQbqZclnxwTsG22f8bM8HtiqvVfCJp1AvsUsc/edit#heading=h.fyozwbk12ity>>

of sample .apk files that after compilation, the .so files just needed to be put in a folder in the .apk, “libs” (though all target architectures needed to be present). I began by following the external jars guide to get the files into the runtime directory. At this point, I could manually insert the files into the correct area of the .apk (libs) before it was packaged, similar to how it would be done in a normal app.

Process

There were a good number of obstacles to overcome on getting the functioning library into App Inventor. Getting a vanilla Android app to use SuperCollider took about week, though my only obstacle was learning to use the Android NDK at its most basic level. I repeatedly attempted to follow the guide on using SC-A in an app²³ without first explicitly following the guide for getting a development branch.²⁴ Though I must have examined the earlier guide (I installed the CrystaX NDK), I missed the all-important step of using it to build the C files! Dan Stowell pointed out the problem after I posted my predicament to the SC-A mailing list. I had no other problems, and was then successfully able to create an app “HelloSuperCollider”, which played a default synth (sine wave) when started. I was overjoyed, though quickly discovered my first audio bug when I closed the application and sound continued to play! (Sending the server a quit message when leaving an app corrected the problem).

I then quickly followed the guide to add the jar to AIA (though I was working on parts of the component itself simultaneously). Having gotten the SC-A Java library into AIA, I hoped I could somehow package the native files with the jar, and spent a day researching how one would do so. I eventually learned that while possible this was not the intended use of jars, and the file

²³ “Use SuperCollider in your Android-Activity.” <<https://github.com/glastonbridge/SuperCollider-Android/wiki/Use-SuperCollider-in-your-Android-Activity>>.

²⁴ “Get a Development Branch.” <<https://github.com/glastonbridge/SuperCollider-Android/wiki/get-a-development-branch>>.

would furthermore have to be extracted to the correct location at runtime anyway. I realized that connecting the native code the “normal” way would be more efficient, and read the NDK documentation²⁵ to find out what this was.

At about this time I made two major mistakes that, by their combination, would not become evident until considerably later. First, when adapting the SuperCollider code from my test app (HelloSuperCollider) to the Instrument component, I forgot to change the string that specified where to find the native libraries. After getting the libraries included in the generated .apk, I tested an app in App Inventor and found SuperCollider functioning. In fact, the files were being loaded from the HelloSuperCollider’s data directory (which was installed on my phone from earlier testing). This error wasn’t discovered until I tried to install a test app on Spertus’s phone to show her, and SuperCollider failed to load. Fortunately, simply changing the directory string constant fixed the problem.

This issue, however, caused me to overlook a second mistake. As I’ll mention in the next chapter, I was trying my best to minimize the space that my component and its resources required. There were over a dozen SuperCollider files, and as many of them seemed to be extensions to the SuperCollider core, I conjectured that some could be removed. I tested this by removing all but the two core files I knew were necessary to run, with the intention of adding files back until it worked. To my surprise, SuperCollider functioned without error using only the two files. I decided to keep the two file configuration until any problems arose. It didn’t hurt that this meant a lot less typing I had to do. Unfortunately, SuperCollider failed to break only because it was loading the files from elsewhere (HelloSuperCollider), and not the App Inventor apk. Fixing the loading location caused the libraries to break when they tried to find the plugins. Replacing these files fixed the issues.

²⁵ “Documentation.” <<http://developer.android.com/tools/sdk/ndk/index.html#Docs>>

Closing

Besides my issues getting AIA to work at all, interfacing with the SuperCollider library was by far the biggest technical hurdle I had to overcome for my component. Between the scant SC-A documentation and the complete lack of native code in App Inventor, I was not able to rely on others for many of my solutions. It is my hope that in addition to the music capabilities of my component, future component developers will be able to take advantage of my examples and add further native libraries in the future. Originally intending to facilitate this through writing a tutorial document, my goal shifted to include a technical enhancement that would help even more.

IV. Selectively Loaded Files

I learned very early in my project that application size was an important concern. My first lesson came when I followed the Xylophone tutorial and found the application file had ballooned in size because of the .wav files I had uploaded with it. The greater size caused a significant increase in loading time. A later application I created with more sound files (5+ MB) even failed to load at all. These incidents helped convince me that sample-based synthesis was not a scalable solution to the lack of music on Android. More importantly, however, the experience told me that application size needed to be watched carefully.

I was not alone in feeling this way, and picked up on comments from the App Inventor Developer's mailing list about AIA's extremely large apk sizes. One factor contributing to the application bloat was the lack of an ability to discriminate resources: an App Inventor application included the library files and resources needed for all possible components, not just the ones being used. I began to become concerned about my component: while using SuperCollider would avoid needing an escalating number of sound files to produce music, the library was still several megabytes large. I worried that this overhead would preclude my component from being integrated into App Inventor, music synthesis being a fairly narrow focus in comparison to the potential space usage. Later in the project, I decided to forego other potential enhancements to the component in order to pursue library loading.

Motivation

As mentioned, a large contributor to the need for selectively loaded files was the large size of current App Inventor apps, which could cause both storage and performance issues. Loading only the resources necessary for an app's components would address this problem in

several ways. It would first reduce the size of current App Inventor applications by omitting unused files (the Twitter library for example). This space saving is likely to be minor; more important are the prospects for future components. Component developers will now be able to include sizable amounts of resources secure in the knowledge that only apps using their components will be affected. Their apps might be larger, but the component will not do any further harm that might make their component unadoptable.

Let's consider the existing Twitter library as an example. Two application files are pictured here. The application is identical in both cases, but the first was compiled before the Twitter libraries were added selectively. The second, identical in all other respects, does not include the Twitter library, and is 100 kB smaller. While not a huge number, this space will be saved by *all* App Inventor Apps that don't use Twitter. Given that the Twitter component has a very narrow applicability and is difficult to set up, this is nearly all AIA apps.



Name	Size	Type	Date Modified
 Blank.apk	1.5 MB	Zip archive	Wed 11 Jul 2012 04:59:45 PM PDT
 Blank2.apk	1.4 MB	Zip archive	Wed 11 Jul 2012 05:16:42 PM PDT

Figure 9 - Relative sizes of an identical app with and without the Twitter component

Functionality and Implementation

We implemented the technique for selective loading using App Inventor's annotation processor. "Annotation processors are Java programs that produce an output or perform actions based on annotations²⁶ (introduced in Java 1.5) that they find in Java source code. The purpose of the App Inventor annotation processors is to produce information about the run-time components needed by different parts of the system."²⁷ The annotation processor gave us a

²⁶ "Annotations." *Java Documentation*. Oracle Co., 7 Sep. 2011. Web. 29 Nov. 2012.

<<http://docs.oracle.com/javase/1.5.0/docs/guide/language/annotations.html>>

²⁷ Spertus, Ellen. "App Inventor Annotation Processors." *Google Docs*. 23 July 2012. Web. 23 Nov. 2012.

way to tag needed resources from within the component code.

```
27 /**
28  * Musical instrument component that can play discrete notes of specified pitches,
29  * volumes and durations.
30  *
31  * @author trevorbadams@gmail.com (Trevor Adams)
32  */
33 @DesignerComponent(version = 1,
34     description = "<p>A musical instrument component that will play notes of the specified " +
35     "pitch, duration and volume. Duration is given in milliseconds and " +
36     "volume is 0 to 100. Pitch can be given as either the frequency (in Hertz), " +
37     "or as the letter and accidental (# or b) of the note, followed by a number specifying " +
38     "the octave.</p>",
39     category = ComponentCategory.MEDIA,
40     nonVisible = true,
41     iconName = "images/instrument.png")
42 @SimpleObject
43 @UsesFiles(fileName = "supercollider.jar", +
44     "libAY_UGen.so, libBinaryOpUGens.so, libChaosUGens.so, libDelayUGens.so, libDemandUGens.so, " +
45     "libDynNoiseUGens.so, libFFT_UGen.so, libFilterUGens.so, libGendynUGens.so, " +
46     "libGrainUGens.so, libIOUGens.so, libLFUGens.so, libMCLDBufferUGens.so, libMCLDFFTUGens.so, " +
47     "libMCLDTreeUGens.so, libMCLDTriggeredStatsUGens.so, libML_UGen.so, libMulAddUGens.so, " +
48     "libNoiseUGens.so, libOscUGens.so, libPanUGens.so, libPhysicalModelingUGens.so, " +
49     "libReverbUGens.so, libscsynth.so, libsndfile.so, libTriggerUGens.so, libUnaryOpUGens.so, " +
50     "libAY_UGen-v7a.so, libBinaryOpUGens-v7a.so, libChaosUGens-v7a.so, libDelayUGens-v7a.so, " +
51     "libDemandUGens-v7a.so, libDynNoiseUGens-v7a.so, libFFT_UGen-v7a.so, libFilterUGens-v7a.so, " +
52     "libGendynUGens-v7a.so, libGrainUGens-v7a.so, libIOUGens-v7a.so, libLFUGens-v7a.so, " +
53     "libMCLDBufferUGens-v7a.so, libMCLDFFTUGens-v7a.so, libMCLDTreeUGens-v7a.so, " +
54     "libMCLDTriggeredStatsUGens-v7a.so, libML_UGen-v7a.so, libMulAddUGens-v7a.so, " +
55     "libNoiseUGens-v7a.so, libOscUGens-v7a.so, libPanUGens-v7a.so, " +
56     "libPhysicalModelingUGens-v7a.so, libReverbUGens-v7a.so, libscsynth-v7a.so, " +
57     "libsndfile-v7a.so, libTriggerUGens-v7a.so, libUnaryOpUGens-v7a.so, " +
58     "sine-inst.scsyndef, saw-inst.scsyndef, triangle-inst.scsyndef, pulse-inst.scsyndef, " +
59     "noise-inst.scsyndef, reverb.scsyndef")
60 public class Instrument extends AndroidNonvisibleComponent
61     implements Component, OnResumeListener, OnStopListener, OnDestroyListener, Deleteable {
62 }
```

Figure 10 - Usage of the @UsesFiles annotation in the Instrument component

When first designing our annotation, Spertus pointed out that its goal would be similar to that of the preexisting @UsesPermissions annotation. @UsesPermissions listed (as a comma separated string) the permissions that each individual component required so they could be compiled later into a list of all the permissions an app would need; we likewise wanted to know the assets required by all of an app's components. Our component was eventually named @UsesFiles, and follows @UsesPermissions' lead closely.

Once we had gotten the annotation recognized, the rest of the functionality was quite simple. Basically, the compiler walks through its list of components and gets all the filenames which components have requested, then attempts to load them from the runtime directory. It is assumed developers have gotten the files there already. The file extension determines what is done with each file: jars are dexed, .so's are put in "lib" and other files are added to the APK's "Assets" directory.

Process

The central decision behind the library loading was the scope of what kinds of files to support. Jars would be supported, being the primary library format. I would also ensure that native libraries were supported, so that SuperCollider would benefit from the improvement. Though this was the original extent of the changes (the annotation was first called `@UsesLibraries`), I was also adding synthdef files to the application assets directory. We decided that making this process generic, so others could use it as well, would be a highly useful feature. By placing the non-library files in the app assets, they should be accessible for most purposes within the app. I later received a suggestion that `@UsesFiles` and `@UsesLibraries` should be two separate annotations; however I had already written the code by that point.

Another issue was the annotation syntax, how component files were specified. `@UsesFiles` is supplied a single string containing the required filenames, separated by commas. Google Engineer Mark Friedman correctly pointed out that using an array would be more convenient and less messy. Unfortunately, the String format was used by `@UsesPermissions`, the annotation ours was based on. Two similar annotations with different paradigms for their input would be confusing; we would have to change `@UsesPermissions` as well. This would involve modifying all existing components with permissions, including ones being independently developed right now. We decided it was better to keep things the way they were.

I knew from the beginning that I would have to make decisions about how to add native libraries, but the choices became much more complicated than I expected. The first challenge arose from the architecture targeted nature of the files: should we support all architectures, and how would we handle such things using the phone's interface? Because the different versions of the .so files need to be identically named, we had to determine a standard for putting the files

in the runtime folder where the Compiler could find and differentiate them. After considering several options, I decided that ARM files (the default architecture) would be taken directly from the runtime directory, while other versions would be placed in appropriately named subfolders.

I also had to decide how component designers would specify which versions of a file to include. I first intended to have Compiler automatically load all files it found (designers could still decide which versions to include by putting the files they wanted in their proper places). Implementation problems (crashes when files were missing) and a suggestion from Spertus convinced me to have designers specify file versions by attaching a suffix to the filename (“foo-v7a.so” would get “foo.so” from the Armeabi-v7a subfolder and place it appropriately).

This method has its own problems. Besides being unintuitive, every filename must be repeated for every supported architecture. Unfortunately, when I decided on this method, I believed there were only two possible Android architectures (the two produced from the NDK by default). Mark Friedman pointed out that a small number of MIPS and x86 devices exist, which would double the size of a fully compatible native library (four instead of two), as well as double the length of the include list. Supporting the additional processors would also require us to target API level 9 (Android 2.3), dropping support for users with previous versions (15% of all devices).

²⁸ This is impossible currently, but could change if significantly more such devices emerge.

Closing

It turned out I had picked a very popular issue to address; due to communication issues, another developer merged his own selective libraries fix to the project shortly before I finished mine! While this precluded my implementation of selective Java libraries, it did not include native code or assets. I am currently rewriting my code for these two features to work with his. It is my

²⁸ “Platform Versions - Current Distribution.” *Android Developers*. Google, 1 Nov. 2012. Web. 23 Nov. 2012. <<http://developer.android.com/about/dashboards/index.html>>

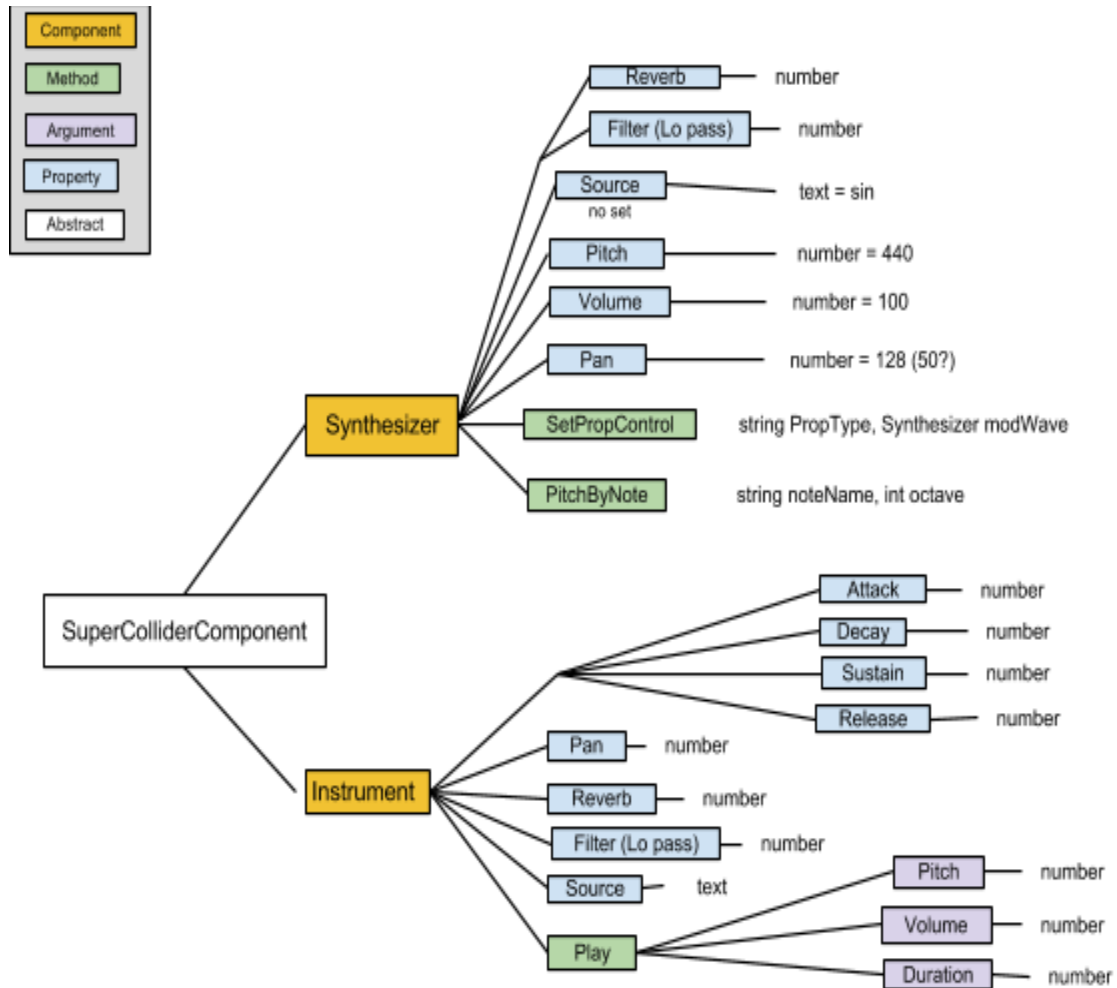
hope that easy inclusion of these resource types will greatly expand what component developers can accomplish in the future.

V. Conclusion

Ultimately, I succeeded in my objectives of bringing music to AIA, pioneering native code inclusion and providing component developers new tools for including resources. Each of these achievements provides immediate benefits to App Inventor, but may offer even more in the potential future developments they enable. There is also room for further improvements to my component itself. I have learned a great deal from both my successes and my mistakes, which may be useful to others as well. In my last chapter, I'll discuss both future possibilities related to my work and lessons I've learned from it.

Possible Extensions

I've mentioned several ways in which this project has changed along the way; these choices were necessary, but some of them represent opportunities for improvements that still exist. The most notable omission is the sister component to Instrument, Synthesizer, which would have offered more robust control of the sound to users and allowed the modulation of arguments by oscillators (using low frequency sound waves to continuously vary another sound). A good way to implement this would be to create a new abstract class, `SuperColliderComponent` (or `SCComponent` for brevity) that both components inherited from. The boilerplate code that sets up the SuperCollider server in Instrument would be moved to `SCComponent`, making it easier for future developers to add their own SuperCollider components.



Of course, the most obvious addition is that of further synths. Most people will want to emulate real instruments (rather than the bare bones oscillators provided), and extra sound-sets will be a necessity for the component to be appreciated. Over time, I have had more success locating precomposed synthdefs, as well as making my own. I will likely be adding several new sounds myself, though the possibilities will never be exhausted.

When I decided to use SuperCollider to add synthesis to App Inventor, I consciously decided that it was a higher priority than several other concerns I noted; these concerns are still waiting to be addressed. First of all, my “starter project”, ensuring that all sounds are loaded when an app starts, remains incomplete. The project was specifically chosen for being (in theory) short and simple; it was only dropped because of my other setbacks. Adding easy

playback speed altering to the Sound component could also make a good supplement to the capabilities of Instrument.

Reflection and Suggestions

Considering the choices I made on the project inevitably brings up things that I might have done differently given a second chance. Unanticipated problems are obviously a large component of software engineering (and I certainly experienced many), and some cannot be prevented. I have learned an enormous amount during this work, however, and it goes without saying that I'm better equipped for many of these problems. I'll definitely be using this knowledge in the future, and perhaps any students reading this can pick up some hints as well.

Working on a large open source project like App Inventor has impressed upon me the necessity of communication in a team. The largest team I'd previously been involved with was only five people, and the multitude of problems that can result from isolation had never occurred to me. A common example is problem-solving; other people might have solutions occur to them that you wouldn't have thought of. Other times you might be able to figure something out alone, but asking will save valuable time.

More dramatically, contact is necessary to adapt to changes or developments in the project. Attending a video meeting after a long hiatus I overheard that we had recently switched our version control from Github to Gerrit. This was surprising, as last I had heard we were using Mercurial! I had missed not one but two transitions. Another example was when a different developer implemented conditional libraries before I had submitted my changes. I had previously written a proposal and gotten it approved, but actively informing the group of your work is necessary to stay noticed.

One concrete thing I would have liked to have done is finished my starter project. I abandoned the project because of the major setbacks I encountered early on, but later found

myself tripping over how to manage commits and submit code. Completing an entire change (albeit a simple one) would have left me much better prepared for submitting my massive changes later (my changes grew so large I had to split them in two).

Final Thoughts

Having dedicated the previous year to this project, I've found it difficult to summarize in a mere thirty or so pages, much less a final few paragraphs. It's almost bewildering, then, to think of the world beyond my small component; it hasn't been particularly relevant for some time. Having explained my project to countless people, I'm quite used to the idea that it will yield some practical benefits: a junior high student might make a simple piano app, or an undergraduate add a C++ graphics library to App Inventor.

But these aren't the reasons I decided to make this component. I've been exposed to numerous open source projects that I could work on. App Inventor is different because of how it aims to change the world. I've often heard computer programming described as something that "you either get, or you don't": it's characterized as an inherent, unlearnable skill. App Inventor has the audacity to suggest that anyone can write a program. It implies that everyone has the right to design software, and that good ideas can come from anywhere.

It's this idea that attracted me to the project. Every day, technology becomes more important to everyday life. As it continues to do so, everyone has the responsibility to control the technological parts of their life, and to demand that they fulfill our needs. The intersection of music and cell phones is just one small example of how computing can enrich the rest of our lives.

Bibliography

Android Developers. Google, Inc., n.d. Web. 8 Nov. 2012. <<http://developer.android.com/>>.

"Annotations." *Java Documentation*. Oracle Co., 7 Sep. 2011. Web. 29 Nov. 2012. <<http://docs.oracle.com/javase/1.5.0/docs/guide/language/annotations.html/>>.

Anshul Bhagi, "Adding App Inventor components that use External JARs." *Google Docs*. Google Inc., 10 Nov. 2012. Web. 29 Nov. 2012. <<https://docs.google.com/document/d/1NyHU8pmQbqZclnxwTsG22f8bM8HtiqvVfCJp1AvsUsc/edit#heading=h.fyozwbkl2itv>>.

"App Inventor Issues List - Search for 'sound'." *App Inventor for Android*. Google Code. Various dates. Web. 23 Nov. 2012. <<http://code.google.com/p/app-inventor-for-android/issues/list?can=2&q=sound&sort=-stars&colspec=ID%20Status%20Summary%20Owner%20Reporter%20Stars>>.

"Error: bash ./install: No such file or directory when installing Installation Manager on Ubuntu 64 10.04 LTS (64-bit)." *IBM Support*. IBM, 25 June 2010. Web. 23 Nov. 2012. <<https://www-304.ibm.com/support/docview.wss?uid=swg21438771>>.

Gibbs, Mark. "App Inventor, Scratch and Simple Programming." *Network World*. 28 July 2010. Web. 2 Nov. 2012. <<http://www.networkworld.com/community/toolshed/app-inventor-scratch-and-simple-programmi>>.

Kirn, Peter. "Ethereal Dialpad Touch App, Development Experience on Android and Beyond." *Create Digital Music*. Create Digital Music, 5 May 2010. Web. 23 Nov. 2012. <<http://createdigitalmusic.com/2010/05/ethereal-dialpad-touch-app-development-experience-on-android-and-beyond/>>.

MasterDroid. "Intro to the three Android Audio APIs." *Wise Android*. 13 July 2010. Web. 29 Nov. 2012. <<http://www.wiseandroid.com/post/2010/07/13/Intro-to-the-three-Android-Audio-APIs.aspx>>.

peterdk. "The state of MIDI support on Android." *Umito.nl*. Umito, 7 May 2010. Web. 23 Nov. 2012. <<http://blog.umito.nl/2010/05/07/midi-on-android.html>>.

Siegler, MG. "Is Google App Inventor A Gateway Drug Or A Doomsday Device For Android?" *TechCrunch*. 11 July 2010. Web. 2 Nov. 2012. <<http://techcrunch.com/2010/07/11/google-app-inventor>>.

- Spertus, Ellen. "How to Add a Component - Internal." *Google Docs*. 16 Oct. 2012. Web. 23 Nov. 2012. <https://docs.google.com/document/d/1l9qLSllxFgootg-IYSwEEqp3yjp0N_u0yxPAhq0aB94/edit#heading=h.gairsbwyrmlq>
- Spertus, Ellen. "App Inventor Annotation Processors." *Google Docs*. 23 July 2012. Web. 23 Nov. 2012.
- "SuperCollider - About." *SuperCollider.Sourceforge*. n.p., n.d. Web. 10 Nov. 2012. <<http://supercollider.sourceforge.net>>.
- SuperCollider-Android*. Github Inc, 12 Feb. 2011. Web. 28 Nov. 2012. <<https://github.com/glastonbridge/SuperCollider-Android/wiki/>>.
- "Support for real-time low latency audio; synchronous play and record [UPDATED]." *Android Annoyances*. Wordpress, n.d. Web. 26 Nov. 2012. <<http://www.androidannoyances.com/post/tag/low-latency-audio>>.
- "Synthdef." *Supercollider Documentation*. 29 Nov. 2012. <danielnouri.org/docs/SuperColliderHelp/ServerArchitecture/SynthDef.html>.
- "TimeZone.class.getDeclaredField("defaultZoneTL") does not exist in JDK 6." *Google App Engine Issues*. Google, 9 May 2012. Web. 23 Nov. 2012. <<http://code.google.com/p/googleappengine/issues/detail?id=6928>>.
- "UGen." *SuperCollider 3.2 Help Files*. SuperCollider, 28 Jan. 2011. Web. 8 Nov. 2012. <<http://danielnouri.org/docs/SuperColliderHelp/UGens/UGen.html>>.
- "What is App Inventor." *MIT App Inventor*. 8 Nov. 2012. Web. 8 Nov. 2012. <<http://experimental.appinventor.mit.edu/learn/whatis/index.html>>
- "What is the NDK?" *Android Developers*. Google, Inc., n.d. Web. 8 Nov. 2012. <<http://developer.android.com/tools/sdk/ndk/overview.html>>
- Wolber, David, Hal Abelson, Ellen Spertus, and Liz Looney. *App Inventor: Create Your Own Android Apps*. O'Reilly Media, 2011. *University of San Francisco Department of Computer Science*. David Wolber. 23 Nov. 2012. <<http://cs.usfca.edu/~wolber/appinventor/bookSplits/ch9Xylophone.pdf>>.