Spring 3-1-2018

NESynth Project

John Falco
Olivet Nazarene University, jpfalco@olivet.edu

CSIS 49X Research Paper

NESynth Project Report

Research Paper

John Falco

In fulfillment of the requirements for CSIS 491 and MULT 500

Abstract

This project was designed to create a program that can interface with user input through a keyboard to produce sounds through the use of the MIDI protocol. The goal functionality was to have the unit be able to interface with a Nintendo Entertainment System to produce synthetic sounds along with sampled sounds through the MIDI protocol. However, this goal was not able to be achieved due to technical limitations. Due to this, the sound of the A203 chip was emulated through use of sampled instruments using MIDI. The goal of this project initially was to also include a USB keyboard hybrid built into hardware that would act as a controller. Due to technical limitations, this goal was also not met. In addition to the above, this project was done in part to fulfil the requirements for an Interdisciplinary Minor, which blends Computer Science with another field of study. This field for the project was Music. Through the study of the MIDI protocol and interfacing with the user, the project took the form it needed to and accomplished its goals during the course of study. In this paper, the limitations of each part that built the NESynth will be analyzed to show how understanding these limitations brought the solutions that make the project what it is, an exercise in limitations.

NESynth is a program that allows a user to interface through the use of MIDI technology to produce sampled sound using a Java Interface. It is a project littered with technical limitations that had to be surpassed in order to make the scheduled release window of only 8 months, a self-dubbed exercise in limitations. Despite all the limitations, however, the project was a success and accomplished its desired goals of producing sound through the use of a keyboard. The key factor as to how this was done was to understand how these limitations were surpassed. This fact lies in understanding these limitations. In this research paper, covering the project, I propose that the best way to produce a quality product is to understand limitations so they can be worked around. To demonstrate this, I will show how each aspect of the NESynth program posed a limitation and what was taken in order to handle these limitations in a way that would not impact the release of the product.

Rather than using a custom-built audio engine, which would have prevented the program from releasing on time, or any other sort of audio software that would needlessly complicate the process, the NESynth uses the MIDI protocol to send signals to the sound hardware of a computer to produce sound. From the computer, it sends data to the actual hardware that produce sounds through use of MIDI Messages. A single midi message in and of itself always contains at least one byte of data ("The Complete MIDI 1.0 Specification"). Each message type is divided into two categories, which are channel messages and system messages. Channel messages, which are encoded with channel numbers, control the instruments being played and also impact how they are being played. System messages are not encoded with channel numbers and are used for things such as timing and status. Modern computers are able to process these messages through synthesizers coded into the computer itself. This fact, in and of itself, posed a limitation in the sense that true hardware sound was not being produced, nor was there any guarantee that the

performance of the emulated hardware being correct. And while there are no true workarounds for this limitation, it does pose implications down the line when trying to interface with the MIDI protocol. One major limitation of this is the fact that any note-on message from the MIDI system must have a note-off message afterwards or the note will fail to play. This creates a limitation with timing of notes especially when trying to read user input without a dedicated system built to handle it properly. This is something that will be discussed further when analyzing the limitations and workarounds using the Java language to interface with the user. Despite this problem, I was able to overcome it using a method in Java known as "Key Bindings", which will be covered in greater detail later on in the paper. There were still other technical limitations of the MIDI protocol that needed to be overcome first, however.

Another significant limitation of MIDI is the timing. Timing using a MIDI system is very imprecise, and attempting to simulate MIDI on many systems, especially Windows users, have multiple issues. According to Martin Walker in an article published in 2007 on the website Soundonsound, "PC users experience annoying MIDI issues, such as erratic timing, events that are consistently recorded too early or too late and doubling (or even tripling) of note data. In extreme cases no data may be recorded at all, or every single MIDI event recorded during a lengthy take may end up appearing at the start of [a] part". This is something I had to deal with extensively while writing the interface that played timing of notes and sequences, as converting machine ticks to human tempo is a frustrating process. However, with some help with some solutions found online, a workaround for playing notes from a sequence was able to be achieved. The solution involved converting a MIDI file into text that could be interpreted and read by the program and stored into the data type I used to store sequence data. This method was adapted by a user known as Sami Koiyu on Stackoverflow. Without their initial code, the project would only

be able to play one single MIDI sequence at a time. Through this understanding of the limitation of the MIDI software, a solution was able to be reached. There was, however, one last limitation of MIDI that needed to be addressed.

One of the final consequences of using MIDI was the fact that most MIDI samples are not of significant quality when played on a machine. This is because MIDI files do not actually store any sound data, rather they store messages that are read by the machine that produces the sound itself ("The Complete MIDI 1.0 Specification"). On many systems this leads to sound quality that is significantly lower than that produced on other machines. This is one of the major criticisms of MIDI, and it was a difficult technical limitation to work around. However, a solution was able to be reached by finding samples of a significantly higher quality online. On a system, instruments are stored in banks and presets, and these can be edited by software and loaded in with special files. These files are known as soundfont files, and through this higher quality samples were able to be obtained and combined to produce the sound of the program. This method was also how the NESynth was able to play retro-inspired sounds that are generated from the A203 chip found on the NES hardware. Through use of a soundfont and sampled data, this data was captured and recorded for playback.

Despite all of MIDI's technical shortcomings, it still remains the pillar of musical interfacing between hardware and software, and has evolved greatly over the last thirty years of use it has experienced. Through its numerous functions and capabilities, the NESynth program was able to produce sound effectively. It also helped to grow my personal understanding of sound design, something I will take and apply into my musical works. By learning about sampling data, I was able to overcome the limitations of MIDI quality, and by accounting for the limitations of timing a MIDI system I was able to produce a product that satisfied the goal of the

project. However, even with all of this understanding, more understanding would be required when tackling the elephant in the room. This elephant is known as Java GUI Interfacing.

As a brief introduction to the troubles and headaches of Java GUIs, it is important to understand the history of how user interfaces were developed using Java. In the original release of Java, JDK 1.0, there was the Abstract Windows Toolkit, known as AWT ("Java™ Platform Standard Ed. 7"). These were classes designed to provide a bare-bones user interface that adopts the native appearance of the underlying operating system (Sundsted). This approach was a necessary one because Java is a programming language built to be platform-independent, and must support a common toolset for GUI interfacing across these platforms. As one would imagine, this comes with problems when components display differently and may even function differently depending on what platform it is being run upon. This fact, coupled with all the complicated coding required by AWT just to get it to function left users dissatisfied with the result. To accommodate users more, a new API package was brought into Java on release 1.2, developed jointly by Oracle and Netscape known as "Java Foundation Classes" ("The Java Foundation Classes"). These classes were combined and renamed to what is now known as Swing, which is the most recognizable GUI interface when coding with Java. Since December 8th, 1998, Swing has been a staple of Java GUI development and is still taught to this day despite new GUI packages being introduced into the library such as JavaFX ten years later. While significantly more powerful and useful when being compared to AWT, Swing has many shortcomings, quirks and annoying behavioral shenanigans that posed significant obstruction when developing the NESynth program.

The single most taxing problem with interfacing a GUI system is the way inputs are handled. This is done through events and listeners, which follow the observer design pattern

(Bevis). The observer pattern allows one of the design goals of object-oriented programming, the principle of loose-coupling, in order to establish a connection between a class and any class that retrieves data from this class. In Swing, this is accomplished through the use of Event Listeners. Most components that interface with a mouse interact with two different listeners, and these are the ActionListener and MouseListener classes. The ActionListener is a unique and user-friendly listener class that handles "actions", hence the name, performed on any compatible Swing component such as JButton or JFileChooser. They function simply, and can even be triggered by the program itself through methods such as the doClick method which fires an ActionEvent to the listener, a feature the NESynth used extensively. Most applications only need to use ActionListeners, as according to Oracle's own tutorial covering ActionListeners, "Action listeners are probably the easiest — and most common — event handlers to implement". Unfortunately, some components do not interface using actions or the ActionListener class. In this instance, the MouseListener class is used. The MouseListener class itself is not overly complex to implement, however there are some issues with how MouseEvents are triggered. Specifically, I was running into issues where multiple events were being generated for the same event when interfacing with the NESynth. This was baffling, but in order to handle this limitation I was able to use a singleton-inspired approach when handling MouseEvents. If a MouseEvent with the exact same timestamp was received more than once, only the first event would run the necessary methods to achieve the results. This prevented double or even triple events from happening. This solution required being able to understand how these events were being generated multiple times in order to prevent them from being read multiple times.

Another significant issue of interfacing with the user lies in the way the keyboard generated input that was read by the program. The traditional approach is through a class known

as a KeyListener. KeyListeners are very universal and simple to understand and maintain, however they lack capabilities. Multiple KeyEvents cannot be handled at the same time as each other, and for the NESynth which needed to be able to play multiple keys at once, this solution simply was not sufficient. Fortunately, there was a way to handle multiple keys at the same time. This is known as “Key Bindings”, a feature that is supported by JComponent and by extension the entire Swing library as it is built off of the JComponent class. This was required when writing the program because not only did I need to handle multiple inputs at once, but I needed to be able to directly link these bindings to the MIDI system that I had developed. This class, known as the SynthListener class, uses Key Bindings when assigned to a JComponent to play notes from the keyboard. It also holds all the functionality to interface with the MIDI system through things such as timing of notes, playing sequences, changing instruments and changing the way instruments sound. To accomplish this, I used a technique that was adapted from an article written by Rob Camick on his Java Tips Weblog site. This technique involved using Key Bindings coupled with extending the AbstractAction Class that performs a stored action whenever a key binding is triggered through the InputMap. This approach allowed for flexible, reusable and versatile code that handles multiple keystrokes being pressed at the same time (Camick). Not only this, but this method of handling keyboard input allowed the MIDI functionality to perform properly, by waiting for the key to be released to send the note off message when playing using the keyboard. Without this, the MIDI drivers would not be able to sound. Through the flexibility of the code, I was able to achieve the result needed to successfully complete the project in this regard, and that was to have a functional MIDI system working that could play notes properly. However, there was yet another barrier in the way of this, and this was due to an error in Java’s code itself.

In terms of legitimate obscurity, the biggest limitation that attempted to halt progress on this project was the actual Java API codes itself, or rather a specific part of it to be specific. Java interfaces to MIDI through the use of an abstract class known as Synthesizer, with abstract methods that are loaded up that directly interface with the sound system installed with Java. In this case, the sound system that was loaded was Java, known as the Gervill Java Softsynth MIDI player. These files are final and cannot be extended to be edited by the user, which in theory is good. However, there was a bug in the code that seemed to be unsolvable until looking at the actual source code itself. When attempting to read the current bank from a channel, the SoftChannel class, the internal Java implementation of the abstract MidiChannel class that the user interfaces with, does not actually load the right data. Changing the channel produces the right results, and this can be confirmed by switching the bank. Tthe actual channel isn't properly handled in the code, and cannot be obtained from the MIDI channel in any conceivable way. The actual method described in the Java API to get the bank from a channel will always return zero, even if it is not actually zero. This is a problem because this means that the current instrument cannot be properly read from a channel when changed, which was an essential part of the user interface. Not only this, but the class couldn't be extended in order to fix this issue. As a workaround, I instead stored the data of the bank and program in a separate class that served as a memento design pattern (Bevis). Because I could not have direct access to the data within the SoftChannel program, instead I stored this information and returned it when needed, in this case the data was restored when the user interface needed to be updated and must re-obtain the current instrument from the MIDI channel. This obstacle was not even very apparent, and it is crazy to think that such an error hasn't been addressed before, and such a workaround was needed to solve the issue. If Java were a more open language and didn't have internal classes hidden, a

memento design pattern wouldn't even have been needed to fix this glaring issue in the source code. This would allow a direct fix of the problem and not clutter up the class with excessive code that shouldn't even need to exist in the first place. Despite this, I was able to understand the issue and was able to overcome the limitation of the bugs in the actual Java source code itself. After all this hardship, there was one final obstacle that remained, and that was babysitting Java's painting protocol.

Swing, as a GUI interfacing tool, is designed very well. However, despite its best design efforts, doing something as simple as displaying the graphics to a user is the single most frustrating process imaginable. In order to properly describe this, I am referring to Nathan Pruett's answer to understanding all the different painting methods contained in the Java API from coderanch.com. In total, there are at least half a dozen different methods to 'paint' a component, and each have their own little quirks. The basic paint method holds instructions to paint a component, but should never be called directly. Instead, paintComponent should be called because paintComponent does not also repaint a border or any child component. There's also repaint, which cannot be overwritten and controls the cycle of paint and update, which is yet another method in this confusing system of painting in order to display graphics to the user. Update, in this method, cleans a component before painting the graphics again. This is responsible for the flickering in the UI, and is difficult to avoid as overriding paint can throw off this delicate balance between paint and update. To compound this frustration even further there is the validate method. This method is a clean slate of sorts, as it removes all components from the parent and paints everything all over again. However, instead you should call revalidate, further making everything needlessly complicated. On top of all these extraneous and confusing methods to paint a simple graphic onto a component, there's also the problem of how paint

actually works. Paint updates the UI passively, only when the system can handle an update. This means that any time-sensitive updates required to be occurring when the user demands it will do one of three things: work, not work or just break everything. The action taken usually leans in favor of the latter two options, which creates frustrating headaches on the part of the user trying to make sure the interface doesn't look like it was from the last millennium. This was a roadblock that took the absolute longest to solve, as simply displaying an image to a user that didn't look terrible wasn't an option. To combat all the issues paint has with drawing graphics onto its components, I used images to load graphics predefined and saved into data. The problem arises when the way to load these images isn't consistent or coherent. The system of loading an image with transparency into the program is complex, even adding something such as text is needless complicated and problematic. In total, I spent nearly a whole month just trying to make the painting code work right in the NESynth project, which is absurd and was a complete waste of time. Not only this, but attempting to dynamically paint as the user inputs data through the keyboard interface caused the program to slow down drastically, and this feature had to be cut for functionality. That limitation was one that I could not overpass using Swing, because it was simply not user friendly enough to achieve. Despite this, and despite the fact that the interface doesn't look amazing, it is thankfully functional, something all the validation, painting and needless complexities of graphical processing in Swing was not able to stop.

For the NESynth project, despite all the limitations and frustrations posed by trying to develop a GUI system using Java, the results achieved were functional and clean. That is the biggest accomplishment by far. The ability to listen to each key individually was an essential part of the project, and making the user interface understandable helped achieve this goal even more. That being said, there were some limitations that could not be surpassed due to hardware,

software and other limitations. Their workarounds accomplished enough to surpass them in their own way however.

In the original design for the NESynth program, one key component was the desire to interface directly with a real NES unit and the A203 chip to produce sound. Such a project had already been accomplished, but not in the same way as this. This product was the chip maestro by Jaroslaw Lupinski. It is a NES cartridge with a MID input connected to the game itself. By plugging any MIDI device into the adaptor, the player can perform live retro-sounding music. Inspired by this, I got into contact with him early in the project, asking him about his project before the chip maestro, which was the standalone NES player. His project documents the same steps I took in the early stages of this project. My goal was to directly interface with the A203 from the chip and produce audio through machine code instructions generated by the Java code. This would be done using the 40 GPIO Header pins on a Raspberry Pi unit. After getting into contact with him asking questions about the circuits on the project, I decided that directly interfacing with the A203 chip was far beyond my capabilities or understanding. Because I didn't have any knowledge of digital systems or electrical engineering at the time, I opted for a workaround for directly interfacing with the NES itself. Instead, I used recorded samples of NES sounds in order to achieve the same effect I had desired from the start of the project. This wasn't as pure as the original design I had wished to use, but reflecting back on it I believe the right choice was made in regards to this choice. Trying to deal with hardware not fully controllable would be prone to errors, especially on hardware that was over thirty years old. Sampling the sound was in fact the better option in this case, as it was easily controllable and could be manipulated in a desirable way. If I am to return to this project in the future, I definitely intend to

study more on circuitry so I too can attempt to recreate the same sort of setup and have a true MIDI-controlled sound driver using real hardware.

The other, and possibly the largest setback to the project was the decision to cut a homemade USB interface using actual hardware to produce keystrokes that would interface into the system. This was a factor that was ruled by my relative shallow understanding in electronic circuitry. Because of this, I felt that using a real USB keyboard would be much more conductive to playing music than some arbitrary plastic shell I built homemade for the project. The designs for this date as far as the beginning of the project, with the user interface revolving around a diagram of what the actual hardware controller would look like in its finished state laid out on a picture. Despite my best efforts to attempt to create this, it simply proved to not be possible to complete this before the open house due to the fact that I didn't have access to the hardware capable of producing such a design in the first place. It is another personal disappointment for me, because I felt it was in the spirit of the program to function like a homemade synthesizer rather than a simple program executed on a computer. This goal and vision was not able to be achieved, and was barred by more limitations than benefits in the end.

The major hardware limitation preventing a true USB keyboard controller from making it into the project was USB's limitations. On keyboards, there usually is a maximum number of keys that can be registered as being pressed at a given time. This number is referred to as "rollover". The usual number ranges from 4-6 depending on which keys are pressed, which order they are pressed in and other factors. To achieve true capacity, I would need to purchase and gut a USB "nkro" keyboard, which stands for "n-key rollover". Nkro is only achievable by specially built hardware and USB drivers, both things that I didn't have or couldn't obtain without spending money. This aspect was against what I felt the spirit of a "do-it-yourself synthesizer"

meant for the project I had envisioned. Spending more money on it was something that I didn't feel fit the project in any way, and so this aspect was dropped. Instead, the workaround is to simply use the regular keyboard on most systems in order to interface with the program. Such a concession was necessary in order to allow the project to deliver on time.

Currently, the NESynth is a completely finished and realized product, but there are things that in the future I desire to address with this project. As described before, I wish to fully build a working hardware interface for the A203 chip much like I had first envisioned for the project, as well as a working USB hardware interface that can function as the input device for the project. Beyond just this, I also desire to implement a new GUI system into the project using JavaFX, the newest version of GUI interfacing with Java.

JavaFX was formally introduced into the JDK release for Java 8 in March of 2014 ("JavaFX Frequently Asked Questions"), but has been supported by Java since the original release in 2008. With the new direct integration into the JDK, JavaFX would become the new GUI user interface as opposed to Swing. However, since it is so new in terms of technological time, new users of Java often are taught Swing instead, as was the case for myself. Looking over JavaFX's operations, I see now that it is so much more powerful and accessible than Swing could ever be, and in Java 9 it will be able to completely replace all the GUI interfacing within the NESynth project with the support of key bindings being implemented in this release. The reason why JavaFX specifically would be better for the project is that the design of the GUI is based off of HTML/CSS. This means that the relative simplicity of designing a web page interface would be possible to translate over into JavaFX and maintain the functionality of Java behind it. This benefit would remove all the struggles of babysitting the paint methods and allow more of the focus to be placed on the actual audio and interface itself, something that was

suffering due to the struggles of the painting methods. With the help of my friend Jorge, we were able to get up a functional interface in less than a day that looked just as good as the one currently implementing. In the future, I will be implementing this technology into the project to improve it drastically and make it function as I had intended it to.

As a whole, the NESynth project has taught me many things when it comes to understanding many areas of both Computer Science and music alike. Without it, I don't feel I would be as great of a programmer as I am currently, as it has helped me shape my abilities in order to overcome the obstacles placed before me. In the end, I was able to produce a product that produced sound through the MIDI system and interfaced with the user through the keyboard. By understanding the limitations of what I was working with, I was able to work around the limitations in the NESynth project, an exercise in limitations.

References

Bevis, Tony (2012). “Java Design Pattern Essentials – Second Edition”. Ability First Limited, UK. ISBN 978-0-9565758-4-5.

Camick, Rob (June 2013) “Motion Using the Keyboard” Retrieved from https://tips4java.wordpress.com/2013/06/09/motion-using-the-keyboard/

“The Complete MIDI 1.0 Detailed Specificatiion.” (1996). MIDI Manufactures Association, CA. Retrieved from https://www.midi.org/specifications/item/the-midi-1-0-specification

“The Java Foundation Classes” (2001). O’Reilly & Associates. Retrieved from https://docstore.mik.ua/orelly/java-ent/jfc/ch01_01.htm

“JavaFX Frequently Asked Questions” (2014) Oracle. Retrieved from http://www.oracle.com/technetwork/java/javafx/overview/faq-1446554.html#5

“Java™ Platform, Standard Edition 7 API Specification” (July 2011) Oracle. Retrieved from https://docs.oracle.com/javase/7/docs/api/overview-summary.html

“The Java™ Tutorials, How to Write an Action Listener” (2017). Oracle. Retrieved from https://docs.oracle.com/javase/tutorial/uiswing/events/actionlistener.html

Koiyu, Sami (Oct. 2010) “Reading MIDI Files in Java”. Retrieved from https://stackoverflow.com/questions/3850688/reading-midi-files-in-java/3850885#3850885

Lupinski, Jared (2016) “Standalone NES Player”. Soniktech.com. Retrieved from http://www.soniktech.com/StandaloneNesPlayer/

Pruett, Nathan (2003). “Whats the exact difference between paint, repaint, validate, update & updateUI ?” Coderanch.com. Retrieved from https://coderanch.com/t/333435/java/Whats-exact-difference-paint-repaint

Sundsted, Todd (Jul. 1996) “Introduction to the AWT.” IDG Communications, inc. Retrieved from https://www.javaworld.com/article/2077188/core-java/introduction-to-the-awt.html

Walker, Martin (Dec. 2007) “Solving MIDI Timing Problems” Sound On Sound Ltd, UK. Retrieved from https://www.soundonsound.com/techniques/solving-midi-timing-problems