Daniel Mendelsohn
November 4, 2013
6.11 Final Project Proposal Draft

An interactive plug-in game for an electric keyboard

Overview

I plan to build a plug in game for an electric keyboard. It is essentially a Guitar Hero variant, but unlike Guitar Hero, it will use an actual working instrument as an input device rather than a faux-instrument game controller

When learning to play a new instrument, it's important for novices to become deeply engaged in the music from an early stage. For experienced players, such engagement comes naturally with the satisfaction of creating music. Novices, however, must work through a significant period of struggle before realizing any impressive results. It is natural to feel somewhat discouraged. Many people fail to get over the learning "hump". Many parents who pay for their children's music lessons have experienced the challenge of motivating a new learner through this difficult period.

In many fields of education, games help drive the learning process and motivate students. This is particularly effective with regards to children. The massive proliferation of educational kids' games have demonstrated the effectiveness of gamification in teaching math, reading, language, typing and myriad other subjects. Thanks to my new 6.111 knowledge, I now see a new opportunity to gamify piano playing.

The user interface is modeled after other popular instrument-based video games, such as Rock Band and the aforementioned Guitar Hero. In the game, notes will flow horizontally across the screen from right to left. There will be a vertical bar near the left edge of the screen. When a note reaches that bar, the user must press the corresponding key on the keyboard. If the user plays the note successfully, he or she is rewarded. If the user fails to play a note, plays the wrong note, or plays a note when no note should be played, he or she is penalized. This format allows for a wide range of difficulty settings; selecting a song with more notes or a higher tempo increases the challenge.

The hardware setup is simple from the user perspective. The user simply plugs in the MIDI output of the keyboard into the FPGA, and connects the VGA output of the FPGA to a monitor. All game data, such as images and note sequences, is stored in memory on the FPGA. The MIDI output is a good choice for the interface to the FPGA due to its simple, compact, and low traffic protocol. Similarly, I chose to output display information via VGA because of its simplicity and familiarity (and because we already have a module to do this thanks to Lab 3). It is important to note that the scope of this project, at least initially, is limited to just one octave, in order to ensure the completion minimum viable product.

IMPLEMENTATION

My design is comprised of four major blocks, as seen in Figure 1, namely a MIDI interface block, an action interpretation block, a game logic block, and a display block. A MIDI interface block receives input from the keyboard and interprets it, relaying that data in a more concise, usable format. An action interpretation block compares game input information from the MIDI interface block (what notes the user *is* playing) to internal game state information (what note the user *should be* playing). Based on that comparison, the action interpretation block analyzes when events (e.g. score changes) must take place, and relays information about such events. A state-less display block, which receives relevant game state data, produces an RGB value to be displayed at a specified coordinate. A game logic FSM ties it all together; it reads data from memory, performs operations such as tallying score, holds the state of the game, and controls the other blocks. It does this by providing data about which notes are in the song and at what times. Different peripheral blocks have different needs; for example, the display block needs information about a note many seconds before the user must play that note (so it can be shown at the top of the screen).

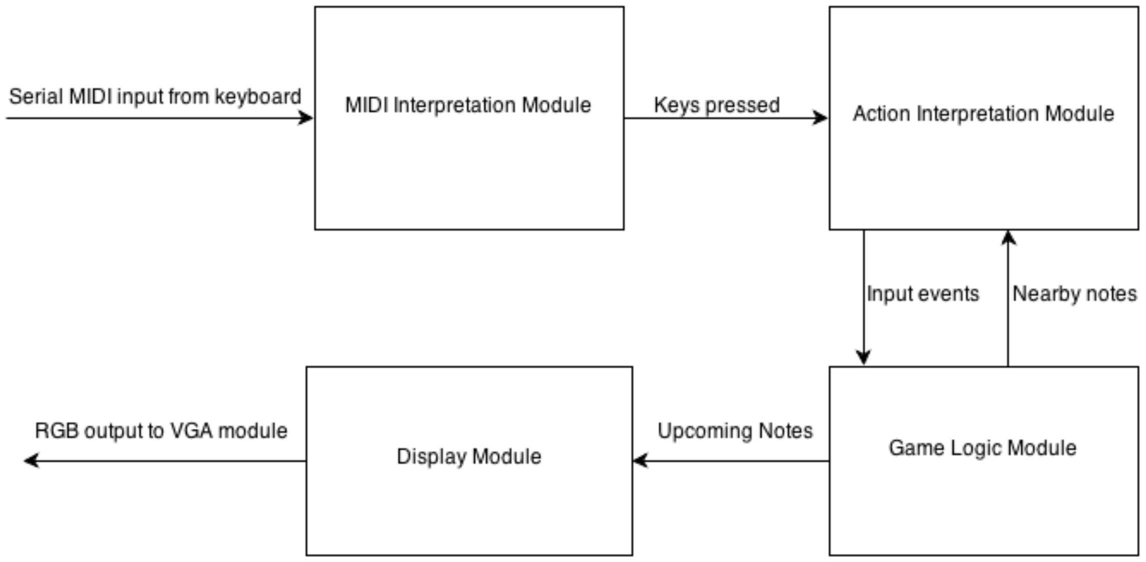

Figure 1. This is a block diagram depicting the largest components of the system. MIDI input is parsed and the notes that the user is playing are compared to the correct notes. The action interpretation block performs this comparison. Based on that comparison, the action interpretation block tells the game logic if the user performed a correct action, an incorrect action, or no action at all. The game logic block reads song data from memory, and distributes it to its neighboring blocks at the correct time. It also incorporates already-interpreted actions into the game state. The display block receives data for upcoming notes from the game logic block, and creates a user interface that visually represents that data (using VGA output).

MIDI Interface Block

This block receives electrical signals from the MIDI output port of an electric keyboard, and interprets it using the MIDI standard's specification. It produces an array of registers, where each register represents the "press/un-pressed" state of a single note.

A simple circuit must be constructed on the proto-board. It must meet the electrical specification for MIDI, so that the signal can be interpreted digitally. The specified circuit can be seen in Figure 2.

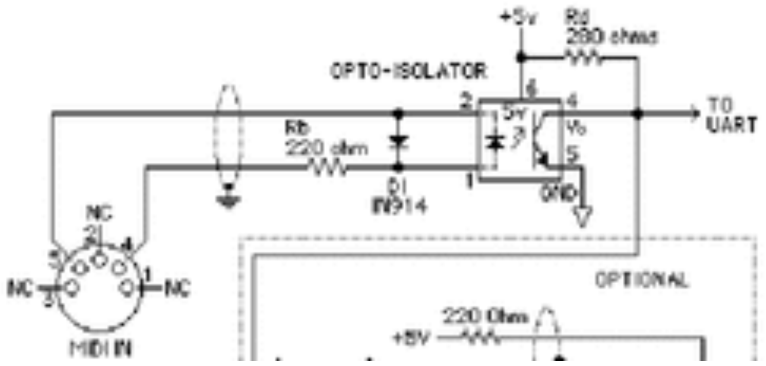

Figure 2. A schematic of the circuit to convert signal from MIDI connector to serial bits. This is as simple as converting the MIDI standard voltage levels to TTL voltage levels.

The block will then interpret the digital signal according to the MIDI standard. The standard calls for a data transmission rate of 31,250 baud. Digital words consist of eight bits, plus a start and stop bit. The first of the eight bits indicates whether this word is a command word or a data word. A MIDI message consists of a command word, along with up to two data words that serve as parameters for the command. The four most significant bits are often enough to specify a number of commands; the least significant four bits allow for us to specify one of 16 channels. This is useful when multiple devices are involved, but in our case we only need one channel. Refer to Table 1, below; it gives more detail about these commands.

MIDI commands

Command	Meaning	#Parameters	Param1	Param2
0x**8**0	Note Off	2	key	velocity
0x**9**0	Note On	2	key	velocity
0x**A**0	Aftertouch	2	key	touch
0x**B**0	Continuous controller	2	controller#	value
0x**C**0	Patch change	2	instrument#	value
0x**D**0	Channel Pressure	1	pressure	
0x**E**0	Pitch bend	2	lsb(7bits)	msb(7bits)
0x**F**0	(non-musical commands)	0		

Table 1. A detailed look at MIDI commands and their parameters

The system will include a simple Universal Asynchronous Receiver/Transmitter (UART) circuit that will take the serial stream of bits and convert them to data words. An FSM will interpret MIDI messages and continuously keep track of which keys are being pressed. It then outputs that information.

As stated before, the system uses the MIDI output of the keyboard (rather than the actual sound) because of the greater simplicity. Thorough testing requires knowing when keys are pressed, and MIDI requires and easy and lightweight way to do this. The implementation of this block should be quick and straightforward.

In addition to ModelSim testing, the system will be tested by simply displaying it's output on a bank of LED lights, and ensuring the lights go on when the corresponding key is pressed and goes of when the key is released.

User Action Interpretation Block

The action interpretation block will take certain game state data as input. In particular, the input will indicate which notes are to be played close to the current time (either just previous to the current time or in the near future). The input will also indicate the exact time at which those notes should be played. Furthermore, the output of the MIDI interface block will also fed as input to the scoring block.

In total, the action interpretation block knows which notes should be played at roughly the current time, and which keys are currently pressed. It combines this information to determine if the correct keys are being pressed at the correct times. It also accounts for some tolerance; users should be allowed to press a key a short amount of time (on the order of tenths of a second) before or after the exact time specified by the game.

This block will output event information as a pulse. For example, when it detects a scoring event such as a correct key press, and incorrect key press, or a missed key press, it encodes the relevant information (type of event plus parameters) into a binary array, and outputs it for one clock cycle. At all other times, the output is 0.

This block will be mostly tested in ModelSim. In simulation, it must correctly interpret each type of event (such as correct key press, etc.) at the appropriate time. It will be very hard to verify this block's functionality via visual inspection since the output information only persists for one clock cycle. This is an ideal situation to use ModelSim.

Display Block

The display block is the interface between the internal logic of the game, and a VGA output module identical to the one used in Lab 3. The graphics will depict a musical staff on the screen, and upcoming notes will flow from right to left along the musical staff. Notes will be depicted using an icon, which this block will retrieve from memory. The final result will look somewhat like the depiction in Figure 3. This format is standard in music and is very familiar to all musicians. For piano players, the musical staff is a ubiquitous way to understand the notes to be played.

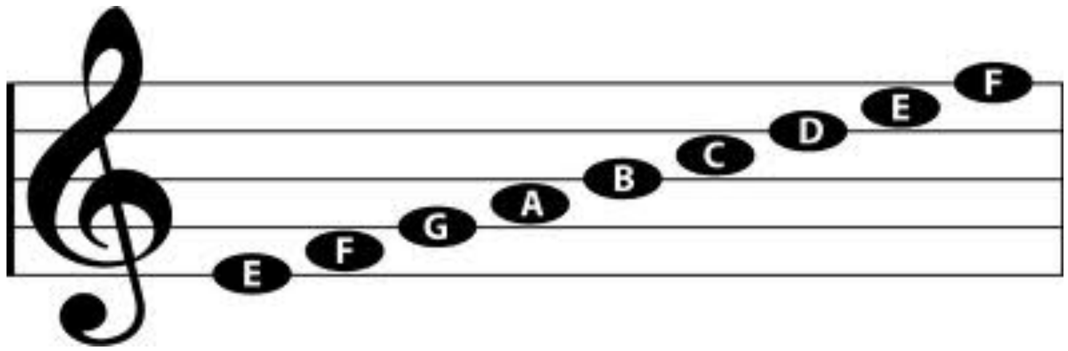

Figure 3. A possible example of what the display might look like; notes flow leftward.

This block will take relevant game state data as input. Namely, its input will be a binary encoding of all the notes that will occur in the next few beats. It will also take as input a pair of numbers representing the location of a pixel. Given the all of the notes, and the location of the pixel, the display block will determine an RGB value for that pixel. Note that this block is state-less, its output is a function of its current input. The VGA module handles the creation of an actual output signal to the monitor.

Due to the large size of this block, and the complexity of its inputs, it is difficult to test thoroughly via ModelSim. Unlike the action interpretation block, the display block is very amenable to visual verification that it is meetings its specification. A test circuit on the FPGA will allow a human operator to manually tinker with the input to the display block. Using that test circuit, the operator will be able to confirm that this block's logic meets specification and the monitor displays the proper image.

Game Logic Block

This block ties it all together. It reads data from memory; this data encodes which notes are to be played and when. It provides relevant subsets of the notes to various blocks (e.g. the display block and the action interpretation block) at the appropriate times. It handles the "play/pause" logic and interprets events from the event interpretation. Thus, it tallies the current score. It will also contain persistent information that encodes various statistics such as note accuracy and high score. This block is essentially a large FSM running the entire game.

The testing for this block will be complicated, and thorough. It will likely be broken down into many small sub-blocks, each with a very specific function. Such sub-modules might include a data-reading block, a game-pausing module, and a scoring module. These sub-modules will mostly tested using ModelSim. Once the individual functionality of each sub-module is correct, a large ModelSim simulation will perform an integration test, thus confirming the correct behavior of the game logic block as a whole.

TIMELINE

The timeline is somewhat aggressive and front-loaded. It is imperative that it not be completed late. During the week of 11/18, each block will be individually built. During the week of 11/25, each block will be thoroughly debugged and refined. Finally, during the week of 12/2, the individually functioning blocks will be integrated together. During the time at the end, user testing will help create a better user experience.