Sound Synthesis with a Guitar Hero Controller

CMPT 768 Final Project

Adrian Kwok (#200136359)
Simon Fraser University

April 20th, 2010

Introduction

For the general populace, learning a new musical instrument without formal training is a daunting task. However, in the past few years, rhythm-centric rhythm games such as Guitar Hero and Rock Band have become popular with people of all ages, and have allowed players to reveal their inner 'rock gods'. These games are easy to pick up and play mainly due to the simplicity of its game controllers, yet are difficult to master because of complicated game mechanics. In many cases (myself included), these games have motivated people to attempt to learn new instruments – the most apparent being the transition from drumming in Rock Band to learning to play on a real drumset. However, the mapping from the plastic guitar to a real guitar is nowhere near as natural as that from the plastic drums to a real drumset; in this vein, my project aims to bridge this gap by taking advantage of the guitar controller's inherent simplicity and familiarity to allow users to create simple, yet expressive music. My project utilizes all the controls available in a standard Guitar Hero controller and maps them to 48 possible fundamental frequencies of an acoustic guitar, with the sounds created in real time using delay effects.

Controller Information

The standard guitar controller for Guitar Hero(fig. 1) consists of five coloured fret buttons(a), a switch-based strumbar(b), a whammy bar(c), a set of control buttons (a directional pad and start/select buttons)(d), and a built-in accelerometer/tilt-sensor. In its intended use in the game, players are shown a scrolling set of pre-placed note indicators corresponding to the current chosen song, and the goal of the game is to hold down the shown frets and strum when the note reaches the bottom of the screen. As with other games in this genre, the more accurate and in-pace with the rhythm the player is, the higher the overall score they receive.

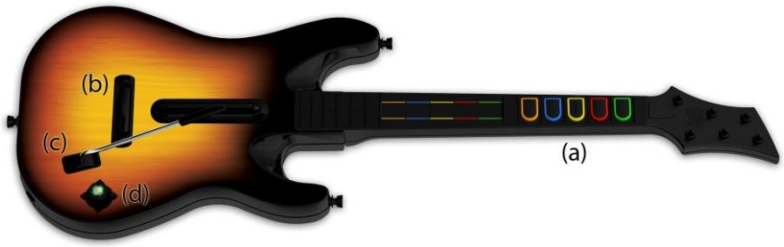


Figure 1: Guitar Hero IV Game Controller

Throughout the implementation process, there was an underlying concern regarding the mapping of these controls to the functions of a real guitar. While some mappings are trivial – the large select button near the strum bar would best function as a palm mute and the whammy bar would best function as an amplitude modulator – some are not that obvious; a real guitar has a much wider frequency range than the five possible frequencies afforded by the five plastic fret buttons, and even if one were to map all possible combinations of these buttons to a frequency, it would still not be enough to encompass all 48 fundamental frequencies of a guitar (number of button combinations: $\sum_{i=1}^{5}\binom{5}{i} = 32 \leq 48$). To compensate for this, I mapped multiple disparate controls to cover all 48 frequencies, and the specifics are described in the implementation section.

Technical Implementation Details

At the beginning of the project, two major technical hurdles were noted: capturing input from the Guitar Hero controller in Java, and creating a convincing guitar sound in real-time also in Java. This section will show the methodology in overcoming these hurdles.

In general, the project was coded with expandability in mind; the basic model-view-controller architectural pattern was followed so that there would be ample compartmentalization between components.

1.) Capturing input from the Guitar Hero controller:

To capture input from the Guitar Hero controller, JInput[i] was used. JInput is a Java API created as a universal interface for all gamepads – not specifically for Guitar Hero controllers. Fortunately, because the original Guitar Hero game allowed a player to use a normal XBOX360 gamepad in place of a Guitar Hero controller[1], the guitar controller itself was mapped to the buttons and axis sticks of a standard XBOX360 gamepad – and since JInput (almost) fully supported the XBOX360 gamepad, implementation was much easier.

Although it was initially difficult to figure out how to use JInput (there is a lack of formal documentation online), most of the buttons ended up being simple to capture – that is, the fret buttons, control buttons, whammy bar, and tilt sensor were trivial to capture and interpret states. However, since upstrums and downstrums were mapped to the 'up' and 'down' of the directional-pad (dpad) in an XBOX360 controller, there was difficulty in capturing strums. The creators of JInput did not have a formal method for capturing the dpad in XBOX360 controllers, since it was seldom used in game development; after much searching for a solution (and on the verge of abandoning Java and switching purely to C++, since Microsoft's XInput API would've been much easier to use), I eventually stumbled upon an obscure project called "PG3B" (Pan-Galactic Gargantuan Gargle Brain[ii]) on Google Code, which somehow managed to capture dpad input by interpolating the POV axis of the XBOX360 controller. I adapted this code and finally managed to get strums working.

To allow for more functionality, I added a simple timer to allow the start and select buttons to have two modes each – that is, holding onto the start button would result in a different action than just quickly pressing and releasing the button.

[1] *In the franchise's infancy, Guitar Hero was bundled with only one guitar controller, with no option to buy another guitar controller separately – so if a player wished to play cooperatively with a friend, the friend would usually be designated to a XBOX360 controller.*

Technical Implementation Details (cont'd)

2.) Creating a convincing guitar sound in real time:

For audio synthesis in real time via Java, the JSyn API was used; JSyn is a unit generator-based synthesis engine, where the user can connect components together to create complex audio circuits. In the case of this project, the Karplus-Strong synthesis technique was used to mimic the acoustic properties of a guitar.

Specifically, the circuit used was:

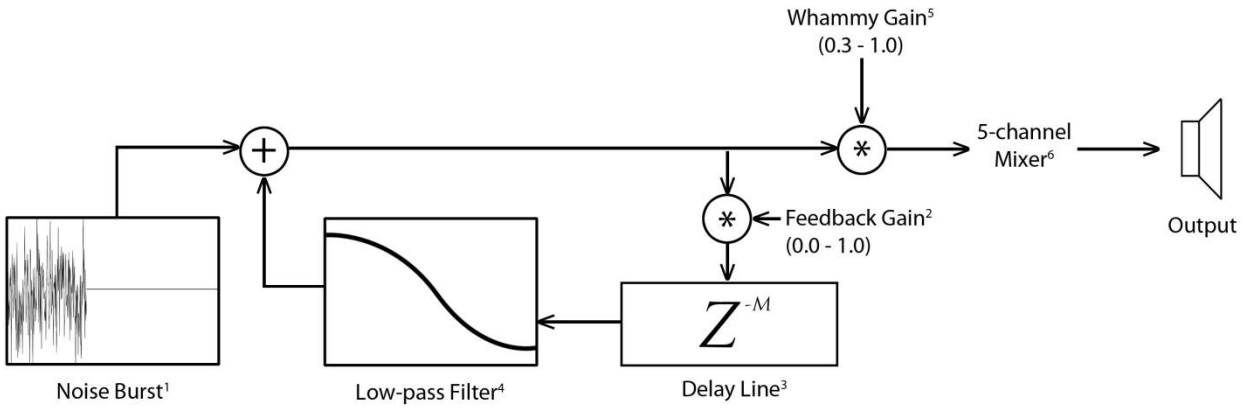


Figure 2: Modified Basic Karplus-Strong Circuit

Explanation:

This project uses five of these circuits in parallel – one for each fret. When a player holds onto a fret and strums (or, just presses a fret in the case where 'strumless' mode is on), a short noise burst is generated[1]. Then, in the feedback loop of this noise burst, the feedback is attenuated by a user-specified gain[2] (i.e., the note decay rate), which is then sent to a delay line of M samples[3]. The length of the delay line M is set to be the sampling rate divided by the fundamental frequency (f_s/f_0) – so M varies depending on the pitch of the fret being played. To complete the feedback loop, the signal is passed through a two-pole (second-order) lowpass filter[4]:

$$y(n) = a_0 x(n) + a_1 x(n-1) + a_0 x(n-2)$$

, where the feedforward coefficients were defined to be $0 \leq a_0 \leq 2a_1$, as recommended in the paper Electric Guitar Synthesis using the Karplus-Strong Algorithm[iii].

Before the signal (noise burst (if applicable) and feedback) is sent to the mixer, a whammy gain[5] of 0.3 to 1.0 is applied to the signal. If the controller's whammy bar is fully depressed, then the gain is set to be 0.3; otherwise, if the whammy bar is not depressed, then the gain is set to be 1.0 (e.g. signal passes through without alteration). At the mixer[6], all 5 circuits (again, each corresponding to each fret) are mixed before being sent to the output.

Practical Implementation Details: Frequency Mapping

As mentioned in the introduction, one of the problems faced during the design of this application was figuring out how to map five fret buttons to the 48 fundamental frequencies afforded by a real guitar (a real guitar has a range of four octaves, and each octave has 12 semitones, resulting in 48 fundamental frequencies). My final solution utilized the tilt sensor in the guitar controller; each fret was mapped to a base frequency, 5 semitones apart from one another. This gives a frequency range shown in the diagram below (where the characters 'G','R','Y', etc. denotes the fret color):

Octave 1												Octave 2											
G					R					Y					B					O			
82Hz					110Hz					146Hz					195Hz					261Hz			

Unfortunately, this does not encompass all the frequencies in the first two octaves (as denoted by the empty cells). This is where the tilt sensor comes into play; depending on the tilt of the guitar, a +0 ... +4 semitone shift is added to the base frequencies above, where a +1 semitone shift from some frequency a is defined to be $2^{0.5} \cdot a$. This results in the diagram below, where the numbers denote the semitone shift from the base fret.

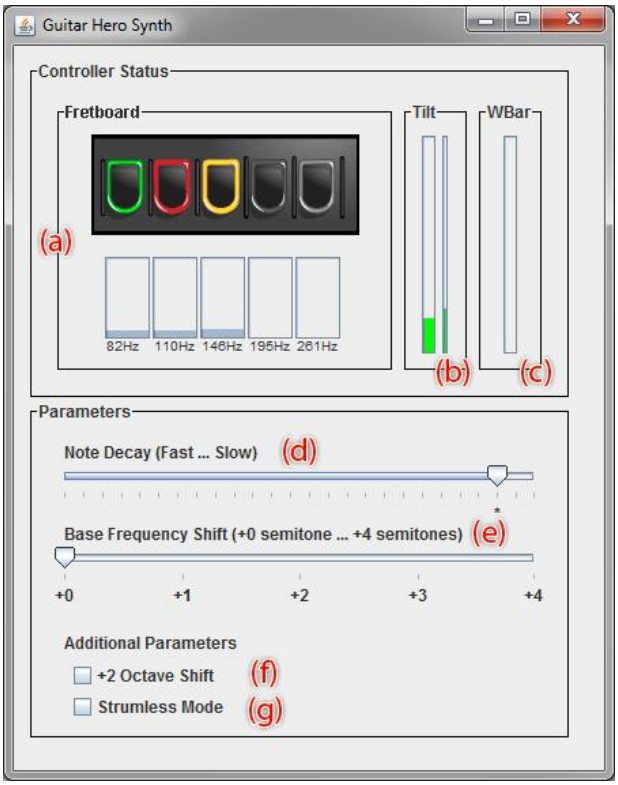
Octave 1												Octave 2											
G	1	2	3	4	R	1	2	3	4	Y	1	2	3	4	B	1	2	3	4	O	1	2	3
82Hz					110Hz					146Hz					195Hz					261Hz			

Since the tilt sensor is very sensitive, the user is allowed to lock the tilt sensor once they have dialled in the semitone shift they desire. Alternatively, the directional pad can be used to change between shift amounts. To cover the remaining two octaves of a real guitar, a "+2 Octave Shift" button is available to the user, which adds +2 octaves to each base frequency (aka a +24 semitone shift).

This mapping, although exhaustive, is unfortunately not very practical, and this drawback is discussed in the "future work" section.

Practical Implementation Details: Features

The GUI for my project is shown below, with subsections listed in red.



I will now explain each subsection in detail.

a.) Fretboard:

When a player presses the frets on the controller, the application shows a graphical representation of the presses. If the player strums and at least one fret is held down, a guitar pluck according to the current base frequencies of the frets will be played. The indicators below the fretboard show the current amplitude of each guitar pluck in real time, and also show the current fundamental frequency of each fret (this varies depending on the semitone shift and octave shift).

b.) Tilt:

As mentioned in the previous frequency mapping section of this report, tilting the guitar will apply semitone shifts to the base frequencies. If the guitar is held horizontal, then no semitones shifts will be applied; conversely, if the guitar is held completely vertical, then a +4 semitone shift will be applied to all base frequencies. The user can lock the tilt sensor (so that it stays at the semitone shift at the time of the lock regardless of the current tilt) by pressing the "select" button on the controller. Similarly, the user can manually change the semitone shift (without using the tilt sensor) by pressing "left" or "right" on the directional pad.

Practical Implementation Details: Features (cont'd)

c.) Whammy Bar:

The GUI provides a graphical representation of the current whammy bar state; if the whammy bar is completely held down, then the indicator will be fully green, and a multiplier of 0.3 will be applied to the output signal (as mentioned previously in the technical details section).

d.) Note Decay:

This parameter allows the user to set the decay rate of strummed notes; more specifically, it sets the feedback loop's gain depending on where the slider is – the faster the requested decay, the lower the gain.

e.) Base Frequency Shift:

This parameter allows the user to forgo the tilt sensor and the directional-pad to set the semitone shift. It is just there for added accessibility.

f.) +2 Octave Shift:

As mentioned in the previous section, this toggle is necessary to map the remaining 2 octaves of a guitar; it applies a +24 semitone shift to all base frequencies (and possibly on top of any other tilt modifiers currently enabled). The user can also toggle this by holding onto the select button on the controller for more than 1 second.

g.) Strumless Mode:

This mimics the "hammer on/pull off" behaviour in the original Guitar Hero game; if this mode is enabled, then the user will be able to play notes without strumming, given that it has been no more than 1 second since the last strum or fret. The user can also toggle this by holding onto the start button on the controller for more than 1 second.

e.) Mute (not shown in GUI):

The user can 'kill' off all decaying notes by pressing the big select button near the strumbar. This effectively applies a feedback loop gain of 0 to all five fret circuits.

Future Work

Electric Guitar Synthesis

In its current implementation, the basic Karplus-Strong technique provides a decent and recognizable acoustic guitar pluck; however, the original intention of this project was to synthesize an electric guitar. Unfortunately, due to time constraints this was not possible.

To mimic the sound of an electric guitar, the Karplus-Strong technique must be extended to take into account distortion, feedback, pickup tones, reverb, pedal effects, and many other properties. Similarly, it would be preferable to mimic the original behaviour of sustained notes in the Guitar Hero games – that is, a note should not decay (e.g. sustain) so long as the fret buttons that originally triggered the notes are still held down.

However, these are not trivial tasks. Electric guitar synthesis seems to be very in-depth, with no real 'easy answer' to model them sufficiently. There have been a few papers written by Sullivan[iv], Jaffe/Smith[v], and Eustace, as well as several other resources available online that would be useful in accomplishing this.

Usability

A fundamental question must be asked: does the user really need to have all 48 frequencies on hand? When a player pops in a rhythm game, all they want to do is to feel like Hendrix for a few hours; they don't want to go through the tedium of learning the guitar for years to experience this. By awkwardly mapping five fret buttons to 48 frequencies – while entertaining for the purposes of this project – the controller ends up being no easier to play around with than a real guitar. As such, a very interesting method to increase usability was suggested by Professor Smyth: allow the user to load in a music file (or subsection of a music file, e.g. a popular guitar riff), and map the most prominent frequencies to the five primary fret buttons, and less prominent frequencies to either combinations of fret buttons, or, similar to how it was implemented in the project, to the tilt sensor. Thus, the user has full control over the rhythm, tone, etc. of the overall piece.

It could be further expanded upon by creating an online database of fret->frequency mappings (and other possible metadata, e.g. guitar tone, amount of distortion, effects, etc.) for popular musicians, songs, or genres. For example, if a user wants to generate a musical score that's grungy in nature, they can load in a 'Cobain' artist preset so that each fret strum is a distorted, heavy chord. Of course, these would just be recommended, starting, presets – the user would be able to tweak other parameters as they see fit. There is much more to be explored in this specific area and could eventually lead to some very interesting (and useful) results; since my current academic interests lie mainly in usability and HCI, this is one major aspect of the project that I would like to pursue further.

Perceived Latency

During my presentation, I mentioned that there were latency issues in Java. After experimenting further, it seems to be a perceived latency due to gap difference in the actuation switch and the 'bottoming out' sound of the strumbar. A simple strum delay will probably remedy this (it's only really noticeable on quick triple strums, etc.).

Quick References:

[i] JInput: https://jinput.dev.java.net/

[ii] Pg3b – "Pan-Galactic Gargantuan Gargle Brain": http://code.google.com/p/pg3b/

[iii] Eustace, G. 2006. Electric Guitar Synthesis using the Karplus-Strong Algorithm: http://www.music.mcgill.ca/~eustace/mumt_614/614_project/Eustace06ksguitar.pdf

[iv] Sullivan, C. 1990. Extending the Karplus-Strong algorithm to synthesize electric guitar timbres using distortion and feedback: http://www.jstor.org/pss/3679957

[v] Jaffe, D., and Smith, J. 1983. Extensions of the Karplus-Strong plucked-string algorithm: http://www.jstor.org/pss/3680063