Surfing on a Sine Wave

6.111 Final Report

Sam Jacobs and Valerie Sarge

1. Overview

The goal of this project was to produce a single-player side-scroller game, titled Surfing on a Sine Wave, in which the player uses a MIDI keyboard to control the landscape traversed by a sprite. The sprite remains at a fixed offset on the screen, moving continuously along a wave collecting coins to score points and avoiding obstacles that end gameplay. The frequency of the wave is determined by the most recent key pressed on the keyboard – low keys correspond to low frequency oscillations, and high keys correspond to high frequency oscillations. If multiple keys are pressed simultaneously, the most recent key press will be used. The sprite stays at a fixed horizontal offset and the player changes the path to cause the sprite to collide with or avoid game objects.

A MIDI keyboard is used for input, and outputs include a VGA monitor and audio. The game state machine responds to frequency data from the keyboard, as well as internal status checks such as hit detection. Frequency data is used to display a sine waveform in the background of the game. Audio output corresponds to the input frequency. Players will attempt to pick up coins to gain a high score while avoiding enemies. The ultimate goal of this project was to produce a fun, engaging, and visually pleasing game with an unconventional control scheme.

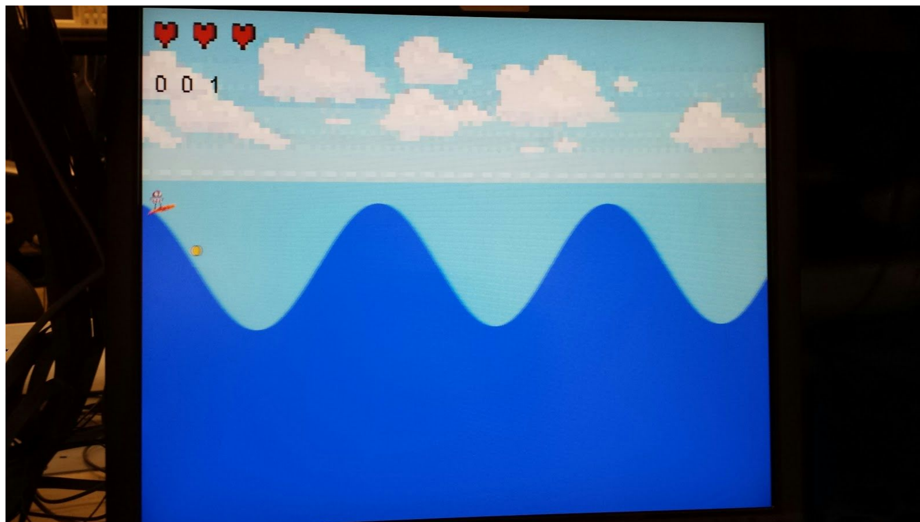

Figure 1. Game play

2. Design

The game will be organized around a central game logic module with peripheral modules for MIDI deserialization, waveform/physics calculation, and sound and video output (Figure 2). Values transmitted between modules will also typically be accompanied by a ready signal.

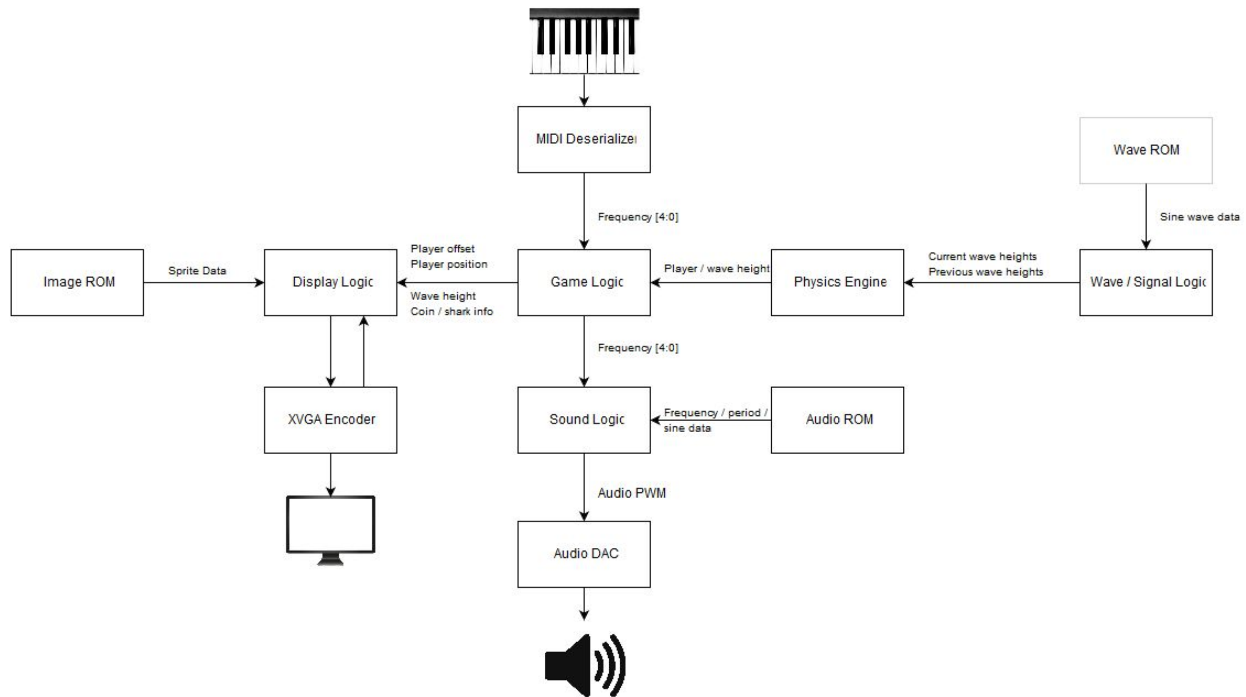

Figure 2. Block diagram; modules described in detail below.

3.1 MIDI Deserialization (Sam)

The decoder module deserializes the signal from the MIDI keyboard and extracts the note value. Because the MIDI signal operates at a baud rate of 31.25 kHz, and our logic uses a 65MHz clock, the decoder module samples every clock cycle, counting 2080 clock cycles for each bit. When the signal is driven low, the module begins recording, loading the first seven bits after the start bit into registers. It then records the next byte of the message, which tells the module what to do with the pitch value information. If the command byte represents neither "Note On" nor "Note Off," the information is discarded and the module returns to its waiting state. For the original design that only accommodated single keypress, if the command byte was a Note On message, the decoder module would send the recorded pitch value information on to the game logic module, ignoring Note Off messages.

In order to accommodate double keypress, the logic is slightly more complicated. The design interprets two keys held down at the same time as a double keypress, but remembers only the last key pressed when the player lets the keys go. On a Note On message, the module checks first to see if key one is being held. If not, the new key index replaces the old index for key one. If yes, it checks to see if a second key is already being held. If no, the new key index becomes the second key index. If a second key is already being held, the new information is disregarded. If the message is a Note Off message, the module will stop broadcasting the corresponding pitch value only if two keys are currently being held. If only one key is held or the Note Off message corresponds to a pitch value not currently stored, the message is ignored.

3.2 Game Logic (Sam)

The game logic module is a central module that keeps track of information about the state of the game, mediating between the various modules. It has two states: start screen and game play. When the player presses a key, the game begins. The speed of scrolling, and therefore the difficulty of the game, is settable using switches on the development board. The game logic module is responsible for converting MIDI indices into frequency indices understood by the physics module.

The meat of the game logic module's function is maintaining information about the sprites on the screen. This involves reading the position of the character sprite from the physics module and pseudorandomly generating collectable sprites. In order to do this in an unexpected way, the game logic uses a module that implements the Mersenne Twister random number generation algorithm in hardware. The number generation requires a seed, which we have chosen to be the state of the switches on the development board. This leads to 2^{16} different possible game states. The generator outputs a new random-ish number every time an input signal pulses high.

The game logic uses six bits of each new random number, in tandem with a three bit frame counter, to generate an object about every 2^9 frames. When a new object is generated, the game logic adds a different seven-bit number taken out of the 32 random bits to a fixed offset to give the new object a vertical position, which it maintains as it crosses the screen. Every new item begins on the far right side of the screen, its horizontal position decremented every frame by the aforementioned speed to move it across towards the player sprite.

The module accomplishes hit detection by watching for overlap between the box containing the character sprite and each of the five object sprites. When the box containing the character overlaps with an object box, the object is removed from the screen and the appropriate game state change is made. If the object is a coin, the score is incremented by one after a collision. If the object is a shark, the player's health is decremented by one. These values are communicated to the display module to be shown on screen, and are reset when the player's health reaches zero and the game enters its start screen state.

3.3 Display ROMs (Sam)

We used read only memories to store images of game sprites for access by the display module. These ROMs each stored a configurable number of 4-bit RGB pixel values for each x,y coordinate of the image. The module returns 12 bits of RGB information to the display module corresponding to the relative x,y coordinate value on the input. These modules were designed to accommodate multiple different frame images of the sprites to enable animation during gameplay. The ROMs contain logic that allows the developer to scale any of the images up in size. These ROMs were used to display all aspects of the game except for the background imagery.

3.4 Wave Logic and Physics (Valerie)

The wave logic portion of the project was split into several modules. The most fundamental module is the wave function ROM. This ROM contains the scaled vertical values of the first quarter of a sine wave period (where the period is 1024 and the maximum height is 768, mirroring the size of the screen). In other words, the smallest pair in this ROM is (0,0) and the largest is (256,768). The ROM also contains various frequency and period pairs within the two-octave range being used. It takes as input a frequency ID (5 bits in width) and an index (10 bits in width), and outputs the corresponding frequency, period, and sine wave height values. For indices over 256, it outputs the correct value by finding the index lower than 256 which outputs an equal value.

This ROM is used as input to the wave logic module. The wave logic module's general function is to store and output the height of a particular-frequency waveform. It receives as input a frequency ID (5 bits in width), which is latched by a ready signal, and a value for the horizontal offset. This module functions more as a wrapper for the ROM than anything else.

Given time, I would change the design of this module to take up significantly less space (at the expense of a few clock cycles of latency). Currently, after receiving the ready signal, the module stores a period of height values from the ROM (stretched or shrunk depending on the frequency) in a 1024-space array. One height value is stored per clock cycle; the module asserts a ready signal when a full period of values have been stored. This protocol does have some benefits. Given an arbitrary index, time-consuming-multiplication has to be done to adjust the index to the correct offset given the new frequency; however, since all indices have to be adjusted, it's possible to just use addition and shifting instead. Additionally, the latency for producing a value is very low. However, there is a significant tradeoff in storage space. The new design would calculate the frequency-adjusted index from the input index (this can be done in two clock cycles accounting for the delay of multiplication), then send this index to the ROM.

Four wave logic modules are contained within the physics module (organized as two pairs). This module calculates and outputs a smooth, time-continuous path for the player character and output levels for the background wave display. Two frequency ID inputs and a corresponding ready wire are used to signal a new key press, or a new chord, to this module. (If only one frequency input is used, all bits of the other should be driven high to signal that it is not being used). One pair of wave logic modules is given these new frequencies, while the other pair retains the previous frequencies. An internal blending coefficient decreases geometrically each clock cycle, so that the output heights gradually change from the prior frequency to the new one. The player height output is updated every frame, while the wave height output is updated each clock cycle (with every new hcount input). Another input, the change in offset, allows for internal sideways scrolling of the sine wave.

The inputs for this module are produced from the game logic module; the outputs are used in game logic and display.

3.5 Audio Output (Valerie)

The most fundamental module of the audio output section is the audio wave module. This module, and its accompanying ROM, serve a similar purpose to the wave logic module; they output a level given a frequency. No input index is given; the progression of time is controlled entirely internally. This has the benefit of requiring neither the filling of an array nor complex calculations; the adjusted index is calculated as in the wave logic module, but this module will never need to produce the height level for an arbitrary index input.

Audio output from the Nexys 4 must be done via PWM. The top level audio module contains six audio wave modules, up to four of which will be used at a time, and produces an output PWM based on the two input frequencies (with ready signal).

This module has two modes of operation. In the first, the audio module produces one or two frequency tones matching those most recently played by the keyboard. In the other, the audio module chooses a base frequency (the key) and a starting chord state and generates simple four-voice music to accompany gameplay. There are states for generating music in both minor and major keys. This music does follow chord progression rules, but doesn't always correctly resolve tendency tones, and doesn't check for parallel fifths and other prohibited movements in formal four-voice music.

The PWM output itself is generated using a six-bit converter. The audio wave modules output levels, which are averaged to produce an overall level between 0 and 63. A counter continuously loops from 0 to 127. When this counter is less than the current level, the PWM output is 1; otherwise, it is 0.

3.6 Display Framework (Valerie)

The display module takes in information from game_logic including the state of the collectables and enemies, the height of the player, the waveform profile, the score, and produces display output one pixel at a time. It also takes input signals from the XVGA module (the one provided for an earlier lab).

The Nexys 4 can display 4-bit color through the VGA output port. As such, the display module produces a 12-bit value for its output. Various sub-modules represent the player character, collectables, enemies, and upper and lower background. These modules each produce their own 12-bit values for the pixel; this value is 0 if they should have no effect on the output for that pixel. For example, a coin sprite will produce a value of 0 for everything outside of its 15x16 square. The pixel values from player sprite, collectables, and enemies are given higher priorities than background pixels, so that they will show up over the background.

Each pixel value is produced by an individual module for that kind of display entity. For instance, the coin module will, for values within its 15x16 square, retrieve the correct pixel value from its individual ROM. This module also receives a frame value to account for animation; this frame value is received from the game logic module in all cases but the player character, for which it is generated internally.

The background pixel values are generated in a slightly different manner; instead of being zero outside of a certain square, they are zero either above or below the given waveform height, which acts as a dividing line between the upper and lower backgrounds.

4. Implementation and Testing

The implementation strategy was to work from the outside in, beginning with the various peripheral modules and integrating them together towards the end of the development cycle. The testing of each module will be detailed individually in this section, in roughly chronological order of their completion.

4.1 MIDI (Sam)

The initial impulse in the game design was to use a keyboard with USB capabilities. Most MIDI keyboards made today use USB technology instead of the legacy MIDI 5-pin DIN connection, so designing the game for USB keyboards would make it more accessible to folks wishing to emulate the game on their own. The MIDI-over-USB protocol is also strictly defined, meaning the game design would be largely device-agnostic, another benefit. However, further research would show that this design would be impossible given the materials at our disposal.

Several factors prevented us from using MIDI-over-USB keyboards. The first was the availability of a USB port on the Nexys 4 Development Board. While there is a USB port on the board, the input from that port is fed directly into a chip that processes the raw USB data and culls out the relevant information to send to the FPGA. That chip is designed to handle only information according to the USB Human Interaction Device subclass protocol. Devices like mice and lettered keyboards use this protocol, but MIDI devices use a different subclass standard. It would be impossible to use the onboard USB port. A conceivable workaround would have been to construct a breakout board for the USB signal, feed the signal to the FPGA through the general purpose input/output jacks provided, and do all the processing on the FPGA. However, the USB protocol involves complicated device handshake procedures and data packetization techniques that would have been messy and difficult to write the Verilog to decode. The other reason we chose a different route was that requiring a breakout board for the USB keyboard detracted from any gains in convenience we might have gotten from using a USB keyboard in the first place.

Fortunately, Gim had a 60-key Casiotone keyboard in the storeroom that used the legacy 5-pin DIN technology. Because legacy MIDI devices run current from a source to a sink to represent an “on” bit instead of driving the voltage high on a pin, it was necessary to build a buffering circuit that would translate the current movement into a sequences of voltages that the FPGA could read through its GPIO. The circuit involved feeding the MIDI signal current through an optocoupler chip. Inside of the optocoupler chip, an LED would turn on when current ran through it, activating a phototransistor that would open and close in synchronization with the

signal current. With source connected to ground and drain connected to a 4.7kΩ resistor pulling up to 3.3V, the regulated voltage was taken from the drain pin and sent into the FPGA for processing.

Writing the deserialization and decoding module also involved some unexpected hurdles. The MIDI protocol defines messages to contain a command byte followed by one or two argument bytes, i.e. the pitch value and the volume. The Casiotone keyboard we used for our project followed an altered version of this scheme. Upon examining the signal on an oscilloscope, it became clear that the first byte represented the pitch value with least significant bit, followed by a byte that represented "Note On" or "Note Off" commands, the values for which did not correspond to the MIDI standard. Because the scheme was consistent between messages, we were able to build a decoder module that works for the particular keyboard we have been using in the lab.

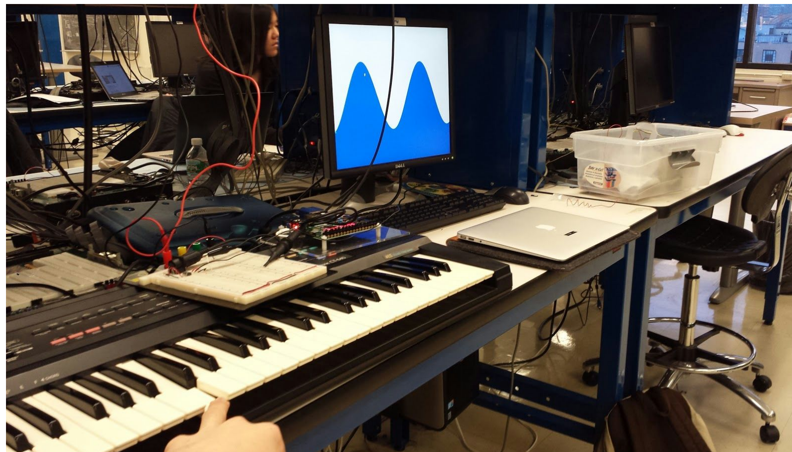

Figure 3. Early integration of MIDI, display, and wave modules.

4.2 Game Logic (Sam)

The development of the game logic module involved accommodating the incremental changes to the peripheral modules, while also designing the features that would make the game playable for an audience unfamiliar with the game internals. Early implementations dealt directly with the wave logic module, which prevented the module from being as abstract as we had intended. It served as the connective tissue that enabled advanced testing and integration of the existing peripheral modules. However, as the various peripheral modules were developed, the game logic module shrank in scope to deal strictly with game state representation. While this testing and development model involved some necessary scope creep and eventual rollback, it was a minor disadvantage of a model that allowed for distributed and parallelized contributions to the project.

The game logic module was the module focused on during the playtesting stage of development. The most major change during playtesting was a modification of the hit detection logic. We made the decision to consider a small zone underneath the player sprite as collision-eligible, even if the sprite and object boxes do not strictly overlap. This accords with the player's perception of descending onto an object below their sprite and accumulating it, even though the boxes containing the objects do not strictly overlap. After further playtesting of this modification, we decided to extend the hit detection zones only for coins, which made the game slightly easier and last longer.

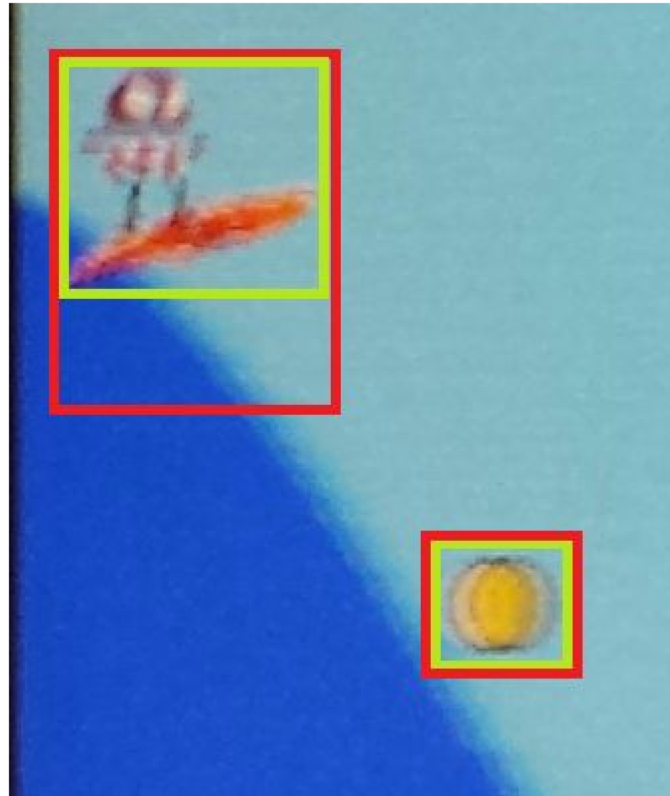

Figure 4. The boundaries of the sprite are illustrated in green, with the effective collision boxes shown in red. When the collision boxes overlap, the object is accumulated.

4.3 Display ROMs (Sam)

While there was some clever logic involved in the design of a 2-dimensional ROM, much of the development work on this module was focused on inputting the lookup table for each image. With the number of images in the game, it became clear very quickly that we needed a more efficient way of generating the display ROMs. I developed an image processing script in Python that outputted the image information in Verilog code, which allowed us to iterate quickly with different images and removed some inefficiencies in our debugging cycles. The script guesses the background color by sampling the top left corner of the image and zeroes out colors within a

small Manhattan distance of the background color in its output. This clustering technique was necessary because of variance of background colors resulting from image compression codecs. Zeroing out the background color makes the background of the image transparent, allowing our sprites to appear as though they are actually in the gameplay scene.

4.4 Wave Logic and Physics (Valerie)

Implementation of this module occurred in steps. The wave logic module was first linked directly to the VGA output (through a very early version of display) and used to display a horizontal line and a ramp. After this proof of concept, the ROM was filled and a basic-frequency sine wave was displayed. The first major checkpoint was the demonstration of the ability to display sine waves of any of the required frequencies.

The ROM method for producing a sine wave was originally intended to be temporary; the idea was to replace it with a small number of Taylor series terms to avoid using much storage space. However, this turned out to be both difficult computationally and expensive in terms of latency, so the idea was abandoned once it became clear that using a ROM was indeed much faster. This idea of the ROM as being preliminary is also part of the reason why the wave logic module still uses the inefficient method of filling an array before asserting the ready signal; this method was easier to implement, and the efficiency seemed unimportant given that it would be replaced. Later, there was not enough time to replace the module.

One of the larger challenges in this project proved to be the writing and integration of the physics module. The initial checkpoint was the production of output waveforms without modification. Implementing the smooth transitions between waveforms turned out to be much more challenging than expected; the initial implementation did gradually switch from one waveform to another, but did not attempt to retain the player's offset along the wave to minimize vertical position changes. Adding this extra feature required some re-structuring of the module, but did result in a much more professional-looking and playable product.

4.5 Audio (Valerie)

The implementation of this module was similar to that of the wave logic module. The initial step involved generating a square wave of the input frequency; following that, the ROM was filled and 6-bit sine wave output was tested. The most time-intensive step was writing the framework, including a finite state machine, for generating the music. This code was tested step-by-step; checkpoints included being able to play a I chord, playing a chord corresponding to the two input frequencies, and finally playing chord progressions.

This module was intended to be able to play a number of different styles of audio output, including a triangle wave. However, during attempts to implement triangle wave output, I discovered that because of the low pass filtering, a special band limited triangle wave output from the ROM, and likely a higher bit resolution on audio output, would be required to make a

reasonable output waveform. Due to these constraints, and a lack of time, only the square wave and sine wave were implemented as waveform types.

Another challenge was figuring out how to play two waveforms simultaneously. Due to the finite resolution on period and frequency, playing two non-identical frequencies simultaneously results in some amount of low-frequency noise. This issue was mitigated by increasing the resolution of period and frequency values in the ROM; however, it still appears for some pairs of frequencies in the square wave output mode. This is another issue I would like to find a better solution for, given more time.

4.6 Display Framework (Valerie)

The display module was implemented early in the development process, but continued to change throughout. Some form of it was necessary to test most other parts, including the wave and game logic modules. The final framework for this module was present in an early version, in which a dummy sprite for the character and the final coin sprite, including animation, were displayed along with the background waveform. This version was sufficient for testing all parts of wave logic and physics, as well as the early game logic.

Not many challenges were encountered in the early development and integration of this module, partially because it was one of the first modules to be written. Some issues arose with the later addition of more ROMs, including the shark and player sprites and the background image; to integrate with these ROMs, some changes and tweaks were required, but no major restructuring.

5. Conclusion

In summary, our project produced a video game in which players attempt to collect coins for points and avoid sharks to avoid losing one of three lives. Unlike most video games, in which the player directly controls the on-screen character, players of our game must use a MIDI keyboard to input one or two frequencies controlling the oscillation path of the player. On-screen, the player appears to slide along ("surf") the upper edge of a sine wave (or the summation of two sine waves) which is displayed in blue. The character always remains at the left side, as coins and sharks approach from the right.

Major challenges that we encountered in our implementation mostly centered around making the game logical and fun to play. For example, early gameplay tests showed that it was necessary to revise the physics module. The game was made much more playable by changing the physics module to conserve vertical height across frequency changes. Though there is potential for improvement, we believe that our current project fulfills the goal we initially set out to accomplish: the production of a fun, engaging, and playable game making use of an unusual control system.