Surfing on a Sine Wave

6.111 Final Project Proposal

Sam Jacobs and Valerie Sarge

1. Overview

This project aims to produce a single-player game, titled 'Surfing on a Sine Wave', in which the player uses a piano to control the frequency with which a character oscillates vertically. A MIDI keyboard will be used for input, and outputs will include a VGA monitor and audio. The game state machine will respond to frequency data from the keyboard, as well as internal status checks such as hit detection. Frequency data will also be used to display a sine (or more complex) waveform in the background of the game. Audio output will correspond to the input frequency and may involve canned or state machine-composed music as well. Players will attempt to pick up coins to gain a high score and, if time permits, will also need to avoid enemies and search for powerups. The ultimate goal of this project is to produce a fun, engaging, and visually pleasing game with an unconventional control scheme.

2. Design

Surfing on a Sine Wave is a side-scroller game in which the player uses a MIDI keyboard to control the landscape traversed by a sprite. The sprite moves 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.

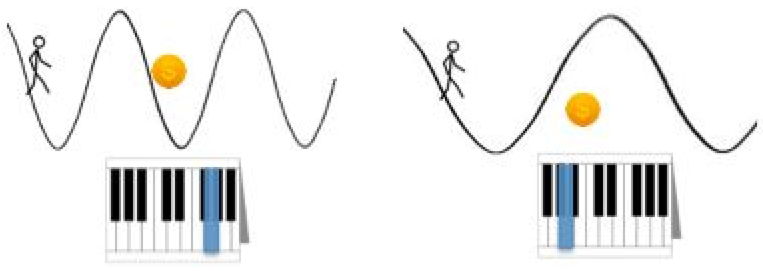

Figure 1: Game appearance mock-up.

As the game progresses, the landscape scrolls increasingly quickly, requiring faster response times from the user. This gameplay mechanism requires an element of strategy different from popular side-scrolling games like Super Mario Bros. where the player controls the motions of the character in reaction to changes in the game landscape. If time allows, the game will also include such bells and whistles as musical accompaniment, power-ups, and different gameplay modes.

The game will display a start screen upon powering up. The player will begin playing at his or her command. The score is counted and displayed on screen. When the character dies after hitting an enemy, the state of the game resets to the startup screen.

3. Implementation

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.

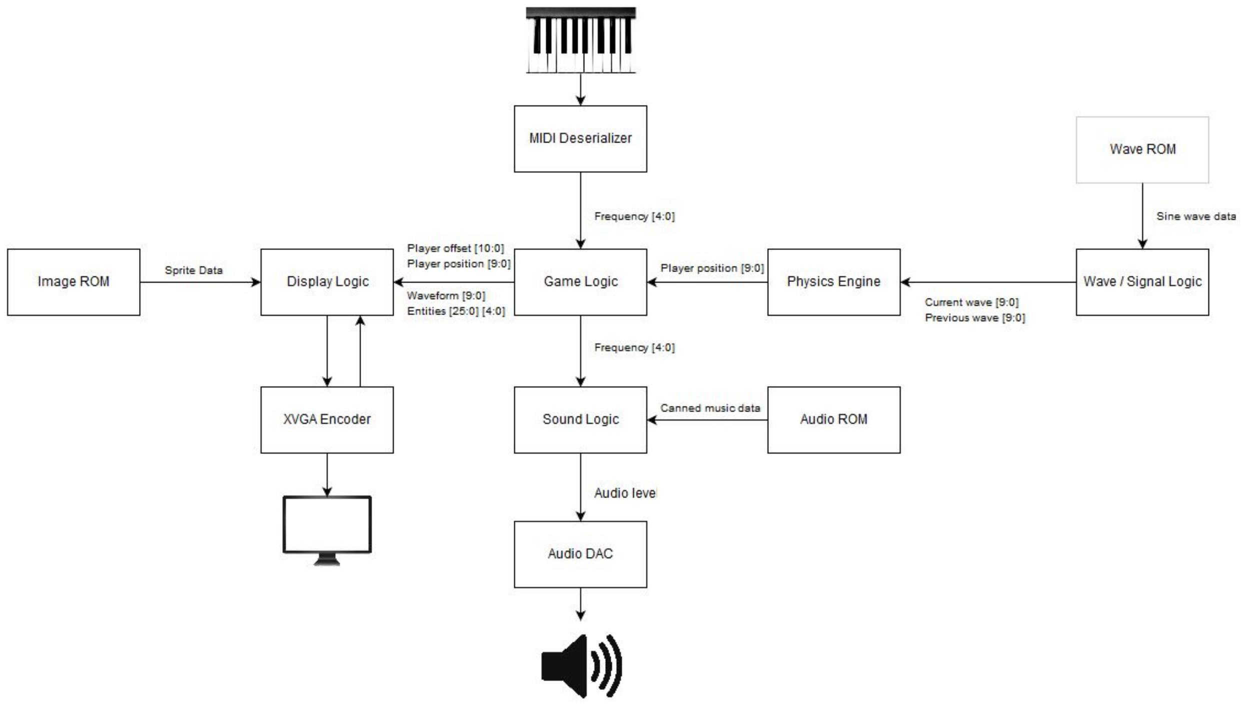

Figure 2: Block diagram; modules described in detail below.

3.1 Game Logic

This central module serves as an interconnect for all the various modules, synthesizing information from the keyboard input module, physics engine, wave engine, and sound engine and issuing information to the display and sound modules. This design strategy ensures modularity. Specifically, the game state machine is responsible for interpreting and communicating information about the game landscape, keeping track of the positions of sprites, the generation of game objects, and the overall state of the game.

3.2 MIDI Deserialization

This module interfaces with the MIDI keyboard used by the player to control gameplay. The keyboard will use the MIDI protocol to communicate with the Nexys 4 FPGA board through an opto-coupling breakout circuit to prevent damage to the FPGA. The circuit sends the current from the MIDI cable through a light emitting diode. Inside the chip, a phototransistor connected to the power line through a pull-up resistor opens and closes according to the light emitted from the diode. This isolates the FPGA pins from potentially damaging voltage bursts and electromagnetic interference.

Inside the FPGA, the MIDI module is responsible for deserializing and synchronizing the processed MIDI signal and outputting an index corresponding to the frequency to the game logic module. The keyboard input module will be responsible for parsing inputs that do not conform to the game rules like simultaneous keypresses.

3.3 Physics Engine

The function of the physics engine is to translate the full waveforms created by the wave logic module into a path that the player will follow. Part of this is to ensure a smooth transition between different frequency waveforms; this will be effected with a blending coefficient that decays exponentially.

The physics engine will receive the two waveforms from the wave logic module and a player offset from the game logic module; it will output (to the game logic module) the player sprite's current vertical position (10 bits), where the top of the screen will have the 0 value.

3.4 Wave / Signal Logic

The wave logic module's function is to calculate the vertical profile of the player's projected path (a 1024-position array with an 10-bit signed value at each position). The center of the screen will have the 0 value. Two profiles (the current and previous frequencies) will be calculated and output to the physics engine and game logic. This module receives a frequency from the game logic module; this will likely be a 5-bit value, but may vary depending on the number of input frequencies.

Two options are being considered for this module's function. The first is to create a 256x10-bit ROM containing the values of sin(x*pi/512) for x from 0 to 255. Reading from this memory, the module can easily and quickly fill in the profiles for any frequency. The second is to utilize a Taylor series to calculate the profiles. The first provides simplicity and rapid calculation for the case of a simple sine wave. The second is a more interesting solution and could make the calculation of different types of waveforms (triangle wave, etc) easier; with the first solution, more values would need to be read from memory, but with the second, only the coefficients of the Taylor series would change. The initial implementation will use a ROM to allow for early testing of other modules; if we reach the point of implementing several different kinds of powerups, Taylor series calculation will be substituted at that point.

3.5 Sound Logic

The sound logic module will control the sound output of the game. Multiple options are being considered for this module. A basic output for this module could be a square wave corresponding to the current frequency input. A more complex option would be to play music from memory; the most fancy option, given time, would be to compose simple music (such as counterpoint or four-voice) based on the current frequency using a state machine. This module will take in the current frequency and output an audio level to the DAC.

3.6 Audio DAC

The DAC module will convert the audio levels it receives from the sound logic module to an audio signal that will go to the headphones or speaker using PWM.

3.7 Display Logic

The display logic module will receive information on the game state from the central game logic module and output one pixel at a time. The number of entities will correspond to the maximum number of collectables and/or enemies that may be on the screen at one time. The basic version of this module will produce a background of two solid colors, divided by the sine wave, and render sprites; potential bells and whistles include loading images that slowly move in the background and/or noise (such as perlin or 1/f noise) that represent a cloudy sky. Introducing these features might require additional modules for noise generation, etc. This module will run on a 65MHz clock in order to display at 60fps and will work with a XVGA module that produces the necessary count and sync signals for encoding.

4. Testing

We plan to build the system from the outside in to facilitate the testing process. Testing will occur in stages, as each module is completed. We are starting with the most important peripheral modules, the keyboard and the display modules. These modules will be connected during early stages of testing with a primitive game state machine, which will grow in complexity to accommodate the modules developed later. Many stages of testing will involve producing some form of VGA output. Initial display testing will require rendering an animated sprite both standing still at the left of the screen and moving left across the screen. Waveform generation will be tested by rendering the sine wave background for several input frequencies. For the testing of the physics module, a sprite (or a square) will switch between frequencies and be shown to follow a reasonable path. Audio output will be tested in stages, and used for testing MIDI input; for this, square wave output will be used. Later features, as well as bells and whistles, will be tested in their planned places, as the framework of the game should be reliable at that point. Early tests of game logic and FSMs will be controlled by the buttons on the Nexys 4; later on, standard playtesting will be used.

5. Timeline

Week ending with	Modules	Tested and working (Sam)	Tested and working (Valerie)	Writing
11/7	Display/VGA; MIDI deserialization.	Begin writing game state machine Verilog.	Character sprite displays; background and collectable proof of concept.	Proposal draft; block diagram draft; proposal presentation draft.
11/14	Wave logic, MIDI deserialization.	Frequencies read correctly from keyboard.	Display shows a wave profile for several test frequencies; this profile moves left smoothly.	Give proposal presentation, revise project proposal.
11/21	Physics engine, audio modules, game logic.	Collectables move smoothly along the screen; player and display are controlled by keyboard.	Player character moves smoothly between frequencies. Audio outputs frequency tones.	None.
11/28	Display and audio bells and whistles, game logic.	Game logic interfaces with audio; FSM functional.	Goals for this week to be determined dependent on what seems achievable. (Bells and whistles!)	Project status report, start on final report.
12/5	Bug fixes / stretch room.	(...)	(...)	Final report.

6. Resources

The required hardware for this project is fairly minimal; resources will include a MIDI keyboard (borrowed from Gim), the MIDI breakout circuit built on a breadboard, MIDI cables and adapters, a monitor, and the Nexys 4.

7. Conclusion

In summary, our project will be a video game where the goal is to collect coins and avoid enemies. Single-keystroke input from a MIDI keyboard will control the oscillation rate of the on-screen character. This character will remain at the left side of the screen, and move upwards and downwards at the input frequency. Collectables and enemies will move left, towards the character, continuously. Prominent challenges include the real-time calculation of waveforms in response to input frequencies and the difficulty of making the game not only playable, but enjoyable and aesthetically pleasing. Our goal is to create a game that is complex, functional, and demonstrates our ability with the FPGA, but is also fun for players.