Laser Conductor

James Noraky and Scott Skirlo

Introduction

After a long week of research, most MIT graduate students like to unwind by playing video games. To feel less guilty about being sedentary all week, some, like the authors of this proposal, prefer to play active games. However, because of the low graduate stipends, most students cannot afford game consoles like the Wii or Xbox, yet alone the accessories necessary to play active games. In this proposal, we describe a plan to democratize active gameplay! We detail how graduate students with access to the 6.111 labkit, a NTSC video camera, a laser galvanometer, and neon green gloves can create a game called *Laser Conductor*.

Laser Conductor is an interactive game where the player assumes the role of a conductor. With the gloves as the baton, the player brandishes different gestures as the music of a symphony plays. The player is directed by arrows drawn by the laser galvanometer, which indicates how the player should move their hands. Figure 1 shows an example of a gesture sequence drawn by the laser galvanometer that the player must imitate. To mimic the expressiveness and energy of a conductor, the gestures will vary in speed and direction corresponding to the the music. The player's gestures are then recognized and scored using visual input from the NTSC video camera. If the player accumulates enough points, the player can then improvise and truly conduct unique music.

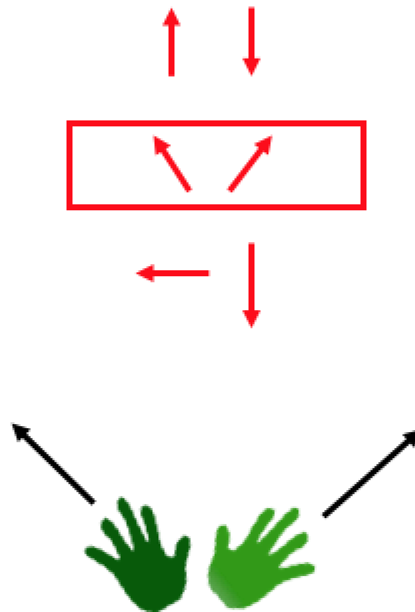

Figure 1: The laser galvanometer draws the current gesture as indicated by the square and the player moves his or her hand in the direction indicated by the arrows.

This proposal is organized as follows. First, we will describe the overall system architecture of the game. Then, we will describe in greater detail the required hardware that we need to control the laser galvanometer as well as the algorithms required to detect gestures from the NTSC camera. In each section, we will indicate who will be in charge and conclude with a description of the challenges and pitfalls we foresee.

System Architecture

Figure 2 describes the high-level architecture of the game. The game takes as input a sequence of pixels from the NTSC camera and outputs control signals for the laser the laser galvanometer and the driver signal for the speaker.

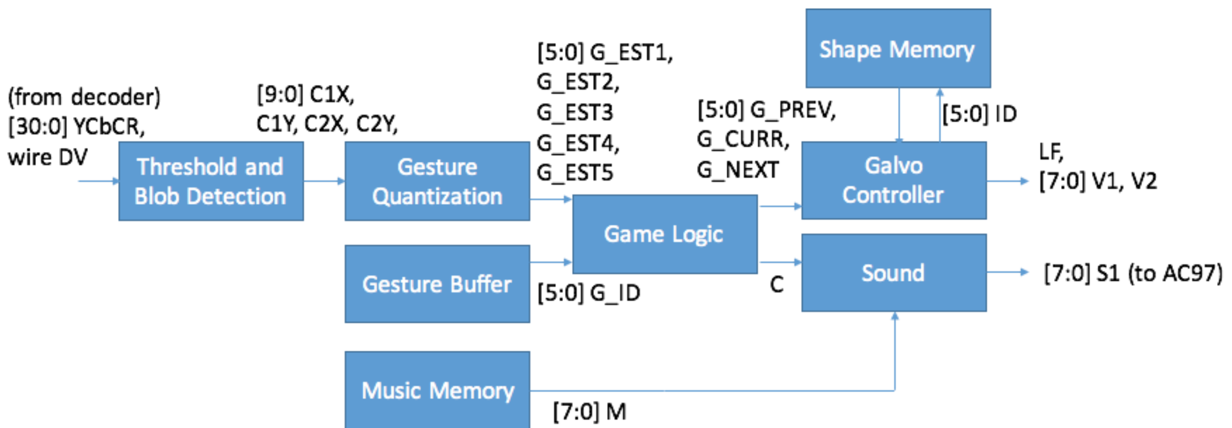

Figure 2: System Architecture of the laser conductor game

Each block in the system is clocked by the system clock. The inputs and outputs are as follows:

- **YCbCR** – This is the 30 bit value of YCbCr value from the NTSC camera.
- **DV** – This is the data valid bit to indicate when the current pixel is valid.
- **C1X, C1Y, C2X, C2Y** – These are the centroid coordinates of the left glove and the right glove, respectively. This will be computed with respect to the camera coordinates.
- **G_EST1, G_EST2, G_EST3, G_EST4, G_EST5** - These are the quantized gesture indices for different time scales. “1” refers to the shortest interval whereas “5” refers to a longest interval.
- **G_ID, G_PREV, G_CURR, G_NEXT, ID** – This is the directed gesture index that the player has to perform.

- **LF** – This flag indicates if the laser is turned on. This feature allows for discontinuous shapes.
- **V1, V2** – The voltage values that are used to control the galvanometer using a DAC.
- **S1** – This is an audio signal that is used to drive a speaker to play music as well as different sound effects to indicate when certain moves are performed correctly.
- **M** – This is the values for the music waveform that is fed to the speaker
- **C** – This flag indicates if sound effects should be overlayed with the music if a move is performed incorrectly.

Due to lack of space, these wires are not depicted in Figure 2.

- **WW** - Write Wait , Flag from game logic FSM which indicates that it wants to write new shapes to the shape memory. When galvo controller finishes reading, it waits for this flag to be deasserted until starting to read again.
- **FR** - Finished Read, The finished read flag indicates to the game logic that the galvo controller FSM is no longer reading the shape buffer and that the shape buffer can be safely modified without putting the galvo controller FSM into an ill-defined state.
- **GS** - Game start, Starts the game
- **GR** - Game reset, Resets the game

Laser Galvanometer System and Shape Memory FSM (Scott Skirlo)

The laser galvanometer system consists of a set of DACs, galvanometers, and a MOSFET for switching the 5 mw red laser (eyesafe!). The DACs are driven by a set of digital outputs from the labkit. In turn these digital outputs are determined by a FSM which runs through the shape memory DRAM. The DRAM will be formatted to contain a list of x,y coordinates (2, 8 bit numbers), and the shape number. The shape number will start at the total number of vertices and be decremented with every successive vertex. The galvo controller FSM will run through the vertices of a given shape until the shape number reaches 1. For simplicity of the galvo-controller FSM, the shapes will be specified in a closed loop, such that the last point listed is the last point plotted. After completing one shape, the LF (Laser on flag) signal will be deasserted until the next listed point is reached for the next shape. The galvo controller FSM will finish it's task once it sees that the shape number is listed as "0". Upon triggering this condition, the galvo FSM will check to see if the game logic needs to update the shape buffer (with the write_Wait (WW) signal), and then pause it's looping so that the buffer can be updated. If we did not do this, the shape memory could be updated while the galvo controller FSM was traversing the shape buffer, potentially trapping the galvo controller FSM in an invalid state. In addition to the write_Wait signal, the galvo controller FSM will indicate to the game_logic FSM that it has finished a given read cycle with the finished_Read signal. This finished_Read signal will be used to trigger the game logic to update the shape memory after the last read.

The DACs will be AD558's available in the lab, whose output can be swept between 0 and +10 V with 8 bit precision. The galvo controller call for a differential voltage between two signal lines

of +10 V or -10 V, that is +5 V=sig1, -5V=sig2 to -5V=sig1, and +5V=sig2. We will obtain this full range by inserting an opamp in a differential amplifier arrangement (with unity gain), with one port connected to a voltage divider yielding +5 V. This will produce an output swinging from -5 V to +5 V. These outputs will be in turn connected to a buffer an inverting buffer to produce a 10 V to -10 V differential input (20 V total). This will allow the full range of the galvo to be accessed. A simple N-mosfet will be used to switch the laser diode in the "low-side switch" configuration.

Game Engine (Scott Skirlo)

The game engine FSM will run the entire system and determine decisions based on the blob detector and gesture recognition FSMs, and in turn control the injection of additional "correct" or "in-correct" sounds on top of the regular music and update the shape buffer. First and foremost when the game is started (GS enabled), the game will progress as normally. When GS is disabled the game will pause and the music will stop playing. When GR is pressed, the game will reset.

The way the game engine will work is as follows. The music will be segmented by hand (or automatically with a matlab script) into intervals associated with a particular set of hand gestures. These intervals will in general be variable time, so the game can start off easy at first and get progressively harder throughout the song. A song with this type of quality may be selected to match the game. When the game starts the song will start playing and a timer will be started counting down the seconds associated with a given gesture in the gesture buffer. The gesture buffer will consist of a set of gestures (0-7 associated with 8 moves), and times (1 to 8 seconds). If the gesture recognition FSM generates the appropriate gesture before the timer goes off, it plays a "victory sound", 1 second long. If the end of the period is reached without recognizing the appropriate gesture, a second failure sound is played. The "victory" or "failure" sounds are added to the song data before it is played by adding a low frequency or high frequency square wave to the signal (or something more elaborate at a later time). After the timer goes off, the next timer is started, and the next gesture in the gesture buffer is searched for in the gesture FSM output.

After a given interval is completed the game engine FSM loads the current set of desired gestures (as arrows), and the next two or three desired gestures after the current gesture into the shape memory. In addition the current set of desired gestures is bounded by a rectangular box. Each of the 8 possible arrows will be specified by a set of points hard coded into the game engine FSM.

Threshold and Blob Detection FSM (James Noraky)

The input of the camera will be green filtered to highlight the green hands the "conductor" will use. The threshold FSM works by applying taking the YCbCr value for each pixel, converting it into its green channel value and thresholding that value. Given the YCbCr values, we can compute the value of the green channel as follows:

$$G = \frac{298}{256}Y - \frac{100}{256}Cb - \frac{208}{256}Cr + 136$$

If the green signal at a given pixel exceeds a certain threshold, it will be represented as a pixel in the binary image. To mitigate the effect of noise that arises due to thresholding, we will apply dilation filters to remove regions that are also green but are not part of the hands. To fill in the gaps that may appear on the hands (e.g), we will also perform a closing. Given the need to set threshold and choose structuring elements, one challenge is to choose the parameters correctly. To do this, we will collect data from the labkit using a serial connection and then experiment with different threshold and structuring elements.

To identify distinct blobs, we will perform integral projections along the horizontal and vertical axis. This will require us to be clever with our addressing so that we can efficiently accumulate the sums. From the integral projections, we can identify the start and end of the bounding box for each hand. From this data, we can then compute the centroids of both hands. As mentioned above, the challenge here will be to reject noise to create a clean binary image. Once we have created a good binary image, we can even get creative by compressing the data stream using run-length encoding and processing that stream directly.

Gesture Quantization FSM (James Noraky)

The gesture recognition FSM works by analyzing the trajectory of each hand by tracking the centroid position. For each hand, we will store a buffer of centroid positions. From these positions, we can compute the slope as follows assuming that the coordinates are centered:

$$m = \frac{\sum_{i=1}^{N} x_i y_i}{\sum_{i=1}^{N} x_i^2}$$

From this slope, we can quantize into 8 separate directions: left, right, up , down, up-right, up-left, down-right, down-left. If there is not enough motion, or one that cannot be categorized, there will be a “no movement” flag asserted. We will implement this logic for different buffer sizes to get the gesture over different time periods. This will be allow us to differentiate between “small” and “large” gestures. Furthermore, during composing mode, the size of the gesture will modify the pitch of the sound. One challenge of this approach is to simplify the computation of the slope. Currently, we are dividing by the sum of squares of the x coordinate. To simplify this, we will explore either simplifications or lookup table approaches. This will introduce errors that we must quantify in a Matlab or Python simulation.

Music Memory and Sound FSM (Scott Skirlo and James Noraky)

The music Memory FSM will store the song for the game, and will start and stop playing according to the start, pause, and reset states of the game logic. Our work here will be generating the gesture sequences that correspond to the music. To do this, we will write a script using MATLAB to generate the gestures.