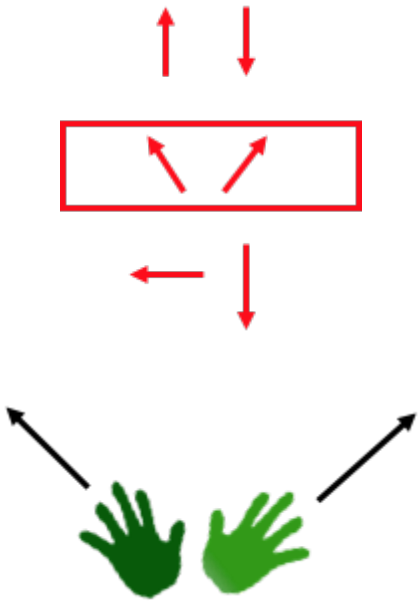
Laser Conductor

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. 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. The figure below shows an example of a gesture sequence drawn by the laser galvanometer that the player must imitate. The player's gestures are then recognized and scored using visual input from the NTSC video camera.



Laser Conductor: 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 report is organized as follows. First, we will describe gesture recognition module. Then, we will describe the memory module. This is followed by the game logic finite state machine. This report concludes with a discussion of the galvanometer control module.

Gesture Recognition

James Noraky

In Laser Conductor, the users play as a conductor and move their hands in directions drawn by the laser galvanometer. A key part of the game is to recognize the user's gestures and compare them against the expected gestures. In this section, we describe how basic gesture recognition is accomplished. The gesture recognition subsystem takes as input images from a camera and outputs, if any, the direction that each hand is moving. Figure 1 shows the key components of this subsystem.

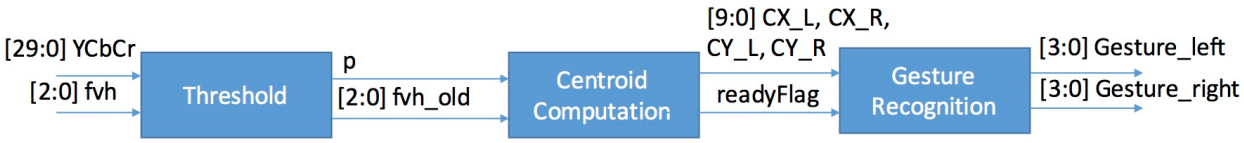


Figure 1: Gesture recognition system with the relevant signals indicated

First, we will briefly describe how the camera works. Then, we describe how the user's hands are first localized. Finally, we will describe how the locations of the hands are translated into gestures. These gestures are fed into the game logic finite state machine.

Interfacing with the Camera

To interface the lab kit with the camera, we use a NTSC camera. This camera outputs an analog signal which encodes the YCbCr color components and the control signals *f, v*, and *h*. As opposed to the raster scan operation of VGA monitors, the frames of a NTSC camera are interlaced. Consequently, there are two fields of lines that correspond to the even and odd lines of the image. This is signaled by the *f* bit. Similar to VGA monitors, the *v* and *h* signals are the vertical and horizontal sync signals. The code that converts the analog signal into a digital form and that obtains the *f*, *v*, and *h* signals are provided in [1].

Hand Localization

To localize their hands, the users wear neon green gloves shown in Figure 2. This color is chosen because it is uncommon in the lab area and would minimize any sources of potential interference. To localize the hands, we first apply a threshold to the image obtained from the NTSC camera to obtain a binary image where '1' represents pixels that are neon green.

The NTSC camera outputs pixel color in the YCbCr color space, where 10 bits are allocated for each component. One approach to identifying the green pixels would be to convert the YCbCr color into the HSV color space, where a threshold can be applied to the hue value. Furthermore, this approach is also appealing because the code to convert YCbCr to RGB and to convert RGB to HSV are both provided [2]. Our approach instead first converts the YCbCr color of each pixel into RGB. We then threshold the green value of each pixel to obtain the binary image. Because we are not converting to HSV, we minimize the latency between the time a pixel is read to when a binary value is outputted. This approach is effective in localizing the gloves as shown in Figure 2. From this binary image, we will approximate center of each glove and track its trajectory to identify the correct gesture.

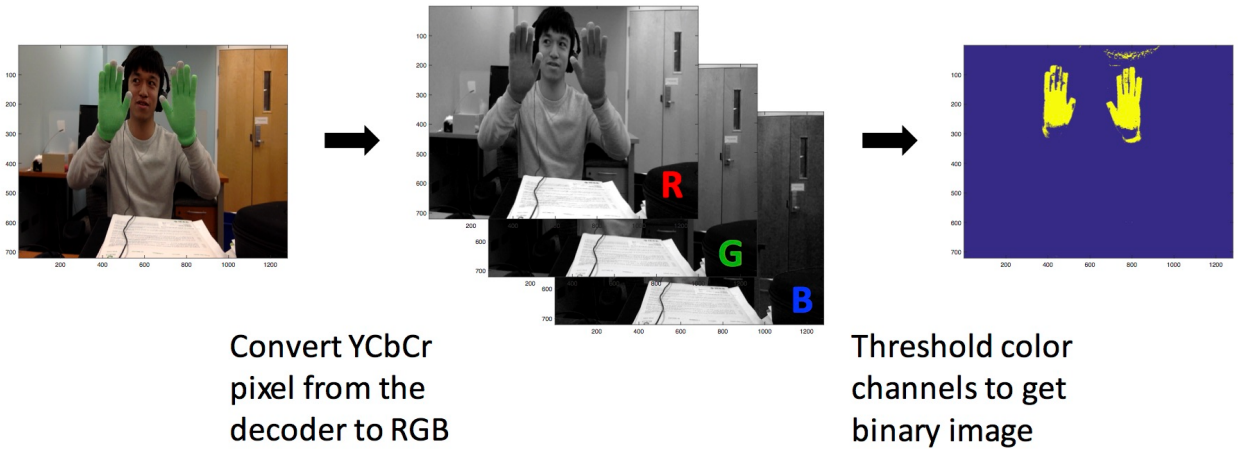


Figure 2: Pipeline that converts the YCbCR into a binary image that localizes the green gloves, from which the centroids of the gloves are then computed.

It should be noted that applying a threshold based on a minimum value alone ($G > G_{thresh}$) is not effective. This approach would produce a binary image that includes both the green gloves and any bright light sources because white light has significant values for all three RGB color channels. Here, we realize that we must also impose an upper bound for the green channel ($G>G_{min}$ && $G<G_{max}$), which accounts for the fact that the green glove does not reflect all of the incident light. With this insight, we were able to robustly localize the gloves. In addition to green, we found that this approach is also effective for identifying blue pixels.

Once we obtain a binary image, our task is to then localize the position of the gloves. Because there will be some spurious noise, we can consider implementing median filters or morphological erosions. However, given the large size of the hands in the binary image, the computation of the centroid is robust to the small amount of noise. Another design choice was whether to segment the gloves. This can be done use some integral projections to figure the axis which separates the images of the two gloves. However, one reasonable assumption is to use the center of the image to separate the two gloves. This simplification leads to robust centroid tracking.

To compute the centroid location in real time, we have as input: a binary pixel obtained after applying the threshold; *f* which denotes the frame number; *v* which is the vsync signal; and *h* which is the hsync signal. Using the *v* and *h* signal, we can compute the x and y coordinate for each pixel. Denoting *p* as the pixel, the centroid location for the left glove is then:

$$(x_{left}, y_{left}) = \frac{\sum_{x<L} p_{x,y} \cdot (x, y)}{\sum p_{x,y}}$$

An analogous computation is made for the right glove. Since we are computing the location in real-time, the numerator and denominator would therefore be partial sums. To compute the centroid, we need to divide. Given the size of the NTSC camera, we can choose an appropriate size register bit width for the partial sum variables. This decision is important because dividers introduce latency to the computation of the centroid coordinate. The latency of a restoring divider is equal to the bit width of its arguments [3]. However, since we simply need to appear real-time, we do not need to be very precise because the player cannot perceive a few cycles of difference in a 27 MHz clock. To start the divider, we make use of the *f* signal to start the division on the transition from an odd to even frame. To figure out this signal, it was helpful to use the logic analyzer to see what signals would be reliable to start the division.

To compute the centroid locations for both the left and right glove (denoted *CX_L, CY_L, CX_R, CY_R*), we use 4 restoring dividers that run in parallel. Because of the latency and potential for difference among the dividers, we need to compute a ready signal that takes into account the status of each divider. This step is important so that spurious coordinates are not passed into the gesture recognition subsystem, which we describe next.

Gesture Recognition

In the game, the laser galvanometer draws different arrows which represent the trajectories in which the user must move their hands. We will refer to these hand trajectories as gestures. In our implementation, we support 8 directions as shown below.

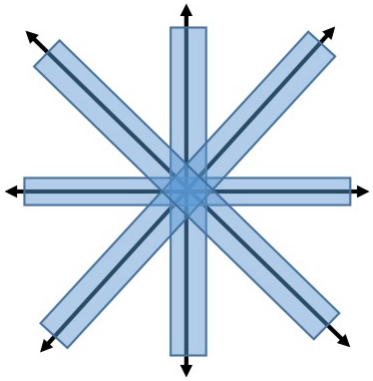


Figure 3: The directions supported by the Laser Conductor game. The blue region represents the margin of error that the centroids are allowed to still be classified correctly.

In the diagram, the blue regions represent the margin in which the trajectories need to be in to be correctly categorized. If the trajectories do not fall in the blue region, then it is classified as an unknown gesture.

Trajectories are computed by maintaining the last 3 centroid positions. A new centroid position is loaded in when the ready signal is asserted from the hand localization subsystem. From these three centroid positions, we can create 2 vectors by subtracting the most recent centroids from the earliest centroid. We can denote these vectors as *(dx, dy)*. Then, we check check if these vectors are contained within the blue region as shown in the diagram. For example, to check if the gloves are moving up, we have to make sure the the *dx* position is within the bounds and the *dy* position keep increasing. For the diagonal directions, we must check the difference between *dx* and *dy* as well as the direction.

We repeat this process for both the left and right glove to output a gesture. This gesture is updated for each new centroid position. These gestures are then fed into the game logic finite state machine.

Future Work

If we had more time, a natural extension would be to detect skin color directly. If the camera sensor has a good color response, skin color can be detected directly without the need of green gloves. For example, the webcam of most Apple laptops have such characteristics. To do this, we can use another NTSC camera or interface a webcam with the lab kit. To interface a webcam with the lab kit, we will describe how large amounts of data can be transferred using the FTDI USB-to-FIFO chip when we describe the music of the game.

Furthermore, we can further increase the robustness of the centroid tracking. To do this, we would use integral projections to localize the centroid of each hand without assuming the central dividing axis. Coupled with a trajectory "smoother," we can also get more robust results. One such approach would involve a Kalman filter.

References

[1] NTSC ZBT Sample Code. http://web.mit.edu/6.111/www/f2016/tools/ntsc_zbtfix.zip

[2] Conversion from YCbCr to RGB. http://web.mit.edu/6.111/www/f2011/tools/ycrcb2rgb.v

[3] Restoring Divider. http://web.mit.edu/6.111/www/f2016/index.html

Music Manager

James Noraky

A key part of the Laser Conductor game play is the music since the user is playing as a conductor. Furthermore, if the user performs a gesture correctly, a chime should indicate that the move is correct. To handle the various music and sounds that will be played, we implemented a music manager that stores and plays both a song and the chime for a correct move. The input of this module are flags from the game logic FSM: (1) a flag that indicates the start of the game and music and (2) a flag that indicates a correct gesture and the start of a chime.

Due to the different durations of the song and the chime, we store the song in the ZBT memory and we store the chime sound in BRAM. Both the song and the chime are uploaded and stored on the lab kit after it has been programed.

Loading the Song onto the Lab Kit

The architecture of the music manager is similar to the recorder in Lab 5. In the recorder lab, we took the microphone data from the AC97 chip, applied a low-pass filter, and stored a decimated version of the filtered signal. One approach to load data onto the lab kit is to connect the headphone output of a computer to the microphone input of the lab kit and store the data like in Lab 5. The benefits of this approach is that it minimizes the complexity of the ZBT timing. However, this means that loading in a song after the lab kit has been programmed can take several minutes.

To minimize the load time, we use the FTDI USB-to-FIFO chip to load data from a computer to the lab kit. To transmit data from the computer, we modified the provided code [1] so that it can compile on the OSX operating system and send larger data packets. We also used the supplied Verilog code to access the data and store it in memory. We then wrote a MATLAB script to convert any arbitrary MP3 file into a format that can be loaded onto the FIFO using our modified C++ code. With these modifications, we can load several minutes worth of music in approximately 10 seconds.

In our implementation, we also bypass the filtering step of the sound input and load the filtered data directly onto the lab kit. During playback, we would apply the same filter to interpolate the signal into 48 kHz. Because of the high transfer rate, the timing for the ZBT memory becomes important. To make sure the ZBT memory is properly functioning, we used the provided the ramclock module [2] to correctly use the ZBT memory. To fully utilize the memory, we also wrote a simple memory manager to fully utilize 32 out of the 36 bits of each ZBT memory row. By doing this, we can store more than a 5-minute song in the ZBT memory.

Loading the Chime onto the Lab Kit

We used a similar approach to load the chime except that instead of ZBT, the chime is stored in the BRAM. In playback mode, when a gesture is played correctly, the chime is overlaid with the music to provide audio feedback to the player.

Putting it All Together

Instead of the record button of Lab 5, we wired the different switches to load in music for the song and for the chime. This is performed after the lab kit has been programed. When both the sounds are loaded, the recorder enters playback mode when the start signal is asserted by the game logic finite state machine.

One potential question is: why did we not store the data in flash? We stored the data in ZBT and BRAM for the possibility of the user being able to change the music instead of using some preprogrammed song. As a future feature, we could include a simple algorithm that converts the loaded music into the different gestures for dynamic game play, allowing our game to accommodate different musical tastes.

References

[1] Data transfer. http://web.mit.edu/6.111/www/f2016/tools/flash_IO.zip

[2] Ramclock module. http://web.mit.edu/6.111/www/f2016/index.html

Game Logic Module

Scott Skirlo

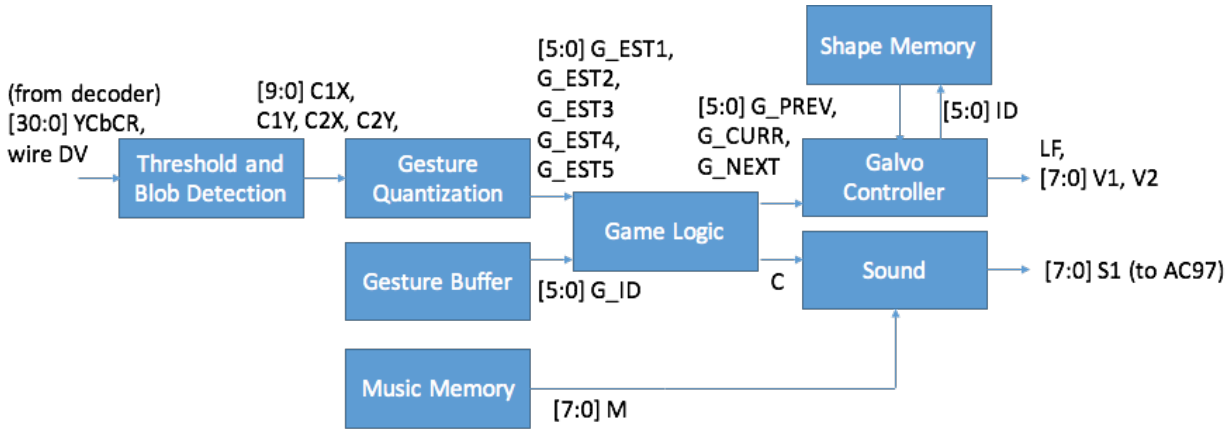


Fig. Schematic diagram of system

I focused on developing the game logic module, galvo_control module and affiliated modules associated with gesture and vector graphics storage.

The toughest link for me in the entire project was figuring out best how to flexibly interface shape retrieval and drawing in a flexible way. Originally I envisioned rewriting a shape RAM every single frame. This RAM would contain a set of coordinates for the galvo to step through and the number of remaining steps in the shape before completion. Upon realizing that one shape was drawn the galvo would go to the next memory address in the RAM and proceed to plot the coordinates for that shape, going until it reached a "end of file" number indicating that it had completely gone through the RAM frame. Originally the game controller would've directly interfaced with a gesture ROM. As I worked through the problem, I realized that flexibly writing a variable number of shapes, of different types and coordinate offsets would be very challenging. Because of this I tried to simplify the information being passed between the game FSM and the galvo controller as much as possible, reducing this only to a shape type and coordinate offset types.

Operation then proceeded as follows. For every new frame the game FSM would fill in a buffer containing a variable number of shape types and offsets depending on the user inputs and the current fetched gestures from the gesture ROM. During a "slave" phase, the game FSM would then respond to the galvo controller for requests for the next shape. Upon assertion of the "request_shape" flag, the game FSM would return the current shape and offset set, and increment the shape buffer and gesture offset buffers. With this information, the galvo controller would then request the shape number and the first vertex number from the shape ROM. Upon receiving these coordinates and the remaining vertex number, the galvo controller would add these to the point offsets and set the x_pos and y_pos values to this. After doing this the galvo_controller starts a timer to allow the shape to be drawn and wait for the galvo to mechanically respond to the new voltages. This wait time was determined by the difference between the original x and y coordinates and the new x and y coordinates to allow the laser line

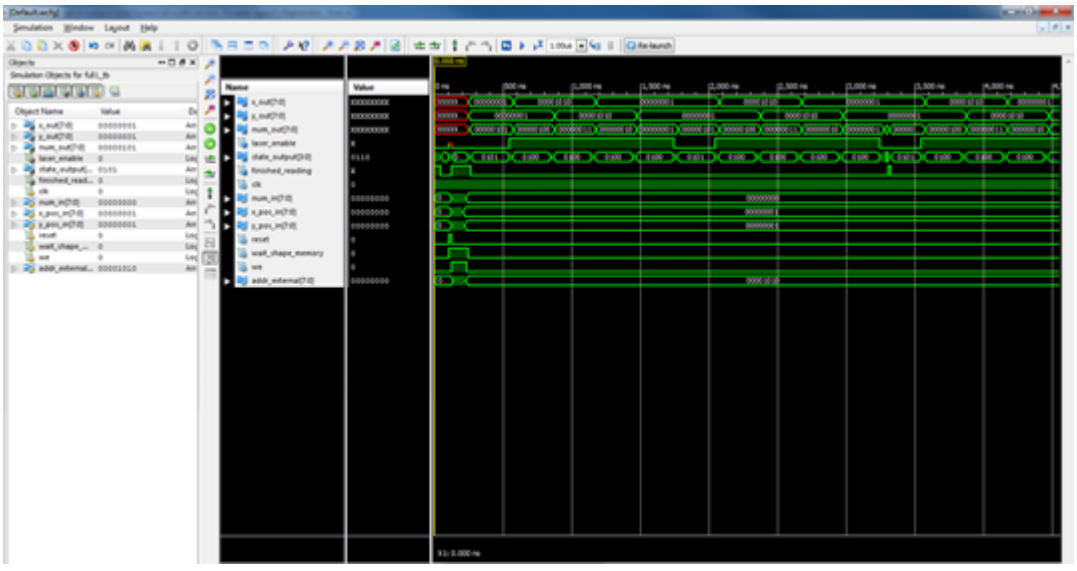
"weight" to be as uniform as possible for the entire vector graphic figure. Upon registering that only one vertex is left from the shapes being drawn, the galvo controller turns the laser off and requests the next shape, if "wait_shape_memory" hasn't been asserted.

"Wait_shape_memory" is asserted by the game_FSM when the final shape in the shape buffer is reached. Upon doing this the game_FSM checks it's own timer to see if a sufficient time has expired to update the next move. If not, it simply loads the shape buffer with the old shapes, and updates the offset coordinate buffers as necessary before returning to slave mode.

Galvo Controller Module

Scott Skirlo

The galvo mirrors were controlled by 2 8-bit DACs from the user output's of the labkit. These in turn are fed to two sets of inputs and inverting inputs. To create the second inverted signal, the DAC output is also fed through an LF-741 in an inverting configuration. The DACs are on a 0 to 5 V range, so the effective differential input voltage of the Galvo controller ranges from 0 to 10 V. There were some small issues with the resistors for setting up the op amp in the inverting configuration. When 1 k resistors were used, we noticed that there tended to be a "ringing" in the DAC output which resulted in a notable fuziness in the screen drawing. Adjusting these to 10 k improved the performance. The addition of filter in a low pass configuration ended up causing oscillation. Originally it was planned that 12 bit serial DACs would be utilized to drive the screen. However, in practice the galvo controller mirrors have so much inertia and mechanical noise, and the fact that the laser spot size is already not so narrow, that in practice having higher resolution is not useful or meaningful.



Laser source

The laser source was obtained cheaply online for a few dollars, and produced a 5 mW red laser output. This red laser was perfectly eye safe at these power levels (class I), so that no extra precautions would be necessary for using the system. Originally we planned on drawing a total of 7 shapes at the same time. Although we had the capacity for this and demonstrated it (see Fig below), for the simplicity of game-play, and to allow the most important arrows to be brighter. At large distances, 6 drawn arrows are barely visible, whereas 2 are quite clear at the distance of 1-10 meters.

There were some plans to update the resolution of the galvo controller to 12 bits.

The galvo had certain interesting quirks associated with it's operation.