INVISIMAZE

A 6.111 Final Project

Stephanie Pavlick ’17 and Elizabeth Zhang ‘16

Fall 2015

INVISMAZE

A 6.111 Final Project

Stephanie Pavlick '17 and Elizabeth Zhang '16

Fall 2015

Table of Contents

1 | Introduction 2

2 | Project Overview 2

System Overview 2

Physical Setup 3

3 | Modules 4

3.1 A Word on Relative and Absolute Coordinates, and the Maze Grid 4

3/2 Game Logic   Libby 5

Finite State Machine 6

Countdowns and Timers 7

Maze Map Selection 8

Virtual versus Live-Action Play 8

3.3 Hex Display and Binary-Code Decimal Converter   Libby 9

3.4 VGA Display   Libby 9

3.5 Object-Recognition and Tracking   Stephanie 10

NTSC Camera Feed to ZBT Memory 10

Color Detection and Center of Mass Calculator 10

Implementation and Testing 11

3.6 Serial Transmitter and Receiver   Steph 14

Testing 15

4 | Improvements and Insights 17

NTSC camera and Color Recognition 17

COM to Maze 17

Transmitter/Receiver 17

Finite State Machine 18

5 | Conclusion 19

1 | Introduction

Invisimaze is an interactive, virtual-reality maze game. The game is developed for a single player who tries to navigate through an open space that is virtually overlaid with an invisible maze. In our specific implementation, Invisimaze is played an indoor location that 5 by 8 feet in dimension.

The player wears two different color gloves to allow the game to track his/her location. Two NTSC cameras are placed above the open maze space and they transfer color data back to the system for analysis. Audio output provides feedback to the player during game play, and timer is automatically started to record how long it took the player to complete the maze.

In addition to the main player's experience, Invisimaze is also well-suited for spectators who can observe and support the player during the game. When the player presses 'Enter' and walks to the designated start area, the maze, which was previously not displayed, appears on the VGA monitor for the spectators to watch the player's progress. Depending on whether the player walks into a wall, or wins/loses the game, the monitor flashes differently to alert the spectators.

Finally, Invisimaze also has a purely virtual playing mode, so that players can also experience the game using the arrow buttons to navigate through the maze because there are space constraints. This way, everyone can have fun playing Invisimaze, no matter where they are!

2 | Project Overview

2.1 System Overview

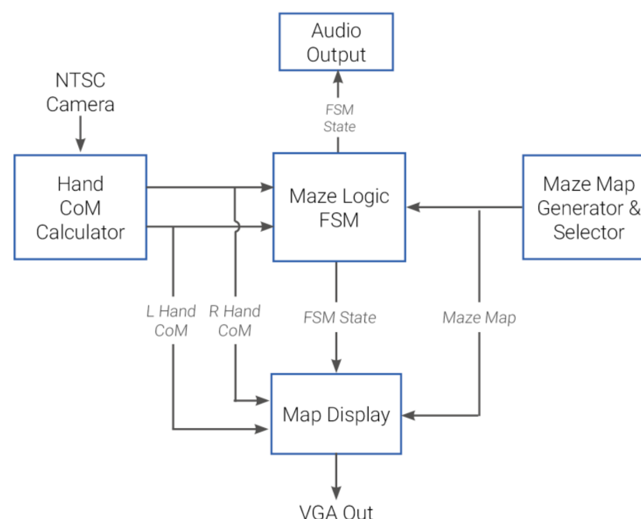

Figure 1 System Overview of the VHDL Implementation

2.2 Physical Setup

To create the game play space, we used two NTSC cameras mounted to the ceiling looking down at the ground. We attempted to set up these cameras such that their fields of view were not overlapping, but they aligned such that the bottom edge of the view of one camera matched as close as possible with the top edge of the view of the second camera. This would give us the maximum game play space available using two cameras, about an 8 foot by 5 foot grid. Each camera would correspond to about half of the maze grid. We also delineated the outline of the play space on the ground in the lab.

Figure 2 Two NTSC Camera Setup. The two cameras were installed in the ceiling using electrical tape, and they were connected to the power sources and Labkits using very long cables.

With such a setup, this introduced discontinuities in the center of mass tracking at the edge between the two cameras. When the object to be tracked was located along the edge common to the two camera's fields of view, the logic had difficulties deciding which camera data to use and where the object was located, as each camera had access to fewer object pixels along this line. Furthermore, it was difficult to get the camera views to align exactly, and we found that some overlap was inevitable.

3 | Modules

3.1 A Word on Relative and Absolute Coordinates, and the Maze Grid

Before delving into the game implementation, a few notes should be made on terminology regarding coordinates and grids. All references to 'relative' and 'raw' coordinates and locations are defined to be the coordinates received from the NTSC cameras – this is because these coordinates are relative to the device performing the tracking in the physical world. Before they can be utilized in the maze game, they must be translated into 'absolute' or 'scaled' coordinates and locations that are universal to the game system.

The absolute coordinates of the game system are determined by the desired size of the maze. In our implementation, we used a predetermined 5 x 8 grid, which maps squarely to the 5 x 8 square foot playing space that we were limited to by the NTSC camera's field of view. Each element within this maze grid has certain associated characteristics, i.e. start area, finish area, path, or wall. Start areas and finish areas are also considered valid locations. Figure 1 provides a graphical description of the maze grid.

The design of the maze grid allows the game to be highly compatible and flexible in terms of "projecting" onto the available physical space. A single grid size can be played in a variety of areas, and alternatively a single area can be host to a variety of grid sizes.

The Maze Map Selection section explains in more depth how coordinates and maze grid indices are calculated, and how the two are used to compute player validity.

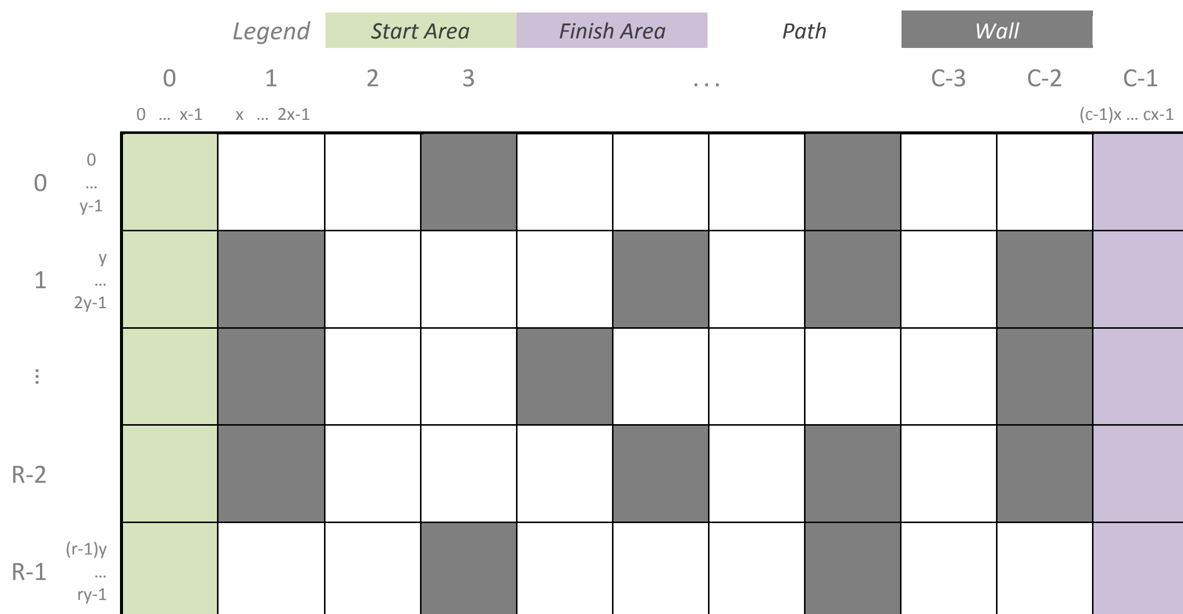

Figure 3 Graphical Description of the Maze Grid. The large numbers are on the outside of the grid (0….C-1) and (0…R-1) are the grid's *indices*. The element at each index has a specific associated characteristic, as depicted by their color fill. The smaller numbers on the outside of the grid (0…x-1, …, (x-1)x … cx -1) are the *absolute coordinates* that map to the specified index. For example, the maze logic module will use each hand CoM's absolute coordinates to determine which element of the grid it is in, and whether that element is a valid or invalid location.

3.2 Game Logic

Libby

The main module controlling the Invisimaze game play is `mapFsm.v`. This module contains a Moore finite state machine with six different states (Figure 2) keeping track of the player's state of play.

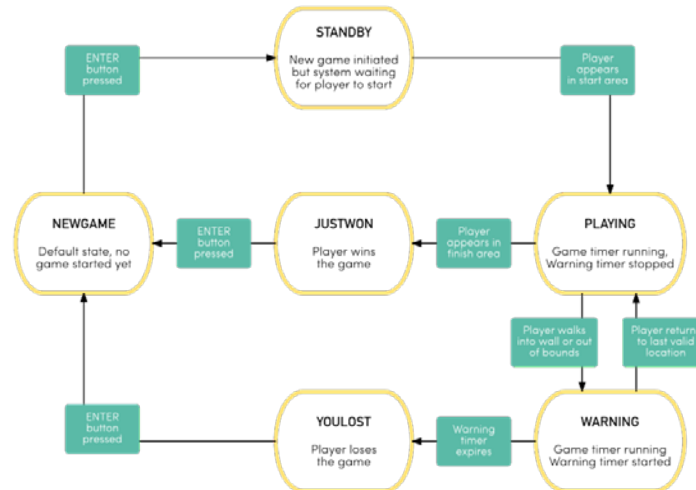

Figure 4 FSM State Diagram. The game's governing finite state machine has six distinct states that are triggered by the associated events shown above.

Each state has a number of associated outputs, including visual and audio feedback. All of these outputs (Table 1) are handled externally by connecting their associated modules, which will be described separately, to the current state of the FSM. The latest implementation has three main types of feedback, namely via the 16-character hex display onboard the Labkit, the analog input of the VGA monitor, and audio, but this design of the system allows the number and types of outputs to be easily extendable.

Table 1 System Outputs Associated with FSM State

	Hex Display	VGA Monitor	Audio
NEWGAME	`'ENTER' to Start`	Black screen, only hand CoM locations displayed	No audio
STANDBY	`Go to start area`	Black screen, only hand CoM locations displayed	"It is your destiny" soundclip played once
PLAYING	`Get goin!     %%%` *%%% is current time (s) since the start of game*	Maze and current hand CoM locations displayed	No audio
WARNING	`Get out! %s left` *% is time remaining in warning countdown*	Maze and CoMs visible, red overlay flashing at a 4 Hz rate	"Don't do dat" soundclip played repeatedly
JUSTWON	`You Won!      %%%` *%%% is total time (sec)*	Maze and CoMs visible, alternating green and blue	"Hasta la vista" soundclip played once
YOULOST	`YOU LOST :(`	Maze and CoMs visible, steady red overlay	"You are terminated" soundclip played once

Finite State Machine [1]

NEWGAME State

In this state, the system is idling and waiting to press the ENTER button on the Labkit to begin.

STANDBY State

The system waits until the player is detected to be in the start area before starting the game. This ensures that the player may proceed to the start area at their own pace after pressing ENTER, and that the player actually starts the game in the start area to avoid any unsportsmanlike playing. The system compares the player's two absolute hand CoM coordinates to the maze grid (see *A Word on Relative and Absolute Coordinates, and* the Maze Grid). When both hands appear in the first column, or some other start area designation, the game begins – the game timer begins, the maze appears on the VGA screen, and the player can proceed to stumble their way through the maze.

PLAYING and WARNING States

These two states are the primary states of play that the player will be in. After the game has started and before it has finished, the player is in the PLAYING state when he/she is in a valid location and in the WARNING state when in an invalid location.

The player's location is taken by averaging the absolute coordinates of his/her two hand CoMs, and then translating those absolute coordinates into a grid index. The validity of location is then primarily determined by comparing this absolute indexed location to the associated element in the maze grid, as described briefly in the STANDBY state. If that element is a path, the current location is valid; if that element is a wall, the current location is invalid and the player enters the WARNING state. Each time the player walks into a valid location, this location index is saved in the lastValidLoc register.

The FSM checks the validity of a player's location by comparing the indexing into the maze map register with the player's absolute location. Recall that the player's location is averaged between the two hand CoM coordinates. The player's location is calculated and compared to the maze map by the following:

Table 2 Verilog Equations for Location Indexing and Validation. sqSize is the binary logarithm of the number of pixels of each grid element. This allows for absolute indexing of each coordinate. Under 'Averaged Location,' the additional right shift is for averaging the two X- and Y-coordinates.

Single Hand CoM

```
wire [2:0] nrx = rx >> sqSize;
wire [2:0] nry = ry >> sqSize;
assign isRValid = mapArr[nrx+nry*8];
```

Averaged Location

```
wire [2:0] nmx = (rx+lx)>>(sqSize+1);
wire [10:0] ysum = ry+ly;
wire [2:0] nmy = ysum>>(sqSize+1);
```

When the game enters the WARNING state, the warning countdown is started. This value is hardcoded to a five second delay, although in future implementations, it could be reprogrammed to increase or decrease the grace time and thus the difficulty of the game. The user has this amount of time to get out of the invalid location.

[1] mapFsm.v

Once the player walks into a wall, the distinction between valid and invalid location is further refined. One major concern that people voiced during the Project Design Presentation was how to prevent players from "walking through" walls, since there are obviously no physical barriers to prevent them from doing so. This concern is addressed by only defining a valid location as one that was directly adjacent to the player's last valid location, provided that said valid location is also a path element (*Figure 3*). If the player walks out of bounds or out of the field of view of the cameras, he/she is also considered in an invalid zone.

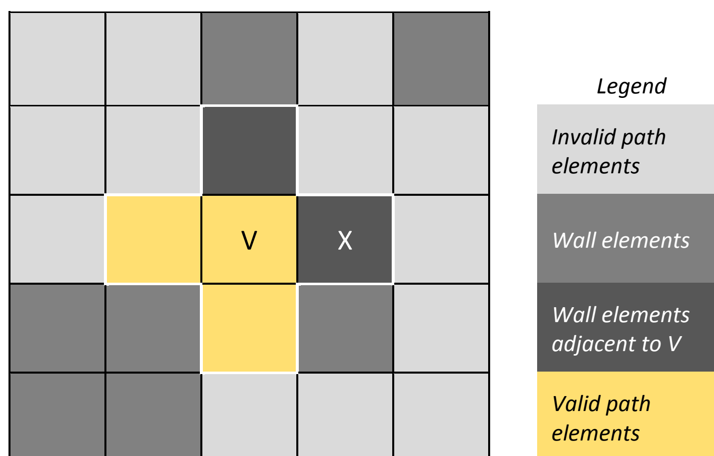

Figure 5 Localization of Valid Locations when in `WARNING` state. This is an arbitrarily selected section of the maze grid. `X` denotes current position, `V` denotes last valid location. The cross outlined in white contains all of the elements directly adjacent to the last valid location. Only the path elements within this cross, otherwise directly adjacent to the last valid location or the last valid location itself, may return the user to the `PLAYING` state. All other path elements are invalid to prevent user from walking through any number of walls.

If the player is able to return to a valid location before the warning countdown expires, they continue playing the game until they reach the finish. Otherwise, the player loses the game.

JUSTWON and YOULOSE States

These two states are the two mutually-exclusive game endings. As mentioned previously, if the player reaches the designated finish area without walking through any walls, then they "win the game." Their total time is held on the hex display so that they can record their time. If the player is unable to return to a valid location after through a wall before the timer expires, he/she enters the `YOULOSE` state and lose the game. Their time is not saved. To return to the idling `NEWGAME` state, the player must press the `ENTER` button.

Countdowns and Timers [2]

There are three different time modules implemented within the game logic subsystem: the 1 Hz clock, the total game timer, and the warning countdown.

The 1 Hz clock is by default generated by a 65 Mhz clock, but it is parameterized so it can be generated from any supplied clock (>1 Hz). In the display module, the parameters are manipulated such that it

[2] `clock_1hz.v, countdown.v, timer.v`

generates 2 Hz and 4 Hz clocks. This clock is also synchronizable so that it starts generating a 1 Hz rate at the same time that its associated timing module starts timing.

The total game timer and the warning countdown each of their own associated 1 Hz clock so that they can count time independently of each other. The total game timer counts up from 0, and it requires a start signal, which is supplied at the transition from `STANDBY` to `PLAYING`, and a stop signal. It will count up to 999, overflow and begin counting again from 0.

The warning countdown module counts down from the supplied time, in this case five seconds. The countdown begins counting anew each time a new `start_timer` signal is supplied. It pulses the `expired` signal high for one clock signal if it finishes counting without being interrupted.

Maze Map Selection[3]

The player can choose which maze he/she would like to play by switching `switch[7:6]`. There are currently four mazes to choose from, but more can be added. Originally, I thought I would need to create a COE file to import the maze, but the size of the maze is small enough that it is more efficient to store in a 40-bit register. The register size is determined by the number of elements in the maze, i.e. the product of the maze width and height.

Each maze map was generate manually by plotting a reasonable maze in Excel, similar to the graphical description of the maze grid in Figure 1, and translating path and wall elements to 1's and 0's, respectively. The most significant bit is element (8, 5), or the lower-right most element; the second most significant bit is element (7, 5); and so on, down to element (0,0), which is the least significant bit.

The maze map can only be changed during the NEWGAME state, before the player gets to see the maze map, so that they cannot get a sneak peak and so that spectators cannot alter the maze course during play.

Virtual versus Live-Action Play[4]

This module allows the flip of a single switch, `switch[3]`, to determine whether the player would like to play virtually (using arrow buttons) or physically (using hands). There is a separate `handController` module for the right and the left hands, each providing a 21-bit value containing the absolute X- and Y-coordinates of the specified hand, irrespective of the manner of playing.

If the switch is high, this module passes through the supplied scaled coordinates calculated from the NTSC camera. Otherwise, it updates and outputs the X- and Y- register values based on debounced arrow button signals. It outputs a 21-bit value which contains the absolute X- and Y-coordinates of the specified hand.

[3] `mapSelect.v`
[4] `handController.v`

3.3 Hex Display and Binary-Code Decimal Converter[5] Libby

The 16-character hex display available on the Labkit provided a digital interface for providing feedback to the player and the spectators. I used the `display_string.v` LED driver module supplied by the staff to display in messages associated with the current FSM state (see Table 1 for a comprehensive list).

For the `PLAYING` and `WARNING` states, I additionally provided the timer and countdown values. This required a binary to binary-converted decimal (BCD) module. I implemented the double-dabble algorithm to convert the values as quickly possible. I additionally allowed the input size of the binary value to be parameterized to allow for fast computation of smaller values. This is because the implemented BCD converter only outputs the converted value when all of the shifts have been completed. The number of shifts required is based on the size of the binary input, so for smaller numbers that are not 10 bits large, they can computed in O(log2) time.

3.4 VGA Display[6] Libby

I integrated all of the visual elements that would output onto the VGA monitor into a single module, and it is additionally controlled by the current state of the FSM. Circle sprites are created for each hand CoM. Due to the computationally intense nature of computing circles (i.e. two multiplications), these calculations are only performed when the `hcount` and `vcount` of the VGA scan are within the bounding box of the circle.

```
// Circle: x^2 + y^2 <= r^2
if (dx > RADIUS || dy > RADIUS) pixel = 0;
else if (dx*dx + dy*dy <= RADIUS * RADIUS) pixel = COLOR;
else pixel = 0;
```

If the player is in the `WARNING`, `YOULOST`, or `JUSTWON` states, then the entire monitor is overlaid with an alpha-blending color layer. The below equation provides an example of alpha-overlaying pure red (255, 0, 0) over the screen.

```
blended <= {(unblended[23:16])*M/N+(RED[23:16])*(N-M)/N,
unblended[15:8]*M/N, unblended[7:0]*M/N};
```

In the top-level module, I also allowed different NTSC camera feeds to be outputted instead of the game-portion of the VGA output, depending on the switch settings. These feeds included the live tracking of the red and green center of masses on top of an unfiltered NTSC feed, and the individual red and green HSV filtered feeds.

[5] `ledDisplay.v, bcd2.v`
[6] `mapDisplay.v`

3.5 Object-Recognition and Tracking

Stephanie

This subsystem has 2 main goals:

1. Perform color detection to analyze which pixels match the colors we would like to track
2. Perform center of mass calculations to determine the location of an object based on the pixels that match its color

It will use the feed from the NTSC camera to determine the object locations and ultimately send these locations to the maze FSM logic to display a person's location within the maze on the screen.

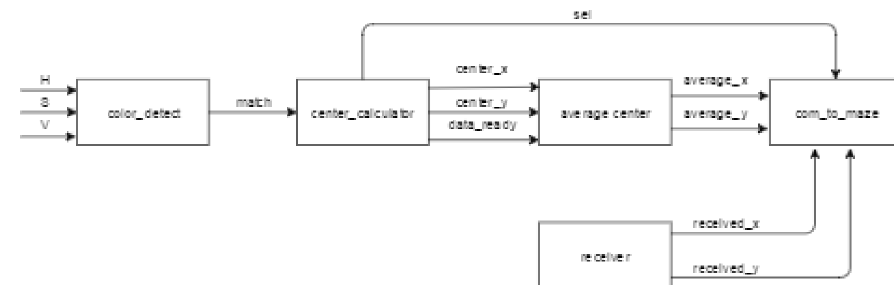

Figure 6 Detailed Block Diagram of Object-Recognition and Tracking Subsystem

NTSC Camera Feed to ZBT Memory

adv7185 and ntsc_decode

These submodules are used together to extract the raw data received by the Labkit from the NTSC camera via the RCA phono jack. As inputs, the ntsc_decode module takes a 27 MHz clock and the tv_in_ycrcb line. It outputs the pixel data in the ycrcb space received from the NTSC camera as well as the field, vertical sync, and horizontal sync lines from the camera. A data_valid line output when the ycrcb data has been fully extracted.

ntsc_to_zbt

This submodule prepares data addresses for writing into the ZBT memory. As inputs, it takes the field, vertical sync, and horizontal synz (fvh) data from the ntsc_decode submodule as well as a data valid line, data to be written to the ZBT memory, system clock, and video clock from the camera. It outputs the address where the data will be written, the data to be written, and a write enable line for the data.

zbt_6111

This submodule writes the NTSC data to the ZBT memory.

vram_display

This submodule generates the pixel to be displayed on the screen by reading from the ZBT memory.

Color Detection and Center of Mass Calculator

YCrCb2RGB

This submodule converts the YCrCb data from the NTSC camera into the RBG color space.

rgb2hsv and divider

The rgb2hsv submodule converts the data in the RGB color space into the HSV color space to enable color detection using HSV values. The HSV converter must use dividing to calculate the values, so it also requires create a divider using an IP core in ISE.

color_detector

This submodule analyzes the hue, saturation, and value of a pixel to determine if it matches the color that is being tracked. It is parameterized, so a user can alter values the low and high thresholds of hue, saturation, and value. If the pixel HSV values are within the parameterized ranges, match will be a logic 1. Otherwise, match will output low.

center_calculator and divider_pipeline

This submodule calculates the location of the center of mass of an object each frame. The module has internal variables to sum hcount and vcount. When the match input is high (corresponding to a pixel match), the module adds the hcount and vcount to the sum_x and sum_y variables, then adds one to the total count of matching pixels. When a new frame is reached, the calculation begins again. The center of mass calculator must use dividing to calculate the values, so it also requires create a divider using an IP core in ISE. The sel output is high if the count of pixels is above a certain threshold, which we decided to be 500. If the sel line is high, we considered the center of mass to be in the frame of view of the master Labkit. Otherwise, the maze game would instead use the center of mass data transmitted form the slave Labkit.

average_center

This submodule averages the center of mass data over the most recent eight frames. When the valid line goes high, the submodule performs the averaging and outputs this new average until it must calculate the next average.

com_to_maze

This module takes in the x and y coordinates of the center of mass in the camera view and outputs its coordinates projected onto the maze. As two NTSC cameras were used for this project, each camera tracks the data in one half of the maze grid. The offset_select input decides whether to offset the pixels such that they can display in either half of the maze grid.

Implementation and Testing

The implementation process of this module involved 4 steps:

1. Modifying code provided by the 6.111 staff to display NTSC feed in color and store color data in ZBT memory
2. Converting RGB data after being read from ZBT memory into the HSV color space and determining ranges of HSV values to represent the colors we wished to track
3. Calculating the center of mass of an object based on its color
4. Averaging center of masses from numerous frames to reduce noise

Most testing and debugging of these modules involved programming the FPGA directly and viewing the camera feed via VGA output.

Many modules used here, namely zbt_6111, vram_display, ntsc_to_zbt, ntsc_decode, YCrCb2RGB, and rgb2hsv, were provided by staff. The code initially displayed the camera feed only in black and white, and the data stored for each pixel was only 8 bits wide. I first modified the staff code to store 18 bits of data for each pixel, using double as much memory, store the data in ZBT memory after being converted to RGB, and display the camera feed in color.

After converting the pixel data to RGB and displaying it, the next objective was to convert to HSV and determine proper values of H, S, and V to represent the colors being tracked. When debugging this portion, I colored each pixel white if it was a match, and black otherwise.

Color Detection in HSV Space

When debugging, I found that edges of objects presented a large amount of noise in the data. Also, the color we initially wished to use, a bright lime green, was too close to white in the HSV space, picking up a large amount of noise from light.

Figure 7 Low SNR Ratio When Tracking Lime Green Glove

After picking values to represent the color in the HSV space, the next step was center of mass calculation. When debugging, I displayed the center of mass overlaid on the camera feed in RGB to determine if it was tracking properly. I also made use of the LED hex display to display the current values of variables such as sum_x and sum_y to ensure they were being calculated. To debug the averaging module, I also displayed the averaged center of mass overlaid in much the same way. I found that after averaging, there was significantly less noise in the calculated center of mass.

HSV Threshold Calibration Tool

Libby

During the process of implementation and testing, we struggled often with having too much noise in the background when trying to detect a certain color. Since much of the success of our project depended on accurate and precise detection of the colored gloves, and since we were using two different NTSC

(which can vary vastly in terms of their sensitivity), I decided it was valuable to have a tool which would allow us to exactly optimize the HSV threshold ranges for color detection for each NTSC camera.

This tool, `camera_test.bit`, uses the NTSC to ZBT to VRAM pipeline that Steph had assembled. I then allowed the user to manually optimize the feed by adjusting the value and the range of the H, S, and V parameters, exactly as one would set the threshold ranges for color detection in the code.

1. Guestimate a rough hue value for the color of choice (e.g. green is 80 – 90)
2. Flip only switch[7] (hue) high. Adjust range value to 5
3. Use up or down arrows to move towards that value. When in the range, pay attention to whether the amount of signal and noise are increasing or decreasing. Settle on a hue value that has the highest signal-to-noise ratio.
4. Flip switch[4] to adjust range. Increase range until you have maximized the signal to noise ratio.
5. Repeat steps 1-4 for the saturation and value/brightness parameters.
6. Flip switch[2] to AND all the signals together and evaluate the quality of your color detection threshold.
7. Record the H, S, V values (displayed on the hex display). Note that the range values associated with each parameter is `parameter ± range`, so your low threshold would be `parameter – range`, and your high threshold would be `parameter + range`.

Note that all parameter values can range from 0 to 255 (as outputted by the `rgb2hsv.v` module)

Table 3 Manually Optimized HSV Values for Individual NTSC Cameras. Note that on Camera B, the V parameter spans the whole range. For some still unknown reason, that specific camera would not recognize an object within any range.

Camera A (Receiver)		**Low**	**High**
Green	H	43	123
	S	105	255
	V	75	225
Red	H	232	252
	S	170	190
	V	95	175

Camera B (Transmitter)		**Low**	**High**
Green	H	70	110
	S	70	135
	V	0	255
Red	H	232	252
	S	140	210
	V	0	255

Center of Mass Calculation

Figure 8 Center of mass (white square) overlaid with the NTSC live feed

The final step in the testing of the center of mass modules was the testing of the com_to_maze submodule. For the testing of this module, I primarily debugged by displaying the converted center of mass locations in the actual maze game. This submodule required finding offset and scaling factors to accurately project the center of masses onto the maze logic. The testing of this submodule occurred simultaneously with integration of the center of mass tracking with the game logic.

3.6 Serial Transmitter and Receiver

Steph

This serial modules have 2 main goals:

1. Send center of mass data serially from slave Labkit to master once per frame
2. Receive the center of mass data by the master Labkit

The serial modules are used to communicate between the two Labkits. Both Labkits perform center of mass calculations simultaneously on the data from their respective NTSC cameras. Once per frame, the slave Labkit sends the center of mass calculated in that frame to the master Labkit. When the calculations cycle to the next frame, the transmitter Labkit sends a start signal, which notifies the receiver Labkit that it should expect to receive new data. The master Labkit continually waits to begin receiving data, then begins receiving after getting the start signal. The transmitter Labkit uses a timer to determine how long it should send a certain signal.

Similar protocol to that used in lab 5b were utilized for the implementation of the receiver and transmitter modules.

Transmitter

The transmitter takes as its inputs a transmitter clock, a ready signal, and data to be transmitted. It outputs a data line with a one-bit width, when can be assigned to an output port on the labkit to allow for transmission of data via a wire to the second labkit. Internally, the transmitter works as a finite state machine to transmit. The signal is transmitted one bit at a time, with the length of a high transmitted determining the value of the bit being transmitted; for example, the start signal is a voltage high for 32 clock cycles, a one is high for 16 clock cycles, and a zero is high for 8 clock cycles, with each high being separated by a low voltage output for 8 clock cycles. The number of clock cycles for each state is parameterized and used as the input to a timer. The timer begins counting down at each transition between states. The state machine remains in the WAIT state until the start signal goes high, then it transitions to the START state. After transmitting the start signal, the code latches the data input at that time into an intermediate variable and begins transmitting the data one bit at a time. The data line is high when the state machine is in either the TRANSMIT_HIGH or START state, and low when it is in the TRANSMIT_LOW or WAIT state.

Receiver

The receiver module takes in a one-bit data line and a sampling clock as input and outputs the received data. There are only two states: WAIT and RECEIVE. The receiver samples the data line once every clock cycle of the input clk. Once the Labkit receives the start signal, which corresponds to data high for about 32 clock cycles, it transitions to the RECEIVE state. In this state, the receiver continues to sample the data. A high signal for 16 clock cycles corresponds to a received one, and a high signal for 8 clock cycles corresponds to a received zero. The receiver has an internal bit counter to determine how many bits have been received and the significance of the most recently received bit. The receiver tracks the received data in the received_data variable, then after it has finished receiving all 42 bits (holding data for the x and y coordinates for two separate colors), it updates the green_x, green_y, red_x, and red_y output variables. These outputs are the locations of the center of masses of the two colors in the NTSC camera before scaling to fit the maze.

Testing

I began by testing the transmitter with predetermined data. I began by running in ModelSim with dummy data, and examining the outputs.

I began testing of the transmitter in hardware first. I did this mostly by assigning the transmitter clock and transmitted data line to ports on the I/O banks and probing them with a scope probe. Initially, I attempted to transmit data using the 65MHz clock used for the VGA and NTSC camera. However, I found this clock was too fast and caused large amounts of noise in the data.

Next, I attempted to run the same code, but transmit using a slower clock. I found that dividing the 65MHz clock by a factor of 4 was sufficient to transmit the data with significantly reduce noise.

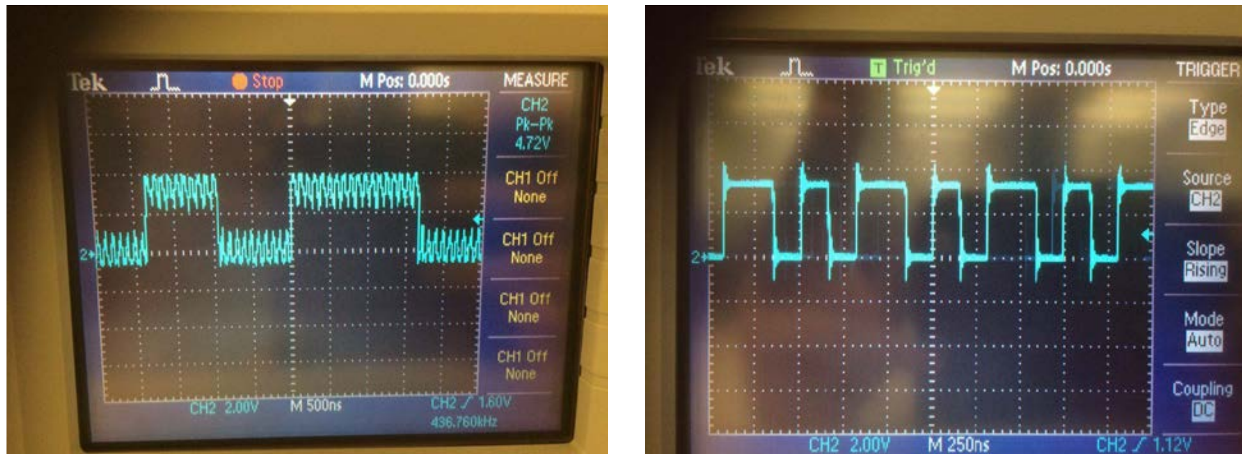

Figure 9 Transmission with a *(left)* 65 MHz clock and *(right)* ~16 MHz clock

To test the receiver, I also began by running simulations in ModelSim. To find whether the data was being correctly received in hardware, I displayed the received data using the LED hex display. I simultaneously displayed the data that was being transmitted on the LED hex display of the transmitter labkit, then I could view if these values were matching.

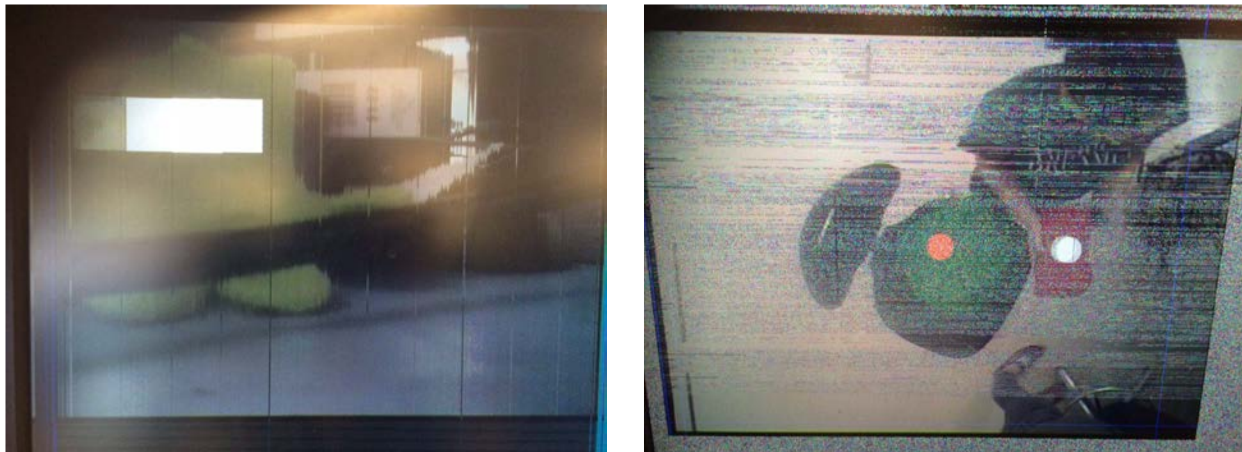

Figure 10 Examples of Glitching NTSC Feed

4 | Improvements and Insights

NTSC camera and Color Recognition

One of the biggest insights we learned about the NTSC camera was the importance of power cycling. Often when repeatedly changing code, even after making changes to the code unrelated to the NTSC camera, the camera feed would unexpectedly glitch. This included problems such as overlaying the entire feed with a purple hue, discontinuity in the frame, only half of the pixels updating, etc. After much trial and error, we found the only way to correct many of these errors was to power cycle the labkit and reprogram the FPGA. This problem caused much frustration throughout the course of our work.

Additionally, we found that each NTSC camera was unique. This meant that we would need to calibrate the HSV values corresponding to a color separately for each NTSC camera used. Initially, we assumed that the HSV values that were sufficient to track a color for one camera would be the same for all NTSC cameras. However, this is not the case.

Finally, we found that there was noise around the edges of the NTSC camera view. Near the center of the frame, the center of mass could be calculated fairly accurately. However, near the edges of the camera view, noise caused a greater amount of pixels to register as a match that were not actually matches to the color we were tracking. Also, we found that the color tracking using HSV values were sensitive to shadows, and the color tracking worked best when the colored object was completely flat, allowing for no wrinkles in fabric or any similar disturbances to disrupt the color tracking.

COM to Maze

After integrating the transmitted data into the game logic, I found that the noise in the transmitted data caused glitches that would affect the maze FSM logic. The data often would glitch and appear in the finish area for a fraction of a second. However, this fraction of a second would cause the game to recognize the player as having won the game. I believe this could have been fixed by averaging the transmitted data, but this was unfortunately not implemented in the final version of the project.

Transmitter/Receiver

Due to noise in the transmitted data, the receiver requires allowance for a large error margin. For example, transmission of a 1 corresponds to the data line being high for 16 clock cycles. However, at the receiver end, it recognizes a signal as a 1 if it samples the data line being high for anywhere between 13 and 19 clock cycles. Originally, I tested the code with a 1 being received if the receiver samples between 15 and 17 high signals at the data line. Through trial and error, I found this range was too small.

Originally, I attempted transmitting the clock that the data was being transmitted at as well as the data line. This way, I thought I would be able to sample the data coming in at the same clock speed that it was being transmitting. However, with two separate lines being transmitted, this created cross talk between the separate data lines. To fix this, I created a separate clock in the receiver that had the same frequency as the transmission clock. This fixed the error of contamination in the data, but it also meant

that to change the clock speed, I needed to alter portions of the code in both the receiver and transmitter project files.

Finite State Machine

While exploring different designs of preventing players from walking through walls, I had initially created an algorithm that would more accurately and realistically determine valid locations. For example, in the example provided by *Figure 3*, the valid path elements which would transition the player out of the WARNING state would not be limited to those directly adjacent to the last valid location. Instead they would also include all of the path elements between the current wall and the next wall on the left. If this algorithm had been used, this would have allowed (0,0), (0,1), (1,0), (1,1), (2,0), and (4,3) (with (0,0) being the upper-left-most element shown) to additionally be considered valid path elements. In the final implementation of Invisimaze, I opted for the simpler algorithm that I described previously because the significantly smaller cost of calculation outweighed the slight increase in accuracy I would have achieved using the more complicated algorithm. The alternative, more accurate algorithm could be better a choice if the maze grid was of a higher resolution (more grid elements per physical square foot) and required a more accurate and realistic method of calculating when the player has walked through a wall or not.

As mentioned previously, the warning countdown time can be improved by designing a reprogramming option. This would allow players to choose how much grace time they are given when they enter an invalid area, thereby increasing or decreasing the difficulty of the game in one facet.

A random number generator could be implemented to randomly select from the available maze maps at the beginning of each new game so that players do not have to select from a predetermined number.

Another fun feature I thought of but did not implement was to have an ultra-difficult (”Arnold Schwarzenegger”) mode where the maze map randomly changes on the player *while* they are playing the game. This would add an extra layer of fun and challenge to the game for those who have mastered it. In terms of implementation, all of the logic would remain the same and does not need any additional modification (this has been tested by manually flipping the switches to choose a different map). The only additional logic needed would be a pseudo-random generator and a more complex valid location localization because the last valid location may not necessarily be valid anymore in the new maze.

Additionally, I learned that when you add two numbers together in Verilog, you must check that the sum is not larger than the binary logarithm of either of the numbers, even if you appropriately size the destination register or wire. This is because when Verilog adds them together, it temporarily stores it in a register the size of the largest addend. This caused some glitches on both the camera and the logic side.

5 | Conclusion

To conclude, we created a virtual reality maze game that a player could navigate through, albeit with some glitches. If we were to continue work on this project, the largest improvements we could make would be averaging of the transmitted data to reduce glitching, some form of correction at the edge between the view of the two cameras, and fully integrating the audio component of the system. However, it was very rewarding to see the system able to track the person's location in the maze based on color.

Working on this project allowed us to develop numerous valuable skills. Most importantly, it allowed us both to become increasingly comfortable working with Verilog. Beyond this, it allowed us to learn a great deal about color recognition, particularly the differences between representation of color in different color spaces, and gave us more experience developing more complex finite state machines and serial transmission protocol.

Overall, the opportunity to work on this project and complete the various 6.111 labs throughout the term were great learning opportunities.