Windows Manager Responsive to User Hand Gestures

Zhenya Gu & Yunjie Ma

Massachusetts Institute of Technology
6.111 Digital Electronics Laboratory Final Project
Fall 2006

Abstract

This project implements a windows manager program responsive to a user's hand movements. Windows can be opened, moved, minimized, and closed by four different hand gestures. The XVGA display contains two windows, two icons, and a task bar to which both windows can be minimized. The project was developed in two parts, one that analyzed the video input and determined the center of mass of the user's hand and the action the user is performing, and another that generates the XVGA display seen by the user and performs the actions determined in the other part. During debugging, common problems had to do with windows not responding to the user's actions. This was due to several factors such as the center of mass going off the screen, logic in the gesture interpreter, and glitching with displaying the BRAM data. These problems were resolved by wiring inputs and outputs to the hex display and observing what actions caused logic problems.

Table of Contents

List of Figures iii
List of Tables iii
1 Windows Manager Overview 1
1.1 Introduction 1
1.2 Setup 1
1.3 Configuration 1
1.4 Display 3
1.5 Usage 3
1.6 Other 5
2 Modules / Implementation 6
2.1 Video Decoder 6
2.2 YCrCb to RGB Converter 7
2.3 Write and Read from ZBT RAM 7
2.4 Center of Mass Finder 7
2.5 Screen Size Calibration 8
2.6 Hand Shape Detector 9
2.7 Hand Size Calibration 9
2.8 Gesture Interpreter 10
2.9 Display 12
3 Testing and Debugging 19
3.1 Read and Write from ZBT RAM 19
3.2 Center of Mass Finder 19
3.3 Hand Shape 20
3.4 Gesture Interpreter 20
3.5 Hand Calibration 21
3.6 Determining Parameters 21
3.7 Display 21
3.8 Putting It Together 22
4 Conclusions 24
References 25
Appendices 26
A Screenshots of windows manager display 26
B Matlab script for generating .coe files 28

List of Figures

Figure 1.1	Photo of project setup	1
Figure 1.2	Hex display if switch[4] is off	2
Figure 1.3	Buttons and switches used for configuration . . .	2
Figure 1.4	Hex display if switch[4] is on	2
Figure 2.1	Block diagram of the video processor . . .	6
Figure 2.2	State transition diagram for the gesture interpreter module .	11
Figure 2.3	Diagram of horizontal and vertical sweep of XVGA .	13
Figure 2.4	Diagram of XVGA pixels	13
Figure 2.5	Block diagram of windows manager	14
Figure 3.1	Screenshot of ModelSim simulation of hand_shape module .	20

List of Tables

Table 1.1	Hand size select switches and associated hand gestures .	2
Table 1.2	Cursor color corresponds to hand gesture . . .	3
Table 2.1	Hand size select switches and associated hand gestures .	9
Table 2.2	Legend for state transition diagram	10
Table 2.3	FSM states, counters, and actions	12
Table 2.4	Sizes of BRAMs used to store images for XVGA display .	15
Table 2.5	Control signals for display	16
Table 2.6	Effect of action on windows properties . . .	17

1 Windows Manager Overview

1.1 Introduction

For this project, a windows manager that is responsive to hand movements in front of the screen was designed and implemented. The idea originated from the major motion picture, *Minority Report*, in which Tom Cruise manipulated windows on a screen simply by moving his hands. Our project emulates this concept by recording the movements of the user's hands with an NTSC camera and translating these movements into commands that change windows displayed on a 1024x768 XVGA screen.

1.2 Setup

The camera faces the computer screen. The user manipulates windows with his or hand between the camera and the screen. A picture of the setup can be seen in Figure 1.1.

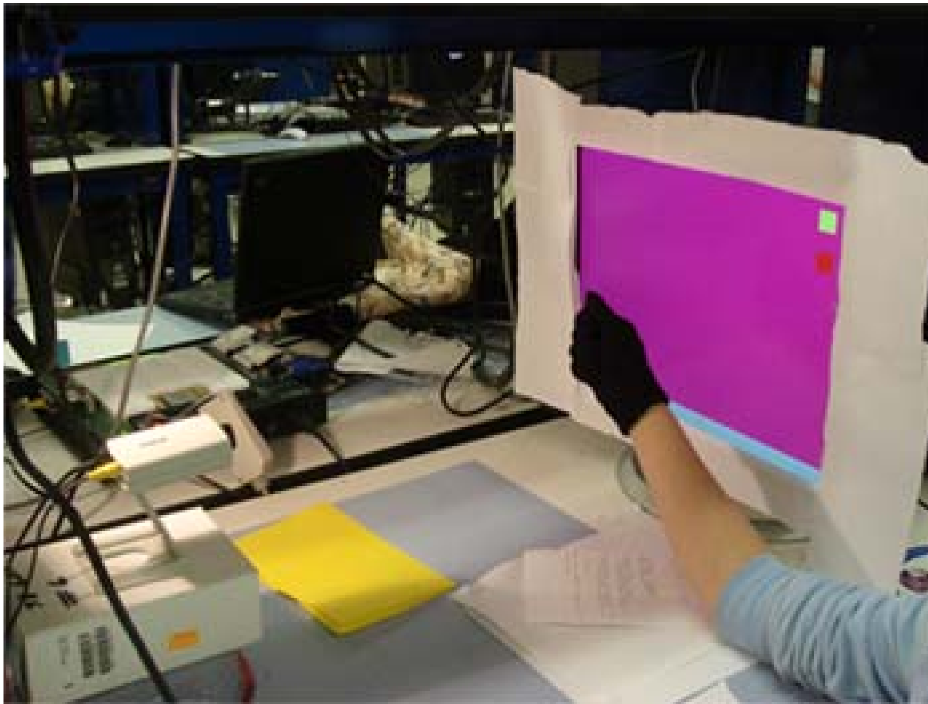

Figure 1.1 Setup – hand between camera and computer screen.

1.3 Configuration

To use the windows manager, the user's hand sizes must first be configured with the system. In order for the windows manager to work, the user must be wearing a black glove. Bright lighting also helps because dark shadows interfere with the video processing. The hand gestures are determined based on hand sizes so calibration of the program to the user's hand is important.

There are four hand gestures to calibrate, open, move, minimize, and close. The user's hand should be in a fist shape for all four actions. The four actions are differentiated by the distance between the hand and the camera. The open gesture should be furthest away from the camera and the close gesture the closest. The four gestures should be a significant distance apart (~ 1.5 inches). Otherwise the windows manager will be difficult to use.

To calibrate a hand shape, hold the hand in the desired position and press the calibrate button for a few seconds. Then program this area into the windows manager by using two switches to select the gesture and pressing the reprogram button. This should be repeated for all four gestures. The hand area and area to be reprogrammed into the windows manager are both displayed on the hex display if switch[4] is off. The user can then check if the area to be programmed is the desired area. See Figure 1.2 for a labeled diagram of the hex display. A table of the switches and the corresponding gestures is shown in Table 1.1. Figure 1.3 shows a diagram of the buttons and switches used for configuration.

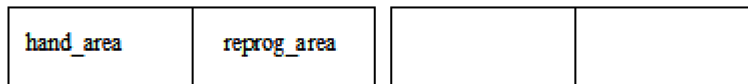

Figure 1.2 Hex display if switch[4] is off. Boxes represent a set of four digits in hex.

Table 1.1 Hand size select switches and associated hand gestures.

Switch[2:1]	Gesture
00	OPEN
01	MOVE
10	MINIMIZE
11	CLOSE

In addition to hand size calibration, the screen size may also need to be calibrated. Screen size calibration requires adjusting parameters in the Verilog file. The Verilog files will need to be recompiled. To calibrate the screen, flip on switch[7] to see the camera window. Next, find the minimum and maximum x and y values the hand can move to without part of the hand leaving the camera window. A 10x10 blob can be moved with the directional buttons and turning on switch[4] will display the x and y coordinates of this blob on the hex display. See Figure 1.4 for a labeling of the different values displayed on the hex display. The xmin, xmax, ymin, and ymax values must be input into the center of mass module.

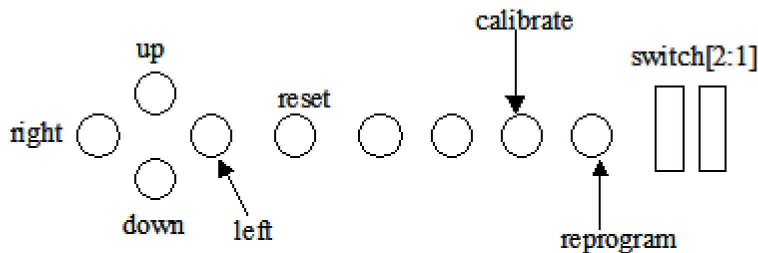

Figure 1.3 Buttons and switches used for configuration.

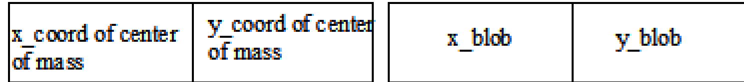

Figure 1.4 Hex display if switch[4] is on.

1.4 Display

Once the user's hand shapes are stored into the system, she is ready to use the windows manager. The screen first displays two icons on the top right corner and a task bar on the bottom. The left side of the task bar will show minimized windows. When a window is selected, a small square corresponding to the window selected will appear on the right side of the task bar. (See Appendix A.1 for photo)

A colored dot will follow the user's hand around the screen. The color of the dot reflects the command the user's hand is communicating to the system. The user controls the windows manager by the relative size of her hand according the camera. So as the user moves her hand closer to and farther from the camera, she will be indicating different commands to the system.

1.5 Usage

Selecting

To open a window, the user must perform the operation open on the icon. To do this, the user will move her hand towards one of the icons so that the curser dot is exactly on top of the icon. The user will put her hand approximately 2-3 inches from the screen or when the curser turns yellow. Her hand should be in a fist. After 2 seconds, the system will register the user's request and the window will open in its default location. The square on the right side of the taskbar will appear accordingly. (See Appendix A.2) If the window is already open, the above steps will bring the selected window on top at its current location. Opening a window from its minimized state requires the same steps as above except the curser will be pointing to the minimized box on the bottom of the task bar. However, opening a window from the task bar will return the window to its last opened location.

Table 1.2 A colored dot follows the user's hand around the screen. The color of the dot corresponds to the gesture hand performs.

Color	**Gesture**
Green •	Idle
Yellow •	Select/Open
Blue •	Move
Red •	Minimize
Light Blue •	Close

The user may also perform the select operation on the window itself. This is important because the other functions cannot be performed to a window unless it is selected. When the user moves his hand so that a yellow curser appears on a window for 2 seconds, she

will have selected this window and have brought the window to the foreground. The user may also choose to deselect all windows by holding a yellow curser above a space that has no windows, icons, or minimized objects for 2 seconds.

Moving

In this windows manager, the user can only move windows around. The icons, taskbar, and minimized objects are all stationary. To move a window, the window must first be selected. The user can see if the window is selected if the corresponding window color is lit on the right side of the task bar. (See Appendix A.3)

Moving a window begins when the user first gives the windows manager the move signal. Her hand should be about 3-4 inches away from the screen. The curser dot will turn blue when she is at the right distance. After the curse has stayed blue for 2 seconds, the window will follow the user's hand around the screen.

The user will be able to move the window to any location as long as her hand does not return to the move position. As a helpful hint, the user can open her hand so that all five fingers are out when she is moving the window around. Because the system registers commands by the size of the user's hand, an opened hand will cover more area and stop the windows manager from recognizing the move position. To active make sure the hand does not enter the move position, the user will need to watch the curser on the screen carefully. As long as the curser does not stay blue for 2 seconds, the user will be able to freely move the window around.

Once the user has found the new location of the window, the user must enter the move position again. The hand, now a fist again, must be about four inches away from the screen and the curser will be blue again. After 2 seconds, the window will no longer follow the hand around the screen.

Minimizing

The minimize function can only be performed on windows. Specifically, the user can only minimize the selected window. The selected window will be obvious to the user by the color of the box on the right side of the taskbar. To perform the operation, the user's hand must be around 4-5 inches away from the screen in. After holding her hand in this position for 2 seconds, the window will disappear and a flat rectangle will appear on the left side of the taskbar. (See Appendix A.4)

There is space on the taskbar for both windows to minimize. The first window minimized will appear on the first location from the left side of the screen. If a second window is minimized, the minimized object will show up to the right of the first minimized object. If the first window minimized is selected to be opened again, the second minimized object will move to the first minimized object's location.

When an object is minimized, the system remembers the location where the window last resided so that if the user chooses to open the window again, it will be at the same location as before.

Closing

The user can only close a selected window. If no window is selected, holding the hand in the select position will reflect no changes on the screen. When a window is selected, the corresponding box will appear on the right side of the task bar. The user will then place her fist approximately 5-6 inches away from the screen. The user will know her hand is in the right location when the curser turns light blue. After holding her hand in the close position for 2 seconds, the window will disappear. The last location of the window will be lost in the system. The user can open the window again only if she performs the select operation on the corresponding icon. Then the window will appear at its default location.

1.6 Other

The windows manager can also be reset by the enter button on the 6.111 lab kit (see Figure 1.2). The user must be careful to not press this button unless a complete reset is needed. Resetting the system will result in deleting all the stored hand sizes of the user and closing all the windows.

2 Modules / Implementation

The implementation of this project was split into two parts, the video processor and the windows manager. The video processor section determines the action the user is performing by analyzing the video input and outputs this action allow with the coordinates of the center of mass of the hand to the windows manager section. The windows manager section takes the action and coordinates and moves the windows accordingly. It also outputs the windows, icons and taskbar to the XVGA display. In addition, the windows manager section sends xvga signals to the video processor.

Video Processor

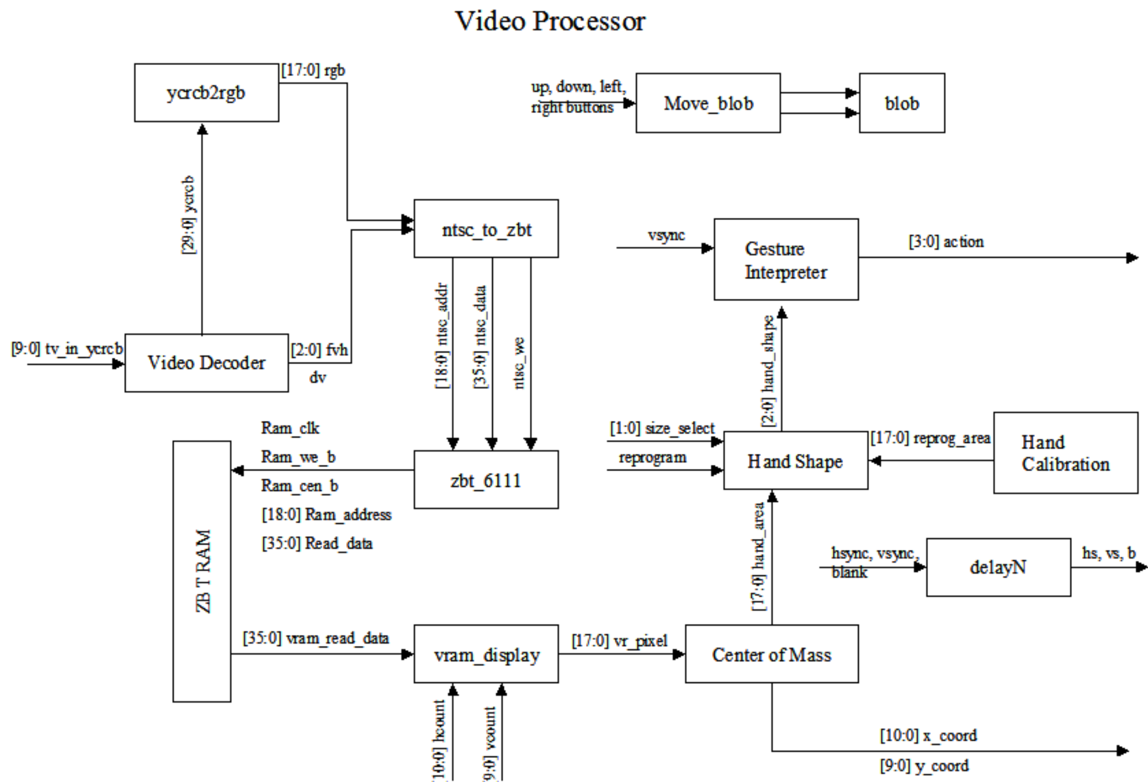

Figure 2.1 Block diagram of the video processor section.

2.1 *Video Decoder*

The video decoder module is taken from the 6.111 website and was not modified.[1] The module takes the video input from the video composite-in port on the 6.111 labkit and outputs a 24-bit number containing the luminance and chrominance values. It also outputs field, vertical and horizontal sync signals and a data valid signal.

[1] Ike Chuang and Chris Terman, *6.111 sample code*, Fall 2005.

2.2 YCrCb to RGB Converter

A YCrCb to RGB converter module was already written. The module takes in 10-bit Y, Cr, and Cb signals and outputs 8-bit red, green and blue signals. This 18-bit rgb signal is input into the ZBT write module.

2.3 Write and Read from ZBT RAM

The 18-bit rgb signal is input into the ntsc_to_zbt module, which prepares this data for storage into the ZBT RAM. The incoming signals also include the 65 MHz system clock, the 25 MHz video clock and field, vertical, and horizontal sync, and data valid signals from the video decoder module. This module outputs an address, data, and a write enable for the write to ZBT module. The incoming video frames are even/odd interlaced, but the even lines are thrown out by the ntsc_to_zbt module and the odd lines are displayed twice.

The 6.111 website's ntsc_to_zbt module wrote only luminance values to the ZBT RAM.[2] The module was modified to display color as well. The width of the ZBT RAM is 36 bits. Originally, each location stored the four pixels worth of data (i.e. 8 bits per pixel of luminance data). In order to store color data as well, each location was modified to hold only two pixels, each with 6-bits of red, green and blue data.

The data from the ntsc_to_zbt module is written into the ZBT RAM by the ZBT driver module, zbt_6111. The write enable is delayed by two clock cycles. This module also contains a two-stage pipeline for the write data.

Data is read from the ZBT RAM by the vram_display module. Since the ZBT RAM has two cycles of read and write latency, the data must be latched at the appropriate time. This module was also modified to handle two pixels per ZBT location instead of the original four. The vram_display module outputs 18-bit pixel data at 65 MHz, and this data is analyzed by the following modules to interpret the user's hand gestures.

2.4 Center of Mass Finder

The center of mass finder module takes in the 18-bit pixel data signal from the ZBT read module as well as hcount and vcount from the xvga module. It outputs the x and y coordinates of the center of mass and the area of the user's hand.

The module compares each incoming pixel with the set threshold rgb values. If the pixel rgb values are below the threshold values, the pixel's hcount is added to the sum of x-coordinates and the vcount to the sum of y-coordinates. These threshold values were determined to be 6'b001111 for red, green and blue to work with the black glove used in this project. These values can be easily modified for different colored gloves. A hand_count register also stores the number of pixels that fall below this threshold.

---

[2] Ike Chuang and Chris Terman, *6.111 sample code*, Fall 2005.

Not all pixels are compared with the threshold values. The pixels are limited by their hcount and vcount. Only those with hcounts and vcounts that fall within the range of the camera window are compared. The ZBT outputs a 1024x768 screen but the camera window is only 720x480 pixels.

To determine the x and y coordinates of the center of mass, this module instantiates four dividers. Two (divider1 and divider2) divide the sum of the coordinates by the value in the hand_count module, for x and y separately. The other two dividers (divider3 and divider4) adjust the coordinates for the 1024x768 display in the windows manager section. At the end of each frame, when vcount is equal to the maximum vcount of the camera window, the divider enable for divider1 and divider2 is set to 1. Hand_count is written to hand_area, which is output to the hand_shape module. Also, a 1 is written to a user register if the hand_area is greater than a minimum size. This is used to decide if the user's hand is present or not. If a hand is not detected, the coordinates are set to (0,0). The quotient of divider1 and divider2 are the x and y coordinates of the center of mass on the 1024x768 output of the ZBT. However, the camera window only occupies a portion of the screen. If the center of mass were to be output to the 1024x768 display of the windows manager, the center of mass would not be centered.

To adjust for this, the x and y coordinates are stretched to match a 1024x768 pixel screen using the x and y ranges of the user's hand movement. The x and y coordinates are multiplied by 1024 and 768 and divided by the x and y ranges, respectively. This is implemented with divider3 and divider4. The x and y ranges are determined by the view of the camera. The ranges must allow all of the user's hand to be shown in the camera. Otherwise, the hand sizes needed for the hand shape module would be incorrect.

In addition, this module also checks if the adjusted x and y coordinates will be out of the range of the windows manager display. If so, then the center of mass is kept at the edge of the screen and stopped from falling off the screen. This feature is important because otherwise, glitching would occur in the memory readouts from the BRAM in the windows manager module due to the negative x, y coordinates.

2.5 Screen Size Calibration

The move_blob and blob modules are used to calibrate the screen size. As mentioned in the center of mass section, all of the user's hand must be in the camera view or the hand size would be skewed. To find this range of x and y values, there's a 10x10 pixel blob that can be moved with the up, down, left, and right buttons. The hex display will show the x and y coordinates of the top left hand corner of the blob. These values should then be input into the center of mass modules under x and y minimum and maximum values for the windows manager screen.

The blob module takes in x and y coordinates and hcount and vcount. It outputs a 3-bit pixel. This module displays pixels for the length of the parameters WIDTH and HEIGHT starting at the input x and y coordinates.

The move blob module takes as input the four directional buttons on the lab kit and vsync from the xvga module. It outputs the x and y coordinates, which are then input into the blob module. The module stores the vsync from the previous clock cycle into a register, old_vsync, and compares this with the current vsync to find the edge of vsync. At the edge of vsync, the x and y coordinates are updated. The speed of the blob is currently set to 1 so it can be maneuvered more precisely, but this speed can be changed. The directional buttons increase or decrease the corresponding coordinate by the speed.

2.6 Hand Shape Detector

The hand shape detector module determines what the user's hand gesture is based on the hand area output from the center of mass finder module. There are default parameters set for each of the four gestures, select/open, move, minimize, and close. These default values are written to registers at reset.

At each positive clock edge, the input hand area is compared to the four hand sizes. If the input falls within the margin of error of one of the sizes, that hand shape is output to hand_shape. Otherwise, hand_shape is set to IDLE.

The different sizes can also be reprogrammed using the reprogram button, the size select switches, and the area output from the hand size calibration module. The two switches are used to select which hand shape to reprogram. The area to be programmed is the output of the hand calibration module. Pressing reprogram sets the shape selected to the new area and this area will be used until reset is pressed.

Table 2.1 Hand size select switches and associated hand gestures.

Switch[2:1]	Gesture
00	OPEN
01	MOVE
10	MINIMIZE
11	CLOSE

2.7 Hand Size Calibration

The hand calibration module is used to measure a new hand size. The module takes in a calibrate signal, a reset signal, and an 18-bit hand_area signal from the center of mass finder module. An 18-bit old hand register stores the area of the hand from the previous clock cycle. A 19-bit hand register stores the sum of the old hand and the input hand_area. At reset, these two registers are set to zero. At every positive edge of the 65 MHz clock, if the calibrate button is pressed, the new hand_area is added to the old_hand value and this sum is written to the hand register.

2.8 Gesture Interpreter

The gesture interpreter module determines the action the user wishes to perform based on the series of hand shapes he or she uses. This module uses a finite state machine to transition between the different commands. Figure 2.2 shows the state transition diagram. A legend for the abbreviations is given in Table 2.2. This module receives a hand_shape from the hand shape detector module and x and y coordinates from the center of mass module. In addition, it receives a hand_ready signal that goes high once a frame and a 2-bit select signal from the windows manager module. The module outputs a 4-bit action to the windows manager module.

At reset, the state is set to RESET. It remains in RESET if hand_shape is 3'b000, and action is set to 4'b0000. At every positive clock edge, the module checks if reset has been pressed. If not, then the module checks if it is at the rising edge of hand_ready. If so, state transitions are made and counters are incremented. The rising edge of hand_ready is checked by comparing a stored hand_ready signal from the previous clock cycle with the hand_ready in the current clock cycle.
If hand_shape becomes one of the four recognized shapes, the state transitions to the corresponding state. However, the action is still set to 4'b0000. In the new state, the hand_shape must be held for a set number of frames before the desired action is output. This is to ensure that random user hand movements do not result in an action. Only shapes that are held for around two seconds will be recognized by the windows manager. Each action has a counter that increments at every rising edge of hand_ready. These counters are stored in open_wait, hold_wait, min_wait, and close_wait. At reset, all of these registers are set to zero. The length of time the user is required to hold each shape before an action is output can be set in the parameters section of the module. Currently, the length is set to 60 frames for all of the shapes.

Table 2.2 Legend for state transition diagram

hs	hand shape
OL	open length
CL	close length
ML	minimize length
HL	hold length
OW	open wait
CW	close wait
MW	minimize wait
HW	hold wait

Open, minimize, and close are very similar states. All three require the appropriate hand shape for the FSM to transition into the state from RESET. Once in the state, if the hand shape stays the same, the state's corresponding counter is incremented every frame. The module also checks if the movement of the center of mass is within the margin of error before incrementing the counter. Therefore, actions are not recognized if the user's hand is continuously moving. This check is implemented by storing the previous clock cycle's coordinates in registers and comparing them with the new coordinates. If the hand shape

changes from that of the state's anytime before the required length of time has passed, the FSM transitions back to RESET. If the required length of time has been completed, the FSM outputs the desired action and transitions to RESET. The counter is also reset to zero.

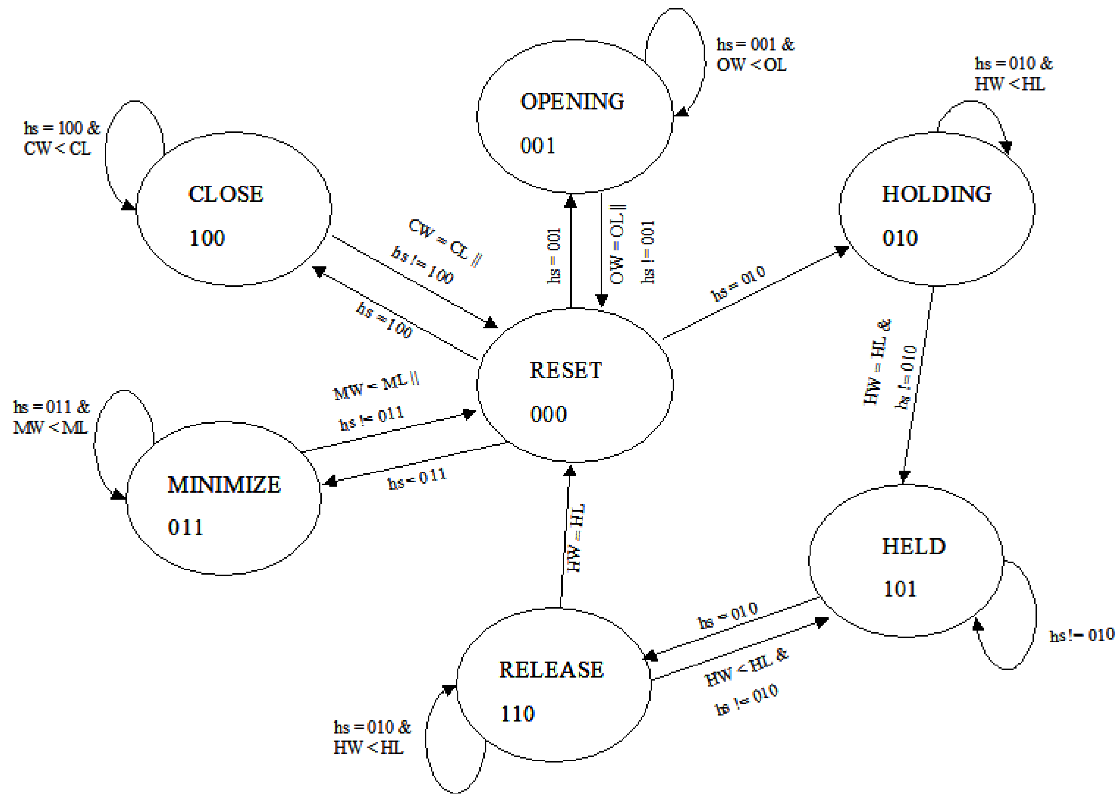

Figure 2.2 State transition diagram for the gesture interpreter module.

Move is a more complex action, since the user requires that the window follow his or her hand for an indefinite amount of time. When hand shape first becomes the move shape in the RESET state, the FSM transitions to the HOLDING state. In HOLDING, the FSM checks for three criteria before it increments the hold counter. The hand shape must be the hold shape and the movement of the center of mass must be within the margin of error. In addition, the select signal from the windows manager module must be nonzero. A nonzero select signal means that one of the windows has been selected. This check is important because if no window is selected, the user has no feedback that he's in the HOLDING state and it becomes difficult to leave that loop.

Once the hold shape has been held for the required length of time, the FSM transitions to the HELD state as soon as the user changes to a different hand shape, and the counter is reset to zero. This is so the user does not have to maintain the hold hand shape while moving the window around. If the user has not yet changed to a different hand shape, the FSM stays in the HOLDING state, but does not increment the counter, and the action signal is set to HOLD_WINDOW. In the HELD state, a hold shape transitions the FSM into RELEASE. Any other hand shape keeps the FSM in HELD. The action is set to HOLD_WINDOW the entire time. In RELEASE, the user must once again keep the hold

shape for a length of time to release the window. This sets action to IDLE. If the hold shape is not held for long enough, the FSM goes back to HELD and the counter is reset to zero. Table 2.3 contains a list of the various variables and registers mentioned in this section.

Table 2.3 List of FSM states, counters, and actions. Actions occur when a counter has been incremented to the required length of time.

State #	State	Counter	Actions
000	RESET	n/a	IDLE
001	OPEN	open_wait	OPEN_WINDOW
010	HOLDING	hold_wait	HOLD_WINDOW
011	MIN	min_wait	MIN_WINDOW
100	CLOSE	close_wait	CLOSE_WINDOW
101	HELD	n/a	HOLD_WINDOW
110	RELEASE	hold_wait	IDLE

Windows Manager

2.9 Display

The screen of the windows manager displays 1024x768 pixels. To drive this many pixels with a refresh rate of 60Hz, a 65-Mhz clock was used throughout the display. The FPGA's digital clock manager was used to make the 65-Mhz clock from the normal 27-Mhz clock. With the 65-Mhz clock, the XVGA module produces the hcount, vcount, hsync, vsync, and blank signals required to drive the video output. The hcount signal sweeps horizontally across the screen while the vcount signal sweeps vertically down the stage. Hsync and vsync signals are high when hcount and vcount reaches the end of the screen respectively.

The color of the screen is made up of three signals: vga out red, vga out green, and vga out blue. These three signals, each an 8-bit number operate at 65-Mhz and put together represent the color at the pixel location indicated by the hcount and vcount.

The second module in the display component of the system is the windows manager module. The windows manager determines the red, green, and blue values (RGB) at each pixel as indicated by hcount and vcount. To simplify matters for the display, instead of outputting a 24-bit number, the windows manager module will only output a 4-bit number. This allows the screen to display 16 different colors, enough for the system. The 4-bit number is finally mapped to the corresponding 24-bit RGB value to be sent to the screen.

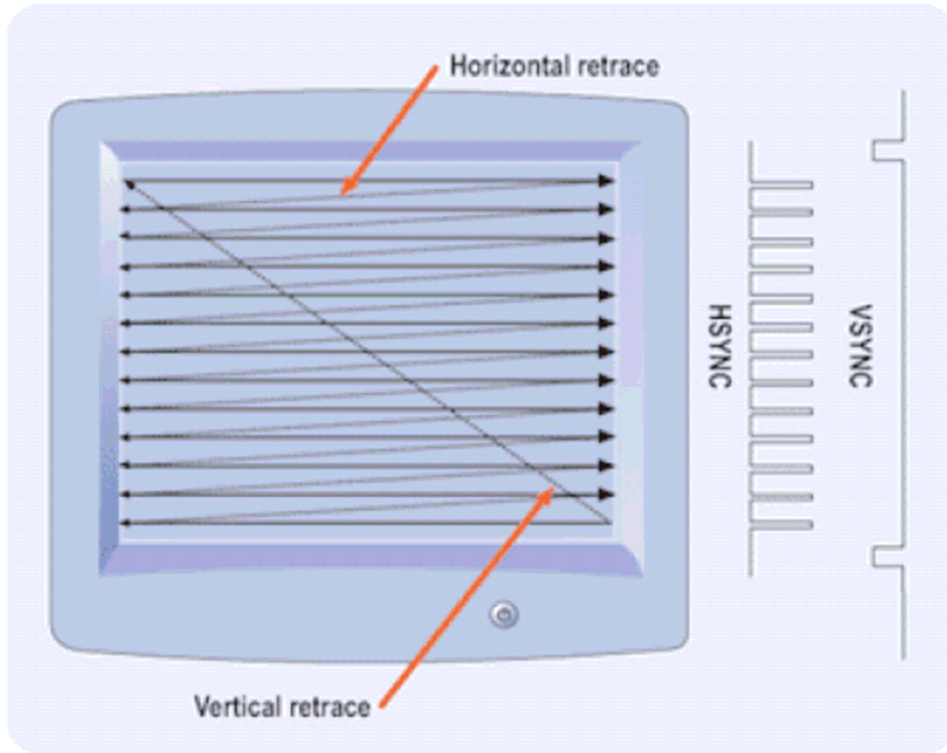

Figure 2.3 Hcount increments by one as it sweeps across the screen and vcount increments by one as it sweeps down the screen. At the end of each horizontal sweep, hsync pulses. At the end of each vertical sweep, vsync will pulse.[3]

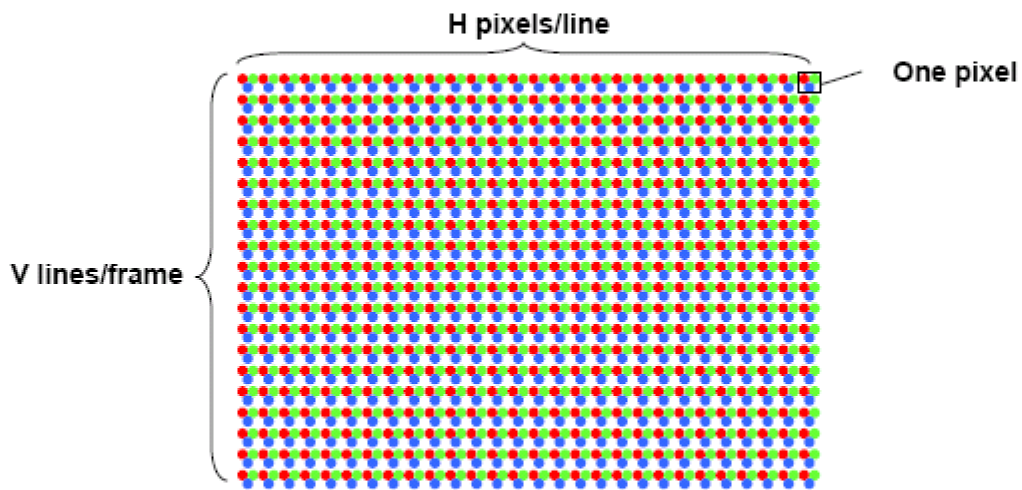

Figure 2.4 The color at each pixel is the combination of three colors: red, green, blue. The intensity of each color is determined by the signals vga out red, vga out green, and vga out blue respectively.[4]

[3] Chris Terman., *6.111 Lecture 16*, Fall 2006, p. 15
[4] Chris Terman, *6.111 Lecture 16*, Fall 2006, p. 2.

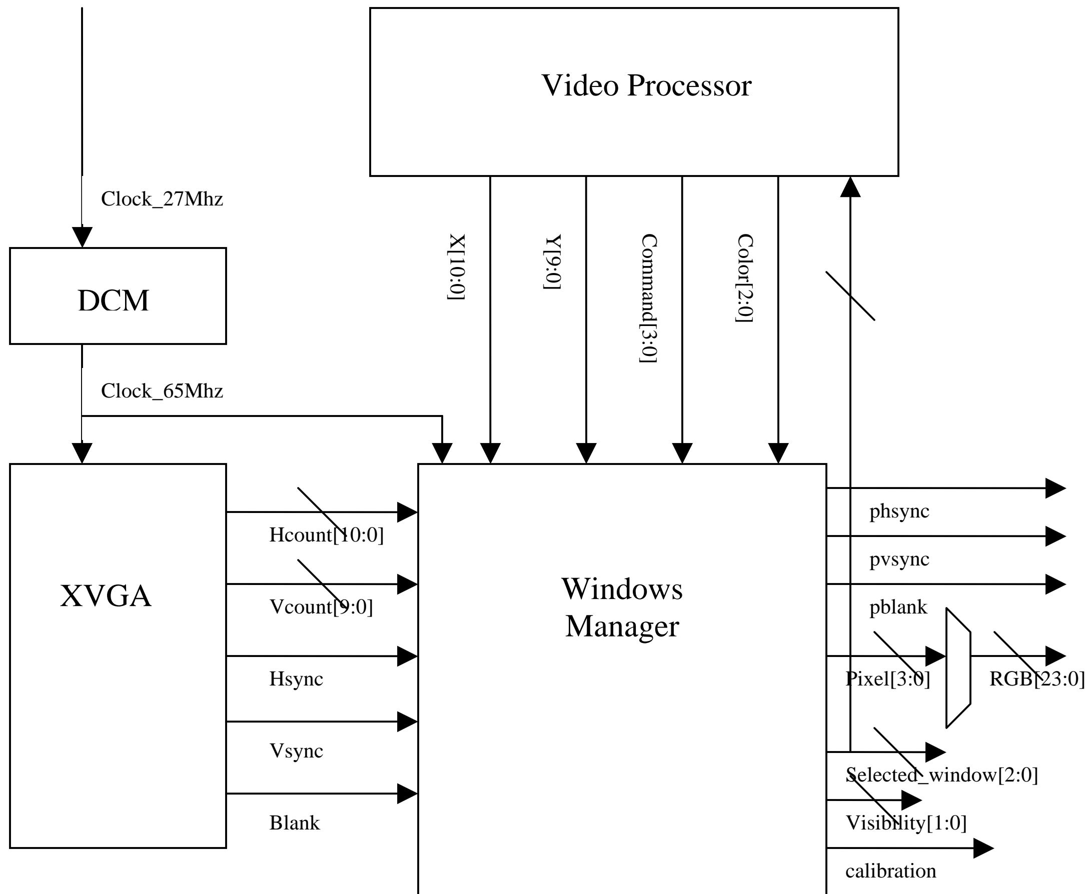

Figure 2.5 The display component uses two modules XVGA and windows manager. The XVGA module creates signals that produce a 1024x768 screen that refreshes at 60Hz. The windows manager takes the video signals produced from the XVGA module and coordinate and command signals to determine a 4-bit pixel that can be translated into 16 different colors.

The phsync, pvsync, and pblank signals are the same signals as hsync, vsync, and blank inputs to the windows manager. They go into the windows manager module in case there exists a delay between the hcount and vcount signals and the pixel signal. In this system, the pixel signal is synchronized with hcount and vcount so phsync, pvsync, and pblank are assigned the same values as hsync, vsync and blank signals.

The selected windows and visibility signals are debugging lines used during the design stage. The are connected to led[7:6] and led[5:4] respectively. The selected window signal is also fed back to the video processor. The feedback stops the video processor from sending out a moving command when no windows are selected. Without the selected window feedback, the FSM in the video processor would enter the moving state without the user knowing thus preventing all other hand gestures to be recognized

The video processor gives the window manager four signals to determine the RGB value at each pixel: x, y command, and color. The color signal corresponds to the hand shape the video processor recognizes. The x and y signals represent the location of the center of

the hand. The windows manager displays a 5 pixel by 5-pixel square at the pixel location given. The curser image is created by a blob sprite that takes as input x, y, color, hcount, and vcount. The blob creates a colored square when hcount is between x and x+5 and when vcount is between y and y+5. The color of the square changes according to the color signal given by the video processor. The purpose behind the color of the cursor is to give feedback to the user about what command the windows manager system sees from her hand shape.

The windows manager is in charge of two windows, window0 and window1. There is also a task bar on the bottom of the screen, made by a sprite like the curser. Other objects that interact with the windows are the icons, icon0 and icon1, minimized objects, minimized0 and minimized1, and the selected box, selected0 and selected1. All eight of these objects that are displayed on the screen by modules that read from a Read Only Memory. Like the blob sprite that creates the curser, each of these modules know the width and height of their respective objects. The windows manager feeds each module the x and y coordinates of their respective objects. The module then returns the necessary pixel colors when hcount and vcount are between x and x+width, and y and y+width. When hcount and vcount are outside the object boundaries, the modules return a blank pixel, which is the background color purple.

Each memory is four bits wide and contains the color information about the object displayed. The memory is created in BRAM and requires only three ports: a clock, address, and data. The data lags the address given by one clock cycle. Each module resets the address signal to zero after an entire screen has printed, or when vsync is high. Then address increments when hcount and ycount are within the given limits.

The size of the windows were limited by the size of the BRAM available on the labkit. Each ROM was initialized by a .coe file through Xilinx's core generator. Microsoft Paint was used to create each of the objects. The images were saved as a 16 color Bitmap. Then a Matlab script created .coe files for each Bitmap file (see Appendix B). The color map from Matlab is the color map used to convert the 4-bit pixel signal into the 24-bit RGB signal.

Table 2.4 Each object's pixel information is stored in a 4-bit wide read only BRAM. Each pixel in the picture is an address in the memory.

Object	Size (width x height)	Memory
Window0	200x350	280k
Window1	350x400	560k
Icon0	50x50	10k
Icon1	50x50	10k
Minimized0	100x30	12k
Minimized1	100x30	12k
Selected0	30x30	3.6k
Selected1	30x30	3.6k

Icons are stationary objects on the screen. They do not move, but are always there as long as windows do not cover them. The icons are physically located at the top right corner of the screen.

There are four sets of signals that together determine if a select object needs to appear on the bottom right of the screen, if a minimized object needs to appear on the taskbar, or if a window is opened on the screen. Selected window, visibility, location0, and location1 each is 2-bits wide and store all the information necessary,.

Table 2.5 The four signals together control which windows, minimized objects, and select box to display. Note that location0 and location1 can be the same only if their values are both 2'b00.

Bit	Selected window	Visibility	Location0	Location1
00	No windows selected; No selected boxes displayed.	No windows are displayed on the screen	No objects minimized at the first minimized location.	No objects minimized at the second minimized location.
01	Window0 is selected; window0 should be on top; select0 is displayed.	Only window0 should be displayed on the screen.	Window0 is minimized at the first minimized location.	Window1 is minimized at the second minimized location.
10	Window1 is selected; window1 should be on top; select1 is displayed.	Only window1 should be displayed on the screen.	Window1 is minimized at the first minimized location.	Window1 is minimized at the second minimized location.
11	N/A	Both window0 and window1 should be displayed on the screen.	N/A	N/A

The selected objects share one space, so only one module is needed for the two objects. The select box modules contains both the select0 and select1 picture memories with two sets of addresses inputs and two sets of data outputs. The select box module receives the selected window signal so that when window0 is selected, the select0 picture appears on the bottom right corner of the screen. Likewise, when window1 is selected, the select1 picture appears on the screen. When neither window is selected, there will be no select box displayed, and the bottom right corner will just contain the right side of the task bar.

There are two spots in the taskbar where windows can be minimized to. The first window minimized will reside on ½ inch from the left edge of the screen on the taskbar. If the second window is also minimized, then it will reside 1 inch to the right of the first location. The windows manager module controls the minimized object by the two signals, location0 and location1. Location0 is the first place a minimized object will reside, and location2 is the second place. Each minimized module, on the other hand, stores the minimized picture that corresponds to the window to be minimized. So each minimized module receives both signals location0 and location1.

For example, when window0 is the first window to be minimized, location0 will store the value 2'b01 while location1 will remain at 2'b00. When the module minimized0 sees the

change in location0 and will put the corresponding minimized object in the first location. When window1 is minimized, locatoin1 will change to 2’b10 and thus minimized1 will appear in the second minimized space on the taskbar. If window0 is to be opened again, location0 changes to 2’b10 while location1 changes to 2’b00.

Windows have the ability to appear, disappear, and move around all over the screen. So the signals that dictate how the windows show up on the screen are visibility, selected window, and the x and y coordinates of each window. Visibility corresponds to if either or both the windows are opened on the screen. The selected window signal dictates which window is on top and if a command is called, which window the action is performed on. The x and y coordinates determine the location of the windows on the screen.

The move function works by storing at all times two sets of the cursor’s x and y coordinates: the current coordinates and the coordinates at the previous clock cycle. When move is performed, the difference in the two coordinates are subtracted and added to the x and y coordinates of the window. Precautions are made so that the windows does not go off the screen when moving. Of course, windows are given the option to cover icons and the task bar.

Table 2.6 Each action affects the properties of the window in different ways.

Action	Prerequisite	Selected window	Visibility	Location0/ Location2	Coordinate	State
Select	The cursor on top of the icon, window, or minimized object	Changes	Changes if select icon/ minimized	Changes if select icon/ minimized	No Change	Changes
Move	A window must be selected	No Change	No Change	No change	Changes accordingly	No Change
Minimize	A window must be selected	Changes to 2’b00	Changes accordingly	Changes accordingly	No change	Changes when another window is still open
Close	A window must be selected	Changes to 2’b00	Changes accordingly	Changes accordingly	No change	Changes when another window is still open

The windows manager functions lastly as a simple two-state finite state machine. The two state, STATE_0over1 and STATE_1over0, corresponds to if window0 or window1 is on

top. This information could easily be recovered without using a finite state machine, but having the layer information stored in registers eliminates additional delays that might have been implemented.

Finally, each of the above properties can change when the video processor give a command. When a window is closed, the selected windows signal returns to 2'b00, the visibility changes to reflect the window closing, and the state may change depending if there are other windows are still open. Refer to Table 4 for the full list of how each command can affect windows properties.

The windows manager module receives a 4-bit color signal from each object module. The manager then uses combinational logic to determine which signal should be sent to the color map and finally translated into the 24-bit RGB value. The combinational logic first detects if the cursor is located at that pixel, then if windows reside at the pixel (the order depends on the state the module is currently in), followed by if icons, minimized objects, and selected box are at the pixel, and finally if the taskbar is at the pixel location. If the object does not reside at the pixel location, then the 4-bit color value will be the same as the background color. The combinational logic then outputs the first signal it detects to be an object.

3 Testing and Debugging

The order the modules were created or modified was ZBT RAM, center of mass finder, hand shape, gesture interpreter, screen size calibration and hand calibration. Some modules could be tested individually using ModelSim, but others such as the center of mass finder had to be connected with the ZBT RAM and the xvga signals to be tested. For such modules, debugging was done using the hex display and switches.

3.1 Read and Write from ZBT RAM

The ZBT RAM module from the 6.111 website already displayed the data read from the RAM onto a 1024x768 XVGA screen, but in black and white. The changes to color could be displayed on the XVGA screen, so if the module worked, this could be immediately recognized by connecting a camera and a screen to the labkit.

3.2 Center of Mass Finder

The biggest problem with the center of mass finder was forgetting to check the divider enable option when generating the divider. Though there was an enable signal, it was being ignored by the divider and the divider continuously divided, instead of just at the end of a frame. The center of mass was therefore constantly changing, even when the black object in the camera view didn't move. This problem was solved by displaying the x and y coordinates, hand area and the divider enable signals on the hex display. While the divider enable and hand_area signals seemed correct, the coordinates kept looping from zero to a very large number. This indicated that the problem might be with the divider and not with the calculations for x and y sum or hand_area.

In addition, a 10x10 pixel blob is generated at the center of mass coordinates to indicate where the calculated center of mass is. This made testing this module very easy. The accuracy of the center of mass finder can be judged by comparing the blob location to the black object in the screen. The center of mass module was tested always with the ZBT RAM and camera connected. To test this module, black objects were placed in front of the camera. A switch also toggled between a normal camera view and a view where all the pixels above the black threshold were turned into white. In the latter view, all the non-white pixels were pixels included in the center of mass calculations.

Later, a few adjustments were made to this module, such as the adjustment of the center of mass to match the 1024x768 XVGA screen and the limiting of the x and y ranges to fit within the screen. To debug these adjustments, black objects were placed in front of the camera and the center of mass could be seen in the ZBT display. The bounding of the center of mass can be tested by bringing the hand to the edges of the screen. When a hand is not present, the 10x10 blob goes to (0,0).

One problem was that the maximum x and y coordinates had to take into account the width of the 10x10 pixel blob. The blob is 10x10 pixel in the 1024x768 screen, but is much smaller in the camera window. Initially, the maximum x and y coordinates took

into account a 10x10 pixel blob in the camera window, but this prevented the center of mass from reaching the edges of the screen in the 1024x768 screen. Reaching the edges of the screen is very important for opening the icons and minimized windows because the icons and taskbar are located very close to or at the edge of the screen.

3.3 Hand Shape

The hand shape module is relatively simple and it could be debugged using ModelSim. To find the correct areas for the different hand shapes, the hex display was programmed to display the area output of the center of mass module. These values are input into the hand shape module as parameters. To test in ModelSim, a testbench was created that input values within the error margin of the hand sizes to test if the correct shape is output. A screenshot of the ModelSim clocking diagram is shown in Figure 3.1.

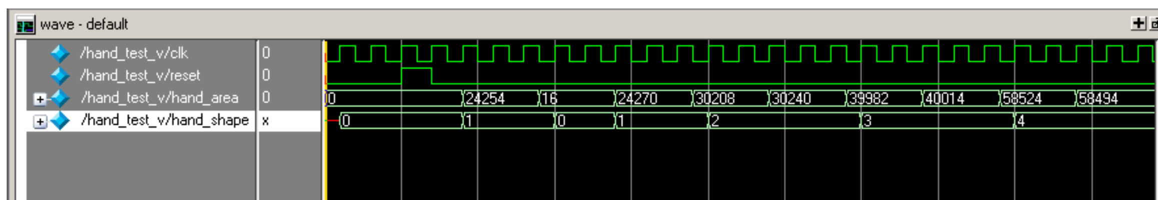

Figure 3.1 Screenshot of ModelSim simulation of hand_shape module.

This module was further tested by connecting it to the video camera and center of mass module. The pixel output was wired to turn all non-black pixels different colors depending on the hand shape output by this module. The reprogramming feature was tested using the hex display. The hand size registers were rewired to be outputs displayed on the hex display. These outputs could then be compared with the hand area values just programmed into the module.

3.4 Gesture Interpreter

This module was debugged both in ModelSim and with the hex display. FSM was tested with a testbench run in ModelSim. The state transitions were mostly bug-free. While testing in ModelSim, the wait lengths were adjusted to 3 pulses of hand_ready instead of 60. This made the state changes must faster, making them easier to observe in ModelSim.

Later, when the gesture interpreter was connected with the other modules, the state and action signals were wired to the hex display. These values could then be checked while testing the various hand gestures.

One change from the original design was in the hold action. It is very difficult for the user to maintain the hold hand shape while moving the window around. If the FSM required the user to maintain this shape the entire time, windows would often be dropped in unintended places. To fix this, the FSM required the hold hand shape only to pick up and drop the window. While moving, the user can use any hand shape except the hold shape, which would drop the window.

3.5 Hand Calibration

The hand calibration module was tested using the hex display as well. The calibration area output was compared to the hand_area output of the center of mass module. If the hand did not move, the two numbers should be similar.

3.6 Determining Parameters

To determine the threshold values for the center of mass module, red, green and blue masks were created. The threshold values of these masks can be adjusted using the up and down buttons on the labkit. Pixels above these threshold values were set to white. This allowed the threshold values for black to be easily determined. The rgb comparisons can also be easily modified to black out all pixels above these threshold values to find objects of different colors. This was used to calibrate the screen size. A red box was drawn in the middle of the screen. Mask was used to black out all the other pixels, and the edges of the red box were determined using the move_blob and blob modules mentioned in the screen calibration section.

3.7 Display

The display component was built function by function. The first stage of the display was to just have windows open, close, and move around. No icons or taskbars existed at this point. Two switches were connected to open and close the two windows. The up, down, left, and right buttons controlled the movement of the windows, without following a cursor. In the next stage, icons and the taskbar, and minimized objects were added to the screen.

Switches became unreliable because their edge wasn't fast enough. So the commands were wired to buttons zero through three. Button0 controlled the move command; button1 controlled the open/select command; button2 controlled the close function; and button3 controlled the minimize function. Switches, instead determined which window was selected for the commands. The cursor was still not implemented. For example, if switch0 was closed and the button 1 was pushed, then the directional buttons would be able to control the movements of the window0. If switch0 stayed closed and button3 was pushed, then window0 would be closed.

A big realization was that the select, close, and minimize button can only be a single 65-Mhz clock. This was decided when the minimize function was tested. When trying to restore the window at the first minimized location, the other window would also restore. The rationality behind this is that opening the first window results in the second window moving to the first location. However, the buttons cannot be held for at exactly 65-Mhz. Likewise, the signals coming from the video processor is a 30-Hz pulse and would cause the same problems. So the three commands were changed so that they each pulsed for 1 65-Mhz clock cycle on a positive edge.

The last step of creating the display was making a test x and y coordinate for the cursor. The coordinates would start in the middle of the screen. When a directional button is pressed, the cursor would along with the direction. This changed the display because the windows module now needed to know where cursor is at certain commands. For example, for the select command, it was crucial to figure out which object the cursor was on each time it was pushed. Logic was also added to the move function so that the difference in the cursor coordinates would be added to the current x and y coordinate of the object itself.

Originally, a frame buffer was to be deployed along with the display. The frame buffer would have been stored in ZBT and required a 2-port read/write interface. However, a couple days into the design process, it was decided that a frame buffer wasn't crucial to the system. So the implementation started without the frame buffer. If at the end, glitches existed in the display, then a frame buffer would have been added. Fortunately, the display did not have any glitches.

3.8 Putting It Together

Even though both sides ran perfectly well by themselves, putting them together helped solve a lot of usability issues. The first version of the windows manager system was impossible to use. The user couldn't tell what shape her hand was in; the cursor moved around too much and was hard to track; and the select function would not work at times.

The first thing added was a method to debug the problems. The two components each already had checking methods implemented, so the trick was to put them together. With the use of the switches, the screen can be turned into a shot of what the camera was reading. Another switch changes the screen so that the hand shape of the user would change the color of the screen. Two more switches on the lab kit can change how the user input her commands. Instead of inputting the coordinates and commands using hand shapes, debugging can take place by using the numbered buttons to simulate commands and the directional buttons to simulate cursor movement. These additions allowed us to test the system more thoroughly.

The first thing realized after many test runs was that the hand shapes all had roughly the same sizes. It was frustrating for the user to guess which hand shape the camera was reading. Instead of creating hand shapes, the user would simply adjust the position of her hand relative to the camera. Moving it closer would result in a bigger hand area and farther a smaller hand area. A design choice was made to completely throw out the hand shapes idea and to just use distance to send commands. By using distance, the difference in the hand areas read by the camera became much higher. To add usability to the system, the color of the cursor now changes to provide feedback to the user. Instead of focusing on commands given, the color of the cursor preludes commands and simply lets the user know the shape that is recognized by the computer. So the user can adjust her hand accordingly before a command is read.

The select box was the next feedback component added to the windows manager system. It was easy to tell which window was selected and which wasn't when the windows were on top of each other, but extremely hard when they were side by side. Furthermore, to emulate real operating systems, selecting over the desktop area would result in deselecting the windows such that other commands would be rendered useless. To ease the confusion, the select box was put in the right side of the taskbar so that the user can see which window she has or has not selected.

One last problem detected in the system was the fact that select sometimes seemed not to work. Many times, no matter how long the select hand shape was held on top of a window, icon, or minimized object, it just would not be selected. After sifting through the code, it was determined that the FSM in the video processor would enter a moving state without the user knowing. In a moving state, all other commands are ineffective until the move hand shape is recognized again. While it is easy to tell when the system is in the moving state when a window is selected (a window would follow the hand around), the FSM would enter the moving state even without a window selected, making it impossible to detect the error. So a feedback was created between the display and the video processor. When there are no windows selected, the FSM in the video processor cannot enter the move state. The problem was solved.

The last component of the windows manager system was to add the ability for multiple users. We all know that everybody's hands are different sizes and the original windows manager was made only for one set of hand shapes. However, a configuration stage was added so that the hand shapes for each command could be recorded into the video processor each time a reprogram button is programmed. So the system now allows other users to use the windows manager simply by reconfiguring the hand sizes needed for each command.

4 Conclusions

The objective of this project was to implement a user hand-controlled windows manager with the 6.111 FPGA labkit. The desired functionality of this program is outlined in the project checklist. The final project achieved all of the basic functionality as well as some of the additional functionality such as hand size calibration.

When putting the two sections of the project together, there were very few glitching problems. A few errors we had dealt with the center of mass falling off the screen and giving windows manager negative coordinates to display the BRAM data. This caused the image to display incorrectly. However, these issues were fixed by limiting the range of the center of mass and also by adjusting some BRAM read code.

However, we did have some usage issues with the gesture interpretation design. Soon after we put the two parts together, we noticed that the usability of the program was low. It was very difficult for the program to interpret the correct gesture. This had to do with the limited area differences between hand gestures such as fist and open hand. Fingers are simply not large enough to alter the area by a significant amount. This idea was thrown out and replaced by distance movements instead. The hand is always in a fist and is simply moved closer or further away from the camera for different gestures. This significantly increased usability.

The next time we have a project like this, one of the improvements we would make in the video processor section is to create good debugging code from the start. We didn't have the masking code until later into the project, whereas that would have helped a great deal with re-evaluating our design at the onset of the project.

References

1. Ike Chuang and Chris Terman, *6.111 sample code*, Fall 2005.

   http://web.mit.edu/6.111/www/f2005/index.html

2. Chris Terman, *6.111 Lecture 16*, Fall 2006, p. 15.

   http://web.mit.edu/6.111/www/f2006/index.html

3. Chris Terman, *6.111 Lecture 16*, Fall 2006, p. 2.

   http://web.mit.edu/6.111/www/f2006/index.html

Appendix A. Screenshots of Windows Manager Display

A.1 Screen at reset

A.2 Screen with one window open

A.3 Moving a window

Appendix B. Matlab Script

Matlab script to convert Bitmap files to .coe files that can be uploaded into BRAM.

```
function BMPtoCOE(image_name)
%Converts a bitmap that stores up to 256 color bitmap a Xilinx .COE
file
%monitor

%read bmp data in and display it to the screen
[imdata,immap]=imread(image_name);
image(imdata);
colormap(immap);
numpixels=numel(imdata);

%create .COE file for image data
COE_file=image_name;
COE_file(end-2:end)='coe';
fid=fopen(COE_file,'w');

%write header information
fprintf(fid,';*********************************************************
*********\n');
fprintf(fid,';****                 BMP file in .COE Format
*****\n');
fprintf(fid,';*********************************************************
*********\n');
fprintf(fid,'; This .COE file specifies initialization values for
a\n');
fprintf(fid,'; block memory of depth= %d, and width=8. In this
case,\n',numpixels);
fprintf(fid,'; values are specified in binary format.\n');

%start writing data to the file
fprintf(fid,'memory_initialization_radix=2;\n');
fprintf(fid,'memory_initialization_vector=\n');
%convert image data to row major
newimdata=transpose(double(imdata));
%write image data to file
for j=1:(numpixels-1)
    fprintf(fid,'%s,\n',dec2bin(newimdata(j), 4));
end
%last data value supposed to have a semicolon instead of a comma
fprintf(fid,'%s;\n',dec2bin(newimdata(numpixels)));
%clean shutdown
fclose(fid)


% immap_size = size(immap);
% COE_map_file = 'colormap.coe';
% fid2=fopen(COE_map_file, 'w');
%
% %write header information
%
fprintf(fid2,';********************************************************
**********\n');
```

```
% fprintf(fid2,';****           BMP Colormap file in .COE Format
*****\n');
%
fprintf(fid2,';*******************************************************
**********\n');
% fprintf(fid2,'; This .COE file specifies initialization values for
the\n');
% fprintf(fid2,'; associated color path. \n');
%
% %start writing data to the file
% fprintf(fid2,'memory_initialization_radix=2;\n');
% fprintf(fid2,'memory_initialization_vector=\n');
%
% for j=1:(immap_size(1)*immap_size(2)-1)
%     fprintf(fid2, '%s,\n', dec2bin(immap(j)*255, 8));
% end
%
% %last data value supposed to have semicolon instead of a comma
% fprintf(fid2, '%s;\n',
dec2bin(immap(immap_size(1)*immap_size(2))*255, 8));
% fclose(fid2);
```