Realtime Light-Saber Generator

By Yuhsin Chen, Michael Price, and Hui Ying Wen

6.111 Introductory Digital Systems Laboratory

Final Project

December 13, 2006

Abstract

This project implements a video special effect in which a light-saber is shown projected from a handle held by a user. The saber image is superimposed on live video feed of the user and handle. Color detection of a marker on the handle, as well as input data from an accelerometer and gyroscope embedded in the handle, aided in constructing the resulting saber image.

TABLE OF CONTENTS

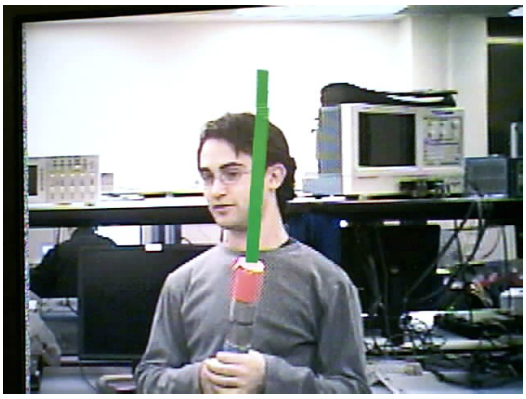
OVERVIEW

Basic Functionality

This machine is an implementation of a video special effect in which a light-saber is projected from a handle held in a user's hand. The image of the saber is tilted, scaled, and skewed according to the user's real-time movements of the handle.

The user, holding the saber handle, stands in front of a camera. Live input from the camera, in NTSC format, is written to a ZBT memory, which is then output in VGA format to a monitor. The image of the light-saber is superimposed on the live video feed.

The handle has three features that aid in this special effect: an accelerometer, a gyroscope, and a red-colored marker. Real-time data from the accelerometer and gyroscope are transmitted to the 6.111 labkit via wires at the base of the saber handle. This data, which identifies the current orientation of the handle, is used in transformation and rotation matrix algorithms which determine the correct tilt, size, and scale of the light-saber image. In addition, a center-of-mass calculation is performed on the pixels, detected from the live video feed, that comprise the colored marker on the handle. In this way, the pixel coordinates of the base of the light-saber, the origin from which the image should be drawn, are calculated. From this information, the quadrilateral comprising the light-saber is drawn on the VGA display as if coming from the user's handle, in the correct length and tilt.

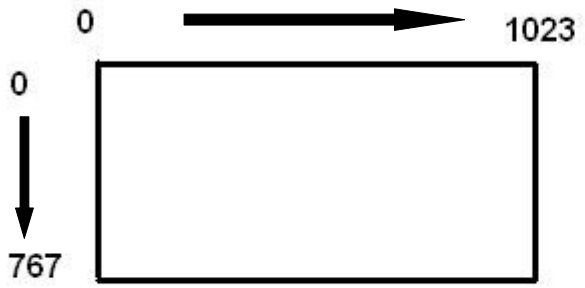


Screen image of saber with handle.

Conventions

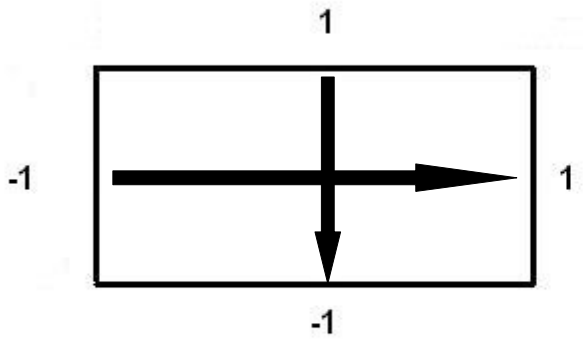
The following conventions are used in the process of calculating and producing the light-saber image: normal vs. pixel space, saber angles, and the ordering of saber points.

“Pixel space” refers to the conventional numbering of pixels on a video display:



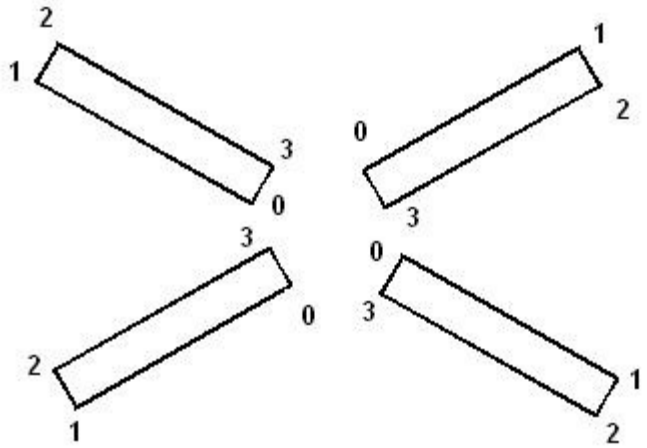
"Pixel space" for a 1024x748 display. This pixel numbering convention corresponds to the digital signals generated for such a display.

"Normal space" is the coordinate frame, still encompassing the screen, in which the transformation and rotation matrix mathematics are performed.



"Pixel space" for a 1024x748 display. This pixel numbering convention corresponds to the digital signals generated for such a display.

The ordering of the four points of the saber image is as follows:



Point ordering of saber image. Points 0, 3 correspond to the base of the saber. Points 1, 2 correspond to the tip of the saber.

Implementation

To fully describe a lightsaber's pose, six degrees of freedom would be needed: the position in [x, y, z] space and the vector representing the direction it is pointing. Collecting accurate information about all of those quantities from within the saber handle would require an expensive inertial measurement unit (IMU). Our system breaks down this problem into more convenient pieces:

- Position of the base of the beam: measured from video
- Orientation of beam: measured by inertial sensors in handle

Besides being cheaper than a single IMU, this approach is not subject to drift errors created by integrating acceleration measurements over time. The Analog Devices ADXL213 2-axis accelerometer and ADIS16100 gyroscope, to provide inertial measurements corresponding to the apparent direction of gravity and the handle's rate of rotation about its axis, were selected. Those measurements can be used to estimate the angles of deflection of the accelerometer from each axis in the global coordinate frame.

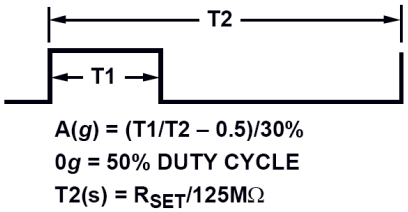
A 1024x768 resolution, 60 Hz monitor was used for video output. Live video feed was taken from an NTSC camera.

The project was implemented on the 6.111 labkit, with various switches and buttons used as inputs for testing, debugging, and various module functionalities (see module descriptions).

Interfacing with External Components

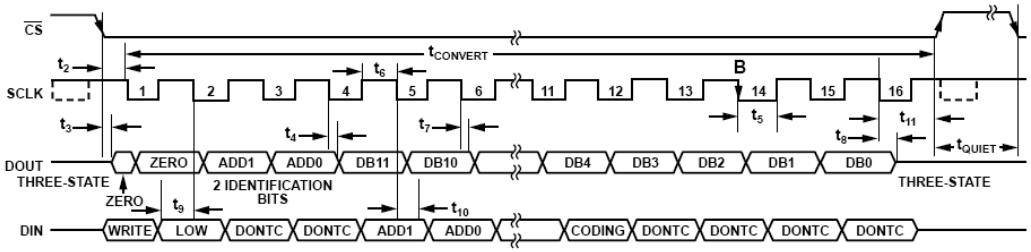
The appropriate communication mechanism was designed for each of the two sensors. The ADIS16100 and ADXL213 chips themselves come in very small surface mount packages, but Analog Devices supplied us with evaluation boards pre-made to support easy use of them (ADIS16100/PCB and ADXL213EB).

The ADXL213 chip needs an external resistor to set the frequency of its PWM output. A tradeoff between measurement accuracy and update speed is inherent in determining the PWM carrier frequency: dynamic performance improves as accuracy declines at higher frequencies. A 1 kHz carrier (using R = 120 kOhm) was selected to balance these desires and match the 1 kHz update rate of the lightsaber pose. The ADXL213 provides two synchronous outputs, one representing the component of acceleration in its 'x' direction and the other for the 'y' direction. The duty cycle of each output is ascertained by counting clock cycles to measure the pulse width and period, as described in section 4.2. A full scale output (0% or 100% duty cycle) corresponds to 1.2g, accomodating any tilt of the lightsaber without rapid acceleration.



ADXL213 Accelerometer Timing Diagram. (from Datasheet)

The ADIS16100 is more advanced, and communicates with the labkit via serial-peripheral interface (SPI). It requires a control command to be sent on startup, selecting between the gyroscope measurement, temperature measurement, and two auxiliary ADC input voltages (we use only the gyroscope). The output, whose update speed depends on the frequency of the serial clock, is a 12-bit unsigned number corresponding to rate of rotation about the vertical axis: 0 represents -300 deg/sec, 2048 represents no rotation, and 4095 represents +300 deg/sec.

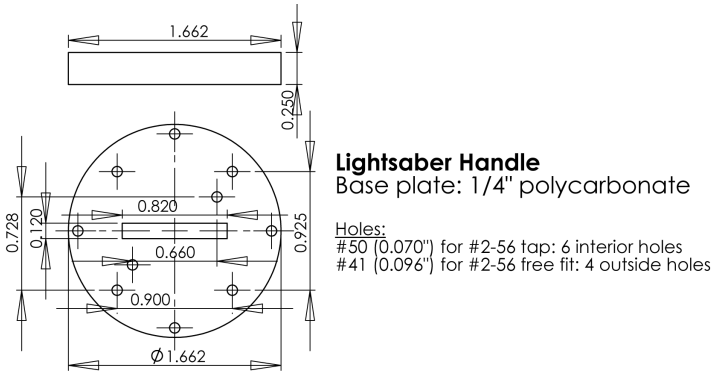


ADIS16100/PCB Accelerometer Timing Diagram. (from Datasheet)

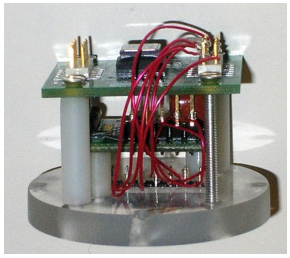
The inertial sensors were built into a lightsaber handle made of PVC pipe. The ADXL213EB and

ADIS16100/PCB are stacked on a circular plastic base fitting into the bottom end of the pipe, as shown in [images]. An 8-pin female header fits into the base, and the header pins are connected to the appropriate posts on the PCBs by wire-wrap. The assembly is held together by #2-56 machine screws. While aesthetics are not the primary goal of the design, we have decorated the lightsaber with high-tech tape insulation and wire grips.

The top end of the handle is wrapped in red paper, which acts as the marker and the origin of the local coordinate system.



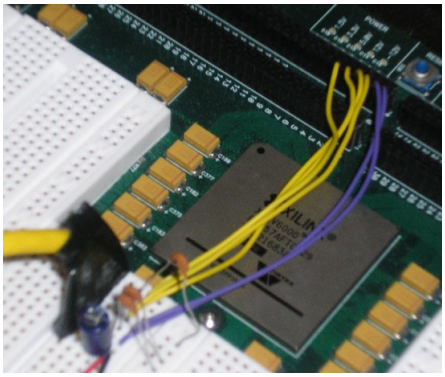
Base plate of light-saber handle, supporting input/output wires.



Accelerometer and gyroscope assembly for base of handle.

Internal and External views of handle.

The lightsaber handle connects to the 6.111 labkit through an 8-wire interface over 15' of Cat5 network cable. We linked the tether to the +5V supply on the labkit (with appropriate decoupling), as well as the user1 I/O port. The wiring diagram is shown below. To eliminate glitches, we filter the accelerometer outputs with 1.0 nF capacitors to ground.

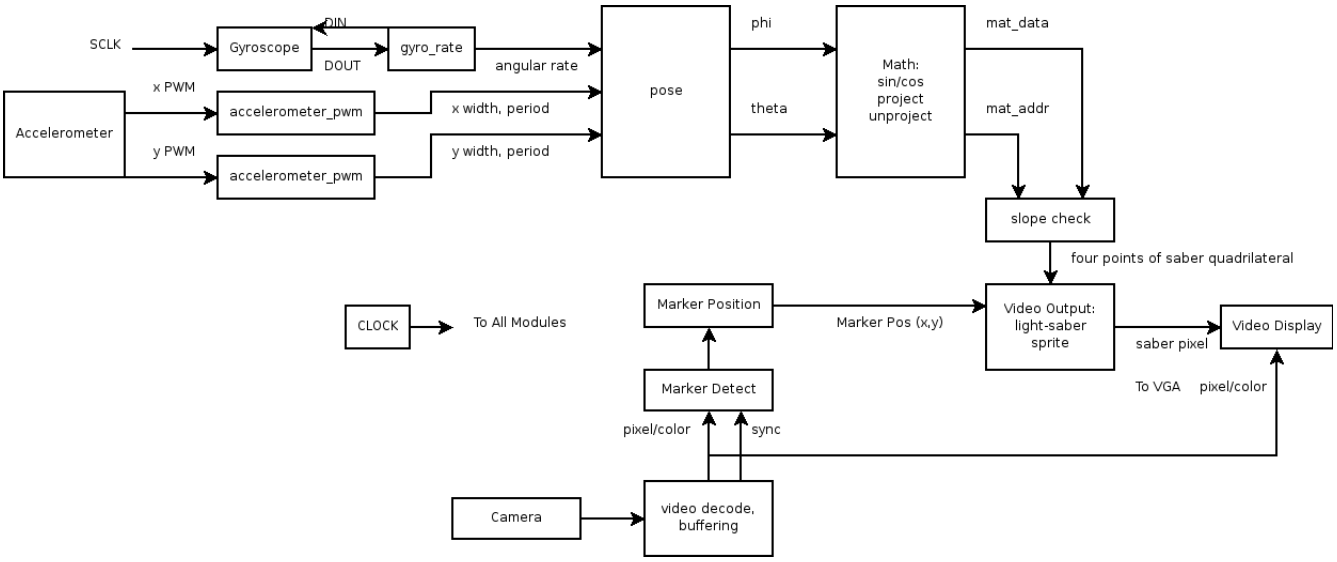


Wiring of external devices to 6.111 labkit.

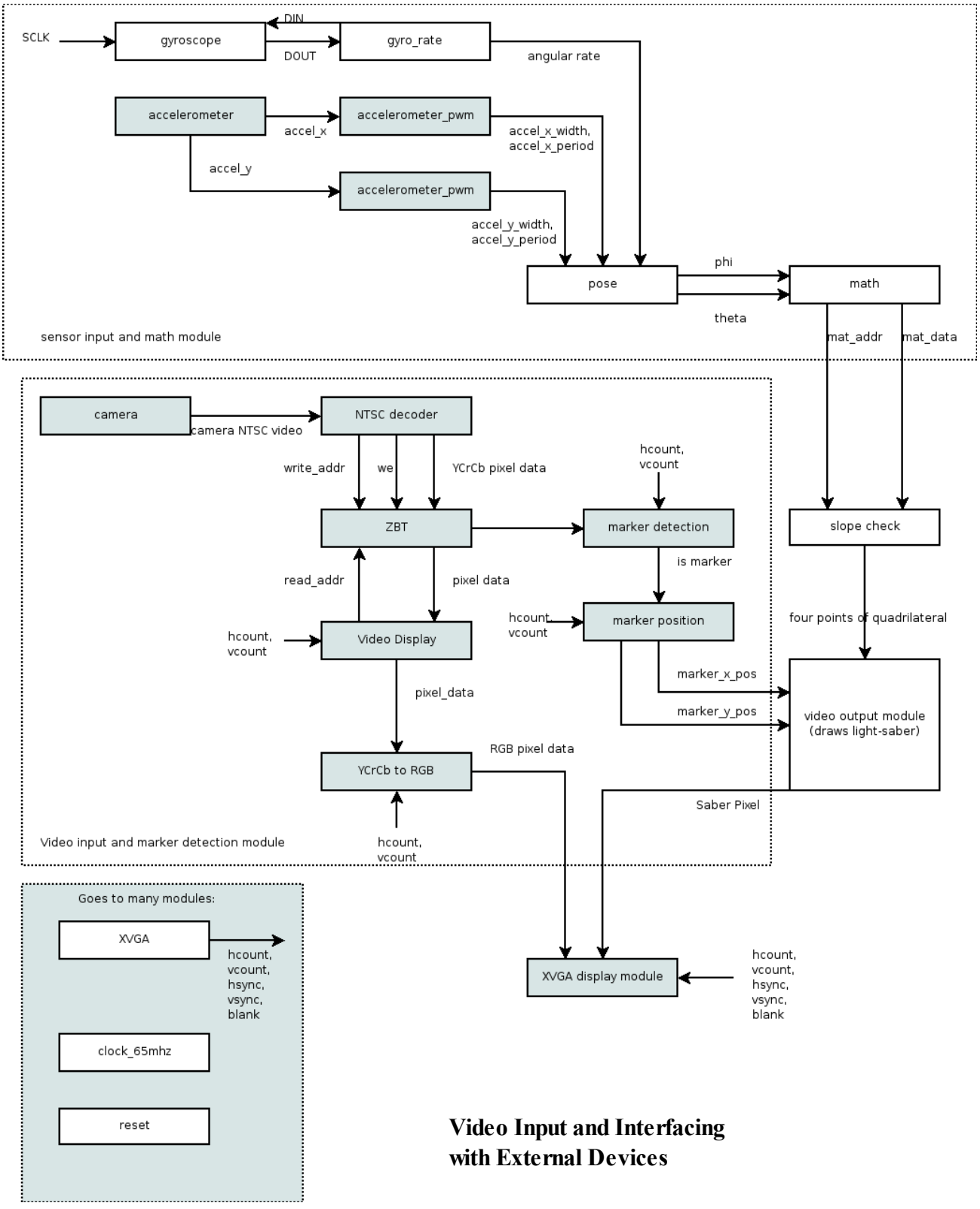
Pin	Description	Source	Destination
1	+5V	Supply rail on labkit	Sensor V_{cc} pins
2	GND	Common ground on labkit	Sensor ground pins
3	Accel_X	ADXL213	`user1[0]`
4	Accel_Y	ADXL213	`user1[1]`
5	CS	`user1[2]`	ADIS16100
6	SCLK	`user1[3]`	ADIS16100
7	DIN	`user1[4]`	ADIS16100
8	DOUT	ADIS16100	`user1[5]`

Wiring table (external devices to 6.111 labkit).

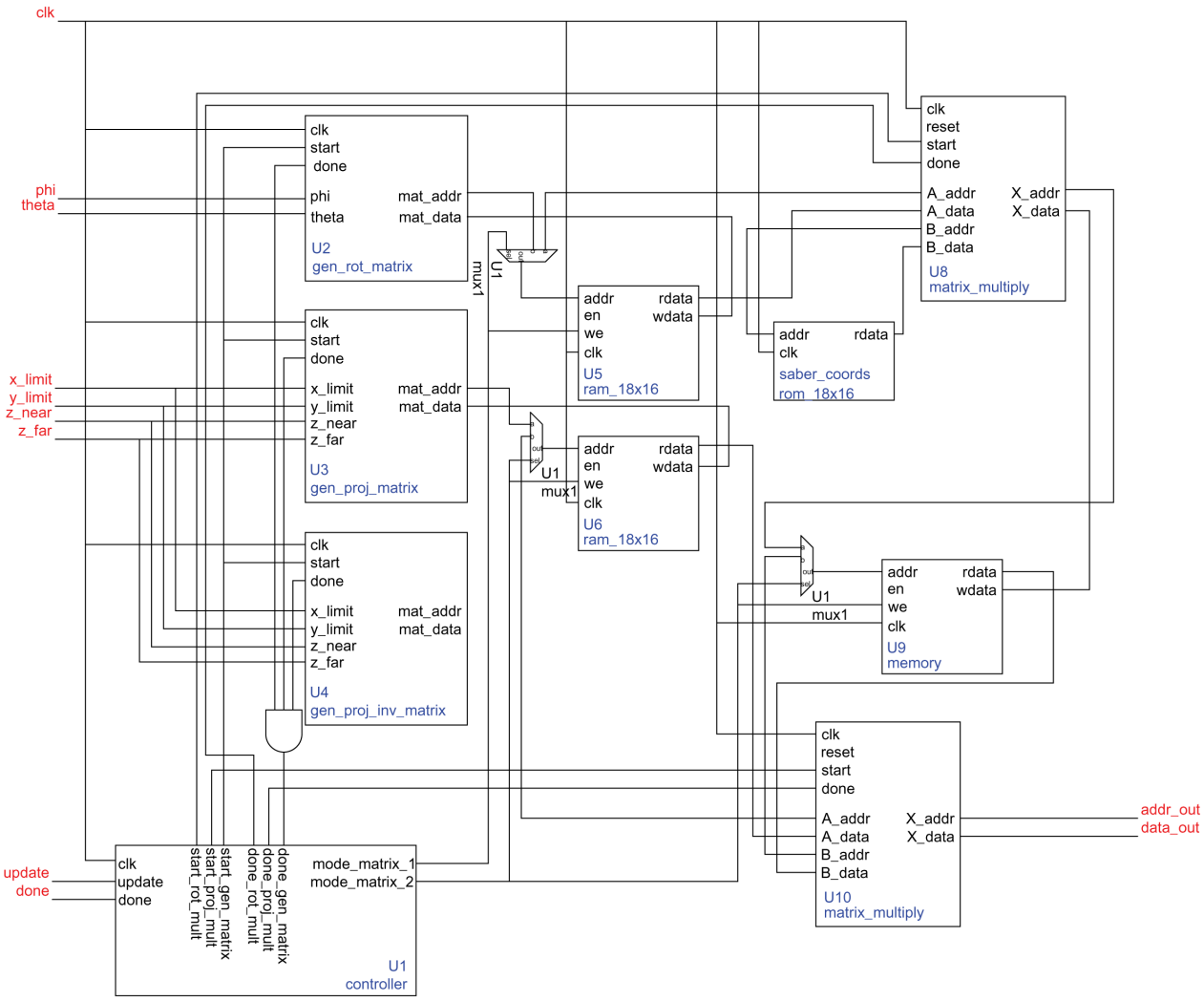
BLOCK DIAGRAMS



General Block Diagram



Video Input and Interfacing with External Devices


clk
phi
theta
x_limit
y_limit
z_near
z_far
update
done
clk
start
done
phi
theta
mat_addr
mat_data
U2
gen_rot_matrix
clk
start
done
x_limit
y_limit
z_near
z_far
mat_addr
mat_data
U3
gen_proj_matrix
clk
start
done
x_limit
y_limit
z_near
z_far
mat_addr
mat_data
U4
gen_proj_inv_matrix
clk
update
done
start_rot_mult
start_proj_mult
start_gen_matrix
done_rot_mult
done_proj_mult
done_gen_matrix
mode_matrix_1
mode_matrix_2
U1
controller
mux1
U1
addr
en
we
clk
rdata
wdata
U5
ram_18x16
mux1
U1
addr
en
we
clk
rdata
wdata
U6
ram_18x16
addr
clk
rdata
saber_coords
rom_18x16
clk
reset
start
done
A_addr
A_data
B_addr
B_data
X_addr
X_data
U8
matrix_multiply
mux1
U1
addr
en
we
clk
rdata
wdata
U9
memory
clk
reset
start
done
A_addr
A_data
B_addr
B_data
X_addr
X_data
U10
matrix_multiply
addr_out
data_out

Math

MODULE DESCRIPTION / IMPLEMENTATION

External Device Inputs

Accelerometer Module

The accelerometer_pwm module took the PWM input from the accelerometer and returned the width and period of the PWM as 18 bit integers. The width and period of the PWM were measured in terms of clock cycle periods, where the clock frequency was 65Mhz. Because the ADXL213 accelerometer we used was dual axis, two instances of accelerometer_pwm were instantiated for each axis x and y. The ratio of the width and period of the PWM was proportional to the sine of the light-saber tilt angle, with gravitational acceleration as reference. The PWM input from the accelerometer was found to be glitchy even when capacitors were added for low-pass filtering, so the PWM signal was fed through a debounce module; a signal had to be held high for 20 clock cycles before the debounce module reported a transition on its output. The 20 clock cycle lower bound was determined experimentally. The accelerometer PWM pulse operated at about 1 khz, so 20 clock cycles on a 65Mhz clock was considered negligible. The width and period of the pulse are much greater than 20 clock periods.

Gyroscope Rate Module

The ADIS16100 yaw-rate gyro had a SPI interface. The digital data available at the SPI port was proportional to the angular rate about the axis normal to the top surface of the package. A serial clock SCLK (at 160khz) was sent to the gyro, as well as the command to return the angular rate.
The gyroscope data out was supposed to return the angular rate. However, the angular rate returned would sometimes spike for no discernible reason, causing the light-saber to appear to spin rapidly on the screen. Careful scrutiny of the gyroscope inputs and outputs on the logic analyzer did not solve the mystery. The SCLK, CS, DIN values were according to specification.

Video Input and Marker Detection

Video Input and ZBT Memory

The video_module took camera input and math module light-saber coordinates and output colored XVGA video output onto the screen. video_module included video_output, a module that calculated and drew the light-saber as a sprite.

The video input module decoded camera NTSC video input and buffered the resulting luminance-chrominance (Y, Cr, Cb) data into the ZBT. 18 bits were buffered to the ZBT for each video pixel; 6 bits each for Y, Cr, Cb. I chose 18 bits per pixel because the ZBT was 36 bits wide; each ZBT write wrote data for two pixels to memory. On video output, the video input module converted the Y Cr Cb pixel values into RGB, and stretched the image to fit the whole 1024x767 XVGA screen. Each pixel from the camera became a cluster of four pixels in the video output. I based marker detection mainly on the Cr value of each pixel.

The 6.111 staff-provided modules were used to decode the camera's NTSC video and write the video

data to the ZBT. Modifications were made to store color video information in the ZBT. The Virtual Juggling project's method of calculating ZBT address was used to down-sample output video to fill the whole screen.

Marker Color Matching

The marker_match module determined whether each pixel in the VGA stream was a marker pixel or not. Each pixel's color determined whether a pixel was a marker pixel. Through casual experimentation in the lab, I chose red as the marker color. A pixel was a marker pixel if its Cr and Y values exceeded the Cr and Y threshold values. The user could increment or decrement the thresholds with buttons. The Cr threshold was the more critical threshold; the Y threshold made very little difference in marker detection.
The camera sometimes exhibited noise in the form of red specks scattered across the image. Noticing that the specks were small and isolated, I filtered out the red specks by signaling a marker pixel only when it belonged to a clump of 7 or more marker-colored pixels on the video output. This filtering idea was suggested by Michael Price.

Marker Position

The marker position module determined position of the marker by taking the average of the position of all the marker pixels.

For each frame, the marker position module takes the VGA stream, hcount and vcount signals, and calculated the total number of marker pixels, as well as the sum of their x coordinates and the sum of their y coordinates.

The marker position module divided the sum of the coordinates by the total number of markers to obtain the average coordinates. The average position was the position of the marker. At the end of every frame, the marker position was stored in a register, to be drawn in the next frame. The position of the marker was a frame late, but this was not noticeable to the human eye.

Math

Pose Extraction

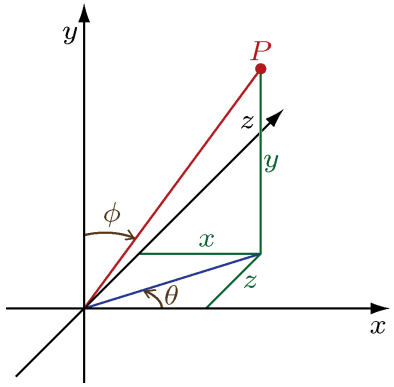
The goal of all these measurements is to end up with two angles that represent the orientation of the lightsaber:

- phi, angle "declined" from the vertical axis
- theta, angle "twisted" around the vertical axis

The PWM signals from the ADXL213 are converted into numerical quantities representing the apparent direction of gravity at the accelerometer. We integrate the angular velocity from the ADIS16100 and rotate the x- and y-components of acceleration by the resulting angle, correcting for "twist" in the lightsaber handle:

$$
\begin{aligned}
A'_x &= A_x \cos\theta_G - A_y \sin\theta_G \\
A'_y &= A_x \sin\theta_G + A_y \cos\theta_G
\end{aligned}
$$

Then the pose angles, phi and theta are computed as follows:



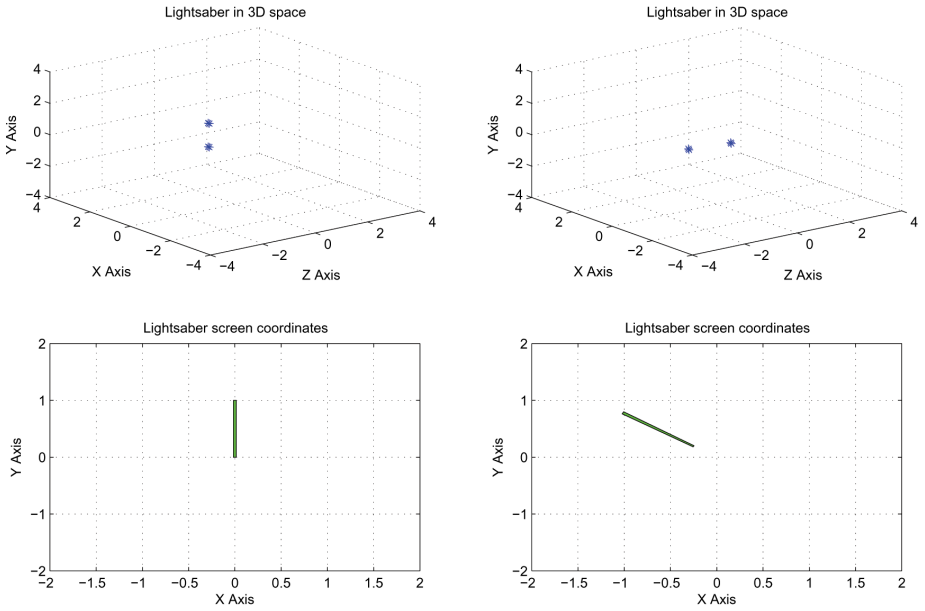
$$
\begin{aligned}
\phi &= \sin^{-1} A'_x \\
\theta &= \sin^{-1} \frac{A'_y}{\cos\phi}
\end{aligned}
$$

This model depends on a quasi-static assumption: the center-of-mass acceleration of the lightsaber must be small in comparison to gravity. Avoiding this assumption would require additional accelerometers (with wider dynamic range) oriented about different axes. Rapid motions of the lightsaber will cause errors in the estimated tilt angles. In practice, we found that this is an acceptable assumption; especially because this model is memoryless and immune to drift.

Transformation

The lightsaber boundaries are initially in its local coordinate frame. We store each point as a column vector in [x; y; z; w] format, so that the coordinates form a convenient 4x4 matrix.

$$
\mathbf{X} = \begin{bmatrix} x_0 & x_1 & x_2 & x_3 \\ y_0 & y_1 & y_2 & y_3 \\ z_0 & z_1 & z_2 & z_3 \\ w_0 & w_1 & w_2 & w_3 \end{bmatrix} = \begin{bmatrix} -W & -W & W & W \\ 0 & L & L & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}
$$



The first stage of the transformation is to transform each point into the global coordinate frame (3-D space). The appropriate matrix, R, is a rotation of phi about the x-axis followed by theta about the z-axis:

$$\begin{aligned}
\mathbf{R} &= R_z(\theta)R_x(\phi) \\
\mathbf{R} &= \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
\mathbf{R} &= \begin{bmatrix} \cos\theta & -\sin\theta\cos\phi & \sin\theta\sin\phi & 0 \\ \sin\theta & \cos\theta\cos\phi & -\cos\theta\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\end{aligned}$$

The second stage, linear projection, squeezes the trapezoidal viewing volume of the camera into screen coordinates, preserving a proper perspective. This technique is used in 3-D graphics systems such as OpenGL.

$$\begin{aligned}
\mathbf{P} &= \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix} \\
\mathbf{P} &= \begin{bmatrix} \frac{z_{near}}{x_{lim}} & 0 & 0 & 0 \\ 0 & \frac{z_{near}}{y_{lim}} & 0 & 0 \\ 0 & 0 & \frac{z_{near}+z_{far}}{z_{near}-z_{far}} & \frac{2z_{near}z_{far}}{z_{near}-z_{far}} \\ 0 & 0 & -1 & 0 \end{bmatrix}
\end{aligned}$$

The final screen coordinates are computed by dividing the x- and y-coordinates of each point by its w-coordinate:

$$\mathbf{x_i} = \mathbf{PRx_i w_i^{-1}}$$

These coordinates map to a screen space of [-1,1] in the x and y directions, regardless of the shape of the 3-D viewing volume.

Pose Module

The pose module computes estimated tilt angles from the raw accelerometer and gyroscope readings, as described:

$$\begin{aligned} A'_x &= A_x \cos\theta_G - A_y \sin\theta_G \\ A'_y &= A_x \sin\theta_G + A_y \cos\theta_G \end{aligned}$$

$$\begin{aligned} \phi &= \sin^{-1} A'_x \\ \theta &= \sin^{-1} \frac{A'_y}{\cos\phi} \end{aligned}$$

Unlike the video output module, the pose module does not face difficult timing demands because the update rate for the accelerometer is only 1 kHz. Hence we use a simple architecture with all of the computations performed by combinational logic. The input is read into a set of registers (accel_x_width, accel_y_width, accel_x_period, accel_y_period, gyro_reading) on every cycle. At this point a counter starts to increment the number of clock cycles that have passed. The result of the multiplications, additions, divisions, sine and cosine lookups (shown in detail in pose.v) is valid and correct after around 45 clock cycles. After 64 clock cycles the counter resets, the resulting estimates new_phi and new_theta are loaded into output registers, and new measurements are loaded into the input registers.

Math Module

Control

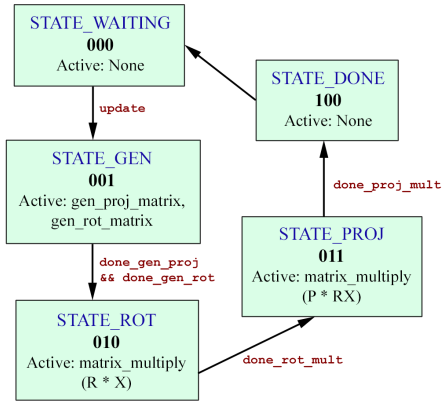
The math module's job is to generate screen coordinates for the lightsaber. We update these coordinates after every gyroscope measurement (10 kHz). Because of this slow speed, and the number of calculations required to generate and multiply the appropriate matrices, the math module uses a streamlined memory-based architecture. The computation involves one active division and twelve 18x18-bit multipliers, and takes about 430 clock cycles (6.5 us) to complete.

The four corners of the lightsaber are represented in column vector format as a 4x4 matrix. The numbers (distances in meters) are represented in an 18-bit fixed point format. With four bits before the decimal point, the format can handle values between -8 and 8 and has a resolution of about 0.25 mm. Each coordinate is stored in an 18x16 block memory, and the math module contains 5 of these memories:

- lightsaber coordinates X (saber_coords_18x16.v) in ROM
- rotation matrix R
- projection matrix P
- rotated coordinates RX
- screen coordinates PRX

Operations within the math module are arbitrated by an instance of math_controller. This controller functions as a major FSM, whereas the other components of math (gen_proj_matrix,

gen_rot_matrix, and matrix_multiply) are minor FSMs that respond to its commands. The controller maintains a state variable representing which stage of the computation the module is in: waiting, generating matrices, applying rotation, applying projection, or done. It drives the appropriate signals to its minor FSMs according to the state. The process of computation is started by driving the controller's update line high; after all of the minor FSMs have responded to their own update commands in sequence, the done line is driven high for one clock cycle.



Math state diagram.

To avoid using dual-ported memories, the math module does not read and write matrices simultaneously. Instead, it uses single-ported memories with the address lines switched by a multiplexer. For example, when gen_rot_matrix is active, the write-enable line for memory R (memmode_R) is driven high, and R uses the address provided by gen_rot_matrix. After gen_rot_matrix completes, the write enable is turned off and R uses the read address requested by matrix_multiply. This mechanism is used for all five matrix memories within the math module.

Matrix Generation

Two independent minor FSM modules, gen_rot_matrix and gen_proj_matrix, are started simultaneously at the beginning of the math update cycle. Each of these modules writes a series of values to memory (corresponding to the matrices given in section 2.1) before raising their own done signals. The math_controller module changes state after both matrices R and P have been written to memory.

An internal counter in gen_rot_matrix cycles through the matrix indices (0 through 15) once after receiving an update command. The angles computed by pose, phi and theta, are used as inputs into two sine/cosine lookup tables. Four multipliers compute the required combinations of sin(theta), cos(theta), sin(phi) and cos(phi). These results are loaded into a register connected to the R memory's wdata line on every clock cycle, and the address lines are delayed by one cycle so as to match the appropriate data. Hence the module writes all 16 values in the space of 17 clock cycles, after which the write enable is driven low.

Computing the projection matrix (gen_proj_matrix) takes much longer because of the delay inherent in division. Since high speed is not a priority, we use the smallest Coregen divider, which (for two 18-bit inputs) has a latency of 47 clock cycles. Twelve of the coefficients in P are constants (0 or -1) and are loaded into the wdata register after just one clock cycle, but the other four have to wait for division to finish. The module uses a simple state machine to wait 64 cycles whenever it detects that a division result is needed for the next value. We combine the lower 4 bits of each quotient with a 15-bit

remainder to generate an 18-bit numerical result in the same format as the inputs. After all values have been written to the P memory, the math_controller module moves on to the first multiply operation.

Matrix Multiplication

A single module, matrix_multiply, is instantiated twice - first to multiply the rotation matrix R by the lightsaber coordinates X, and again to multiply the projection matrix P by the rotated coordinates R*X. This module computes the product of two 4x4 matrices, with results in the same 18-bit format, performing one scalar multiplication at a time for 64 clock cycles.

Multiplying two 4x4 matrices A and B requires computing 16 output values, and each output value is the dot product of a row of A with a column of B - a total of 64 scalar multiplications (as shown below). Instead of calculating the appropriate addresses for the input memories, the module simply follows a pattern specified in ROM (indices_A_4x64 and indices_B_4x64). The matrix_multiply module does one such multiplication per clock cycle, accumulating the result of the dot product in a 36-bit register. On every 4th clock cycle, the appropriate bit range [31:14] of this register is written to memory and it is replaced with the first component of the next dot product. A 2-cycle delay line for the output address is needed to account for the delay in reading a value from input memory (1 cycle) and loading the result of a multiplication into the wdata register (1 cycle).

$$\mathbf{X} = \mathbf{AB} = \begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 & a_7 \\ a_8 & a_9 & a_{10} & a_{11} \\ a_{12} & a_{13} & a_{14} & a_{15} \end{bmatrix} \begin{bmatrix} b_0 & b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 & b_7 \\ b_8 & b_9 & b_{10} & b_{11} \\ b_{12} & b_{13} & b_{14} & b_{15} \end{bmatrix}$$

Indices in **A** (row 2) — Indices in **B** (column 3)

$$x_6 = a_4 b_2 + a_5 b_6 + a_6 b_{10} + a_7 b_{14}$$

The matrix multiplication process takes a total of 67 clock cycles, after which matrix_multiply drives its done line so math_controller can proceed with the next step. The reason for using a 36-bit register internally is to reduce the impact of numerical precision errors, but the output of this module does not exactly match results predicted by floating-point calculations. In practice, however, the resolution is more than appropriate since it takes an error of 40 LSBs to shift any point by 1 cm.

Output Access

The design of the math module lends itself to easy integration with other parts of the lightsaber generator. Because the screen coordinates of the lightsaber points are stored in the PRX RAM, they can be accessed at any time when the math module is not actively writing values. As with any other memory, data is returned on the next clock edge following a particular address. The addresses map to the relevant screen coordinates as follows:

PRX **Address**	**Coordinate**
0	x_0
1	x_1
2	x_2
3	x_3
4	y_0
5	y_1
6	y_2
7	y_3

Because it takes around 350 clock cycles after an update command for math_controller to begin writing the final coordinates, the video output module can read and store the values of the x- and y-coordinates starting from the same exact time (the gyroscope chip select output, for example).

Video Output

Video output consists of four modules: *slope check*, *video output*, *lines*, and *orientation*. *Video output* is the top-level "sprite" module that draws the pixels of the light-saber. *Lines* and *orientation* are internal to *video output*, while *slope check* is external to it.

Slope Check

Inputs
vclock – 65 Mhz pixel clock
data_ready – signal from math module that new set of saber points has been written to RAM
data_in – point coordiantes from RAM

Outputs
addr – RAM memory address to read from
x0, y0, x1, y1, x2, y2, x3, y3 – normal-space coordinates of four points of saber (18-bit signed, with 14 bits of precision)

This module, external to the main *video output* module, processes newly calculated coordinates of each of the four saber points. At every high pulse of data_ready from the math module, it reads from RAM eight new coordinate values: x0, y0, x1, y1, x2, y2, x3, y3. These values are stored in registers and passed to the main *video output* module.

In addition, if any of the points create vertical lines, i.e., if any of the x-values are equivalent, one of the x-values is offset by a small amount before being passed to the main video module. This eliminates the problem of having to deal with drawing lines with infinite slopes.

Orientation

Inputs
x0, y0, x1, y1, x2, y2, x3, y3 – normal-space coordinates of saber points, passed from slope check module

Outputs
up, right – denotes screen orientation of saber

This module is internal to the main video output module. It assigns up = 1 if the saber tip should be pointed upwards on the screen, and -1 otherwise. Likewise, it assigns right = 1 if the saber tip should be pointed to the right side of the screen, and -1 otherwise. This way, the saber image is assigned one of four orientations: up-right, up-left, down-right, and down-left.

Lines

Inputs

vclock – 65 Mhz pixel clock
up, right – denotes orientation of saber
hcount_in – incoming XVGA signal denoting horizontal index of current pixel (0..1023)
vcount_in – incoming XVGA signal denoting vertical index of current pixel (0..767)
x0, y0, x1, y1, x2, y2, x3, y3 – normal-space coordinates of four saber points, passed from slope check module
BASE_X, BASE_Y – pixel-space coordinates of saber base, passed from marker-detection module

Outputs
value01, value12, value23, value03 – value of each of four boundary lines in pixel space
vcount01, vcount12, vcount23, vcount03 – scaled vcount values (see description below)

This module is also internal to the main video output module. It performs calculations of the four boundary lines of the saber image.

First, the normal-space point coordinates (x0, y0, x1, y1, x2, y2, x3, y3) are converted to pixel-space values (x0_pixels, y0_pixels, etc.). Then, a pipe-lined calculation of the line equation y = mx + b is performed. To avoid the high latency of division, the "y" and "b" terms are multiplied with the denominator of "m." Below is the right-hand equation for the line connecting points 0 and 1:

$$-1*right*(y1 - y0)(hcount – x0\_pixels – BASE\_X) + right*(x1 – x0)(BASE\_Y – y0\_pixels)$$

Since the slope of a line connecting two points may be positive or negative depending on the saber's orientation, multiplication by the inputs "right" and "up" yield the correct slope sign. The slope is further multiplied by -1, as the vertical axis is positive downwards in screen pixel space. This right-hand side of the equation is assigned to the outputs value01, value12, value23, and value03.

The left-hand side of the line equation is as follows, for the line connecting points 0 and 1:

$$right*(x0 – x1)*vcount$$

These are assigned to the output values vcount01, vcount12, vcount23, and vcount03.

Video Output

Inputs
test01, test12, test23, test03 – wired to labkit buttons for testing and debugging modes
vclock – 65 Mhz pixel clock
reset
x0, y0, x1, y1, x2, y2, x3, y3 – saber point coordinates from slope check module (18-bit signed, with 14 bits of precision)
hcount_in - horizontal index of current pixel (0..1023)
vcount_in - vertical index of current pixel (0..767)
hsync - XVGA horizontal sync signal (active low)
vsync - XVGA vertical sync signal (active low)
blank - XVGA blanking (1 means output black pixel)
base_x, base_y – pixel coordinates of base of saber, from marker-detection module

Outputs
phsync - output horizontal sync
pvsync - output vertical sync
pblank - output blank signal
pixel - saber pixel

This is the top-level video "sprite" module that draws the saber image. Depending on the orientation of the saber, the pixels above or below a certain line is colored. Specifically, the scaled vcount values (vcount01, vcount12, vcount23, vcount03) are compared to the values of the lines in pixel space (value01, value12, value23, value03). Applying this condition to all four boundary lines results in the saber image area being colored in.

XVGA

The xvga.v module, running at a 65Mhz pixel clock, was provided by the 6.111 staff to generate VGA signals for a 1024x768, 60Hz display.

DISCUSSION

External Device Input

In collecting input data from the accelerometer, glitches were present in the pulse width modulation, as described in the "Accelerometer Module" description.

Our system's ability to compensate for the yaw (twist) of the lightsaber handle was impaired by the poor performance of our gyroscope. To the best of our knowledge, the interface we designed (gyro_rate module) satisfied the ADIS16100 interface specifications. We could observe the command (writing to the control register) being sent to the sensor's DIN pin, and the output packet arriving from DOUT as specified in the datasheet. However, the numerical values of angular velocity decoded from DOUT did not make sense. The gyroscope reported a value of 2048 (zero velocity) most of the time, even as it was being rotated. We also observed occasional spikes near the limits of its range (300 deg/sec). There was insufficient time to disassemble the lightsaber handle, replace the sensor and carefully diagnose the problem. We used a switch on the labkit to disable gyroscope measurements (setting the angle estimate to zero). The change required the user to hold always hold the handle in the same way; tilt measurements based only on the accelerometer would become meaningless if the saber was twisted or re-gripped. We look forward to understanding the cause of this malfunction.
my issues

We were generally satisfied with the performance of the lightsaber handle interface - it was convenient to have a single Cat5 cable, and the incoming signals were clean enough despite the long length of the tether. A few minor modifications for a future project would include:

- use of a different type of cable with 8 twisted pairs
- wireless interface

- modular sensor wiring
- PCB for decoupling and signal routing at the labkit

The Cat5 cable did its job, but we did not take advantage of its twisted-pair construction. In Ethernet wiring, 4 signals are transmitted differentially - one across each twisted pair, so that noise and interference (present in equal quantities at each wire) cancels out. We used all 8 wires for different signals, including power and ground. As a result, there was significant high-frequency coupling between wires; this was most noticeable on the serial interface to the gyroscope. Slowing down the signals with load capacitors worked well enough to remove glitches from the digital signals, but that is far from an ideal solution. Using eight twisted pairs, or individually shielded wires, would mitigate the problem.

A wireless interface (using battery power for the lightsaber) would be ideal, making this device as accessible and intuitive as a Nintendo Wiimote. Such an interface could easily be constructed using a ZigBee or similar radio module and a PIC or AVR microcontroller on each end, interfacing the sensors to the transmitter and the receiver to the labkit. The implementation of this interface, however, is not relevant to the rest of the digital system and would be best left to a 6.115 or 6.101 project.

Video Input

Several different colors were tried for the marker. The camera used was a security camera, and was very sensitive to light. Using LEDs for the marker actually caused over-saturation in the camera – instead of seeing a bright green LED, a white dot would appear on the output video, because the LED is too bright. A less sensitive camera and the choice to use red colored paper worked well for marker detection.

Math

We did not have time to apply "normalized device coordinates" in the math module. While the lightsaber beam does (correctly) appear shorter as the handle is tilted towards or away from the camera, the perspective is incorrect because variation of the w-coordinates has not been divided out from each point. A minor modification to include this feature would significantly improve the realism of our lightsaber display.

Furthermore, the pose estimation problem turned out to be much more difficult than anticipated. Firstly, the measurements from the sensors we use can not distinguish between "up" and "down" - any given set of measurements could be caused by two different poses. Secondly, it was difficult to achieve the desired numerical accuracy in pose estimates with reasonably sized (10x1024) lookup tables for the inverse trigonometric functions. The solution to these problems would be to reformulate the entire scenario to use a different set of variables (perhaps direction vectors or quaternions instead of tilt angles). However, such a solution would inevitably require more sensors and more complicated signal processing, approaching the complexity of a full-blown IMU. We feel that the pose estimation approach described above approaches the limits of accuracy and drift performance for this reasonably simple sensor platform.

Video Output

Our ability to refine the performance of the lightsaber generator was constrained by the limited time available for completing the project. The finished system displays a quadrilateral that moves as the user tilts the handle, but the coordinates of this quadrilateral are not as realistic as we would have liked because of imperfections both in the sensors themselves and the computational process.

Line calculations involved in drawing the saber image were pipe-lined. Nevertheless, in the final product, artifacts of intermediate values were drawn as moving specks outside of the saber quadrilateral. The cause of this is still uncertain, but further pipe-lining might have resolved the problem.

One problem that was realized in the last stages of the project was the discrepancy between the use of normal-space and pixel-space in output of the saber image. Unlike pixel space, our normal space used is not "squared," i.e., although our screen is rectangular, both the vertical and horizontal axes of the normal space ranged from -1 to 1. However, the normal space points are used directly in the saber image line calculations. This may have created a negligible stretching of the saber quadrilateral. Unfortunately, there was not enough time to fix this problem.

A feature that was also left out due to lack of time was support for multiple lightsabers. Nothing about the design of this system prevents the addition of another set of modules operating in parallel; the second saber could use a different marker color, and correspond to a different color of beam. The second lightsaber could be constructed identically to the first one, using an additional eight communication lines to communicate with the labkit. In addition, saber image effects such as glow, blur, and anti-aliasing were not implemented due to time spent on debugging the basic modules. The same applied for sound effects coupled with the motion of the saber.

REFERENCES AND ACKNOWLEDGMENTS

6.111-staff provided Verilog modules
<http://web.mit.edu/6.111/www/f2005/index.html>

Analog Devices. ADXL213 Data Sheet, ADIS16100/PCB Data Sheet

Special thanks to Gim P. Hom and Analog Devices.

An appendix of Verilog code is attached separately.