6.111 Final Project:
Wireframe Simulator

Sarah Cheng

Wireframes provide a "skeleton" for three-dimensional models by outlining the edges and vertices of the object. In this project, we attempt to create an interactive wireframe simulator in which the user can load their own models, rotate and move them around, and view the results in real time. The simulator uses simple transforms and perspective projection to produce the images.

1 Introduction

Wireframes are the most basic, fundamental representations of three-dimensional objects. They provide the foundations on which more sophisticated graphics techniques and models are built. In this project, we attempt to implement a basic wireframe simulator, along with very basic interactive capabilities. Users are allowed users to load their own figures, and rotate, pan around, and zoom in and out on them, viewing their objects at different angles and perspectives. After all, a two-dimensional representation of a static three-dimensional object just "isn't" 3D!

We will begin by talking about the proposed behavior of the simulator in Section 2. Section 3 lays out the original plans for implementing the functionality. Section 4 relates the many trials and tribulations (and there are many!) while building the project. The report concludes in Section 5 with reflections on the overall experience.

2 Functionality

Had the project been successful, it would allow users to input the coordinates of a three-dimensional object and specific viewing parameters, and it would display a wireframe model of the object. The user can then interact with the model by pressing buttons, allowing them to rotate, pan around, and zoom in and out on the model.

2.1 ROM Input

Object coordinates and viewing parameters are specified through a user-provided ROM. The ROM is 99 bits wide; the three leftmost bits function as opcodes, indicating the type of data, while the other 96 bits are "arguments." The ROM is delimited by a special opcode, `111`, designating "end of file." All instructions after this line are ignored. The opcodes and argument formats are listed in Table 1.

The coordinates should be specified as 16-bit signed numbers. The most significant bit is the sign bit, the next four the integral part, and the last 11 the fractional part. In other words, only the numbers in the range from -16 to 15.9995 can be entered. Segments are specified by concatenated pairs of endpoints, and the points themselves are concatenated ordered triples.

The viewing plane is in the form $z = a$, where a is the user-specified constant. This representation is sufficient, because, given any arbitrary plane, the axes can be rotated so that the plane satisfies $z = a$. For simplicity, we require that $z_{eye} > a$, and that $z_{eye} \geq 0$.

2.2 User Interaction

While the system is running, the user can interact with the simulator via push buttons. The simulator can rotate the scene, pan around, and zoom in and out, all in real time. Users can zoom in and out with buttons 1 and 0, respectively. Panning and rotating are done with the four arrow buttons. Turning switch 0 on activates rotation when one of the arrow buttons are pressed; leaving it off enables panning.

3 Implementation

Again, the implementation described here is only the original plan of action. Most of the code is in a "broken" state and does not reflect any of the functionality described here.

3.1 Data Representation

3.1.1 Numbers

Most numbers are represented with a fixed-point scheme using 16 bits of data. The 11 least significant bits represent the fractional part of the number, while the next 4 represent the integral part. The last bit is reserved as a sign bit. All numbers use 2's complement representation. This means that all values are restricted to the range from -16 to 15.9995. For sake of simplicity, no overflow checking or handling is ever done.

Addition and subtraction do not present problems; however, multiplication and division require some special care. Addition and subtraction work correctly bit-by-bit, and the sum or difference of two 16-bit numbers is simply another 16-bit result. Multiplication of two 16-bit numbers yield a 32-bit product, where the bottom 22 bits represent the fractional part, the next 4 the integral part, and the top 2 bits representing the sign. To keep lengths of numbers constant and manageable, the sign bit, the bottommost 4 bits of the integral part, and the top 11 bits of the fractional part are extracted after each multiplication. Division is done through a divider module provided by Coregen. The module is capable of signed division with 12 fractional bits, with the top bit denoting the sign of the entire expression and the bottom 11 bits corresponding to the fractional value. Thus, to maintain the 16-bit numerical representation, the quotient is left-shifted 11 bits while maintaining its sign, and the bottom 11 bits replaced by the bottom 11 bits of the divider's fractional output.

3.1.2 Coordinate system

Before we proceed, we must make the distinction between mathematical coordinates, screen coordinates, and memory coordinates. Screen coordinates represent the pixels on the display monitor, with $(0, 0)$ representing the top left pixel. Thus, all screen coordinate values are integers. The x-coordinate increases to the right, while the y-coordinate increases downwards.

Mathematical coordinates, on the other hand, are simply rectangular coordinates with points plotted along a set of right-handed orthogonal axes. The coordinates can take any real-numbered values. The xy-plane is perpendicular to the plane of display, with the x-coordinate increasing to the right and the y-coordinate increasing *upwards*. The z-axis points perpendicularly out of the screen, increasing towards the viewer.

Memory coordinates are identical to screen coordinates, except that, because each memory bit represents a 2x2 square of pixels, they are scaled down by a factor of 2.

All 2D and 3D coordinates are represented as concatenated busses of 16-bit numbers. For example, $(0, 1, -1)$ would be represented as `48'h0000_0800_F800` (recall that the top bit is for sign, and the next four bits are for the integral parts of the number). Line segments are represented either as two separate points, or as a pair of concatenated points (e.g., a 96-bit bus for a pair of 3D coordinates).

3.2 Modules

The simulator operates on each segment of the input object in a sequence of stages. At a reset, the initialization module parses each line of the input ROM, passes the user-specified values to the appropriate modules, and fills a small block of RAM with the segments of the object, specified by x-, y-, and z-coordinates of the endpoints (see Table 1). When all segments have been written, the system enters normal operation.

During normal operation, each segment passes through three stages every frame (see Figure 1). In the first stage, the transformation module reads and processes one segment at a time from the

coordinates RAM. The module rotates, pans, or zooms around the image depending on user input. The output of the transformation module is passed on to the next stage, the projection module, whose purpose is to project a three-dimensional image onto a two-dimensional plane (which, in this case, is the monitor screen). These two-dimensional coordinates are then passed on to the last stage, which handles translating these mathematical coordinates into screen pixel coordinates and displaying the segments.

Because each stage takes different amounts of time – even variable amounts of latency, in the case of the display module – there must be a control module to coordinate all the other modules. Otherwise, a faster module can process and output more data than the next module can handle, resulting in data loss. Each module reports to the control module when it is done with its current input, and when all are done, the control module advances the data in each stage one step forward. In other words, the wireframe simulator uses a 3-stage pipeline with variable latency at each stage.

3.2.1 Control

The control module orchestrates data flow between modules by sending and receiving handshaking signals to and from all modules. The modules report a `done` signal to the control module to signal that their output data is ready and valid. When the control module hears `done` signals from all modules under its control, it sends a `start` signal to all modules, allowing data to advance to the next stage. In addition, the control module receives line and pixel counts from the VGA controller to issue `newframe` signals at the end of each frame. This signal is useful for the display buffer controller to swap the double buffers, for example, and for the transformation module to start reading segments from RAM back from the beginning again.

The control module also comes into play after the user resets. The user reset only directly goes to the initialization and control modules; the other modules are reset by the control module after initialization completes. While the system is in initialization mode, the control module lets the initialization module read each input from ROM and write each segment to the input RAM. During this phase, no `start` signals are sent. When the control module receives a `done` signal from the initialization module, it exits initialization mode, sends a `module_reset` signal to all other modules, and proceeds in normal operation.

3.2.2 Initialization

When the user presses the reset button, the system enters initialization mode. Here, the initialization module reads and parses one line at a time from a user-provided ROM. The first three digits indicate the type, while the rest of the data are akin to "arguments." The eye, viewing plane, and viewing window boundaries are latched as outputs to the respective modules (see Figure 1). The rest are definitions for line segments, whose coordinates are copied (minus the 3-digit opcode) to the coordinates RAM.

When the initialization module reaches opcode `111` representing "end of file", it stops reading and sends a "done" signal to the control module. It then holds its address, data, and write enable lines to the coordinates RAM under high impedance, so that the transformation module may start accessing it.

3.2.3 Transformation

All rotating, panning, and zooming actions are done by transforming each vertex of the object while keeping the screen and eye stationary (as opposed to, for example, keeping the object stationary and moving the eye and viewing planes). The transformation module is responsible for recalculating

Opcode	Description	p_1			p_2		
`d[98:96]`	–	`d[95:80]`	`d[79:64]`	`d[63:48]`	`d[47:32]`	`d[35:16]`	`d[15:0]`
`000`	Eye position	x_e	y_e	z_e	–	–	–
`001`	View plane, $z = a$	a	–	–	–	–	–
`010`	Window boundaries	x_l	y_l	x_h	y_h	–	–
`100`	Line segment	x_1	y_1	z_1	x_2	y_2	z_2
`111`	End of file	–	–	–	–	–	–

Table 1: ROM Input Formats

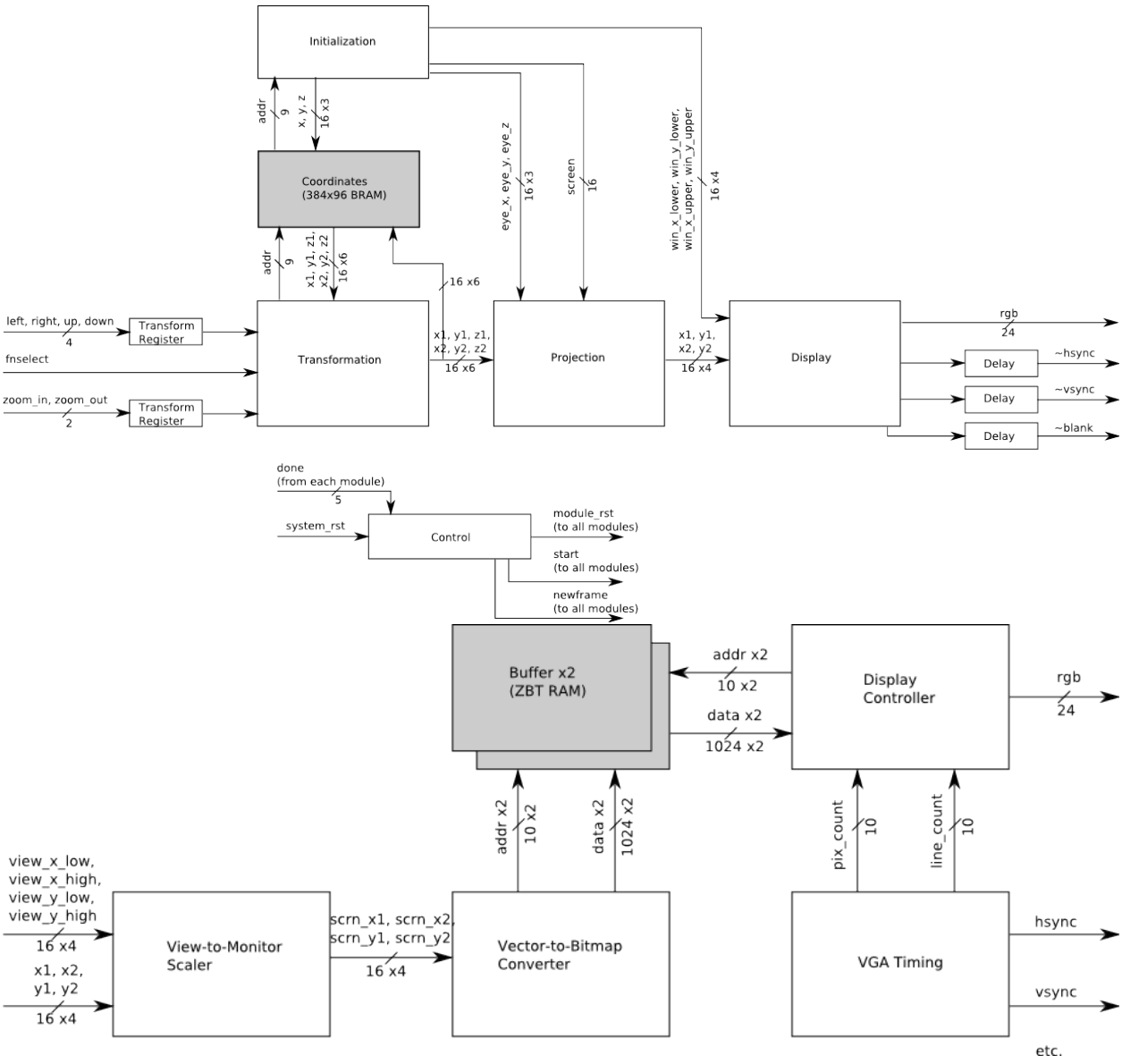

Figure 1: Module block diagrams

High-level block diagram (top) and a more detailed view of the display modules (bottom)

all these coordinates. The transformed coordinates are written back into the coordinates RAM to reflect the updated locations.

Again, users can specify which transformation to apply using buttons and the function select switch. These inputs are first debounced and synchronized, then passed to a transform request register. The register holds user requests until next frame. At a `newframe` signal from the control module, this register passes its value to the main transformation module and resets its status to unpushed; otherwise, it continues to output its old value. Without this register, if a user pushes button in the middle of a frame, some segments would remain in their old positions while the rest of the segments would move to their transformed positions. By holding off user requests until the next frame, transformations can be applied to *all* segments within a frame.

When the control module issues a `start` signal, the transformation module writes the results of the previous segment to RAM, and reads and latches the data at the next address. It takes the outputs from the transform request registers and performs the appropriate transformation, outputting the result to the projection module. For the sake of simplicity, only one transformation is performed per frame. In other words, if a user pushes both the "pan left" and "zoom in" buttons at the same time, only the panning transformation will be done. The transformation module stops reading from new addresses when it encounters all 1s, signifying end of data. The address counter is set back to 0 at a `newframe` signal.

Rotations are performed by a constant angle ϵ each time, so that $\cos\epsilon$ is represented as `16'b111_1111_1111` (≈ 0.99951171875 in our fixed-point representation). Therefore, $\sin\epsilon = \sqrt{1-\cos^2\epsilon} \approx 0.000976$, or approximately `16'b000_0000_0010`. Then, rotating up/down yields the transformed coordinates (x', y', z'), where

$$(x', y', z') = (x, y\cos\epsilon \pm z\sin\epsilon, \pm y\sin\epsilon + z\cos\epsilon)$$

and similarly, rotating left/right yields:

$$(x', y', z') = (x\cos\epsilon \pm z\sin\epsilon, y, \pm x\sin\epsilon + z\cos\epsilon)$$

Panning simply adds or subtracts a constant $\delta = 0.0625$ (i.e., `16'b0000_1000_0000`) to the x- or y- coordinates. Namely, panning up/down gives:

$$(x', y', z') = (x, y \mp \delta, z)$$

and panning left/right gives:

$$(x', y', z') = (x \pm \delta, y, z)$$

Zooming in and out is just another translation in the z-direction. More specifically, zooming adds or subtracts 2δ to the z-coordinate. Mathematically, zooming in/out can be described as:

$$(x', y', z') = (x, y, z \pm 2\delta)$$

While zooming in or rotating, there is the possibility that vertices cross the viewing plane. This situation is allowed, as the user may decide to zoom back out or rotate back later. This case is handled by the projection module.

3.2.4 Projection

The projection module takes the two 3D coordinates from the transformation module and applies perspective projection. The initialization module provides the locations of the eye and the viewing plane.

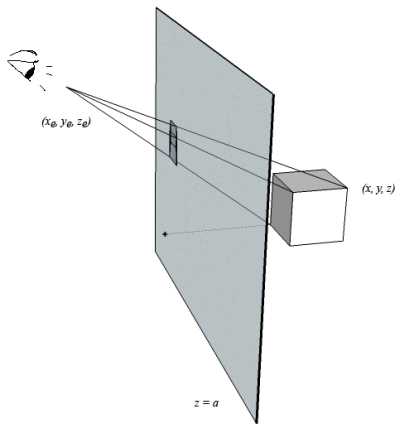

Figure 2: Perspective Projection

The idea behind perspective projection. The points of intersection between the lines (representing rays of light) and the screen can be solved using simple congruences presented below.

On a `start` signal, the projection module latches the output data from the transformation module; at all other times, the outputs are assumed to be invalid and are ignored. As the viewing plane is simplified to the form $z = a$, the point (x, y, z) can be projected onto the plane by solving the simple proportions (see Figure 2):

$$\frac{x_e - x_p}{x_e - x} = \frac{y_e - y_p}{y_e - y} = \frac{z_e - a}{z_e - z}$$

where (x_p, y_p) are the projected coordinates, and (x_e, y_e, z_e) is the location of the viewer. From this, we see that we can easily solve for (x_p, y_p):

$$x_p = x_e - \frac{z_e - a}{z_e - z}(x_e - x)$$
$$y_p = y_e - \frac{z_e - a}{z_e - z}(y_e - y)$$

The values of (x_p, y_p) are output to the display module.

As mentioned previously, it is certainly possible that the endpoints of the segment are in front of the screen, i.e., $z > a$. If both endpoints are in front of the screen, then the entire segment will not be displayed. If one of the two are in front – say, p_1 is in front and p_2 is behind, then only the part of the segment behind the screne is visible. Then, p_2 is projected as normally, while the other endpoint is the intersection of the segment with the viewing plane. The problem becomes analogous to the one above, with $p_1 = (x_1, y_1, z_1)$ replacing (x_e, y_e, z_e).

3.2.5 Display

The display stage is perhaps the complex of the three stages (see Figure 1. It consists of translating mathematical vector coordinates into pixel bitmaps, then writing these bitmaps into a display buffer, and finally reading the buffer and displaying its contents.

The display component was by far the most difficult and time-consuming part. See Section 4 for the first (and failed) design of the display module.

Double buffering and buffer controller. The display information is stored in a double buffer, so that the line drawing module writes to one of them while the display module is reading reading from the other. At a `newframe` signal, the two memory modules swap, so that the now-filled buffer is read and displayed on screen. A display buffer in RAM is ideal due to the random-access requirement of the line-drawing module.

The buffers themselves are BRAM modules that are 512 bits wide and 384 rows deep, with each bit representing a 2x2 square of pixels. Each bit is either 0, for an unlit pixel, or 1, for a lit. Thus, one entire row of pixels is read or written at a time.

The buffer controller manages which RAM module is displayed from and which is written to. It keeps an internal `ramsel` flag that toggles at every `newframe` signal. This flag determines which RAM block is accessed by which module. The module interacts with the display and line drawing modules via single address bus, data bus, and enable lines for each, and relays the address and data signals to the appropriate RAM blocks.

Writing to an address requires reading existing data from the address, taking its bitwise `OR` with the part of the segment to be written, and writing that result back into the address. If data were written directly, it would overwrite any previous pixels drawn there. However, this also means that a buffer needs to be zeroed after reading. To do this, after each read by the display module, each address location is overwritten with 0s. The buffer controller module also takes care of both the zeroing out and the `OR`ing operations.

Line drawing module. The line drawing module takes the 2D coordinate output from the projection module, scales it to the display screen (according to (x_{lower}, y_{lower}) and (x_{upper}, y_{upper}) from the initialization module), and figures out which pixels must be lit. This is by far the most difficult part, and the code for this module is still not complete.

Let $(x_1, y_1), (x_2, y_2)$ denote the two endpoints with $y_1 < y_2$, $(x_l, y_{l)}$ the lower left corner of the window in mathematical coordinates, and $(x_l, y_{l)}$ the upper right corner. Then, the memory coordinates $(x_{1,ram}, y_{1,ram}), (x_{2,ram}, y_{2,ram})$ can be expressed as

$$(x_{1,ram}, y_{1,ram}) = \left(\frac{x_1 - x_l}{x_h - x_l} \cdot 512, \left(1 - \frac{y_1 - y_l}{y_h - y_l}\right) \cdot 384\right),$$
$$(x_{2,ram}, y_{2,ram}) = \left(\frac{x_2 - x_l}{x_h - x_l} \cdot 512, \left(1 - \frac{y_2 - y_l}{y_h - y_l}\right) \cdot 384\right)$$

The main idea is to find the amount x changes in memory coordinates for every memory-coordinate change in y. Then, the difference between these are filled with 1s, and the entire row is written to memory at once. Figure 3 shows this graphically. Compared to calculating one pixel at a time, this potentially saves a lot of memory read/write time overhead, and is not too much more complicated to implement.

For every change in the y-direction in memory coordinates, y changes by $-(y_h - y_l)/384$ in mathematical coordinates. Similarly, an x-direction change in memory coordinates translates to a $(x_h - x_l)/512$ change in mathematical coordinates. In mathematical coordinates, for every change in y, x changes by $(x_2 - x_1)/(y_2 - y_1)$ (inverse of the slope). Then, for every *memory*-coordinate change in y, x changes by

$$-\frac{x_2 - x_1}{y_2 - y_1} \cdot \frac{y_h - y_l}{384}$$

in *mathematical* coordinates, or

$$-\frac{x_2 - x_1}{y_2 - y_1} \cdot \frac{y_h - y_l}{384} \cdot \frac{512}{x_h - x_l}$$

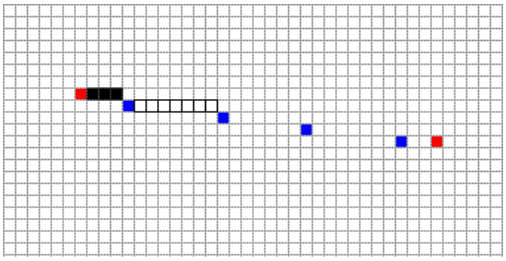

Figure 3: Line Drawing between Points

> The red squares mark the start and end points, while the blue mark every new change in y in memory coordinates. The highlighted squares on the second row show the positions that must be filled in with 1s.

in memory coordinates.

Using these, the module iterates down each value of y until the memory coordinate reaches $y_{2,ram}$. Then, x is filled until $x_{2,ram}$. Both mathematical and screen coordinates are incremented at each iteration. Even though only the integral part of memory coordinates is relevant for storage and display, the fractional part is kept to minimize the skew due to rounding.

Display module and VGA controller. The display module presents an address to the buffer controller, and waits the controller to perform all the interfacing with the memory. The address simply corresponded to $\lfloor y_{screen}/2 \rfloor$. Each pixel is colored green if the $\lfloor x_screen/2 \rfloor$th bit is 1; it is left black otherwise.

The VGA controller is identical to the controller implemented for Lab 4, except with numbers (and counter sizes) adjusted for 1024x768 display. Since the display module goes through the memory controller to read from RAM, there is a 6-cycle delay between the VGA's screen coordinate and the RGB output from the display module, not including the 2-cycle DAC delay. Thus, the `hsync`, `vsync`, and `blank` signals are passed through an 8-cycle delay before arriving at the labkit's VGA outputs.

4 Development and Testing

The initial plan was to implement the display modules first. Then, it would be easy to spot bugs in the transformation and projection modules, as the images would not look right. Given my awkward choice of fixed-point numerical representation, implementing the display first would give an easier, faster yes-or-no answer to whether the system was working properly.

Within the display stage, each submodule was implemented right-to-left. The VGA timing was resolved first, then the display module was set to always output green. Next, the display module was to interface with the buffer control, which would read from memory. The idea was that the bottom, most bare-bones layer should be tested first, and gradually add layers on top.

Initially, display buffering was to be done in ZBT RAM, with one bit of data to a pixel. In addition, the buffer control was only concerned with RAM chip selection and timing. It would have been up to the display module to zero out previous addresses, and the line drawing module to read previous data and apply bitwise `OR`.

Interfacing with the ZBT was when the trouble first began. The first issues appeared when the ZBT clock deskewer from the course website was added, in which the code refused to synthesize.

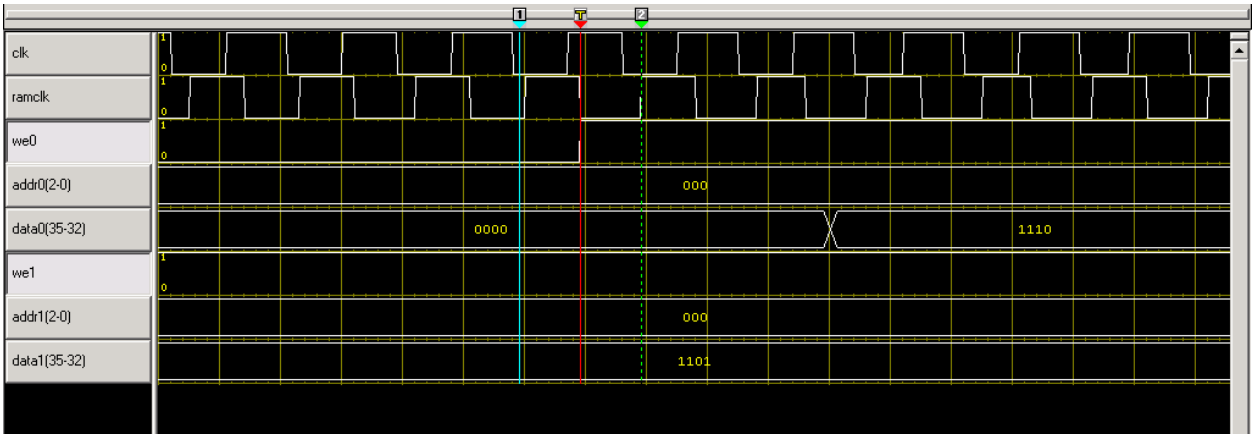

Figure 4: Logic Analyzer Output of ZBT Memory Tester

> Probing the memory pins directly using the logic analyzer. The timings are perfectly consistent with the data sheet, with a low `we` signal at time 1, and a a value of `0000` driven onto the data bus two cycles later. When the write enable is pulled high, data read from the same address is most definitely not `0000`.

This issue took about a day's worth of time to resolve, until Gim helped determine that the deskewer simply was not compatible with the pixel clock generator. The clock deskewer was scrapped, and an inverted pixel clock was simply used as the RAM clock.

In the meantime, the transform and projection modules were written, and quickly tested with ModelSim. However, its results were hard to decipher due to the choice of numerical representation. Panning and zooming seemed to simulate correctly at a quick glance, so these modules were set aside for the moment to focus on the ZBT again.

The buffer control module was simulated in ModelSim and its timing tweaked to allow consecutive reads and writes while maintaining all timing specifications. When this was synthesized again along with the display modules, it gave similar results. Finally, a separate memory tester project was created, with manual data, address, and write enable lines (and initially, a manual clock). The data seemed to be read correctly, as the same addresses returned the same value each time; the problem seemed to be that nothing was ever written. The timings and values were then more closely examined using the logic analyzer. No apparent problems were found; the timings seemed perfectly consistent with the data sheet (see Figure 4). The memory tester was synthesized on a different lab kit, with similar results.

Finally, two days before checkoff date, it was decided that no further progress can be made along this route, and the ZBT RAMs were dismissed in favor of built-in BRAMs. For simplicity, the RAM is made wide enough so that one row corresponds to entire lines of pixels. In order for two such BRAM blocks to fit, they were reduced to 512x384, so that each bit encoded four (i.e., 2x2) pixels of display.

As a result, many modules had to be rewritten; for example, the buffer controller timing no longer applied. The data width was no longer 36 bits, which also changed the addressing scheme. Other significant changes were made during the rewrite, such as delegating zeroing and `OR`ing write data tasks to the buffer controller. These changes made the line drawing and display modules much simpler, but introduced a slough of bugs too late into the project.

Other unforseen problems began to surface with these new changes, however. For example, the display module might have worked correctly with the older, 1 pixel/bit design, but the zeroing

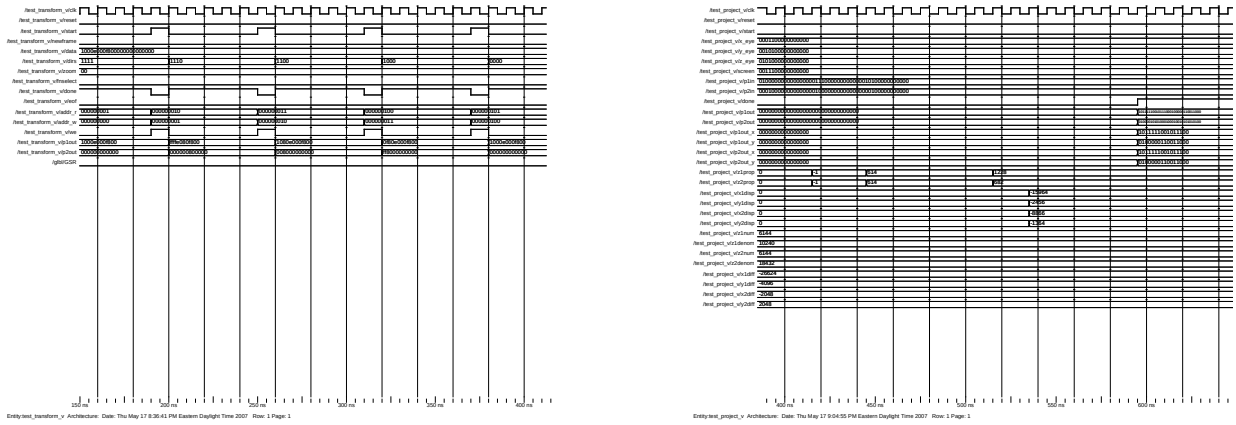

Figure 5: ModelSim Testing Waveforms

Last-minute changes that broke the transformation (left) and projection (right) modules further.

memory locations is now done by the buffer controller instead of the display module after each read. Since each memory location accounts for two rows of pixels now, every other line now appears completely blank. Workarounds for this were attempted, but none were successful.

In the course of writing this paper, more bugs were found in the rotation code of the transformation module. Attempts to fix these seemed to break the existing code even more, further delaying this paper (see Figure 5).

5 Conclusion and Reflection

This project was riddled with nothing but bugs and issues. However, it had been a rather valuable experience, as it had taught me exactly how `not` to approach a problem. For example, complete overhauls should not be considered so close to a deadline; even though the project may be at a complete dead-end, the chances of breaking the existing implementation are too likely.

This project was not all grief and anguish, however. At the beginning of the project, there were no plans of specifically implementing a graphics "pipeline." Nevertheless, it was extremely interesting to see how such a pipelined structure naturally emerged, simply out of my initial efficiency considerations. It is always much more worthwhile to develop these engineering concepts in a hands-on, interactive way, rather than reading about them in books or learning in lecture.