Pong: 2 Players, 1 Point Perspective

William Navarre
Emmanuel Akinbo

Table of Contents

Abstract

We planned to extend the lab 3 pong game to the case of two players on separate FPGAs and viewing the game on separate monitors. This involves creating a serialized communication channel to transmit messages from one FPGA to another, and poses the challenge of keeping track of which of the two identical FPGAs should serve as leader (from the perspective of gameplay, this means deciding which way the ball should go first). We also planned to have a 1-point perspective “3D” view of the game which carries its own difficulties due to the complexity of implementing the trigonometry involved

Introduction

We broke down the pong game into a main module, which keeps track of all the state of the game, two communication modules (input and output), and two graphical rendering systems that allow for 1-point perspective and standard view.

The details of serialized communication are dealt with by the transmission and receiver modules, making it easy for us to easily change the implementation without changing each module's extremely simple interface. This is what will allow us to easily prototype different physical layer communication channels.

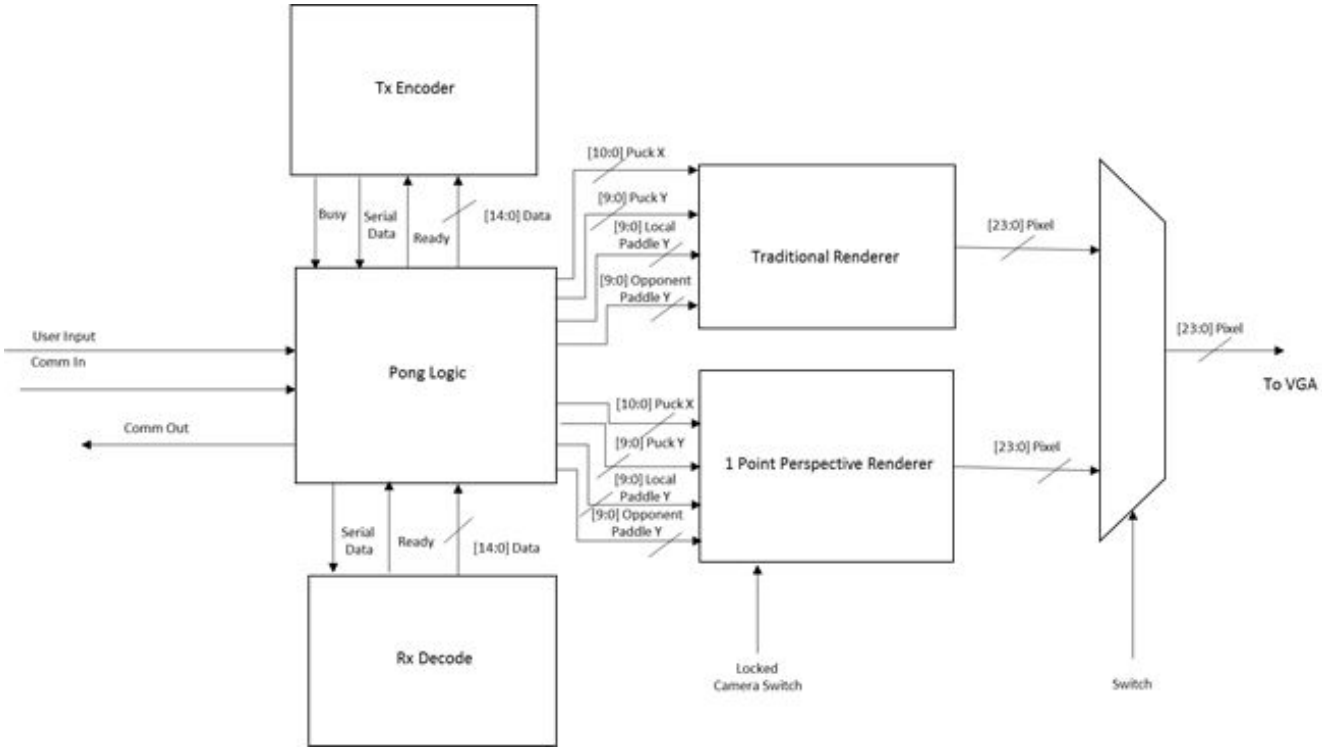
The bulk of the work is really handled in the main pong module so that the visualization modules can be stateless (module pipelining which may be necessary for the 1-point perspective view).

The 1-point perspective view created several challenges of its own: mostly, how to implement the trigonometry, and how to implement (or avoid implementing) division.

We did not achieve our goal of implementing a 1-point perspective view. We did create a viable but slightly buggy two-version of the game. We also implemented a reliable wired communication protocol and an insufficiently reliable infra-red version.

Overview

Our main modules were the Pong Logic, the Tx Encoder, the Rx Decoder, the Traditional Renderer, and the 1 Point Perspective Renderer. These modules can be seen within the following block diagram.



The Pong Logic was our top level module that controlled most of the aspects of the game as well as giving information about the location of the puck and paddles to the encoder and renderers. At the start of the game, the puck starts in the middle of the screen with both the local user and remote user paddles in the default position on left and right hand walls, respectively, centered halfway down the screen. Whichever player is player 1 will start receiving the puck, and the game progresses as expected in default pong. The ball bounces off of the top and bottom walls and the paddles, and if a player fails to hit the ball before it hits their respective wall, the game stops and that player loses.

To handle the processing of the location of items on the screen, we made it such that the local user's FPGA only processed the location of the puck when the puck was arriving towards the local user's paddle. When the puck was travelling away from the local user's paddle, it received data from the remote FPGA as to where to update the puck's location. At all times, the local FPGA was sending data about the location of the local paddle's position and receiving data as the location of the remote user's paddle.

What makes our game more interesting is the ability to switch the perspective that you view the game. By flipping a switch on the FPGA, you would change your video output from a top down perspective to a 1 point perspective, viewing the game from a point just above and behind your paddle. This creates a “pseudo 3D” viewpoint that you can enjoy while your opponent has as much freedom to choose top-down or perspective mode as they please.

Implementation

With regards to our design choices, we chose to use the following items towards completing our project:

- Two Virtex-2 FPGAs (6.111 lab FPGA)
- Two IR LED transmitters
- Two IR receiver chip
- 1k ohm resistors
- 47 ohm resistors
- 3.3 uf capacitors
- 2N2222 BJT transistors
- 0.1 uf capacitor
- Ultrasonic transmitter and receiver

As we did not aim to use any additional memory storage, the Virtex-2 lab kits were sufficient for the use of our project. The remainder of the items listed were used in constructing the transmission and reception modules.

William worked mainly with the transmission and reception modules, and it was his implementation of the traditional renderer from Lab 3 that was tweaked for the purpose of implementing the project. Emmanuel worked on the creation of the 1 Point Perspective module.

Module: Pong Logic

We shall spare the details of this particular module, as it is the same working as in lab 3. A few points of optimization are worth discussing.

It is instructive to consider that the actual pong logic -- distinct from any module actually programmed in Verilog -- is the location of each of these objects, in order that they might be displayed to the screen.

It also must maintain a bit of state that will allow it to explicitly understand the idea that one of the two FPGAs is the master at any given time, and implement the idea that this bit flips on a bounce. Initially, the value of the bit is set by a switch on the labkit -- the differing position of that switch is what differentiates the two game stations at the start of the game. Logic is also there to, in accordance with that state, accept and reject incoming “updates,” namely positions of the puck and the remote paddle, and the puck’s speed.

Module: Pong Transmission

The pong transmission module is an intermediate between the low-level transmission module and the main pong module. It is given access through its inputs to the various state information (x, y coordinates and puck speed) that need to be passed to the remote FPGA. Once each screen refresh, it is activated so that it begins to transmit all the location information about each of the items on the screen that it has a hand in computing.

Module: Transmission

Two transmitters were prepared. Each simply serialized data and sent it over the channel. Different encoding schemes were used for each.

The first, which was ultimately used in our pong game demonstration, relied on a fast and reliable wired connection. Given the extreme speed with which wires transmit information (the delay due to capacitance is apparently not significant), this transmitter was able to operate on the timescale of the 27 megahertz clock cycle. The principle convention is that a single bit is represented by asserting its value on the line for 9 clock cycles. In order to attract the attention of the receiver before sending a message, the transmitter asserts 1 for 3 clock cycles, then asserts 0 for 3 clock cycles, and then asserts 0 for 3 more clock cycles. Immediately after this preamble, data bits are simply sent in sequence.

It is the convention of both versions of the module that a number of bits fixed at compile time (15 for pong) are transmitted as a “packet.” The values of those bits are set on an input port to the module. At the time that a “ready” bit is asserted, the values

present at that time are saved to registers for reference throughout the rest of the transmission process.

The second transmitter differs fundamentally due to the employment of a different channel, but incidentally due to the different encoding scheme. The serialized data is ultimately broadcast by an infra-red LED. The principle convention of the channel is that a 40 kilohertz wave sent to the LED represents the notion of asserting a 1 on the line, while sending the LED a constant 0 would represent the notion of asserting a 0.

So much for the details of the channel. The principle convention of the encoding is that a bit is transmitted over an 1,800 microsecond window: 600 microseconds asserting 1,600 asserting the value of the bit, and 600 asserting 0. Each bit of the packet is sent in sequence in this manner. Note that each bit contains exactly one rise and one fall: a box!

Ignoring the 40 kilohertz wave for a moment -- since in practice that modulation will actually be dealt with external to the module itself -- the transmitter can rely on a (slow) clock with a period of 600 microseconds in principle. (We used a clock with period 300 microseconds to offer more flexibility).

Observe that while the principle convention does not specify the start of a packet explicitly, a theoretical perfect connection will not be troubled by this, since a receiver can know that the first bit sent is the first bit of the first packet, and the n+1th is the first bit of the second packet, where n is the number of bits per packet.

But in practice, a perfect connection is never available. Note in particular that a bit may be completely lost if, e.g., the LED is blocked temporarily, leading to a catastrophic frameshift!

To deal with this practical problem that does not exist in theory, we simply require that the transmitter take a rest for at least 1,800 microseconds after each packet. This allows the receiver to deduce that this should be a packet border.

It would have simplified the design to use the same style of encoding for each of the two transmitters, but it happened that we had designed the wired first, and realized only after that its conventions would not work well over IR. It is worth recounting why. And I shall not merely belabor the point that it needs to be slowed down to even have a

prayer of being successfully modulated onto a 40 kilohertz wave. There is a more subtle problem at work here.

Observe that in the wired connection convention, when a message consists only of 1s, the value 1 will be asserted throughout almost the entire packet: everything save for a third of a length of a bit in the preamble. We observed that this is very bad news for the IR communication equipment! The IR receiver seems to experience a sort of sensory fatigue after a long time receiving a signal that is intended to assert a logical 1. The exact explanation alluded us (perhaps because the computer science student without any understanding of analog circuitry was in charge of this part of the project!), but we came to the general conclusion that a signal that changes value frequently is the kind that the IR channel happens to prefer.

Module: Reception

The receivers, in turn, decode the information they receive over the serialized connection, so that the encoding schema is the same as for the transmitter. After a full fixed-width packet of information is received, each receiver works the same: it asserts the value of the packet onto the output port, and asserts for a time a 1 on a special port that indicates that a new packet has arrived.

The wired receiver has two main states. The first would be the “wait” state, during which a shift register is employed to record successive values on the line, and logic checks each time whether or not there is an (approximate) match with the preamble sequence: such a match is an indication to move to the receive state. Since a bit is transmitted in 9 clock cycles, we simply cycle through the different bit indices, allowing 9 clock cycles each. On the fifth clock cycle of each bit, we record that value on the line into a buffer at the appropriate index in order to populate the data in the packet.

From our subjective vantage point, the receiver for the IR communication system is more elegant, since it maintains less state. Crucially, it keeps track of which bit of the data stream it is currently receiving -- this value is naturally set to 0 upon reset! It is easy to increment this value each time the bit is received, and to consider the packet complete when a sufficient number of bits have been received.

The way of receiving a particular bit is to simply record the value on the line at 900 microseconds after the rising edge of that particular bit’s block wave.

To take advantage of the frameshift-combatting idle period after each packet, a separate simple module is employed. Running on a 300 microsecond clock, it asserts 1 whenever the last several values on the line have all been 0. The asserted 1 causes the receiver to reset itself.

It is worth observing that this particular way of taking advantage of the idle period combats not only the idle-period problem, but also the problem of lost bits. A crude (and ultimately insufficient!) error correction to be sure, if a bit is entirely missing, the packet of which it was a part is completely discarded since a requisite number of bits will not have been received!

We were unable to use the IR system effectively. A test arrangement showed that bit errors were unreasonably frequent in a one-way communication situation. If we had put more time, we probably could have used an error correction protocol or additional over-sampling to fix the problem to a large degree: though not completely, as we really noticed that line-of-site was blocked fairly easily by e.g. fingers that were even making a conscious effort to keep out of the way!

A more interesting and different problem arose when a two-way communication is attempted. It seems as though the IR receivers prefer to detect the stream from the local transmitter rather than the remote one!

It appears fairly likely that this has to do with electrical interference across the FPGA board itself, since we could not eliminate the signal by attempting to block the infrared emissions with our hand, our cell phones, or other implements. It is unclear what exact cause brought us the frustrating phenomenon. Additional capacitors across the power nodes of the IR receiver did not solve the problem. One guess is a fluctuating ground, but by what mechanism that could cause the problem we observed is not obvious.

Module: Traditional Renderer

The traditional renderer betrays the simplicity of pong's classic 2D table. All that was necessary was to draw two paddles (rectangles) and a square puck.

Locations for paddles are specified with y-coordinates only, since their locations along the x-axis is fixed. Each item's location is specified with regard to its center, and the sizes of the various arguments are passed to the module as compile-time parameters.

Module: 1 Point Perspective Renderer

The 1 Point Perspective Renderer was the main challenge with regards to graphics. To accomplish this, the perspective renderer was split into multiple submodules:

- onePtPerspective - the top level module that took in the 2D puck coordinates, the local paddle coordinates, the remote paddle coordinates, hcount, vcount, and a clock. It would display the pixel that would be painted on the screen at coordinate (hcount, vcount). It tied together inputs and outputs for all of the other submodules as well as dealt with the timing for when to shift output calculations into the inputs of necessary modules
- cosine - took in a value for the phase that was left shifted by 5 bits and outputted the corresponding value of cosine, also left shifted by 5 bits
- sine - similar to the cosine module except that it outputted the corresponding value of sine left shifted by 5 bits
- sixPoints - took in values for the x and y coordinates for the center of an object as well as values for the depth and width of the object in the 2D coordinates and outputted 6 xy-coordinates that would be transformed and create a 3D representation in a subsequent module
- mappingModule - took in an xy-coordinate from the 2D coordinate system as well as the cosine and sine of the x,y, and z orientations of the camera and ouputted an xy-coordinate that represented the transformation location of the point relative to the camera in a perspective viewpoint
- blob3D - a more complex version of the blob module from Lab 3 that took in 6 transformed xy-coordinates (outputs of mappingModule) that represent the coordinates of the rectangular prism (either a puck or a paddle) that needed to be displayed on the screen as well as hcount and vcount; it displayed a colored pixel if hcount and vcount represented a point within the boundaries of the rectangular prism and a black pixel otherwise. The outputted pixel was sent to onePtPerspective to be outputted to the monitor
- insideTriangle - a module that took in 3 xy-coordinates as well as hcount and vcount and outputted a boolean that was true if the coordinate (hcount, vcount) was within the triangle and false if the coordinate was not inside of the triangle; was used within blob3D for determining which pixels to paint with or without color - issues with this module will be discussed later
- insideTrapezoid - a module that took in 4 xy-coordinates as well as hcount and vcount and outputted a boolean that was true if the coordinate (hcount, vcount)

was within the trapezoid and false otherwise; was used within blob3D for determining which pixels to paint with or without color

Part of the challenge came from the algorithm for mapping the coordinates from 2D pong to a 3D image that would then be mapped to 2D pixel locations on the screen. To accomplish the 3D perspective view, the following algorithm was used to do the mapping:

$$\begin{bmatrix} \mathbf{d}_x \\ \mathbf{d}_y \\ \mathbf{d}_z \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta_x) & \sin(\theta_x) \\ 0 & -\sin(\theta_x) & \cos(\theta_x) \end{bmatrix} \begin{bmatrix} \cos(\theta_y) & 0 & -\sin(\theta_y) \\ 0 & 1 & 0 \\ \sin(\theta_y) & 0 & \cos(\theta_y) \end{bmatrix} \begin{bmatrix} \cos(\theta_z) & \sin(\theta_z) & 0 \\ -\sin(\theta_z) & \cos(\theta_z) & 0 \\ 0 & 0 & 1 \end{bmatrix} \left(\begin{bmatrix} \mathbf{a}_x \\ \mathbf{a}_y \\ \mathbf{a}_z \end{bmatrix} - \begin{bmatrix} \mathbf{c}_x \\ \mathbf{c}_y \\ \mathbf{c}_z \end{bmatrix} \right)$$

$$\begin{aligned} \mathbf{b}_x &= \frac{\mathbf{e}_z}{\mathbf{d}_z}\mathbf{d}_x - \mathbf{e}_x \\ \mathbf{b}_y &= \frac{\mathbf{e}_z}{\mathbf{d}_z}\mathbf{d}_y - \mathbf{e}_y \end{aligned}.$$

Where:

- a_x,y,z is the location of the object being projected
- c_x,y,z is the location of the camera viewing the object
- θ_x,y,z is the orientation of the camera viewing the object
- e_x,y,z is the location of the viewer relative to the display surface
- b_x,y is the projection of the image on the monitor

There were a number of things that made this module difficult to handle, especially with regards to this algorithm. The first difficulty came from having to deal with the division of e_z/d_z in the final step for mapping the coordinate to its 1-Point-Perspective location. Because division is not easily implementable within verilog, we had to find a way to create a module that could circumvent this. After researching different things such as CORDIC algorithms that likely would have taken longer than our six-week project scope to implement, we stumbled across the divider IP Core provided within the Xilinx package. The divider generator uses a Radix-2 non restoring algorithm that solves one bit of the quotient during each cycle through addition and subtraction. So the larger the bit width of the quotient, the more clock cycles it would take to finish the entirety of the division. This meant that in order to implement the division for the algorithm, the module would have to wait quite a few clock cycles before outputting the new coordinate for any point, so it became necessary to pause calculations to wait for the ready signal from the divider.

What was also interesting was dealing with the output of the divider module. The calculations that were necessary required the use of numbers smaller than 1, and the divider module provides both an integer output as well as your choice of either an integer remainder or a binary fractional output. Binary fractions are represented similarly to binary integers, where the leftmost bit is the largest power of 2 and the rightmost bit is the smallest power of two, but the value of the fraction is determined from left to right, given that it will always start with the leftmost bit at 2^-1 and the rightmost bit is 2^-n where n is the number of bits wide the fractional value is. In essence, fractional binary 100 is equivalent to 1000000, which is ½ whereas fractional binary 001 (⅛) is not equivalent to 0000001 (1/128). However, when doing addition, subtraction, and multiplication, you can simply concatenate the integer and fractional components (provided that you extend the fractional components to the same number of bits) and perform the same operations. To properly read the output, you would count the number of bits that composed the fractional components of each number and sever that many bits from the right side of the output number to determine the fractional output. The remaining bits to the left of this severance point were the integer component of the answer.

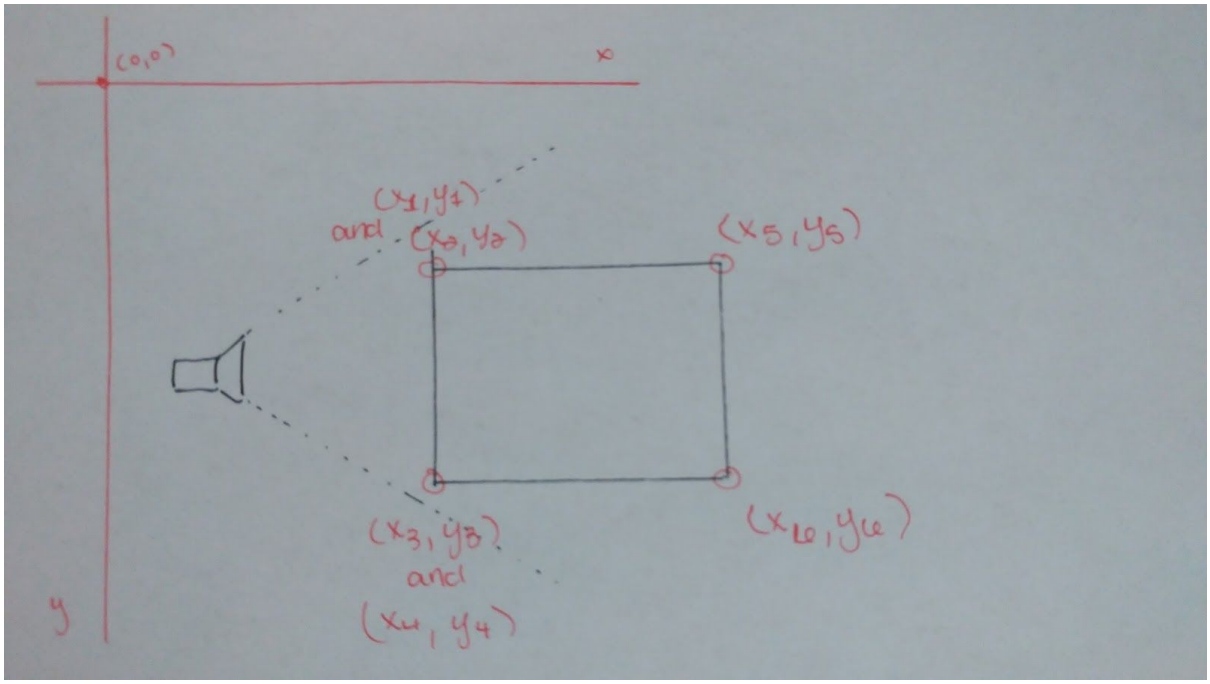
```
//To multiply 101.75 by 4.25
wire [6:0] x_integer;
wire [3:0] x_fractional;
wire [6:0] y_integer;
wire [3:0] y_fractional;
wire [21:0] result;

wire [3:0] result_fractional;
wire [17:0] result_integer;

assign x_integer = 101; // 7'b1100101
assign x_fractional = 4'b1100; // (1/2) + (1/4) = .75
assign y_integer = 4;
assign y_fractional = 4'b0100;

assign result = {x_integer, x_fractional} * {y_integer, y_fractional};
assign result_integer = result [21:4];18'b000000000110110000
assign result_fractional = result[3:0]; // 4'b0110 = .4375 fractional
```

An example of how to multiply while using fractional and integer binary numbers

The other tricky part was dealing with the cosine and sine calculations. First, rather than finding algorithms for exactly computing the values, we created a simplified calculation for cosine and sine that roughly approximated the actual values. Using input phases that were between -pi/2 and pi/2, the value of cosine was evaluated at 1-x where phase x was in the range pi/2 > x > 0 and 1 + x for phase -pi/2 < x < 0. For sine, if the phase was in the range -0.6 < x < 0.6 (all values in radians), then the output was evaluated to be x; if the phase was in the range 0.6 < x < pi/2, the output was 0.825*x

+.105, a close approximation of the sine curve for the region where phase is greater than 0.6; if the phase was in the range -pi/2 < x < -0.6, then the output was 0.825*x - .105. However, since Verilog does not allow for floating point values, we took the ceiling of these values multiplied by 32 as well. Hence, the output was also scaled by a factor of 32 when it was inputted to the mapping module for transformation. Within the mapping module, the calculations were completed and then right shifted by 5 bits. By our estimates this would provide enough precision to create the perspective mapping without having to deal with absurdly large bit-widths during the matrix multiplications.

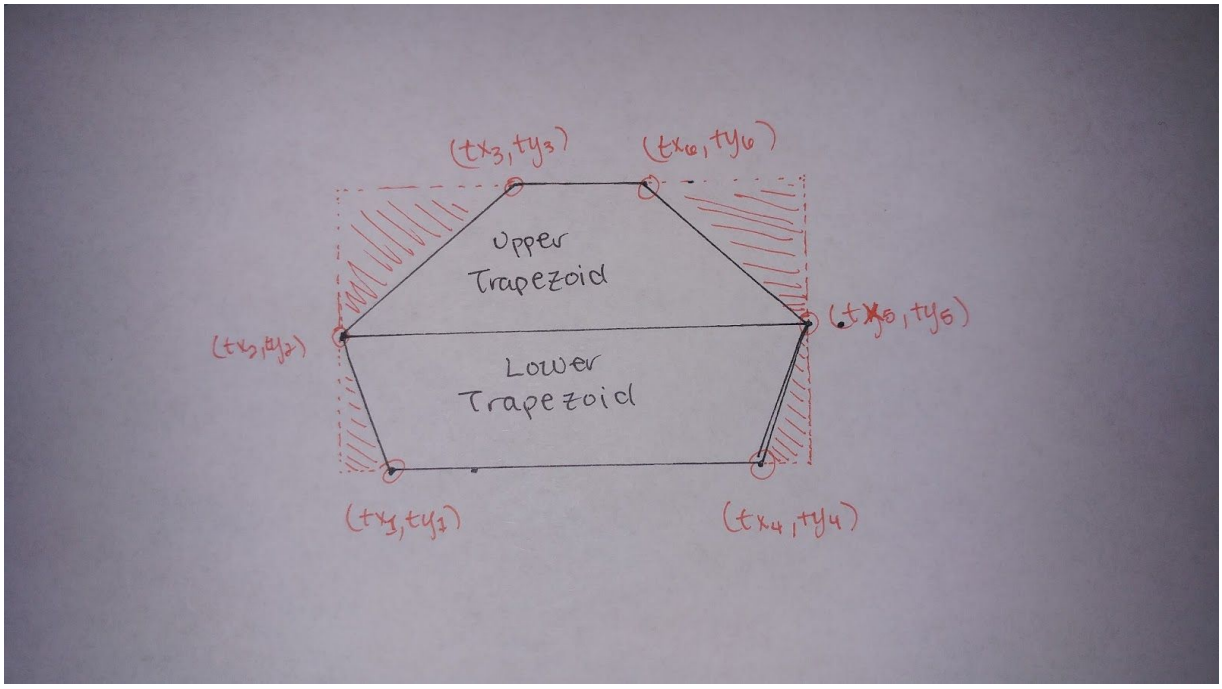
One other consideration that had to be made was how to paint the objects on the screen given that they were not simple rectangles. In Lab 3, we were able to contain each paddle and the puck's boundaries within a "blob" module that would paint a colored output pixel if the coordinate (hcount, vcount) was within the range (x < hcount < x +width, y < vcount < y + height). However, because we were trying to create perspective skewed rectangular prism, we had to adjust our blob module to provide a different edge detection.



2D Mapping of an object. The camera is shown at the left, and the six points to be transformed are also shown.

Originally, we had blob3D check if hcount fell within the extreme left x value of the transformed prism (tx_2 in the figure) and the extreme right x value (tx_5 in the figure) and if vcount fell within the upper and lower y values of the trapezoid (ty3 and ty1,

respectively). If it did fall within these values, then we would check whether the values fell within the shaded triangles within the figure below. If the point was not within the shaded triangle zone, then the point fell within the actual object we intended to paint. This detection was accomplished within the insideTriangle module. We researched algorithms that were able to determine whether a point was inside a triangle and found one that did not require division.



The transformation of the 2D object to a 1-Point perspective view. The shaded regions are the triangles we attempted to exclude using insideTriangle. The Upper and Lower Trapezoids are the trapezoids used within the insideTrapezoide module. Tx_1…..Tx_6 and Ty_1….Ty_6 are the transformed coordinates of x_1…..x_6 and y_1….y_6 of the previous figure

However, this algorithm had two critical flaws. The first was that edge detection was not guaranteed to work, and this tended to leave graphical artifacts during testing. The other flaw was that changing the ordering of the inputted points would cause the algorithm to fail, a problem we discovered in Verilog code that was confirmed by further testing within Python.

To circumvent this issue, we switched to an algorithm that was far simpler and more reliable. We viewed our 2D viewpoint of the rectangular prism as two trapezoids stacked on top of each other. Then, we selected the the left and right lines of each trapezoid(for example, in the top trapezoid the lines are (tx_2, ty_2) → (tx_3, ty_3) and (tx_5, ty_5) → (tx_6, ty_6), respectively) and represented them in the form (x2 - x1)*y + (y1 - y2) *x = b. This equation comes from the following steps:

$$y = m * x + b; m = \frac{y_2 - y_1}{x_2 - x_1}$$

$$y * (x_2 - x_1) - x(y_2 - y_1) = b$$

$$y * (x_2 - x_1) + x * (y_1 - y_2) = b$$

If you can guarantee the property y_1 < y_2, then in normal xy-coordinates, the following will hold:

If

$$vcount * (x_2 - x_1) + hcount * (y_1 - y_2) > b = y_2 * (x_2 - x_1) + x_2 * (y_1 - y_2)$$

Then (hcount, vcount) is to the right of the line.

Else if

$$vcount * (x_2 - x_1) + hcount * (y_1 - y_2) < b = y_2 * (x_2 - x_1) + x_2 * (y_1 - y_2)$$

Then (hcount, vcount) is to the left of the line

However, because (0,0) on the screen is in the top left corner of the screen instead of the bottom right (the y value increases going down rather than going up), the signs on the comparators are flipped, which gives the following statements:

If

$$vcount * (x_2 - x_1) + hcount * (y_1 - y_2) > b = y_2 * (x_2 - x_1) + x_2 * (y_1 - y_2)$$

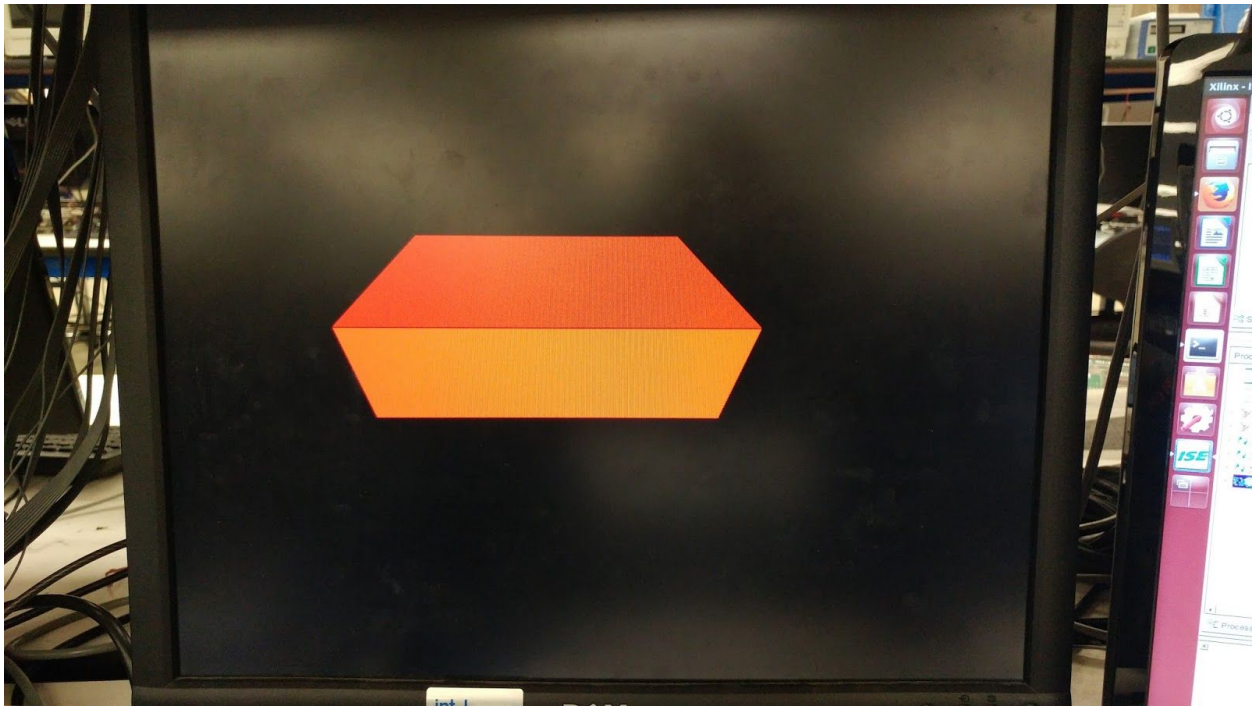
Then (hcount, vcount) is to the left of the line

Else if

$$vcount * (x_2 - x_1) + hcount * (y_1 - y_2) < b = y_2 * (x_2 - x_1) + x_2 * (y_1 - y_2)$$

Then (hcount, vcount) is to the right of the line.

Using these properties, we created the insideTrapezoid module which checks if a point is to the right of the left side line, to the left of the right side line, and within the upper and lower boundary lines of the trapezoid. All that remained in blob3D afterwards was discovering which trapezoid (hcount, vcount) was in and displaying the appropriate pixel color for each trapezoid, as demonstrated here:



Test output of a rectangular prism using the insideTrapezoid modulewithin blob3D

Review

As for our 1-Point perspective output, we were unable to finish this module by the end of the project period. Many of the issues arose from trying to circumvent division for the mapping module as well as timing the various calculations within the onePtPerspective module. Some of the modules that were implemented required multiple clock cycles to finish calculations, and we struggled getting new inputs for calculation modules to shift in at the right time (e.g., not giving new inputs in the middle of calculating the project coordinates for the previous points). For any group that wishes to do this project, a great deal of time regarding timing, ready signals, and “chip enables” (enable inputs for calculation modules) should be planned out, especially for debugging.

Conclusion

Several things stymied our ability to finish the project properly, most of all our lack of technical skill and our skill at estimating the time that tasks would require. As an example of lack of both, William spent approximately 8 hours in the lab trying to get the IR receiver to simply receive signal from the transmitter. He was not aware at the time that a changing signal was required to get interesting output on the oscilloscope, so wasted a lot of time wondering why the system wasn’t working. Meanwhile, the task of

merely establishing the channel was estimated to be a quick one. He had reason to doubt he had assembled the thing correctly anyhow: unfamiliar with analog electronics, he was not confident that he had wired the transistor and other components correctly.

The peculiarities of 6.111, and its logistics relative to our own bad habits presented a surprising level of challenge, including timing constraints. In order to begin work on the IR system, access to the supply closet was required in order to secure parts, but that room was only open during the day. And no lab work at all could be completed during 12 a.m. and 9 a.m. This created a challenge since at least one member of the team had a sleep schedule that had become 180 degrees out of phase with the sun's shining during the week following Thanksgiving -- a week that was intended to be used to implement the meat of the project but was instead spent constantly groggy, and which was slated as the time to secure that equipment. The inclusion of this passage is not in jest, but is a recognition of the fact that a bad sleep schedule has real consequences. Plainly, one of the things that 6.111 tests is our ability to keep a sleep schedule, and that particular assessment was not one at which we fared well: it set the communications side of the project back a full week.

We also learned that communication over IR is a lot harder than we had expected. Perhaps we should have figured as much, since that particular channel is mostly used for things like remote controls, which do not need to maintain connectivity longer than a few seconds, and the user can easily adjust the remote so that it gets a better connection.

Overall, this project exposed us to the complexity of digital hardware and the difficulties in timeline creation for projects. Having had this experience, we are now more prepared for planning out the timeline for creating an FPGA based project.