3D Digital Stereoscope

6.111 Final Project

Joshua Monzon
Tony Ng
Steve Zhou

TA: Javier Castro

Abstract
The goal of this project is to design and implement a 3-D rendering system that will generate two images from different points of view to create a stereoscopic effect, generating a 3-D illusion. The system takes perspective projections of a 3-D virtual environment, and then renders, shades, and outputs the images onto a monitor. The user can move around the 3-D virtual world and look at different objects while both of the stereoscopic images are being processed and updated simultaneously in real time. The project will be implemented on a Xilinx Field Programmable Gate Array (FPGA) and the 3-D effect will be aided by the use of a stereoscope viewer.

Table of Contents

Introduction and Design Overview

Our project is to design a robust and powerful 3D engine that would process a 3D pixel map of a world and convert it to two 2D images which will be displayed on two screens, each viewed by one eye. The purpose of this is to trick the eye into believing that the person is actually in a 3D environment. The reason why this works is because the human left and right eyes see slightly different images. If we can correctly extract these two images and display them to the corresponding eyes then it is possible to trick the brain into believing that it is seeing a 3D object.

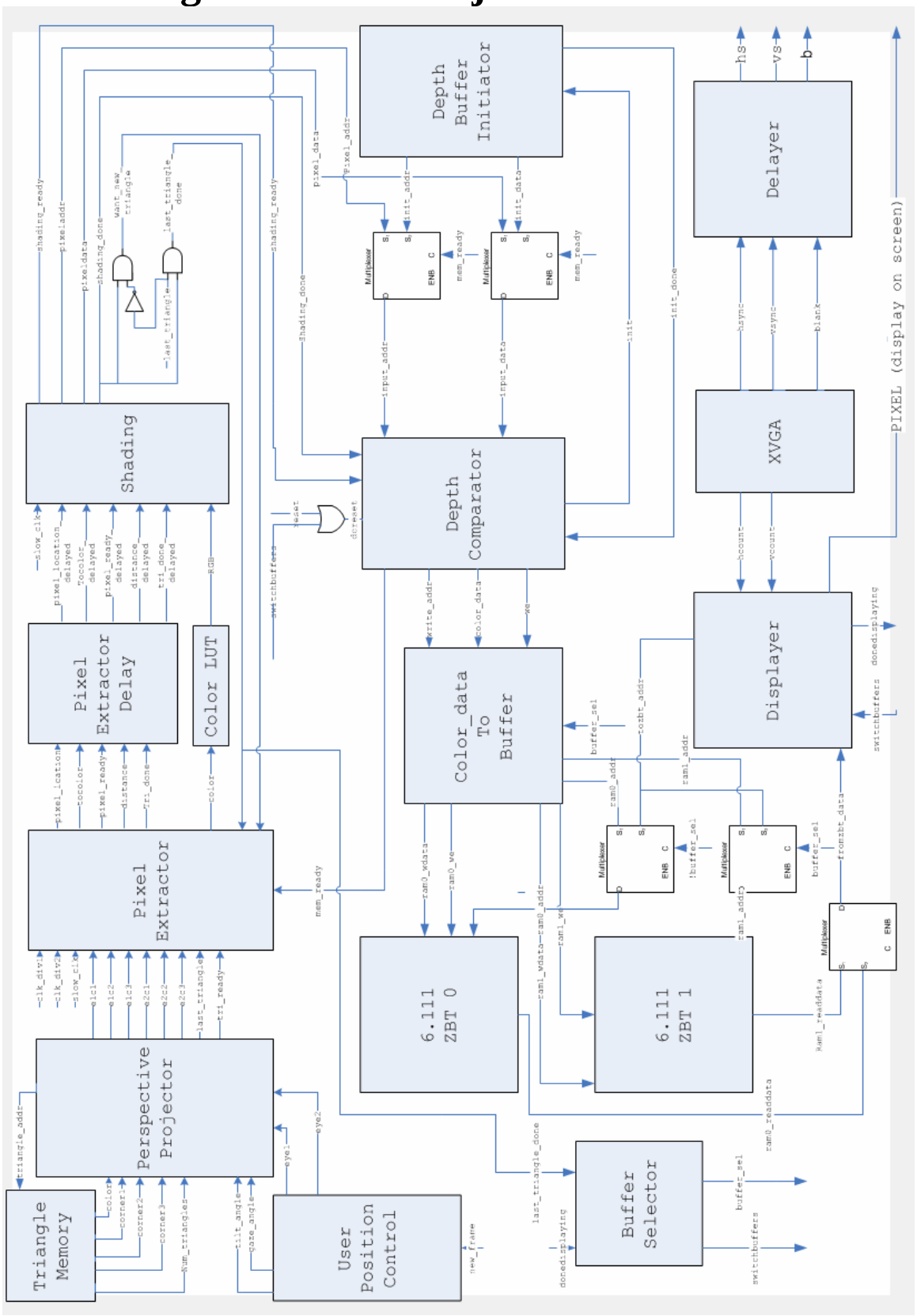
This project is very interesting because it adds another flavor to numerous applications of 3D graphics technology. One such application of this is the virtual manipulation of 2 dimensional objects. For example, imagine a 3D paint program. If one can "hold" a 2D paint brush and virtually paint in the computer screen, then he has more fine control of the brush instead of just using a mouse to paint, for example.

3D imaging is a difficult task because it requires a large amount of computational power and time. Therefore, nowadays, most computers have a specific piece of hardware which is dedicated to rendering and displaying graphics. Our project aims to build a similar type of hardware, although more primitive than what is being used nowadays. We would employ the idea of pipelining and parallel processing to render our images into the screen. Ray casting would be used as a method to project our images. This method is used by classical 3D first person shooter games such as Doom and Wolfenstein. We would also use triangles to create all our images.

To make this complex task less daunting, we divided our 3D engine into 3 stages, each group member in charge of one stage. The first stage involves the processing of user interaction signals which enables the user to move around the world. This stage involves the projection of 3D objects to a 2D plane which corresponds to the field of view of the user. This field of view changes as the user moves and calculating this behavior can be very complicated. The outputs of the first stage are the 3 coordinates of the corners of the triangle and the depth associated with each of these corners. The second stage involves the shading of all the triangles that are projected in the screen. The challenge involved in this stage is being able to extract the pixels which form the triangle given 3 coordinate locations, then somehow use only the depth information of the three corners to extrapolate the depth of *all* the pixels. After this, it needs to use a shading algorithm which in turn uses the depth of the pixels to shade and render the images. The last stage comprises the memory management of these pixels. The complexity lies in the fact that this stage has to access the memory to check if an incoming pixel is located very deep in the world and is being overlapped by previous pixel. Depending on the result, it decides whether to write the incoming data. Dual buffering is employed to make the display smoother. Designing an effective system which controls the writing and displaying of these two buffers is a very tricky task.

To fully coordinate our stages, we used a precise and elaborate feedback communication system very much like the major and minor FSM concept.

Block Diagram of our Project



1. 3D to 2D Projection and User Input Control

1.1. Triangle Memories

The triangle memories module is responsible for holding all the information necessary to construct the three dimensional world. The only thing that needs to be stored are triangles, since everything in the three dimensional world is constructed out of triangles. It is inefficient to store all the pixels of the individual triangles, so only an 8-bit color of a triangle and the coordinates of the three corners of the triangle are stored. This contains enough information to reconstruct a triangle perfectly without wasting space on storing extra data. Each corner of a triangle is simply a point in space, so an x, y, and z coordinate needs to be stored. Each x, y, and z coordinate is 12 bits large, so a corner requires 36 bits. So in total, each triangle takes up 116 bits of memory.

The triangle memories are constructed out of BRAMs on the FPGA. Xilinx allows data to be loaded into these BRAMs upon startup, so it is easy to change the contents of the world by loading a different ".coe" file containing different triangle data. The triangle memories module has a signal with the number of triangles stored in it coming out. This signal is used for optimization, because once that number of triangles have been processed, there is no need to look for more triangles because the rest of the memory is empty.

In implementation, the triangle memories made of four pieces of single port block memory. One of the memories contains the color of the triangles. Each of the other three contains the x, y, and z coordinates of one corner. The address inputs of these four blocks of RAM are all wired to the same address input, so each address input will output a specific triangle.

1.2. Perspective Projector

The perspective projector is responsible for mapping triangles in three dimensional space onto a two dimensional plane. In the digital stereoscope, the two dimensional plane is the pixels on the monitor. To understand how perspective projection works, let a point in three dimensional space represent the location of an eye. Now place a plane directly in front of the eye in the direction that the eye is looking towards. For each point in every object that is in the three dimensional space, draw a line from that point to the eye. Where the line intersects the plane is where that point would be projected onto. If the plane were represented by a monitor, the point that the line intersects the plane is the pixel that the object should project onto. If the plane is selected such that it represents the eye's field of vision, then lines that do not intersect the plane cannot be seen

Projecting every point of every object in the three dimensional space would be an enormous computational task. It would be impossible to generate images in real time. However, since all objects are constructed out of triangles, it is possible to only project

the three corners of each triangle. The rest of the triangle can be drawn on the two dimensional plane once the projection of the three corners is known.

Even when working with triangles, it is still not easy to draw a line and figure out where it would intersect the plane in front of the eye. There is quite a bit of math involved and it is difficult to do quickly in hardware. Also, considering that there are two eyes at different locations, everything must be processed twice, meaning fast algorithms must be used in order to make movement through the world happen in real time. For further simplification, clipping is ignored. If any corner of a triangle is off the screen, the entire triangle is removed from the screen. This avoids difficult and lengthy calculations required to handle these cases. Many simplifications must be made in order for graphics processing to be done in real time on the FPGA.

The algorithm used in this perspective projector is a ray casting algorithm. A vector is created from an eye to a corner in a triangle. To find out which column of the screen this particular corner belongs on, find the angle between the gaze angle (the direction the eye is looking towards) and the vector on the xy-plane (or the ground). Arctangent can be used to find this angle. If the angle between the gaze and the vector is zero, then the eye is gazing towards the corner, so the corner belongs in the middle column of the screen. By deciding how many radians or degrees are represented by each pixel (in this implementation, each pixel represents 1/160 radians), it is possible to map points onto columns of pixels on the screen. Note that since the height, or z coordinates, is ignored in this calculation, it is impossible to know which row of the screen the point projects onto. In order to figure out the row that the projected point belongs on, a similar calculation can be repeated. By taking distance from the eye to the point on the xy-plane and the difference in height of the eye and the point, arctangent can be used to find this angle, which when compared to the tilt angle in a similar fashion, will determine which row the point should project onto. This algorithm does not exactly project onto a flat plane, because as angles are increased linearly, it sweeps through a round area, not a plane. However, as long as the field of vision is not too large, the algorithm produces a good estimate of where the points belong on a two dimensional plane in front of the eye.

The perspective projector takes as inputs: `clk`, `slow_clk`, `corner1`, `corner2`, `corner3`, `inColor`, `eye1`, `eye2`, `gaze_angle`, `tilt_angle`, `start`, `num_triangles`, and `reset_last_triangle`. The three corners, the two eyes, gaze angle, and tilt angle are used in the projection calculations, as described above. The normal clock runs fast, which is required for the monitor to function correctly. However, the calculations of arctangent and square root do not always work correctly at speeds this fast, so a slower clock is used to do these calculations. The start signal tells the perspective projector to project the next triangle in memory. The projector also receives the total number of triangles stored in memory, which allows the projector to know that it cannot find more triangles to project after it has completed that many. The reset last triangle signal tells the perspective projector to start projecting all the triangles from the beginning all over again.

As outputs, the perspective projector sends: `outColor`, `triangle_addr`, `e1c1`, `e1c2`, `e1c3`, `e2c1`, `e2c2`, `e2c3`, `done`, and `last_triangle`. The projector sends a triangle

address to the memory to obtain triangle data to perform calculations on. Six points are outputted, three containing information on the projection from the first eye and three containing data from the second eye. This information includes the two dimensional x and y coordinates of the project and the depth of the point. Depth is important in determining what objects should appear in front and what objects should appear in back when multiple objects project to the same area on the screen. The done signal informs other modules that the calculations are done and the projector is ready to do more calculations. The last triangle signal tells other modules that all the triangles in memory have been projected.

The perspective projector contains three small sub-modules. The arctangent estimator sub-module simply takes in the x and y coordinates of two points and then it does the logic necessary to convert the data into a form that the CORDIC algorithm (packaged in Xilinx) can calculate and return the arctangent. The distance finder sub-module takes in two points and applies the distance formula to it, using CORDIC for square root. The angle to screen converter sub-module does some simple math necessary to determine which angles corresponds to which pixels, which is used by the ray casting algorithm described earlier.

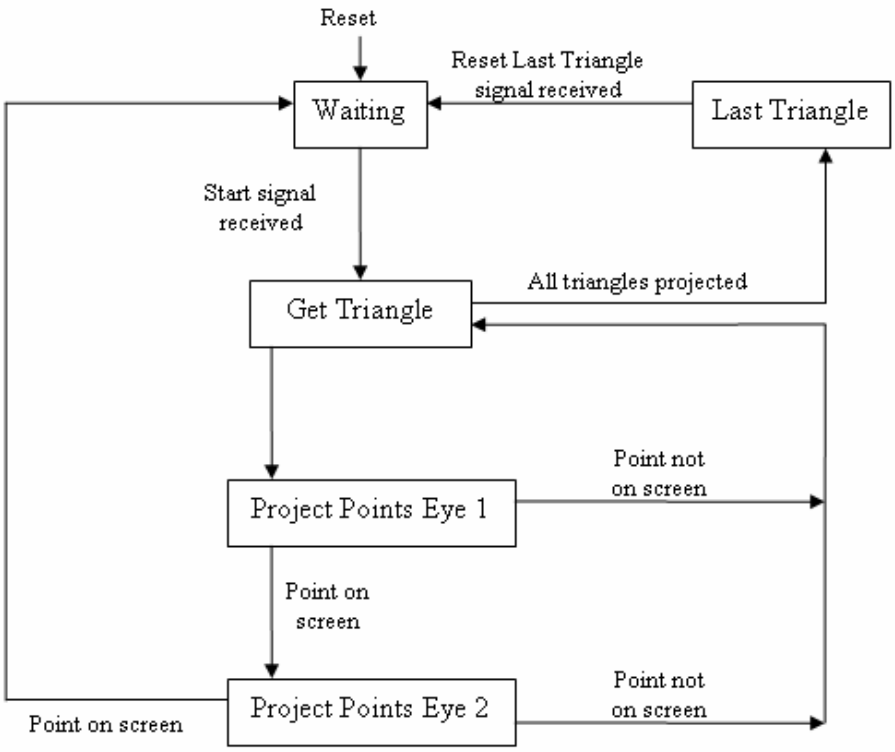


Figure 1.2.1. FSM Diagram for Perspective Projector. Controls the functionality of the Perspective Projector

The perspective projector operates as a finite state machine as shown in Figure 1.2.1. Initially, it starts in the waiting state. During this state, when it receives a start signal, the perspective projector requests a triangle from memory and then attempts to project that triangle to the screen from the point of view of the first eye. If the corners are not on

screen, the triangle is thrown out and the perspective projector tries to get a new triangle. If the triangle projects onto the screen, then the projector tries to project the triangle with the second eye. If the triangle is off the screen, then the triangle is thrown out and the projector tries to obtain a new triangle. If the point is on the screen, the perspective projector signals that it is done calculating and that the output data contains valid triangles on the screen and enters the waiting state to allow other modules to use the data. If the perspective projector tries to get a triangle but it has already projected all the triangles in memory, then it enters a last triangle state. In this state, the projector signals other modules that the last triangle has been processed and then enters the wait state. This signal is important because when all the other modules finish using the data, then a frame has just been completely processed.

Testing the perspective projector is difficult because it is a very large module and some things are not very obvious through just observing the outputs, such as whether the projection looks realistic. The signals can be tested through a test bench and by the logic analyzer to make sure they operate correctly. The best way to test whether the projects look nice is to set up the projector so it actually outputs to a monitor and then move around the world to make sure that the three corners of the triangle project correctly.

1.3. Position Controller

The user position controller is responsible for controlling how the user moves around the world. It takes as input the keyboard data and keyboard clock pins from the lab kit. Upon reset, the position controller assigns the user an initial starting position, gaze, and tilt angles. Now in response to the user's inputs, the position controller generates new coordinates for the location of the eyes and the new gaze and tilt angles.

The user's position can be represented by one point. However, one of the eyes should be to the left of this point and the other eye should be to the right. When the user decides to walk around the world, both eyes should follow the movement. Also, when the user turns, the eyes should rotate while keeping the distance between the two eyes the same. This behavior of the eyes is created by constantly updating the location of the eyes based on a small lookup table that calculates sine and cosine values of the gaze angle. At every clock cycle, the two eyes are moved next to the user's position, a point, and placed perpendicular to the gaze angle.

When a user wants to move forward, he probably wants to move towards what he is starting at, or towards the gaze angle. Sine and cosine values are needed to move at specific angles, but fortunately the eyes use sine and cosine of the gaze angle too, so this can be reused. Also, the sine and cosine table does not have to be too accurate as the screen is made of pixels and the world is divided into integer coordinates, so there has to be quite a bit of estimation involved anyways.

One of the sub-modules of the position controller is the keyboard controller. The keyboard controller maps ASCII values from the keyboard into moments. Also, the keyboard controller is responsible for decided when to allow a move. The user should not

be allowed to move unless a frame has finished processing. Otherwise, while some objects are being projected onto the screen, the eyes can move and half one an object may be projected somewhere else. They keyboard controller samples the latest input since the last new frame and uses that as the user's input. Finally, the keyboard controller maps what keys do what actions. Copying the usual keys used by most first person shooter video games, the user controls are summarized in the following table:

User Actions and Associated Keystrokes

Action	**Keyboard input**
Move Forward	W
Move Backward	S
Move Left	A
Move Right	D
Turn Left	Q
Turn Right	E
Look Up	R
Look Down	F

There are two more sub-modules in the position controller, provided courtesy of Professor Terman and Professor Chuang. The first of these is the PS/2 ASCII input module. This module maps all the keyboard inputs into ASCII values. Within this module is the second sub-module, the PS/2 driver. The PS/2 driver reads a stream of one bit data from the PS/2 port and combines the bits into data that can be used by other modules.

Testing the user position controller is difficult to do in simulations because it involves physical keyboard inputs. So, to test all the modules, the lab kit was programmed to display the two eyes on the screen. The screen serves as the x and y coordinates, or the ground, of the world. So when the user moves around, it is easy to see how the eyes move and it is easy to check whether the user is moving in the right direction. Other information such as gaze and tilt angles can be read using the logic analyzer or simply displayed on LEDs on the lab kit.

2. Projector Shader Module

The projector shader module acts as a "black box" that encapsulates three major submodules. The purpose of this module is to wire up these three different submodules correctly and, latch the output data from the shading module, and correctly determine the value for `want_new_triangle` and `last_triangle_shading_done` control signals. The purpose of this module was to process the corner locations of a triangle for both eyes, and output a sequence of pixel addresses and its corresponding data to the ZBT so it can be written to the buffers. In addition, `shading_ready` signal must be high for 3 clock cycles before the first valid set of pixel address and data, and `shading_done` must be high for 3 clock cycles after the last set of pixels for a triangle. The exact details of how this was made to work will be discussed in the pixel extractor, pixel delay, color LUT, and shading modules, and a couple of additional output signals are discussed below.

`Want_new_triangle` signal is passed to the perspective projector module, and acts as a control signal to inform that module to begin its computations for a new triangle. This signal is only high when `shading_ready` is high from the shading module, which means that the rendering of the current triangle has been completed, and the `last_triangle` signal is low. If the `last_triangle` is high, it means that our system has just completed rendering the last triangle, and the buffer needs to be switched and cleared. Thus, we do not need to wait until the memory of the new buffer is ready to be written to before asking the perspective projector for a new triangle.

`Last_triangle_shading_done` is used to inform the buffer interface module that the shading of the last triangle has been finished. This means that all of our triangles in our world have been processed, and their information stored in one the ZBTs, and a switch of the ZBTs is required. This signal is held high for three clock cycles at which point `shading_done` is high and `last_triangle` is high.

This module was tested first using a testbench followed by probing with a logic analyzer. We set the input to our system up as a FSM, which switches between two states, each state outputting a set of triangle inputs along with their control signals that would have otherwise been provided to me by the perspective projector module. Using this setup, we first made sure that the after a `shading_ready` signal is high, data for pixel address and pixel data begins to flow, changing once every three clock cycles. Second, we made sure that the `shading_done` signal was high for three clock cycles after the set of pixel address and data was outputted. Third, we checked to see that the `want_new_triangle` signal is set high after the first triangle has been processed, which in turn switched the state of our FSM to output the data for the second triangle. Next, we checked to make sure after all of the pixels of the second triangle has been looped through, and a shading ready signal is high, the `want_new_triangle` signal is set to low, and the signal `last_triangle_shading_done` is set to high. Lastly, we made sure that all the pixel addresses for both eyes of both triangles were looped through by the pixel extractor, and each of those addresses corresponded to the correct pixel data, which includes the pixel's distance and color.

2.1. Pixel Extractor Module

The pixel extractor module takes as input `mem_ready` signal from buffer interface, `tri_ready` and `last_triangle` signals from perspective projector, as well as the corner locations of the triangle for each eye. Using this information, it needs to control the outputs of `pixel_ready`, `tri_done`, `tocolor`, `pixel_location`, and `distance` signals. This module should have the following behavior. When memory is ready, the `pixel_location`, `distance` and `tocolor` signals are not of much concern, since nothing is being written to memory. However, we made sure that `pixel_ready` and `tri_done` signals are both low, since they should only be high when there are valid pixel address and data between them. When memory transitions to ready, we first set `tri_done` high for 3 clock cycles. Then, we wait for the a `tri_ready` signal to be sent from the perspective projector, and once that signal has been obtained, we start processing the triangle, and output a sequence of pixel addresses with their distance and to color signals. After we are done looping through all of the appropriate pixels in a triangle for both eyes, we fire a `tri_done` signal, which is held high for 3 clock cycles, and then proceed to the state when we wait for the next `tri_ready` signal. All of the logic for this module is done once every 3 clock cycles, since the specification is that we must hold each pixel address, and data valid for 3 clock cycles so the buffer interface module would have enough time to determine how to update the ZBTs.

In order to loop through all the appropriate pixels, we first loop through the ones for the triangle seen by the first eye, then loop through the ones for the triangle seen by the second eye. For each eye, we find the minimum and maximum x and y coordinates for the triangle, and loop through all the pixels that has an x value between the min and max x coordinate, and a y value between the min and max y coordinate. The reason why we only loop through these pixel locations is all the other pixels will never be inside the triangle, thus it does not need to be updated. For each of the pixels that we do loop through, we determine if the pixel is inside the triangle or not. To do this, we realized the fact that three lines can be drawn for a pixel location to the locations of the three corners, which would produce three triangles. As seen in the Figure 2.1.1, where the solid lines represent the triangle, and the point represents the pixel in question, we can see that if we

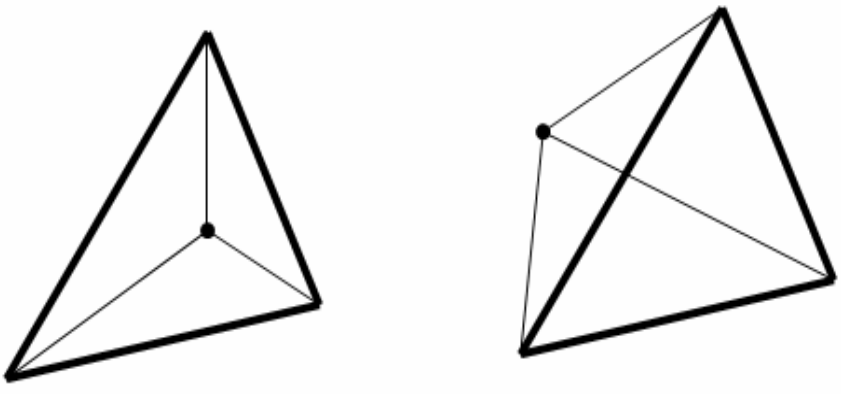


Figure 2.1.1. Determining if a point is within the triangle or not. Notice that if a point is in the triangle the sum of the areas of the 3 sub triangles is equal to the area of the bold triangle.

sum up the three areas of the sub triangles made by the pixel and two of the triangle's corners, and that area is equal to the area of the complete triangle, then the pixel must be inside of the triangle. If not, the pixel must be somewhere outside of the triangle. Using this algorithm, we were able to reliably determine whether a point was inside or outside of the triangle

If a point is found to be inside of the triangle, then we would like the pixel to be updated in the buffer, thus we set the `tocolor` signal to be high. If the point is not inside of the triangle, we set the `tocolor` signal to be low.

The logic of this process is controlled by a FSM. The transition diagram of the state machine is shown below in Figure 2.1.2:

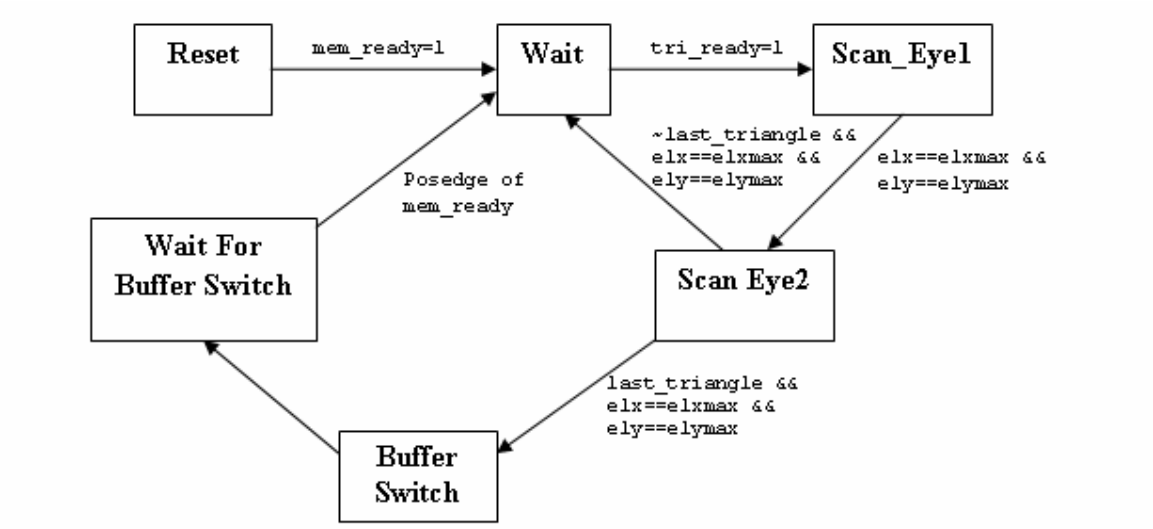


Figure 2.1.2. State transition diagram of the pixel extractor FSM. Regulates the pixel address and pixel data outputs as well as the other control signals

Reset State: When the FSM is in its reset state, it waits for a memory ready signal from the memory informing it that the memory has not been cleared, and is ready for new data. In this state, the outputs of shading ready and done are both zero. When it receives the memory ready signal, the state jumps to the wait state, and triangle done signal is set high which informs the perspective projector module to start work on projecting the next triangle.

Wait State: When the FSM is in its wait state, it knows that the memory is ready. It will stay in this state until a triangle ready signal is high. When the signal is high, which means that the corner locations of the triangle for each eye is valid, the FSM sets the x and y location of the pixel to its minimum values, outputs a `tri_ready` signal of 1, and jumps to the scan eye one state.

Scan Eye One State: In this state, the FSM loops through all the pixels with location inside a rectangle that circumscribes the triangle. When it is finished with the last pixel, it jumps to state scan eye 2, and sets the pixel value to the minimum values for eye 2.

Scan Eye Two State: In this state, the FSM loops through all the pixels with location inside a rectangle that circumscribes the triangle for eye 2. When looping is complete, it checks to see if the triangle was the last triangle. If not, it returns to the wait state and fires a `tri_done` high signal. If it was the last triangle, it jumps to the buffer switch state.

Buffer Switch State: In this state, the FSM simply fires a `tri_done` high signal, and immediately jumps to the wait for buffer switch state.

Wait for Buffer Switch State: This state acts pretty much the same as the reset state. It basically will wait for the memory to be not ready, and then wait until it is ready again before firing a `tri_done` high signal, and returning to the wait state.

Testing of the perspective projector was done using the logic analyzer. Once again, we used the FSM input described above, and made sure that the output pixel location is synchronous with the `tocolor` signal, and the `tocolor` signal was correct in such a way that pixel locations located outside of the triangle does not have a `tocolor` signal of 1, and all those pixel locations located inside of the triangle have a `tocolor` signal of 1. We also checked to make sure that all pixel locations were within a `tri_ready` high and a `tri_done` high signal.

2.1.1. Submodules

Triangle Area Module

The purpose of this module is to calculate twice the area of a triangle, given the (x, y) coordinate of all three of the triangles. This calculation was done by computing the magnitude of the cross product of two vectors, which in effect calculates twice the area of the triangle.

This module was tested using ModelSim testbenches. We tried three coordinates, and arranged them in 6 different combinations and the testbench outputted the correct response for all 6 sets of inputs.

Pixel Distance Module

The purpose of this module is to provide distance estimation to the pixel given the area of the three sub triangles and the distance of the three corners. The idea behind our estimation is if a subtriangle's area is large, it means the pixel is located near the corner that is not in the subtriangle. Thus, if we divide the area of the sub triangle by the total area of the triangle, then that ratio should represent how much weight the

distance of the opposite corner will have on the distance of the current pixel. Thus, if we add up all the three rations multiplied by their respective opposite corner distances, then that would give us a rough estimation for the distance of the pixel.

Testing this module was completed through ModelSim testbenches again. We set distance values for each of the three corners, and made up subtriangle areas. We tested it fairly thoroughly with various different values, and the module worked successfully.

2.2. Color LUT

The color LUT module converts an eight bit color to a 24 bit RGB value. The eight bit input is divided up into 3 bits for R, 3 bits for G, and 2 bits for B. The smaller the input is for each color component, the smaller the output is, and we tried to keep the relative ratio of the input to its max 2 or 3 bit value and the output to its max 8 bit value to be as similar as possible.

To test this module, we used the 8 switches on the labkit to represent our 8 bit input colors, and viewed the 24 bit color on the screen.

2.3. PixelExtractorDelay Module

This module was required because division had a latency of about 30 clock cycles. Thus, in order to match the pixel location with the correct distance values, we had to delay all of the signals coming out of the pixel extractor other than distance by about 30 clock cycles. In order to do so, we used arrays of registers.

We basically tested this module when we tested the Projector_Shader module as a whole. We made sure that after a `shading_ready` signal was high, the next pixel address was correct, and its corresponding data was correct.

2.4. Shading Module

This module is the last step in processing and modifying a pixel's attributes. It takes as input the pixel location which is 1 20 bit concatenation of its x and y coordinate, and converts it to the correct pixel address in the ZBT buffers. The correct ZBT address is 640*y coordinate + x coordinate. It also takes as input the `tri_ready`, and `tri_done` signals, which it simply latches and passes straight through the module. It also obtains the distance and tocolor signals from the PixelExtractorDelay module, and changes the distance output to max 11 bit number if tocolor signal is low. The reason for this is, if we do not need this pixel colored, it is implied that the pixel is not in the triangle, thus that pixel's address should not be updated to some color in the ZBT.

The way we designed our system is that the person is holding a light source that shines in all directions. Thus, the closer the object is from the eye, the brighter the object gets, and

the further the object is from the eye, the darker the object gets. We set the break point to 1024. If the object is exactly 1024 away from the eye, then the color at that pixel is exactly the color of the triangle that was stored in memory. If the object is closer than 1024, we subtract the 1024 by the distance, and add the top 7 bits of that number to each of the RGB components of the triangle color. Of course, we cap it so that the color can not exceed 8'FF. Similarly, if the object is further than 1024, we subtract that distance by 1024, and subtract each of the RGB components of the triangle color by that number. Again, we cap it so that the minimum value of each color component can't go below 0. Doing so will in effect darken or lighten the object depending on its depth from the viewer.

This module was briefly tested using ModelSim testbenches. We hardwired the inputs to certain values, and observed the behavior at the output. The main thing we wanted to test for here was that the output color seemed correct, such that closer objects are lighter colored, and farther objects are darker colored. We also wanted to make sure that distance was set to 11'h7FF, when the tocolor signal is 0. Other than those two, all of the other output signals were simply latched, and we tested the output behavior of the entire Projector_Shader module as a whole to ensure their correct behavior.

3. Read and Write of Pixel Information

Processed data from the shading module needs to get to the screen somehow. Therefore, our project requires some sort of memory. To efficiently store and display pixel color data, we used a double buffer memory scheme. In this scheme, at any point in time, one of the memory buffers is being refreshed by new pixel information, while the other buffer is being displayed by the screen. Once the buffer that is being refreshed is done refreshing, and if the buffer that displays data on the screen is finished displaying a frame, the buffers switch. This way, the buffer that contains new data is now being displayed on the screen and the old data is now being refreshed.

This process involves three main functions, memory write, memory read and the control and coordination of these two processes.

3.1. Memory Write

This section describes how the system stores and handles the processed pixel data.

3.1.1. Buffer Interface Module

The buffer interface module is the heart of the memory write process. It interfaces the shading module with the onboard memory by accepting the pixel information which are processed by the shading module, namely the depth and color of each pixel, and decides whether to store this pixel in the memory or not.

To decide whether to store a pixel in the memory or not, the buffer interface module first checks if the pixel already exists in the memory. One might wonder why the same pixel would be stored twice. Consider the situation, in which there are two overlapping objects in the screen. These two objects have pixels which share the same position. The only difference between them, is that one of these pixels is *deeper* than the other and therefore we do not want to store that pixel anymore because we do not even see it in the screen. Suppose the pixel already exists in the memory, then the existing pixel's depth is compared with the new pixel's depth and whichever pixel is less deep is retained and written to the memory. If the pixel does not exist, we simply store this in memory.

The buffer interface module contains an internal BRAM memory which stores the depth values of the pixels which are being refreshed. It also connects to two onboard SRAM ZBTs to simulate the action of a double buffer.

To test the buffer interface module, the testbench tool using different input values. Tests were run exhaustively ensuring that all possible scenarios were explored. After passing the testbench, we connected the buffer interface module and created a test module which basically stores a green square to the ZBTs and another mini display module which basically displays data from the ZBT. We then compiled this code to the labkit to see if indeed the green square is displayed in the screen. Upon the encounter of bugs, we used

the logic analyzer to debug our code. The final test was to connect this module to the rest of the other modules and ensuring that the correct data is appearing.

3.1.2. Submodules

Depth Buffer Module

This module contains the internal memory of the buffer interface module. It consists of two BRAMS connected to each other. Together they form a memory whose entries are 11 bits long and 204800 addresses deep. This module stores the depth value associated with each pixel and is accessed by the depth comparator module to compare the depths incoming pixel data to the depths of previously stored pixel data.

We tested this module using the testbench and ensured that data is properly written and read by the BRAMS.

Buffer Initiator Module

To be able to store the pixels correctly, the current buffer being written to must be cleared prior to memory write. To do this, we need to initialize the addresses of all the memories used to store pixel information. For this project, the buffer initiator module initializes all the addresses of the ZBT SRAMs (2^{19} addresses) to the color black. It also initializes the internal depth buffer BRAM (204800 addresses) to contain the maximum depth (4095).

It takes about 2^{19} clock cycles to refresh the memory which is roughly $1/130^{th}$ of a second since we are using a 65 Mhz clock. This design could be further optimized by simply, initializing the first 204800 addresses of the ZBTs since we only use these addresses anyway. However, in an effort to catch unexpected glitches, we decided to clear up the whole ZBT memory every refresh to ensure that if somehow we end up displaying an invalid address (an address greater than 204800), the screen would show the color black.

This module takes in as inputs the signals, `clock` and `init`. The clock signal is used to time the flow of the data out of the module. The init signal, on the other hand, is a control signal from the depth comparator module signaling the buffer initiator module to begin initiating the memories. This module sends out the signals `pixeldata`, `pixeladdress`, and `init_done`. `pixeldata` is a 35 bit data which is composed of 24 bit black color and 11 bit max address depth. `pixeladdress` is the address where these data is being written to. This address increments by one each clock cycle during initiation. The `init_done` signal is a control signal to inform the depth comparator module that the initiation process is complete.

We simply used testbench to test this module and ensured that the desired behavior is observed.

Depth Comparator Module

The workhorse of the buffer initiator module, the depth comparator module (DCM) has multiple functions. It acts as a controller to the buffer initiator module, it reads the data coming from the shader module, compare this data by accessing the internal depth buffer, then decide whether to write new incoming data or not. This module is implemented by a 5 state FSM. Refer to Figure 3.1.1 for the FSM state transition diagram.

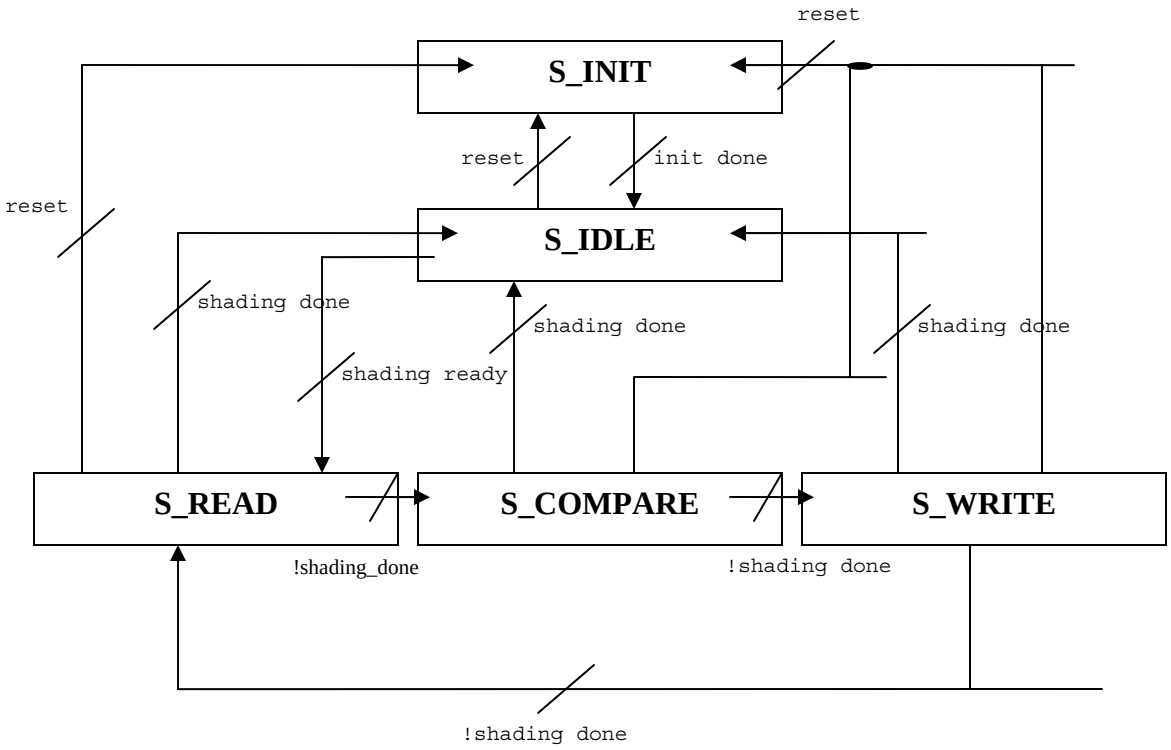


Figure 3.1.1. The depth comparator module FSM state transition diagram. At reset, the depth comparator goes to the initialization state, when initialization is done, it goes to the idle state. Upon receiving a mem_ready signal it goes to the read, compare, write states loop. This loop comprises the reading of a pixel data, comparing it with the memory, and finally writes the proper data. This loop is interrupted if the shading_done signal is received, and the depth comparator goes to its idle state.

The DCM's inputs are `clock`, `reset`, `shading_ready`, `shading_done`, `init_done`, `inputdata` and `pixeladdr`. It's outputs are `writeaddr`, `colordata`, `we`, `init`, and `mem_ready`. The `clock` signal is used to create a synchronized logic. The `reset` signal informs the DCM if the system is resetting or if we are switching buffers. If reset is high, the DCM then goes to its init state and sends an `init` signal to the depth buffer initiator module. The DCM stays in the same state until it receives the `init_done` signal from the depth buffer initiator module. Upon receiving this signal, it then goes to its idle state, firing a `mem_ready` signal and waiting for the `shading_ready` signal from the shading module.

Interfacing with the shader module can be a bit tricky. The shader module holds the data it sends for 3 clock cycles. Because of this when the DCM receives the `shading_ready` signal, it has to delay this ready signal by two clocks to time the state change correctly. After this delay, the DCM enters the read, compare and write state loop. At this loop, it only interacts with a single data from the shader module. At the read state, the DCM accesses the memory element associated with `pixeladdr`. During the compare state, it compares the depth of the memory element with the depth bits of `inputdata`. It then decides whether we should write `inputdata` to memory depending on whether it's deeper or not. Then finally, during the write state, the DCM writes the correct data to the memory or simply leaves the data alone. It keeps on going through this loop for each data passed in, until it receives the `shading_done` signal which makes it go to the idle state.

To communicate with the ZBT, the DCM, passes `colordata`, the 24 bit color signal from `inputdata` to the colordata_to_buffer module for further routing. It also passes `writeaddr`, the address associated with `colordata` along with `we`, the control signal which signifies whether we should write to the ZBT or not.

We tested this module by using both the testbench and logic analyzer to clear the bugs.

Colordata_to Buffer Module

This module properly routes the pixel address and color data to the correct ZBT. It also assigns the final write enable signals to both ZBTs. Its inputs are `clock`, `reset`, `buffer_sel`, `writeaddr`, `colordata`, and `we`. Its outputs are `ram0_wdata`, `ram0_addr`, `ram0_we`, `ram1_wdata`, `ram1_addr`, and `ram1_we`.

To route `we`, `colordata` and `writeaddr` to the correct ZBT, this module uses the `buffer_sel` signal. The `buffer_sel` signal informs the module which buffer is currently being written. For example, if `buffer_sel` is equal to 1, then we are writing to ZBT ram 1. Therefore, the colordata to buffer module assigns `we`, `colordata` and `writeaddr` to `ram1_we`, `ram1_wdata`, and `ram1_addr`, respectively. It then assigns the value 0 to the ram 0 pins, namely, `ram0_we`, `ram0_wdata`, and `ram0_addr` since we are not writing to ram 0. If `buffer_sel` is equal to 0, it does the opposite thing.

This module was tested using the testbench to see if the desired routing scheme was working.

3.1.3. Top Level Labkit Connections

The color data, address, and write enable signals coming from the buffer interface module get passed to two instances of another ZBT interface module created by one of the staff. This module is called zbt_6.111. This takes care of the final preparation before the data gets passed to the ZBT. The color data and write enable signals get directly

wired to the zbt_6.111 modules. The address, however need to pass through a multiplexer since only one type of address needs to be read by each ZBT – either the write address or the read address. The multiplexer is controlled by the buffer select signal.

3.2. Memory Read

This section describes how the system reads and displays the pixel data stored in the ZBT memory.

3.2.2. Displayer Module

The displayer module controls what appears in the monitor. Takes in `vclock`, `reset`, `hcount`, `vcount`, and `fromzbt_data` as inputs and outputs `tozbt_addr`, `pixel`, and `donedisplaying`. The `hcount and vcount` signals inform the displayer module what pixel is currently being refreshed in the screen. The displayer then decides whether the current `hcount and vcount` signals appear within a set boundary of pixels in which correspond to the viewpoints of the eyes. These boundaries are two 320 by 320 "screens" located in the center of the monitor. Refer to Figure 3.2.1 to see how these two screens are located.

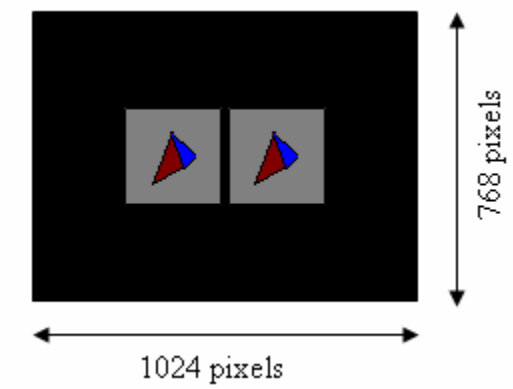


Figure 3.2.1. Position of the two 320 by 320 screens. The two screens are the two grey boxes displaying a pyramid. The black rectangle represents the monitor screen.

This module behaves differently depending whether `hcount` and `vcount` are inside these two screens. If `hcount` and `vcount` are not inside these screens or if `reset` is high, then displayer should be displaying a black screen. In order to this, it assigns `tozbt_addr` to a special address in memory which contains the color black. However, if the displayer module is within the two screens, it has to work harder and do more computations to access the correct memory address associated with the given `hcount` and `count`.

A 3 stage pipeline was used to access the pixel data associated with `hcount` and `vcount`. Figure 3.2.2 shows a schematic of the pipeline used. The first stage involves the computation of the offset address which corresponds to the current `hcount` and `vcount`. The displayer does this by first normalizing the coordinates of `hcount` and `vcount` with respect to the upper leftmost corner of screen one. In effect, the upper leftmost corner

was defined to be 0,0 while `hcount` and `vcount` where defined to be xoffset and yoffset respectively. Figure 3.2.3 shows how xoffset and yoffset are determined. To obtain the offset address we calculate 640*yoffset + xoffset and we latch this for one clock cycle to avoid glitches. The second stage involves passing this value of offset address to `tozbt_addr` which is then sent to the appropriate ZBT to access the color value associated with `hcount` and `vcount`. The third stage involves accessing `fromzbt_data` and assign it to the `pixel` data output. All in all, the total latency time of this pipeline is 5 clock cycles.

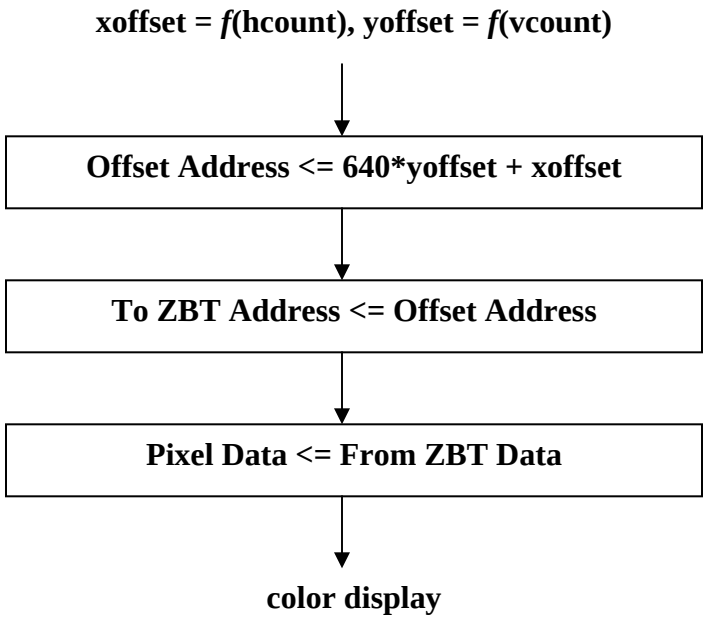


Figure 3.2.2. 3 Stage pipeline of the displayer module. Takes 5 clock cycles from the time hcount and vcount are generated from the time the corresponding pixel color appears on the screen.

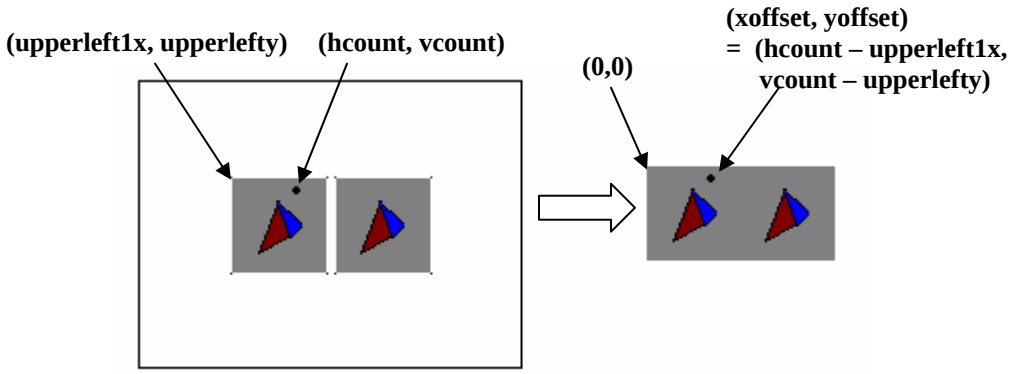


Figure 3.2.3. Graphical interpretation of xoffset and yoffset. Left figure shows the absolute position of hcount and vcount vs the upperleft corner of the two screents. Right figure describes the transformation of coordinates by calculating xoffset and yoffset.

Aside from displaying pixels on the screen, the displayer module also communicates with the buffer selector module by informing it that one screen full of data has been displayed. This occurs when `hcount` and `vcount` are equal to 1029 and 767 repectively.

To test the displayer, we first used testbench to check if the delays are correct. We then connected this to the bufferinterface module and check if we can display data which we inputted to the ZBT. We probed the connections using the logic analyzer to debug the circuit.

3.2.3. Top Level Connections

The address coming out of the displayer is multiplexed with the data being written to the ZBT and gets routed to the proper ZBT by the `buffer_sel` signal. The displayer module is also clocked by a 65 MHz clock since it uses XVGA graphics.

3.2.4. XVGA Module

This module was provided by the 6.111 staff. It basically generates signals to display pixels on the screen. This module is connected to the displayer module and together they display pixels on the screen.

3.2.5. DelayN Module

A module used to delay `hsync`, `vsync` and `blank` signals by 5 clock cycles to compensate for the 5 clock cycle delay of the displayer pipeline. This was also provided by the 6.111 staff.

3.3. Read and Write Control

3.3.1. Buffer Selector Module

Such an elaborate read and write system, needs some sort of a control and for this system, it is provided by the buffer selector module. This module controls the switching of the buffers and ensures that this is not detected by someone viewing the screen. This module's inputs are `clock`, `reset`, `donedisplaying`, and `triangledone` while its outputs are `switchbuffers` and `buffer_sel`.

We want our buffers to switch roles every time one frame is finished rendering and if the monitor is finished displaying. In order to achieve this, the buffer selector module listens for the `triangledone` signal sent by the shader module. It then latches this data in a register called `tdreg`, since `triangledone` is a pulse. At the same time, the buffer selector also listens for the `donedisplaying` signal sent by the displayer module and if `tdreg` or `triangledone` and `donedisplaying` are high. If this combined signal is high, then a signal called `switch` goes high.

The switching of the buffers is controlled by a two state FSM. Figure 3.3.1 shows the state transition diagram of this FSM. The states of this FSM are called s_writeto0 and s_writeto1. At `reset`, the default state is s_writeto0. During this state, we are writing to buffer 0, thus, `buffer_sel` must be set to 0. When it is time to switch buffers, in other words, `switch` is high, `switchbuffers` is set to 1 for one clock cycle, `buffer_sel` is set to 1 and we transition to the second state which is s_writeto1. The behavior of this state is similar to the first state except that `buffer_sel` is set to 1 all throughout this state.

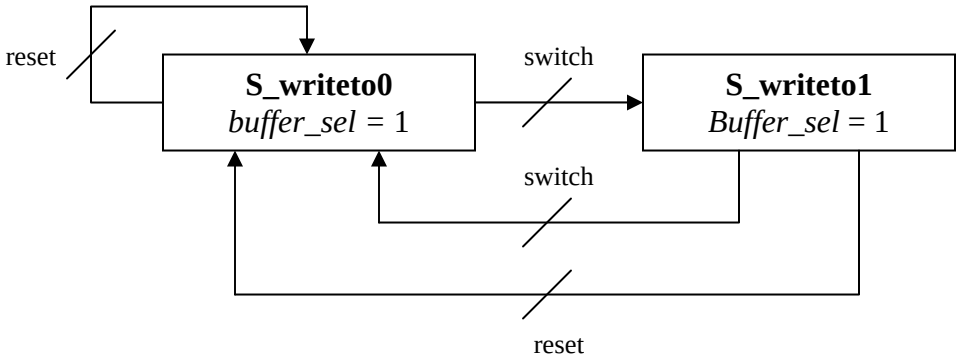


Figure 3.3.1. State transition diagram for the buffer selector. Note that at reset, the FSM transitions to s_writeto0, otherwise, the states just switch.

We tested this module by simply using testbench because it is very straightforward and does not involve tricky time constraints. We also tested this module by integrating it with our other modules and displaying a green square and a red triangle which are stored in buffer 0 and buffer 1 respectively. We then connected the `buffer_sel` signal to the LEDs. We used switches to freeze the frame and checked if the proper picture appears with respect to the `buffer_sel` signal..

4. 3D World Generation

The 3D world we implemented consists of a cube, walls, and 3D block letters which spell MIT. We generated the 3D world using triangles. Because of time constraint, we did not have time to generate more complex objects. However, simple objects are enough since this project is testing, not our artistic skills, but the engine we developed to display images.

The system we have developed only needs the coordinates of the triangles which make up the world to generate its 3D image. To find these coordinates, we first drew our objects on paper and then started creating coordinates for the triangles using Excel. We realized that each the objects were merely translations of similar rectangles in the x, y, and z axis. Using this method, it was actually quite easy to finish hand plotting the cube and the MIT block. However, the wall was a tough task. It requires 260 triangles to render – therefore, a total of 780 coordinates. However, we realized that the wall is just made up of shifted triangles so we made a program in MATLAB to automatically generate the wall and store the coordinates in a matrix format.

To create the .coe files required by the triangle ROM we used MATLAB to transform the coordinates we have extracted to a .coe file written in decimal radix. We concatenated the x, y, and z coordinates of each triangle into one 36 bit decimal value.

If we had more time, we could have written more extensive MATLAB scripts to automatically generate specific polygons in the 3D plane. This way we could have made more complex looking objects.

5. Testing and Debugging

The testing tools we used were the Xilinx testbench, ModelSim Waveform analyzer, and the Tektronix logic analyzer. Each of these tools has a specific task in the testing and debugging of our project.

Before attempting to integrate, we exhaustively tested each module with either the testbench or the waveform analyzer. The testbench was used for circuits which do not have tight timing constraints. Slightly more complicated logic was then placed and routed then tested using the waveform analyzer which gave a more accurate timing diagram of the modules. It is a better test that the testbench in the sense that it actually maps the device and provides a rough estimate of the timing delays associated with the circuit. After ensuring that each module works independently from each other we finally integrated.

As expected, timing bugs and glitches began to appear. One of the first problems we encountered is the appearance of colored lines at the screen. At first we had absolutely, no idea, why this occurred but after asking Professor Chuang for his opinion, we realized that some of our calculations could not be calculated in one clock. Therefore we have to adjust our code and make our clock run a little bit slower. Particularly, some of our modules gets triggered only once every three clocks. Since we made the clock slower, we have to compensate for the loss in speed by pipelining our modules. This ensured first of all that our data are latched; secondly this is a form of optimization that would produce huge benefits in the long run.

The next types of problems we encountered are communication and mistiming of data signals. One example to characterize this problem is when our screen freezes, and buffer select never changes. This problem occurred because of a miscommunication between three essential modules in our project. We tracked down the cause of this problem by using the logic analyzer. It is a very powerful tool, since you could trigger at specific areas of logic. It immediately pinpointed to us where and what the problem was.

After we fixed our communications and accommodated all our propagation delays, our project was still crashing. More weird lines begin appearing and pixels aren’t being shaded. The screen is also flickering hard. At this point, we had absolutely no clue what was going on until Professor Terman informed us that clocks overload. Apparently, these clock signals get routed to a lot of components and he told us that a clock could efficiently drive around a 5000 component load. When we looked at our synthesis report, we found out that our clock is driving almost 13000 components. We then decided to create two more clocks which we use to drive our divider modules which contain the largest amount of clock load. We did this and we got rid of the lines in the screen, however our screen was still flickering. After attempts of optimizing our logic even further, we realized that bitwise operations produce horrendous amounts of flicker, we then decided to multiplex our data before assigning them to wires. This was the last piece of the puzzle, the flicker was gone and we finally saw solid 3D objects appear in the screen.

One thing which made debugging and testing a challenge was the long synthesis time. Due to the large amount of resources we use: 70% of the slices, 99% of the memory, 80% of the bonded IOBs, our project took about 30 to 40 minutes to compile. We were forced to debug smartly because we could only test our new logic once every thirty minutes. Thus, we made sure we gathered enough information from our logic each time we synthesized it. To do this, we used all of our logic analyzer connectors, LEDs, and the 64 Hex Display.

Overall, we believe that we learned a lot of concepts and tricks in the debugging process. We are relieved that after spending hours debugging, we finally got our project to run.

Conclusion

This project was a huge learning experience for us. We started this project knowing basically nothing about graphics processing but we finished with a primitive yet working 3D engine. The whole process of designing, implementing, and debugging this complex project definitely taught us the challenges of a graphics card designer. As we were working on this project, we happened to read specifications and literature about graphics cards and surprisingly, we understood a lot of terminology that graphics designers use. Terms like overclocking, unlocking pipelines, 256 bit buses, and 256 Megabytes of onboard memory made sense.

This project was very neat in the sense that we got to apply most of the concepts we learned in class to get our project up and running. The previous labs definitely helped prepare us in terms of designing, debugging, and efficient coding, but there are so many things that could arise in such a complex project. We learned that unexpected things occur – even though we thought that our logic was perfect, there always seems to be a problem somewhere. For example, each of our different modules seem to work fine independently, however, when we integrated our modules, we found that insidious problems in the form of data path conflicts, module miscommunications, timing glitches, clock overloading occur. Debugging and tracking down these problems was definitely the most time-consuming part, since for most of the time we do not understand what is making our project fail. However, in the end it was satisfying when we finally got our 3D engine running.

We made several design changes in order for the project to work in the time allotted. Our initial plan was to display two different images into two different screens and integrate two laboratory kits together. However, after realizing that our memory is not enough, we decided on displaying 2 smaller images on one screen. We also had a more elaborate design in our image shading and rendering, however due to the limitations of the math modules, we made simplifications in our shading module. Our group also chose from a number of projection schemes to project our 3D images into a 2D plane. In the end we have to use ray casting, a method used by games such as Doom. It involves simpler mathematics unlike modern methods which involves huge matrix multiplications. These calculations are beyond the scope of our knowledge, as we did not take any graphics class prior to this project. Thus it would be almost impossible for us to implement it in our labkit and produce optimal results. Modern graphics cards process these calculations using dedicated hardware and optimized chips to achieve the desired results in minimal time. Our FPGA might not be able to emulate this ability.

We had a great time working on the project albeit the frustrations and long hours during the last few days. In the end, we were very satisfied with the results. We would like to take this opportunity to thank Professors Terman and Chuang for the incredible wealth of knowledge they shared regarding the issues we had especially with clock overloading. We also wish to thank all the teaching assistants, Javier, Willie, Jae and Eric, for giving us invaluable assistance and advice whenever we have questions. We believe that this class was really excellent and an excellent way to learn digital logic design.

```
/*********************************************************************************
Written by: Joshua Monzon, Tony Ng, Steven Zhou
6.111 Final Project: Digital Stereoscope

The top level module used to compile the whole project into the labkit. Contains integrated
submodule to create a digital stereoscope. Displays a 3D rendered world in the monitor from
two different viewpoints. These viewpoints correspond to what each eye sees. The user could
use the keyboard to move around the world.

all rights reserved.

Acknowledgements: Labkit code was made by Nathan Ickes.
*********************************************************************************/
module digital_stereoscope(beep, audio_reset_b,
              ac97_sdata_out, ac97_sdata_in, ac97_synch,
           ac97_bit_clock,

           vga_out_red, vga_out_green, vga_out_blue, vga_out_sync_b,
           vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync,
           vga_out_vsync,

           tv_out_ycrcb, tv_out_reset_b, tv_out_clock, tv_out_i2c_clock,
           tv_out_i2c_data, tv_out_pal_ntsc, tv_out_hsync_b,
           tv_out_vsync_b, tv_out_blank_b, tv_out_subcar_reset,

           tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1,
           tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
           tv_in_i2c_clock, tv_in_i2c_data, tv_in_fifo_read,
           tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,

           ram0_data, ram0_address, ram0_adv_ld, ram0_clk, ram0_cen_b,
           ram0_ce_b, ram0_oe_b, ram0_we_b, ram0_bwe_b,

           ram1_data, ram1_address, ram1_adv_ld, ram1_clk, ram1_cen_b,
           ram1_ce_b, ram1_oe_b, ram1_we_b, ram1_bwe_b,

           clock_feedback_out, clock_feedback_in,

           flash_data, flash_address, flash_ce_b, flash_oe_b, flash_we_b,
           flash_reset_b, flash_sts, flash_byte_b,

           rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

           mouse_clock, mouse_data, keyboard_clock, keyboard_data,

           clock_27mhz, clock1, clock2,

           disp_blank, disp_data_out, disp_clock, disp_rs, disp_ce_b,
           disp_reset_b, disp_data_in,

           button0, button1, button2, button3, button_enter, button_right,
           button_left, button_down, button_up,

           switch,

           led,

           user1, user2, user3, user4,

           daughtercard,

           systemace_data, systemace_address, systemace_ce_b,
           systemace_we_b, systemace_oe_b, systemace_irq, systemace_mpbrdy,

           analyzer1_data, analyzer1_clock,
           analyzer2_data, analyzer2_clock,
           analyzer3_data, analyzer3_clock,
           analyzer4_data, analyzer4_clock);

    output beep, audio_reset_b, ac97_synch, ac97_sdata_out;
    input  ac97_bit_clock, ac97_sdata_in;
```

```
output [7:0] vga_out_red, vga_out_green, vga_out_blue;
output vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock,
    vga_out_hsync, vga_out_vsync;

output [9:0] tv_out_ycrcb;
output tv_out_reset_b, tv_out_clock, tv_out_i2c_clock, tv_out_i2c_data,
    tv_out_pal_ntsc, tv_out_hsync_b, tv_out_vsync_b, tv_out_blank_b,
    tv_out_subcar_reset;

input  [19:0] tv_in_ycrcb;
input  tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2, tv_in_aef,
    tv_in_hff, tv_in_aff;
output tv_in_i2c_clock, tv_in_fifo_read, tv_in_fifo_clock, tv_in_iso,
    tv_in_reset_b, tv_in_clock;
inout  tv_in_i2c_data;

inout  [35:0] ram0_data;
output [18:0] ram0_address;
output ram0_adv_ld, ram0_clk, ram0_cen_b, ram0_ce_b, ram0_oe_b, ram0_we_b;
output [3:0] ram0_bwe_b;

inout  [35:0] ram1_data;
output [18:0] ram1_address;
output ram1_adv_ld, ram1_clk, ram1_cen_b, ram1_ce_b, ram1_oe_b, ram1_we_b;
output [3:0] ram1_bwe_b;

input  clock_feedback_in;
output clock_feedback_out;

inout  [15:0] flash_data;
output [23:0] flash_address;
output flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b, flash_byte_b;
input  flash_sts;

output rs232_txd, rs232_rts;
input  rs232_rxd, rs232_cts;

input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

input  clock_27mhz, clock1, clock2;

output disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b;
input  disp_data_in;
output  disp_data_out;

input  button0, button1, button2, button3, button_enter, button_right,
    button_left, button_down, button_up;
input  [7:0] switch;
output [7:0] led;

inout [31:0] user1, user2, user3, user4;

inout [43:0] daughtercard;

inout  [15:0] systemace_data;
output [6:0]  systemace_address;
output systemace_ce_b, systemace_we_b, systemace_oe_b;
input  systemace_irq, systemace_mpbrdy;

output [15:0] analyzer1_data, analyzer2_data, analyzer3_data,
    analyzer4_data;
output analyzer1_clock, analyzer2_clock, analyzer3_clock, analyzer4_clock;

//////////////////
//
// I/O Assignments
//
//////////////////

// Audio Input and Output
```

```
assign beep= 1'b0;
assign audio_reset_b = 1'b0;
assign ac97_synch = 1'b0;
assign ac97_sdata_out = 1'b0;

// Video Output
assign tv_out_ycrcb = 10'h0;
assign tv_out_reset_b = 1'b0;
assign tv_out_clock = 1'b0;
assign tv_out_i2c_clock = 1'b0;
assign tv_out_i2c_data = 1'b0;
assign tv_out_pal_ntsc = 1'b0;
assign tv_out_hsync_b = 1'b1;
assign tv_out_vsync_b = 1'b1;
assign tv_out_blank_b = 1'b1;
assign tv_out_subcar_reset = 1'b0;

// Video Input
assign tv_in_fifo_read = 1'b1;
assign tv_in_fifo_clock = 1'b0;
assign tv_in_iso = 1'b1;
assign tv_in_clock = clock_27mhz;//1'b0;

////////////////
// ZBT SRAMs
////////////////
// enable RAM0 pins
assign ram0_ce_b = 1'b0;
assign ram0_oe_b = 1'b0;
assign ram0_adv_ld = 1'b0;
assign ram0_bwe_b = 4'h0;

// enable RAM1 pins
assign ram1_ce_b = 1'b0;
assign ram1_oe_b = 1'b0;
assign ram1_adv_ld = 1'b0;
assign ram1_bwe_b = 4'h0;

assign clock_feedback_out = 1'b0;

// Flash ROM
assign flash_data = 16'hZ;
assign flash_address = 24'h0;
assign flash_ce_b = 1'b1;
assign flash_oe_b = 1'b1;
assign flash_we_b = 1'b1;
assign flash_reset_b = 1'b0;
assign flash_byte_b = 1'b1;

// RS-232 Interface
assign rs232_txd = 1'b1;
assign rs232_rts = 1'b1;

// User I/Os
assign user1 = 32'hZ;
assign user2 = 32'hZ;
assign user3 = 32'hZ;
assign user4 = 32'hZ;

// Daughtercard Connectors
assign daughtercard = 44'hZ;

// SystemACE Microprocessor Port
assign systemace_data = 16'hZ;
assign systemace_address = 7'h0;
assign systemace_ce_b = 1'b1;
assign systemace_we_b = 1'b1;
assign systemace_oe_b = 1'b1;

// use FPGA's digital clock manager to produce a
// 65MHz clock (actually 64.8MHz)
```

```
wire clock_65mhz_unbuf,clock_65mhz,xclock_65mhz,x2clock_65mhz,x3clock_65mhz;
DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_65mhz_unbuf));
// synthesis attribute CLKFX_DIVIDE of vclk1 is 10
// synthesis attribute CLKFX_MULTIPLY of vclk1 is 24
// synthesis attribute CLK_FEEDBACK of vclk1 is NONE
// synthesis attribute CLKIN_PERIOD of vclk1 is 37
BUFG vclk2(.O(clock_65mhz),.I(clock_65mhz_unbuf));
BUFG vclk3(.O(xclock_65mhz),.I(clock_65mhz_unbuf));
BUFG vclk4(.O(x2clock_65mhz),.I(clock_65mhz_unbuf));

wire clk = clock_65mhz;          // default clock for the light load modules
wire clk_div1 =  xclock_65mhz;   // clock of one of the high clock load divider modules
wire clk_div2 =  x2clock_65mhz;  // clock of the other high clock load divider module

// power-on reset generation
wire power_on_reset;             // remain high for first 16 clocks
SRL16 reset_sr (.D(1'b0), .CLK(clk), .Q(power_on_reset),
    .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
defparam reset_sr.INIT = 16'hFFFF;

// ENTER button is user reset
wire reset,user_reset;
debounce db1(power_on_reset, clk, ~button_enter, user_reset);
assign reset = (user_reset || power_on_reset) ? 1 : 0;

// Display module used for debugging
reg [63:0] dispdata;
display_16hex hexdisp1(reset, clk, dispdata, disp_blank, disp_clock, disp_rs, disp_ce_b,
    disp_reset_b, disp_data_out);

// VGA Output.  In order to meet the setup and hold times of the
// AD7125, we send it ~clock_65mhz.
assign vga_out_red = pixel[23:16];
assign vga_out_green = pixel[15:8];
assign vga_out_blue = pixel[7:0];
assign vga_out_sync_b = 1'b1;    // not used
assign vga_out_pixel_clock = ~clock_65mhz;
assign vga_out_blank_b = ~b;
assign vga_out_hsync = hs;
assign vga_out_vsync = vs;

///////////////
// Debugging
///////////////
always @(posedge clk)
    dispdata <= {ram0_wdata, 9'b0, ram0_addr};
assign led = ~{state, ~buffer_sel, buffer_sel, mem_ready, lasttriangledone};

//////////////////////////////////////////////////////////////////////////////////
// SLOW Clock - runs at 1/3 the speed of the 65 MHz clock. Used to accomodate the shader
// and comparator modules which require 3 clock cycles to do their computations
//////////////////////////////////////////////////////////////////////////////////
reg slow_clk;
reg [1:0] counter;

always @(posedge clk) begin
    if (reset) begin
        counter<=0;
        slow_clk<=1;
    end

    else if (counter==2) begin
        counter <= 0;
        slow_clk <= 1;
    end

    else begin
        counter <= counter+1;
        slow_clk <= 0;
    end
end
```

```
 //////////////////////
 // XVGA video signals
 //////////////////////
wire [10:0] hcount;
wire [9:0]  vcount;
wire hsync, vsync, blank;
xvga xvga1(clk, hcount, vcount, hsync, vsync, blank);

 ////////////////////////////////////////////////
 // ZBT SRAMs - connections to the onboard SRAMs
 ////////////////////////////////////////////////
wire [35:0] ram0_read_data, ram1_read_data;
wire [35:0] ram0_wdata, ram1_wdata;
wire ram0_we, ram1_we;
wire [18:0] zbt0_addr, zbt1_addr;

zbt_6111 zbt0(zbt0_clk, 1'b1, ram0_we, zbt0_addr,
       ram0_wdata, ram0_read_data,
       ram0_clk, ram0_we_b, ram0_address, ram0_data, ram0_cen_b);

zbt_6111 zbt1(zbt1_clk, 1'b1, ram1_we, zbt1_addr,
       ram1_wdata, ram1_read_data,
       ram1_clk, ram1_we_b, ram1_address, ram1_data, ram1_cen_b);

 ////////////////////////////////////////////////////////////////////////////
 // Displayer - extracts data from the ZBTs and displays them on the screen
 ////////////////////////////////////////////////////////////////////////////
wire buffer_sel, switchbuffers;
wire [23:0] pixel;
wire sds;
wire [35:0] fromzbt_data = buffer_sel? ram0_read_data : ram1_read_data ;
wire [18:0] tozbt_addr;
wire donedisplaying;

 displayer disp (clk, reset, hcount, vcount, fromzbt_data,
       tozbt_addr, pixel, donedisplaying, switchbuffers);

////////////////////////////////////////////////////////////////////////////
 // Read and Write muxes - multiplexers to route the proper address and
 // clocks to the correct ZBTs
 ////////////////////////////////////////////////////////////////////////////
 assign zbt0_addr = buffer_sel ? tozbt_addr: ram0_addr;
 assign zbt1_addr = buffer_sel ? ram1_addr : tozbt_addr;
 assign zbt1_clk = clk;
 assign zbt0_clk = clk;

 ////////////////////////////////////////////////////////////////////////////
 // Delay Path - delays the hsync, vsync, and blank signals by 5 clock
 // cycles to accomodate the 3 stage pipeline of the displayer module
 ////////////////////////////////////////////////////////////////////////////
 wire   b, hs, vs;

 delayN dn1(clk, hsync, hs);
 delayN dn2(clk, sync, vs);
 delayN dn3(clk, blank, b);

 ////////////////////////////////////////////////////////////////////////////
 // 3D World - User Movement, Projection, Rendering, Shading, Storage Path
 ////////////////////////////////////////////////////////////////////////////
 wire [17:0] pixeladdr;
 wire [34:0] pixeldata;
 wire [18:0] ram0_addr, ram1_addr;
 wire [32:0] e1c1, e1c2, e1c3, e2c1, e2c2, e2c3;
 wire [15:0] gaze_angle, tilt_angle;
 wire [35:0] eye1, eye2;
 wire [9:0] num_triangles;
 wire [8:0] triangle_addr;
 wire [7:0] inColor;
 wire [35:0] corner1, corner2, corner3;
 wire [7:0] color;
```

```
    // Control and Communication Signals
    wire shading_ready, shading_done;
    wire mem_ready;
    wire tri_ready;
    wire last_triangle;
    wire lasttriangledone;
    wire want_new_triangle;

    reg old_frame, new_frame;

    always @ (posedge clk) begin
        if (slow_clk) begin
            old_frame <= last_triangle;
            new_frame <= (last_triangle & (~ old_frame));
        end
    end

    // Outputs the corners and color of triangles stored in memory
    trianglememory tmem(.addr(triangle_addr), .clk(clk), .din1(36'b0), .din2(36'b0),
                        .din3(36'b0), .dinc(8'b0), .dout1(corner1), .dout2(corner2),
                        .dout3(corner3), .doutc(inColor), .we(1'b0),
                        .num_triangles(num_triangles));

    // Takes inputs from keyboard and updates eye position, gaze and tilt angles
    position_controller posctrl(.clock(clk), .new_frame(new_frame), .reset(reset),
                        .keyboard_clock(keyboard_clock), .keyboard_data(keyboard_data),
                        .eye1(eye1), .eye2(eye2), .gaze_angle(gaze_angle),
                        .tilt_angle(tilt_angle));


    // Projects corners of triangle in 3-D space onto the screen
   perspectiveprojector pp(.clk(clk), .slow_clk(slow_clk), .reset(reset),
                        .corner1(corner1), .corner2(corner2), .corner3(corner3),
                        .inColor(inColor), .eye1(eye1), .eye2(eye2), .gaze_angle(gaze_angle),
                        .tilt_angle(tilt_angle), .start(want_new_triangle),
                        .num_triangles(num_triangles), .reset_last_triangle(lasttriangledone),
                        .outColor(color), .triangle_addr(triangle_addr),.e1c1(e1c1), .e1c2(e1c2),
                        .e1c3(e1c3), .e2c1(e2c1), .e2c2(e2c2), .e2c3(e2c3), .done(tri_ready),
                        .last_triangle(last_triangle));

    // Extracts pixels' addresses and colors as well as routing signals
    Projector_Shader ps (.clk(clk), .clk_div1(clk_div1), .clk_div2(clk_div2),
                        .slow_clk(slow_clk),
                        .reset(reset), .mem_ready(mem_ready), .tri_ready(tri_ready), .e1c1(e1c1),
                        .e1c2(e1c2), .e1c3(e1c3), .e2c1(e2c1), .e2c2(e2c2), .e2c3(e2c3),
                        .color(color), .last_triangle(last_triangle),
                        .shading_ready(shading_ready), .shading_done(shading_done),
                        .pixel_address(pixeladdr), .pixel_data(pixeldata),
                        .want_new_triangle(want_new_triangle),
                        .last_triangle_shading_done(lasttriangledone));

    // Controls which buffer is being written to and read from
   bufferselector bselector(.clock(clk), .reset(reset), .donedisplaying(donedisplaying),
                        .triangledone(lasttriangledone),.switchbuffers(switchbuffers),
                        .buffer_sel(buffer_sel));

    // Interfaces the shader module to the ZBTS, controls overlap of images
    bufferinterface datatobuffer (.clock(clk), .reset(reset), .shading_ready(shading_ready),
                        .shading_done(shading_done),.pixeldata(pixeldata), .pixeladdr(pixeladdr),
                        .ram0_wdata(ram0_wdata), .ram0_addr(ram0_addr), .ram0_we(ram0_we),
                        .ram1_wdata(ram1_wdata), .ram1_addr(ram1_addr), .ram1_we(ram1_we),
                        .mem_ready(mem_ready), .buffer_sel(buffer_sel),
                        .switchbuffers(switchbuffers));
endmodule

//////////////////////////////////////////////////////////
// xvga: Generate XVGA display signals (1024 x 768 @ 60Hz)
//////////////////////////////////////////////////////////
module xvga(vclock, hcount, vcount, hsync, vsync, blank);
   input vclock;
```

```
   output [10:0] hcount;
   output [9:0] vcount;
   output  vsync;
   output  hsync;
   output  blank;

   reg    hsync, vsync, hblank, vblank, blank;
   reg [10:0]   hcount;    // pixel number on current line
   reg [9:0] vcount;   // line number

   // horizontal: 1344 pixels total
   // display 1024 pixels per line
   wire      hsyncon, hsyncoff, hreset, hblankon;
   assign    hblankon = (hcount == 1023);
   assign    hsyncon = (hcount == 1047);
   assign    hsyncoff = (hcount == 1183);
   assign    hreset = (hcount == 1343);

   // vertical: 806 lines total
   // display 768 lines
   wire      vsyncon, vsyncoff, vreset, vblankon;
   assign    vblankon = hreset & (vcount == 767);
   assign    vsyncon = hreset & (vcount == 776);
   assign    vsyncoff = hreset & (vcount == 782);
   assign    vreset = hreset & (vcount == 805);

   // sync and blanking
   wire      next_hblank, next_vblank;
   assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
   assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
   always @(posedge vclock) begin
      hcount <= hreset ? 0 : hcount + 1;
      hblank <= next_hblank;
      hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;  // active low

      vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;
      vblank <= next_vblank;
      vsync <= vsyncon ? 0 : vsyncoff ? 1 : vsync;  // active low

      blank <= next_vblank | (next_hblank & ~hreset);
   end
endmodule

////////////////////////////
// parameterized delay line
////////////////////////////
module delayN(clk, in, out);
   input clk;
   input in;
   output out;

   parameter NDELAY = 5;

   reg [NDELAY-1:0] shiftreg;
   wire        out = shiftreg[NDELAY-1];

   always @(posedge clk)
     shiftreg <= {shiftreg[NDELAY-2:0],in};

endmodule
```

```
/*********************************************************************************************
Written by: Joshua Jen Monzon
6.111 Final Project: Digital Stereoscope

This module interfaces the shading module with the ZBT SRAMS. Takes in control signals
when the pixels of a triangle need to be stored in the SRAMS and when a triangle is done
rendering. It also takes in the corresponding pixeldata and pixel address where the data
will be stored. It then routes the data to the proper ZBT pins depending which buffer is
selected and whether the data needs to be written or not. Comprised of different submodules
which performs a specific function in the buffer interface.
*********************************************************************************************/
module bufferinterface (clock, reset, shading_ready, shading_done, pixeldata, pixeladdr,
       ram0_wdata, ram0_addr, ram0_we, ram1_wdata, ram1_addr, ram1_we,
       mem_ready, buffer_sel, switchbuffers);

    input clock, reset;          // global signals
    input shading_ready;         // means rendered pixels are starting to be sent
    input shading_done;          // means a triangle is done rendering
    input [34:0] pixeldata;      // {24b'color, 11b'depth}, information of a pixel
    input [17:0] pixeladdr;      // address to store the pixel (640*y + x)

    input buffer_sel;            // current buffer being written to
    input switchbuffers;         // pulse indicating buffers are switching

    output [35:0] ram0_wdata, ram1_wdata;   // color data written to corresponding ZBT's
    output [18:0] ram0_addr, ram1_addr;     // address where the data is written to
    output ram0_we, ram1_we;
    output mem_ready;           // signal indicating memory is ready to accept data

    wire [34:0] inputdata, bufferinitdata;
    wire [18:0] writeaddr, bufferinitaddr, inputaddr;
    wire [23:0] colordata;
    wire mem_ready, init_done, init, we, switchbuffers, buffer_sel;
    wire dcreset;

    ////////////////////////////////////////////////////////
    // Refresh buffers during reset or when switching buffers
    ////////////////////////////////////////////////////////
    assign dcreset = (reset || switchbuffers) ? 1 : 0;

    ////////////////////////////////////////////////////////////////////////////////////
    // Chooses what data will be written in the memory, if we are initializing
    // then we want data from the initiator otherwise we get data from the shading module
    ////////////////////////////////////////////////////////////////////////////////////
    assign inputdata = mem_ready ? pixeldata : bufferinitdata;
    assign inputaddr = mem_ready ? {1'b0, pixeladdr} : bufferinitaddr;

    ////////////////////////////////////////////////////////////////////////
    // Compares depth and makes decisions whether to write to the memory or not
    ////////////////////////////////////////////////////////////////////////
    depthcomparator dc (.clock(clock), .reset(dcreset), .shading_ready(shading_ready),
        .shading_done(shading_done), .init_done(init_done), .inputdata(inputdata),
        .pixeladdr(inputaddr), .writeaddr(writeaddr), .colordata(colordata), .we(we),
        .init(init), .mem_ready(mem_ready));

    ////////////////////////////////////////////////////////////////////////
    // Initiates the memory of the buffers during reset or switching
    ////////////////////////////////////////////////////////////////////////
    depthbufferinitiator dbi (.clock(clock), .init(init), .pixeldata(bufferinitdata),
        .pixeladdr(bufferinitaddr), .init_done(init_done));

    ////////////////////////////////////////////////////////////////////////
    // The module which process data from the displayer and depthcomparator and
    // assigns the proper signals to be passed to the zbt rams depending on
    // what buffer are we reading from or writing to.
    ////////////////////////////////////////////////////////////////////////
    colordata_to_buffer ctb (.clock(clock), .reset(reset), .buffer_sel(buffer_sel),
        .writeaddr(writeaddr), .colordata(colordata), .we(we),
        .ram0_wdata(ram0_wdata), .ram0_addr(ram0_addr), .ram0_we(ram0_we),
        .ram1_wdata(ram1_wdata), .ram1_addr(ram1_addr), .ram1_we(ram1_we));
endmodule
```

```
/*********************************************************************************
Written by: Joshua Jen Monzon
6.111 Final Project: Digital Stereoscope

This module interfaces the depth comparator module with the ZBT SRAMS. Takes in the
colordata and the pixel address to write these data. It also takes in signals from the
buffer select module informing what buffer is currently being written. Also takes in
write enable signal from the depth comparator which tells whether the current color data
should be written or not to the ZBT. This module takes care of the routing of the signals
to the appropriate ZBT.
*********************************************************************************/
module colordata_to_buffer(clock, reset, buffer_sel, writeaddr, colordata, we,
    ram0_wdata, ram0_addr, ram0_we, ram1_wdata, ram1_addr, ram1_we);

    input clock, reset, buffer_sel, we;   // control signals
    input [18:0] writeaddr;               // address to write data
    input [23:0] colordata;               // color information

    output [35:0] ram0_wdata, ram1_wdata; // data written to ZBTs
    output [18:0] ram0_addr, ram1_addr;   // address to write data in the ZBTs
    output ram0_we, ram1_we;              // write enable control signals to the ZBTs

    ////////////////////////////////////////////////////////////
    // Parameters which correspond to the data which is passed to
    // the ZBT when the colordata should not be written.
    ////////////////////////////////////////////////////////////
    parameter datahighimpedance = 0;
    parameter addrhighimpedance = 0;
    parameter wehighimpedance = 0;

    ////////////////////////////////////////
    // Assigns values to be written to ZBT 0
    ////////////////////////////////////////
    assign ram0_wdata = ((!reset) && (!buffer_sel) && we) ? {12'h000, colordata} :
                         datahighimpedance;
    assign ram0_addr = ((!reset) && (!buffer_sel)) ? writeaddr : addrhighimpedance;
    assign ram0_we= ((!reset) && (!buffer_sel)) ? we : 0;

    ////////////////////////////////////////
    // Assigns values to be written to ZBT 1
    ////////////////////////////////////////
    assign ram1_wdata = ((!reset) && buffer_sel && we) ?    {12'h000, colordata} :
                         datahighimpedance;
    assign ram1_addr = ((!reset) && buffer_sel) ? writeaddr : addrhighimpedance;
    assign ram1_we = ((!reset) && buffer_sel) ? we : 0;

endmodule
```

```
/*********************************************************************************
Written by: Joshua Jen Monzon
6.111 Final Project: Digital Stereoscope

This module prepares the ZBT SRAMS as well as the internal depth buffer (BRAM) for
incoming data. Basically, this clears all the memory used by passing in a default depth
of MAXDEPTH to all the addresses of the BRAM and default color BLACK to all the addresses
of the ZBT SRAM which is being written to.
*********************************************************************************/
module depthbufferinitiator (clock, init, pixeldata, pixeladdr, init_done);

    input clock;                    // global signal
    input init;                     // signal from the depth comparator informing
                                    // this module to start initiating

    output [34:0] pixeldata;        // default pixel data
    output [18:0] pixeladdr;        // address to write the data
    output init_done;               // signal informing the depth comparator that
                                    // initiation is done
    reg [18:0] pixeladdr;
    reg init_done;

    parameter black = 24'h0;
    parameter maxdepth = 11'h7ff;   // in decimal = 4095 – max size of our world
    parameter maxaddress = 524287;  // in decimal = 2^19 – this number could actually be as low
                                    // as 204800 (faster initiation speeds) since our viewpoints
                                    // are only comprised of 204800 pixels but was set to the
                                    // maximum ZBT address in case displayer glitches and reads
                                    // from an unused address – this way, the monitor would show
                                    // the color black.
    parameter minaddress = 0;

    ////////////////////////////////////////////////////////////////////
    // Combinational logic hardwiring the pixel data to black and maxdepth
    ////////////////////////////////////////////////////////////////////
    assign pixeldata = {black, maxdepth};

    always @(posedge clock) begin
        if (init) begin                     // receives init signal, therefore start initiation
            pixeladdr <= minaddress;        // sets pixel address to minimum address
            init_done <= 0;                 // NOT done initiating
        end

        else if (!init_done && (pixeladdr < maxaddress)) begin      // not max address
            pixeladdr <= pixeladdr + 1;                             // loops to all addresses
        end

        else if (!init_done && (pixeladdr >= maxaddress)) begin     // reach max address
            pixeladdr <= maxaddress;                                // stops increment
            init_done <= 1;                                         // DONE initiating
        end
    end
endmodule
```

```
/*********************************************************************************
Written by: Joshua Jen Monzon
6.111 Final Project: Digital Stereoscope

This module comprises displays pixel information stored from the ZBT SRAMs to the
monitor. It takes in hcount and vcount (the position of the pixel being displayed in the
screen) and computes the appropriate address associated with these two positions. It then
interfaces with the ZBT by passing this address and collecting the appropriate data
associated with this address and displays it to the screen. Communicates with the
buffer selector module informing it when it is done displaying one screenful of data.
*********************************************************************************/
module displayer (vclock, reset, hcount, vcount, fromzbt_data, tozbt_addr, pixel, donedisplaying);

    input vclock;                // 65MHz clock
    input reset;                 // global reset
    input [10:0] hcount;         // horizontal index of current pixel (0..1023)
    input [9:0] vcount;          // vertical index of current pixel (0..767)
    input [35:0] fromzbt_data;   // pixel data from the ZBT

    output [23:0] pixel;         // pixel color to be displayed
    output [18:0] tozbt_addr;    // ZBT address to read data from
    output donedisplaying;       // pulse indicating whether screen is finished displaying

    wire blackscreen;
    wire insidescreen1;
    wire insidescreen2;
    wire [10:0] xoffset;
    wire [9:0] yoffset;
    wire [10:0] upperleft1x, upperlefty, upperleft2x;

    parameter halfgap = 8;           // one half of the gap between 2 eye viewpoints
    parameter xscreenmaxmidpt = 512; // x value of midpoint of the screen
    parameter yscreenmaxmidpt = 384; // y value of midpoint of the screen
    parameter xscreenmax = 1024;     // max width of the screen
    parameter yscreenmax = 768;      // max height of the screen
    parameter delay = 5;             // total pipeline latency

    reg [18:0] offsetaddress, tozbt_addr;
    reg [23:0] pixel;
    reg donedisplaying;
    reg temp;

    assign upperleft1x = 185;        // upper left x coord of 1st eye viewpoint
    assign upperlefty = yscreenmaxmidpt - 160 - 1;
                                     // upperleft y coord viewpoint (same for both eyes)
    assign upperleft2x = 521;        // upper left x coord of 2nd eye viewpoint.

    ////////////////////////////////////////////////////////////////////////////
    // Checks if hcount and vcount is within any of the two eye viewpoints in the
    // screen. If it is these two signals will be high depending on which screen is
    // hcount and vcount is.
    ////////////////////////////////////////////////////////////////////////////
    assign insidescreen1 =
        ((hcount >= upperleft1x) && (hcount < (upperleft1x + 320)) &&
        (vcount >= upperlefty) && (vcount < (upperlefty + 320))) ? 1 : 0;
    assign insidescreen2 =
        ((hcount >= upperleft2x) && (hcount < (upperleft2x + 320)) &&
        (vcount >= upperlefty) && (vcount < (upperlefty + 320))) ? 1: 0;

    ////////////////////////////////////////////////////////////////////////////
    // If hcount and vcount is not in the screen or if we are resetting the system,
    // then the corresponding pixel displayed must be black.
    ////////////////////////////////////////////////////////////////////////////
    assign blackscreen = reset? 1 : (((~insidescreen1 && ~insidescreen2)) ? 1 : 0);


    ////////////////////////////////////////////////////////////////////////////
    // These two values are defaulted to 0 and 320 respectively if blackscreen.
    // Nothing special about these two numbers - can be anything. However, if
    // we are in screen 1, then we compute a certain offset to the hcount and vcount to
    // normalize our screen to zero coordinates with 0,0 represented by upperleft1x,
```

```
    // upperlefty.
    ////////////////////////////////////////////////////////////////////////////
    assign xoffset = insidescreen1 ? (hcount - upperleft1x) : insidescreen2 ?
        (hcount - upperleft1x - halfgap - halfgap) : 0;
    assign yoffset = ~blackscreen ? (vcount - upperlefty) : 320;

    always @ (posedge vclock) begin
        ////////////////////////////////////////////////////////////////////
        // 3 Stage Pipelined Display (5 clock latency, 1/clock throughput)
        ////////////////////////////////////////////////////////////////////

        // 1) Computes the offset address (640*yoffset + xoffset)
        //    to avoid glitching let this compute for one clock
        offsetaddress <= ((yoffset<<7) + (yoffset<<9) + xoffset);

        //  2)Assigns the calculated offset value if we are not blackscreen
        //    and if we are   blackscreen, just read from a remote address
        //    to avoid glitching again we latch this data for one clock
        tozbt_addr <= ((~blackscreen) ? offsetaddress : 204800);

        // 3) Assigns pixel to be displayed to the data we got from the
        //    ZBT using the address we passed in - comes in 2 clocks later
        //    and we latch this for another cycle to stabilize
        pixel <= fromzbt_data[23:0];

        ////////////////////////////////////////////////////////////////////////
        // Done Displaying Pulse, fired each time we finish displaying a screen
        // Delayed by 5 clocks to compensate for our latency
        ////////////////////////////////////////////////////////////////////////

        temp <= (hcount == 1023+delay) & (vcount == 767);
        donedisplaying <= (hcount == 1023+delay) & (vcount == 767) & ~temp;
    end
endmodule
```

```
/*************************************************************************************
Written by: Joshua Jen Monzon
6.111 Final Project: Digital Stereoscope

Controls the switching of the ZBT SRAM buffers. Switches whenever all triangles are
finished rendering AND the screen is finished displaying data. This is because we do
not want to switch while we are halfway displaying the screen - smooth transitioning.
*************************************************************************************/
module bufferselector(clock, reset, donedisplaying, triangledone, switchbuffers, buffer_sel);

    input clock, reset;       // global signals
    input donedisplaying;     // screen finished displaying
    input triangledone;       // triangles done rendering

    output switchbuffers;     // pulse indicating buffer is switching
    output buffer_sel;        // indicates which buffer should be WRITTEN

    wire switch;

    reg tdreg, buffer_sel, state, switchbuffers;

    parameter s_writeto0 = 0;     // state 0: writing to buffer 0, reading from buffer 1
    parameter s_writeto1 = 1;     // state 1: writing to buffer 1, reading from buffer 0


    // logic which decides if we are switching. This is true only if buffer is done refreshing
    // the triangles and done displaying one screen.
    assign switch = ((tdreg || triangledone) && donedisplaying) ? 1 : 0;


    always @ (posedge clock) begin
        ////////////////////////////////////
        // Pressing reset, sets up our machine
        // and changes buffer_sel = 0
        ////////////////////////////////////
        if (reset)  begin
            buffer_sel <= 0;          // buffer to be refreshed first is buffer 0
            state <= s_writeto0;      // reset state
            tdreg <= 0;               // clear triangledone register
            switchbuffers <= 0;       // not switching
        end

        else begin
            switchbuffers <= switch;  // latches switch data to avoid glitches

            // when buffer is switched, clear register, otherwise listen for triangledone pulse
            tdreg <= switchbuffers ? 0 : triangledone ? 1 : tdreg;

            ////////////////////////////////////////////
            // 2 state FSM controlling buffer selection
            ////////////////////////////////////////////
            case (state)
                s_writeto0: begin     // Writing to buffer 0
                    buffer_sel <= switch ? 1 : 0;
                    state <= switch ? s_writeto1: s_writeto0;
                end

                s_writeto1: begin     // Writing to buffer 1
                    buffer_sel <= switch ? 0 : 1;
                    state <= switch ? s_writeto0: s_writeto1;
                end
            endcase
        end
    end
endmodule
```

```
/*********************************************************************************
Written by: Joshua Jen Monzon
6.111 Final Project: Digital Stereoscope

Creates the internal depth buffer memory used to compare depths of pixels. Concatenates
two BRAMS together, since XILINX won't allow you to synthesize a single BRAM with the
required dimensions. Allows both read and write capabilities.

Dimensions: 12 bits wide, 204800 address deep
*********************************************************************************/
module depthbuffer (addr, clock, din, dout, we);
    input clock;          // global clock
    input we;             // write enable signal
    input [17:0] addr;    // address to read or write data
    input [10:0] din;     // input data for writing

    output [10:0] dout;   // output data

    depthbuffer3x18 db1 (addr, clock, din[10:8], dout[10:8], we);   // 3 highest bits memory
    depthbuffer8x18 db2 (addr, clock, din[7:0], dout[7:0], we);     // 8 lowest bits memory
endmodule
```

```
/*********************************************************************************
Written by: Joshua Jen Monzon
6.111 Final Project: Digital Stereoscope

Contains an internal BRAM memory which stores depth information of each pixel. This module
ensures that the pixels overlap each other properly. To do this, it checks if      the memory
contains pixel information associated with a NEW incoming pixel address. If it does, it
compares the new pixel's depth with the previously stored pixel's depth and stores the pixel
which is less deep -> deeper pixels are behind, thus we shouldn't see them. This module
also passes along the new color information to the colordatatobuffer module (for further
processing)and assigns a write enable signal if this new color information should be
displayed (meaning it's the least deep pixel in a certain location). Also controls the
initiation of the BRAM and ZBTs by passing init signals to the depth buffer initiator module.
*********************************************************************************/
module depthcomparator (clock, reset, shading_ready, shading_done, init_done, inputdata,
    pixeladdr, writeaddr, colordata, we, init, mem_ready);

    input clock;              // 65Mhz clock
    input reset;              // High if reset button is pre
    input [34:0] inputdata;   // 36 bit input data from either the shading module
                              // or the buffer initiator module {24 bit color, 12 bit depth}
    input [18:0] pixeladdr;   // Address where the pixel is stored = y*640 + x
    input shading_ready;      // Signal from the shading module indicating when a triangle
                              // needs to be rendered
    input shading_done;       // Signal from the shading module indicating rendering is done
    input init_done;          // Control signal from the buffer initiator module indicating that
                              // buffer initiation is done

    output [18:0] writeaddr;  // Address where the pixel is stored to the buffer = y*640 + x
    output [23:0] colordata;  // 24 bit color from the pixel data
    output we;                // Determines whether we should write to the memory
    output mem_ready;         // Informs other modules when they can start processing
    output init;              // Informs other bufferinitiator module to start initiating

    wire [10:0] newdepthdata; // Depthvalue being passed in from the shading module
    wire [10:0] memdepthdata; // Depthvalue stored in the memory

    reg [2:0] state;
    reg [10:0] writedata;
    reg mem_ready;
    reg init;
    reg shading_ready1, shading_ready2;
    reg we;

    parameter s_init = 0;     // initialization state - clears all memories
    parameter s_idle = 1;     // idle state - just waiting for incoming processed triangles
    parameter s_read = 2;     // read state - reads pixel data
    parameter s_compare = 3;  // compare state - compares memdepthdata to newdepthdata
    parameter s_write = 4;    // write state - writes pixel data

    assign newdepthdata = inputdata[10:0];
    assign colordata = inputdata[34:11];
    assign writeaddr = pixeladdr;

    ////////////////////////
    // Depth Buffer Memory
    ////////////////////////
    depthbuffer db (pixeladdr[17:0], clock, writedata , memdepthdata, we);


    //////////////////
    // 5 state FSM
    //////////////////
    always @ (posedge clock) begin
        ////////////////////////////////////////////////////////////////
        // Must conform to the timing specifications of the shading module
        // which has a throughput of 1/3 clock, so must latch shading_ready
        // signal and delay it twice to time the switching from idle state
        // to read state properly.
        ////////////////////////////////////////////////////////////////
```

```
shading_ready1 <= shading_ready;
shading_ready2 <= shading_ready1;

////////////////////////////////////////////////////////////////
// At reset we begin initializing our depthmemory to have depth
// values equal to the highest depth so each of them could be
// overwritten by incoming depth values from the shading module.
// During this initialization state, we set the ready signal to be
// low since we don't want our other modules to do processing while
// we are instantiating our depth memory. We send an init signal to
// the initiator module for it to start sending init values to our
// depthbuffer. We also clear the ZBT by passing in black values
// to each of its addresses.
////////////////////////////////////////////////////////////////
if (reset) begin
    mem_ready <= 0;
    we <= 1;
    state <= s_init;
    writedata <= newdepthdata;
    init <= 1;
end

else begin
    case (state)
        ////////////////////////////////////////
        // We are at the init state
        ////////////////////////////////////////
        s_init: begin
            init <= 0;
            writedata <= newdepthdata;

            // Checks if we not are done instantiating: if so
            // we continue instantiating.
            if (!init_done || init) begin
                mem_ready <= 0;
                we <= 1;
                state <= s_init;
            end

            // We are done instantiating so we fire the memory ready signal
            // and stop writing to memory. We then transition to read state.
            else if (init_done) begin
                mem_ready <= 1;
                we <= 0;
                state <= s_idle;
            end
        end

        ////////////////////////////////////////////////////////////////
        // We are at the idle state: our comparator does nothing until
        // the shading module is ready to send us values
        ////////////////////////////////////////////////////////////////

        s_idle: begin
            writedata <= writedata;
            init <= 0;
            mem_ready <= 1;
            we <= 0;
            state <= shading_ready2 ? s_read : s_idle;
            // if shading module has triangles to send, then
            // we must switch to the read state. note that shading
            // ready is delayed by 2 clock cycles
        end

        ////////////////////////////////////////////////////////////////
        // We are at the read state: here we are reading values being
        // sent from the shader module and accessing old pixel values
        // stored in the memory corresponding to the address sent
        ////////////////////////////////////////////////////////////////
        s_read: begin
            init <= 0;
```

```
                mem_ready <= 1;
                writedata <= writedata;
                we <= 0;       // we don't want to write in the next state
                if (!shading_done) begin
                    state <= s_compare;
                end
                else begin     // if we receive a shading done, system goes idle
                    state <= s_idle;
                end
            end

            ////////////////////////////////////////////////////////////////
            // We are at the compare state: here we are comparing depth
            // values from the shading module vs our the depth values stored
            // in our memory. Whichever is less deep, we retain.
            ////////////////////////////////////////////////////////////////
            s_compare: begin
                init <= 0;
                mem_ready <= 1;

                if (!shading_done) begin
                    we <= (newdepthdata <= memdepthdata)? 1 : 0;
                    writedata <= (newdepthdata <= memdepthdata)? newdepthdata : memdepthdata;
                    state <= s_write;
                end

                else begin
                    state <= s_idle;
                    we <= 0;
                    writedata <= writedata;
                end
            end

            ////////////////////////////////////////////////////////////////
            // We are at the write state: here we are writing values to the
            // BRAM based on which depth is less deeper
            ////////////////////////////////////////////////////////////////
            s_write: begin
                init <= 0;
                mem_ready <= 1;
                writedata <= writedata;
                we <= 0;
                if (!shading_done) begin  // means more pixels are coming, read again
                    state <= s_read;
                end
                else begin                // no more pixels are coming, go idle
                    state <= s_idle;
                end
            end
        endcase
    end
  end
endmodule
```

```
/*************************************
Created By: Zhou, Hao (Steven)           *
6.111 Final Project: Digital Stereoscope *
*************************************/

/*************************************************************************************************
The projector shader module serves as a black box that exchanges data with the perspective
controller and buffer interface modules.  It includes in it a pixel extractor module, color LUT
module, and shading module.  It also acted as a buffer to correctly finalize the output to the
other buffers
*************************************************************************************************/
module Projector_Shader (clk, clk_div1, clk_div2, slow_clk, reset, mem_ready, tri_ready, e1c1,
          e1c2, e1c3, e2c1, e2c2, e2c3, color, last_triangle, shading_ready, shading_done,
          pixel_address, pixel_data, want_new_triangle, last_triangle_shading_done);


    input clk;                        // Clock for sequential logic
    input clk_div1, clk_div2;         // Clock for the two dividers
    input slow_clk;                   // Signal that is a high every three clock cycles
    input reset;                      // Global reset
    input mem_ready;                  // Held high when the ZBTs are ready to be written to
    input tri_ready;                  // High for three clock cycles when new triangle is ready
    input last_triangle;              // Held high when rendering the last triangle
    input [32:0] e1c1, e1c2, e1c3;    // Pixel locations of the three corners for the left eye
    input [32:0] e2c1, e2c2, e2c3;    // Pixel locations of the three corners for the second eye
                                      // [32:23]: X loc; [22:13]: Y loc; [12:0]: Distance
    input [7:0] color;                // Eight bit color to represent the color of triangle


    output shading_ready;                 // High for 3 clock when ready to pass data to buffer
    output shading_done;                  // High for 3 clock when triangle is done rendering
    output last_triangle_shading_done;    // High for 3 clock when last triangle is done
    output want_new_triangle;             // Request for location of a new triangle
    output [17:0] pixel_address;          // Address of the pixel, top 9 bits are x location,
                                          // bottom 9 bits are y location
    output [34:0] pixel_data;             // Data of the pixel, top 24 bit the RGB value,
                                          // bottom 11 is the distance

    wire [19:0] pixel_location, pixel_location_delayed;
    wire tocolor, tocolor_delayed;
    wire pixel_ready, pixel_ready_delayed;
    wire tri_done, tri_done_delayed;
    wire [10:0] distance, distance_delayed;
    wire [23:0] rgb;

    // wire and reg declarations are used to latch the data
    wire shading_ready_temp, shading_done_temp;
    reg shading_ready, shading_done;
    wire last_triangle_shading_done_temp, want_new_triangle_temp;
    reg last_triangle_shading_done, want_new_triangle;
    wire [34:0] pixel_data_temp;
    reg [34:0] pixel_data;
    wire [17:0] pixel_address_temp;
    reg [17:0] pixel_address;

    // last triangle shading done is high when shading is done and the triangle is the last
    assign last_triangle_shading_done_temp = reset? 0 : ((last_triangle && shading_done) ? 1 : 0);

    // want a new triangle if shading is done with previous triangle and it was not the last
    // triangle
    assign want_new_triangle_temp = ((~last_triangle && shading_done) ? 1 : 0);

    // Pixel Extractor Module
    PixelExtractor pe (
        // Inputs:
        .clk(clk), .clk_div1(clk_div1), .clk_div2(clk_div2), .slow_clk(slow_clk),
        .reset(reset), .e1c1(e1c1), .e1c2(e1c2), .e1c3(e1c3), .e2c1(e2c1), .e2c2(e2c2),

        .e2c3(e2c3), .mem_ready(mem_ready), .tri_ready(tri_ready), .last_triangle(last_triangle),
```

```
        // Outputs:
        .pixel_location(pixel_location), .tocolor(tocolor), .pixel_ready(pixel_ready),
        .distance(distance), .tri_done(tri_done));

    // Pixel Extractor Delay module, which delay certain output so pixel location corresponds to
    // their distance values.
    PixelExtractorDelay ped (
        // Inputs:
        .clk(clk), .pixel_location(pixel_location), .tocolor(tocolor),
        .pixel_ready(pixel_ready), .tri_done(tri_done), .distance(distance),

        // Outputs:
        .pixel_location_delayed(pixel_location_delayed), .tocolor_delayed(tocolor_delayed),
        .pixel_ready_delayed(pixel_ready_delayed), .tri_done_delayed(tri_done_delayed),
        .distance_delayed(distance_delayed));

    // Color LUT, which converts an eight bit color to a 24 bit color
    ColorLUT LUT (.color(color), .rgb(rgb));

    // Shading module, which takes the delayed and inputs and outputs the buffer address which
    // corresponds to the pixel location, and the data that corresponds with the pixel
    Shading shad (
        // Inputs:
        .clk(clk), .slow_clk(slow_clk), .pixel_ready(pixel_ready_delayed),
        .tri_done(tri_done_delayed), .tocolor(tocolor_delayed),
        .pixel_location(pixel_location_delayed), .distance(distance_delayed), .rgb(rgb),

        // Outputs
        .shading_ready(shading_ready_temp), .pixel_address(pixel_address_temp),
        .pixel_data(pixel_data_temp), .shading_done(shading_done_temp));

    // Latch the output data
    always @(posedge clk) begin
        if(slow_clk) begin
            shading_ready <= shading_ready_temp;
            shading_done <= shading_done_temp;
            last_triangle_shading_done <= last_triangle_shading_done_temp;
            want_new_triangle <= want_new_triangle_temp;
            pixel_address <= pixel_address_temp;
            pixel_data <= pixel_data_temp;
        end
    end

endmodule
```

```
/*************************************
Created By: Zhou, Hao (Steven)          *
6.111 Final Project: Digital Stereoscope *
*************************************/

/*************************************************************************************************
The Pixel Extractor module takes as input the 3 corner locations and their distances to the
respective eyes of the triangles, and loops through suitable pixels that are in or near the
triangles.  For each of these pixels, it determines the distance from the corresponding eye to
the pixel, and whether or not that pixel needs to be colored, which is determined by if the pixel
is inside a triangle.
*************************************************************************************************/

module PixelExtractor(clk, clk_div1, clk_div2, slow_clk, reset, e1c1, e1c2, e1c3, e2c1, e2c2,
       e2c3, mem_ready, tri_ready, last_triangle, pixel_location, tocolor, pixel_ready, distance,
       tri_done);

    input clk;                        // Clock for sequential logic
    input clk_div1, clk_div2          // Clock for the two dividers
    input slow_clk;                   // Signal that is a high every three clock cycles
    input reset;                      // Global reset
    input mem_ready;                  // Held high when the ZBTs are ready to be written to
    input tri_ready;                  // High for three clock cycles when new triangle is ready
    input last_triangle;              // Held high when rendering the last triangle
    input [32:0] e1c1, e1c2, e1c3;    // Pixel locations of the three corners for the left eye
    input [32:0] e2c1, e2c2, e2c3;    // Pixel locations of the three corners for the second eye
                                      // [32:23]: X loc; [22:13]: Y loc; [12:0]: Distance

    output tocolor;                   // High when the pixel location is inside the triangle
    output tri_done;                  // High for 3 clocks when done with rendering triangle
    output pixel_ready;               // High for 3 clocks when triangle pixel data is ready
    output [19:0] pixel_location;     // {x, y} location for the pixel
    output [10:0] distance;           // Distance of the pixel from the eye

    // state parameters
    parameter S_reset  = 3'b000;
    parameter S_wait   = 3'b001;
    parameter S_scan_eye1 = 3'b010;
    parameter S_scan_eye2 = 3'b011;
    parameter S_buffer_switch = 3'b100;
    parameter S_wait_for_buffer_switch = 3'b101;

    // Wires to store twice the area of the entire triangle, and the area of the three triangles
    // that is produced using the pixel point and the three corners of the triangle for each eye
    wire [17:0] e1Area_full, e1Area_sub1, e1Area_sub2, e1Area_sub3, e1Area_subTotal;
    wire [17:0] e2Area_full, e2Area_sub1, e2Area_sub2, e2Area_sub3, e2Area_subTotal;

    // X and Y max and min values for the triangle for each eye
    wire [9:0] e1xmax, e1ymax, e1xmin, e1ymin;
    wire [9:0] e2xmax, e2ymax, e2xmin, e2ymin;

    reg [9:0] e1x, e1y, e2x, e2y;        // Current x and y location of the eye
    reg [2:0] state;                     // State of the FSM
    reg tri_done, tocolor, pixel_ready;
    reg [19:0] pixel_location;
    wire [10:0] e1distance, e2distance;  // Distance of the pixel from the two eyes
    reg [10:0] distance;                 // Distance for that pixel location
    wire inTriangle1, inTriangle2;       // Whether a pixel location is in the triangles

    // Determines twice the area of the entire triangle for eye 1.
    TriangleArea triangle1_full
        (.x1(e1c1[32:23]), .y1(e1c1[22:13]), .x2(e1c2[32:23]),
         .y2(e1c2[22:13]), .x3(e1c3[32:23]), .y3(e1c3[22:13]), .area(e1Area_full));

    // Determines twice the area of the three sub triangles for eye 1.
    TriangleArea triangle1_subl
        (.x1(e1c1[32:23]), .y1(e1c1[22:13]), .x2(e1c2[32:23]),
         .y2(e1c2[22:13]), .x3(e1x), .y3(e1y), .area(e1Area_sub1));
    TriangleArea triangle1_sub2
        (.x1(e1c1[32:23]), .y1(e1c1[22:13]), .x2(e1x),
         .y2(e1y), .x3(e1c3[32:23]), .y3(e1c3[22:13]), .area(e1Area_sub2));
```

```
TriangleArea triangle1_sub3
    (.x1(e1x), .y1(e1y), .x2(e1c2[32:23]),
     .y2(e1c2[22:13]), .x3(e1c3[32:23]), .y3(e1c3[22:13]), .area(e1Area_sub3));

// Determines twice the area of the three sub triangles for eye 2.
TriangleArea triangle2_full
    (.x1(e2c1[32:23]), .y1(e2c1[22:13]), .x2(e2c2[32:23]),
     .y2(e2c2[22:13]), .x3(e2c3[32:23]), .y3(e2c3[22:13]), .area(e2Area_full));
// Determines twice the area of the three sub triangles for eye 2.
TriangleArea triangle2_subl
    (.x1(e2c1[32:23]), .y1(e2c1[22:13]), .x2(e2c2[32:23]),
     .y2(e2c2[22:13]), .x3(e2x), .y3(e2y), .area(e2Area_sub1));
TriangleArea triangle2_sub2
    (.x1(e2c1[32:23]), .y1(e2c1[22:13]), .x2(e2x),
     .y2(e2y), .x3(e2c3[32:23]), .y3(e2c3[22:13]), .area(e2Area_sub2));
TriangleArea triangle2_sub3
    (.x1(e2x), .y1(e2y), .x2(e2c2[32:23]),
     .y2(e2c2[22:13]), .x3(e2c3[32:23]), .y3(e2c3[22:13]), .area(e2Area_sub3));

// Determines the distance from eye 1 to a particular pixel
PixelDistance pd1 (clk_div1, slow_clk, e1Area_sub1[17:5], e1Area_sub2[17:5],
    e1Area_sub3[17:5], e1Area_full[17:5], e1c1[12:0], e1c2[12:0], e1c3[12:0], e1distance);

// Determines the distance from eye 2 to a particular pixel
PixelDistance pd2 (clk_div2, slow_clk, e2Area_sub1[17:5], e2Area_sub2[17:5],
    e2Area_sub3[17:5], e2Area_full[17:5], e2c1[12:0], e2c2[12:0], e2c3[12:0], e2distance);

// Determines the sum of twice the area of each of the sub triangles for eye 1 and eye 2
assign e1Area_subTotal = e1Area_sub1 + e1Area_sub2 + e1Area_sub3;
assign e2Area_subTotal = e2Area_sub1 + e2Area_sub2 + e2Area_sub3;

// A pixel is inside a triangle if the sum of the area of the three sub triangles is equal to
// the total area
assign inTriangle1 = (e1Area_subTotal == e1Area_full) ? 1 : 0;
assign inTriangle2 = (e2Area_subTotal == e2Area_full) ? 1 : 0;

// Determines the x and y min and max locations for each eye
assign e1xmax = (e1c1[32:23]>=e1c2[32:23]) ?
                   ((e1c1[32:23]>=e1c3[32:23]) ? e1c1[32:23] : e1c3[32:23]):
                   ((e1c2[32:23]>=e1c3[32:23]) ? e1c2[32:23] : e1c3[32:23]);
assign e1ymax = (e1c1[22:13]>=e1c2[22:13]) ?
                   ((e1c1[22:13]>=e1c3[22:13]) ? e1c1[22:13] : e1c3[22:13]):
                   ((e1c2[22:13]>=e1c3[22:13]) ? e1c2[22:13] : e1c3[22:13]);
assign e1xmin = (e1c1[32:23]<=e1c2[32:23]) ?
                   ((e1c1[32:23]<=e1c3[32:23]) ? e1c1[32:23] : e1c3[32:23]):
                   ((e1c2[32:23]<=e1c3[32:23]) ? e1c2[32:23] : e1c3[32:23]);
assign e1ymin = (e1c1[22:13]<=e1c2[22:13]) ?
                   ((e1c1[22:13]<=e1c3[22:13]) ? e1c1[22:13] : e1c3[22:13]):
                   ((e1c2[22:13]<=e1c3[22:13]) ? e1c2[22:13] : e1c3[22:13]);
assign e2xmax = (e2c1[32:23]>=e2c2[32:23]) ?
                   ((e2c1[32:23]>=e2c3[32:23]) ? e2c1[32:23] : e2c3[32:23]):
                   ((e2c2[32:23]>=e2c3[32:23]) ? e2c2[32:23] : e2c3[32:23]);
assign e2ymax = (e2c1[22:13]>=e2c2[22:13]) ?
                   ((e2c1[22:13]>=e2c3[22:13]) ? e2c1[22:13] : e2c3[22:13]):
                   ((e2c2[22:13]>=e2c3[22:13]) ? e2c2[22:13] : e2c3[22:13]);
assign e2xmin = (e2c1[32:23]<=e2c2[32:23]) ?
                   ((e2c1[32:23]<=e2c3[32:23]) ? e2c1[32:23] : e2c3[32:23]):
                   ((e2c2[32:23]<=e2c3[32:23]) ? e2c2[32:23] : e2c3[32:23]);
assign e2ymin = (e2c1[22:13]<=e2c2[22:13]) ?
                   ((e2c1[22:13]<=e2c3[22:13]) ? e2c1[22:13] : e2c3[22:13]):
                   ((e2c2[22:13]<=e2c3[22:13]) ? e2c2[22:13] : e2c3[22:13]);

// Determines when memory switched from not ready to ready, meaning a buffer has just been
// cleared and is now ready for its first triangle
reg old_mem_ready;
always @(posedge clk) begin
    if (reset) begin
        old_mem_ready<=0;
    end
    else if (slow_clk) begin
        old_mem_ready <= mem_ready;
```

```
    end
end

// Finite State Machine, which loops through all the pixels in the triangle for both eyes,
// and determines the correct output for each pixel.
always @(posedge clk) begin
    // On reset, set the value to default
    if (reset) begin
        tri_done <= 0;
        tocolor <= 0;
        state <= S_reset;
        e1x<=0;
        e1y<=0;
        e2x<=0;
        e2y<=0;
        pixel_location <= 0;
        pixel_ready <= 0;
        distance <= 0;
    end

    // Compute FSM logic every thee clock cycles
    else if (slow_clk) begin
        case (state)
            // In reset state, the output values are held at default, and the FSM waits
            // for a memory ready signal before transition to the wait state
            S_reset: begin
                tri_done <= mem_ready ? 1 : 0;
                state <= mem_ready ? S_wait : S_reset;
                tocolor <= 0;
                e1x<=0;
                e1y<=0;
                e2x<=0;
                e2y<=0;
                pixel_location <= 0;
                pixel_ready <= 0;
                distance <= 0;
            end

            // In the wait state, the FSM waits for a triangle ready signal, which means a
            // triangle is ready to be rendered.  It sets the pixel location of both eyes
            // to their minimum values, and proceeds to loop through the pixels inside the
            // first triangle
            S_wait: begin
                state <= tri_ready ? S_scan_eye1 : S_wait;
                pixel_ready <= tri_ready ? 1 : 0;
                tocolor <= 0;
                tri_done <= 0;
                e1x <= e1xmin;
                e1y <= e1ymin;
                e2x <= e2xmin;
                e2y <= e2ymin;
                pixel_location <= 0;
                distance <= 0;
            end

            // In scan eye 1 state, the FSM loops through all the pixels enclosed by the
            // rectangle that circumscribe the triangle for eye 1, and for each pixel, it
            // determines the pixel's distance from the eye, and whether or not it should
            // color the pixel, which depends on whether or not the pixel is actually inside
            // the triangle.
            S_scan_eye1: begin
                // Once the maximum pixel location has been reached, the FSM jumps to scan
                // pixels in the eye 2 triangle
                if (e1x==e1xmax && e1y==e1ymax) begin
                    e1x <= 0;
                    e1y <= 0;
                    state <= S_scan_eye2;
                    tri_done <= 0;
                    pixel_location <= {e1x, e1y};
                    tocolor <= inTriangle1;
                    distance <= e1distance;
```

```
            pixel_ready <= 0;
        end

        // If the x value of the pixel reaches the max, the x location resets to zero,
        // and the y location increments by one.
        else if (e1x==e1xmax) begin
            e1x <= e1xmin;
            e1y <= e1y+1;
            state <= S_scan_eye1;
            tri_done <= 0;
            pixel_location <= {e1x, e1y};
            tocolor <= inTriangle1;
            distance <= e1distance;
            pixel_ready <= 0;
        end

        // If neither of the above conditions hold, increment the x location of the
        // pixel by 1.
        else begin
            e1y <= e1y;
            e1x <= e1x+1;
            state <= S_scan_eye1;
            tri_done <= 0;
            pixel_location <= {e1x, e1y};
            tocolor <= inTriangle1;
            distance <= e1distance;
            pixel_ready <= 0;
        end

    end

    // Much the same logic as scan eye 1 state, except this state loops through all
    // the pixels inside a triangle for eye 2.  Except in this case, all the x
    // locations of the triangle is incremented by 320 since it's on the right half of
    // the screen
    S_scan_eye2: begin
        // If the max x and y location has been reached in this case, it means that
        // all of the pixels in both eyes for this particular triangle has been
        // rendered, thus it will check if it is the last triangle, if not, it will go
        // back to its wait state to wait for the next triangle ready signal; if so,
        // it will jump to the buffer switch state.
        if (e2x==e2xmax && e2y==e2ymax) begin
            e2x <= 0;
            e2y <= 0;
            state <= last_triangle ? S_buffer_switch: : S_wait;
            tri_done <= 0;
            pixel_location <= {e2x+320, e2y};
            tocolor <= inTriangle2;
            distance <= e2distance;
            pixel_ready <= 0;
        end

        else if (e2x==e2xmax) begin
            e2x <= e2xmin;
            e2y <= e2y+1;
            state <= S_scan_eye2;
            tri_done <= 0;
            pixel_location <= {e2x+320, e2y};
            tocolor <= inTriangle2;
            distance <= e2distance;
            pixel_ready <= 0;
        end

        else begin
            e2y <= e2y;
            e2x <= e2x+1;
            state <= S_scan_eye2;
            tri_done <= 0;
            pixel_location <= {e2x+320, e2y};
            tocolor <= inTriangle2;
            distance <= e2distance;
```

```
                    pixel_ready <= 0;
                end
            end

            // This state is entered when the triangle that was just rendered was the last
            // triangle.  This state will simply fire a triangle done signal, then immediately
            // jump to the wait for buffer switch state
            S_buffer_switch: begin
                tri_done <= 1;
                state <= S_wait_for_buffer_switch;
                tocolor <= 0;
                e1x<=0;
                e1y<=0;
                e2x<=0;
                e2y<=0;
                pixel_location <= 0;
                pixel_ready <= 0;
                distance <= 0;
            end

            // This state is entered when we are waiting for the a new buffer to be ready
            // The reason we enter this state instead of just returning to the wait state is
            // because we want to wait until the next rising edge of memory ready, which would
            // imply a buffer switch has occurred, and the new buffer has been cleared
            S_wait_for_buffer_switch: begin
                if (~old_mem_ready && mem_ready) begin
                    tri_done <= 1;
                    state <= S_wait;
                end
                else begin
                    tri_done <= 0;
                    state <= S_wait_for_buffer_switch;
                end
                tocolor <= 0;
                e1x<=0;
                e1y<=0;
                e2x<=0;
                e2y<=0;
                pixel_location <= 0;
                pixel_ready <= 0;
                distance <= 0;
            end
        endcase
    end
end
endmodule
```

```
/*****************************************
Created By: Zhou, Hao (Steven)           *
6.111 Final Project: Digital Stereoscope *
*****************************************/


/*************************************************************************************************
Pixel Distance module generates a rough estimate for the distance values of a point inside a
triangle given the distances to the corners of the triangle, and the area of the three sub
triangles.  The reasoning behind this algorithm is, if the area of a sub triangle is large, then
the point in question must be closer to the corner not included in the sub triangle, thus the
weight of the distance of that corner should be greater than that of the other corners.
*************************************************************************************************/

module PixelDistance(clk, slow_clk, a1, a2, a3, at, c1dist, c2dist, c3dist, pixdist);
    input clk;                // Clock for the divider modules
    input slow_clk;           // Clock enable signal for the divider modules
    input [12:0] a1;          // Area of sub triangle with corners 1 and 2
    input [12:0] a2;          // Area of sub triangle with corners 1 and 3
    input [12:0] a3;          // Area of sub triangle with corners 2 and 3
    input [12:0] at;          // Total area of the triangle
    input [12:0] c1dist;      // Distance to corner 1
    input [12:0] c2dist;      // Distance to corner 2
    input [12:0] c3dist;      // Distance to corner 3
    output [10:0] pixdist;    // Distance to pixel

    wire [26:0]  a1c3, a2c2, a3c1;
    wire [26:0]  quot1, quot2, quot3, quott;
    wire [12:0] remainder1, remainder2, remainder3;
    wire rfd1, rfd2, rfd3;

    // Computes the product of areas of sub triangles and the distance of the opposite corner
    assign a1c3 = a1*c3dist;
    assign a2c2 = a2*c2dist;
    assign a3c1 = a3*c1dist;

    // Divide each of the values above by the total area of the triangle
    divider27by13 d1 (.clk(clk), .dividend (a1c3), .divisor(at), .quot(quot1), .ce(slow_clk),
           .remd(remainder1), .rfd(rfd1));
    divider27by13 d2 (.clk(clk), .dividend (a2c2), .divisor(at), .quot(quot2), .ce(slow_clk),
           .remd(remainder2), .rfd(rfd2));
    divider27by13 d3 (.clk(clk), .dividend (a3c1), .divisor(at), .quot(quot3), .ce(slow_clk),
           .remd(remainder3), .rfd(rfd3));

    // The summation of all those quotients give a rough estimate of what the total distance is\
    assign quott = quot1 + quot2 + quot3;

    // Since we only store 11 bits of distance data in memory, we take the top 11 bits
    assign pixdist = quott[12:2];

endmodule
```

```
/**************************************
Created By: Zhou, Hao (Steven)           *
6.111 Final Project: Digital Stereoscope *
**************************************/


/*************************************************************************************************
The purpose of this module is to compute twice the area of a triangle, given its coordinates on a
x, y plane.  Twice the area is simply the magnitude of the cross product.
*************************************************************************************************/

module TriangleArea(x1, y1, x2, y2, x3, y3, area);

    input [9:0] x1, y1, x2, y2, x3, y3;      // (x, y) locations of the corners of the triangle
    output [17:0] area;                      // Area of the triangle


    wire signed [10:0] Sx1, Sy1, Sx2, Sy2, Sx3, Sy3;        // Sign extend all the coordinates
    wire signed [18:0] area_signed;                         // Signed area value
    wire [17:0] area;                                       // Absolute area value

    assign Sx1 = {1'b0, x1};
    assign Sy1 = {1'b0, y1};
    assign Sx2 = {1'b0, x2};
    assign Sy2 = {1'b0, y2};
    assign Sx3 = {1'b0, x3};
    assign Sy3 = {1'b0, y3};

    // Calculate the cross product
    assign area_signed = ((Sx1-Sx2)*(Sy1-Sy3)-(Sy1-Sy2)*(Sx1-Sx3));

    // Take the absolute value and set that as triangle area
    assign area = area_signed[18:18] ? ~area_signed[17:0]+1'b1 : area_signed[17:0];

endmodule
```

```
/****************************************
Created By: Zhou, Hao (Steven)           *
6.111 Final Project: Digital Stereoscope *
****************************************/

/*************************************************************************************************
The Pixel Extractor Delay module is fairly straight forward.  It takes input signals and delays
each one of them a set number of clock cycles.  This was needed because the divider module had a
latency of about 30 clock cycles, and I had to delay the other signals about 30 clock cycles so
the pixels would correspond to their respective distance values.
*************************************************************************************************/

module PixelExtractorDelay(clk, pixel_location, tocolor, pixel_ready,
        tri_done, distance, pixel_location_delayed, tocolor_delayed, pixel_ready_delayed,
        tri_done_delayed, distance_delayed);

    // Input signal of clock, and the signals that need to be delayed
    input clk;
    input [19:0] pixel_location;
    input tocolor;
    input pixel_ready;
    input tri_done;
    input [10:0] distance;

    // Delayed output signals
    output [19:0] pixel_location_delayed;
    output tocolor_delayed;
    output pixel_ready_delayed;
    output tri_done_delayed;
    output [10:0] distance_delayed;

    reg [19:0] temp_location [29:0];
    reg temp_tocolor [29:0];
    reg temp_pixel_ready [29:0];
    reg temp_tri_done [29:0];
    reg [10:0] distance_delayed;

    reg [19:0] pixel_location_delayed;
    reg tri_done_delayed, tocolor_delayed, pixel_ready_delayed;

    always @(posedge clk) begin
        // Delay pixel location by 30 clock
        temp_location[0] <= pixel_location;
        temp_location[1] <= temp_location[0];
        temp_location[2] <= temp_location[1];
        temp_location[3] <= temp_location[2];
        temp_location[4] <= temp_location[3];
        temp_location[5] <= temp_location[4];
        temp_location[6] <= temp_location[5];
        temp_location[7] <= temp_location[6];
        temp_location[8] <= temp_location[7];
        temp_location[9] <= temp_location[8];
        temp_location[10] <= temp_location[9];
        temp_location[11] <= temp_location[10];
        temp_location[12] <= temp_location[11];
        temp_location[13] <= temp_location[12];
        temp_location[14] <= temp_location[13];
        temp_location[15] <= temp_location[14];
        temp_location[16] <= temp_location[15];
        temp_location[17] <= temp_location[16];
        temp_location[18] <= temp_location[17];
        temp_location[19] <= temp_location[18];
        temp_location[20] <= temp_location[19];
        temp_location[21] <= temp_location[20];
        temp_location[22] <= temp_location[21];
        temp_location[23] <= temp_location[22];
        temp_location[24] <= temp_location[23];
        temp_location[25] <= temp_location[24];
        temp_location[26] <= temp_location[25];
        temp_location[27] <= temp_location[26];
        temp_location[28] <= temp_location[27];
```

```
temp_location[29] <= temp_location[28];
pixel_location_delayed <= temp_location[29];

// Delay tocolor by 30 clock
temp_tocolor[0] <= tocolor;
temp_tocolor[1] <= temp_tocolor[0];
temp_tocolor[2] <= temp_tocolor[1];
temp_tocolor[3] <= temp_tocolor[2];
temp_tocolor[4] <= temp_tocolor[3];
temp_tocolor[5] <= temp_tocolor[4];
temp_tocolor[6] <= temp_tocolor[5];
temp_tocolor[7] <= temp_tocolor[6];
temp_tocolor[8] <= temp_tocolor[7];
temp_tocolor[9] <= temp_tocolor[8];
temp_tocolor[10] <= temp_tocolor[9];
temp_tocolor[11] <= temp_tocolor[10];
temp_tocolor[12] <= temp_tocolor[11];
temp_tocolor[13] <= temp_tocolor[12];
temp_tocolor[14] <= temp_tocolor[13];
temp_tocolor[15] <= temp_tocolor[14];
temp_tocolor[16] <= temp_tocolor[15];
temp_tocolor[17] <= temp_tocolor[16];
temp_tocolor[18] <= temp_tocolor[17];
temp_tocolor[19] <= temp_tocolor[18];
temp_tocolor[20] <= temp_tocolor[19];
temp_tocolor[21] <= temp_tocolor[20];
temp_tocolor[22] <= temp_tocolor[21];
temp_tocolor[23] <= temp_tocolor[22];
temp_tocolor[24] <= temp_tocolor[23];
temp_tocolor[25] <= temp_tocolor[24];
temp_tocolor[26] <= temp_tocolor[25];
temp_tocolor[27] <= temp_tocolor[26];
temp_tocolor[28] <= temp_tocolor[27];
temp_tocolor[29] <= temp_tocolor[28];
tocolor_delayed <= temp_tocolor[29];

// Delay pixel ready by 30 clocks
temp_pixel_ready[0] <= pixel_ready;
temp_pixel_ready[1] <= temp_pixel_ready[0];
temp_pixel_ready[2] <= temp_pixel_ready[1];
temp_pixel_ready[3] <= temp_pixel_ready[2];
temp_pixel_ready[4] <= temp_pixel_ready[3];
temp_pixel_ready[5] <= temp_pixel_ready[4];
temp_pixel_ready[6] <= temp_pixel_ready[5];
temp_pixel_ready[7] <= temp_pixel_ready[6];
temp_pixel_ready[8] <= temp_pixel_ready[7];
temp_pixel_ready[9] <= temp_pixel_ready[8];
temp_pixel_ready[10] <= temp_pixel_ready[9];
temp_pixel_ready[11] <= temp_pixel_ready[10];
temp_pixel_ready[12] <= temp_pixel_ready[11];
temp_pixel_ready[13] <= temp_pixel_ready[12];
temp_pixel_ready[14] <= temp_pixel_ready[13];
temp_pixel_ready[15] <= temp_pixel_ready[14];
temp_pixel_ready[16] <= temp_pixel_ready[15];
temp_pixel_ready[17] <= temp_pixel_ready[16];
temp_pixel_ready[18] <= temp_pixel_ready[17];
temp_pixel_ready[19] <= temp_pixel_ready[18];
temp_pixel_ready[20] <= temp_pixel_ready[19];
temp_pixel_ready[21] <= temp_pixel_ready[20];
temp_pixel_ready[22] <= temp_pixel_ready[21];
temp_pixel_ready[23] <= temp_pixel_ready[22];
temp_pixel_ready[24] <= temp_pixel_ready[23];
temp_pixel_ready[25] <= temp_pixel_ready[24];
temp_pixel_ready[26] <= temp_pixel_ready[25];
temp_pixel_ready[27] <= temp_pixel_ready[26];
temp_pixel_ready[28] <= temp_pixel_ready[27];
temp_pixel_ready[29] <= temp_pixel_ready[28];
pixel_ready_delayed <= temp_pixel_ready[29];

// Delay triangle done signal by 30 clock cycles
temp_tri_done[0] <= tri_done;
```

```
        temp_tri_done[1] <= temp_tri_done[0];
        temp_tri_done[2] <= temp_tri_done[1];
        temp_tri_done[3] <= temp_tri_done[2];
        temp_tri_done[4] <= temp_tri_done[3];
        temp_tri_done[5] <= temp_tri_done[4];
        temp_tri_done[6] <= temp_tri_done[5];
        temp_tri_done[7] <= temp_tri_done[6];
        temp_tri_done[8] <= temp_tri_done[7];
        temp_tri_done[9] <= temp_tri_done[8];
        temp_tri_done[10] <= temp_tri_done[9];
        temp_tri_done[11] <= temp_tri_done[10];
        temp_tri_done[12] <= temp_tri_done[11];
        temp_tri_done[13] <= temp_tri_done[12];
        temp_tri_done[14] <= temp_tri_done[13];
        temp_tri_done[15] <= temp_tri_done[14];
        temp_tri_done[16] <= temp_tri_done[15];
        temp_tri_done[17] <= temp_tri_done[16];
        temp_tri_done[18] <= temp_tri_done[17];
        temp_tri_done[19] <= temp_tri_done[18];
        temp_tri_done[20] <= temp_tri_done[19];
        temp_tri_done[21] <= temp_tri_done[20];
        temp_tri_done[22] <= temp_tri_done[21];
        temp_tri_done[23] <= temp_tri_done[22];
        temp_tri_done[24] <= temp_tri_done[23];
        temp_tri_done[25] <= temp_tri_done[24];
        temp_tri_done[26] <= temp_tri_done[25];
        temp_tri_done[27] <= temp_tri_done[26];
        temp_tri_done[28] <= temp_tri_done[27];
        temp_tri_done[29] <= temp_tri_done[28];
        tri_done_delayed <= temp_tri_done[29];

        // latch distance value
        distance_delayed <= distance;
    end
endmodule
```

```
/**************************************
Created By: Zhou, Hao (Steven)           *
6.111 Final Project: Digital Stereoscope *
**************************************/

/*************************************************************************************************
The Color LUT table converts an 8 bit color value to a 24 bit color value.  The first three bit
of the 8 bit color scheme represent red, the next 3 represent green, and the last 2 represent
blue, since the human eye is less sensitive to blue.  The 24 bit RGB value has 8 bits for each
color
*************************************************************************************************/

module ColorLUT(color, rgb);
    input [7:0] color;            // 8 bit color input
    output [23:0] rgb;            // 24 bit rgb output

    reg [23:0] rgb;

    // Acts like a look up table in memory, converts each component (R, G, B) to a 8 bit value
    always @ (color)   begin
        case (color[7:5])
             0: rgb[23:16] = 8'h00;
             1: rgb[23:16] = 8'h24;
             2: rgb[23:16] = 8'h48;
             3: rgb[23:16] = 8'h6c;
             4: rgb[23:16] = 8'h90;
             5: rgb[23:16] = 8'hb4;
             6: rgb[23:16] = 8'hd8;
             7: rgb[23:16] = 8'hff;
        endcase

        case (color[4:2])
             0: rgb[15:8] = 8'h00;
             1: rgb[15:8] = 8'h24;
             2: rgb[15:8] = 8'h48;
             3: rgb[15:8] = 8'h6c;
             4: rgb[15:8] = 8'h90;
             5: rgb[15:8] = 8'hb4;
             6: rgb[15:8] = 8'hd8;
             7: rgb[15:8] = 8'hff;
        endcase

        case (color[1:0])
             0: rgb[7:0] = 8'h00;
             1: rgb[7:0] = 8'h55;
             2: rgb[7:0] = 8'haa;
             3: rgb[7:0] = 8'hff;
        endcase
    end

endmodule
```

```
/*************************************
Created By: Zhou, Hao (Steven)          *
6.111 Final Project: Digital Stereoscope *
*************************************/

/*********************************************************************************************
The shading module takes the triangle's RGB value and the current pixel's distance from the eyes,
and makes the image brighter or darker depending on the proximity of the object to the eye.
*********************************************************************************************/

module Shading(clk, slow_clk, pixel_ready, tri_done, tocolor, pixel_location, distance, rgb,
       shading_ready, pixel_address, pixel_data, shading_done);

    input clk;                    // Clock for sequential logic
    input slow_clk;               // High once every three clocks
    input pixel_ready;            // First pixel data for the triangle is ready to be sent
    input tri_done;               // Triangle is done rendering
    input tocolor;                // Signal to determine whether the pixel need to be colored
    input [19:0] pixel_location;  // {X, Y} location of the pixel
    input [10:0] distance;        // distance from the eye to that pixel
    input [23:0] rgb;             // RGB value for that pixel

    output shading_ready;         // Shading Module is ready to output pixel address and data
    output shading_done;          // Shading Module has completed outputting address and data
    output [17:0] pixel_address;  // Pixel address in ZBT memory
    output [34:0] pixel_data;     // Pixel data to store in ZBT memory

    parameter [7:0] COLORMAX = 8'hff;     // Max color for R, G, or B.

    wire [9:0] pixel_x, pixel_y;
    wire [6:0] shading_add, shading_sub;
    wire [7:0] r, g, b;
    wire [34:0] pixel_data_temp;

    reg [34:0] pixel_data;
    reg [7:0] new_r, new_g, new_b;        // The new colors of the pixel
    reg [10:0] new_distance;              // New distance of the pixel
    reg shading_ready, shading_done;
    reg [17:0] pixel_address;

    // Assigns the X and Y locations
    assign pixel_x = pixel_location[19:10];
    assign pixel_y = pixel_location[9:0];

    // Depending on the most significant bit of distance, we either subtract values from R, G,
    // and B, or we add values to it.  The amount we add or subtract is equal to the absolute
    // value of the difference between the pixel distance and 1024.
    assign shading_add = distance [9:3];
    assign shading_sub = ~distance [9:3] + 1;

    // Extract the RGB values of the triangle
    assign r = rgb[23:16];
    assign g = rgb[15:8];
    assign b   = rgb[7:0];

    // This sequential block obtains the new "shaded" color of the triangle, as well as latches
    // all of the output data
    always @(posedge clk) begin
        // Proceed with logic only when slow clock is high
        if (slow_clk) begin
            shading_ready <= pixel_ready;
            shading_done <= tri_done;
            // Pixel address is 640 times the y value plus the x value
            pixel_address <= (pixel_x + ((pixel_y<<9)+(pixel_y<<7)));
            // Determines the new color of the R, G, and B components of the triangle
            new_r <= (distance[10] == 1) ? ((r > shading_add) ? (r - shading_add) : 0) :
                ((r < (COLORMAX - shading_sub)) ? (r + shading_sub) : COLORMAX);
            new_g <= (distance[10] == 1) ? ((g > shading_add) ? (g - shading_add) : 0) :
                ((g < (COLORMAX - shading_sub)) ? (g + shading_sub) : COLORMAX);
            new_b <= (distance[10] == 1) ? ((b > shading_add) ? (b - shading_add) : 0) :
                ((b < (COLORMAX - shading_sub)) ? (b + shading_sub) : COLORMAX);
```

```
            new_distance <= distance;
            pixel_data <= pixel_data_temp;
        end
    end

    // Assigns the pixel color to black, and distance to max distance if the pixel does not
    // need to be colored, meaning the pixel is not inside of the triangle
    assign pixel_data_temp = tocolor ? {new_r, new_g, new_b, new_distance} : 35'h7ff;
endmodule
```

```
/*****************************************
** trianglememory                       **
** Created by: Tony Ng                  **
** 6.111 Final Project: Digital Stereoscope  **
*****************************************/

/*************************************************************************************
The triangle memory module stores the coordinates of the three corners of a triangle and the
color of the triangle.
*************************************************************************************/
module trianglememory(addr, clk, din1, din2, din3, dinc, dout1, dout2, dout3, doutc, we,
num_triangles);
    input [8:0] addr;                // address input into all four BRAMS
    input clk;
    input [35:0] din1, din2, din3;    // data inputs for three points of triangle
    input [7:0] dinc;                 // data input for color
    input we;      // write enable, set to "1" to overwrite current address with input data

    output [35:0] dout1, dout2, dout3;    // output containing three corners of triangle
                                          // x coordinate in bits [35:24], y coordinates
                                          // in bits [23:12], and z coordinate in [11:0]

    output [7:0] doutc;           // output containing 8 bit color of triangle
    output [9:0] num_triangles;   // output containing the number of triangles to process

    wire [9:0] num_triangles;
    wire [7:0] doutc;

    assign num_triangles = 376;   // set this to the number of triangles stored in ram

    // individual BRAMs containing triangle data
    xram512x36 xram(addr, clk, din1, dout1, we);
    yram512x36 yram(addr, clk, din2, dout2, we);
    zram512x36 zram(addr, clk, din3, dout3, we);
    cram512x8 cram(addr[8:0], clk, dinc, doutc, we);
endmodule
```

```
/*****************************************
** perspectiveprojector                 **
** Created by: Tony Ng                  **
** 6.111 Final Project: Digital Stereoscope  **
*****************************************/
/*************************************************************************************************
The perspective projector module loops through a triangle memory module. For every triangle in
the memory, the perspective projector calculates where it would project onto the screen based on
where the two eyes in the world are located and the gaze and tilt angles that the eyes are
looking towards. The projector processes a triangle when it receives a start signal and returns a
done signal when it has completed calculations. Also, a last_triangle signal is turned on when
the last triangle in the memory is processed, signaling the end of the frame.
*************************************************************************************************/
module perspectiveprojector(clk, slow_clk, reset, corner1, corner2, corner3, inColor, eye1, eye2,
    gaze_angle, tilt_angle, start, num_triangles, reset_last_triangle, outColor, triangle_addr,
    e1c1, e1c2, e1c3, e2c1, e2c2, e2c3, done, last_triangle);

    input clk, slow_clk;      // two different clocks
    input reset;              // set to "1" to reset perspective projector

    input [35:0] corner1;     // contains coordinates of a corner of a triangle
    input [35:0] corner2;
    input [35:0] corner3;
                              // Note: x coordinate in [35:24], y in [23:12], and z in [11:0]

    input [7:0] inColor;      // 8-bit color of triangle
    input [35:0] eye1, eye2;  // coordinates of the two eyes
                              // x coordinate in [35:24], y in [23:12], z in [11:0]

    input [15:0] gaze_angle;  // gaze angle (angle eye is looking at in xy-plane: the azimuth)
    input [15:0] tilt_angle;  // tilt angle (angle eye is looking up or down: the altitude)
    input start;                  // signal to project next angle
    input [8:0] num_triangles;    // total number of angles to project per frame
    input reset_last_triangle;    // signal that frame has been finished
                                  // start processing triangles from beginning again

    output [7:0] outColor;        // output of 8-bit color of triangle
    output [8:0] triangle_addr;   // address of triangle in memory

    output [32:0] e1c1, e1c2, e1c3;
    output [32:0] e2c1, e2c2, e2c3;   // 2-D coordinates of projection
                                      // x coordinate in [32:23], y coordinate in [22:13]
                                      // depth of triangle in bits [12:0]

    output done;          // signals that output is ready, or valid
    output last_triangle; // signals that all triangles have been processed

    reg [7:0] outColor;
    reg [8:0] triangle_addr;
    reg done;
    reg last_triangle;
    reg [32:0] e1c1, e1c2, e1c3, e2c1, e2c2, e2c3;

    reg [3:0] delay_counter;  // gives time for calculations to stabilize

    reg [35:0] eye;           // contains current eye being processed

    // calculate angle of the vector formed by the eye and a corner of the triangle with
    // respect to the +x axis when projected onto xy-plane, or the ground (the azimuth angle)
    wire [15:0] xy_angle1, xy_angle2, xy_angle3;
    arctan_estimator atan01(clk, {1'b0,eye[35:24]}, {1'b0,eye[23:12]}, {1'b0,corner1[35:24]},
                                 {1'b0, corner1[23:12]}, xy_angle1);
    arctan_estimator atan02(clk, {1'b0,eye[35:24]}, {1'b0,eye[23:12]}, {1'b0,corner2[35:24]},
                                 {1'b0,corner2[23:12]}, xy_angle2);
    arctan_estimator atan03(clk, {1'b0,eye[35:24]}, {1'b0,eye[23:12]}, {1'b0,corner3[35:24]},
                                 {1'b0,corner3[23:12]}, xy_angle3);

    // calculate the distance of the eye to corner when projected onto the xy-plane (the ground)
    wire [12:0] xy_dist1, xy_dist2, xy_dist3;
```

```
distance_finder temp1(clk, {eye[35:12], 12'b000000000000},
    {corner1[35:12], 12'b000000000000}, xy_dist1);
distance_finder temp2(clk, {eye[35:12], 12'b000000000000},
    {corner2[35:12], 12'b000000000000}, xy_dist2);
distance_finder temp3(clk, {eye[35:12], 12'b000000000000},
    {corner3[35:12], 12'b000000000000}, xy_dist3);

// calculate the angle that a corner is above or below the eye, or the altitude angle
wire [15:0] xz_angle1, xz_angle2, xz_angle3;
arctan_estimator atan07(clk, 13'b0000000000000, {1'b0,eye[11:0]}, xy_dist1[12:0],
                             {1'b0,corner1[11:0]}, xz_angle1);
arctan_estimator atan08(clk, 13'b0000000000000, {1'b0,eye[11:0]}, xy_dist2[12:0],
                             {1'b0,corner2[11:0]}, xz_angle2);
arctan_estimator atan09(clk, 13'b0000000000000, {1'b0,eye[11:0]}, xy_dist3[12:0],
                             {1'b0,corner3[11:0]}, xz_angle3);


// Using previous calculations, determines which row/column on the
// screen that the corner of the triangle projects onto.
// check = "1" if the corner is on the screen, otherwise output is unspecified
wire check1, check2, check3, check4, check5, check6;
wire [9:0] x1, x2, x3, y1, y2, y3;
angle_screen_converter convert01(gaze_angle, xy_angle1, x1, check1);
angle_screen_converter convert02(gaze_angle, xy_angle2, x2, check2);
angle_screen_converter convert03(gaze_angle, xy_angle3, x3, check3);
angle_screen_converter convert04(tilt_angle, xz_angle1, y1, check4);
angle_screen_converter convert05(tilt_angle, xz_angle2, y2, check5);
angle_screen_converter convert06(tilt_angle, xz_angle3, y3, check6);

// if the 3 corners all have valid (x,y) coords, then on_screen set to "1"
wire on_screen;
assign on_screen = (check1 && check2 && check3 && check4 && check5 && check6) ? 1 : 0;

// finds the unprojected distance of the eye to the corner
wire [12:0] dist_c1, dist_c2, dist_c3;
distance_finder df1(clk, eye, corner1, dist_c1);
distance_finder df2(clk, eye, corner2, dist_c2);
distance_finder df3(clk, eye, corner3, dist_c3);

// ***** states *****
parameter WAITING = 0;            // wait for start signal
parameter GET_TRIANGLE = 1;       // get a new triangle from memory
parameter PROJECT_POINTS_1 = 2;   // attempt to project points from eye1
parameter PROJECT_POINTS_2 = 3;   // attempt to project points from eye2
parameter LAST = 4;               // all triangles processed

parameter MAXADDR=9'b111111111;   // largest possible address for triangle memory
reg [2:0] state;


always @ (posedge clk) begin
    if (slow_clk) begin
        if (reset) begin              // initial starting state
            triangle_addr <= MAXADDR;
            done <= 0;
            state <= WAITING;
            last_triangle <= 0;
            delay_counter <= 1;
            outColor <= inColor;
            e1c1 <= 0;
            e1c2 <= 0;
            e1c3 <= 0;
            e2c1 <= 0;
            e2c2 <= 0;
            e2c3 <= 0;
        end

        else begin
            case (state)
                WAITING: begin        // waiting for start signal
                    if (start) begin  // found start signal, prepare to get triangle
```

```
        triangle_addr <= triangle_addr+1;
        done <= 0;
        state <=    GET_TRIANGLE;
        last_triangle <= 0;
        delay_counter <= 1;
        outColor <= inColor;
        e1c1 <= 0;
        e1c2 <= 0;
        e1c3 <= 0;
        e2c1 <= 0;
        e2c2 <= 0;
        e2c3 <= 0;
    end
    else begin         // no start signal, keep everything the same
        triangle_addr <= triangle_addr;
        done <= 0;
        state <=    WAITING;
        last_triangle <= last_triangle;
        delay_counter <= 1;
        outColor <= inColor;
        e1c1 <= e1c1;
        e1c2 <= e1c2;
        e1c3 <= e1c3;
        e2c1 <= e2c1;
        e2c2 <= e2c2;
        e2c3 <= e2c3;
    end
end

GET_TRIANGLE: begin           // try to get a triangle from memory
    if (triangle_addr  >= num_triangles) begin   // no triangles left
        triangle_addr <= triangle_addr;
        done <= 0;
        state <=   LAST;
        last_triangle <= 0;
        delay_counter <= 1;
        outColor <= inColor;
        e1c1 <= 0;
        e1c2 <= 0;
        e1c3 <= 0;
        e2c1 <= 0;
        e2c2 <= 0;
        e2c3 <= 0;
    end
    else begin                // got a triangle, prepare to project from eye1
        triangle_addr <= triangle_addr;
        done <= 0;
        state <=   PROJECT_POINTS_1;
        last_triangle <= 0;
        delay_counter <= 1;
        outColor <= inColor;
        e1c1 <= 0;
        e1c2 <= 0;
        e1c3 <= 0;
        e2c1 <= 0;
        e2c2 <= 0;
        e2c3 <= 0;
    end
end

PROJECT_POINTS_1: begin   // project triangle onto eye1
    eye <= eye1;
    if (delay_counter==0) begin   //calculations stabilized, check projection
        if (on_screen) begin      //projection on screen, now try on eye2 next
            triangle_addr <= triangle_addr;
            done <= 0;
            state <=   PROJECT_POINTS_2;
            last_triangle <= 0;
            delay_counter <= 1;
            outColor <= inColor;
            e1c1 <= {x1, y1, dist_c1};
```

```
            e1c2 <= {x2, y2, dist_c2};
            e1c3 <= {x3, y3, dist_c3};
            e2c1 <= e2c1;
            e2c2 <= e2c2;
            e2c3 <= e2c3;
        end

        else begin     // projection not on screen, ignore and get new triangle
            triangle_addr <= triangle_addr+1;
            done <= 0;
            state <=   GET_TRIANGLE;
            last_triangle <= 0;
            delay_counter <= 1;
            outColor <= inColor;
            e1c1 <= 0;
            e1c2 <= 0;
            e1c3 <= 0;
            e2c1 <= 0;
            e2c2 <= 0;
            e2c3 <= 0;
        end
    end

    else begin         // delay while calculating, keep everything the same
        triangle_addr <= triangle_addr;
        done <= 0;
        state <=   PROJECT_POINTS_1;
        last_triangle <= 0;
        delay_counter <= delay_counter + 1;
        outColor <= inColor;
        e1c1 <= e1c1;
        e1c2 <= e1c2;
        e1c3 <= e1c3;
        e2c1 <= e2c1;
        e2c2 <= e2c2;
        e2c3 <= e2c3;
    end
end


PROJECT_POINTS_2: begin   // project triangle onto eye2
    eye <= eye2;
    if (delay_counter==0) begin   // calculations stabilized, check protection
        if (on_screen) begin      // projection is on screen, signal done
            triangle_addr <= triangle_addr;
            done <= 1;
            state <=   WAITING;
            last_triangle <= 0;
            delay_counter <= 1;
            outColor <= inColor;
            e1c1 <= e1c1;
            e1c2 <= e1c2;
            e1c3 <= e1c3;
            e2c1 <= {x1, y1, dist_c1};
            e2c2 <= {x2, y2, dist_c2};
            e2c3 <= {x3, y3, dist_c3};
        end

        else begin     // projection not on screen, go get a new triangle
            triangle_addr <= triangle_addr+1;
            done <= 0;
            state <=   GET_TRIANGLE;
            last_triangle <= 0;
            delay_counter <= 1;
            outColor <= inColor;
            e1c1 <= 0;
            e1c2 <= 0;
            e1c3 <= 0;
            e2c1 <= 0;
            e2c2 <= 0;
            e2c3 <= 0;
```

```
                        end
                    end

                    else begin     // delay until calculations are done, keep everything same
                        triangle_addr <= triangle_addr;
                        done <= 0;
                        state <=   PROJECT_POINTS_2;
                        last_triangle <= 0;
                        delay_counter <= delay_counter + 1;
                        outColor <= inColor;
                        e1c1 <= e1c1;
                        e1c2 <= e1c2;
                        e1c3 <= e1c3;
                        e2c1 <= e2c1;
                        e2c2 <= e2c2;
                        e2c3 <= e2c3;
                    end
                end

                LAST: begin    // all triangles in memory processed
                    if (reset_last_triangle) begin    // start looping through triangles again
                        triangle_addr <= MAXADDR;
                        done <= 1;
                        state <=   WAITING;
                        last_triangle <= 0;
                        delay_counter <= 1;
                        outColor <= inColor;
                        e1c1 <= 0;
                        e1c2 <= 0;
                        e1c3 <= 0;
                        e2c1 <= 0;
                        e2c2 <= 0;
                        e2c3 <= 0;
                    end
                    else begin     // wait for signal to start looping through triangles again
                        triangle_addr <= MAXADDR;
                        done <= 1;
                        state <=   LAST;
                        last_triangle <= 1;
                        delay_counter <= 1;
                        outColor <= inColor;
                        e1c1 <= 0;
                        e1c2 <= 0;
                        e1c3 <= 0;
                        e2c1 <= 0;
                        e2c2 <= 0;
                        e2c3 <= 0;
                    end
                end
            endcase
        end
    end
end

endmodule
```

```
/*****************************************
** angle_screen_converter                **
** Created By: Tony Ng                   **
** 6.111 Final Project: Digital Stereoscope  **
*****************************************/

/*************************************************************************************************
The angle screen converter takes in the angle of the eye and the angle of a point with respect to
a particular set of axes and determines whether the point should appear on the screen. If the
point appears on the screen, on_screen is set high and the output contains the pixel where the
point should appear. If the point is not on the screen, on_screen is low and the output is
unspecified.
*************************************************************************************************/
module angle_screen_converter (eye_angle, point_angle, out, on_screen);
    input [15:0] eye_angle, point_angle; //angle of eye and a point
    output [9:0] out;  //pixel on screen that point projects onto
    output on_screen;  //"1" when out is valid, which is when point projects onto screen
                       //"0" when point projects off screen, out is garbage data

    wire [16:0] delta;
    wire [16:0] delta2;
    wire on_screen;

    parameter TWO_PI = 16'b1100100100001111;         // two pi
    parameter  PI = 16'b0110010010000001;            // pi
    parameter HALF_SCREEN = 16'b0001010000000000;    // radians on half of the screen
                                                     // set to 5/8 pi

    parameter FULL_SCREEN = 16'b0010100000000000;    // radians displayed on entire screen
                                                     // set to 1.25 pi

    // rotate angles so that the eye points to the middle of the screen
    assign delta = (HALF_SCREEN + point_angle) - eye_angle;

    // fix angles that are more than 2pi so they range from 0 to 2pi
    assign delta2 = (delta < TWO_PI) ? delta : ((delta[16] == 1) ? delta + TWO_PI : delta -
                    TWO_PI);

    // if angle is off the screen, set on_screen to "0", otherwise set to "1"
    assign on_screen = (delta2 > FULL_SCREEN) ? 0 : 1;

    // convert from radians to pixels ranging from 0 to 319
    assign out = 319 - delta2[14:5];
endmodule
```

```
/*****************************************
** arctan_estimator                     **
** Created By: Tony Ng                  **
** 6.111 Final Project: Digital Stereoscope  **
*****************************************/

/*************************************************************************************************
The arctangent estimator takes as input two vectors and returns the angle between them. The angle
is a binary fraction: the first three bits represent the integer part and the last 13 bits
represent the fraction part (in decimal, this part is equivalent to the decimal value of the 13
bits divided by 2^13).
*************************************************************************************************/
module arctan_estimator (clk, tail_x, tail_y, head_x, head_y, angle);

    input clk;
    input [12:0] tail_x;   // x-coord of tail of vector
    input [12:0] tail_y;   // y-coord of tail of vector
    input [12:0] head_x;   // x-coord of head of vector
    input [12:0] head_y;   // y-coord of head of vector

    output [15:0] angle;   // output angle, first 3 bits integer part
                           // last 13 bits, fraction part = (angle[12:0])/2^13

    wire [15:0] theta;
    wire [13:0] x_component, y_component;
    wire [15:0] angle;

    // shift vectors to origin and convert to two's complement
    assign x_component = {1'b0, head_x} + (~{1'b0, tail_x} + 1);
    assign y_component = {1'b0, head_y} + (~{1'b0, tail_y} + 1);

    // calculate arctangent of vector
    // Note: CORDIC estimator returns value between -pi and pi
    arctan_cordic estimator(x_component, y_component, clk, theta);  // Xilinx CORDIC module

    //note: 110.0100100001111 approximately 2*pi unsigned
    //converts angle to the range 0 to 2*pi unsigned
    //makes sure output is positive angle
    assign angle = (theta[15] == 0) ? theta : 16'b1100100100001111 - (~theta+1);

endmodule
```

```
/*********************************************
** distance_finder                          **
** Created By: Tony Ng                      **
** 6.111 Final Project: Digital Stereoscope  **
*********************************************/
/*****************************************************************************************
The distance finder computes the distance between the eye and a point.
*****************************************************************************************/
module distance_finder(clk, eye, point, out);
    input clk;
    input [35:0] eye;      // location of eye
    input [35:0] point;    // location of point
    output [12:0] out;     // distance from eye to point

    wire [11:0] x1, x2, y1, y2, z1, z2;

    assign x1 = eye[35:24];   // x coordinate of eye
    assign x2 = point[35:24]; // x coordinate of point
    assign y1 = eye[23:12];   // y coordinate of eye
    assign y2 = point[23:12]; // y coordinate of point
    assign z1 = eye[11:0];    // z coordinate of eye
    assign z2 = point[11:0];  // z coordinate of point

    wire signed [25:0] temp;
    wire [25:0] in;
    wire [12:0] out;

    wire signed [12:0] delta_x, delta_y, delta_z;
    assign delta_x = x1-x2;
    assign delta_y = y1-y2;
    assign delta_z = z1-z2;

    assign in = delta_x*delta_x + delta_y*delta_y + delta_z*delta_z;
    wire throw_away;   // useless bit since distance is never this long in the world
    sqrt_cordic sqrt (in, clk, {throw_away, out});   // Xilinx CORDIC module

endmodule
```

```
/*****************************************
** position_controller                  **
** Created By: Tony Ng                  **
** 6.111 Final Project: Digital Stereoscope  **
*****************************************/

/*************************************************************************************************
The position controller reads inputs from the keyboard and updates the position of the eyes and
the gaze and tilt angles based on the user moving around in the world
*************************************************************************************************/
module position_controller(clock, new_frame, reset, keyboard_clock, keyboard_data, eye1, eye2,
    gaze_angle, tilt_angle);

    input clock;
    input new_frame;                        // enables outputs to change
    input reset;
    input keyboard_clock, keyboard_data;    // from ps2
    output [35:0] eye1, eye2;               // location of eye1 and eye2
    output [15:0] gaze_angle, tilt_angle;   // gaze and tilt angles of eyes

    reg [35:0] position;
    wire [3:0] movement;

    // parameters for movement
    parameter NO_ACTION = 0;
    parameter LEFT = 1;
    parameter RIGHT = 2;
    parameter FORWARD = 3;
    parameter BACKWARD = 4;
    parameter TURN_LEFT = 5;
    parameter TURN_RIGHT = 6;
    parameter LOOK_UP = 7;
    parameter LOOK_DOWN = 8;

    parameter MOVE_SPEED = 2;     // movement speed for left, right, forward, and backward
    parameter TURN_SPEED = 512;   // movement for turn left, turn right, look up, and look down

    parameter TWO_PI = 51471;
    parameter PI_OVER_THREE = 8578;
    parameter  PI = 16'b0110010010000001;

    // figures out what user is typing
    keyboard_controller
           keycontrol(.clock(clock), .new_frame(new_frame), .reset(reset), .keyboard_clock(keyboa
           rd_clock), .keyboard_data(keyboard_data), .movement(movement));

    wire [3:0] d_x, d_y;
    wire [3:0] d_x_complement, d_y_complement;
    wire [7:0] d_eye_out;

    // lookup table to determine location of eyes and direction of movement
    eye_lut64x8 eyelocation(gaze_angle[15:10], clock, d_eye_out);

    // used to determine direction of movement
    assign d_x[3:0] = d_eye_out[7:4];        // contains approximate cos(gaze_angle)
    assign d_y[3:0] = d_eye_out[3:0];        // contains approximate sine(gaze_angle)
    assign d_x_complement = ~d_x + 1;        // two's complement of d_x
    assign d_y_complement = ~d_y + 1;        // two's complement of d_y

    parameter reset_position =36'h3E83E83E8;     // (1000, 1000, 1000)
    parameter reset_eye1 = 36'h3EF3E83E8;        // (1007, 1000, 1000)
    parameter reset_eye2 = 36'h3E13E83E8;        // (993, 1000, 1000)
    parameter reset_gaze = 12868;                // Facing North, or +y direction
    parameter reset_tilt = 0;                    // Looking flat, straight

    reg [15:0] gaze_angle, tilt_angle;
    reg [35:0] eye1, eye2;

    always @ (posedge clock) begin
        if(reset) begin
            eye1 <= reset_eye1;       // initial eye positions
```

```
        eye2 <= reset_eye2;
    end

    // calculates location of eyes based on position and gaze
    if( new_frame) begin
        if(d_y[3]==1 && d_x[3]==1) begin
            eye1[35:24] <= position[35:24] - d_y_complement;
            eye2[35:24] <= position[35:24] + d_y_complement;
            eye1[23:12] <= position[23:12] + d_x_complement;
            eye2[23:12] <= position[23:12] - d_x_complement;
        end
        else if(d_y[3]==1 && d_x[3]==0) begin
            eye1[35:24] <= position[35:24] - d_y_complement;
            eye2[35:24] <= position[35:24] + d_y_complement;
            eye1[23:12] <= position[23:12] - d_x;
            eye2[23:12] <= position[23:12] + d_x;
        end

        else if(d_y[3]==0 && d_x[3]==1) begin
            eye1[35:24] <= position[35:24] + d_y;
            eye2[35:24] <= position[35:24] - d_y;
            eye1[23:12] <= position[23:12] + d_x_complement;
            eye2[23:12] <= position[23:12] - d_x_complement;
            end
        else if(d_y[3]==0 && d_x[3]==0) begin
            eye1[35:24] <= position[35:24] + d_y;
            eye2[35:24] <= position[35:24] - d_y;
            eye1[23:12] <= position[23:12] - d_x;
            eye2[23:12] <= position[23:12] + d_x;
        end
    end
end

always @ (posedge clock) begin
    if(reset) begin               // set initial position, gaze, and tilt angles
        position <= reset_position;
        gaze_angle <= reset_gaze;
        tilt_angle <= reset_tilt;
    end

    if( new_frame) begin          // only update on a new frame
        case(movement)
            NO_ACTION: begin      // no key pressed, keep everything the same
                position <= position;
                tilt_angle <= tilt_angle;
                gaze_angle <= gaze_angle;
            end

            LEFT: begin           // move left (of gaze direction)
                if(d_x[3] == 1 && d_y[3] == 1) begin
                    position[23:12] <= position[23:12] - (d_x_complement<<MOVE_SPEED);
                    position[35:24] <= position[35:24] + (d_y_complement<<MOVE_SPEED);
                end

                else if(d_x[3] == 1 && d_y[3] == 0) begin
                    position[23:12] <= position[23:12] - (d_x_complement<<MOVE_SPEED);
                    position[35:24] <= position[35:24] - (d_y<<MOVE_SPEED);
                end

                else if(d_x[3] == 0 && d_y[3] == 1) begin
                    position[23:12] <= position[23:12] + (d_x<<MOVE_SPEED);
                    position[35:24] <= position[35:24] + (d_y_complement<<MOVE_SPEED);
                end

                else if(d_x[3] == 0 && d_y[3] == 0) begin
                    position[23:12] <= position[23:12] + (d_x<<MOVE_SPEED);
                    position[35:24] <= position[35:24] - (d_y<<MOVE_SPEED);
                end
             end

            RIGHT: begin      // move right (of gaze direction)
```

```
    if(d_x[3] == 1 && d_y[3] == 1) begin
        position[23:12] <= position[23:12] + (d_x_complement<<MOVE_SPEED);
        position[35:24] <= position[35:24] - (d_y_complement<<MOVE_SPEED);
    end

    else if(d_x[3] == 1 && d_y[3] == 0) begin
        position[23:12] <= position[23:12] + (d_x_complement<<MOVE_SPEED);
        position[35:24] <= position[35:24] + (d_y<<MOVE_SPEED);
    end

    else if(d_x[3] == 0 && d_y[3] == 1) begin
        position[23:12] <= position[23:12] - (d_x<<MOVE_SPEED);
        position[35:24] <= position[35:24] - (d_y_complement<<MOVE_SPEED);
    end

    else if(d_x[3] == 0 && d_y[3] == 0) begin
        position[23:12] <= position[23:12] - (d_x<<MOVE_SPEED);
        position[35:24] <= position[35:24] + (d_y<<MOVE_SPEED);
    end
end

FORWARD:   begin      // move forward (towards gaze direction)
    if(d_x[3] == 1 && d_y[3] == 1) begin
        position[35:24] <= position[35:24]-(d_x_complement<<MOVE_SPEED);
        position[23:12] <= position[23:12]-(d_y_complement<<MOVE_SPEED);
    end

    else if(d_x[3] == 1 && d_y[3] == 0) begin
        position[35:24] <= position[35:24]-(d_x_complement<<MOVE_SPEED);
        position[23:12] <= position[23:12]+(d_y<<MOVE_SPEED);
    end

    else if(d_x[3] == 0 && d_y[3] == 1) begin
        position[35:24] <= position[35:24]+(d_x<<MOVE_SPEED);
        position[23:12] <= position[23:12]-(d_y_complement<<MOVE_SPEED);
    end

    else if(d_x[3] == 0 && d_y[3] == 0) begin
        position[35:24] <= position[35:24]+(d_x<<MOVE_SPEED);
        position[23:12] <= position[23:12]+(d_y<<MOVE_SPEED);
    end
end

BACKWARD: begin       // move backwards (opposite of gaze direction)
    if(d_x[3] == 1 && d_y[3] == 1) begin
        position[35:24] <= position[35:24]+(d_x_complement<<MOVE_SPEED);
        position[23:12] <= position[23:12]+(d_y_complement<<MOVE_SPEED);
    end

    else if(d_x[3] == 1 && d_y[3] == 0) begin
        position[35:24] <= position[35:24]+(d_x_complement<<MOVE_SPEED);
        position[23:12] <= position[23:12]-(d_y<<MOVE_SPEED);
    end

    else if(d_x[3] == 0 && d_y[3] == 1) begin
        position[35:24] <= position[35:24]-(d_x<<MOVE_SPEED);
        position[23:12] <= position[23:12]+(d_y_complement<<MOVE_SPEED);
    end

    else if(d_x[3] == 0 && d_y[3] == 0) begin
        position[35:24] <= position[35:24]-(d_x<<MOVE_SPEED);
        position[23:12] <= position[23:12]-(d_y<<MOVE_SPEED);
    end
end

TURN_LEFT: begin      // turn left
    if(gaze_angle+TURN_SPEED >= TWO_PI)
        gaze_angle <= gaze_angle+TURN_SPEED - TWO_PI;
    else
        gaze_angle <= gaze_angle+TURN_SPEED;
end
```

```
                TURN_RIGHT: begin      // turn right
                    if(gaze_angle-TURN_SPEED >= TWO_PI)
                        gaze_angle <= gaze_angle + TWO_PI - TURN_SPEED;
                    else
                        gaze_angle <= gaze_angle-TURN_SPEED;
                end

                LOOK_UP:   begin      // look up
                    if(tilt_angle + TURN_SPEED >= TWO_PI)
                        tilt_angle <= tilt_angle + TWO_PI + TURN_SPEED;
                    else if ((tilt_angle + TURN_SPEED >= TWO_PI – PI_OVER_THREE) || (tilt_angle +
                                TURN_SPEED <= PI_OVER_THREE))
                    tilt_angle <= tilt_angle + TURN_SPEED;
                end

                LOOK_DOWN: begin      // look down
                    if(tilt_angle-TURN_SPEED >= TWO_PI)
                        tilt_angle <= tilt_angle + TWO_PI - TURN_SPEED;
                    else if ((tilt_angle – TURN_SPEED >= TWO_PI - PI_OVER_THREE) || (tilt_angle –
                                TURN_SPEED <= PI_OVER_THREE))
                        tilt_angle <= tilt_angle - TURN_SPEED;
                end

                default: begin
                    position <= position;
                    tilt_angle <= tilt_angle;
                    gaze_angle <= gaze_angle;
                end
            endcase
        end
    end
endmodule
```

```
/*********************************************
** keyboard_controller                      **
** Created By: Tony Ng                      **
** 6.111 Final Project: Perspective Projector    **
*********************************************/
/*************************************************************************************************
Converts specific keyboard inputs into a code for movement and ignores other keystrokes. Keeps
track of latest keystroke until new_frame turns higher, in which case the buffer is cleared, and
the signal for no keystroke is outputted through movement.
*************************************************************************************************/
module keyboard_controller(clock, new_frame, reset, keyboard_clock, keyboard_data, movement);
    input clock, new_frame, reset, keyboard_clock, keyboard_data;
    output [3:0] movement;            // bits encoding movement
    wire [7:0] ascii;
    wire ascii_ready;
    ps2_ascii_input

    keyboard(.clock_27mhz(clock), .reset(reset), .clock(keyboard_clock), .data(keyboard_data),
         .ascii(ascii), .ascii_ready(ascii_ready));

    reg [7:0] temp, key_pressed;
    always @ (posedge clock) begin    // hold last keystroke until new_frame
        if(ascii_ready && new_frame)
            begin
                key_pressed <= ascii;
                temp <= 0;
            end
        else if(ascii_ready)
            temp <= ascii;
        else if(new_frame)
            begin
                key_pressed <= temp;
                temp <= 0;
            end
    end

    parameter NO_ACTION = 0;      // code for movements
    parameter LEFT = 1;
    parameter RIGHT = 2;
    parameter FORWARD = 3;
    parameter BACKWARD = 4;
    parameter TURN_LEFT = 5;
    parameter TURN_RIGHT = 6;
    parameter LOOK_UP = 7;
    parameter LOOK_DOWN = 8;
    parameter UP = 9;
    parameter DOWN = 10;
    parameter A = 8'h41;          // code for keystrokes
    parameter D = 8'h44;
    parameter  W = 8'h57;
    parameter S = 8'h53;
    parameter Q = 8'h51;
    parameter  E = 8'h45;
    parameter  R = 8'h52;
    parameter  F = 8'h46;

    reg [3:0] movement;
    always @ (key_pressed) begin
        case(key_pressed)         // assign movement based on keystrokes
            A: movement = LEFT;
            D: movement = RIGHT;
            W: movement = FORWARD;
            S: movement = BACKWARD;
            Q: movement = TURN_LEFT;
            E: movement = TURN_RIGHT;
            R: movement = LOOK_UP;
            F: movement = LOOK_DOWN;
            default: movement = NO_ACTION;
        endcase
    end
endmodule
```

```
// Date:   24-Oct-05
// Author: C. Terman / I. Chuang
//
// PS2 keyboard input for 6.111 labkit
//
// INPUTS:
//
//   clock_27mhz   - master clock
//   reset         - active high
//   clock         - ps2 interface clock
//   data          - ps2 interface data
//
// OUTPUTS:
//
//   ascii         - 8 bit ascii code for current character
//   ascii_ready   - one clock cycle pulse indicating new char received


//////////////////////////////////////////////////////////////////////////////

module ps2_ascii_input(clock_27mhz, reset, clock, data, ascii, ascii_ready);

    // module to generate ascii code for keyboard input
    // this is module works synchronously with the system clock

    input clock_27mhz;
    input reset;              // Active high asynchronous reset
    input clock;              // PS/2 clock
    input data;               // PS/2 data
    output [7:0] ascii;       // ascii code (1 character)
    output ascii_ready;       // ascii ready (one clock_27mhz cycle active high)

    reg [7:0] ascii_val;      // internal combinatorial ascii decoded value
    reg [7:0] lastkey;        // last keycode
    reg [7:0] curkey;         // current keycode
    reg [7:0] ascii;          // ascii output (latched & synchronous)
    reg ascii_ready;          // synchronous one-cycle ready flag

    // get keycodes
    wire        fifo_rd;          // keyboard read request
    wire [7:0]  fifo_data;        // keyboard data
    wire        fifo_empty;       // flag: no keyboard data
    wire        fifo_overflow;    // keyboard data overflow

    ps2 myps2(reset, clock_27mhz, clock, data, fifo_rd, fifo_data, fifo_empty, fifo_overflow);

    assign fifo_rd = ~fifo_empty;  // continuous read
    reg key_ready;

    always @(posedge clock_27mhz) begin
        // get key if ready
        curkey <= ~fifo_empty ? fifo_data : curkey;
        lastkey <= ~fifo_empty ? curkey : lastkey;
        key_ready  <= ~fifo_empty;

        // raise ascii_ready for last key which was read
        ascii_ready <= key_ready & ~(curkey[7]|lastkey[7]);
        ascii <=  (key_ready & ~(curkey[7]|lastkey[7])) ? ascii_val : ascii;
    end

    always @(curkey) begin        //convert PS/2 keyboard make code ==> ascii code
        case (curkey)
            8'h1C: ascii_val = 8'h41;     //A
            8'h32: ascii_val = 8'h42;     //B
            8'h21: ascii_val = 8'h43;     //C
            8'h23: ascii_val = 8'h44;     //D
            8'h24: ascii_val = 8'h45;     //E
            8'h2B: ascii_val = 8'h46;     //F
            8'h34: ascii_val = 8'h47;     //G
            8'h33: ascii_val = 8'h48;     //H
            8'h43: ascii_val = 8'h49;     //I
```

```
            8'h3B: ascii_val = 8'h4A;     //J
            8'h42: ascii_val = 8'h4B;     //K
            8'h4B: ascii_val = 8'h4C;     //L
            8'h3A: ascii_val = 8'h4D;     //M
            8'h31: ascii_val = 8'h4E;     //N
            8'h44: ascii_val = 8'h4F;     //O
            8'h4D: ascii_val = 8'h50;     //P
            8'h15: ascii_val = 8'h51;     //Q
            8'h2D: ascii_val = 8'h52;     //R
            8'h1B: ascii_val = 8'h53;     //S
            8'h2C: ascii_val = 8'h54;     //T
            8'h3C: ascii_val = 8'h55;     //U
            8'h2A: ascii_val = 8'h56;     //V
            8'h1D: ascii_val = 8'h57;     //W
            8'h22: ascii_val = 8'h58;     //X
            8'h35: ascii_val = 8'h59;     //Y
            8'h1A: ascii_val = 8'h5A;     //Z

            8'h45: ascii_val = 8'h30;     //0
            8'h16: ascii_val = 8'h31;     //1
            8'h1E: ascii_val = 8'h32;     //2
            8'h26: ascii_val = 8'h33;     //3
            8'h25: ascii_val = 8'h34;     //4
            8'h2E: ascii_val = 8'h35;     //5
            8'h36: ascii_val = 8'h36;     //6
            8'h3D: ascii_val = 8'h37;     //7
            8'h3E: ascii_val = 8'h38;     //8
            8'h46: ascii_val = 8'h39;     //9

            8'h0E: ascii_val = 8'h60;     // `
            8'h4E: ascii_val = 8'h2D;     // -
            8'h55: ascii_val = 8'h3D;     // =
            8'h5C: ascii_val = 8'h5C;     // \
            8'h29: ascii_val = 8'h20;     // (space)
            8'h54: ascii_val = 8'h5B;     // [
            8'h5B: ascii_val = 8'h5D;     // ]
            8'h4C: ascii_val = 8'h3B;     // ;
            8'h52: ascii_val = 8'h27;     // '
            8'h41: ascii_val = 8'h2C;     // ,
            8'h49: ascii_val = 8'h2E;     // .
            8'h4A: ascii_val = 8'h2F;     // /

            8'h5A: ascii_val = 8'h0D;     // enter (CR)
            8'h66: ascii_val = 8'h08;     // backspace

            //  8'hF0: ascii_val = 8'hF0;     // BREAK CODE
            default: ascii_val = 8'h23;       // #
        endcase
    end
endmodule
```

```
//////////////////////////////////////////////////////////////////////
// new synchronous ps2 keyboard driver, with built-in fifo, from Chris Terman

module ps2(reset, clock_27mhz, ps2c, ps2d, fifo_rd, fifo_data, fifo_empty, fifo_overflow);

    input clock_27mhz, reset;
    input ps2c;               // ps2 clock
    input ps2d;               // ps2 data
    input fifo_rd;            // fifo read request (active high)

    output [7:0] fifo_data;   // fifo data output
    output  fifo_empty;       // fifo empty (active high)
    output  fifo_overflow;    // fifo overflow - too much kbd input

    reg [3:0] count;          // count incoming data bits
    reg [9:0] shift;          // accumulate incoming data bits

    reg [7:0] fifo[7:0];      // 8 element data fifo
    reg fifo_overflow;
    reg [2:0] wptr, rptr;     // fifo write and read pointers

    wire [2:0] wptr_inc = wptr + 1;

    assign fifo_empty = (wptr == rptr);
    assign fifo_data = fifo[rptr];

    // synchronize PS2 clock to local clock and look for falling edge
    reg [2:0] ps2c_sync;
    always @ (posedge clock_27mhz)    ps2c_sync <= {ps2c_sync[1:0],ps2c};

    wire sample = ps2c_sync[2] & ~ps2c_sync[1];

    always @ (posedge clock_27mhz) begin
        if (reset) begin
            count <= 0;
            wptr <= 0;
            rptr <= 0;
            fifo_overflow <= 0;
        end

        else if (sample) begin
            // order of arrival: 0,8 bits of data (LSB first),odd parity,1
            if (count==10) begin
                // just received what should be the stop bit
                if (shift[0]==0 && ps2d==1 && (^shift[9:1])==1) begin
                    fifo[wptr] <= shift[8:1];
                    wptr <= wptr_inc;
                    fifo_overflow <= fifo_overflow | (wptr_inc == rptr);
                end
                count <= 0;
            end

            else begin
                shift <= {ps2d,shift[9:1]};
                count <= count + 1;
            end
        end

        // bump read pointer if we're done with current value.
        // Read also resets the overflow indicator
        if (fifo_rd && !fifo_empty) begin
            rptr <= rptr + 1;
            fifo_overflow <= 0;
        end
    end
endmodule
```

```
//
// File:   zbt_6111.v
// Date:   27-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
//
// Simple ZBT driver for the MIT 6.111 labkit, which does not hide the
// pipeline delays of the ZBT from the user.  The ZBT memories have
// two cycle latencies on read and write, and also need extra-long data hold
// times around the clock positive edge to work reliably.
//

////////////////////////////////////////////////////////////////////////////
// Ike's simple ZBT RAM driver for the MIT 6.111 labkit
//
// Data for writes can be presented and clocked in immediately; the actual
// writing to RAM will happen two cycles later.
//
// Read requests are processed immediately, but the read data is not available
// until two cycles after the intial request.
//
// A clock enable signal is provided; it enables the RAM clock when high.

module zbt_6111(clk, cen, we, addr, write_data, read_data,
          ram_clk, ram_we_b, ram_address, ram_data, ram_cen_b);

    input clk;                    // system clock
    input cen;                    // clock enable for gating ZBT cycles
    input we;                     // write enable (active HIGH)
    input [18:0] addr;            // memory address
    input [35:0] write_data;      // data to write
    output [35:0] read_data;      // data read from memory
    output  ram_clk;              // physical line to ram clock
    output  ram_we_b;             // physical line to ram we_b
    output [18:0] ram_address;    // physical line to ram address
    inout [35:0]  ram_data;           // physical line to ram data
    output  ram_cen_b;            // physical line to ram clock enable

    // clock enable (should be synchronous and one cycle high at a time)
    wire    ram_cen_b = ~cen;

    // create delayed ram_we signal: note the delay is by two cycles!
    // ie we present the data to be written two cycles after we is raised
    // this means the bus is tri-stated two cycles after we is raised.

    reg [1:0]   we_delay;

    always @(posedge clk)
        we_delay <= cen ? {we_delay[0],we} : we_delay;

    // create two-stage pipeline for write data

    reg [35:0]  write_data_old1;
    reg [35:0]  write_data_old2;
    always @(posedge clk)
        if (cen)
            {write_data_old2, write_data_old1} <= {write_data_old1, write_data};

    // wire to ZBT RAM signals

    assign      ram_we_b = ~we;
    assign      ram_clk = ~clk;     // RAM is not happy with our data hold
                                    // times if its clk edges equal FPGA's
                                    // so we clock it on the falling edges
                                    // and thus let data stabilize longer
    assign      ram_address = addr;

    assign      ram_data = we_delay[1] ? write_data_old2 : {36{1'bZ}};
    assign      read_data = ram_data;

endmodule
```

```
//////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Hex display driver
//
//
// File:   display_16hex.v
// Date:   24-Sep-05
//
// Created: April 27, 2004
// Author: Nathan Ickes
//
// This module drives the labkit hex displays and shows the value of
// 8 bytes (16 hex digits) on the displays.
//
// 24-Sep-05 Ike: updated to use new reset-once state machine, remove clear
// 02-Nov-05 Ike: updated to make it completely synchronous
//
// Inputs:
//
//   reset       - active high
//   clock_27mhz - the synchronous clock
//   data        - 64 bits; each 4 bits gives a hex digit
//
// Outputs:
//
//    disp_*     - display lines used in the 6.111 labkit (rev 003 & 004)
//
//////////////////////////////////////////////////////////////////////////////

module display_16hex (reset, clock_27mhz, data_in, disp_blank, disp_clock, disp_rs, disp_ce_b,
       disp_reset_b, disp_data_out);

    input reset, clock_27mhz;     // clock and reset (active high reset)
    input [63:0] data_in;         // 16 hex nibbles to display

    output disp_blank, disp_clock, disp_data_out, disp_rs, disp_ce_b, disp_reset_b;

    reg disp_data_out, disp_rs, disp_ce_b, disp_reset_b;

    //////////////////////////////////////////////////////////////////////////////
    //
    // Display Clock
    //
    // Generate a 500kHz clock for driving the displays.
    //
    //////////////////////////////////////////////////////////////////////////////

    reg [5:0] count;
    reg [7:0] reset_count;
    wire      dreset;
    wire      clock = (count<27) ? 0 : 1;

    always @(posedge clock_27mhz) begin
        count <= reset ? 0 : (count==53 ? 0 : count+1);
        reset_count <= reset ? 100 : ((reset_count==0) ? 0 : reset_count-1);
    end

    assign dreset = (reset_count != 0);
    assign disp_clock = ~clock;
    wire clock_tick = ((count==27) ? 1 : 0);

    //////////////////////////////////////////////////////////////////////////////
    //
    // Display State Machine
    //
    //////////////////////////////////////////////////////////////////////////////

    reg [7:0] state;          // FSM state
    reg [9:0] dot_index;      // index to current dot being clocked out
    reg [31:0] control;       // control register
    reg [3:0] char_index;     // index of current character
```

```
reg [39:0] dots;          // dots for a single digit
reg [3:0] nibble;         // hex nibble of current character
reg [63:0] data;

assign disp_blank = 1'b0; // low <= not blanked

always @(posedge clock_27mhz)
    if (clock_tick) begin
        if (dreset) begin
            state <= 0;
            dot_index <= 0;
            control <= 32'h7F7F7F7F;
        end
    else

    casex (state)
        8'h00: begin
            // Reset displays
            disp_data_out <= 1'b0;
            disp_rs <= 1'b0; // dot register
            disp_ce_b <= 1'b1;
            disp_reset_b <= 1'b0;
            dot_index <= 0;
            state <= state+1;
        end

        8'h01: begin
            // End reset
            disp_reset_b <= 1'b1;
            state <= state+1;
        end

        8'h02: begin
            // Initialize dot register (set all dots to zero)
            disp_ce_b <= 1'b0;
            disp_data_out <= 1'b0;    // dot_index[0];
            if (dot_index == 639)
                state <= state+1;
            else
                dot_index <= dot_index+1;
        end

        8'h03: begin
            // Latch dot data
            disp_ce_b <= 1'b1;
            dot_index <= 31;      // re-purpose to init ctrl reg
            state <= state+1;
        end

        8'h04: begin
            // Setup the control register
            disp_rs <= 1'b1; // Select the control register
            disp_ce_b <= 1'b0;
            disp_data_out <= control[31];
            control <= {control[30:0], 1'b0};    // shift left
            if (dot_index == 0)
                state <= state+1;
            else
                dot_index <= dot_index-1;
        end

        8'h05: begin
            // Latch the control register data / dot data
            disp_ce_b <= 1'b1;
            dot_index <= 39;      // init for single char
            char_index <= 15;     // start with MS char
            data <= data_in;
            state <= state+1;
        end

        8'h06: begin
```

```
            // Load the user's dot data into the dot reg, char by char
            disp_rs <= 1'b0;                    // Select the dot register
            disp_ce_b <= 1'b0;
            disp_data_out <= dots[dot_index];   // dot data from msb
            if (dot_index == 0)
                if (char_index == 0)
                    state <= 5;                 // all done, latch data
                else begin
                    char_index <= char_index - 1; // goto next char
                    data <= data_in;
                    dot_index <= 39;
                end
            else
                dot_index <= dot_index-1;       // else loop thru all dots
        end
    endcase
end

always @ (data or char_index)
    case (char_index)
        4'h0:      nibble <= data[3:0];
        4'h1:      nibble <= data[7:4];
        4'h2:      nibble <= data[11:8];
        4'h3:      nibble <= data[15:12];
        4'h4:      nibble <= data[19:16];
        4'h5:      nibble <= data[23:20];
        4'h6:      nibble <= data[27:24];
        4'h7:      nibble <= data[31:28];
        4'h8:      nibble <= data[35:32];
        4'h9:      nibble <= data[39:36];
        4'hA:      nibble <= data[43:40];
        4'hB:      nibble <= data[47:44];
        4'hC:      nibble <= data[51:48];
        4'hD:      nibble <= data[55:52];
        4'hE:      nibble <= data[59:56];
        4'hF:      nibble <= data[63:60];
    endcase

always @(nibble)
    case (nibble)
        4'h0: dots <= 40'b00111110_01010001_01001001_01000101_00111110;
        4'h1: dots <= 40'b00000000_01000010_01111111_01000000_00000000;
        4'h2: dots <= 40'b01100010_01010001_01001001_01001001_01000110;
        4'h3: dots <= 40'b00100010_01000001_01001001_01001001_00110110;
        4'h4: dots <= 40'b00011000_00010100_00010010_01111111_00010000;
        4'h5: dots <= 40'b00100111_01000101_01000101_01000101_00111001;
        4'h6: dots <= 40'b00111100_01001010_01001001_01001001_00110000;
        4'h7: dots <= 40'b00000001_01110001_00001001_00000101_00000011;
        4'h8: dots <= 40'b00110110_01001001_01001001_01001001_00110110;
        4'h9: dots <= 40'b00000110_01001001_01001001_00101001_00011110;
        4'hA: dots <= 40'b01111110_00001001_00001001_00001001_01111110;
        4'hB: dots <= 40'b01111111_01001001_01001001_01001001_00110110;
        4'hC: dots <= 40'b00111110_01000001_01000001_01000001_00100010;
        4'hD: dots <= 40'b01111111_01000001_01000001_01000001_00111110;
        4'hE: dots <= 40'b01111111_01001001_01001001_01001001_01000001;
        4'hF: dots <= 40'b01111111_00001001_00001001_00001001_00000001;
    endcase

endmodule
```

```
// Switch Debounce Module
// use your system clock for the clock input
// to produce a synchronous, debounced output

module debounce (reset, clock, noisy, clean);
    parameter DELAY = 270000;   // .01 sec with a 27Mhz clock
    input reset, clock, noisy;
    output clean;

    reg [18:0] count;
    reg new, clean;

    always @(posedge clock)
        if (reset) begin
            count <= 0;
            new <= noisy;
            clean <= noisy;
        end
        else if (noisy != new) begin
            new <= noisy;
            count <= 0;
        end
        else if (count == DELAY)
            clean <= new;
        else
            count <= count+1;

endmodule
```