3D Tetris

Charity Midenyo, Jenessa Rodriguez, Bradley Seymour

6.111

Professors Gim Hom and Joe Steinmeyer

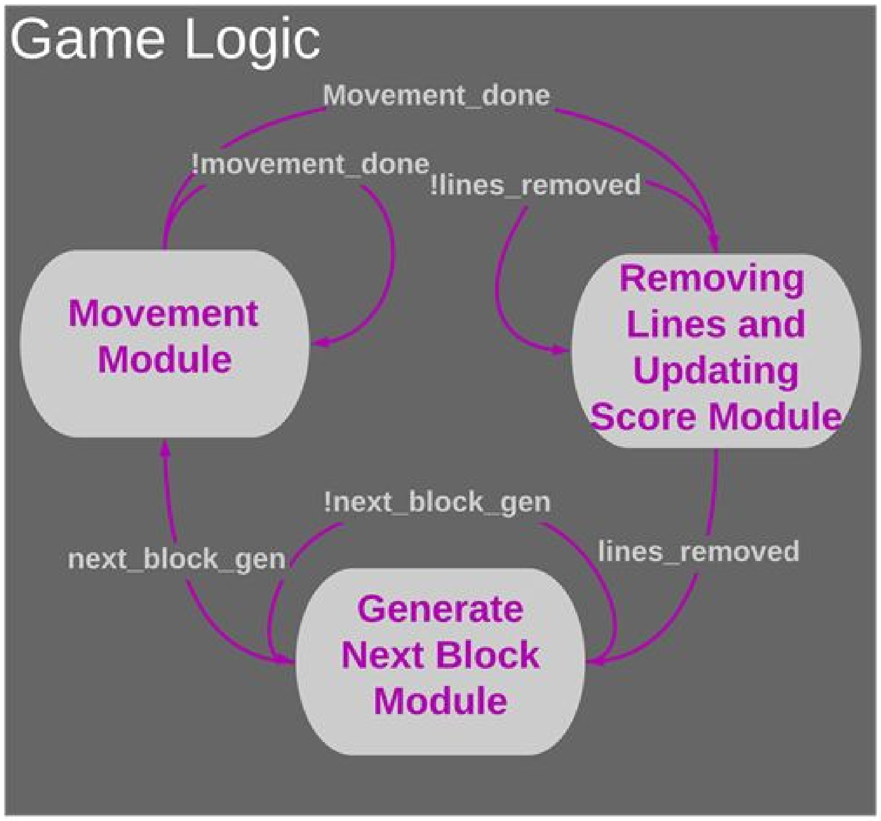
11 December, 2019

Introduction

35 years ago, the world was introduced to the Tetris . Since then, Tetris has grown exponentially, with all various conceivable versions having been created. One of the more rare versions of Tetris would be the 3D version. 3D Tetris was first introduced to the world in 1996 on Nintendo's Virtual Boy console. The failure of the Virtual Boy as a console notwithstanding, the game itself attempted to revolutionize the way Tetris was played. The next version of a 3D Tetris would be presented to the world as Tetrisphere. Released in 1997 for the Nintendo 64, Tetrisphere proved the concept of a 3D version of Tetris to be viable. Assuming an appropriate representation of the playing field and control system implemented by the developers, a new standard for how 3D Tetris should be played was born. Our team intends to create a version of 3D Tetris, utilizing these games as inspiration for our personal take on how 3D Tetris should look, play, and most importantly, feel.

Our 3D Tetris was broken into a couple main jobs, the 2D screen showing the score, the next block and title screen, translating the controller outputs into something the FPGA could read, the game logic, and the 3D implementation of the 2D playground. Charity Midenyo tackled the 2D screen and the controller outputs, Bradley Seymour worked on the 3D implementation, and Jenessa Rodriguez dealt with the game logic.

Game Module



Game Logic Overview

With the way that Tetris works, there's really three states of the game. The first state is when the next block goes from the next block screen to the playground. Then the new piece starts falling down. The last state is checking to see if there are any rows that need to be cleared and updating the score accordingly. Thus, this is the way I broke up the code. The main module of the game logic was a finite state machine that switched between the three states mentioned above. The three submodules called in the state machine were called Move, Remove, and Generate.

Generate Module

The first state that the Tetris game enters is the Generate module. While the game is not in the Generate module, this module should output the next block. The block that is shown is calculated through a pseudo random number generator module. I say pseudo because randomness doesn't exist in the digital world. In order to mimic randomness, I had a counter count up every clock cycle, and I pulled the value of the counter after a variable number of clock cycles. Once the main finite state machine was in the Generate state, the state of the Generate module switched from showing the next block in the next block square to the side of the playground and added the block to the top of the playground. The next value was then grabbed from the pseudo RNG module and sent it to the next block square. Once the next block was showing, a flag was raised and the main module switches to Movement state.

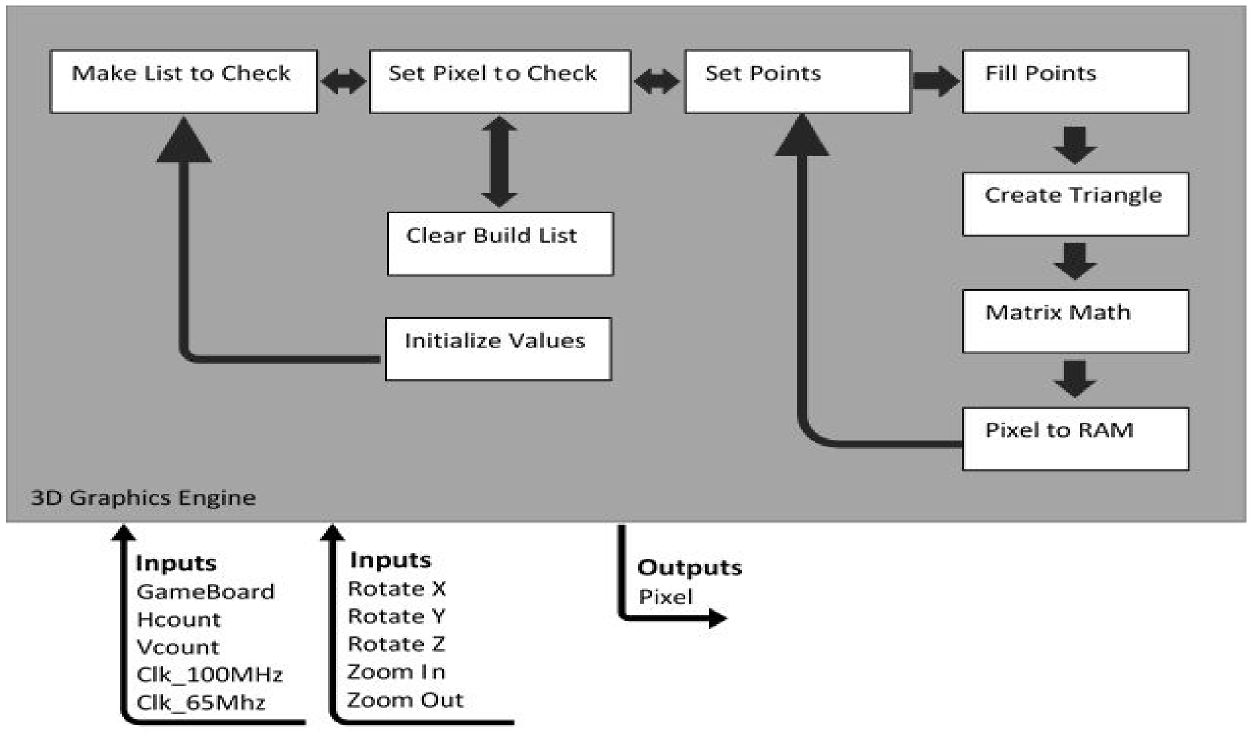
Movement Module

The Movement module deals moving the block down, left, and right, and rotating left and right, and when the block should stop moving left, right, and down. Once the main module is in the movement state, the movement module calls the Moving Down module. The Moving Down module counts up to one second, and once it has been one second the playground is shifted down. In order to check to see which blocks should be moved down, I created another list that contains all of the blocks that shouldn't be moving, which is updated after the new block has stopped moving. This list is also how I checked to see if the block should stop moving down; if the block that is falling is touching a value in the extra list, then it should stop moving. While counting, the Moving Down module calls the moving left, moving right, rotating left, and rotating right modules. When the block isn't moving down, it is free to move left, move right, rotate left, or rotate right. The moving left and moving right module is just a simple shift of the values left and right. Unfortunately, I couldn't figure out a way to generalize the rotation, so it would have ended up as a giant finite state machine with each rotation hardcoded, had I had the time to code it. Once the falling block touches a value in the other list, the other list is updated and a flag is raised. Once the flag is raised, the main finite state machine switches to the Remove state.

Remove Module

The next and final module is the for the Remove state, which is another fsm, as there are three things that need to be done in this state: removing the lines, shifting all the lines down, and updating the score accordingly. In the removing lines state of the Remove module, I checked to see if there are lines that are full of values, and if there is, I update the playground and the list full of the stationary values. Once I've removed all the lines that I need to remove, the state switches to the move lines down state. In this state, I checked to see if there is a line that is empty, and I shift all the values above that line down one line. Once I checked all 20 lines, I switch states to the update score state. The update score takes the current score and the number of lines removed and outputs the new score. Once the score is updated, the module raises a flag and the main finite state machine switches to the Generate module. This then repeats until the top line of the playground is occupied by blocks.

3D Module



Unlike normal 3D engines which are fed a mesh of triangles required to create an object, the engine needed to take the gameboard and directly extrapolate how many pieces to make and where to make them. To accomplish this, a large FSM was created to control the flow of the graphics engine. On first launch, an initialization state would occur to ensure all required values and arrays were set to zero prior to calculating a 3D game board. The next stage would take the game board array passed in from the Game Module and begin searching the game board for filled locations in order to create a more manageable list of calculations. This list expanded the 3bit encoding of which piece was in each block to a 3 byte encoding. The bit encoding is as follows, {Filled/Empty Flag, Top, Bottom, East, West, 10bit X Location, 9 bit Y location}. The Top, Bottom, East and West flags indicate if the neighboring block is present. These flags were setup to create a rudimentary pruning algorithm in order to minimize the number of triangles for which calculations were required. These flags were inadvertently abandoned during the coding process. After creating the short list of blocks to create, the FSM started at the top left of the board, working pixel by pixel, determining if a block would be present at that location. To accomplish this, the process of drawing a block was utilized to determine the 12 bit RGB color value that would be present This method was chosen due to future uncertainty of the boards location and rotation within the 240x480 pixel play field. A 3bit by 115200 depth BRAM was created to hold the calculated value returned for each pixel. The engine was then further developed to display the values calculated and stored in the BRAM following the completion of all calculations. This step never achieved success due to reasons discussed within the challenges section of this report. After determining a value for each pixel within the 240 x 480 play area, a final stage of the FSM was called in order to clear the shortened list of block requiring calculation in preparation for a new list to be created.

During the late stages of the design process, concessions were made in order to favor a potential working, yet less featureful final product. The major concession during this time was the rejection of rotation and the ability to zoom in and out of the game board. The overall depth of the game board was fixed at 10 units, and the rotation was discarded altogether.

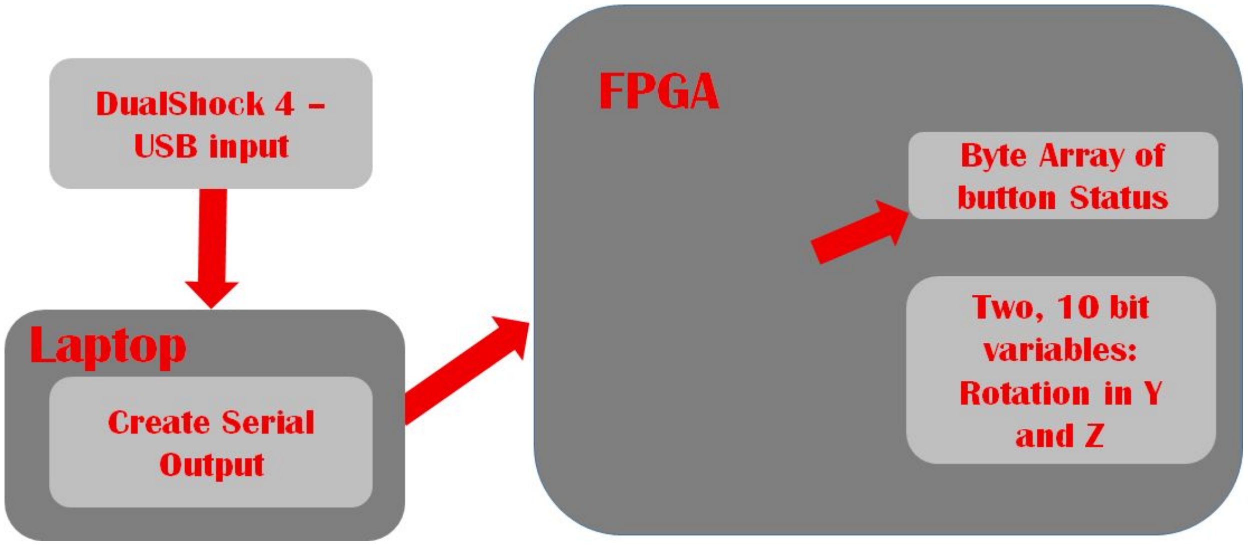
Triangle Calculation

Each cube was to represent a portion of a Tetromino. These cubes were broken into 12 triangles, 2 representing each face. These triangles were mapped in 3D space and required projection from 3D space onto 2D space for display. The Triangles were packed into 12 independent 2D arrays. Each array, representing a single triangle, contained 3 sets of coordinates, the X Location, Y Location, and the Z Location. These 12 arrays containing each triangle were sent in parallel to the Matrix module. This module would determine the values for Field of View, Aspect Ratio, and the correction factor for the users perspective, designated by Znear, the distance from the users eyes to the screen, and Zfar, the value representing the farthest visible point within the 3D field. These values were used to create 12 new arrays of modified triangles representing their projection from 3D space onto 2D space. During the design stage, the decision was made to use fixed point decimals to represent the necessary precision of calculation.   Additionally, the division operations that are required to map 3D objects to 2D space were precalculated and hard coded into the system. This choice was made as the user would not be moving into or out of the frame of view, they would simply rotate the playing field about each axis. By removing these calculations from the Matrix module, the number of cycles required to modify each triangle was significantly reduced. Moving forward, it is possible to completely negate some of these calculations, such as the users perspective correction factor, which lies relatively close to one, and could ultimately be removed for simplicity of calculation.

Field of View

It was determined necessary to calculate the users field of view, as this would create the effect of zooming into or out of the playing field. These calculations were again quite time intensive as they are represented by sinusoidal functions. To decrease latency from calculations, these values were predetermined and hardcoded into the Field of View module as a lookup table.

Serial Input



The serial input is made up of two parts a python script and a verilog module. We used a DualShock4 controller for our project. The controller transmits over the usb protocol. The controller was connected to a computer, which was running the python script. The python script used python's serial, struc and pygame modules. The pygame module was used to parse the controller input into values that could be sent as a bytes of data in a serial communication with an fpga. The scruc and serial was used to create and send byte arrays to send to the fpga.

We used a FT2232H chip, to send the serial data from the computer to the fpga. The uart_input module, proceed the serial input into variables that other modules could use.

The api for the serial communications were five bytes of data. The first byte represented pressed on not pressed for eight buttons on the controller. The other four, bytes were a magnitude rom -128 to 127, representing rotational motion of the board in 3 direction as well as as a zoom. The serial module would take in the serial input and generate the outputs to the modules.

2D display Module

- tetris_board - This module displays the 2d array of the falling tetris blocks. It takes in a 2d array as an input and outputs the correct pixel color for the blocks in the grid
- next_block - displays the next block, to the right of the tetris board, in a similar fashion to how display_board works

- get_index - A helper for display board, uses the values of vcount and hcount to find the corresponding, index in the tetris array. With this we could then find the correct color to display
- get_relative_index - The display is broken up into a grid. Each block in the grid is a smaller pixel grid. This method find the relative x and y coordinates of vcount and hcount within the smaller pixel grid
- title_screen- displays an image of teris
- top_level_ display - Deals with the overlay priority of all of the different modules, on the background grid S

All together the to module shows a 2d tetris board as well as a the next block in an aesthetic manner.

Challenges

Game Module

Another one of the challenges was how the Movement module should be implemented. My first attempt was to call all the movement submodules (moving down, moving right, moving left, rotating right, and rotating left). However, because moving down needed to be constantly called, it made it hard to mux the outputs of each submodule. What I ended up doing was that once the main module was in the movement state, the moving down module was called. Once the moving down module was called, if the counter was still counting, then the move left, move right, rotate left, and rotate right submodules were called (exclusively, in order to prevent assigning the new playground more than one value). If the counter had reached one second, none of the submodules would be able to assign an output, and the block moved down one line.

Another one of the challenges (HOORAY REPETITION) was the RNG module. I've had the idea to use a counter to pick a pseudo random value, but it's implementation was a challenge from the start. My first attempt was to use combinational logic to add to the counter, but since everything else was sequential logic, using combinational logic would just give me multiple values for the next block value within a clock cycle, which would have given the value multiple drivers and a non viable value. I then tried to use sequential logic to grab a value from the RNG module, but that proved to be glitchy as well, although I'm still not quite sure why. The first value would be fine and not glitchy, but any value after that would be alternating violently, as if there were multiple drivers. In the end, I was not able to find an implementation that worked, and would like to continue to explore ways that it does work.

One of the most tedious part of my code would have definitely been the rotation module. Hard coding every rotation for every block switching between different states of the block would have been absolutely awful. After I did some very intensive research trying to find the code for the Tetris code translated into Verilog, I found that rotation could be accomplished by XOR- ing each block with a certain value to rotate in a specific direction. This would still require a huge finite state machine to XOR each block with its specific value for that rotation direction, but it is definitely better than hard coding each rotation and switching between each rotation. I would love to put in the time and effort to try and find a more general way to rotate each block, but I unfortunately did not have the time to put any more time than I already did to trying to find a way to generalize it.

The biggest challenge I faced was that I couldn't get the main fsm to work properly for more than once cycle. The first block would generate and move down to the bottom of the screen perfectly fine, but once the next block was generated, the main fsm switched through the states every few clock cycles, despite the Movement state needing many more than a couple of clock cycles to complete. In the end, I was not able to figure out why the flag for the Movement module was being raised too rapidly. I made sure that flag being raised from the moving down module was specific and was being lowered on the next clock cycle, and the module worked perfectly fine the first time the block was moved down. This is also something I would like to continue to work on and try to find a solution as to why this main finite state machine was not working properly.

3D Module

Two separate methods were ultimately attempted during the creation process of the 3D module. The first method proved to be ineffective. This method attempted mirror a software implementation by creating massive parallel pipelines of data such that the entire game board could be calculated on the fly. These calculations looked to take the location of the game board as fixed, and determine each triangle for display. The goal was to make a large quantity of sprites, each one representing a triangle which had been projected from 3D space onto 2D space. After creating a 3D board on the screen, the user would then have been required to manually center the playing field within the allotted space on the screen. This method represented a true 3D implementation complete the ability to move about a 3D environment. Many of the details incorporated into the final design originated from this first approach. The largest difficulty in calculating a complete 3D field on the fly exists from a pure lack of resources on the FPGA. There existed an insufficient number of DSP modules to perform the mathematical operations required to calculate such vast quantities of data within a single or even several clock cycles. Additionally, due to the use of 2 and 3 byte numbers, this created a requirement for upto 40 bit numbers to represent the results of these calculations, the DSP modules on the FPGA were already pushed to, and sometimes past, their limits, requiring the software to attempt to create larger DSP modules from individual logic slices to handle these calculations. Following the shift to a FSM controlled 3D engine, further difficulty was found when attempting to prune the game board into smaller, easier to calculate chunks. These difficulties produced synthesis run times in excess of an hour per run due to asking Vivado to create distributed ram throughout the FPGA in order to store these values. Additional difficulties were discovered during the implementation of the matrix mathematics module. It is believed that this module was the root of many problems in the final hours leading up to the final project checkoff. The first of many problems stemmed from initialization of the arrays after programming the FPGA. This was resolved by creating a reset initialization as well as pruning the 4x4 matrix to a 1x6 vector to increase compile time and readability of the code. After ensuring the matrix math module was computing a projection at all, it is believed that this module due to attempting fixed point decimal operations was failing to produce numbers larger than zero, thus any useful output. This stems from a lack of data return from the BRAM during the display phase of operations.

Triangle Projection

Several difficulties were encountered while attempting to implement the triangle projection and drawing system. The first and simplest to solve was drawing a triangle on the screen. After some searching, a rasterization algorithm was found allowing for efficient calculations and drawing of a triangle within a single Vclock cycle. This algorithm exploited the normals of the vectors comprising each leg of a single triangle to minimize repeat calculations of a pixel within the triangle shape. This was the first step towards attempting to create the now defunct on the fly calculations of the 3D engine.

Unknowingly, finding a way to pack a 3D array with triangles proved to be quite troublesome. The initial attempts worked to utilize C structs as a basis for which to create a vector representing a single vertex, then a vector of vertices to represent a triangle, and finally an array of triangles. Eventually, it was discovered that without the ability to use pointers, creating a nested struct resembling something like a linked list was next to impossible. This ultimately led the decision to calculate the 12 triangles for a single block at one time.

Setting the Pixel to Check

After abandoning on the fly calculation, a more methodical approach of checking each pixel was attempted. This required creating two counters to track the x and y location of each pixel calculation. It was discovered, a few hours before the final checkoff, that this approach was again computationally expensive. An oversight occurred in these final hours of the method created earlier to minimize the number of triangles to calculate. Had the flags representing the top, bottom, east, and west faces been utilized, this could have potentially increased the speed with which a single block could be computed, due to potentially parallel computation of neighboring blocks. Looking back, an even better approach could have been to determine how large a chunk of the screen was filled, thus only requiring several large triangles to represent the block of tetrominoes a player had stacked up. The method of checking each block on the playing field, even with only ten blocks hard coded as a testing platform, proved to take an exorbitant amount of time. Utilizing XVGA timing standards, each frame was required to be calculated and ready for drawing within 16.7ms. Computing approximately 4 horizontal lines of 240 pixels took around 16.7ms. This would place an entire game board calculation time around two seconds per run, completely unacceptable for a smooth, real time graphics engine.

Serial Input

For this project the plan was to use a dualshock 4 controller, which is a Playstation controller to play Tetris and rotate the board. Initially I wanted to attach the controller to the FPGA via a usb type a cable. I wasn't sure what kind of data the controller would be sending so I looked it up. When I wrote the verilog code and connected it to the FPGA, there was no signal from the controller. After a few days of troubleshooting I gave up and went with an idea that Joe suggested.

The new plan is to connect the controller to my laptop and send a serial signal from my laptop, to the FPGA using a FT2232H chip (same as from lab 2), them parse the serial input to get the button press data. The controller didn't interface with my computer properly so I found a driver that would work on linux, called ds4drv. From there I downloaded ds4drv a dualshock 4 driver for linux and I wrote a python script, using parts from the code in lab two and tutorial information from pygame.

My laptop died in the days leading up to the project and had to recalibrate my python script so a last minute challenge I didn't anticipate was having to edit my code to work on a windows machine.

2D display Module

The function of the 2d display module was to have a working 2d display for debugging early on in the project and also to have a rudimentary game working if the 3d display did not end up working. The challenge with this module was that I had to process the input from a 2d verilog array into a visual grid. My initial idea was to create a Tetris block and pass the location and color of the block but this did not end up working because, it of the timing of the strobe. It then became clear that I needed a way to explicitly map each box to it corresponding spot in the original array. I created this in combinational logic and form the location of Vcount and Hcount from the xvga module I could figure out the correct index array to output and therefore the correct color of the pixels.

The biggest challenge was the time I had left and also some misfortune. In the few days leading up to the deadline I was out of campus. I had planned to spend all of my free time working on the project but unfortunately my laptop completely stopped working and I could not make the progress I intended to over those few days. I addition, I had not backed up my module file yet which meant that we had to remove the harddrive from laptop to access the information. As a result of all of that I was behind schedule on the project in terms of coding the modules and creating coe files.

Advice

In general, a well oiled machine is equivalent to a functional group dynamic. Some of the ways to achieve this stem from ensuring routine meetings within the group aside from the required meetings with the project advisor.   Other methods to achieve this type of group function stem from working together, or at a minimum, aside each other during lab hours. This method of working on the project helps to open lines of communication between group members simply by sitting next to each other. Had our group practiced this better, it would have ensured we were all on the same page regarding overall project design and work flow from the highest levels of abstraction all the way down to the nitty gritty details such as bit depths and organization within modules. This would have also worked to instill a stronger sense of responsibility to the group and the project creation.

One piece of advice I would give to future students in terms of the game logic would be to try to isolate the falling block from the playground. One of the issues that kept coming up was that I didn't necessarily know where the block was, so I had to make each process increasing general for each movement. Being able to find the block would make moving down, checking the boundaries, and rotating left and right significantly easier to implement. Working with a four by four matrix containing the block allows for more specific manipulation of the block and may remove the need for a list containing the stationary block altogether. I would definitely be harder to code initially, but the rest of the movement would be significantly easier.

When attempting to tackle a more complex project, such as the 3D engine, it is recommended that research into algorithms that are easily implementable on an FPGA should be conducted. There exists many articles relating to the process of implementing a 3D engine, and with the rising popularity of FPGAs in the academic as well as hobby sectors, papers and articles surface daily to help you along the path towards a functioning product. Other resources outside of the class such as Reddit's r/FPGA are also treasure troves of knowledge and experience from industry experts, academics, and hobbyists that should not readily be dismissed, more likely than not, someone has run into your problem, and can provide guidance on a path forward to a functioning solution.

Appendix A: Game Module Verilog

RNG Module:

```
`timescale 1ns / 1ps

module rng_module(
        input clock,
        input less_go,
        input reset,
        output logic [2:0] value,
        output logic done
        );
logic [2:0] count;
always_ff @(posedge clock)
        begin
                if (reset)
                        begin
                                count <= 3'b0;
                                value <= 3'b0;
                                done <= 1'b0;
                        end
                else if (less_go)
                        begin
                                value <= count;
                                done <= 1'b1;
                        end
                else
                        begin
                                count <= count + 1;
                                done <= 1'b0;
                        end
        end
endmodule
```

Appendix B: 2D Graphics Verilog

```
module display_top_level(   input clk_100mhz,
        input[15:0] sw,
        input btnc, btnu, btnl, btnr, btnd,
        input logic [1:0]  jb,

        output[3:0] vga_r,
        output[3:0] vga_b,
        output[3:0] vga_g,
        output vga_hs,
        output vga_vs,
        output led16_b, led16_g, led16_r,
        output led17_b, led17_g, led17_r,
        output[15:0] led,
        output ca, cb, cc, cd, ce, cf, cg, dp,  // segments a-g, dp
        output[7:0] an          // Display location 0-7
        );
        logic clk_65mhz;
        logic blank;
        logic reset;

        // create a clock divider
        clk_wiz_0 clkdivider(.clk_in1(clk_100mhz), .clk_out1(clk_65mhz), .reset(0));
        synchronize #(.NSYNC(3)) rst (.clk(clk_65mhz), .in(btnc), .out(reset));

        logic ready;
        logic [7:0] controls, x_rotation, y_rotation, z_rotation, zoom;
        logic [7:0] data_blast[4:0];
        logic[31:0] time_count;
        logic [1:0] falling_edge;
        logic reading_data;
        logic bit_in = jb[1];


        wire [31:0] data;      //  instantiate 7-segment display; display (8) 4-bit hex
        wire [6:0] segments;
        assign {cg, cf, ce, cd, cc, cb, ca} = segments[6:0];
        display_8hex display(.clk_in(clk_65mhz),.data_in(data), .seg_out(segments), .strobe_out(an));

        assign  dp = 1'b1;  // turn off the period

//      assign led = sw;                // turn leds on
        // display 0123456 + sw[3:0]
        assign led16_r = btnl;          // left button -> red led
        assign led16_g = btnc;                  // center button -> green led
        assign led16_b = btnr;                  // right button -> blue led
        assign led17_r = btnl;
        assign led17_g = btnc;
        assign led17_b = btnr;

        wire [10:0] hcount;   // pixel on current line
        wire [9:0] vcount;     // line number
        wire hsync, vsync;
        wire [11:0] board_out, next_block_out, background_out, title_out, next_blob_out, tetris_title_out;//, score_out;
        reg [11:0] rgb;
        xvga xvga1(.vclock_in(clk_65mhz),.hcount_out(hcount),.vcount_out(vcount),
        .hsync_out(hsync),.vsync_out(vsync),.blank_out(blank));

        logic [11:0] score_out = 12'h111;


        wire phsync,pvsync,pblank;
        logic [7:0] game_level = 0;                 // level
        logic [9:0] [2:0] block_colors[19:0];
        logic [9:0] [2:0] next_block[19:0];

//logic [9:0] [2:0] block_colors[19:0] = {{3'd4, 3'd1, 3'd1, 3'd1, 3'd1, 3'd1, 3'd1, 3'd1, 3'd1, 3'd1},
//                                      {3'd3, 3'd4, 3'd3, 3'd3, 3'd3, 3'd3, 3'd3, 3'd3, 3'd3, 3'd3},
//                                      {3'd1, 3'd1, 3'd4, 3'd1, 3'd1, 3'd1, 3'd1, 3'd1, 3'd1, 3'd1},
//                                      {3'd3, 3'd3, 3'd3, 3'd4, 3'd3, 3'd3, 3'd3, 3'd3, 3'd3, 3'd3},
//                                      {3'd1, 3'd1, 3'd1, 3'd1, 3'd4, 3'd1, 3'd1, 3'd1, 3'd1, 3'd1},
//                                      {3'd3, 3'd3, 3'd3, 3'd3, 3'd3, 3'd4, 3'd3, 3'd3, 3'd3, 3'd3},
//                                      {3'd1, 3'd1, 3'd1, 3'd1, 3'd0, 3'd1, 3'd4, 3'd1, 3'd1, 3'd1},
//                                      {3'd3, 3'd3, 3'd3, 3'd3, 3'd1, 3'd3, 3'd3, 3'd4, 3'd3, 3'd3},
//                                   {3'd1, 3'd1, 3'd1, 3'd1, 3'd2, 3'd1, 3'd1, 3'd1, 3'd4, 3'd1},
//                                   {3'd3, 3'd3, 3'd3, 3'd3, 3'd3, 3'd3, 3'd3, 3'd3, 3'd3, 3'd4},
//                                   {3'd1, 3'd1, 3'd1, 3'd1, 3'd4, 3'd1, 3'd1, 3'd1, 3'd1, 3'd5},
//                                   {3'd3, 3'd3, 3'd3, 3'd3, 3'd5, 3'd3, 3'd3, 3'd3, 3'd5, 3'd3},
//                                   {3'd1, 3'd1, 3'd1, 3'd1, 3'd6, 3'd1, 3'd1, 3'd5, 3'd1, 3'd1},
//                                   {3'd3, 3'd3, 3'd3, 3'd3, 3'd7, 3'd3, 3'd5, 3'd3, 3'd3, 3'd3},
//                                   {3'd1, 3'd1, 3'd1, 3'd1, 3'd1, 3'd5, 3'd1, 3'd1, 3'd1, 3'd1},
//                                   {3'd3, 3'd3, 3'd3, 3'd3, 3'd5, 3'd3, 3'd3, 3'd3, 3'd3, 3'd3},
//                                   {3'd1, 3'd1, 3'd1, 3'd7, 3'd7, 3'd7, 3'd1, 3'd1, 3'd1, 3'd1},
//                                   {3'd3, 3'd3, 3'd5, 3'd7, 3'd7, 3'd7, 3'd3, 3'd3, 3'd3, 3'd3},
//                                   {3'd1, 3'd5, 3'd1, 3'd7, 3'd7, 3'd7, 3'd1, 3'd1, 3'd1, 3'd1},
//                                   {3'd5, 3'd3, 3'd3, 3'd7, 3'd7, 3'd7, 3'd3, 3'd3, 3'd3, 3'd3}};


        logic [23:0] player_score;
        logic [2:0] state;
        logic done_with_lines;
        logic done_moving;
        logic done_showing;
        logic [2:0] old_value;
        logic [2:0] new_value;

        tetris_board dis(
        .clk_in(clk_65mhz),
        .rst_in(reset),
        .block_colors(block_colors),     // colors of the block
        .hcount_in(hcount),.vcount_in(vcount),
        .hsync_in(hsync),.vsync_in(vsync),.blank_in(blank),
        .phsync_out(phsync),.pvsync_out(pvsync),.pblank_out(pblank),
        .tetris_out(board_out));           // pixel  // r=11:8, g=7:4, b=3:0

        next_block_mod next(
        .clk_in(clk_65mhz),
        .rst_in(reset),
        .block_colors(block_colors),
        .hcount_in(hcount),.vcount_in(vcount),
        .hsync_in(hsync),.vsync_in(vsync),.blank_in(blank),
        .phsync_out(phsync),.pvsync_out(pvsync),.pblank_out(pblank),
        .next_block_out(next_block_out));          // pixel  // r=11:8, g=7:4, b=3:0

        title_screen screen(
        .clk_in(clk_65mhz),
        .rst_in(reset),
        .hcount_in(hcount),.vcount_in(vcount),
        .hsync_in(hsync),.vsync_in(vsync),.blank_in(blank),
        .phsync_out(phsync),.pvsync_out(pvsync),
        .pblank_out(pblank),.title_out(title_out));

        logic [4:0] x_relative;
        logic [4:0] y_relative;
        get_relative_index(.x_in(hcount), .y_in(vcount), .clk_in(clk_65mhz), .x_relative(x_relative), .y_relative(y_relative));
        draw_block box(.x_in(x_relative), .y_in(y_relative), .color_in(12'h222), .clk_in(clk_65mhz), .color_out(background_out));


        wire border = (hcount==0 | hcount==1023 | vcount==0 | vcount==767 |
        hcount == 512 | vcount == 384);
        parameter SIZE = 23;
        parameter y_min = 100;
        parameter x_min = 200;
        parameter y_max = 580;
        parameter x_max = 440;

        parameter y_min_next = 268;
        parameter x_min_next = 512;
        parameter y_max_next = 364;//4
        parameter x_max_next = 584;//3

        parameter y_min_score = 100;
        parameter x_min_score = 512;
        parameter y_max_score = 224;
        parameter x_max_score = 944;

        next_block_blob puck1(  .clk_in(clk_65mhz),
                .x_in(x_min_score),
                .y_in(y_min_score),
                .hcount_in(hcount),
                .vcount_in(vcount),
                .pixel_out(next_blob_out));


        reg b,hs,vs;
        always_ff @(posedge clk_65mhz) begin
        hs <= hsync;
        vs <= vsync;
        b <= blank;

        if (sw[1:0] == 2'b01) rgb <= {12{border}}; // 1 pixel outline of visible area (white)
        else if (sw[1:0] == 2'b10) rgb <= {{4{hcount[8]}}, {4{hcount[7]}}, {4{hcount[6]}}} ;  // color bars
        else if (sw[1:0] == 2'b11) rgb <= title_out;
//      else if (vcount  == y_min | vcount ==  y_max | hcount == x_min | hcount == x_max) rgb <= 12'h333;
        else if (vcount > y_min && vcount< y_max && hcount > x_min && hcount < x_max) rgb <= board_out;
        else if (vcount > y_min_next && vcount< y_max_next && hcount > x_min_next && hcount < x_max_next) rgb <=
next_block_out;
        else if (vcount > y_min_score && vcount< y_max_score && hcount > x_min_score && hcount < x_max_score) rgb <=
next_blob_out;
        else if (vcount > 12'd3 && vcount < 12'd749 && hcount > 12'd7 && hcount < 12'd1017 ) rgb <= background_out;
//      else if (vcount > 12'd604 && vcount< y_max_score && hcount > x_min_score && hcount < x_max_score) rgb <=
next_blob_out;
        else rgb <= 12'h000;
        end

        // the following lines are required for the Nexys4 VGA circuit - do not change
        assign vga_r = ~b ? rgb[11:8]: 0;
        assign vga_g = ~b ? rgb[7:4] : 0;
        assign vga_b = ~b ? rgb[3:0] : 0;

        assign vga_hs = ~hs;
        assign vga_vs = ~vs;

endmodule


module tetris_board(         input clk_in,
                input rst_in,
                input [9:0] [2:0] block_colors [19:0],         // colors of the block
                input [10:0] hcount_in,             // horizontal index of current pixel (0..1023)
                input [9:0]  vcount_in,             // vertical index of current pixel (0..767)
                input hsync_in,                     // XVGA horizontal sync signal (active low)
                input vsync_in,                     // XVGA vertical sync signal (active low)
                input blank_in,                     // XVGA blanking (1 means output black pixel)

                input [11:0] x_min,
                input [11:0] y_min,

                output phsync_out,                  // horizontal sync
                output pvsync_out,                  // vertical sync
                output pblank_out,                  // blanking
                output logic [11:0] tetris_out,   // pixel  // r=11:8, g=7:4, b=3:0
                output logic [8:0] index
        );

        logic [11:0] pixel_out;
        logic [4:0] block_size = 5'd24;


        logic [11:0] x_idx;
        logic [11:0] y_idx;
        logic [4:0] x_relative;
        logic [4:0] y_relative;
        get_index idx( .x_in(hcount_in), .y_in(vcount_in), .x_idx(x_idx), .y_idx(y_idx));
        get_relative_index(.x_in(hcount_in), .y_in(vcount_in), .clk_in(clk_in), .x_relative(x_relative), .y_relative(y_relative));
        draw_block box(.x_in(x_relative), .y_in(y_relative), .color_in(pixel_out), .clk_in(clk_in), .color_out(tetris_out));


        logic [2:0] color = block_colors[19- y_idx][9 - x_idx];

        always_comb begin
        case (color)
        3'd0: pixel_out = 12'h000;
        3'd1: pixel_out = 12'h0CC;//i
        3'd2: pixel_out = 12'h707;//t
        3'd3: pixel_out = 12'h0C0;//S
        3'd4: pixel_out = 12'hC00;//Z
        3'd5: pixel_out = 12'hC60;//L
        3'd6: pixel_out = 12'h00C;//J
        3'd7: pixel_out = 12'hCC0;//0
        endcase
        end



endmodule

module get_relative_index(  input clk_in,
                input[11:0] x_in,
                input[10:0] y_in,
                output logic [4:0] x_relative,
                output logic [4:0] y_relative);
  parameter size = 23;
  parameter y_min = 4;
  parameter x_min = 8;
  logic [10:0] old_y_in;

        always_ff @(posedge clk_in)begin

        if (y_in < y_min) begin
        y_relative <= 5'd0;
        end else if (y_relative >= size  && old_y_in < y_in) begin
        y_relative <= 5'd0;
        end else if (old_y_in < y_in) begin
        y_relative <= y_relative + 5'd1;
        end

        old_y_in <= y_in;

        if (x_in <= x_min) begin
        x_relative <= 5'd0;
        end else if (x_relative >= size) begin
        x_relative <= 5'd0;
        end else begin
        x_relative <= x_relative + 5'd1;
        end
        end

endmodule

module get_index( input[11:0] x_in,
        input[10:0] y_in,
        output logic[8:0] x_idx,
        output logic [4:0] y_idx
        );
        parameter y_min = 100;
        parameter x_min = 200;
        parameter y_max = 580;
        parameter x_max = 440;
        parameter size = 23;
        logic[8:0] temp;
        logic[8:0] temp1;


        always_comb begin
        if (y_in < y_min) temp = 8'd0;
        else if (y_in < y_min + 12'd24) temp = 8'd0;
        else if (y_in < y_min + 12'd48) temp = 8'd1;
        else if (y_in < y_min + 12'd72) temp = 8'd2;
        else if (y_in < y_min + 12'd96) temp = 8'd3;
        else if (y_in < y_min + 12'd120) temp = 8'd4;
        else if (y_in < y_min + 12'd144) temp = 8'd5;
        else if (y_in < y_min + 12'd168) temp = 8'd6;
        else if (y_in < y_min + 12'd192) temp = 8'd7;
        else if (y_in < y_min + 12'd216) temp = 8'd8;
        else if (y_in < y_min + 12'd240) temp = 8'd9;
        else if (y_in < y_min + 12'd264) temp = 8'd10;
        else if (y_in < y_min + 12'd288) temp = 8'd11;
        else if (y_in < y_min + 12'd312) temp = 8'd12;
        else if (y_in < y_min + 12'd336) temp = 8'd13;
        else if (y_in < y_min + 12'd360) temp = 8'd14;
        else if (y_in < y_min + 12'd384) temp = 8'd15;
        else if (y_in < y_min + 12'd408) temp = 8'd16;
        else if (y_in < y_min + 12'd432) temp = 8'd17;
        else if (y_in < y_min + 12'd456) temp = 8'd18;
        else if (y_in < y_min + 12'd480) temp = 8'd19;
        else temp = 8'd0;


        if (x_in < x_min) temp1 = 8'd0;
        else if (x_in < x_min + 12'd24) temp1 = 8'd0;
        else if (x_in < x_min + 12'd48) temp1 = 8'd1;
        else if (x_in < x_min + 12'd72) temp1 = 8'd2;
        else if (x_in < x_min + 12'd96) temp1 = 8'd3;
        else if (x_in < x_min + 12'd120) temp1 = 8'd4;
        else if (x_in < x_min + 12'd144) temp1 = 8'd5;
        else if (x_in < x_min + 12'd168) temp1 = 8'd6;
        else if (x_in < x_min + 12'd192) temp1 = 8'd7;
        else if (x_in < x_min + 12'd216) temp1 = 8'd8;
        else if (x_in < x_min + 12'd240) temp1 = 8'd9;
        else temp1 = 8'd0;
        x_idx = temp1;
        y_idx = temp;
        end
Endmodule

module draw_block( input[4:0] x_in,
        input[4:0] y_in,
        input [11:0] color_in,
        input clk_in,
        output logic [11:0] color_out
        );


        always_comb begin
        if (x_in == 5'd0 | x_in == 5'd23 | y_in == 5'd0 | y_in == 5'd23) color_out = 12'h000;
        else if ((x_in == 5'd4 | x_in == 5'd19) && (y_in > 5'd3 && y_in < 5'd20)) color_out = 12'h000;
        else if ((y_in == 5'd4 | y_in == 5'd19) && (x_in > 5'd3 && x_in < 5'd20)) color_out = 12'h000;
        else if ((y_in < 5'd4 | y_in > 5'd19) && (x_in < 5'd4 | x_in > 5'd19) && (x_in == y_in)) color_out = 12'h000;
        else if ((y_in < 5'd4 | y_in > 5'd19) && (x_in < 5'd4 | x_in > 5'd19) && (x_in + y_in == 5'd23)) color_out = 12'h000;
        else color_out = color_in;
        end

Endmodule

`timescale 1ns / 1ps

module next_block(        input clk_in,
            input rst_in,
            input [9:0] [2:0] block_colors [19:0],        // colors of the block
            input [10:0] hcount_in,        // horizontal index of current pixel (0..1023)
            input [9:0]  vcount_in,        // vertical index of current pixel (0..767)
            input hsync_in,                // XVGA horizontal sync signal (active low)
            input vsync_in,                // XVGA vertical sync signal (active low)
            input blank_in,                // XVGA blanking (1 means output black pixel)

            input [11:0] x_min,
            input [11:0] y_min,
            output phsync_out,                  // horizontal sync
            output pvsync_out,                  // vertical sync
            output pblank_out,                  // blanking
            output logic [11:0] next_block_out,          // pixel  // r=11:8, g=7:4, b=3:0
            output logic [8:0] index
        );



        logic [11:0] pixel_out;
        logic [4:0] block_size = 5'd24;

        logic [4:0] x_idx;
        logic [4:0] y_idx;
        logic [4:0] x_relative;
        logic [4:0] y_relative;
        get_next_index idx(.x_in(hcount_in), .y_in(vcount_in), .x_idx(x_idx), .y_idx(y_idx));
        logic [2:0] color = block_colors[19 - y_idx][9 - x_idx];
        get_relative_index(.x_in(hcount_in), .y_in(vcount_in), .clk_in(clk_in), .x_relative(x_relative), .y_relative(y_relative));
        draw_block box(.x_in(x_relative), .y_in(y_relative), .color_in(pixel_out), .clk_in(clk_in), .color_out(next_block_out));



        always_comb begin

        case (color)
        3'd0: pixel_out = 12'h000;
        3'd1: pixel_out = 12'h0CC;//i
        3'd2: pixel_out = 12'h707;//t
        3'd3: pixel_out = 12'h0C0;//S
        3'd4: pixel_out = 12'hC00;//Z
        3'd5: pixel_out = 12'hC60;//L
        3'd6: pixel_out = 12'h00C;//J
        3'd7: pixel_out = 12'hCC0;//0

        endcase
        end

endmodule


`timescale 1ns / 1ps

module title_screen(           input clk_in,
            input rst_in,
            input [10:0] hcount_in,             // horizontal index of current pixel (0..1023)
            input [9:0]  vcount_in,             // vertical index of current pixel (0..767)
            input hsync_in,                     // XVGA horizontal sync signal (active low)
            input vsync_in,                     // XVGA vertical sync signal (active low)
            input blank_in,                     // XVGA blanking (1 means output black pixel)


            output phsync_out,                  // horizontal sync
            output pvsync_out,                  // vertical sync
            output pblank_out,                  // blanking
            output logic [11:0] title_out       // pixel  // r=11:8, g=7:4, b=3:0
        );
```

```
            logic [11:0] tetris_title_out;
            logic [11:0] next_block_blob_out;
            logic [11:0] score_blob_out;
            logic [11:0] zero_out;

//          image_rom YourInstanceName (.clka(clka), .addra(addra), .douta(douta));
            tetris_title_blob obj(  .clk_in(clk_in),
                        .x_in(12'd256),
                        .y_in(12'd256),
                        .hcount_in(hcount_in),
                        .vcount_in(vcount_in),
                        .pixel_out(tetris_title_out));

            zero_blob obj2(  .clk_in(clk_in),
                        .x_in(12'd256),
                        .y_in(12'd450),
                        .hcount_in(hcount_in),
                        .vcount_in(vcount_in),
                        .pixel_out(zero_out));
//          always @ (posedge clk_in) begin


            always_comb begin
            title_out = tetris_title_out | next_block_blob_out|zero_out;
            end
endmodule
```

Appendix C: Serial Input Code

```
module uart_input(  input clk_100mhz,
            input reset_in,
            input bit_in,

            output logic ready,
            output logic [7:0] controls,
            output logic signed[7:0]  x_rotation,
            output logic signed [7:0] y_rotation,
            output logic signed [7:0] z_rotation,
            output logic signed [7:0] zoom,

  );
            logic[31:0] time_count;
            logic[3:0] data_count;
            logic[3:0] bit_count;
            logic [1:0]falling_edge;
            logic reading_data;
            logic end_bit;
            logic found_edge;

            logic [7:0] data_blast[4:0];

            parameter DIVISOR = 564;//868;
            parameter HALF = 282;//434;
            parameter FALLING_EDGE = 2'b10;
            parameter LOW = 0;
            parameter HIGH = 1;

            always_ff @(posedge clk_100mhz) begin
            falling_edge <= {falling_edge[0], bit_in};
            //time_count <= time_count + 32'd1;

            if (ready) begin
            controls   <= data_blast[0];
            x_rotation <= data_blast[1];
            y_rotation <= data_blast[2];
            z_rotation <= data_blast[3];
            zoom       <= data_blast[4];
            end

            if (reset_in) begin
            time_count   <= 32'd0;
            reading_data <= 1'd0;
            data_count   <= 4'd0;
            bit_count  <= 4'd0;
            end_bit    <= 1'd0;
            end else if (reading_data && time_count >= DIVISOR && !end_bit) begin
            time_count <= 32'd0;

            data_blast[data_count][bit_count] <= bit_in;


            if (bit_count >= 3'd7) begin
            bit_count  <= 3'd0;
            end_bit <= 1'd1;
            end else begin
            bit_count <= bit_count + 3'd1;
            end

            end else if (end_bit  && time_count >= DIVISOR) begin
            time_count <= 32'd0;
            reading_data <= 1'd0;
            if (data_count >= 3'd4) begin
            ready <= 1'd1;
            data_count  <= 3'd0;
            end else begin
            data_count <=  data_count + 1;
            end
            end_bit <= 1'd0;
            end else if (!reading_data && time_count >= HALF && found_edge) begin
            time_count <= 32'd0;
            reading_data <= 1'd1;
            ready <= 1'd0;
            found_edge <= 1'd0;
            end else if (!reading_data && falling_edge == FALLING_EDGE) begin
            time_count <= 32'd0;
            found_edge <= 1'd1;
            ready <= 1'd0;
            end else if ( time_count >= DIVISOR) time_count <= 32'd0;
            else time_count <= time_count + 32'd1;
```

```
        end

endmodule


import pygame
import serial.tools.list_ports
import struct
import time

#uses code from Joe
def get_usb_port():
        usb_port = list(serial.tools.list_ports.grep("USB"))
        if len(usb_port) == 1:
                print("Automatically found USB-Serial Controller: {}".format(usb_port[0].description))
                return usb_port[0].device
        else:
                ports = list(serial.tools.list_ports.comports())
                port_dict = {i:[ports[i],ports[i].vid] for i in range(len(ports))}
                usb_id=None
                for p in port_dict:
                        print("UART" in str(port_dict[p][0]))
                        if port_dict[p][1]==1027 and "UART" in str(port_dict[p][0]): #for generic USB Devices
                                usb_id = p
                if usb_id== None:
                        return False
                else:
                        print("Found it")
                        print("USB-Serial Controller: Device {}".format(p))
                        return port_dict[usb_id][0].device

def openSerialPort(s):
        if s:
                ser = serial.Serial(port = s,
                        baudrate=115200,
                        parity=serial.PARITY_NONE,
                        stopbits=serial.STOPBITS_ONE,
                        bytesize=serial.EIGHTBITS,
                        timeout=0.01) #auto-connects already I guess?
                print("Serial Connected!")
                if ser.isOpen():
                         print(ser.name + ' is open...')
                return ser
        else:
                print("No Serial Device :/ Check USB cable connections/device!")
                exit()
                return False

def sendSerialData(ser, controls, x_rotation,y_rotation, z_rotation,zoom):
        try:
                ser.write(struct.pack('>B', controls))
                ser.write(struct.pack('>B', x_rotation))
                ser.write(struct.pack('>B', y_rotation))
                ser.write(struct.pack('>B', z_rotation))
                ser.write(struct.pack('>B', zoom))
        except Exception as e:
                print(e)

s = get_usb_port()  #grab a port
ser = openSerialPort(s)

pygame.init()

pygame.joystick.init()

keys = {"up": 0, "down": 1, "left": 2, "right": 3, "square": 4, "x": 5, "circle": 6, "triangle": 7}
controls = [0] * 8
board_movement =[0]*4
# print(str("".join(map(str, [0,0,0]))))
# print(int("".join(map(str, [0,0,0])), 2)) # changing binart to string
# up - positive, right - positive
x_rotation = 0 # right/lrft - left
y_rotation = 0 # up/down    - left
z_rotation = 0 # right/left - right
zoom = 0      # up/down     - right

updating = True
send = True
joystick_count = pygame.joystick.get_count()
while updating:
        time.sleep(0.1)
        pygame.event.get()

        # for each  joystick but were only going to get the firt one
        # for i in range(1)

        joystick = pygame.joystick.Joystick(1)
        joystick.init()

        axes = joystick.get_numaxes()
        # print(joystick.get_axis(0))
        for i in range(axes):
      axis = joystick.get_axis(i)

        board_movement = [joystick.get_axis(0),#left x - x
                        joystick.get_axis(1), # left y - y
                        joystick.get_axis(2), #right x - z
                        joystick.get_axis(5)] # right y - zoom

        buttons = joystick.get_numbuttons()

        controls[4] = joystick.get_button(0) #square
        controls[5] = joystick.get_button(1) #x
        controls[6] = joystick.get_button(2) # circle
        controls[7] = joystick.get_button(3) # triangle

        right_left, up_down = joystick.get_hat(0)
        print("arrows:", right_left, up_down)
        if up_down == -1:
                print("pressing down")
                controls[1] = 1 #down
                controls[0] = 0 # up
        elif up_down == 1:
                controls[1] = 0 #down
                controls[0] = 1 # up
        elif up_down == 0:
                controls[1] = 0 #down
                controls[0] = 0 # up

        if right_left == -1:
                controls[2] = 1 #left
                controls[3] = 0 # right
        elif right_left == 1:
                controls[2] = 0 #left
                controls[3] = 1 # right
        elif right_left == 0:
                controls[2] = 0 #left
                controls[3] = 0 # right

        for i, num in enumerate(board_movement):
                rotation = int(num * 2**7)
                if rotation >= 128:
                        rotation = 127
                elif rotation < 0:
                        rotation = 2**8 + rotation
                board_movement[i] = rotation

        x_rotation = board_movement[0] # right/lrft - left
        y_rotation = board_movement[1] # up/down    - left
        z_rotation = board_movement[2] # right/left - right
        zoom = board_movement[3]      # up/down     - right
        controls_num = int("".join(map(str, controls)), 2)
        print(controls_num, x_rotation,y_rotation, z_rotation,zoom)
        # if controls_num != 0:
        sendSerialData(ser, controls_num, x_rotation,y_rotation, z_rotation,zoom)

pygame.quit()
```

Appendix D: 3D Module Verilog

```
`timescale 1ns / 1ps

module top_level( input clk_100mhz,
//                       input[15:0] sw,
                        input btnc, //btnu, btnl, btnr, btnd,
                        output  [3:0] vga_r,
                        output  [3:0] vga_b,
                        output  [3:0] vga_g,
                        output vga_hs,
                        output vga_vs,

                        output logic [15:0] led//,
//                        output[15:0] led,
//                        output ca, cb, cc, cd, ce, cf, cg, dp,  // segments a-g, dp
//                        output[3:0] an    // Display location 0-7
                          );

    logic reset,set_matrix_values;
    logic tpd_delay = 0;
    logic clk_65mhz, vclock, sysclk;

    logic [10:0] h_pixel = 200;
    logic [10:0] hcount;    // pixel number on current line
    logic [9:0] v_pixel = 100;
    logic [9:0] vcount;
    logic vsync, hsync, blank;
    logic [4:0] vert_counter;
    logic [3:0] horiz_counter;

    logic write_enable = 0;
    logic read_enable;
    logic [19:0] write_addr = 0;
    logic [19:0] read_addr = 0;     // line number
    logic [1:0] matrix_delay = 0;
    logic [2:0] to_memory_a = 0;
    logic [2:0] from_memory_b;
    logic [2:0] temp = 0;

    logic [11:0] pixel;
    logic [11:0] line;
    logic [11:0] box;
    logic [11:0] triangle_to_store;
    logic [11:0] triangle_to_display;
    logic [11:0] triangle [11:0];

    logic [3:0] x_loc = 0;
    logic [4:0] y_loc = 0;
    logic [3:0] z_loc = 0;

    logic [2:0] three_bit_counter = 0;
    logic [7:0] build_list_counter  = 0;
    logic [7:0] build_list_length = 0;

    logic [23:0] fov = 45;
    logic [4:0] state = 0;

    logic signed [23:0] point1 [3:0];
    logic signed [23:0] point2 [3:0];
    logic signed [23:0] point3 [3:0];
    logic signed [23:0] point4 [3:0];
    logic signed [23:0] point5 [3:0];
    logic signed [23:0] point6 [3:0];
    logic signed [23:0] point7 [3:0];
    logic signed [23:0] point8 [3:0];

    logic signed [23:0] triangle1   [2:0][3:0];
    logic signed [23:0] triangle2   [2:0][3:0];
    logic signed [23:0] triangle3   [2:0][3:0];
    logic signed [23:0] triangle4   [2:0][3:0];
    logic signed [23:0] triangle5   [2:0][3:0];
    logic signed [23:0] triangle6   [2:0][3:0];
    logic signed [23:0] triangle7   [2:0][3:0];
    logic signed [23:0] triangle8   [2:0][3:0];
    logic signed [23:0] triangle9   [2:0][3:0];
    logic signed [23:0] triangle10 [2:0][3:0];
    logic signed [23:0] triangle11 [2:0][3:0];
    logic signed [23:0] triangle12 [2:0][3:0];

    logic signed [23:0] modified_triangle1[2:0][3:0];
    logic signed [23:0] modified_triangle2[2:0][3:0];
    logic signed [23:0] modified_triangle3[2:0][3:0];
    logic signed [23:0] modified_triangle4[2:0][3:0];
    logic signed [23:0] modified_triangle5[2:0][3:0];
    logic signed [23:0] modified_triangle6[2:0][3:0];
    logic signed [23:0] modified_triangle7[2:0][3:0];
    logic signed [23:0] modified_triangle8[2:0][3:0];
    logic signed [23:0] modified_triangle9[2:0][3:0];
    logic signed [23:0] modified_triangle10[2:0][3:0];
    logic signed [23:0] modified_triangle11[2:0][3:0];
    logic signed [23:0] modified_triangle12[2:0][3:0];

    logic [6:0] temp_state;

    //      cols bits                    rows
    logic [9:0][2:0] block_colors[19:0] = {  {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
                                             {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
                                             {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
                                             {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
                                             {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
                                             {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
                                             {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
                                             {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
                                             {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
                                             {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
                                             {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
                                             {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
                                             {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
                                             {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
                                             {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
                                             {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
                                             {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
                                             {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
                                             {3'd4, 3'd1, 3'd3, 3'd1, 3'd5, 3'd1, 3'd1, 3'd7, 3'd1, 3'd1},
                                             {3'd4, 3'd1, 3'd3, 3'd1, 3'd5, 3'd1, 3'd1, 3'd7, 3'd1, 3'd1}}; //top left is (0,0)*/
    logic [23:0] build_list [19:0] = {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0,
                                      3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0};
//                                      {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
//                                      {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
//                                      {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
//                                      {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
//                                      {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
//                                      {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
//                                      {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
//                                          {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
//                                          {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
//                                          {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
//                                          {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
//                                          {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
//                                          {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
//                                          {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
//                                          {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
//                                          {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
//                                          {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0},
//                                          {3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0, 3'd0}};*/



   parameter HEIGHT = 768,
                  WIDTH   = 1024,
                  MAX_PIXELS = 17'd115200,
                  X_OFFSET = 10,
                  Y_OFFSET = 10,
                  Z_OFFSET = 1,
                  INITIALIZE_VALUES = 5'b0,
                  SET_POINTS = 5'b00001,
                  CREATE_TRIANGLE = 5'b00010,
                  MATRIX_MATH = 5'b00100,
                  DRAW = 5'b01000,
                  PIXEL_TO_RAM = 5'b10000,
                  SET_PIXEL_TO_CHECK = 5'b10001,
                  MAKE_LIST_TO_CHECK = 5'b10010,
                  FILL_POINTS = 5'b10100,
                  BEGIN_DISPLAY = 5'b11000,
                  CLEAR_BUILD_LIST = 5'b01001,
                  WAIT_FOR_VSYNC = 5'b01010,
                  NUM_ROWS_TO_FILL = 3,
                  NUM_COLS_TO_FILL = 3,
                  T_BLOCK = 3'b010,
                   I_BLOCK = 3'b001,
                   J_BLOCK = 3'b110,
                   L_BLOCK = 3'b101,
                   Z_BLOCK = 3'b100,
                   S_BLOCK = 3'b011,
                   O_BLOCK = 3'b111;


//   clk_wiz_0 clk_gen (.clk_out1(clk_65mhz), .clk_out2(clk_148mhz), .clk_out3(clk_400mhz), .reset(0),
.clk_in1(clk_100mhz));

//   blk_mem_gen_0 screen_buff_1( .clka(sysclk),  .clkb(vclock),
//                                         .addra(write_addr),  .addrb(read_addr),
//                                         .dina(to_memory_a),  .doutb(from_memory_b),
//                                         .wea(write_enable), .enb(read_enable));

   clk_wiz_1 clk_gen (.clk_out1(clk_65mhz), .reset(0), .clk_in1(clk_100mhz));

   blk_mem_gen_1 screen_buff_1( .clka(sysclk),           .clkb(vclock),
                                        .addra(write_addr),   .addrb(read_addr),
                                        .dina(to_memory_a), .doutb(from_memory_b),
                                        .wea(write_enable),   .enb(read_enable));



   assign vclock = clk_65mhz;
   assign sysclk = clk_100mhz;

   xvga              xvga1 (.vclock_in(vclock), .hcount_out(hcount), .vcount_out(vcount), .vsync_out(vsync),
.hsync_out(hsync), .blank_out(blank));

   debounce        reset_debounce(.clk_in(sysclk), . rst_in(0), .bouncey_in(btnc), .clean_out(reset));

   draw_line        play_area (.vclock_in(vclock), .hcount_in(hcount), . vcount_in(vcount), .rgb_out(box));



   draw_triangle    triangle_1  (.vclock_in(sysclk),  .sysclk(sysclk),  .triangle(modified_triangle1), .hcount_in(h_pixel),
.vcount_in(v_pixel),  .rgb_out(triangle[0]));
   draw_triangle    triangle_2  (.vclock_in(sysclk),  .sysclk(sysclk),  .triangle(modified_triangle2), .hcount_in(h_pixel),
.vcount_in(v_pixel),  .rgb_out(triangle[1]));
   draw_triangle    triangle_3  (.vclock_in(sysclk),  .sysclk(sysclk),  .triangle(modified_triangle3), .hcount_in(h_pixel),
.vcount_in(v_pixel),  .rgb_out(triangle[2]));
   draw_triangle    triangle_4  (.vclock_in(sysclk),  .sysclk(sysclk),  .triangle(modified_triangle4), .hcount_in(h_pixel),
.vcount_in(v_pixel),  .rgb_out(triangle[3]));
   draw_triangle    triangle_5  (.vclock_in(sysclk),  .sysclk(sysclk),  .triangle(modified_triangle5), .hcount_in(h_pixel),
.vcount_in(v_pixel),  .rgb_out(triangle[4]));
   draw_triangle    triangle_6  (.vclock_in(sysclk),  .sysclk(sysclk),  .triangle(modified_triangle6), .hcount_in(h_pixel),
.vcount_in(v_pixel),  .rgb_out(triangle[5]));
   draw_triangle    triangle_7  (.vclock_in(sysclk),  .sysclk(sysclk),  .triangle(modified_triangle7), .hcount_in(h_pixel),
.vcount_in(v_pixel),  .rgb_out(triangle[6]));
   draw_triangle    triangle_8  (.vclock_in(sysclk),  .sysclk(sysclk),  .triangle(modified_triangle8), .hcount_in(h_pixel),
.vcount_in(v_pixel),  .rgb_out(triangle[7]));
   draw_triangle    triangle_9  (.vclock_in(sysclk),  .sysclk(sysclk),  .triangle(modified_triangle9), .hcount_in(h_pixel),
.vcount_in(v_pixel),  .rgb_out(triangle[8]));
   draw_triangle    triangle_10(.vclock_in(sysclk),  .sysclk(sysclk),  .triangle(modified_triangle10), .hcount_in(h_pixel),
.vcount_in(v_pixel),  .rgb_out(triangle[9]));
   draw_triangle    triangle_11(.vclock_in(sysclk),  .sysclk(sysclk),  .triangle(modified_triangle11), .hcount_in(h_pixel),
.vcount_in(v_pixel),  .rgb_out(triangle[10]));
   draw_triangle    triangle_12(.vclock_in(sysclk),  .sysclk(sysclk),  .triangle(modified_triangle12), .hcount_in(h_pixel),
.vcount_in(v_pixel),  .rgb_out(triangle[11]));

   matrix              #(.HEIGHT(HEIGHT), .WIDTH(WIDTH))   my_mat_1    ( .sysclk(sysclk),  .tri_array(triangle1),  .reset(reset),
.field_of_view(fov), .set_matrix_values(set_matrix_values),  .corrected_array(modified_triangle1));
   matrix              #(.HEIGHT(HEIGHT), .WIDTH(WIDTH))   my_mat_2    ( .sysclk(sysclk),  .tri_array(triangle2),  .reset(reset),
.field_of_view(fov), .set_matrix_values(set_matrix_values),  .corrected_array(modified_triangle2));
   matrix              #(.HEIGHT(HEIGHT), .WIDTH(WIDTH))   my_mat_3    ( .sysclk(sysclk),  .tri_array(triangle3),  .reset(reset),
.field_of_view(fov), .set_matrix_values(set_matrix_values),  .corrected_array(modified_triangle3));
   matrix              #(.HEIGHT(HEIGHT), .WIDTH(WIDTH))   my_mat_5    ( .sysclk(sysclk),  .tri_array(triangle4),  .reset(reset),
.field_of_view(fov), .set_matrix_values(set_matrix_values),  .corrected_array(modified_triangle4));
   matrix              #(.HEIGHT(HEIGHT), .WIDTH(WIDTH))   my_mat_4    ( .sysclk(sysclk),  .tri_array(triangle5),  .reset(reset),
.field_of_view(fov), .set_matrix_values(set_matrix_values),  .corrected_array(modified_triangle5));
   matrix              #(.HEIGHT(HEIGHT), .WIDTH(WIDTH))   my_mat_6    ( .sysclk(sysclk),  .tri_array(triangle6),  .reset(reset),
.field_of_view(fov), .set_matrix_values(set_matrix_values),  .corrected_array(modified_triangle6));
   matrix              #(.HEIGHT(HEIGHT), .WIDTH(WIDTH))   my_mat_7    ( .sysclk(sysclk),  .tri_array(triangle7),  .reset(reset),
.field_of_view(fov), .set_matrix_values(set_matrix_values),  .corrected_array(modified_triangle7));
   matrix              #(.HEIGHT(HEIGHT), .WIDTH(WIDTH))   my_mat_8    ( .sysclk(sysclk),  .tri_array(triangle8),  .reset(reset),
.field_of_view(fov), .set_matrix_values(set_matrix_values),  .corrected_array(modified_triangle8));
```

```
  matrix           #(.HEIGHT(HEIGHT), .WIDTH(WIDTH))   my_mat_9    ( .sysclk(sysclk),  .tri_array(triangle9),  .reset(reset),
.field_of_view(fov), .set_matrix_values(set_matrix_values),  .corrected_array(modified_triangle9));
  matrix           #(.HEIGHT(HEIGHT), .WIDTH(WIDTH))   my_mat_10   ( .sysclk(sysclk),  .tri_array(triangle10),  .reset(reset),
.field_of_view(fov),  .set_matrix_values(set_matrix_values), .corrected_array(modified_triangle10));
  matrix           #(.HEIGHT(HEIGHT), .WIDTH(WIDTH))   my_mat_11   ( .sysclk(sysclk),  .tri_array(triangle11),  .reset(reset),
.field_of_view(fov),  .set_matrix_values(set_matrix_values), .corrected_array(modified_triangle11));
  matrix           #(.HEIGHT(HEIGHT), .WIDTH(WIDTH))   my_mat_12   ( .sysclk(sysclk),  .tri_array(triangle12),  .reset(reset),
.field_of_view(fov),  .set_matrix_values(set_matrix_values), .corrected_array(modified_triangle12));

//   assign led[15]  =1;


  always_ff @(posedge sysclk) begin
      if(reset) begin
        x_loc <= 0;
        y_loc <= 0;
        state <= 5'b0;
        led <= 15'b0;
        for (int k = 0; k <200; k++) begin
          build_list[k] <= 1'b0;
          end
//         for (int j = 0; j < NUM_ROWS_TO_FILL; j++) begin
//           for ( int i = 0; i < NUM_COLS_TO_FILL; i++) begin
//             block_colors[i][j] = three_bit_counter;
//             three_bit_counter <= three_bit_counter +1;
//             end // i loop
//           end // j loop
        end
      else begin
        case(state)
          SET_PIXEL_TO_CHECK: begin
            set_matrix_values <= 0;
            if(v_pixel >= 580 && h_pixel >= 440) begin // need to reset to top left
              led[14] <= 1;
              v_pixel <= 100;
              h_pixel <= 200;
              state <= CLEAR_BUILD_LIST;
              temp_state <= 6'b100000;
              end
            else if (h_pixel >= 440 && v_pixel <= 580) begin // need to increase row
              led[13] <= 1;
              h_pixel <= 200;
              v_pixel <= v_pixel +1;
              state <= SET_POINTS;
              temp_state <= 6'b010000;
              end
            else if (h_pixel > 200 || (h_pixel == 200 && v_pixel >100)) begin
              led[11] <= 1;
              h_pixel <= h_pixel +1;
              state <= SET_POINTS;
              temp_state <= 6'b001000;
              end
            else if (h_pixel == 200 && v_pixel == 100) begin
              led[12] <= 1;
              h_pixel <= h_pixel +1;
              state <= MAKE_LIST_TO_CHECK;
              temp_state <= 6'b000100;
```

```
//                        h_pixel <= 200;
//                        v_pixel <= 100;
                        end
//                    else /* if (h_pixel < 440 && v_pixel < 580)*/ begin
//                        led[11] <= 1;
//                        state <= SET_POINTS;
//                        end
//                    else begin
//                        state <= MAKE_LIST_TO_CHECK;
//                        h_pixel <= 200;
//                        v_pixel <= 100;
//                        end
//                    led[0] <= 1;
                    end // set pixel value

                MAKE_LIST_TO_CHECK: begin

//                        for (int i = 0; i <= 19; i++) begin
//                            for ( int j = 0; j <= 9; j++) begin
                            if ( /*horiz_counter >= 10 && */ vert_counter >= 20) begin
                                horiz_counter  <= 0;
                                vert_counter <= 0;
                                build_list_length <= build_list_counter;
                                build_list_counter <= 0;
                                state <= SET_POINTS;
                                led[7] <= 1;
                                end
                            else if( horiz_counter >= 10 && vert_counter <20) begin
                                horiz_counter <= 0;
                                vert_counter <= vert_counter +1;
                                state <= MAKE_LIST_TO_CHECK;
                                led[8] <= 1;
                                end
                            else begin
                                    if ( block_colors[vert_counter][horiz_counter] > 0 && horiz_counter < 10 && vert_counter < 20)
begin //if the location has a block to draw
                                        if ( vert_counter == 0 && horiz_counter == 0) begin //top left
                                            build_list[build_list_counter] <= {1'b1, 1'b1, block_colors[vert_counter+1][horiz_counter] >
0 ? 1 : 0,  block_colors[vert_counter][horiz_counter+1] > 0 ? 1 : 0, 1'b1, vert_counter*24+200, horiz_counter * 24+100};
                                            led[9] <= 1;
                                            end
                                        else if (vert_counter == 0 && horiz_counter == 9) begin // top right
                                            build_list[build_list_counter] <= {1'b1, 1'b1,  block_colors[vert_counter+1][horiz_counter]
> 0 ? 1'b0 : 1'b1,1'b1,1'b1,block_colors[vert_counter][horiz_counter-1] > 0 ? 1'b0 : 1'b1, vert_counter*24+200, horiz_counter
* 24+100};
                                            led[3] <= 1;
                                            end
                                        else if ( vert_counter == 19 && horiz_counter == 0) begin // bottom left
                                            build_list[build_list_counter] <= { 1'b1, block_colors[vert_counter - 1][horiz_counter] > 0 ?
0 : 1, 1'b1, block_colors[vert_counter][horiz_counter+1] > 0 ? 0 : 1, 1'b1, vert_counter*24+200, horiz_counter * 24+100};
                                            led[4] <= 1;
                                            end
                                        else if ( vert_counter == 19 && horiz_counter == 9) begin // bottom right
                                            build_list[build_list_counter] <= {1'b1, block_colors[vert_counter -1][horiz_counter] > 0 ?
0 : 1,  1'b1, 1'b1, block_colors[vert_counter][ horiz_counter -1] > 0 ? 0 : 1, vert_counter*24+200, horiz_counter * 24+100};
                                            led[5] <= 1;
                                            end
                                        else begin // all other places on board
                                          build_list[build_list_counter] <= {1'b1, block_colors[vert_counter-1][horiz_counter] > 0 ? 0
: 1, block_colors[vert_counter+1][horiz_counter] > 0 ? 0 : 1, block_colors[vert_counter][ horiz_counter +1] > 0 ? 0 : 1,
block_colors[vert_counter] [ horiz_counter - 1 ] > 0 ? 0 : 1, vert_counter*24+200, horiz_counter * 24+100};
                                          led[6] <= 1;
                                          end
                                    build_list_counter <= build_list_counter + 1'b1;
                                    state <= MAKE_LIST_TO_CHECK;
                                    end

                                  end // ( if block_colors)

//                                end // i loop
//                              end // j loop
                        horiz_counter <= horiz_counter +1;
                        led[1] <= 1;
                        if(vert_counter > 19) led[14] <= 1;
                        end // make_list state

                      SET_POINTS: begin
                        if(build_list_counter > build_list_length) begin
                          x_loc = 0;
                          y_loc = 0;
                          build_list_counter <= 0;
                          state <= SET_PIXEL_TO_CHECK;//fix this with something useful
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                          end
                        else begin
                          build_list_counter = build_list_counter >= 0 ? build_list_counter +1: 0;
                          x_loc <= build_list[build_list_counter][18:10];
                          y_loc <= build_list[build_list_counter][9:0];
                          state <= FILL_POINTS;
                          end
                        led[2] <= 1;
                        end
                      FILL_POINTS: begin
                          point1[3] <= x_loc;  point1[2] <= y_loc;  point1[1] <= z_loc;  point1[0] <= z_loc;
                          point2[3] <= x_loc;  point2[2] <= y_loc + Y_OFFSET*(y_loc > 0? y_loc: 1);  point2[1] <= z_loc;  point2[0] <=
z_loc;
                          point3[3] <= x_loc + X_OFFSET*(x_loc > 0? x_loc: 1);  point3[2] <= y_loc + Y_OFFSET*(y_loc > 0? y_loc: 1);
point3[1] <= z_loc;  point3[0] <= z_loc;
                          point4[3] <= x_loc + X_OFFSET*(x_loc > 0? x_loc: 1);  point4[2] <= y_loc;  point4[1] <= z_loc;  point4[0] <=
z_loc;
                          point5[3] <= x_loc + X_OFFSET*(x_loc > 0? x_loc: 1);  point5[2] <= y_loc;  point5[1] <= z_loc +
Z_OFFSET*(z_loc > 0? z_loc: 1);  point5[0] <= z_loc + Z_OFFSET*(z_loc > 0? z_loc: 1);
                          point6[3] <= x_loc + X_OFFSET*(x_loc > 0? x_loc: 1);  point6[2] <= y_loc + Y_OFFSET*(y_loc > 0? y_loc: 1);
point6[1] <= z_loc + Z_OFFSET*(z_loc > 0? z_loc: 1);  point6[0] <= z_loc + Z_OFFSET*(z_loc > 0? z_loc: 1);
                          point7[3] <= x_loc;  point7[2] <= y_loc + Y_OFFSET*(y_loc > 0? y_loc: 1);  point7[1] <= z_loc +
Z_OFFSET*(z_loc > 0? z_loc: 1);  point7[0] <= z_loc + Z_OFFSET*(z_loc > 0? z_loc: 1);
                          point8[3] <= x_loc;  point8[2] <= y_loc;  point8[1] <= z_loc + Z_OFFSET*(z_loc > 0? z_loc: 1);  point8[0] <=
z_loc + Z_OFFSET*(z_loc > 0? z_loc: 1);

                          state <= CREATE_TRIANGLE;
//                       led[3] <= 1;
                          end

                      CREATE_TRIANGLE: begin
```

```
                triangle1[0][3] <= point1[3];  triangle1[0][2] <= point1[2];  triangle1[0][1] <= point1[1];  triangle1[0][0] <=
point1[0];
                triangle1[1][3] <= point2[3];  triangle1[1][2] <= point2[2];  triangle1[1][1] <= point2[1];  triangle1[1][0] <=
point2[0];
                triangle1[2][3] <= point3[3];  triangle1[2][2] <= point3[2];  triangle1[2][1] <= point3[1];  triangle1[2][0] <=
point3[0];

                triangle2[0][3] <= point4[3];  triangle2[0][2] <= point4[2];  triangle2[0][1] <= point4[1];  triangle2[0][0] <=
point4[0];
                triangle2[1][3] <= point1[3];  triangle2[1][2] <= point1[2];  triangle2[1][1] <= point1[1];  triangle2[1][0] <=
point1[0];
                triangle2[2][3] <= point3[3];  triangle2[2][2] <= point3[2];  triangle2[2][1] <= point3[1];  triangle2[2][0] <=
point3[0];

                triangle3[0][3] <= point3[3];  triangle3[0][2] <= point3[2];  triangle3[0][1] <= point3[1];  triangle3[0][0] <=
point3[0];
                triangle3[1][3] <= point6[3];  triangle3[1][2] <= point6[2];  triangle3[1][1] <= point6[1];  triangle3[1][0] <=
point6[0];
                triangle3[2][3] <= point4[3];  triangle3[2][2] <= point4[2];  triangle3[2][1] <= point4[1];  triangle3[2][0] <=
point4[0];

                triangle4[0][3] <= point4[3];  triangle4[0][2] <= point4[2];  triangle4[0][1] <= point4[1];  triangle4[0][0] <=
point4[0];
                triangle4[1][3] <= point6[3];  triangle4[1][2] <= point6[2];  triangle4[1][1] <= point6[1];  triangle4[1][0] <=
point6[0];
                triangle4[2][3] <= point5[3];  triangle4[2][2] <= point5[2];  triangle4[2][1] <= point5[1];  triangle4[2][0] <=
point5[0];

                triangle5[0][3] <= point5[3];  triangle5[0][2] <= point5[2];  triangle5[0][1] <= point5[1];  triangle5[0][0] <=
point5[0];
                triangle5[1][3] <= point6[3];  triangle5[1][2] <= point6[2];  triangle5[1][1] <= point6[1];  triangle5[1][0] <=
point6[0];
                triangle5[2][3] <= point7[3];  triangle5[2][2] <= point7[2];  triangle5[2][1] <= point7[1];  triangle5[2][0] <=
point7[0];

                triangle6[0][3] <= point5[3];  triangle6[0][2] <= point5[2];  triangle6[0][1] <= point5[1];  triangle6[0][0] <=
point5[0];
                triangle6[1][3] <= point7[3];  triangle6[1][2] <= point7[2];  triangle6[1][1] <= point7[1];  triangle6[1][0] <=
point7[0];
                triangle6[2][3] <= point8[3];  triangle6[2][2] <= point8[2];  triangle6[2][1] <= point8[1];  triangle6[2][0] <=
point8[0];

                triangle7[0][3] <= point8[3];  triangle7[0][2] <= point8[2];  triangle7[0][1] <= point8[1];  triangle7[0][0] <=
point8[0];
                triangle7[1][3] <= point7[3];  triangle7[1][2] <= point7[2];  triangle7[1][1] <= point7[1];  triangle7[1][0] <=
point7[0];
                triangle7[2][3] <= point2[3];  triangle7[2][2] <= point2[2];  triangle7[2][1] <= point2[1];  triangle7[2][0] <=
point2[0];

                triangle8[0][3] <= point8[3];  triangle8[0][2] <= point8[2];  triangle8[0][1] <= point8[1];  triangle8[0][0] <=
point8[0];
                triangle8[1][3] <= point2[3];  triangle8[1][2] <= point2[2];  triangle8[1][1] <= point2[1];  triangle8[1][0] <=
point2[0];
                triangle8[2][3] <= point1[3];  triangle8[2][2] <= point1[2];  triangle8[2][1] <= point1[1];  triangle8[2][0] <=
point1[0];
```

```
            triangle9[0][3] <= point2[3];  triangle9[0][2] <= point2[2];  triangle9[0][1] <= point2[1];  triangle9[0][0] <=
point2[0];
            triangle9[1][3] <= point7[3];  triangle9[1][2] <= point7[2];  triangle9[1][1] <= point7[1];  triangle9[1][0] <=
point7[0];
            triangle9[2][3] <= point6[3];  triangle9[2][2] <= point6[2];  triangle9[2][1] <= point6[1];  triangle9[2][0] <=
point6[0];

            triangle10[0][3] <= point2[3];  triangle10[0][2] <= point2[2];  triangle10[0][1] <= point2[1];  triangle10[0][0] <=
point2[0];
            triangle10[1][3] <= point6[3];  triangle10[1][2] <= point6[2];  triangle10[1][1] <= point6[1];  triangle10[1][0] <=
point6[0];
            triangle10[2][3] <= point3[3];  triangle10[2][2] <= point3[2];  triangle10[2][1] <= point3[1];  triangle10[2][0] <=
point3[0];

            triangle11[0][3] <= point1[3];  triangle11[0][2] <= point1[2];  triangle11[0][1] <= point1[1];  triangle11[0][0] <=
point1[0];
            triangle11[1][3] <= point8[3];  triangle11[1][2] <= point8[2];  triangle11[1][1] <= point8[1];  triangle11[1][0] <=
point8[0];
            triangle11[2][3] <= point5[3];  triangle11[2][2] <= point5[2];  triangle11[2][1] <= point5[1];  triangle11[2][0] <=
point5[0];

            triangle12[0][3] <= point1[3];  triangle12[0][2] <= point1[2];  triangle12[0][1] <= point1[1];  triangle12[0][0] <=
point1[0];
            triangle12[1][3] <= point5[3];  triangle12[1][2] <= point5[2];  triangle12[1][1] <= point5[1];  triangle12[1][0] <=
point5[0];
            triangle12[2][3] <= point4[3];  triangle12[2][2] <= point4[2];  triangle12[2][1] <= point4[1];  triangle12[2][0] <=
point4[0];

            state <= DRAW;

//           led[4] <= 1;
            end
          MATRIX_MATH: begin
              matrix_delay <= matrix_delay < 3 ? matrix_delay +1 : state <= DRAW;

            led[5] <= 1;
            end
          DRAW: begin
            matrix_delay <= 0;
            if(|triangle[0] == 0 && |triangle[1] == 0 &&|triangle[2] == 0 &&|triangle[3] == 0 &&
              |triangle[4] == 0 && |triangle[5] == 0 &&|triangle[6] == 0 &&|triangle[7] == 0 &&
              |triangle[8] == 0 && |triangle[9] == 0 &&|triangle[10] == 0 &&|triangle[11] == 0 &&
               build_list_counter < build_list_length)
               begin
                 state <= PIXEL_TO_RAM;;
                 triangle_to_store <= 0;
                 end
            else begin
               triangle_to_store <= build_list[build_list_counter][22:20];
               state <= PIXEL_TO_RAM;
               end

//           led[6] <= 1;
            end
          PIXEL_TO_RAM: begin
            if (write_enable) begin
               write_enable <= 0;
```

```
                    write_addr <= write_addr < MAX_PIXELS ? write_addr+1 : 0;
                    state <= SET_POINTS;
                    end
                  else begin
                    write_enable <= 1;
                    to_memory_a <= triangle_to_store;
                    state <= PIXEL_TO_RAM;
                    end

//                 led[7] <= 1;
                  end //pixel_to_ram

                CLEAR_BUILD_LIST: begin
                  for(int i = 0; i < 200; i++) begin
                    build_list[i] <= 0;
                    end //for loop
                  state <= SET_PIXEL_TO_CHECK;

//                 led[8] <= 1;
                  end  //CLEAR_BUILD_LIST

//               WAIT_FOR_VSYNC: begin
//                 state <= vsync ? SET_PIXEL_TO_CHECK : WAIT_FOR_VSYNC;
////                 led[9] <= 1;
//                 end

//               BEGIN_DISPLAY: begin
//                 if (hcount >= 200 && hcount  < 440 && vcount >= 100 && vcount < 580) begin
//                   triangle_to_display <= from_memory_b;
//                   read_addr <= read_addr +1;
//                   state <= BEGIN_DISPLAY;
//                   end
//                 else begin
//                   read_addr <= 0;
//                   state <= 0; // <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
//                   end
//                 end

                INITIALIZE_VALUES: begin
                  v_pixel <= 100;
                  h_pixel <= 200;
                  for (int i = 0; i <200; i++) begin
                    build_list[i] <= 0;
                    end
                  build_list_counter <= 0;
                  build_list_length <= 0;
                  vert_counter <= 0;
                  horiz_counter <= 0;
//                 led[11] <= 1;
                  state <= SET_PIXEL_TO_CHECK;
                  set_matrix_values <= 1;
                  end

                default: begin
                state <= SET_PIXEL_TO_CHECK;
//                 led[10] <= 1;
                  end
```

```
            endcase
        end

        //pick a color to draw a pixel
        led[15] <= build_list_counter == 0 ? 1: 0;
    end //always_ff sysclk

    always_ff @(posedge vclock ) begin
        if (hcount >= 200 && hcount  < 440 && vcount >= 100 && vcount < 580) begin
            read_enable <= 1;
            case(from_memory_b)
                I_BLOCK: begin
                    pixel <= 12'h0cc;
                    end
                T_BLOCK: begin
                    pixel <= 12'h707;
                    end
                S_BLOCK: begin
                    pixel <= 12'h0c0;
                    end
                Z_BLOCK: begin
                    pixel <= 12'hc00;
                    end
                L_BLOCK: begin
                    pixel <= 12'hc60;
                    end
                J_BLOCK: begin
                    pixel <= 12'h00c;
                    end
                O_BLOCK: begin
                    pixel <= 12'hcc0;
                    end
                endcase
//              pixel <={from_memory_b[2], 3'b0, from_memory_b[1], 3'b0, from_memory_b[0], 3'b0};
            read_addr <= read_addr < MAX_PIXELS ? read_addr +1 : 0;
            end
        else if ( |box ) begin
            read_enable <= 0;
            pixel <= 12'hfff;
            end
        else begin
            read_addr <= 0;
            read_enable <= 0;
//              pixel <= 12'hCCC;
            end
//          if(vcount >= 100 && hcount >= 200 && vcount < 580 && hcount < 240) begin
//              temp <= from_memory_b;
//              pixel <= {4*temp[2], 4*temp[1], 4*temp[0]};

//              end



//          else if ( |triangle) begin
//              pixel <= triangle;
//              end
//          else begin
//                  pixel <= box;
//                  end
          end


//        always_comb begin
//           if(make_build_list) begin
//              for (int j = 0; j < 20; j++) begin
//                 for ( int i = 0; i < 10; i++) begin
//                    if ( |block_colors[j][i]) begin
//                       build_list[build_list_counter] = block_colors[j][i];
//                       end
//                    end // i loop
//                 end // j loop
//              end // list making if logic
//           end //always_comb

        assign vga_r = pixel[11:8];
        assign vga_g = pixel[7:4];
        assign vga_b = pixel[3:0];
        assign vga_hs = ~hsync;
        assign vga_vs = ~vsync;

endmodule //top_level

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Create Date: 11/15/2019 04:50:55 PM
// Design Name:
// Module Name: draw_triangle
// Additional Comments:
//  Triangle algorithm from "A more efficient
//  triangle rasterization algorithm implemented in FPGA" by X Qang, F Guo, M Zhu.
//  https://ieeexplore.ieee.org/document/6376782
//
//////////////////////////////////////////////////////////////////////////////////


module draw_triangle (  input vclock_in,
                        input sysclk,
                        input signed  [23:0] triangle   [2:0][3:0],
                        input signed [12:0] hcount_in,
                        input signed [11:0] vcount_in,
                        output logic [11:0] rgb_out
                        );

    parameter X_OFFSET = 24,
              Y_OFFSET = 24;

    logic signed [28:0] e1, e2, e3;
    logic [8:0] i = 1;
    logic [8:0] j = 1;
    logic ifflag=0;
    logic updateflag = 0;
    logic signed [19:0] x1, x2, x3;
    logic signed [19:0] y1, y2, y3;
    logic [11:0] pixel = 0;
```

```
//    assign rgb_out = pixel;
/*    always_ff @(posedge vclock_in) begin
//        if(~|hcount_in && ~|vcount_in) begin
//            updateflag = 1;
        for(int k = 0; k <20; k++) begin
            for(int l = 0; l<10; l++) begin

//                        x1 = j*X_OFFSET;
//                        x2 = (j+1)*X_OFFSET;
//                        y1 = i*Y_OFFSET;
//                        y2 = (i+1)*Y_OFFSET;
//                        y3 = (i+1)*Y_OFFSET;
//                        x3 = (j+1)*X_OFFSET;
//                        e1<=-(y2-y1)*(hcount_in-x1)+(x2-x1)*(vcount_in-y1);
//                        e2<=-(y3-y2)*(hcount_in-x2)+(x3-x2)*(vcount_in-y2);
//                        e3<=-(y1-y3)*(hcount_in-x3)+(x1-x3)*(vcount_in-y3);
//                        if((e1 >= 0) && (e2 >= 0) && (e3 >= 0)) begin
//                                pixel[3:0] <= block_colors[(i*10)+j][0]*15;
//                                pixel[7:4] <= block_colors[(i*10)+j][1]*15;
//                                pixel[11:8] <= block_colors[(i*10)+j][2]*15;
//                                end
//                        end // if check for color

                //indexing into game_board, first i iteration will result in all zeros when
                // checking for a color, thus separate if to catch that error and correct
                if( i == 0) begin
                    //offset x and y from gameboard vector to actual position
                    if( block_colors[j] >= 1) begin // if there is a piece of a tetronimo at this location
                        //set the x and y location in 2 dimensions
                        x1 = j*X_OFFSET;
                        x2 = (j+1)*X_OFFSET;
                        y1 = i*Y_OFFSET;
                        y2 = (i+1)*Y_OFFSET;
                        y3 = (i+1)*Y_OFFSET;
                        x3 = (j+1)*X_OFFSET;
                        e1<=-(y2-y1)*(hcount_in-x1)+(x2-x1)*(vcount_in-y1);
                        e2<=-(y3-y2)*(hcount_in-x2)+(x3-x2)*(vcount_in-y2);
                        e3<=-(y1-y3)*(hcount_in-x3)+(x1-x3)*(vcount_in-y3);
                        if((e1 >= 0) && (e2 >= 0) && (e3 >= 0)) begin
                                pixel[3:0] <= block_colors[(i*10)+j][0]*15;
                                pixel[7:4] <= block_colors[(i*10)+j][1]*15;
                                pixel[11:8] <= block_colors[(i*10)+j][2]*15;
                                end
                        end // if check for color
                    else begin
                        pixel <= 0;
                        end // if color check else
                end

                else begin
                    // complete repeat of code above with exception of indexing into block color array
                     if( block_colors[(i*10)+j] >= 1) begin // if there is a piece of a tetronimo at this location
                        x1 = j*X_OFFSET;
                        x2 = (j+1)*X_OFFSET;
                        y1 = i*Y_OFFSET;
                        y2 = (i+1)*Y_OFFSET;
                        y3 = (i+1)*Y_OFFSET;
                        x3 = (j+1)*X_OFFSET;
                        e1<=-(y2-y1)*(hcount_in-x1)+(x2-x1)*(vcount_in-y1);
                        e2<=-(y3-y2)*(hcount_in-x2)+(x3-x2)*(vcount_in-y2);
                        e3<=-(y1-y3)*(hcount_in-x3)+(x1-x3)*(vcount_in-y3);
                        if((e1 >= 0) && (e2 >= 0) && (e3 >= 0)) begin
                                pixel[3:0] <= block_colors[(i*10)+j][0]*15;
                                pixel[7:4] <= block_colors[(i*10)+j][1]*15;
                                pixel[11:8] <= block_colors[(i*10)+j][2]*15;
                                end
                        end // if check for color
                    else begin
                        pixel <= 0;
                        end // if color check else
                    end

                j <= j == 9? 0: j+1;
                end //j loop
              i <= i == 19? 0: i + 1;
              end // i loop
            end //always_ff  */


// Functioning code to draw one triangle.
always_ff @(posedge sysclk) begin
        e1 = - (triangle[1][2] - triangle[0][2]) * (hcount_in - triangle[0][3]) + (triangle[1][3] - triangle[0][3]) * (vcount_in -
triangle[0][3]);
        e2 = - (triangle[2][2] - triangle[1][2]) * (hcount_in - triangle[1][3]) + (triangle[2][3] - triangle[1][3]) * (vcount_in -
triangle[1][2]);
        e3 = - (triangle[0][2] - triangle[2][2]) * (hcount_in - triangle[2][3]) + (triangle[0][3] - triangle[2][3]) * (vcount_in -
triangle[2][2]);
//      e1 = - (y2-y1)*(hcount_in-x1)+(x2-x1)*(vcount_in-y1);
//      e2 = - (y3-y2)*(hcount_in-x2)+(x3-x2)*(vcount_in-y2);
//      e3 = - (y1-y3)*(hcount_in-x3)+(x1-x3)*(vcount_in-y3);
        if((e1 >= 0) && (e2 >= 0) && (e3 >= 0)) begin
//              ifflag <= 1;
                pixel <= 12'hFFF;
                end
        else begin
//          ifflag <= 0;
          pixel <= 0;
          end

        end //always ff 400mhz


//looping every piece of the board every clock cycle
//always_ff @(posedge vclock_in) begin
//          for(int k = 0; k <20; k++) begin
//              for(int l = 0; l<10; l++) begin
//              //indexing into game_board, first i iteration will result in all zeros when
//              // checking for a color, thus separate if to catch that error and correct
//              if( i == 0) begin
//                  //offset x and y from gameboard vector to actual position
//                  if( block_colors[j] >= 1) begin // if there is a piece of a tetronimo at this location
//                      //set the x and y location in 2 dimensions
//                      x1 <= j*X_OFFSET;
//                      x2 <= (j+1)*X_OFFSET;
//                      y1 <= i*Y_OFFSET;
//                      y2 <= (i+1)*Y_OFFSET;
//                      y3 <= (i+1)*Y_OFFSET;
//                      x3 <= (j+1)*X_OFFSET;
//                      rgb_out <= pixel; //|pixel ? block_colors[j] : 0;
//                      end
////                  else begin
////                      rgb_out <= 0;
////                      end // if color check else
//                  end

//              else begin
//                  // complete repeat of code above with exception of indexing into block color array
//                   if( block_colors[(i*10)+j] >= 1) begin // if there is a piece of a tetronimo at this location
//                      x1 <= j*X_OFFSET;
//                      x2 <= (j+1)*X_OFFSET;
//                      y1 <= i*Y_OFFSET;
//                      y2 <= (i+1)*Y_OFFSET;
//                      y3 <= (i+1)*Y_OFFSET;
//                      x3 <= (j+1)*X_OFFSET;
//                      rgb_out <= pixel;// |pixel ? block_colors[ (i*10) + j ] : 0;
//                      end
////                  else begin
////                      rgb_out <= 0;
////                      end // if color check else
//                  end
//              j <= j == 9? 0: j+1;
//              end //j loop
//          i <= i == 19? 0: i + 1;
//          end // i loop

//          end //always_ff vclock
endmodule

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Create Date: 11/20/2019 06:18:09 PM
// Design Name:
// Module Name: matrix
// Project Name:
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module matrix #(parameter HEIGHT = 1024,
                          WIDTH   = 768)
              (  input sysclk,
                 input reset,
                 input signed [23:0] tri_array [2:0][3:0],
                 input signed [23:0] field_of_view,
                 input set_matrix_values,
                 output logic signed [23:0] corrected_array [2:0][3:0]
                 );
```

```
logic signed [23:0] correction_matrix [6:0];
logic signed [23:0] aspect_ratio;
logic signed [23:0] fov_multiplier;
//logic signed [23:0] z_near, z_far, z_ratio;
//logic signed [23:0] z_near_q;
logic signed [47:0] temp, temp1, temp2, temp3, temp4, temp5;

aspect_ratio modifier ( .sysclk(sysclk),
                        .height(HEIGHT),
                        .width(WIDTH),
                        .aspectRatio(aspect_ratio));

 field_of_view find_fov(.sysclk(sysclk),
                       .FOV(field_of_view),
                       .fov_multiplier(fov_multiplier)
                       );



always_ff @(posedge sysclk) begin
    if(reset  ||  set_matrix_values) begin
//      temp =
        correction_matrix[0] <=  (aspect_ratio*fov_multiplier) >>>12;
        correction_matrix[1] <= fov_multiplier;
        correction_matrix[2] <= 13'b1__0000_0010_1001;
        correction_matrix[3] <= -14'b10__0000_0010_1001;
        correction_matrix[4] <= 1;
       correction_matrix[5] <= 20'b01010_0001_1001_1010;
       correction_matrix[6] <= 20'b01010_0001_1001;
        end // if reset
    else begin
        corrected_array[3][3] <= (tri_array[3][3]*correction_matrix[0])>>>20;
        corrected_array[3][2] <= (tri_array[3][2]*correction_matrix[1])>>>20;
        corrected_array[3][1] <= (tri_array[3][1]*correction_matrix[1])>>>20;
//      temp <= (tri_array[3][0]*correction_matrix[2])>>>20;
        corrected_array[3][0] = ((tri_array[3][0]*correction_matrix[2])>>>20) - correction_matrix[6];
//      temp1 <= correction_matrix[5] >>>4;

        corrected_array[2][3] <= (tri_array[2][3]*correction_matrix[0])>>>20;
        corrected_array[2][2] <= (tri_array[2][2]*correction_matrix[1])>>>20;
        corrected_array[2][1] <= (tri_array[2][1]*correction_matrix[1])>>>20;
        temp <= (tri_array[3][0]*correction_matrix[2])>>>20;
        corrected_array[2][0] <= ((tri_array[3][0]*correction_matrix[2])>>>20) - (correction_matrix[5] >>>4);
//      temp2 <= correction_matrix[5] >>>4;
//      corrected_array[3][0] <= temp - temp2;

        corrected_array[1][3] <= (tri_array[1][3]*correction_matrix[0])>>>20;
        corrected_array[1][2] <= (tri_array[1][2]*correction_matrix[1])>>>20;
        corrected_array[1][1] <= (tri_array[1][1]*correction_matrix[1])>>>20;
        corrected_array[1][0] <= ((tri_array[1][0]*correction_matrix[2])>>>20) - correction_matrix[6];
//      temp <= (tri_array[3][0]*correction_matrix[2])>>>20;
//      temp3 <= correction_matrix[5] >>>4;
        end
    end // always_FF
endmodule
```