

# Realistic 3D Gaming

## 6.111 Final Project

Ranbel Sun, Daniel Whitlow

### ***Abstract***

In order to meet the demands for a realistic 3D gaming experience, extensive computation resources are required, either in software, hardware, or both. Rendering high quality graphics in real time monopolizes available resources to the point where there is very little left for the actual game. The advantage of handling rendering in hardware is that designing the hardware to handle this task from the ground up allows for more efficiency at the cost of flexibility (think dedicated GPUs vs. CPUs). Our goal is to create a 3D rendering system that will produce realistic-looking graphics, and apply them to implement a 3D game.

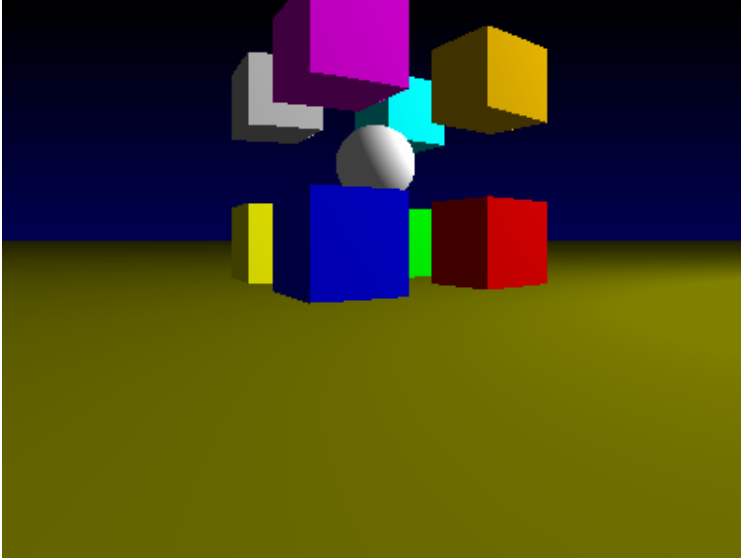
# Table of Contents

Overview (Daniel).....	1
Description (Daniel) .....	2
Scene Data Formats (Daniel) .....	3
3D Renderer (Daniel).....	3
Input Sequencer (Daniel) .....	4
Ray Generator (Daniel).....	4
Intersection Tester (Daniel).....	5
Illuminator (Daniel).....	5
Painter (Daniel) .....	6
Game Logic (Ranbel).....	6
Game Controller (Ranbel).....	7
Game FSM (Ranbel) .....	8
Collision Detector (Ranbel) .....	9
Pseudo-Random Object Generator (Ranbel).....	10
Video Output (Ranbel).....	11
Frame Buffer (Ranbel) .....	11
Background Image Processor (Ranbel).....	11
XVGA (Ranbel) .....	11
Testing.....	12
3D Renderer (Daniel).....	12
Game Controller (Ranbel).....	13
Game FSM (Ranbel) .....	13
Collision Detector (Ranbel) .....	14
Pseudo-Random Object Generator (Ranbel).....	14
Frame Buffer (Ranbel) .....	14
Background Image Generator (Ranbel) .....	14
Conclusion (Daniel & Ranbel).....	15
Acknowledgements .....	15
References .....	16
Appendices.....	17
Appendix A: 3D Renderer Test Environment Module (Daniel).....	17
Appendix B: Ray Tracer Module (Daniel).....	26
Appendix C: Ray Generator Module (Daniel) .....	29
Appendix D: Intersect Tester Module (Daniel).....	31
Appendix E: Illuminator Module (Daniel).....	36
Appendix F: Game Controller (Ranbel).....	41
Appendix G: Game FSM (Ranbel).....	46
Appendix H: Collision Detector (Ranbel).....	49
Appendix I: Shape Collision Detection Modules (Ranbel).....	50
Appendix J: Frame Buffer (Ranbel).....	53
Appendix K: Pseudo-Random Object Generator (Ranbel).....	56
Appendix L: 2D Display (Ranbel) .....	58
Appendix M: 2D Test Bench (Ranbel) .....	61

## Table of Figures

Figure 1: Ray casted image from the software prototype renderer .....	1
Figure 2: Screenshot from Nintendo's Starfox on the SNES .....	2
Figure 3: High level project block diagram .....	2
Figure 4: Formats of scene objects passed to the renderer.....	3
Figure 5: 3D Renderer block diagram.....	4
Figure 6: Components of Phong Reflection [1] .....	5
Figure 7: Game logic block diagram.....	7
Figure 8: Game FSM diagram.....	9
Figure 9: The renderer prototype in action.....	12

## Overview



**Figure 1: Ray casted image from the software prototype renderer**

Ray tracing is a 3D rendering algorithm that basically operates like a camera in reverse. Where normally, rays of light would be emitted from a source, reflecting off one or more objects before reaching the viewer, ray tracing casts rays from the viewer, called primary rays, and tests for intersection with an object, then traces reflection rays and light rays from that object.

Ray casting is similar to ray tracing, but it only generates primary rays, not reflection or light rays. It is considerably less expensive to implement, but it is significantly less realistic.

In either case, ray calculations can be performed independently of one another, making these rendering algorithms prime candidates for parallelized implementation in hardware.

With the 3D renderer, we intended to implement a game similar to Nintendo's Starfox—a linear flying shooter originally made for the SNES.

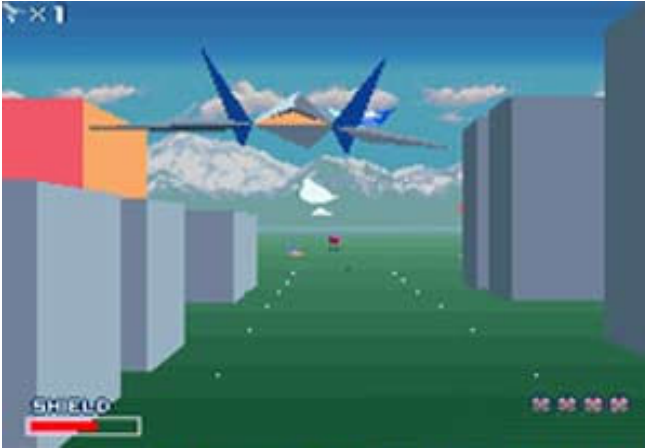


Figure 2: Screenshot from Nintendo's Starfox on the SNES

## Description

At a high level, the game logic module accepts user input from the labkit, and processes this input to create scene data to pass to the 3D renderer. The renderer processes the scene data and outputs pixel data to the background image processor, where a background image is added to the pixels which did not have an object to display. Finally, the pixel data is written to the frame buffer. At the same time, the XVGA module generates the signals required for the 1024x768 VGA display, and reads the color data from the frame buffer.

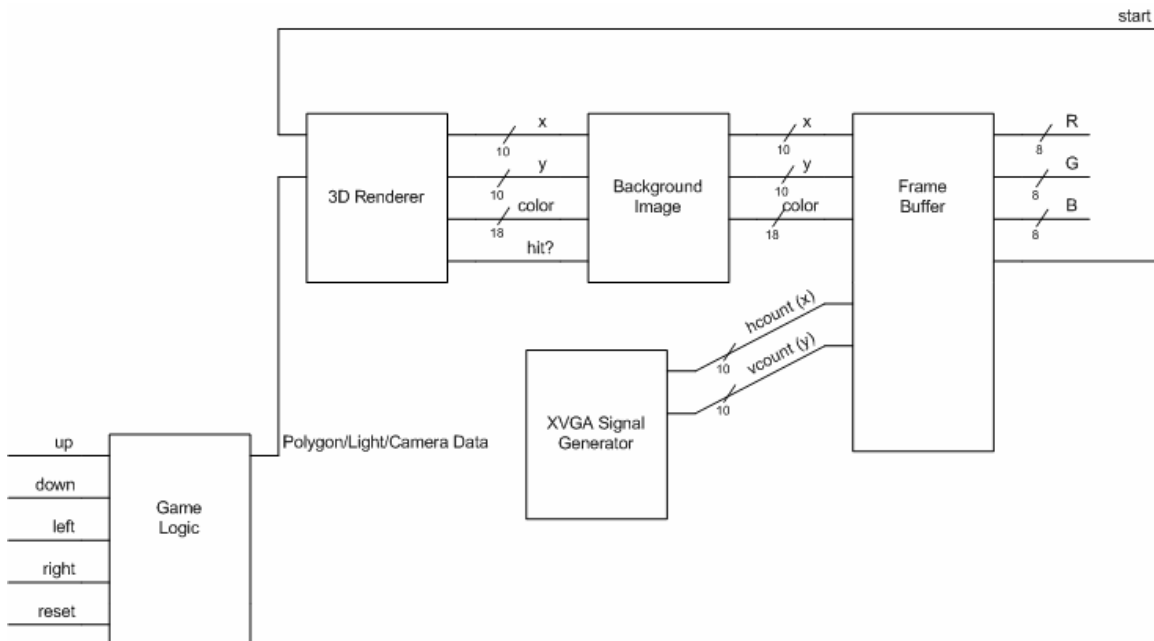


Figure 3: High level project block diagram

## Scene Data Formats

The scene data is represented as a list of shapes, lights, and a camera (the viewpoint). Shapes are 65-bit values with their type (plane, sphere, or box) defined by the 2 high order bits. Lights are 36-bit values, and the camera is a 45-bit value. The exact formats are detailed in the following figure:

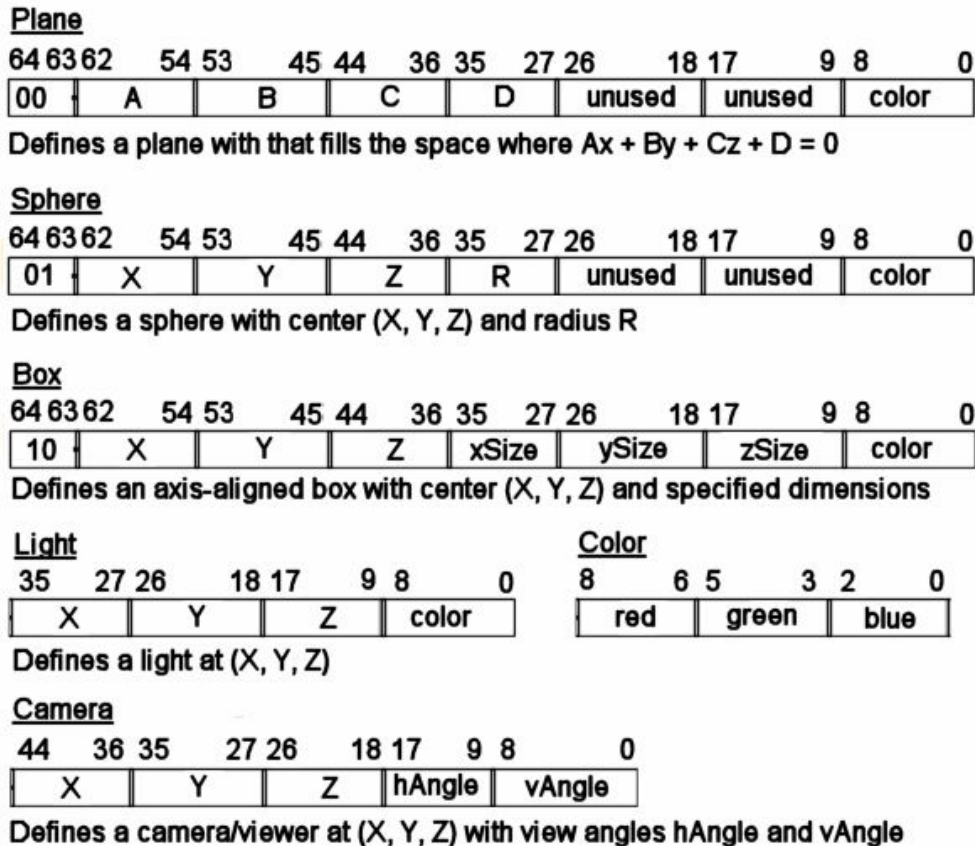


Figure 4: Formats of scene objects passed to the renderer

All position/distances fields (X, Y, Z, A, B, C, D, R, xSize, ySize, and zSize) are encoded as signed 9-bit integers. Color fields are encoded as 9-bit RGB values (3 bits for each color). Angle fields (hAngle, vAngle) are unsigned and encoded as  $INPUT * 2\pi / 2^9$  radians, where INPUT is the value of the field. hAngle specifies the camera's orientation on the xy-plane, and vAngle specifies the angle between the camera's view vector and the xy-plane. Interpreted using a 3D polar coordinate system, hAngle would be equivalent to theta, and vAngle would be equivalent to phi.

## 3D Renderer

The 3D Renderer accepts a list of shapes, lights, and a viewpoint (camera), and returns a series of 18-bit colors corresponding to pixels on the display, along with a done signal when its output data is ready, and a hit signal indicating that the ray for the current display pixel intersected something. It is partially pipelined, with a latency of 144 cycles and a throughput of 1/8 using a 65Mhz clock. The throughput of less than 1 was chosen

in order to conserve resources, since the renderer uses a large number of dividers and square root modules for intersection testing and illumination.

All of the numbers used to store results from 3D math operations are 18-bit signed fixed point precision numbers with 14-bits of precision. Input values are limited within the range of -1.0 to 1.0 to prevent overflows, and the relatively high precision helps to prevent most underflows.

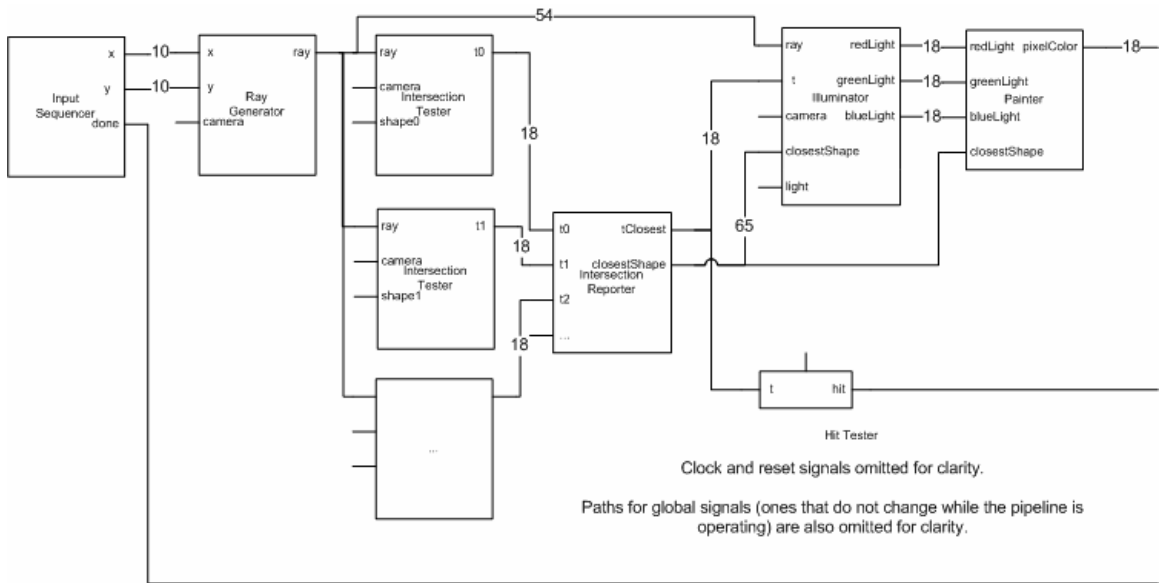


Figure 5: 3D Renderer block diagram

## Input Sequencer

The input sequencer produces the done signal used to signal the frame buffer to store pixel colors from the renderer, as well as the (x, y) coordinate pair corresponding to a pixel on the display. This pair is passed to the ray generator to determine the primary ray that corresponds to this display pixel.

The input sequencer starts at (0, 0), and outputs the coordinate pairs from left to right, top to bottom. If a reset or start signal is sent to the module, it restarts at (0, 0).

## Ray Generator

The ray generator accepts an (x, y) pair from the input sequencer, as well as the camera signal from the ray tracer module. Using the angles specified by the camera and the coordinate pair from the input sequencer, it produces a normalized vector corresponding to the delta-t of the primary ray for the specified display pixel.

## Intersection Tester

The intersect module accepts a camera and normalized ray, as well as a shape to test for intersection. It outputs the time value of the intersection (the amount that the delta-t of the ray must be multiplied by to reach the intersection point from the viewer point), or a negative value if the ray never intersects the object.

Since each intersection tester is only responsible for testing intersection with one shape, one tester module is required for each shape in the scene. The advantage of this is that rendering time does not scale as the number of on-screen objects increases. The disadvantage is that the maximum number of objects that can be in a scene is limited by hardware resources.

## Illuminator

The illuminator module uses a modified form of Phong reflection to determine the amount of light shed on a particular spot in the rendered scene. Phong reflection models the total amount of light on an area by summing three factors: ambient light, diffuse light, and specular light. Ambient light is a generally small factor which prevents areas that are unlit by actual light sources from being pitch black. Diffuse light models the way that rough surfaces are illuminated, and specular light models reflection of light on shiny surfaces. During prototype construction, specular light did not appear to add significant value to the rendered scenes, so it was omitted from the hardware renderer for performance reasons.

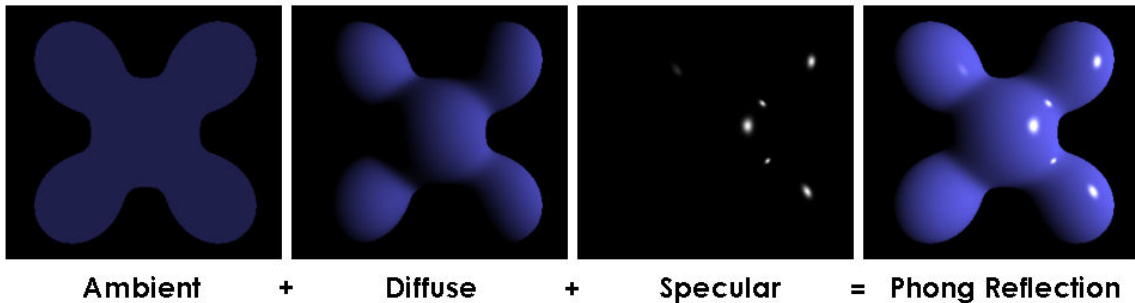


Figure 6: Components of Phong Reflection [1]

Ambient light is actually added in the painter module, so if multiple illuminators are used, ambient light will only be added once. Diffuse light is determined by getting the non-normalized normal vector,  $N$ , of the shape at the intersection point of the ray and the non-normalized vector,  $L$ , from the intersection point to the light. The factor is then calculated with the formula  $(N * L) / (|N| * |L|)$ . This factor is multiplied by the intensity of each color in the light to determine the amount of diffuse light of each color shed by this light source.

Since each illuminator is only responsible for illumination for one light, one module is required for each light in the scene. The advantage of this is that rendering time does not



scale as the number of on-screen lights increases. The disadvantage is that the maximum number of lights that can be in a scene is limited by hardware resources.

## **Painter**

The painter module accepts the amount of light contributed to the area by all light sources, adds ambient light, and limits the total factor to a maximum of 1 and a minimum of 0. It then multiplies this factor by the color of the shape provided, and returns an 18-bit RGB color as a result.

## ***Game Logic***

The game logic was designed to meet the ray tracer's specifications, handling only primitive shapes and outputting shape, light, and camera data in the format outlined in Figure 3. The game mechanics were inspired by the Starfox game, in which the user controls a "ship" to avoid obstacles that are constantly moving closer in the third dimension. The user may also adjust the camera to view the scene from various angles and select the speed at which obstacles move forward in time. Since chip space requirements increase with the number of shapes rendered, the number of objects in the scene was limited to 8. The block diagram for the game is shown below:

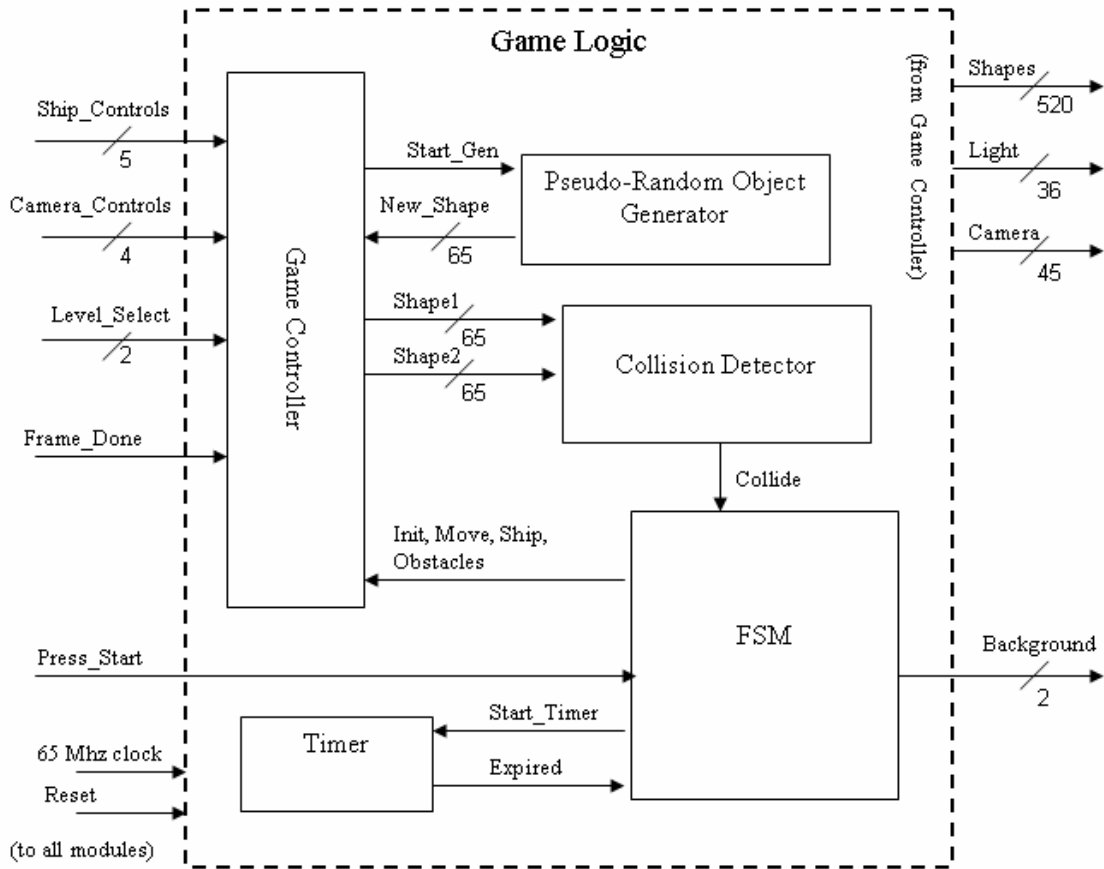


Figure 7: Game logic block diagram

The game logic consists of a main shape buffer controller, a finite state machine, a collision detection module, and a pseudo-random shape generator. The inputs correspond to user controls, and the shape, camera, and light buffer is read by the ray tracer. The FSM outputs a 2-bit background selection to the background image processor.

### Game Controller

The game controller module maintains the buffer for the world's scene information and updates it based on user input. Input is currently taken from the labkit buttons and switches, but a PS/2 keyboard using the provided module would allow for more input possibilities. The ship can be controlled in two dimensions (y and z) and can also be rotated 90 degrees by interchanging the y and z sizes. Movement of the ship is bounded by the world coordinate system such that it does not overflow past the screen boundary. In the world coordinate system, (0, 0, 0) corresponds to the center of the screen and (0, -1, 0) would correspond to a 2-pixel offset to the left. This representation is used to save on the number of shape representation bits and to facilitate ray tracing calculations. The camera vertical and horizontal angles are adjustable, and although the light is kept at a constant location and color for testing, it would not be difficult to assign user control keys to the light parameters.

A BRAM buffer is maintained for the shape data to be read by the 3-D renderer, and it updates when a frame has been processed. The first shape in the buffer is the ship, the second is a plane representing ground, and the other 6 are allocated for obstacles. When an obstacle moves forward past the world's x-coordinate boundary ( $x = +256$ ), the next 5 obstacles are shifted forward in the buffer, the score is incremented, and a new object from the shape generator (at  $x = -200$ ) is stored in the newly vacated shape memory. This way, only the first obstacle needs to be checked for scene drop-off. Upon reset, obstacles are not displayed by setting the shape ID to  $2'b11$ , but they all have an initial  $x$  of 256.

## **Game FSM**

The game display logic can be described by the finite state machine diagramed in the figure below. This module controls what the screen shows at various stages of the game, including what background image is used, whether ship and obstacles are shown, and if they are moving. Upon reset, the FSM is at state  $S\_title$ , which simply displays a welcome image for the game and initializes the number of lives. When the start key is pressed, the state transitions to  $S\_static$ , which shows the background image used for gameplay and the ship. A timer is set so that one start key may be used to transition between consecutive states. The timer module is similar to the one written for Lab 3, using a 1 Hz enable signal from a 65 MHz clock divider to count down the number of seconds.

The obstacles appear in  $S\_move$ , and the move signal triggers the movement and generation of obstacles. When the collision detector module detects a collision, the background image changes, the ship disappears, and the number of lives is decremented. If there are still lives remaining, the module returns to  $S\_static$ . Otherwise, an end screen is displayed and the game must be reset.

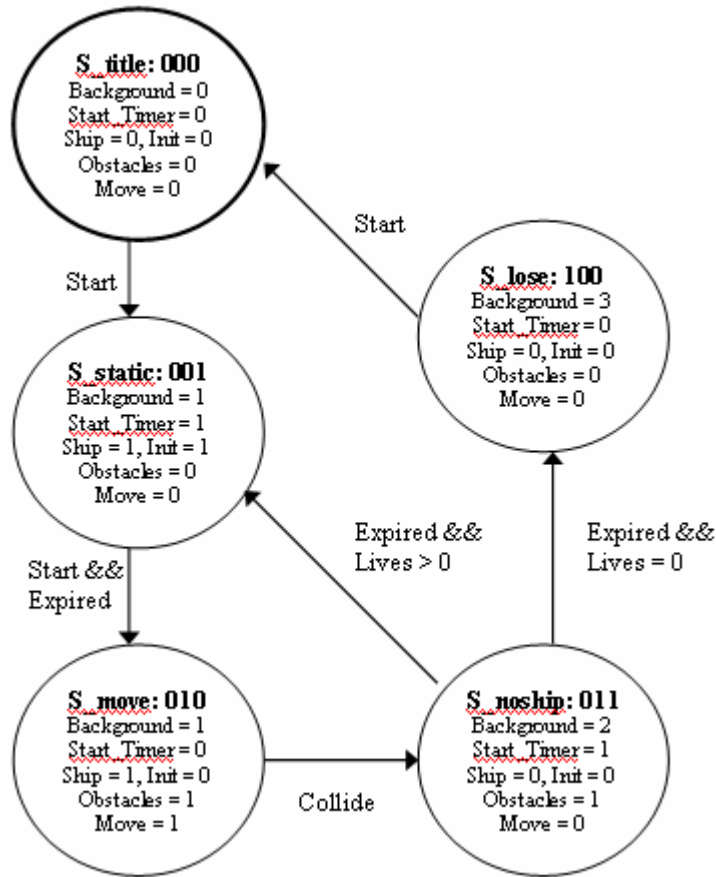


Figure 8: Game FSM diagram

This finite state machine describes the high level display logic that tells the background image processor what backgrounds to display and also tells the game controller if ships and/or obstacles appear and whether or not they are moving. It takes a start signal from external user input, and all other control signals are from modules within the game logic.

### Collision Detector

Since the 3-D renderer was limited to primitive, axis-aligned shapes, bounding volumes were used to check for collision between the 8 shapes displayed on the screen. The collision detector module takes two shapes from the game controller, checks their shape identifiers, and detects if a collision occurred based on object location, size, and speed. Three submodules handled the different cases: sphere to sphere, box to box or sphere, and box to plane. The ship was represented as a box and the plane remained static to represent the ground level. Sphere to plane was not considered because having obstacles overlapping with the plane would add to scene variety and would not hinder game mechanics. Movement in time was accounted for by predicting object dislocation using their velocity vectors.

## **Spheres**

Two spheres are considered to intersect if and only if the distance between their centers is at most the sum of their radii.

## **Axis-Aligned Boxes and Spheres**

Axis-aligned boxes are tested by comparing their minimum and maximum extrema. Objects use center-extent representation and the sizes are in terms of half the total dimensions, so the distance between the centers is compared to the sum of the distances to the edges of the box. A box and sphere collision is checked in a similar fashion by extending box dimensions by the radius of the sphere in each direction and testing if the sphere center falls within the bounding volume. This method may detect false collisions when the sphere is located at a box corner, but it was chosen for the low computation cost. If more precision is needed, a tree hierarchy could be implemented. If a box to sphere collision is detected, we would then compute the point in the box closest to the sphere's center and see if this point is contained in the sphere.

## **Boxes and Planes**

This module checks for collision between the ship and a ground plane by comparing the z edges of the box to the constant z location of the plane. The plane may be located either above or below the ship.

## **Pseudo-Random Object Generator**

In order to keep game-play interesting and to avoid hard-coding and storing different shape information, the object generator uses 4 pseudo-random numbers to create new shapes in the encoding shown in Figure 3. The module outputs a new shape when it receives a start signal from the game controller. If the shape ID happens to be 2'b11, a shape is output with no pixel color such that the timing of object appearance is also variable. Box sizes are half of the total dimension to avoid division costs in collision detection. Since the object sizes are signed for compatibility, the first bit is always set to 0. The box sizes are limited to less than the maximum size (second bit is also 0) so that objects do not fill the entire screen.

## **Linear Shift Register**

A linear shift register (LFSR) module generates 9-bit pseudo-random numbers using a feedback function. A 9-bit register is created with a specified starting state. Next, the XOR of two tap bits (bits 0 and 5) is computed, the register contents are shifted right by 1, and the XOR result is moved to the high order bit. [2]

## **Video Output**

### **Frame Buffer**

The frame data is stored in the two onboard ZBT SRAMs. This module acts as one memory device to read and write pixel data from rendered pixels that have passed through the Background Image Processor. This is accomplished with a double buffer system, the idea being that at any point in time, one ZBT is being read and the other is being written to. To read, the module inputs hcount and vcount from the XVGA module to determine the correct pixel to output to the display. To write, it inputs RGB pixel values along with a done signal. The done signal allows pixels to arrive with variable throughput. The pixel data is provided in the order they will be displayed, so the frame buffer keeps count of the display coordinate to write the pixel to the correct address.

Because the project uses 1024 x 768 resolution and 18-bit RGB, it would require over 700K memory locations if one pixel was stored per location. Each ZBT has only 512K addresses, so a more optimal memory utilization method is necessary. Since the ZBT has 36-bit wide memory, two pixels are written to each location. The 10 higher order address bits correspond to the pixel location, and the 9 lower order bits to pixelx[9:1]. Since each incoming pixel increments pixelx by 1, this scheme conveniently assigns two consecutive pixels to one location.

When the last pixel in a frame has been written, the two ZBTs swap role and a flip signal is sent to the input sequencer to start rendering a new frame.

### **Background Image Processor**

The Background Image Processor displays images behind the rendered objects instead of black. Pixel data from the 3-D renderer is passed through this module with a Hit signal. Hit indicates whether the pixel is rendered or not. If the pixel is not rendered, the module replaces it with the corresponding pixel in the background image. A basic version of this module is currently implemented within both the 3-D renderer and game logic test benches. The 3-D renderer displays a gradient when no intersection is detected. The game logic outputs background selector bits to the Background Image Processor to choose between four different images. The images generated show character strings corresponding to the state of the game. The idea would be to eventually have four high resolution images stored in ROM.

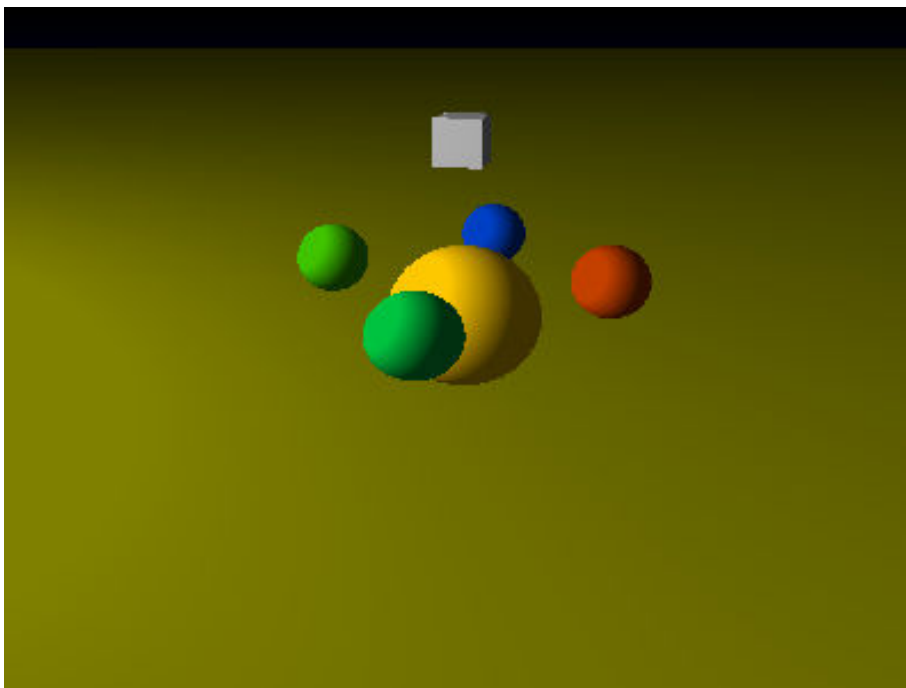
### **XVGA**

The video output module was provided by the 6.111 staff and is configured to 1024 x 768 resolution. Each 6-bit color from the 18-bit RGB is shifted left by 2 to send 24-bit color to the display.

# Testing

## ***3D Renderer***

The first stage of developing the 3D renderer was writing a prototype in software. This helped to debug issues with the ray casting and illumination algorithms without the hardware adding an extra layer of complexity. After the prototype was polished to a satisfactory degree, the design was split into loosely coupled modules and implemented in Verilog. By organizing the design in this manner, test benches could be created to test each module individually, allowing bugs to be isolated more easily. The lowest level math modules (fixed point multipliers, dividers, square roots, etc.) were tested with a full range of test cases, while mid level modules such as the intersect tester and illuminator were tested with a few test cases, supplemented by a full system test on the FPGA.



**Figure 9: The renderer prototype in action**

In debugging the renderer, precision issues, underflows, and overflows were common problems. At the root of these problems were 18-bit wide fixed point numbers with 9-bits of precision, which was later changed to 14-bits of precision in an attempt to improve performance. In retrospect, it may have been a better idea to use additional hardware and higher latency in order to take advantage of floating point numbers. Another big problem was small bugs introduced to the ray casting and illumination algorithms during the process of optimizing the pipeline. A faulty assumption was made that the calculations being performed were the correct calculations, since the algorithms worked in the software renderer. However, the implementation of the software renderer was much different, and as such, it did not contain these optimizations, so it took a while to locate the bugs.

Later on, the renderer was changed to use higher-precision numbers, at the cost of maximum magnitude. It could only do this with either high precision, making the edges of the shapes appear more smooth and realistic, or with working shading, as increasing the precision somehow removed most of the functionality of the illuminator.

Realizing that since using low (9-bit) precision numbers was the only change before the illuminator malfunctioned, an attempt was made to keep high (14-bit) precision in the rest of the renderer while reverting to low precision in the illuminator. Unfortunately, this caused the illuminator to stop working completely, and everything was illuminated only with ambient light, so the change was reverted.

Testing for multiple shapes was also performed independently of the high precision change, but long build times resulted in the high precision feature being prioritized. After the project deadline passed, it became apparent that the multi-shape bug was caused by passing the first shape in the scene to the illuminator and hit checker modules, rather than the intersected shape.

## ***Game Controller***

Because the game was designed for a 3-D environment and because the renderer was not complete, an alternate 2-D display was needed to test and debug the game logic. Rectangular and circular blobs represented boxes, planes, and spheres, and the third dimension was output to the logic analyzer. An unexpected amount of time was spent setting up this system. The world coordinates had to be translated to hcount and vcount pixel coordinates, keeping signed numbers in mind. Signed numbers and alternate coordinates was the reason why the ship was not showing up on the display and why it did not move in the proper direction. There were also timing issues which were sufficiently resolved by updating the object locations with a slower clock using a divider module). This was actually more representative of the renderer because a start signal needs to be sent from the input sequencer anyways.

When the system was finally displaying, it was noticed that all of the obstacles updated very rapidly. This was caused by two main issues: the obstacle buffer had a bug in which all the shape information shifted left except for the x location, and the initial values for the obstacle x locations were set incorrectly. When these were fixed, an obstacle would remain on the screen for a few seconds before vanishing and being replaced by another obstacle.

## ***Game FSM***

The game logic was tested using labkit switches representing the transition signals. The LED hex display showed the background, move, ship, init, and obstacle outputs. One problem was that the transition from S\_title to S\_static and S\_static to S\_move used the same start signal. Thus, because pressing the start button asserts start for more than 1 cycle, the FSM would not pause in S\_static. This was resolved by setting a timer in S\_Static such that a 2 second wait is required before the transition can occur.



## ***Collision Detector***

The independent collision submodules were tested using a ModelSim testbench, first on static objects and then on moving objects. Static objects were plotted in Matlab for different cases to quickly determine if there is supposed to be an intersection. Afterwards, the ship and an obstacle were displayed on the 2-D system and buttons were assigned to allow the ship to move in three dimensions. A digit on the hex display indicated whether or not a collision occurred, and the logic analyzer was used to view the x coordinates. Unfortunately, this setup was using an early version of the game display testbench (before the random obstacles were able to be displayed) and I was unable to get the two objects to display again for the project checkoff.

## ***Pseudo-Random Object Generator***

This module was tested in ModelSim and verified to be generating shape data with no logical pattern. Finally, the objects on the display were observed to be varying and of a reasonable size.

## ***Frame Buffer***

Debugging this module took a lot more time than expected, mainly due to the tricky timing requirements. One reason it took so long was that I did not know to use ModelSim until several days passed. It proved to be an extremely effective tool at ensuring write operation, namely the 2-cycle write delay, the correct operation of the 2-pixel register in response to done signals, and the correct flipping of the ZBTs. Another oversight was that the ZBTs also needed an inverted clock.

To test the Frame Buffer, pixel data and done signals were sent to the module and the data from the read ZBT was displayed on the screen. First a solid pixel color was shown to display, meaning that data was being written. Then, 8 color bars were displayed to show that the correct pixel location was being read. Initially, color lines were scattered across the screen and constantly moved up the screen. A switch was assigned to disable flip, and when flip was disabled, the image remained constant. One timing issue was the second pixel not being latched properly in the 36 bit register to be written, due to pipelining errors. The frame flip signal was also incorrect, because the ram0\_data and ram1\_data were flipping before the ZBT could write the last pixel. A flip\_delay signal was created to resolve this. Finally, there was a vertical bar of blue on the left side of the screen, which was fixed by delaying hsync and vsync. The frame buffer has occasional spot flickers across the screen which may be fixed using a DCM clock manager.

## ***Background Image Generator***

The game background image processor was tested right after the FSM module and was shown to select the right images corresponding to the different states. The ray tracer

image processor was shown to be working by displaying a color gradient behind the object being rendered.

## **Conclusion**

Although we were unable to get the 3-D game fully integrated and running, we were able to show that it is possible to implement a 3-D renderer in hardware and that it is possible to write a game compatible with its rendering capabilities. The overall design was feasible, but unforeseen project delays and troubleshooting prevented the completion of the project.

In the end, the 3D renderer was capable of displaying one shape with one light at a time with high precision and marginally functional shading. With one or two days more time, it would've been quite possible to get it fully up to spec (8 shapes simultaneously at high precision with working shading). Since the 3D renderer wasn't completed, it was not integrated with the rest of the project, so a makeshift 2D renderer had to be made to display the game logic.

There are many improvements that can be made to the game. The Collision Detector was never fully tested with the rest of the game logic, and it would be ideal to check collision detection for newly generated objects such that it would always be possible to avoid an object. A life gauge that indicates score, as well as storing high resolution background images in ROM, would also be valuable enhancements to the game. In retrospect, perhaps it would have been more constructive to split up the rendering work and get that functioning before working on the game logic. I feel like much of the time and effort put into the project was not effectively spent. However, although the results did not meet goals and expectations, I definitely leave the course with a digital design toolbox, debugging skills, patience, and a better understanding and appreciation for the intricacies of digital systems.

## **Acknowledgements**

We would like to thank all of the 6.111 staff, especially Gim and Alex, for promptly answering questions, helping to debug modules, and for spending long, long hours in lab with all of us. We would also like to thank Adam for his design guidance on the 3D renderer.

## References

- [1] Phong Shading. (2008, November 04). Wikipedia. Retrieved Dec 9, 2008 from [http://en.wikipedia.org/wiki/Phong\\_shading](http://en.wikipedia.org/wiki/Phong_shading)
- [2] Tap Sequence Lists. (2001, August 06). Linear Feedback Shift Registers. Retrieved Dec 9, 2008 from <http://homepage.mac.com/afj/taplist.html>

# Appendices

## Appendix A: 3D Renderer Test Environment Module

```
////////////////////////////////////
//
// Pushbutton Debounce Module (video version)
//
////////////////////////////////////

module debounce (input wire reset, clock, noisy,
                 output reg clean);

    reg [19:0] count;
    reg new;

    always @(posedge clock)
        if (reset) begin new <= noisy; clean <= noisy; count <= 0; end
        else if (noisy != new) begin new <= noisy; count <= 0; end
        else if (count == 650000) clean <= new;
        else count <= count+1;

endmodule

////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
////////////////////////////////////

module lab5skeleton (

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

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

    clock_27mhz,
```

```

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

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

switch,

led,
);

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

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

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

input wire clock_27mhz;

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

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

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

// SRAMs
assign ram0_adv_ld = 1'b0;

assign ram0_cen_b = 1'b0;

```

```

assign ram0_ce_b = 1'b0;
assign ram0_oe_b = 1'b0;
assign ram0_bwe_b = 4'h0;

assign ram1_adv_ld = 1'b0;
assign ram1_cen_b = 1'b0;
assign ram1_ce_b = 1'b0;
assign ram1_oe_b = 1'b0;
assign ram1_bwe_b = 4'h0;

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

// use FPGA's digital clock manager to produce a
// 65MHz clock (actually 64.8MHz)
wire clock_65mhz_unbuf,clock_65mhz;
DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_65mhz_unbuf));
// synthesis attribute CLKFX_DIVIDE of vclk1 is 10
// synthesis attribute CLKFX_MULTIPLY of vclk1 is 24
// synthesis attribute CLK_FEEDBACK of vclk1 is NONE
// synthesis attribute CLKIN_PERIOD of vclk1 is 37
BUFG vclk2(.O(clock_65mhz),.I(clock_65mhz_unbuf));

// power-on reset generation
wire power_on_reset; // remain high for first 16 clocks
SRL16 reset_sr (.D(1'b0), .CLK(clock_65mhz), .Q(power_on_reset),
    .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
defparam reset_sr.INIT = 16'hFFFF;

// ENTER button is user reset
wire reset,user_reset;
debounce
db1(.reset(power_on_reset),.clock(clock_65mhz),.noisy(~button_enter),.clean(user_reset)
);
assign reset = user_reset | power_on_reset;

wire forward, back, left, right, up, down, hRotate, vRotate;
debounce
db2(.reset(power_on_reset),.clock(clock_65mhz),.noisy(~button_up),.clean(forward));
debounce
db3(.reset(power_on_reset),.clock(clock_65mhz),.noisy(~button_down),.clean(back));
debounce
db4(.reset(power_on_reset),.clock(clock_65mhz),.noisy(~button_left),.clean(left));

```

```

debounce
db5(.reset(power_on_reset),.clock(clock_65mhz),.noisy(~button_right),.clean(right));
debounce db6(.reset(power_on_reset),.clock(clock_65mhz),.noisy(~button3),.clean(up));
debounce
db7(.reset(power_on_reset),.clock(clock_65mhz),.noisy(~button2),.clean(down));
debounce
db8(.reset(power_on_reset),.clock(clock_65mhz),.noisy(~button1),.clean(hRotate));
debounce
db9(.reset(power_on_reset),.clock(clock_65mhz),.noisy(~button0),.clean(vRotate));

wire [1:0] dispswitch = switch[7:6];
wire [2:0] shapswitch = switch[5:3];
wire [2:0] lightswitch = switch[2:0];

// generate basic XvGA video signals
wire [10:0] hcount;
wire [9:0] vcount;
wire hsync,vsync,blank;
xvga xvga1(.vclock(clock_65mhz),.hcount(hcount),.vcount(vcount),
           .hsync(hsync),.vsync(vsync),.blank(blank));

assign ram0_clk = ~clock_65mhz; //helps meet timing requirements of ram
assign ram1_clk = ~clock_65mhz;

//ray trace test
reg [64:0] shape;
reg [35:0] light;
reg signed [9:0] x, y, z, hAngle, vAngle;
wire [44:0] camera;
wire done, hit;
reg start;
wire [17:0] pixelcolor;
always @(shapswitch)
begin
    case (shapswitch)
        3'b000: shape = {2'b00, 9'sb0_0000_0000, 9'sb0_0000_0000, 9'sb1_1000_0000,
9'sb0_1000_0000, 9'sb0_0000_0000, 9'sb0_0000_0000, 9'b111_111_111};
        3'b001: shape = {2'b00, 9'sb0_0000_0000, 9'sb0_0000_0000, 9'sb0_1000_0000,
9'sb0_1000_0000, 9'sb0_0000_0000, 9'sb0_0000_0000, 9'b111_111_111};
        3'b010: shape = {2'b00, 9'sb0_0000_0000, 9'sb1_1000_0000, 9'sb0_0000_0000,
9'sb0_1000_0000, 9'sb0_0000_0000, 9'sb0_0000_0000, 9'b111_111_111};
        3'b011: shape = {2'b00, 9'sb0_0000_0000, 9'sb0_1000_0000, 9'sb0_0000_0000,
9'sb0_1000_0000, 9'sb0_0000_0000, 9'sb0_0000_0000, 9'b111_111_111};
        3'b100: shape = {2'b01, 9'sb0_0010_0000, 9'sb0_0000_0000, 9'sb0_0000_0000,
9'sb0_0010_0000, 9'sb0_0000_0000, 9'sb0_0000_0000, 9'b111_111_111};
    endcase
end

```

```

    3'b101: shape = {2'b01, 9'sb0_0010_0000, 9'sb0_0000_0000, 9'sb0_0000_0000,
9'sb0_1000_0000, 9'sb0_0000_0000, 9'sb0_0000_0000, 9'b111_111_111};
    3'b110: shape = {2'b10, 9'sb0_0010_0000, 9'sb0_0000_0000, 9'sb0_0000_0000,
9'sb0_0010_0000, 9'sb0_0010_0000, 9'sb0_0010_0000, 9'b111_111_111};
    default: shape = {2'b10, 9'sb0_0010_0000, 9'sb0_0000_0000, 9'sb0_0000_0000,
9'sb0_1000_0000, 9'sb0_1000_0000, 9'sb0_1000_0000, 9'b111_111_111};
    endcase
end
always @(lightswitch)
begin
    case (lightswitch)
        3'b000: light = {9'sb0_0000_0000, 9'sb0_0000_0000, 9'sb0_0000_0000,
9'b111_111_111};
        3'b001: light = {9'sb0_0000_0000, 9'sb0_0000_0000, 9'sb0_0000_0000,
9'b111_000_000};
        3'b010: light = {9'sb0_0000_0000, 9'sb0_0000_0000, 9'sb0_0000_0000,
9'b000_111_000};
        3'b011: light = {9'sb0_0000_0000, 9'sb0_0000_0000, 9'sb0_0000_0000,
9'b000_000_111};
        3'b100: light = {9'sb0_1000_0000, 9'sb0_0000_0000, 9'sb0_0000_0000,
9'b111_111_111};
        3'b101: light = {9'sb0_0000_0000, 9'sb0_1000_0000, 9'sb0_0000_0000,
9'b111_111_111};
        3'b110: light = {9'sb0_0000_0000, 9'sb0_0000_0000, 9'sb0_1000_0000,
9'b111_111_111};
        default: light = {9'sb0_0000_0000, 9'sb0_0000_0000, 9'sb1_1000_0000,
9'b111_111_111};
    endcase
end
assign camera = {x, y, z, hAngle, vAngle};
reg [23:0] moveCounter;
always @(posedge clock_65mhz)
begin
    //allow camera to move
    if (reset)
    begin
        x <= 9'sb0_0000_0000;
        y <= 9'sb0_0000_0000;
        z <= 9'sb0_0000_0000;
        hAngle <= 9'sb0_0000_0000;
        vAngle <= 9'sb0_0000_0000;
        moveCounter <= 0;
    end
    else
    begin
        moveCounter <= moveCounter + 1;
    end
end

```



```

if (moveCounter == 0)
begin
  if (forward)
    x <= x + 1;
  else if (back)
    x <= x - 1;
  else if (left)
    y <= y - 1;
  else if (right)
    y <= y + 1;
  else if (up)
    z <= z + 1;
  else if (down)
    z <= z - 1;
  else if (hRotate)
    hAngle <= hAngle + 5'b10000;
  else if (vRotate)
    vAngle <= vAngle + 5'b10000;
  end
end
end
ray_tracer rt(clock_65mhz, reset, start, camera, shape, light, done, hit, pixelcolor);

//ghetto background image generator
// needed to verify buffer operation independently of ray tracer
reg [9:0] dispX, dispY;
reg [17:0] writepixel;
always @(posedge clock_65mhz)
begin
  if (reset)
  begin
    dispX <= 0;
    dispY <= 0;
  end
  else if (done)
  begin
    if ((dispX != 1023 || dispY != 767))
    begin
      dispX <= (dispX == 1023) ? 0 : (dispX + 1);
      dispY <= (dispX == 1023) ? (dispY + 1) : dispY;
    end
  else
  begin
    dispX <= 0;
    dispY <= 0;
    start <= 1;
  end
end

```

```

        end
    end
    else if (start)
        start <= 0;

    if (hit)
        writepixel <= pixelcolor;
    else
        writepixel <= {1'b0, dispX[9:5], 6'b00_0000, 1'b0, dispY[9:5]};
    end

    wire flip;
    wire [17:0] pixelout;
    reg [17:0] pixel;
    framebuffer
    test_buffer(.clk(clock_65mhz), .reset(reset), .done(done), .flip(flip), .pixelcolor(writepixel),
    .hcount(hcount), .vcount(vcount), .vgacolor(pixelout),
    .ram0_data(ram0_data), .ram0_address(ram0_address),.ram0_we_b(ram0_we_b), .ram1_data(ram1_data),
    .ram1_address(ram1_address), .ram1_we_b(ram1_we_b), .flipswitch(1'b1), .weswitch(1'b1));

    always@ (posedge clock_65mhz) begin
        pixel <= pixelout;
    end

    reg [5:0] hdelay, vdelay, bdelay;
    reg [17:0] rgb;
    reg b,hs,vs;
    always @(posedge clock_65mhz)
    begin
        hdelay <= {hdelay[4:0], hsync};
        vdelay <= {vdelay[4:0], vsync};
        bdelay <= {bdelay[4:0], blank};
        hs <= hdelay[5];
        vs <= vdelay[5];
        b <= bdelay[5];
        rgb <= pixel;
    end

    // VGA Output. In order to meet the setup and hold times of the
    // AD7125, we send it ~clock_65mhz.
    assign vga_out_red = {rgb[17:12], 2'b0};
    assign vga_out_green = {rgb[11:6], 2'b0};

```

```

assign vga_out_blue = {rgb[5:0], 2'b0};
assign vga_out_sync_b = 1'b1; // not used
assign vga_out_blank_b = ~b;
assign vga_out_pixel_clock = ~clock_65mhz;
assign vga_out_hsync = hs;
assign vga_out_vsync = vs;

//debug display
reg [63:0] dispdata;
wire [63:0] dispdataWrite, dispdataCamera, dispdataShape, dispdataMove;
assign dispdataWrite = {4'b0000, pixelcolor[17:14], pixelcolor[11:8], pixelcolor[5:2],
4'b0000, writepixel[17:14], writepixel[11:8], writepixel[5:2], 6'b000000, dispX[9:0],
6'b000000, dispY[9:0]};
assign dispdataCamera = {3'b000, x, 3'b000, y, 3'b000, z, hAngle, 3'b000, vAngle,
3'b000, 4'b0000};
assign dispdataShape = shape[64:1];
assign dispdataMove = {40'h0000000000, moveCounter};
always @(*)
begin
case (dispswitch)
2'b00: dispdata = dispdataCamera;
2'b01: dispdata = dispdataWrite;
2'b10: dispdata = dispdataShape;
2'b11: dispdata = dispdataMove;
default: dispdata = dispdataWrite;
endcase
end
display_16hex hexdisp1(reset, clock_65mhz, dispdata,
disp_blank, disp_clock, disp_rs, disp_ce_b,
disp_reset_b, disp_data_out);
assign led = ~{done, hit, flip, start, reset, 3'b000};

endmodule

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//
// xvga: Generate XVGA display signals (1024 x 768 @ 60Hz)
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

module xvga(input wire vclock,
output reg [10:0] hcount, // pixel number on current line
output reg [9:0] vcount, // line number
output reg vsync,hsync,blank);

// horizontal: 1344 pixels total

```

```

// display 1024 pixels per line
reg hblank,vblank;
wire hsyncon,hsyncoff,hreset,hblankon;
assign hblankon = (hcount == 1023);
assign hsyncon = (hcount == 1047);
assign hsyncoff = (hcount == 1183);
assign hreset = (hcount == 1343);

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

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

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

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

```

## **Appendix B: Ray Tracer Module**

```
`timescale 1ns / 1ps
`default_nettype none

/**
shapes, lights, and camera should stay constant throughout operation until next start
*/

//TODO: figure out how to port all data over
//input [64:0] shapes [15:0], input [35:0] light,
module ray_tracer (input wire clock, reset, start, input wire [44:0] camera, input wire
[64:0] shape, input wire [35:0] light, output reg done, hit, output reg [17:0] color);
    localparam PIPELINE_STAGE_LENGTH = 8;

    localparam NUM_INTERSECTORS = 8;
    localparam NUM_LIGHTS = 1;

    localparam RED_AMBIENT = 15'sb000_100_000_000_000;
    localparam GREEN_AMBIENT = 15'sb000_100_000_000_000;
    localparam BLUE_AMBIENT = 15'sb000_100_000_000_000;

    localparam POS_ONE = 18'sb000_100_000_000_000_000;

    //output pixel sequence
    wire [9:0] x, y;
    wire pipelineClock, doneStart;
    input_sequencer #(PIPELINE_STAGE_LENGTH(PIPELINE_STAGE_LENGTH))
is(clock, reset, start, x, y, pipelineClock, doneStart);

    //determine ray for display pixel
    wire [53:0] ray;
    ray_generator rg(clock, pipelineClock, reset, start, x, y, camera, ray);

    //TODO: use multiple intersectors
    wire signed [17:0] tClosest;
    intersect_tester it(clock, pipelineClock, reset, start, camera, ray, shape, tClosest);
    wire [64:0] shapeClosest = shape;

    //TODO: make sure this delay is correct
    wire [53:0] rayDelayed;
    pipeline #(DELAY(6), .WIDTH(54)) rayDelay(pipelineClock, reset, ray, rayDelayed);

    //do lighting
    wire signed [17:0] redLight, greenLight, blueLight;
```

```
illuminator illum(clock, pipelineClock, reset, start, camera, rayDelayed, shapeClosest,
light, tClosest, redLight, greenLight, blueLight);
```

```
wire signed [17:0] totalRedLight, totalGreenLight, totalBlueLight;
assign totalRedLight = redLight + RED_AMBIENT;
assign totalGreenLight = greenLight + GREEN_AMBIENT;
assign totalBlueLight = blueLight + BLUE_AMBIENT;
```

```
wire signed [17:0] actualRedLight, actualGreenLight, actualBlueLight;
assign actualRedLight = (totalRedLight > POS_ONE) ? POS_ONE : totalRedLight;
assign actualGreenLight = (totalGreenLight > POS_ONE) ? POS_ONE :
totalGreenLight;
assign actualBlueLight = (totalBlueLight > POS_ONE) ? POS_ONE : totalBlueLight;
```

```
wire signed [17:0] shapeRed, shapeGreen, shapeBlue;
assign shapeRed = {1'b0, shape[8:6], {3{shape[6]}}, 11'b000_0000_0000};
assign shapeGreen = {1'b0, shape[5:3], {3{shape[3]}}, 11'b000_0000_0000};
assign shapeBlue = {1'b0, shape[2:0], {3{shape[0]}}, 11'b000_0000_0000};
```

```
wire signed [17:0] pixelRed, pixelGreen, pixelBlue;
FXUmul pixelRedMul(shapeRed, actualRedLight, pixelRed);
FXUmul pixelGreenMul(shapeGreen, actualGreenLight, pixelGreen);
FXUmul pixelBlueMul(shapeBlue, actualBlueLight, pixelBlue);
```

```
//TODO: make sure this delay is correct
wire hitP, hitDelayed;
assign hitP = tClosest > 0;
pipeline #(.DELAY(9), .WIDTH(1)) hitDelay(pipelineClock, reset, hitP, hitDelayed);
```

```
//TODO: make sure this delay is correct
wire doneLine;
pipeline #(.DELAY(144), .WIDTH(1), .ENABLE_RESET(1)) doneLineDelay(clock,
reset, doneStart, doneLine);
```

```
//placeholder code
always @(posedge clock)
begin
  if (reset)
  begin
    done <= 0;
    hit <= 0;
    color <= 18'b000_000_000_000_000;
  end
  else
  begin
    done <= doneLine;
  end
end
```

```
hit <= hitDelayed;  
color <= {pixelRed[16:11], pixelGreen[16:11], pixelBlue[16:11]};  
end  
end  
endmodule
```

## Appendix C: Ray Generator Module

```
`timescale 1ns / 1ps
//^default_nettype none

module ray_generator
  (input wire clock, pipelineClock, reset, start, input wire [9:0] xIn, yIn, input wire [44:0]
  camera, output reg [53:0] ray);
  localparam DISPLAY_WIDTH = 1024;
  localparam DISPLAY_HEIGHT = 768;
  localparam H_VIEW_ANGLE = 12'b0100_0000_0000; // pi / 2 radians
  localparam V_VIEW_ANGLE = 12'b0011_0000_0000; // pi * 3 / 8 radians
  localparam HALF_H_VIEW_ANGLE = H_VIEW_ANGLE / 2;
  localparam HALF_V_VIEW_ANGLE = V_VIEW_ANGLE / 2;
  localparam PIXEL_ANGLE = 12'b0000_0000_0001; // pi / 2048 radians

  //determine ray angles
  // shift input angles into 12-bits
  // representation is x * (2pi/2^12) radians
  wire [11:0] hAngle, vAngle;
  assign hAngle = {camera[17:9], 3'b000};
  assign vAngle = {camera[8:0], 3'b000};

  reg [11:0] theta, phi;
  always @(posedge pipelineClock)
  begin
    theta <= (hAngle - HALF_H_VIEW_ANGLE) + (xIn * PIXEL_ANGLE);
    phi <= (vAngle - HALF_V_VIEW_ANGLE) + (yIn * PIXEL_ANGLE);
  end

  //determine ray equation
  // get sin and cos of angles
  wire signed [14:0] theta_sin, theta_cos, phi_sin, phi_cos;
  reg signed [17:0] theta_sin18, theta_cos18, phi_sin18, phi_cos18;
  sincos
  theta_sincos(.CLK(clock), .THETA(theta), .SINE(theta_sin), .COSINE(theta_cos));
  sincos phi_sincos(.CLK(clock), .THETA(phi), .SINE(phi_sin), .COSINE(phi_cos));
  always @(posedge pipelineClock)
  begin
    theta_sin18 <= {{3{theta_sin[14]}}, theta_sin};
    theta_cos18 <= {{3{theta_cos[14]}}, theta_cos};
    phi_sin18 <= {{3{phi_sin[14]}}, phi_sin};
    phi_cos18 <= {{3{phi_cos[14]}}, phi_cos};
  end

  // get dx, dy, dz of ray
```



```
wire signed [17:0] dx, dy, dz;
FXUmul dxmul(theta_cos18, phi_cos18, dx);
FXUmul dymul(theta_sin18, phi_cos18, dy);
assign dz = phi_sin18;

//package ray components and set other outputs
// ray format is {dx, dy, dz}
always @(posedge pipelineClock)
begin
    ray <= {dx, dy, dz};
end
endmodule
```

## Appendix D: Intersect Tester Module

```
`timescale 1ns / 1ps
`default_nettype none

module intersect_tester
  (input wire clock, pipelineClock, reset, start, input wire [44:0] camera, input wire [53:0]
  ray, input wire [64:0] shape, output reg signed [17:0] t);
  localparam PLANE = 2'b00;
  localparam SPHERE = 2'b01;
  localparam BOX = 2'b10;

  //get camera/ray values
  wire signed [17:0] x, y, z, dx, dy, dz;
  assign x = {{2{camera[44]}}, camera[44:36], 7'b000_0000};
  assign y = {{2{camera[35]}}, camera[35:27], 7'b000_0000};
  assign z = {{2{camera[26]}}, camera[26:18], 7'b000_0000};
  assign dx = ray[53:36];
  assign dy = ray[35:18];
  assign dz = ray[17:0];

  //get shape values
  wire [1:0] type;
  wire signed [17:0] sx, sy, sz, sa, sb, sc;
  wire signed [17:0] sd, sr, xSizeHalf, ySizeHalf, zSizeHalf;

  assign type = shape[64:63];

  assign sx = {{2{shape[62]}}, shape[62:54], 7'b000_0000};
  assign sa = {{2{shape[62]}}, shape[62:54], 7'b000_0000};
  assign sy = {{2{shape[53]}}, shape[53:45], 7'b000_0000};
  assign sb = {{2{shape[53]}}, shape[53:45], 7'b000_0000};
  assign sz = {{2{shape[44]}}, shape[44:36], 7'b000_0000};
  assign sc = {{2{shape[44]}}, shape[44:36], 7'b000_0000};
  assign sd = {{2{shape[35]}}, shape[35:27], 7'b000_0000};
  assign sr = {{2{shape[35]}}, shape[35:27], 7'b000_0000};
  assign xSizeHalf = {{2{shape[35]}}, shape[35:27], 7'b000_0000} >>> 1;
  assign ySizeHalf = {{2{shape[26]}}, shape[26:18], 7'b000_0000} >>> 1;
  assign zSizeHalf = {{2{shape[17]}}, shape[17:9], 7'b000_0000} >>> 1;

  wire signed [17:0] xL, xH, yL, yH, zL, zH;
  diffTwo xLDiff(sx, xSizeHalf, xL);
  diffTwo yLDiff(sy, ySizeHalf, yL);
  diffTwo zLDiff(sz, zSizeHalf, zL);
  sumTwo xHSum(sx, xSizeHalf, xH);
  sumTwo yHSum(sy, ySizeHalf, yH);
```

```

sumTwo zHSum(sz, zSizeHalf, zH);

//PLANE INTERSECT TEST
wire signed [17:0] ax, adx, by, bdy, cz, cdz;
FXUmul axmul(sa, x, ax);
FXUmul adxmul(sa, dx, adx);
FXUmul bymul(sb, y, by);
FXUmul bdy mul(sb, dy, bdy);
FXUmul czmul(sc, z, cz);
FXUmul cdzmul(sc, dz, cdz);

reg signed [17:0] pNum, pDen;
always @(posedge pipelineClock)
begin
    pNum <= -(ax + by + cz + sd);
    pDen <= adx + bdy + cdz;
end

wire signed [17:0] tPlane, tPlaneP, pDenDelayed;
FXUdiv #(.LP_OUTPUT(1)) tpDiv(clock, reset, pNum, pDen, tPlaneP);
//TODO: make sure this delay is correct
pipeline #(.DELAY(4), .WIDTH(18)) pDenRegDelay(pipelineClock, reset, pDen,
pDenDelayed);
assign tPlane = (pDenDelayed == 0) ? -1 : tPlaneP;

//SPHERE INTERSECT TEST
wire signed [17:0] xDist, yDist, zDist;
diffTwo xDistDiff(x, sx, xDist);
diffTwo yDistDiff(y, sy, yDist);
diffTwo zDistDiff(z, sz, zDist);

wire signed [17:0] xDistD, yDistD, zDistD, xDist2, yDist2, zDist2, r2;
FXUmul xDistDMul(dx, xDist, xDistD);
FXUmul yDistDMul(dy, yDist, yDistD);
FXUmul zDistDMul(dz, zDist, zDistD);
FXUmul xDist2Mul(xDist, xDist, xDist2);
FXUmul yDist2Mul(yDist, yDist, yDist2);
FXUmul zDist2Mul(zDist, zDist, zDist2);
FXUmul r2Mul(sr, sr, r2);

wire signed [17:0] distD, dist2;
sumThree distDSum(xDistD, yDistD, zDistD, distD);
sumThree dist2Sum(xDist2, yDist2, zDist2, dist2);

wire signed [17:0] qb, qc;
assign qb = distD <<< 1;

```

```

diffTwo qcDiff(dist2, r2, qc);

wire signed [17:0] qb2, qc4;
FXUmul qb2Mul(qb, qb, qb2);
FXUmul qc4Mul(qc, 18'sb110_000_000_000_000, qc4);

wire signed [17:0] discPrereg;
sumTwo discCalc(qb2, qc4, discPrereg);

reg signed [17:0] discriminant, negQb;
always @(posedge pipelineClock)
begin
    discriminant <= discPrereg;
    negQb <= -qb;
end

wire signed [17:0] sqrt;
FXUsqrt sqrtCalc(clock, reset, discriminant, sqrt);

wire signed [17:0] t0, t1, tSphereP, tSphere;
diffTwo t0Calc(negQb, sqrt, t0);
sumTwo t1Calc(negQb, sqrt, t1);
assign tSphereP = (sqrt < 0) ? -1 : ((t0 > 0) ? (t0 >>> (5 + 1)) : (t1 >>> (5 + 1)));
//TODO: make sure this delay is correct
pipeline #(.DELAY(3), .WIDTH(18)) tSphereDelay(pipelineClock, reset, tSphereP,
tSphere); //equalize depth with other shape pipelines

//BOX INTERSECT TEST
wire signed [17:0] xLnumPrereg, xHnumPrereg, yLnumPrereg, yHnumPrereg,
zLnumPrereg, zHnumPrereg;
diffTwo xLnumCalc(xL, x, xLnumPrereg);
diffTwo xHnumCalc(xH, x, xHnumPrereg);
diffTwo yLnumCalc(yL, y, yLnumPrereg);
diffTwo yHnumCalc(yH, y, yHnumPrereg);
diffTwo zLnumCalc(zL, z, zLnumPrereg);
diffTwo zHnumCalc(zH, z, zHnumPrereg);

reg signed [17:0] xLnum, xHnum, yLnum, yHnum, zLnum, zHnum, dxReg, dyReg,
dzReg;
reg parallelMiss;
always @(posedge pipelineClock)
begin
    xLnum <= xLnumPrereg;
    xHnum <= xHnumPrereg;
    yLnum <= yLnumPrereg;
    yHnum <= yHnumPrereg;

```

```

zLnum <= zLnumPrereg;
zHnum <= zHnumPrereg;
dxReg <= dx;
dyReg <= dy;
dzReg <= dz;
parallelMiss <= (dx == 0 && (x < xL || x > xH)) || (dy == 0 && (y < yL || y > yH)) ||
(dz == 0 && (z < zL || z > zH));
end

```

```

wire signed [17:0] tXL, tXH, tYL, tYH, tZL, tZH;
wire parallelMissDelayed, xInvalid, yInvalid, zInvalid;
FXUdiv #(.LP_OUTPUT(1)) tXLdiv(clock, reset, xLnumPrereg, dxReg, tXL);
FXUdiv #(.LP_OUTPUT(1)) tXHdiv(clock, reset, xHnumPrereg, dxReg, tXH);
FXUdiv #(.LP_OUTPUT(1)) tYLdiv(clock, reset, yLnumPrereg, dyReg, tYL);
FXUdiv #(.LP_OUTPUT(1)) tYHdiv(clock, reset, yHnumPrereg, dyReg, tYH);
FXUdiv #(.LP_OUTPUT(1)) tZLdiv(clock, reset, zLnumPrereg, dzReg, tZL);
FXUdiv #(.LP_OUTPUT(1)) tZHdiv(clock, reset, zHnumPrereg, dzReg, tZH);
//TODO: make sure this delay is correct
pipeline #(.DELAY(4), .WIDTH(1)) pmDelay(pipelineClock, reset, parallelMiss,
parallelMissDelayed);
pipeline #(.DELAY(4), .WIDTH(1)) dxRegDelay(pipelineClock, reset, (dxReg == 0),
xInvalid);
pipeline #(.DELAY(4), .WIDTH(1)) dyRegDelay(pipelineClock, reset, (dyReg == 0),
yInvalid);
pipeline #(.DELAY(4), .WIDTH(1)) dzRegDelay(pipelineClock, reset, (dzReg == 0),
zInvalid);

```

```

wire xSwap, ySwap, zSwap;
wire signed [17:0] tXmin, tXmax, tYmin, tYmax, tZmin, tZmax;
assign xSwap = tXL > tXH;
assign ySwap = tYL > tYH;
assign zSwap = tZL > tZH;
assign tXmin = xSwap ? tXH : tXL;
assign tYmin = ySwap ? tYH : tYL;
assign tZmin = zSwap ? tZH : tZL;
assign tXmax = xSwap ? tXL : tXH;
assign tYmax = ySwap ? tYL : tYH;
assign tZmax = zSwap ? tZL : tZH;

```

```

wire signed [17:0] tNearP2, tNearP, tNear, tFarP2, tFarP, tFar;
assign tNearP2 = (xInvalid) ? 18'b100_000_000_000_000 : tXmin;
assign tNearP = (tNearP2 > tYmin || yInvalid) ? tNearP2 : tYmin;
assign tNear = (tNearP > tZmin || zInvalid) ? tNearP : tZmin;
assign tFarP2 = (xInvalid) ? 18'b011_111_111_111_111 : tXmax;
assign tFarP = (tFarP2 < tYmax || yInvalid) ? tFarP2 : tYmax;
assign tFar = (tFarP < tZmax || zInvalid) ? tFarP : tZmax;

```

```
wire signed [17:0] tBox;
assign tBox = ((tNear > tFar) || parallelMissDelayed) ? -1 : ((tNear > 0) ? tNear : tFar);

always @(posedge pipelineClock)
begin
  case(type)
    PLANE: t <= tPlane;
    SPHERE: t <= tSphere;
    BOX: t <= tBox;
    default: t <= -1;
  endcase
end
endmodule
```

## **Appendix E: Illuminator Module**

```
`timescale 1ns / 1ps
`default_nettype none

module illuminator
  (input wire clock, pipelineClock, reset, start, input wire [44:0] camera, input wire [53:0]
  ray, input wire [64:0] shape, input wire [35:0] light, input wire signed [17:0] t, output reg
  signed [17:0] redLight, greenLight, blueLight);
  localparam PLANE = 2'b00;
  localparam SPHERE = 2'b01;
  localparam BOX = 2'b10;

  localparam MAX_DIFF = 18'sb00_0000_0001_0000_0000;
  localparam NEG_ONE = 18'sb111_100_000_000_000;
  localparam POS_ONE = 18'sb000_100_000_000_000;

  //get camera/ray values
  wire signed [17:0] x, y, z, dx, dy, dz;
  assign x = {{2{camera[44]}}, camera[44:36], 7'b000_0000};
  assign y = {{2{camera[35]}}, camera[35:27], 7'b000_0000};
  assign z = {{2{camera[26]}}, camera[26:18], 7'b000_0000};
  assign dx = ray[53:36];
  assign dy = ray[35:18];
  assign dz = ray[17:0];

  //get shape values
  wire [1:0] type;
  wire signed [17:0] sx, sy, sz, sa, sb, sc;
  wire signed [17:0] sd, sr, xSizeHalf, ySizeHalf, zSizeHalf;

  assign type = shape[64:63];

  assign sx = {{2{shape[62]}}, shape[62:54], 7'b000_0000};
  assign sa = {{2{shape[62]}}, shape[62:54], 7'b000_0000};
  assign sy = {{2{shape[53]}}, shape[53:45], 7'b000_0000};
  assign sb = {{2{shape[53]}}, shape[53:45], 7'b000_0000};
  assign sz = {{2{shape[44]}}, shape[44:36], 7'b000_0000};
  assign sc = {{2{shape[44]}}, shape[44:36], 7'b000_0000};
  assign sd = {{2{shape[35]}}, shape[35:27], 7'b000_0000};
  assign sr = {{2{shape[35]}}, shape[35:27], 7'b000_0000};
  assign xSizeHalf = {{2{shape[35]}}, shape[35:27], 7'b000_0000} >>> 1;
  assign ySizeHalf = {{2{shape[26]}}, shape[26:18], 7'b000_0000} >>> 1;
  assign zSizeHalf = {{2{shape[17]}}, shape[17:9], 7'b000_0000} >>> 1;

  wire signed [17:0] xL, xH, yL, yH, zL, zH;
```

```

diffTwo xLDiff(sx, xSizeHalf, xL);
diffTwo yLDiff(sy, ySizeHalf, yL);
diffTwo zLDiff(sz, zSizeHalf, zL);
sumTwo xHSum(sx, xSizeHalf, xH);
sumTwo yHSum(sy, ySizeHalf, yH);
sumTwo zHSum(sz, zSizeHalf, zH);

//get light values
wire signed [17:0] lx, ly, lz;
wire signed [17:0] lred, lgreen, lblue;

assign lx = {{2{light[35]}}, light[35:27], 7'b000_0000};
assign ly = {{2{light[26]}}, light[26:18], 7'b000_0000};
assign lz = {{2{light[17]}}, light[17:0], 7'b000_0000};

assign lred = {4'b0000, light[8:6], 11'b000_0000_0000};
assign lgreen = {4'b0000, light[5:3], 11'b000_0000_0000};
assign lblue = {4'b0000, light[2:0], 11'b000_0000_0000};

//get intersect point
wire signed [17:0] xDist, yDist, zDist;
FXUmul #(.LP_INPUT(1)) xDistMul(dx, t, xDist);
FXUmul #(.LP_INPUT(1)) yDistMul(dy, t, yDist);
FXUmul #(.LP_INPUT(1)) zDistMul(dz, t, zDist);

wire signed [17:0] xHit, yHit, zHit;
sumTwo xHitCalc(x, xDist, xHit);
sumTwo yHitCalc(y, yDist, yHit);
sumTwo zHitCalc(z, zDist, zHit);

//PLANE NORMAL
wire signed [17:0] sdxuPlane, sdyuPlane, sdzuPlane;
assign sdxuPlane = sa;
assign sdyuPlane = sb;
assign sdzuPlane = sc;

//SPHERE NORMAL
wire signed [17:0] sdxuSphere, sdyuSphere, sdzuSphere;
diffTwo sdxuSphereCalc(xHit, sx, sdxuSphere);
diffTwo sdyuSphereCalc(yHit, sy, sdyuSphere);
diffTwo sdzuSphereCalc(zHit, sz, sdzuSphere);

//BOX NORMAL
wire signed [17:0] xLdiff, xHdiff, yLdiff, yHdiff, zLdiff, zHdiff;
diffTwo xLdiffCalc(xHit, xL, xLdiff);
diffTwo xHdiffCalc(xHit, xH, xHdiff);

```



```

diffTwo yLdiffCalc(yHit, yL, yLdiff);
diffTwo yHdiffCalc(yHit, yH, yHdiff);
diffTwo zLdiffCalc(zHit, zL, zLdiff);
diffTwo zHdiffCalc(zHit, zH, zHdiff);

wire signed [17:0] xLdiffAbs, xHdiffAbs, yLdiffAbs, yHdiffAbs, zLdiffAbs, zHdiffAbs;
abs xLAbs(xLdiff, xLdiffAbs);
abs xHAbs(xHdiff, xHdiffAbs);
abs yLAbs(yLdiff, yLdiffAbs);
abs yHAbs(yHdiff, yHdiffAbs);
abs zLAbs(zLdiff, zLdiffAbs);
abs zHAbs(zHdiff, zHdiffAbs);

//tests for differences from box bounds smaller than 0.5
wire xLdiffSmall, xHdiffSmall, yLdiffSmall, yHdiffSmall, zLdiffSmall, zHdiffSmall;
assign xLdiffSmall = xLdiffAbs < MAX_DIFF;
assign xHdiffSmall = xHdiffAbs < MAX_DIFF;
assign yLdiffSmall = yLdiffAbs < MAX_DIFF;
assign yHdiffSmall = yHdiffAbs < MAX_DIFF;
assign zLdiffSmall = zLdiffAbs < MAX_DIFF;
assign zHdiffSmall = zHdiffAbs < MAX_DIFF;

wire signed [17:0] sdxuBox, sdyuBox, sdzuBox;
assign sdxuBox = xLdiffSmall ? NEG_ONE : (xHdiffSmall ? POS_ONE : 0);
assign sdyuBox = yLdiffSmall ? NEG_ONE : (yHdiffSmall ? POS_ONE : 0);
assign sdzuBox = zLdiffSmall ? NEG_ONE : (zHdiffSmall ? POS_ONE : 0);

//LIGHT NORMAL
wire signed [17:0] ldxuPrereg, ldyuPrereg, ldzuPrereg;
diffTwo ldxuCalc(xHit, lx, ldxuPrereg);
diffTwo ldyuCalc(yHit, ly, ldyuPrereg);
diffTwo ldzuCalc(zHit, lz, ldzuPrereg);

//get non-normalized normal of light and not-necessarily-normalized normal of
intersected shape
reg signed [17:0] sdxu, sdyu, sdzu;
reg signed [17:0] ldxu, ldyu, ldzu;
always @(posedge pipelineClock)
begin
  case (type)
    PLANE:
      begin
        sdxu <= sdxuPlane;
        sdyu <= sdyuPlane;
        sdzu <= sdzuPlane;
      end
  end
end

```

```

SPHERE:
begin
  sdxu <= sdxuSphere;
  sdyu <= sdyuSphere;
  sdzu <= sdzuSphere;
end
BOX:
begin
  sdxu <= sdxuBox;
  sdyu <= sdyuBox;
  sdzu <= sdzuBox;
end
default:
begin
  sdxu <= 0;
  sdyu <= 0;
  sdzu <= 0;
end
endcase
ldxu <= ldxuPrereg;
ldyu <= ldyuPrereg;
ldzu <= ldzuPrereg;
end

//get normalized dot product of normal vectors
// if I have to say "normalize* normal" one more time, I will be a broken man
wire signed [17:0] sdxu2, sdyu2, sdzu2, ldxu2, ldyu2, ldzu2, sldxu, sldyu, sldzu;
FXUmul sdxu2mul(sdxu, sdxu, sdxu2);
FXUmul sdyu2mul(sdyu, sdyu, sdyu2);
FXUmul sdzu2mul(sdzu, sdzu, sdzu2);
FXUmul ldxu2mul(ldxu, ldxu, ldxu2);
FXUmul ldyu2mul(ldyu, ldyu, ldyu2);
FXUmul ldzu2mul(ldzu, ldzu, ldzu2);
FXUmul sldxumul(sdxu, ldxu, sldxu);
FXUmul sldyumul(sdyu, ldyu, sldyu);
FXUmul sldzumul(sdzu, ldzu, sldzu);

wire signed [17:0] sSumPrereg, lSumPrereg, slSumPrereg;
sumThree sSumPCalc(sdxu2, sdyu2, sdzu2, sSumPrereg);
sumThree lSumPCalc(ldxu2, ldyu2, ldzu2, lSumPrereg);
sumThree slSumPCalc(sldxu, sldyu, sldzu, slSumPrereg);

reg signed [17:0] sSum, lSum, slSum;
always @(posedge pipelineClock)
begin
  sSum <= sSumPrereg;

```

```

lSum <= lSumPrereg;
slSum <= slSumPrereg;
end

wire signed [17:0] sLength, lLength;
FXUsqrt sSqrt(clock, reset, sSum, sLength);
FXUsqrt lSqrt(clock, reset, lSum, lLength);

wire signed [17:0] tLength;
FXUmul tLMul(sLength, lLength, tLength);

wire signed [17:0] slSumDelayed;
//TODO: make sure this delay is correct
pipeline #(.DELAY(2), .WIDTH(18)) slSumDelay(pipelineClock, reset, slSum,
slSumDelayed);

wire signed [17:0] nDotL;
FXUdiv nDotLDiv(clock, reset, slSumDelayed, tLength, nDotL);

wire signed [17:0] lightFactor;
assign lightFactor = (nDotL < 0) ? 0 : nDotL;

//calculate amount of light shed on the pixel from this source
wire signed [17:0] redLightPrereg, greenLightPrereg, blueLightPrereg;
FXUmul redLightMul(lred, lightFactor, redLightPrereg);
FXUmul greenLightMul(lgreen, lightFactor, greenLightPrereg);
FXUmul blueLightMul(lblue, lightFactor, blueLightPrereg);

always @(posedge pipelineClock)
begin
redLight <= redLightPrereg;
greenLight <= greenLightPrereg;
blueLight <= blueLightPrereg;
end
endmodule

```

## **Appendix F: Game Controller**

```
module game2(clk, reset, init, move, up,down,left,right, space, rotate, speedin,  
camera_left, camera_right, camera_up, camera_down, shapes, lights, cameras,  
sequencer_done, counter, scoreout);
```

```
    input clk;  
    input reset,init;  
    input move;  
    input up, down, left, right;  
    input space, rotate;  
    input [1:0] speedin; //from majorfsm output  
    input camera_left, camera_right, camera_up, camera_down;  
    //output reg [64:0] mem_in;  
    //output reg [3:0] address;  
    //output reg we;  
    input sequencer_done;  
    output reg [3:0] counter; //address counter for writing to bram  
    output reg [519:0] shapes; //stream the 8 shapes in the scene  
    output reg [35:0] lights;  
    output reg [44:0] cameras;  
    output [3:0] scoreout;
```

```
parameter speed = 40;
```

```
wire collide;
```

```
wire objectgen_ready;  
wire [64:0] new_shape;  
reg objectgen_start;  
objectgen gen(clk, reset, objectgen_start, new_shape, objectgen_ready);
```

```
//Size of Ship (e.g. flat rectangular box) //
```

```
wire signed [8:0] shiphalf_x = 100;  
wire signed [8:0] shiphalf_y= 100;  
wire signed [8:0] shiphalf_z = 40;  
wire signed [8:0] ship_initx = 200; //initial position of ship  
wire signed [8:0] ship_inity = 0;  
wire signed [8:0] ship_initz = 0;  
wire signed [8:0] ship_color = 9'b100100000; //
```

```
//Equation of Plane
```

```
wire signed [8:0] plane_a = 0;
```

```

wire signed [8:0] plane_b = 0;
wire signed [8:0] plane_c = 1;
wire signed [8:0] plane_d = -100;
wire signed [8:0] plane_color = 9'b100000000;

//Camera
wire signed [8:0] camera_x = -256;
wire signed [8:0] camera_y = 0;
wire signed [8:0] camera_z = 0;
wire signed [8:0] hAngle_init = 0;
wire signed [8:0] vAngle_init = 32;

//Light
wire signed [8:0] light_x = 0;
wire signed [8:0] light_y = 100;
wire signed [8:0] light_z = 200;
wire signed [8:0] light_color = 9'b100100000; //green light

// x+ game_speed per cycle for every obstacle
// need to test that there are at least 2 obstacles spaced apart along y axis at least the width
of the ship,
//OR along z axis the height of the ship
//constrain min and max size of obstacles
//New obstacles appear every "t" seconds, so they are evenly spaced along x-axis.

reg signed [8:0] shipx, shipy, shipz;
reg signed [8:0] hAngle, vAngle;
reg signed [8:0] o1x, o2x, o3x, o4x, o5x, o6x; //6 obstacles
reg [64:0] o1, o2, o3, o4, o5, o6;
reg game_end;
reg [64:0] score;
assign scoreout = score[3:0];
parameter move_speed = 10;
wire yedge, zedge;

wire collide;
assign yedge = ((shipy >=255) | (shipy <= -255)) ? 1 : 0;
assign zedge = ((shipz >= 255) | (shipz <= -255)) ? 1 : 0;

/* Instantiate collision detector -> not completely working
wire [5:0] speed1, speed2;
wire [64:0] shape1, shape2;
collision_detector cdetect(clk, reset, shape1,shape2, speed1, speed2, collide);
*/

always @(posedge clk) begin

```

```

if (reset|init) begin
//Initialize positions, Store initial ship, camera info into registers
  shipx <= ship_initx;
  shipy <= ship_inity;
  shipz <= ship_initz;
  hAngle <= hAngle_init;
  vAngle <= vAngle_init;
  o1x <= 180;
  o2x <= 150;
  o3x <= 100;
  o4x <= 50;
  o5x <= -50;
  o6x <= -100;
  score <= 0;
end

else if (move) begin

  ///////////////////////////////////////////////////
  if (camera_right)
    hAngle <= hAngle + 64; //hAngle: pan right pi/4 radians: THETA = 2^6 = 64;
  if (camera_left)
    hAngle <= hAngle - 64;
  if (camera_up)
    vAngle <= vAngle + 64; //vAngle
  if (camera_down)
    vAngle <= vAngle - 64;

  ///////////////////////////////////////////////////
  if (sequencer_done) begin
    if (!yedge) begin
      //if (right && !left)
      if (right)
        shipy <= shipy + move_speed; //y-axis
      else if (left) shipy <= shipy - move_speed;

      end
    if (!zedge) begin
      // if (down && !up)
      if (down) shipz <= shipz - move_speed; //z-axis
      else if (up) shipz <= shipz + move_speed;

      end

    if (rotate) begin
      shipy <= shipz; //swap dimensions of y and z
    end
  end

```

```

    shipz <= shipy;
    end

    else begin
        shipy <= shipy;
        shipz <= shipz;
    end

end

//update obstacle locations.
if (sequencer_done) begin
    o1x <= o1x + speed;
    o2x <= o2x + speed;
    o3x <= o3x + speed;
    o4x <= o4x + speed;
    o5x <= o5x + speed;
    o6x <= o6x + speed;
end

if (o1x >= 200) begin
    objectgen_start <= 1;

    o1 <= {o2[64:63], o2x, o2[53:0]};
    o2 <= {o3[64:63], o3x, o3[53:0]};
    o3 <= {o4[64:63], o4x, o4[53:0]};
    o4 <= {o5[64:63], o5x, o5[53:0]};
    o5 <= {o6[64:63], o6x, o6[53:0]};
    o6 <= new_shape;

    //Update x!!
    {o1x,o2x,o3x,o4x,o5x,o6x} <= {o2x,o3x,o4x,o5x,o6x,new_shape[62:54]};
    score<=score+1;
    end
else objectgen_start <= 0;

if (collide) game_end =1;
else game_end = 0;

end
end

always @(posedge clk) begin
//Write to memory if finished reading from buffer
if (reset) counter <= 0;

```

```

else if (sequencer_done) begin
lights[35:0] <= {light_x, light_y, light_z, light_color};
cameras[44:0] <= {camera_x, camera_y, camera_z, hAngle, vAngle};
shapes[519:455] <= {2'b10, shipx,shipy,shipz,shiphalf_x, shiphalf_y, shiphalf_z,
ship_color};
shapes[454:390] <= {2'b00,plane_a, plane_b,plane_c,plane_d, 18'b0, plane_color};
    shapes[389:325] <= o1;
    shapes[324:260] <= o2;
    shapes[259:195] <= o3;
    shapes[194:130] <= o4;
    shapes [129:65] <= o5;
    shapes [64:0] <= o6;
    end
end

endmodule

```



## Appendix G: Game FSM

```
module majorfsm(clk, reset, press_start, collide, expired, level_sel,
start_timer, background, ship, obstacles, init, move, lives, speed);
  input clk, reset, press_start, collide, expired;
  input [1:0] level_sel;
  output reg [1:0] speed; //how fast the objects move, only set on title screen
  output [1:0] background; //00=title, 01=gameplay, 10=crash, 11=gameover
  output reg start_timer;
  output ship, obstacles, init, move; //to display or not to display
  output reg [1:0] lives; //how many lives you have left: do not output
  reg [2:0] state, next_state;
```

```
//5 States
```

```
parameter S_title = 0;
parameter S_static = 1;
parameter S_move = 2;
parameter S_noship = 3;
parameter S_lose = 4;
```

```
always @* begin
  if (reset) begin
    next_state = S_title;
    start_timer = 0;
    lives = 4;
  end
  else
    case (state)
      ////////////////
      S_title: begin
        start_timer = 0;
        lives = 3;
        speed = level_sel;
        if (press_start) begin
          next_state = S_static;
        end
        start_timer = 1;
      end
      else
        next_state = S_title;
    end

  ////////////////
  S_static: begin
    start_timer = 0;
```

```

    if (expired & press_start)
        next_state = S_move;
    else
        next_state = S_static;
    end

    ////////////
S_move: begin
    if (collide) begin
        next_state = S_noship;
        start_timer = 1;
    end
    else begin
        next_state = S_move;
        start_timer = 0;
    end
end

    ////////////
S_noship: begin
    start_timer=0;
    if (expired)
        if (lives > 0) begin
            next_state = S_static;
            lives = lives -1;
        end
        else begin
            next_state = S_lose;
            lives = 0;
        end
    else
        next_state = S_noship;
    end

    ////////////
S_lose: begin
    start_timer = 0;
    if (press_start)
        next_state = S_title;
    else
        next_state = S_lose;
    end

    ////////////
default: next_state = S_title;
endcase

```

```

end

always @(posedge clk) state <= next_state;

//assign outputs which are dependent only on state
//Background: 0=title, 1=gameplay, 2=crash, 3=gameover
assign background = (state == S_lose)? 3:
    ((state == S_noship)? 2:
    ((state == S_title)? 0: 1));
assign ship = ((state == S_static)|(state == S_move))? 1 : 0; //show ship in static and
move states
assign obstacles = ((state == S_title)|(state == S_lose))? 0: 1; //show obstacles in states
except title and lose
assign move = (state == S_move)? 1: 0; //move forward in time in move state
assign init = (state == S_static)? 1: 0;

endmodule

```

## **Appendix H: Collision Detector**

```
module collision_detector(clk, reset, shape1,shape2, speed1, speed2, collide);
input clk, reset;
input [64:0] shape1, shape2;
input [5:0] speed1, speed2;
output collide;

wire collide1, collide2, collide3;

//extract shape information from game controller buffer
wire [1:0] shape1_id = shape1[64:63];
wire [8:0] x1 = shape1[62:54];
wire [8:0] y1 = shape1[53:45];
wire [8:0] z1 = shape1[44:36];
wire [8:0] r1 = shape1[35:27];
wire [26:0] sizexyz1 = shape1[35:9];

wire [1:0] shape2_id = shape2[64:63];
wire [8:0] x2 = shape2[62:54];
wire [8:0] y2 = shape2[53:45];
wire [8:0] z2 = shape2[44:36];
wire [8:0] r2 = shape2[35:27];
wire [26:0] sizexyz2 = shape2[35:9];

wire sphere;

//check shape ids and pick appropriate collide module
assign sphere = (shape2_id == 2'b01) ? 1: 0; //two spheres
assign collide = ((shape1_id == 2'b10) && (shape2_id==2'b00))? collide3: //plane_ship
collision
    ((shape1_id == 2'b10)? collide2: //ship to box or sphere obstacle
    (((shape1_id == 2'b01)&&(shape1_id == 2'b01))? collide1: 0 //sphere to
sphere
    ));
wire [8:0] sphere_radius;
assign sphere_radius = (sphere)? r2: 0;

box_box_sphere c1(clk, reset, x1, y1, z1, x2, y2, z2,
    sizexyz1, sizexyz2, speed1, speed2, sphere_radius, collide1);
sphere_sphere c2(clk, reset, x1, y1,z1, x2, y2, z2, r1, r2, speed1, speed2, collide2);
plane_ship c3 (clk, reset, z1, sizexyz1[18:9], z2, collide3);

endmodule
```

## **Appendix I: Shape Collision Detection Modules**

//.....Sphere to Sphere

```
module sphere_sphere (clk, reset, x1, y1, z1, x2, y2, z2, r1, r2, speed1, speed2, collide);
    input clk;
    input reset;
    input [8:0] x1, y1, z1, x2, y2, z2; //signed
    input [8:0] r1, r2;
    input [5:0] speed1, speed2; //[speedx, speedy, speedz], signed, 4 different speed settings
    = speed*2 pixels/cycle
    output reg collide;
```

```
    wire signed [8:0] xs1 = x1 + speed1[5:4];
    wire signed [8:0] xs2 = x2 + speed2[5:4];
    wire signed [8:0] ys1 = y1 + speed1[3:2];
    wire signed [8:0] ys2 = y2 + speed2[3:2];
    wire signed [8:0] zs1 = z1 + speed1[1:0];
    wire signed [8:0] zs2 = z2 + speed2[1:0];
```

```
    reg signed [19:0] distance_r, distance_x, distance_y, distance_z;
```

```
    always @ (posedge clk) begin
        if (reset) collide <= 0;
        else begin
```

```
            //answer shows up 2 cycles later, maybe need to change pipeline depending on tpd
            distance_r <= (r1+r2)*(r1+r2);
            distance_x <= (x2-x1)*(x2-x1);
            distance_y <= (y2-y1)*(y2-y1);
            distance_z <= (z2-z1)*(z2-z1);
```

```
            if ((distance_x <= distance_r) && (distance_y <= distance_r) && (distance_z <=
            distance_r)) begin
                collide <= 1;
            end
            else collide <= 0;
            end
        end
```

```
endmodule
```

//.....Box to Box, axis-aligned.

//.....Box to Sphere, sphere is object #2

```
module box_box_sphere (clk, reset, x1, y1, z1, x2, y2, z2,
    sizexyz1, sizexyz2, speed1, speed2, sphere_radius, collide);
```

```

input clk;
input reset;
input [8:0] x1, y1, z1, x2, y2, z2;
input [26:0] sizexyz1, sizexyz2; //each dimension is half the total length
input [5:0] speed1, speed2;
input [8:0] sphere_radius;
output collide;

reg collidex, collidey, collidez;

//position of center
wire signed [8:0] xs1 = x1 + speed1[5:4];
wire signed [8:0] xs2 = x2 + speed2[5:4];
wire signed [8:0] ys1 = y1 + speed1[3:2];
wire signed [8:0] ys2 = y2 + speed2[3:2];
wire signed [8:0] zs1 = z1 + speed1[1:0];
wire signed [8:0] zs2 = z2 + speed2[1:0];

//Divide by 2 to find distance from center
wire [8:0] halfx1= sizexyz1[26:18] + sphere_radius;
wire [8:0] halfy1= sizexyz1[17:9] + sphere_radius;
wire [8:0] halfz1= sizexyz1[8:0] + sphere_radius;
wire [8:0] halfx2; //= sizexyz2[26:18];
wire [8:0] halfy2; //= sizexyz2[17:9];
wire [8:0] halfz2; //= sizexyz2[8:0];

assign halfx2=(sphere_radius>0)? 0: sizexyz2[26:18];
assign halfy2=(sphere_radius>0)? 0: sizexyz2[17:9];
assign halfz2=(sphere_radius>0)? 0: sizexyz2[8:0];

//If box to sphere
//Extend x,y,z dimensions of box by 2*R. Then detect collision with centerpoint of sphere.
//If box to box
//center1-center2 <= half1 + half2

always @(posedge clk) begin
  if (reset) collidex <= 0;
  else begin
    if (xs1>xs2) begin
      if ((xs1-xs2) <= (halfx1 + halfx2)) collidex <= 1; //distance between centers <=
widths of both boxes
    else collidex <= 0;
    end
    if (xs1<=xs2) begin
      if ((xs2-xs1) <= (halfx1+halfx2)) collidex <= 1;

```

```

else collidex <= 0;
end

if (ys1>ys2) begin
  if ((ys1-ys2) <= (halfy1 + halfy2)) collidey <= 1;
  else collidey <= 0;
  end
if (ys1<=ys2) begin
  if ((ys2-ys1) <= (halfy1+halfy2)) collidey <= 1;
  else collidey <= 0;
  end

if (zs1>zs2) begin
  if ((zs1-zs2) <= (halfz1 + halfz2)) collidez <= 1;
  else collidez <= 0;
  end
if (zs1<=zs2) begin
  if ((zs2-zs1) <= (halfz1+halfz2)) collidez <= 1;
  else collidez <= 0;
  end
end //else !reset

end //always

assign collide = collidex && collidey && collidez;

endmodule

//.....Ground_plane to ship
module plane_ship(clk, reset, shipz, shiphalf_z, planez, collide );
input clk, reset;
input [8:0] shipz, shiphalf_z, planez;
output reg collide;

wire min, max;
//lowest point on ship is above planez
assign min = ((shipz - shiphalf_z)>=planez)? 1: 0;
//or highest point on ship below planez
assign max = ((shipz + shiphalf_z)<=planez)? 1: 0;

always @ (posedge clk) begin
if (reset) collide <= 0;
else if (min ^ max) collide <= 1;
end
endmodule

```

## **Appendix J: Frame Buffer**

```
module framebuffer(clk, reset, done, flip, pixelcolor, hcount, vcount, vgacolor,  
ram0_data, ram0_address, ram0_we_b, ram1_data, ram1_address, ram1_we_b);
```

```
input clk;  
input reset ;  
input done;  
input [17:0] pixelcolor;  
input [10:0] hcount;  
input [9:0] vcount;  
output reg [17:0] vgacolor;  
inout [35:0] ram0_data, ram1_data;  
output [18:0] ram0_address, ram1_address;  
output reg ram0_we_b, ram1_we_b;
```

```
output reg flip; //flip=1 --> write to ram1  
            //flip=0 --> write to ram0  
            reg flip_delay, flip_delay1;  
reg write_enable;  
reg [1:0] we_delay;
```

```
always@ (posedge clk) begin  
ram1_we_b<= (~flip);  
ram0_we_b<= (flip);  
flip_delay1<= flip;  
flip_delay<= flip_delay1;  
//delay switching of ram_data function by 2 in order to write the last pixel  
end
```

```
reg [18:0] writeaddress;  
wire [18:0] readaddress;  
reg [35:0] writedata;  
reg [35:0] writedata_old1, writedata_old2;  
reg [35:0] last_readdata, data_latched;  
wire [35:0] ram_readdata;  
reg [9:0] pixelx = 0; //up to 1023  
reg [9:0] pixely = 0; //up to 767
```

```
always @(posedge clk) begin  
write_enable <= (done && (pixelx[0]==1)); //for testing purposes on Modelsim  
we_delay<= (we_delay==1)? 2: 0;  
end
```

```
always @(posedge pixelx[0]) begin
```



```

    we_delay <= 1;
end

always @ (posedge clk) begin
    if(reset) begin
        pixelx<=0;
        pixely<=0;
        flip <= 1;
    end

    else begin
        if (done) begin

            if (pixelx >= 10'b111111111) begin //1023
                pixelx<=0;
                if (pixely >= 10'b101111111) begin //767
                    pixely<=0;
                end
                else pixely <= pixely+1;
            end
            else if (pixelx < 10'b111111111) begin
                pixelx <= pixelx+1;

                if (pixelx==0 && pixely ==0) flip <= (~flip);
            end

            if (pixelx[0]==0) begin //if first pixel in location is received
                writedata [17:0] <= {pixelx[6],5'b0,pixelx[5],5'b0,pixelx[4],5'b0};
            end
            else begin //if second pixel in location
                writedata[35:18] <= writedata[17:0];
                writedata [17:0] <= {pixelx[6],5'b0,pixelx[5],5'b0,pixelx[4],5'b0};
            end
        end
        writedata_old1 <= writedata;
        writedata_old2 <= writedata_old1;
        writeaddress <= {pixely[9:0], pixelx[9:1]}; //pixely[9:0] + pixelx[9:1] will give one
        address every 2 pixels

        //read data
        last_readdata<= (hcount[0]==0) ? ram_readdata: last_readdata;
        data_latched <= (hcount[0]==1) ? data_latched: last_readdata;
    end
end

```

```
always @* begin
vgacolor <= (hcount[0]==0)? data_latched[35:18] : data_latched[17:0];
end
```

```
assign ram0_address = flip? readaddress: writeaddress;
assign ram1_address = flip? writeaddress: readaddress;
assign ram0_data = (!flip_delay) ? writedata_old2 : {36{1'bZ}};
assign ram1_data = flip_delay? writedata_old2 : {36{1'bZ}};
assign ram_readdata = flip_delay? ram0_data: ram1_data;
assign readaddress = {vcount[9:0], hcount[9:1]};
```

```
endmodule
```

## **Appendix K: Pseudo-Random Object Generator**

```
//generate next object of random type, coordinates, sizes, and color
module objectgen(clk, reset, start, shape, ready);
```

```
    input clk, reset, start;
    output reg [64:0] shape;
    output reg ready;
```

```
    wire signed [8:0] random1, random2, random3, random4;
    randomgen #(.ROOT(4)) r1(clk, reset, random1);
    randomgen #(.ROOT(13)) r2(clk, reset, random2);
    randomgen #(.ROOT(28)) r3(clk, reset, random3);
    randomgen #(.ROOT(23)) r4(clk, reset, random4);
```

```
    //Determine type of object
    wire [1:0] type;
    assign type = random1[4:3];
```

```
    //determine color of object
    wire [8:0] color;
    assign color = {random2[2:0], random3[2:0], random4[2:0]};
```

```
    always @ (posedge clk) begin
        if (reset) begin
            shape [64:63] <= 2'b11;
            shape [62:0] <= 63'b0;
            ready <= 0;
        end
        else begin
            if (start) begin
                shape [64:63] <= type;
                shape [8:0] <= color;
                case (type)
```

```
                    2'b00: begin //Plane, parallel to xy-plane
                        shape[62:54] <= -200; //a
                        shape[53:45] <= 0; //b
                        shape[44:36] <= 1; //c
                        shape[35:27] <= random4; //d
                        shape[26:9] <= 0; //unused
                    end
```

```
                    2'b01: begin //Sphere
                        shape[62:54] <= -200; //x: new objects appear at x=-200
                        shape[53:45] <= random2; //y
```

```

    shape[44:36] <= random3; //z
    shape[35:27] <= {1'b0, random4[8:1]}; //radius
    shape[26:9] <= 0; //unused
end

2'b10: begin //Box
    shape [62:54] <= -200; //x  represents center
    shape [53:45] <= random2; //y
    shape [44:36] <= random3; //z
    shape [35:27] <= {2'b0, random4[6:0]}; //xsize: positive # between 0 and 128
(**actually half the size**)
    shape [26:18] <= {2'b0, random2[7:1]}; //ysize
    shape [17:9] <= {2'b0, random1[3:0],random1[7:5]}; //zsize
end

default: begin
    shape[62:54] <= -200;
    shape [53:0] <= 0;
end
endcase
end //if(start)
end //else

end //always

endmodule

```

## Appendix L: 2D Display

```
module display_shape (clk, reset, shape, hcount, vcount,
hsync, vsync, blank, phsync, pvsync, pblank, pixelout);
    input clk, reset;
    input [64:0] shape;
    input [10:0] hcount;
    input [9:0] vcount;
    input hsync, vsync, blank;

    output phsync, pvsync, pblank;
    output reg [8:0] pixelout;

    wire [8:0] pixel_rectangle, pixel_circle;
    wire [8:0] color = shape[8:0];
    // wire signed [8:0] WIDTH = shape[35:27]+ shape[61:57];
    // wire signed [8:0] RADIUS = WIDTH;
    // wire signed [8:0] HEIGHT = shape[26:18]+ shape[61:57];

    wire [9:0] WIDTH = shape[53:45]+ shape[53:45];
    wire [8:0] RADIUS = shape[35:27];
    wire [9:0] HEIGHT = shape[44:36]+shape[44:36];

    wire signed [8:0] worldz = shape[44:36];
    wire signed [8:0] worldy = shape[53:45];
    wire signed [8:0] d = shape[35:27];

    //Translate world coordinates to hcount/vcount coordinates:
    //
    wire [9:0] y = (worldy+256)*2;
    wire [9:0] z = worldz+384;

    rectangle rect1(y,z,hcount,vcount,WIDTH,HEIGHT,color,pixel_rectangle);
    circle circ1(y,z,hcount,vcount,RADIUS,color,pixel_circle);
    rectangle rect2(y,z,hcount,vcount,500,5,color,pixel_plane);

    always @(posedge clk) begin
        if (reset) begin
            pixelout <= 0;
        end
        else begin
            case (shape[64:63])
                2'b00: pixelout <= pixel_plane;
                2'b01: pixelout <= pixel_circle;
                2'b10: pixelout <= pixel_rectangle;
            endcase
        end
    end
endmodule
```

```

        default: pixelout <= {3'b000,3'b000,3'b000};
    endcase
end
end

assign phsync = hsync;
assign pvsync = vsync;
assign pblank = blank;

endmodule

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

////////////////////////////////////////////////////////////////
//
// rectangle: generate rectangle on screen
//
////////////////////////////////////////////////////////////////
module rectangle
    (input [9:0] x, input [9:0] y, //world coordinates
    input [10:0] hcount,
    input [9:0] vcount,
    input [9:0] WIDTH, HEIGHT,
    input [8:0] color,
    output reg [8:0] pixel);

    always @ (x or y or hcount or vcount) begin
        if (((hcount >= x && hcount < (x+WIDTH)) &&
            (vcount >= y && vcount < (y+HEIGHT)))
            pixel = color;
        else pixel = 0;
    end
endmodule

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

////////////////////////////////////////////////////////////////
//
// circle: generate circle on screen
//
////////////////////////////////////////////////////////////////
module circle
    (input [9:0] x, input [9:0] y, //world coordinates

```

```

input [10:0] hcount,
input [9:0] vcount,
input [8:0] RADIUS,
input [8:0] color,
output reg [8:0] pixel);

wire [20:0] xcal;
wire [18:0] ycal;
wire [20:0] radius2;

assign xcal = (hcount-(x+RADIUS))*(hcount-(x+RADIUS));
assign ycal = (vcount-(y+RADIUS))*(vcount-(y+RADIUS));
assign radius2 = RADIUS*RADIUS;

always @(x or y or hcount or vcount) begin

    //      if      (((hcount-(x+RADIUS))*(hcount-(x+RADIUS))      +      (vcount-
(y+RADIUS))*(vcount-(y+RADIUS)))< RADIUS**2) pixel = COLOR;

    if ((xcal + ycal) < radius2) pixel <= color;
    else pixel <= 0;
end

endmodule

```

## Appendix M: 2D Test Bench

```
/////////////////////////////////////////////////////////////////
//
// Pushbutton Debounce Module (video version)
//
/////////////////////////////////////////////////////////////////

module debounce (input reset, clock, noisy,
                 output reg clean);

    reg [19:0] count;
    reg new;

    always @(posedge clock)
        if (reset) begin new <= noisy; clean <= noisy; count <= 0; end
        else if (noisy != new) begin new <= noisy; count <= 0; end
        else if (count == 650000) clean <= new;
        else count <= count+1;

endmodule

/////////////////////////////////////////////////////////////////
////////LabKIT//////////////////////////////////////
/////////////////////////////////////////////////////////////////

module gametest (beep, audio_reset_b, ac97_sdata_out, ac97_sdata_in, ac97_synch,
                ac97_bit_clock,

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

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

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

                clock_feedback_out, clock_feedback_in,

                rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

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



```

mouse_clock, mouse_data, keyboard_clock, keyboard_data,

clock_27mhz, clock1, clock2,

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

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

switch,

led,

user1, user2, user3, user4,

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

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

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

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

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

input clock_feedback_in;
output clock_feedback_out;

output rs232_txd, rs232_rts;
input rs232_rxd, rs232_cts;

inout [15:0] flash_data;

```

```

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

input mouse_clock, mouse_data, keyboard_clock, keyboard_data;

input clock_27mhz, clock1, clock2;

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

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

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

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

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

// Audio Input and Output
assign beep= 1'b0;
assign audio_reset_b = 1'b0;
assign ac97_synch = 1'b0;
assign ac97_sdata_out = 1'b0;
// ac97_sdata_in is an input

// SRAMs
assign ram0_data = 36'hZ;
assign ram0_address = 19'h0;
assign ram0_adv_ld = 1'b0;
assign ram0_clk = 1'b0;
assign ram0_cen_b = 1'b1;
assign ram0_ce_b = 1'b1;
assign ram0_oe_b = 1'b1;
assign ram0_we_b = 1'b1;

```

```

assign ram0_bwe_b = 4'hF;
assign ram1_data = 36'hZ;
assign ram1_address = 19'h0;
assign ram1_adv_ld = 1'b0;
assign ram1_clk = 1'b0;
assign ram1_cen_b = 1'b1;
assign ram1_ce_b = 1'b1;
assign ram1_oe_b = 1'b1;
assign ram1_we_b = 1'b1;
assign ram1_bwe_b = 4'hF;
assign clock_feedback_out = 1'b0;
// clock_feedback_in is an input

// RS-232 Interface
assign rs232_txd = 1'b1;
assign rs232_rts = 1'b1;
// rs232_rxd and rs232_cts are inputs

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

// PS/2 Ports
// mouse_clock, mouse_data, keyboard_clock, and keyboard_data are inputs

/*
// LED Displays
assign disp_blank = 1'b1;
assign disp_clock = 1'b0;
assign disp_rs = 1'b0;
assign disp_ce_b = 1'b1;
assign disp_reset_b = 1'b0;
assign disp_data_out = 1'b0; */
// disp_data_in is an input

// Buttons, Switches, and Individual LEDs
// assign led = 8'hFF;
// button0, button1, button2, button3, button_enter, button_right,
// button_left, button_down, button_up, and switches are inputs

```

```

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

// Logic Analyzer
// assign analyzer1_data = 16'h0;
// assign analyzer1_clock = 1'b1;
assign analyzer2_data = 16'h0;
assign analyzer2_clock = 1'b1;
// assign analyzer3_data = 16'h0;
//assign analyzer3_clock = 1'b1;
assign analyzer4_data = 16'h0;
assign analyzer4_clock = 1'b1;

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

// use FPGA's digital clock manager to produce a
// 65MHz clock (actually 64.8MHz)
wire clock_65mhz_unbuf,clock_65mhz;
DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_65mhz_unbuf));
// synthesis attribute CLKFX_DIVIDE of vclk1 is 10
// synthesis attribute CLKFX_MULTIPLY of vclk1 is 24
// synthesis attribute CLK_FEEDBACK of vclk1 is NONE
// synthesis attribute CLKIN_PERIOD of vclk1 is 37
BUFG vclk2(.O(clock_65mhz),.I(clock_65mhz_unbuf));

// power-on reset generation
wire power_on_reset; // remain high for first 16 clocks
SRL16 reset_sr (.D(1'b0), .CLK(clock_65mhz), .Q(power_on_reset),
    .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
defparam reset_sr.INIT = 16'hFFFF;

// ENTER button is user reset
wire reset,user_reset;
debounce
db1(.reset(power_on_reset),.clock(clock_65mhz),.noisy(~button_enter),.clean(user_reset)
);
assign reset = user_reset | power_on_reset;

// keyboard input
wire [7:0] ascii;
wire char_rdy;
ps2_ascii_input kbd(clock_65mhz, reset, keyboard_clock,

```

```

    keyboard_data, ascii, char_rdy);

//game controls:
wire up, down, left, right, space, rotate; //for moving ship in 2D, starting game, and
flipping 90deg.
wire [1:0] speed; //4 different speeds/levels
wire camera_left, camera_right, camera_up, camera_down; //for rotating camera
horizontally and vertically

/* Ps/2 keyboard
assign left = (ascii ==8'h41)? 1: 0; //a
assign right = (ascii ==8'h44)? 1 :0; //d
assign up = (ascii == 8'h57)? 1: 0; //w
assign down = (ascii == 8'h53)? 1:0; //s
assign space = (ascii == 8'h20)? 1:0; //space
assign rotate = (ascii == )? 1: 0; //rotate
*/
debounce db2(.reset(reset),.clock(clock_65mhz),.noisy(~button_up),.clean(down));
debounce db3(.reset(reset),.clock(clock_65mhz),.noisy(~button_down),.clean(up));
debounce db4(.reset(reset),.clock(clock_65mhz),.noisy(~button_left),.clean(left));
debounce db5(.reset(reset),.clock(clock_65mhz),.noisy(~button_right),.clean(right));
debounce db6(.reset(reset),.clock(clock_65mhz),.noisy(~button0),.clean(space));
debounce db7(.reset(reset),.clock(clock_65mhz),.noisy(~button1),.clean(rotate));
debounce
db8(.reset(reset),.clock(clock_65mhz),.noisy(~button2),.clean(camera_down));
debounce db9(.reset(reset),.clock(clock_65mhz),.noisy(~button3),.clean(camera_up));
assign speed = 0;

// generate basic X VGA video signals
wire [10:0] hcount;
wire [9:0] vcount;
wire hsync,vsync,blank;
xvga xvga1(.vclock(clock_65mhz),.hcount(hcount),.vcount(vcount),
.hsync(hsync),.vsync(vsync),.blank(blank));

reg start_timer=0;
wire timer_enable;
divider_general div1(clock_65mhz, reset, start_timer, timer_enable);

// feed X VGA signals to gamedisplay
wire [8:0] pixel;
wire phsync,pvsync,pblank;

```

```

//Collision tests
/*
  assign shape1[64:63]= 2'b01; //green sphere
  assign shape1[62:36] = {9'd100, 9'd300, 9'd300};
  assign shape1[35:27] = 9'd50;
  assign shape1[8:0] = 8'b100100000;
*/
  assign shape2[64:63]= 2'b01; //blue sphere
  assign shape2[62:36] = {9'd100, 9'd100, 9'd100};
  assign shape2[35:27] = 9'd100;
  assign shape2[8:0] = 8'b110100100;
*/
/*
  assign shape1[64:63]= 2'b10; //green box
  assign shape1[62:36] = {9'd500, 9'd500, 9'd500};
  assign shape1[35:27] = 9'd50;
  assign shape1[26:18] = 9'd50;
  assign shape1[17:9] = 9'd50;
  assign shape1[8:0] = 8'b100100000;
*/
/*
  assign shape2[64:63]= 2'b10; //blue box = ship
  assign shape2[62:36] = {9'd100, 9'd100, 9'd100};
  assign shape2[35:27] = 9'd50;
  assign shape2[26:18] = 9'd100;
  assign shape2[17:9] = 9'd50;
  assign shape2[8:0] = 8'b100100000;
*/
/*
  wire [8:0] x1 = shape1[62:54];
  wire [8:0] y1 = shape1[53:45];
  wire [8:0] z1 = shape1[44:36];
  wire [8:0] r1 = shape1[35:27];
  // wire [26:0] sizexyz1 = shape1[35:9];
  wire [8:0] sphere_radius = r1;

  reg [8:0] y2,z2;
  wire [8:0] x2 = shape2[62:54];
  //wire [8:0] y2 = shape2[53:45];
  //wire [8:0] z2 = shape2[44:36];
  wire [26:0] sizexyz2 = shape2[35:9];
*/
  always @ (posedge clock_65mhz) begin
  if (reset) begin

```

```

y2 <= shape2[53:45];
z2 <= shape2[44:36];
end
if ((hcount == 1023)&&(vcount ==767)) begin
  if (up) z2 <= z2 + 5;
  if (down) z2 <= z2 - 5;
  if (right) y2 <= y2 + 5;
  if (left) y2 <= y2 - 5;
end
end

display_shape displayobstacle(clk, reset, shape1, hcount, vcount,
hsync, vsync, blank, phsync, pvsync, pblank, pixelobstacle);

display_shape displayship(clk, reset, shape2, hcount, vcount,
hsync, vsync, blank, phsync, pvsync, pblank, pixelship);

// sphere_sphere ss1(clock_65mhz, reset, x1, y1,z1, x2, y2, z2, r1, r2, speed1, speed2,
collide);
// box_box_sphere bbs1 (clock_65mhz, reset, x2, y2, z2, x1, y1, z1,
//   sizexyz2, sizexyz1 , speed2, speed1, sphere_radius, collide);

*/

wire [3:0] counter;
reg sequencer_done;
wire [519:0] shapes;
wire [44:0] cameras;
wire [35:0] lights;
wire move;
wire init;
wire [3:0] scoreout;

game2 gtest(clock_65mhz, reset, init, move, up,down,left,right, space, rotate, speed,
camera_left, camera_right, camera_up, camera_down, shapes, lights, cameras,
sequencer_done, counter,scoreout);

always @ (posedge clock_65mhz) begin
  if (timer_enable) sequencer_done <= 1;
  else begin
    sequencer_done <= 0;

```

```

end
end

wire [8:0] pixel1, pixel2, pixel3, pixel4, pixel5, pixel6, pixel7, pixel8;
display_shape disp1(clock_65mhz, reset, shapes[519:455], hcount, vcount, hsync, vsync,
blank,
phsync, pvsync, pblank, pixel1);

display_shape disp2(clock_65mhz, reset, shapes[389:325], hcount, vcount, hsync, vsync,
blank,
phsync, pvsync, pblank, pixel2);

display_shape disp3(clock_65mhz, reset, shapes[324:260], hcount, vcount, hsync, vsync,
blank,
phsync, pvsync, pblank, pixel3);

display_shape disp4(clock_65mhz, reset, shapes[259:195], hcount, vcount, hsync, vsync,
blank,
phsync, pvsync, pblank, pixel4);

display_shape disp5(clock_65mhz, reset, shapes[194:130], hcount, vcount, hsync, vsync,
blank,
phsync, pvsync, pblank, pixel5);

display_shape disp6(clock_65mhz, reset, shapes[129:65], hcount, vcount, hsync, vsync,
blank,
phsync, pvsync, pblank, pixel6);

display_shape disp7(clock_65mhz, reset, shapes[64:0], hcount, vcount, hsync, vsync,
blank,
phsync, pvsync, pblank, pixel7);

display_shape disp8(clock_65mhz, reset, shapes[454:390], hcount, vcount, hsync, vsync,
blank,
phsync, pvsync, pblank, pixel8);

wire ship;
assign showship = switch[4];
assign pixel = (showship)? pixel1|pixel2|pixel3|pixel4|pixel5|pixel6|pixel7|pixel8: pixel1;

wire [1:0] background;
//assign background = switch[7:6];

//wire [1:0] backgroundfsm;

```



```

wire expired;
wire start_timerfsm;
wire level_sel;
assign level_sel = switch[3:2];
wire collidefsm;
assign collidefsm = switch[0];
wire [1:0] speedfsm;
//wire shipfsm;
wire obstacles;
wire [1:0] lives;
timer timer1(clock_65mhz, reset, start_timerfsm, expired);

majorfsm fsm1(clock_65mhz, reset, space, collidefsm, expired, level_sel,
start_timerfsm, background, ship, obstacles, init, move, lives, speedfsm);

//Generate the background images:
//Title: 00
wire [8*8-1:0] cstring00 = "START";
wire [2:0] bgpixel00;
char_string_display                                bg1(clock_65mhz,hcount,vcount,
bgpixel00,cstring00,11'd500,9'd384);
//Game: 01
//Crash: 10
wire [8*8-1:0] cstring10 = "CRASH";
wire [2:0] bgpixel10;
char_string_display                                bg2(clock_65mhz,hcount,vcount,
bgpixel10,cstring10,11'd500,9'd384);
//Lose: 11
wire [8*8-1:0] cstring11 = "GAMEOVER";
wire [2:0] bgpixel11;
char_string_display                                bg3(clock_65mhz,hcount,vcount,
bgpixel11,cstring11,11'd500,9'd384);

reg [8:0] rgb;
reg b,hs,vs;
always @(posedge clock_65mhz) begin
    hs <= phsync;
    vs <= pvsync;
    b <= pblank;
    rgb <= pixel;
    case(background)
        2'b00: rgb <= {bgpixel00[0],2'b00,bgpixel00[1],2'b00,bgpixel00[2],2'b00};
        2'b01: rgb <= pixel;
        2'b10: rgb <= {bgpixel10[0],2'b00,bgpixel10[1],2'b00,bgpixel10[2],2'b00};
        2'b11: rgb <= {bgpixel11[0],2'b00,bgpixel11[1],2'b00,bgpixel11[2],2'b00};
    endcase
end

```

```

    endcase
end
// VGA Output. In order to meet the setup and hold times of the
// AD7125, we send it ~clock_65mhz.
assign vga_out_red = {rgb[8:6],5'b0};
assign vga_out_green = {rgb[5:3],5'b0};
assign vga_out_blue = {rgb[2:0],5'b0};
assign vga_out_sync_b = 1'b1; // not used
assign vga_out_blank_b = ~b;
assign vga_out_pixel_clock = ~clock_65mhz;
assign vga_out_hsync = hs;
assign vga_out_vsync = vs;

assign led = {3'b0, up, down, left, right, char_rdy};

//Hex display for debugging
wire [63:0] dispdata;
display_16hex hexdisp1(reset, clock_65mhz, dispdata,
    disp_blank, disp_clock, disp_rs, disp_ce_b,
    disp_reset_b, disp_data_out);
assign dispdata[54:32]=0;
assign dispdata[58:55] = lives;
assign dispdata[63:59] = scoreout;
assign dispdata[31:28] = rotate;
assign dispdata[27:24] = right;
assign dispdata[23:20] = left;
assign dispdata[19:16] = obstacles;
assign dispdata[15:12] = init;
assign dispdata[11:8] = move;
assign dispdata[7:4] = ship;
assign dispdata[3:0]= background;
//assign dispdata[8:0] = o1x;
//assign dispdata[7:0] = last_ascii;
//assign dispdata[7:0] = shape[7:0];
//assign dispdata[11:8] = collide;
wire [8:0] shipx = shapes[517:509];
wire [8:0] shipy = shapes[508:500];
wire [8:0] o1x = shapes[387:379];
assign analyzer1_data[0] = sequencer_done;
assign analyzer1_data[9:1]=shipx; //shipx
assign analyzer1_data[15:10]=shipy[5:0]; //shipy[5:0]
assign analyzer1_clock = clock_65mhz;
assign analyzer3_data[8:0] = o1x; //o1x
assign analyzer3_data[15:12] = 0;
assign analyzer3_data[11:9] = shapes[8:6]; //shipy[8:6]
assign analyzer3_clock = clock_65mhz;

```

endmodule