Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.111 - Introductory Digital Systems Laboratory
Gary M. Matthias, Timothy M. Mwangi and Anthony J. Quivers
May 17, 2007

Design and Implementation of a 3-D Game of Pool on an FPGA Using the Major-Minor FSM Setup

Abstract:

In this project we use the major-minor finite state machine setup to implement a virtual form of the popular game, pool, using a field programmable gate array device. Specifically, we will obtain the position of a player around an empty pool table from the output of a NTSC video camera. Likewise, we will obtain the position of the tip of the pool stick and calculate the pivot angle based on colored reference points on both the pool table and the pool stick. Furthermore, the speed of the pool stick is calculated by digitizing the data from an accelerometer. On any particular shot, if the speed of the pool stick is greater than some threshold value, and the reference points of the pool table and the pool stick are within a specified distance, the shot will be extrapolated and the tip of the stick will strike the white ball. Any moving ball will strike balls in its path and the boundaries of the table according to the laws of motion in classical mechanics. The movement of each ball will be controlled by an identical minor FSM and a single major FSM will mediate collisions between the balls, table, edges and stick and apply the dissipative force of the pool table surface. A user who plays this game will be able to see the stick's position, relative to the pool table and the positions of the balls on the table on a screen in three-dimensions using ray tracing. Note however, that there are no physical balls and the player relies solely on the screen for information about the positions of different balls. We expect our solution to closely mimic a real game of pool. There are three main implementation and design challenges posed by this project. The first is using a camera to accurately determine the position and velocity of objects in its field of view. The second is creating a model of multiple collisions that is accurate, elegant and simple enough to be implemented using the major-minor FSM setup within the constraints of available resources. The third is to efficiently display the pool game in three dimensions with a relocatable point of observation, also within these resource constraints.

Table of Contents

Table of Figures

List of Tables

Introduction

This project is an implementation of the two-player game of pool on the Xilinx 6.111 FPGA Labkit, a field programmable gate array device. It also makes use of an NTSC video camera, a DE-ACCM3D three-dimensional accelerometer from Dimension Engineering, an AD7810Y analog-to-digital converter from Analog Devices. The entire project can be broken up into three primary components: the input module, the gaming and physics module, and the display module. There is also a pool table, shown in Figure 1, in the physical world through which the user can input information.

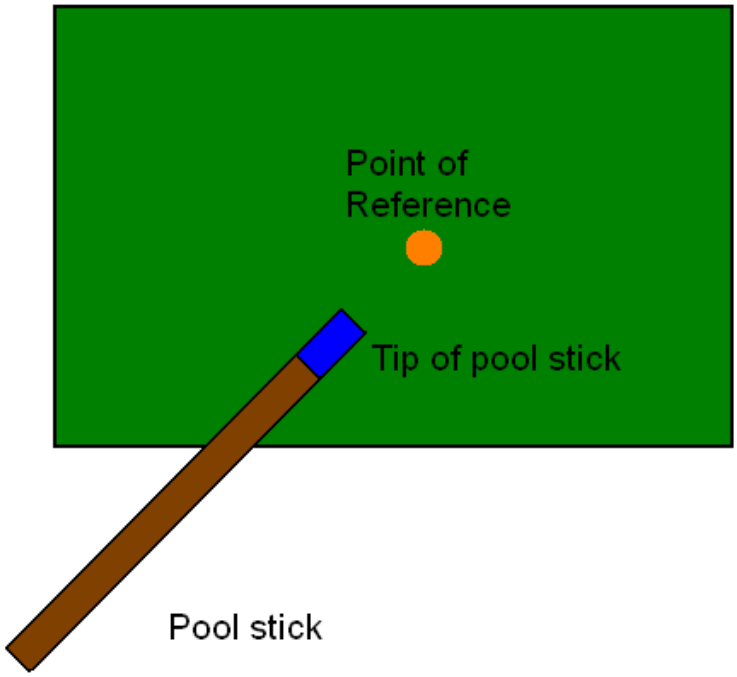

Figure 1: Pool Table as Seen from NTSC Video Camera

The video camera hangs directly above the pool table with a downward perspective. The pool table itself has a green color, whereas the center of the pool contains an orange, circular point-of-reference. The pool stick is brown and contains blue tip. The color-coded points of reference at the center of the pool table and at the tip of pool stick are used by the system to locate the relative locations of the two points by using chrominance and luminance thresholds and then calculating the center of mass. This color coding is necessary in order to simulate a pool ball strike. The relative points are used by the system to calculate the distance and the angle between the two points.

The input module is the interface by which the user can input data into the system. This module employs the video camera data and the digitized accelerometer data. Using the information supplied by these devices the input module constantly calculates the center of mass of the orange point-of-reference in the center of the pool table and the blue point-of-reference on the tip of the pool stick. When the two points are close enough to each other, the input module sends an enable signal to the game and physics module to simulate a pool ball strike. At this point, the inputs module calculates the speed of the strike based on the information received from the analog-to-digital converter, and parametrizes this speed into x- and y-components, based on the angle between the two reference points. This module is important for integrating analog input into a digital environment.

The game and physics module is the module that carries out the movements of the pool balls. This module includes take into account collisions between the balls and boundaries, using the fundamentals of momentum in classical mechanics. It also considers the dissipative force of the billiard cloth on a real pool table. However, for this implementation angular momentum will be ignored for the order of complexity that it would add to the design. This module also has all of the game logic, inherent with all of the standard rules of a game of pool. The game and physics module uses the concepts of a major-minor finite state machine to encourage modularity and as a result of having to carry out the same operations repeatedly and simultaneously.

The display module is the visual output sent to the user through the VGA card on the FPGA Labkit. The resolution of the display is 1024x768 with a refresh rate of 60 Hz. It uses a 65 Mhz pixel clock. The display module makes use of the 4MB of zero bus turnaround (ZBT) RAM on the FPGA Labkit. For each pixel of the display, it calculates the color value of the pixel using the concept of ray tracing. This means that for each pixel, based on a observer's point of reference, it traces a path from the observer to the first opaque pixel it runs into. This module requires optimization of timing in order to store, retrieve and calculate data efficiently.

Camera and Accelerometer Input Module

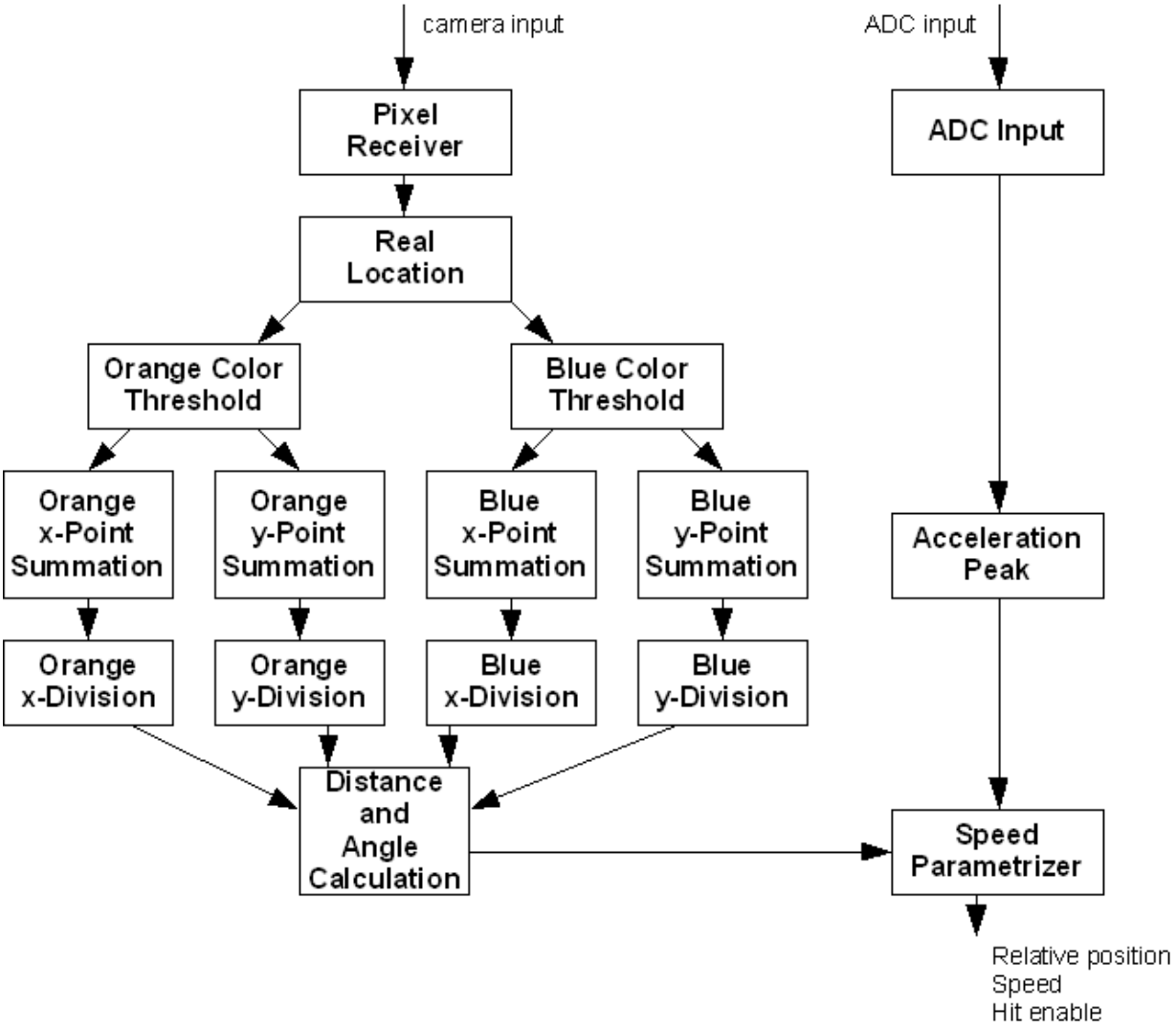

Figure 2: Abstracted Block Diagram of Camera and Accelerometer Input Module.

The inputs supermodule is used in this system to evaluate the data received from the three devices mentioned before: a video camera, a DE-ACCM3D accelerometer and a AD7810Y analog-to-digital-converter. As inputs, it receives the 27 MHz clock signal, a system reset signal, the eight-bit video data for luminance and chrominance, and the one-bit data from the analog-to-digital converter to which the accelerometer is attached. This supermodule outputs to the physics and game supermodule, the relative x- and y-position between the pool stick and the point of reference on the pool table, the current x- and y- speed of the pool stick, and an enable signal for a pool stick strike. It also sends an enable signal and a modified clock signal to the ADC.

1. NTSC Decoding

The FPGA labkit receives video data from the video camera in NTSC video format at a rate of 27 MHz, with one chrominance value and one luminance value, each of eight-bits, sent every two clock cycles. There are two types of chrominance values, blue chrominance and red chrominance, that refer specifically to content of blue and red of a particular pixel, whereas, there is one type of luminance value that refers to the brightness of a particular pixel without regards to its color. Thus, the types of video data are received at ratio of 4:2:2 (luminance, blue chrominance, red chrominance). This format has a total of 864 pixels and 525 lines, with an active resolution of 720 pixels by 485 lines. The video data is interlaced, meaning that all even lines from 0 to 484 are received together, followed by all odd lines from 1 to 483. For the purpose of keeping track of active states and interlacing, there is a reference code that appears at the beginning and end of every horizontal blanking state, 8'hFF followed by 8'h00. This reference code is then followed by 8'h00 and then followed by 8 bits: 1, f, v, h, v xor h, f xor h, f xor v, f xor v xor h. The value f is 0 when the video data comes from even lines of the interlaced data and 1 when the video data comes from the odd lines of the interlaced data. The value of v is 0 when there is no vertical blanking, and 1 when there is vertical blanking. The value of h is 0 when there is no horizontal blanking, and 1 when there is horizontal blanking.

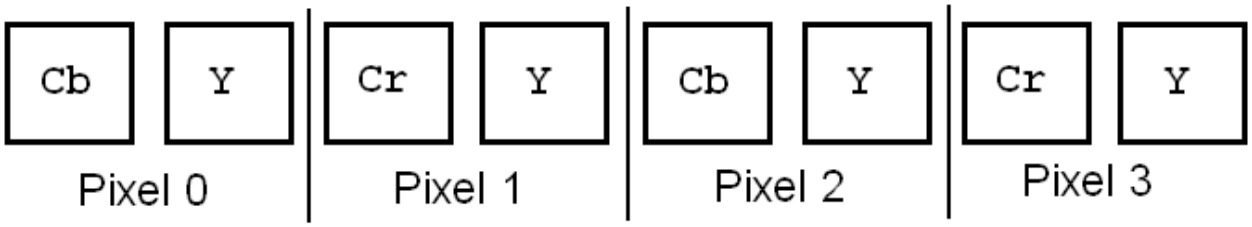

Figure 3: Data Sequence of Video Encoded in NTSC 4:2:2 Format.

2. Pixel Receiver Module (pixel_receiver.v)

Using this NTSC specification, the pixel receiver module decodes the NTSC data from the video camera so that the data can be evaluated. After the system has been reset, the pixel receiver must determine the current pixel and line based on the f, v, and h signals received after the reference codes. The pixel receiver uses the transition from f=0, v=1, h=0 to f=0, v=0, h=1 and then sets the current pixel and line to 721 and 18, respectively. It has four states: STATE_CHROMA_BLUE, STATE_LUMA_1, STATE_CHROMA_RED, and STATE_LUMA_2. After setting the current pixel and line values, this module also sets its state to STATE_CHROMA_BLUE. Then, on the next clock cycle, the pixel receiver module stores the eight-bits of the video signal in a register for the current blue chrominance value, and then it sets its state to STATE_LUMA_1. On the next clock cycle, the current

luminance value is stored in a register for the current luminance and then the state is set to STATE_CHROMA_RED. On the following clock cycle, the value is stored in a register for the current red chrominance value, and then the state is set to STATE_LUMA_2. Finally, on the next clock cycle, the current luminance value is stored in a register for the current luminance and then the state is set to STATE_CHROMA_BLUE. Then, the cycle repeats itself while simultaneously adjusting the pixel and line values every two clock cycles. On clock cycles for which the state is either STATE_LUMA_1 or STATE_LUMA_2, a signal pixel_ready is set to 1 for one clock cycle to indicate to the connected module that has the correct luminance and chrominance values for the current pixel.

3. Real Location Module (real_location.v)

The values for the current pixel are correct for active pixels, but as a result of interlacing and vertical blanking periods, the current line is usually incorrect. The real location module converts the NTSC line and pixel from the pixel receiver module to the location on the 720x485 resolution picture that it represents. This module determines the horizontal pixel value to be active if it is between 1 and 719, inclusively. The data for pixel 0 is inactive for this implementation because the data is incomplete, since it only has luminance and blue chrominance values, without red chrominance. The real location module determines the vertical line value to be active, if it is between 19 and 261, inclusively, sequentially corresponding to even lines from 0 to 484; or if it is between 282 and 523, inclusively, sequentially corresponding to odd lines from 1 to 483. If the horizontal pixel and the vertical line are both within the active range, then its correct mapping from the NTSC pixel to the real pixel is sent as output by the real location module, and another output, pixel_ready, is set to 1 for one clock cycle to indicate to the next module that the pixel is active.

4. Color Threshold (color_threshold.v), Point Summation (point_summation.v) and Division Modules (division.v)

In this implementation, there will be an orange point of reference on the pool table and a blue point of reference on the pool stick. The color threshold module determines whether an active pixel meets specified minima and maxima for each of the luminance and chrominance values. The values for detecting orange and blue are in the table below.

Table 1.

Threshold Values for Detecting Orange and Blue		
Field	Orange	Blue
Minimum Luminance	8'b00000000	8'b00000000
Maximum Luminance	8'b11111111	8'b11111111
Mininum Red Chrominance	8'b10100000	8'b00000000
Maximum Red Chrominance	8'b11111111	8'b10010100
Mininum Blue Chrominance	8'b00000000	8'b10001100
Maximum Blue Chrominance	8'b01111100	8'b11111111

If a particular pixel meets a threshold requirement, an enable signal, along with the current pixel and line, are sent as outputs by the color threshold module to two point summation modules, each for the pixel and line respectively. Thus, there are a total of two color threshold modules for the two colors, and a total of four point summation modules in the entire system.

When a point summation module receives an enable signal from a color threshold module, it takes the current value of either the pixel or line and adds it to a registered sum. In addition, it increments a registered count by 1. With these two values, the centers of mass of the x and y values for the two colors can be calculated with a division. And the end of each NTSC picture cycle, defined to be active pixel 719 and line 483 in this implementation, the color threshold module sends a signal to the point summation module for it to reset its registers for the sum and count to 0. Then, the point summation signal indicates to the division module that it has completed its summation. This signal enables division. The division module receives the registered sum and count from the point summation module and divides the two. There are four division modules in the system that serve this purpose.

5. Distance and Angle Calculation Module (angle_calculation.v)

Using the centers of mass for the pixels that meet the orange and blue thresholds, this module internally calculates the x-distance and the y-distance between the two points. The distance between the points will then be used by this module to calculate whether the squared distance: $x^2 + y^2$ are within a certain distance of each other. If the two points are close enough, then this means that a strike has been made to the pool ball.

In addition to calculating the distance this module approximates the angle by comparing the x and y components. This module evaluates the power of two that separates the x- and y-distances by comparing the bits of one component to the shifted bits of another component. The angle calculation is precise to 4 bits of comparison. There are 32 possible directions. The figure below shows the ratios between the x- and y-components that correspond to 9 of these angles. These can be applied to negative x- and/or y-components to get the remaining directions.

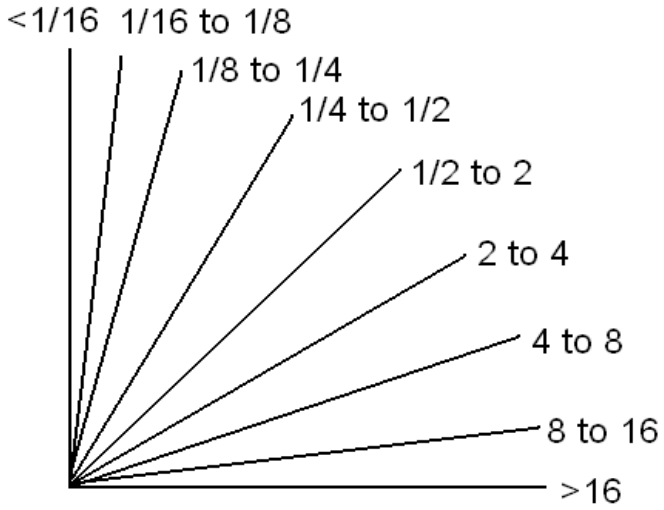

Figure 4: Ratio Between Positive x- and y-Components and Their Corresponding Angles

6. ADC Input Module (adc_in.v)

This module provides an enable signal to the AD7810Y analog-to-digital converter in order to digitize a voltage from the DE-ACC3D accelerometer. The ADC input module also sends a clock signal to set up the individual bits from the ADC. Simultaneously, this module receives the bits,

starting with the most significant of the 10 bits and stores it. Finally, when all bits have been stored, this value is sent as output for further calculations. This module runs in a continuous cycle of sending an enable signal followed by clock cycles and receiving data.

7. Acceleration Peak Module (acceleration_peak.v)

This module simply evaluates the maximum value that the ADC input module has sent as output within the past second.

8. Speed Parametrizer Module (speed_parametrizer.v)

Using the distance and angle calculation module and acceleration peak module. The speed parametrizer module first transforms the acceleration peak into a value that represents the intensity of the pool ball strike and then finally, based on the angle, converts it into x- and y-speed parameters. These speed values combined with a value to indicate a pool strike is sent as output from the input super-module to the physics and game super-module.

9. Testing the Inputs Module

To test this module, there is a graphical interface to track the centers of mass of the two points on the pool table and the current acceleration from the ADC. Figure 5 shows the graphical display used to test the inputs module.

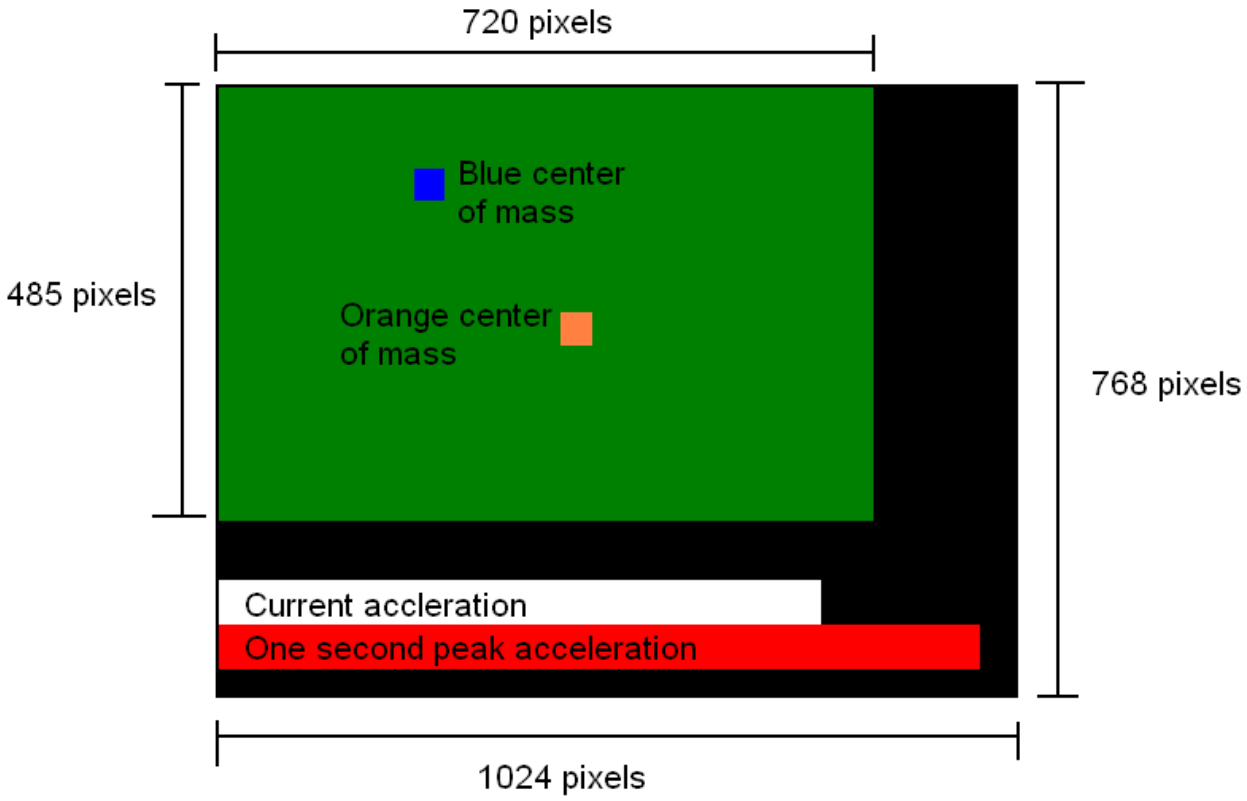

Figure 5: Testing Interface for Inputs Module.

Pool Game Physics Simulator and Gaming Logic Console

1. General Overview

The Gaming Logic Console was designed to model the physics and logical progression of a two player game of pool. This module receives data from the input module representing shots the user(s) take. From this, the positions of all 16 balls on the table are output to the 3D graphics module at a rate of 75 cycles per second to match this systems' on-screen refresh rate.

The Gaming Logic Console monitors the operation of 16 individual ball modules, a Collision Manger FSM, a Wall Dynamics Module and a Collision Dynamics Module (Figure 7). This architecture allows for the physical state of each ball to be computed and manipulated in a concurrent manner, and also simulation of ball collisions and interactions. In the Gaming Logic Console balls are to maneuver over a 1000x2000 unit region representing the pool table. Balls can collide with each other or the edges of the table (or the walls), and can also be scored in the pockets of the table. The dimensions for the elements of the pool table are outlined in figure 1.

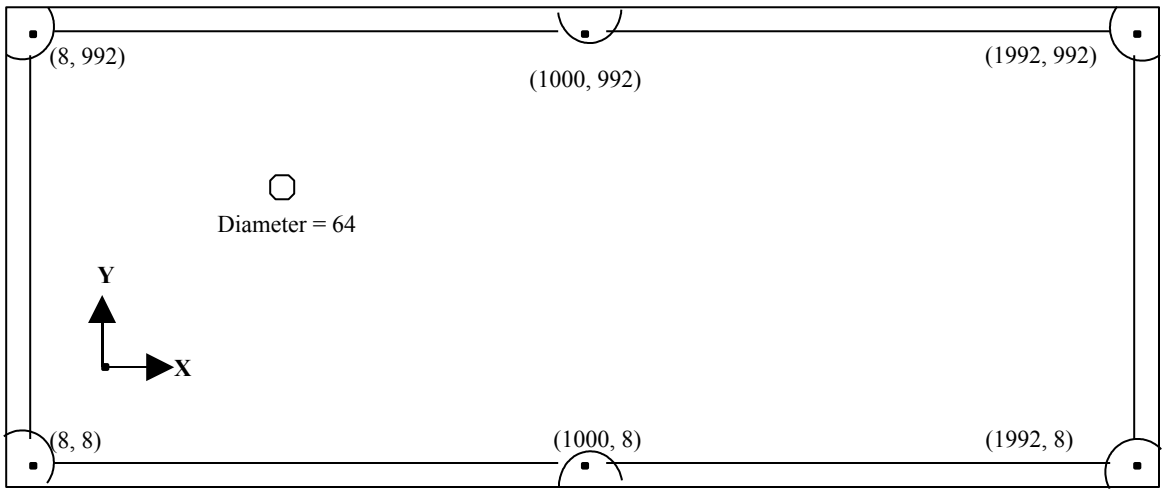

Figure 6: Pool Table Schematic, the dimensions and table layout. The Table dimensions are 2000 x 1000, the wall pads are 8 units in thickness.

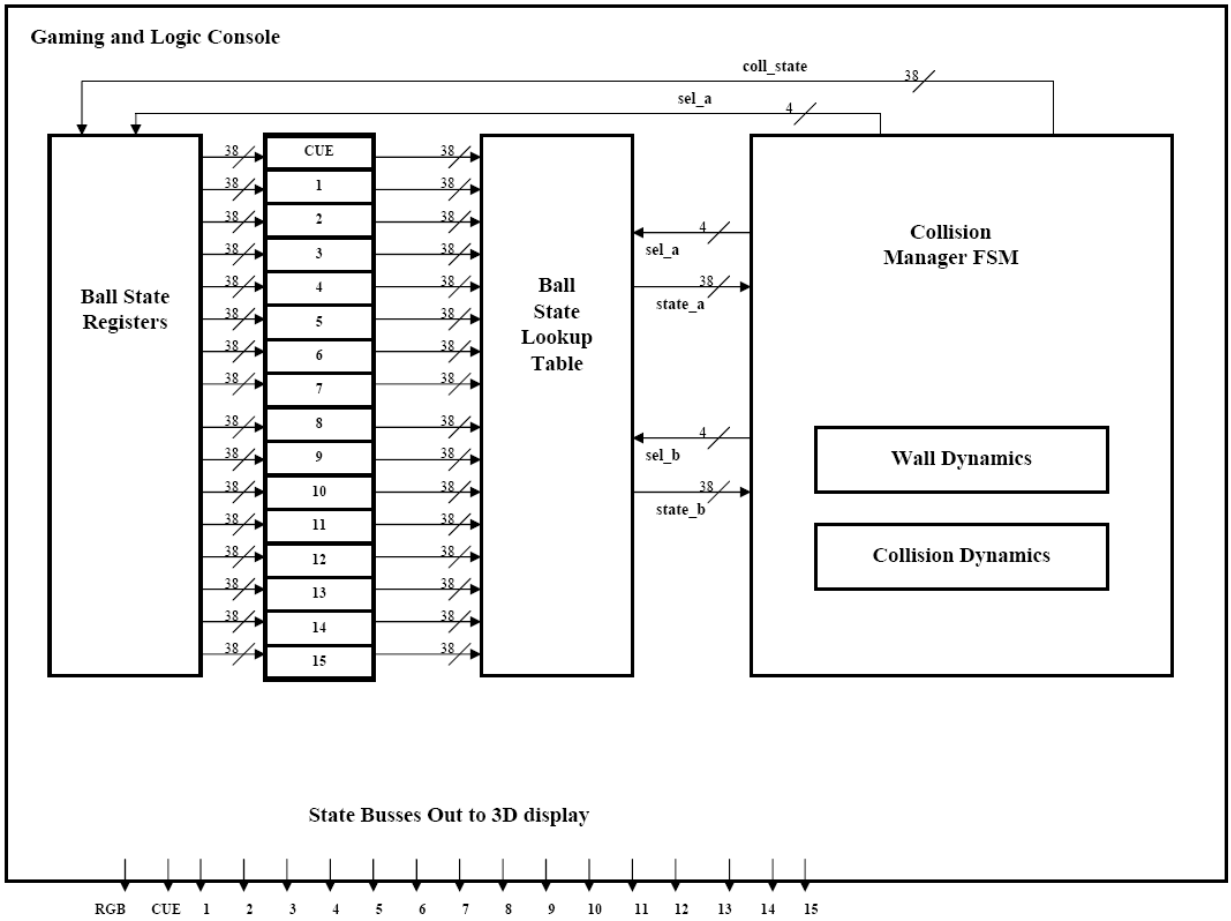

Figure 7: Implementation of the Gaming and Logic Console. This system controls the states of all balls n the table and outputs positions to the 3D display unit.

Each ball module is comprised of a "Ball State Register" and a "Ball State FSM" (Figure 8). The state register stores the physical state of the ball and is updated at every frame cycle. The Ball State FSM is what allows each ball to maneuver on the pool table, modeling the physical characteristics of a rolling sphere subjected to non-conservative forces in motion such as retardation. The Collision Manger FSM uses two lookup tables implemented as 16 to 1 multiplexers to check each ball coupling for collisions, and uses the Wall Dynamics Module and the Collision Dynamics Module to compute the final ball velocities resulting from ball-to-wall and ball-to-ball collisions respectively.

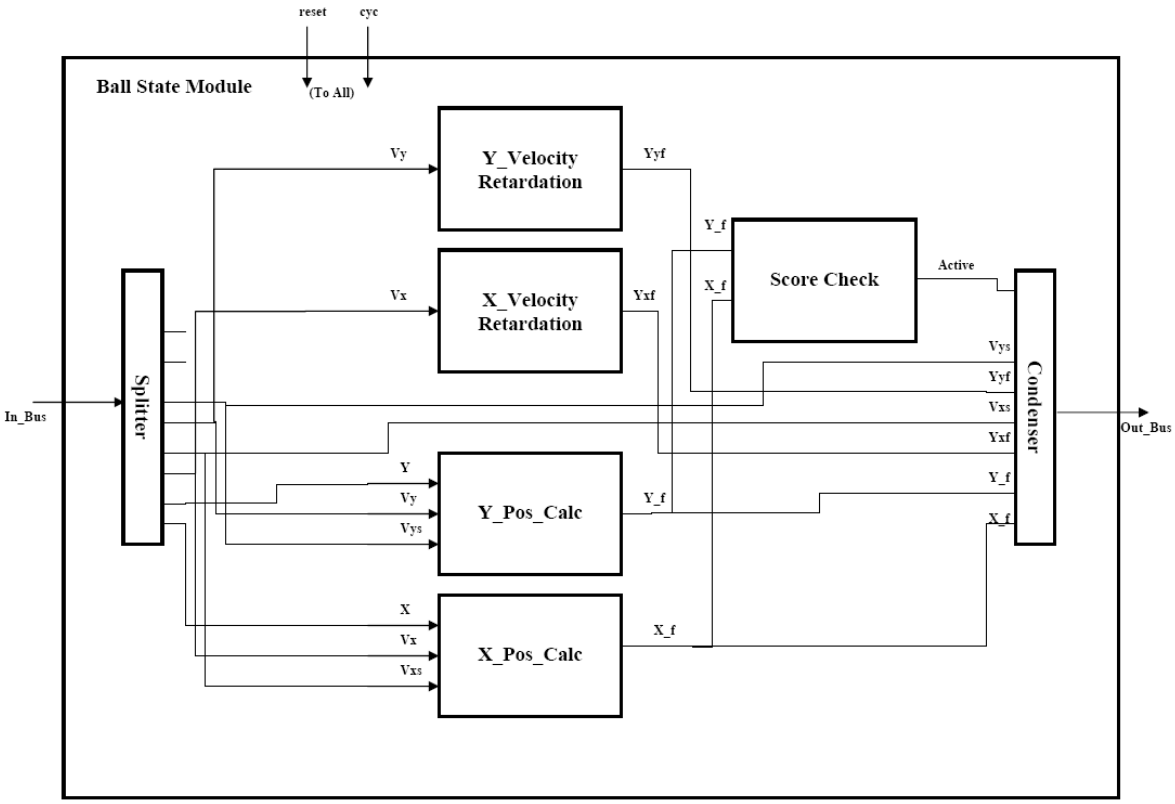

Figure 8: Ball State Module. This module controls the position of a ball on the table, and simulates the balls motion with retardation forces at a given velocity. The Score Check module determines if the ball has been scored.

2. State Bus Implementation

To decrease the number of individual inter-module connections for each ball state variable, the state variables of each ball are compressed into 38-bit state buses. The state bus provides the Gaming Console an abstraction for the transport of ball state data. The data positioning of state variables in the bus are outlined in table 1. This data structure is what is primarily used to propagate ball state information from one module in the Gaming Console to another.

Ball State Bus Implementation		
State Variable	**Bit or Range of Bits**	**Decription**
X	Bits 0 -10	X coordinate of ball on table
Y	Bits 11-21	Y coordinate of ball on table
X_VEL	Bits 22-27	Velocity of ball on the X-axis
X VEL SIGN	Bit 28	Direction of X-Velocity (0 for positive, 1 for negative in X)
Y VEL	Bits 29-34	Velocity of ball on the Y-axis
Y VEL SIGN	Bit 35	Direction of Y-Velocity (0 for positive, 1 for negative in Y)
STILL	Bit 36	This bit is high when the ball is still on table
ACTIVE	Bit 37	This bit is high when the ball in on table, low when scored

Table 2: The state bus implementation. Here are all of the state variable assignments for ball state data that is transported to and from modules in the Gaming Logic Console.

The state bus abstraction is completed with constructor modules known as “condensers” and selector modules known as “splitters”.

3. Collision Management and Dynamics

The Collision Management system within the Gaming and Logic Console implements a Finite State Machine to check each ball for collisions with the wall and collisions with other balls at each frame cycle. Essentially this module controls two balls selectors that are referenced by lookup tables with every state bus fed into it. By controlling these two selection addresses the FSM is able to loop through all balls on the table checking for collisions. The state diagram for this state machine is outlined in Figure 9.

For each ball, collisions with the wall are detected. In the event of a wall collision the velocity of the ball undergoing the collision is changed. For ball-to-ball collisions a second lookup table with the second selection address is used to couple the balls, and determine if collisions between each ball occur. This results in the creation of an inner loop where each ball where is checked for collision conditions with every other ball.

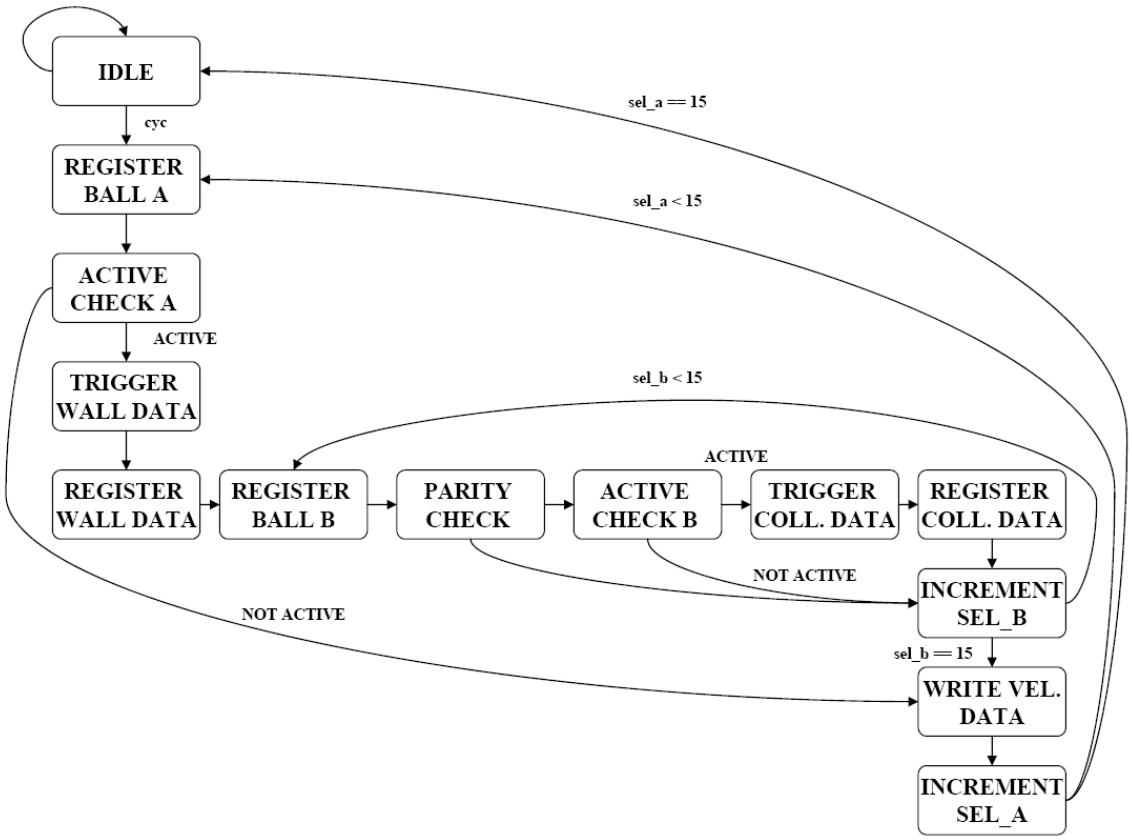

Figure 9: Control Manager State Diagram. This FSM checks the state of each ball for collisions with the wall of the table. Then it couples in all other balls on the table to check for ball to ball interactions. Sel_a is the index of the ball being checked, sel_b is the index of the ball being checked for a collision with ball A.

The final velocity components of a ball resulting from a collision are computed mathematically (figure 5) as a function of its own initial velocity, that of the coupled ball, and the offset of the collision.

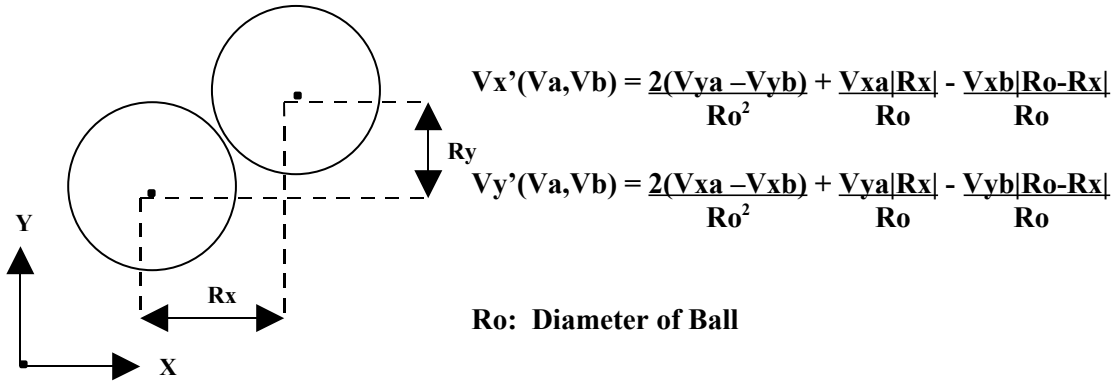

Figure 10: The final velocity components of the a ball resulting from a collision are computed as function of the velocities of both ball involved in the collision.

This calculation is done for every coupling however these new components are registered for the velocity of a ball if and only if that ball is involved in a collision. This is to occur on the "Register Collision Data" state of the State Machine. The computation structure is an assembly of 6-bit subtracters, adders and multipliers (Figure 11). This system is able to compute the set of final velocity components Vx' and Yy' for one ball in a collision. Since each ball on the table is checked against every other ball, every ball involved in a collision will have it's final components computed and registered on the same progression of the State Machine in one frame. After the FSM is finished with a ball it compresses the new resulting velocity components into a new state bus which is written to the state register of that ball.

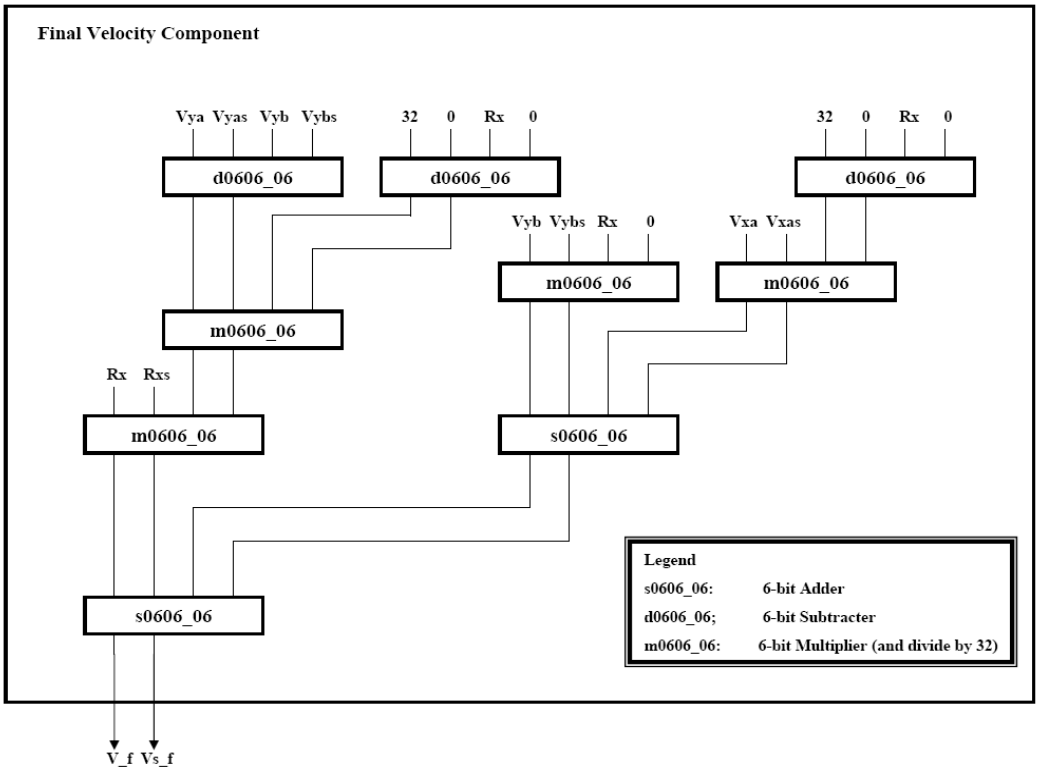

Figure 11: This is the implementation of the system that computed the result for on component of the collision dynamics expression in figure 5. For all components the magnitude of an integer is represented with a 6-bit number, such as Vya or V_f, it is accompanied with a sign bit (Vyas or Vs_f).

4. System Performance and Results

The implementation and of the Gaming and Logic Console had critical defects which would result from synthesis on the FPGA. The game of pool would turn out to be a very complex system of interconnected modules especially if the state of all 16 balls is pipelined in the design. Despite the fact that the all of the elements of the design could be implemented on the space on the FPGA, there weren't enough interconnections left to connect all of the modules together. Another major fault that contributed to the failure of this design was the volume of data transported around the chip. The interconnections between several instantiations were occupied by 38-bit busses. This would make connecting the elements of the system more difficult with the minimal resources that were available for interconnections. A remedy for both of these issues in this design would enable the use of memory on the chip. Implementing Block Ram to store the state variables of the balls would reduce the complexity of the Gaming Console significantly, however the Control FSM would be more involved and overall the system will be much slower.

Three-Dimensional Graphics Module

1. Introduction

Perspective ray tracing is a computationally intensive method of rendering objects in 3-D. Given the position of an eye viewing an object or scene, a virtual screen is created by making a horizontal cut between the eye and the scene to be viewed. To determine the color values of a particular pixel, a line is drawn from the eye through the position of the pixel on the virtual screen and extrapolated into the scene to be viewed. If the line does not intersect an object in the scene, the color of the pixel is set to black. If the pixel intersects an object in the scene, the U and V values of the pixel are set to those of the object it intersected. Another line is then drawn from the first point of intersection to the light source. If this line intersects another object, the Y value of the pixel is set to a constant G. This G represents the amount of illumination that would be present in a shadow in the scene due to light that bounces off other objects. If the second line does not intersect another object, first we get the dot product of the normal of the surface at the first point of intersection, and the vector from the first point of intersection to the light source. The maximum value of Y, minus G is multiplied by this dot product. G is added to the resulting value, to give the Y value of the pixel. The YUV values for each pixel are converted to RGB and written to the first ZBT SRAM. When the last pixel's RGB value has been written to memory, the ZBT SRAM's are switched so that values are written to the second SRAM. Whenever values are being written to one of the SRAM's, the RGB values are read from the other SRAM and displayed on the screen.

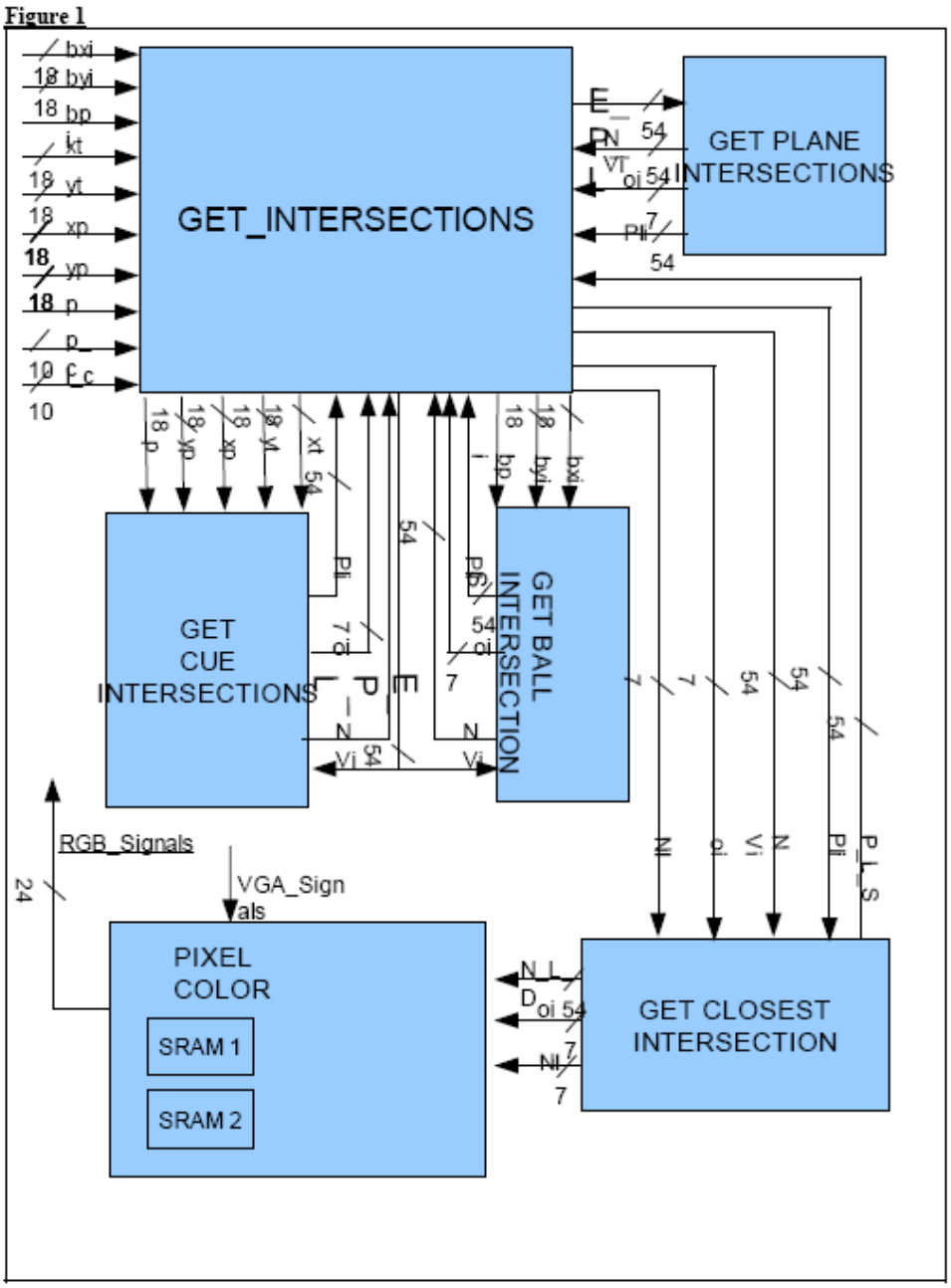

Figure 12: Block Diagram for Perspective Ray Tracing

2. EYE_PLANE_LINE Module

Given the pixel and line count, this module calculates the x, y and z coordinate of the pixel on the virtual screen.

3. GET_BALL_INTERSECTION Module

This module takes in the x, y and z coordinates of the center of a ball (x3, y3, and z3). It also takes in two points on a line(x1, x2, y1, y2, z1 and z2). It calculates:

a = (x2 – x1)^2 + (y2 – y1)^2 + (z2 – z1)^2

b = 2[(x2 - x1) (x1 - x3) + (y2 - y1) (y1 - y3) + (z2 - z1) (z1 - z3)]

c = x3^2 + y3^2 + z3^2 + x1^2 + y1^2 + z1^2 - 2[x3 x1 + y3 y1 + z3 z1] - r^2
and solves for the smaller value of u in the quadratic equation: au2+bu+c = 0;
Solution: -b -sqrt(b*b – 4ac)/2a

If (b*b - 4 ac) is less than zero, it sets it's output signal *Pin* to 1, indicating that the line does not intersect the sphere. Otherwise, it sets *Pin* to 2, indicating that there is an intersection and returns the x, y and z coordinates of the intersection given by:

x= u (x2-x1); y = u (y2-y1) and z = u(z2-z1).

Before it completes calculating u and the point of intersection, *Pin* is set to 4 to indicate that the computation is in progress.

4. GET_PLANE_INTERSECTION Module

This module takes three points (x1, x2, x3, y1, y2, y3, z1, z2, and z3) defining a finite rectangular plane. It calculates the coefficients for the equation of the plane:
A = y1 (z2 - z3) + y2 (z3 - z1) + y3 (z1 - z2)
B = z1 (x2 - x3) + z2 (x3 - x1) + z3 (x1 - x2)
C = x1 (y2 - y3) + x2 (y3 - y1) + x3 (y1 - y2)
D = -(x1 (y2 z3 - y3 z2) + x2 (y3 z1 - y1 z3) + x3 (y1 z2 - y2 z1))

It also takes two points on a line (X1, X2, Y1, Y2, Z1 and Z2) and calculates:
Num = (A X1 + B Y1 + C Z1 +D)
Den = (A (X1-X2) + B (Y1-Y2) + C (Z1-Z2))

If Den is equal to zero, it sets it's *Pin* output signal to 1 indicating that the line does not intersect the plane. Otherwise it calculates:
x4=(x2-x1), y4=u(y2-y1) and z4 = u (z2-z1).

It also calculates:
num1= (y4-y1)-((y2-y1)(x4-x1)/(x2-x1)); den1=(y3-y4)-((y2-y1)(x3-x1)/(x2-x1))
v = num1/den1; u = ((x4-x1) – v(x3-x1/(x2-x1));

If both u and v are less than 1, it sets it's output signal *Pin* to 2 indicating that the line intersects the finite plane and returns the point of intersection x4, y4 and z4, otherwise it sets it's output signal *Pin* to 1, to indicate that the line does not intersect the plane. Before this module finishes calculating u and v, *Pin* is set equal to 4 to indicate that the computation is in progress.

5. CLOSEST Module

This module takes 6 points (x1...x6, y1...y6, z1...z6) as input and returns the point that is closest to the eye:
min{(Eye_x – xi)^2 + (Eye_y – yi)^2 +(Eye_z – zi)^2} for 1<=i<=6.

6. GET_CUBE_INTERSECTIONS Module

This module takes 2 points on a line and 6 sets of three points that each define a rectangular plane. It calls GET_PLANE_INTERSECTION for each of the 6 planes and then calls the closest module to get the closest point of intersection of the line defined by the 2 points and the six finite planes.

7. GET_CUE_INTERSECTION Module

Given two points on a line (e1 and e2) and the x, y and z co-ordinates of 2 points on the cue (p: px,py,pz and t: tx,ty,tz), this module calculates p2: px2= px – (((px-tx)/CueLength)*CueWidth); py2= py – (((py-ty)/CueLength)*CueWidth) and pz2 = pz, if the gradient ((px-tx)/(py-ty)) is positive. t2 is calculated similarly from t. The calculations for p2 and t2 when the gradient is equal to zero or negative are similar. With these four points, it gets 4 other points (ps, p2s, ts and t2s)by subtracting the CueWidth from the value of the z coordinate of each point. It then calls GET_CUBE_INTERSECTIONS, with the 6 planes that are defined by connecting p, p2, t, t2, ps, p2s, ts and t2s. If the line defined by e1 and e2 intersects one of these planes, it returns the closest point of intersection and the normal to the plane in which the closest point of intersection is found.

8. GET_PLANES_INTERSECTIONS Module

Is a module that calls GET_CUBE_INTERSECTIONS with 12 planes that form the pool table. It also calls GET_PLANE_INTERSECTION with the plane that forms the top of the pool table and checks whether the intersection point returned falls in one of the holes of the pool table, in which case it sets the color of the intersection point to black. It returns the closest point of intersection with the planes that form the pool table.

9. GET_BALLS_INTERSECTION Module

This module calls the GET_BALL_INTERSECTION module 15 times, (once for each of the pool balls) and then calls CLOSEST twice to get the closest point of intersection with any of the pool balls.

10. GET_INTERSECTIONS Module

This module calls the GET_PLANES_INTERSECTIONS, GET_BALL_INTERSECTIONS and GET_CUE_INTERSECTIONS modules. It then calls the CLOSEST MODULE to get the closest point of intersection and returns the x, y and z coordinates and the normal of the closest point of intersection.

11. PIXEL_COLOR Module

This module calls the EYE_PIXEL_LINE module to get the x, y, and z coordinates of the current pixel on the virtual screen. It then calls the GET_INTERSECTIONS module, first to get the closest point of intersection with any object in the scene and a second time to get the closest point of intersection with any object between the first point and the light source in the scene. If the first point does not exist (*Pin* signal of GET_INTERSECTION equal to 1), the Y value of the pixel whose color was being calculated is set to 0. If the first point exists, the U and V values of the color of the pixel are set to those of the object that contains the first point. If the second object is equal to the first, the point is not in shadow and the Y value of the point is set to (MAX_Y – G)*DOT; Where MAX_Y = 256, G=60 and DOT is the dot product of the normal at the first point and the unit vector between the first point and the light source. The YUV values are converted to RGB values as shown below:

$$R = Y + 1.13983\ V$$
$$G = Y - 0.39466\ U - 0.58060\ V$$
$$B = Y + 2.03211\ U$$

The RGB values obtained are then written to the address (line_count*799 + pixel_count) in the first SRAM. While values are read from the second SRAM and sent to the display. When the line_count and pixel_count are both equal to 1, the SRAM's are switched so that values are read from the first SRAM

and written to the second SRAM.

12. Testing and Debugging

Unfortunately, I was unable to fully debug the ray tracing modules, however I did learn a lot from the process which took me 6 days after I had finished writing all the modules. The central problem I had was a attempting to implement a design that was too large for the FPGA. The error I got repeatedly was:

ERROR:Pack:18 - The design is too large for the given device and package.

The reason I got this error was because I attempted to do most of the computations in the modules in parallel, using combinational logic. 32-bit single cycle dividers and multipliers take up more space on the FPGA than I had thought. The solution to this problem is to use FSMs instead of combinational logic and do the computations one at a time. For example, in the GET_BALL_INTERSECTIONS module, I would do the calculation for the first ball and when that was complete, transition to another state where I would do the calculation for the second ball. The individual modules alone compile and get mapped to the LUT's on the FPGA successfully but when they are combined, the design becomes too large. While the solution of using an FSM instead of combinational logic is a simple one, I discovered it late after implementing and attempting to debug the original design for six days. The problem was compounded by the fact that xilinx 8.2 ise in the lab gives an uninformative error message that says (The device has encountered an exceptional condition and needs to terminate). It was only after I tried to compile my design on xilinx 9.1 ise that I got the error message above and begun trying to reduce the size of my design. The lesson learned is an important one despite the disappointment of not completing my part of the final project.

Conclusions

The Camera and Accelerometer Input Module was completed successfully. Together they can trace a moving point of reference of a particular color in a camera's field of view. Also, it could accurately detect and calculate the movement of the pool stick and approximate the angle between reference points for speed parametrization.

The Physics and Game Logic Console had crucial defects due to having a large number of interconnections. Though it successfully synthesized to the FPGA, the number of interconnections were impractical for proper functionality. Using the block RAM attached to the FPGA would have been a more practical option for it would have dramatically decreased the complexity of this console and the entire project as a whole.

The Three-Dimensional Graphics Module was too large to be programmed onto the FPGA and needs to be reduced in size by using FSM's for successive computations instead of attempting to perform all computations in parallel. This is a project that I (Timothy M. Mwangi) will complete over the summer for my own satisfaction. Were it possible, I would request for an incomplete in the class and complete my part of the final project over the summer. The lessons learned in digital design during the final project were very important. Specifically, the need to economize on the resources available to a digital designer and the need to do incremental development of large systems were the most important lessons that we learned during the final project. In retrospect, the size and scope of our project was simply too large. More so because none of us had prior experience in 3-D graphics.