ECE532 – FPGA Billiards

Perter Lin
David McTavish
Peng Xue

April 2015

Contents

List of Figures

1 Overview

1.1 Introduction

The FPGA application we have implemented is a 2 player billiards game that uses the standard eight ball rule. The setup includes a monitor which displays the game state, a camera which tracks the position of a cue stick which the player can use to hit the cue ball. The motivation behind this project is that it is an alternative to real life billiard which has a high purchase cost and can be cumbersome to setup. While there are already video games which simulate billiards, these games are played using a controller or a mouse and do not provide the physical experience of playing billiards. We believe our game will this problem as an inexpensive alternative to real life billiards without sacrificing the physical experience.

1.2 Goal

We want to create a realistic billiards experience that will not be dampened by the need to learn a new controller system. With this pool-cue interface, we hope to create a life-like billiards experience that will feel like and look like the traditional game, but without the financial and spatial commitment.

1.3 System Block Diagram

The block diagram for the final system is given in Figure 1.

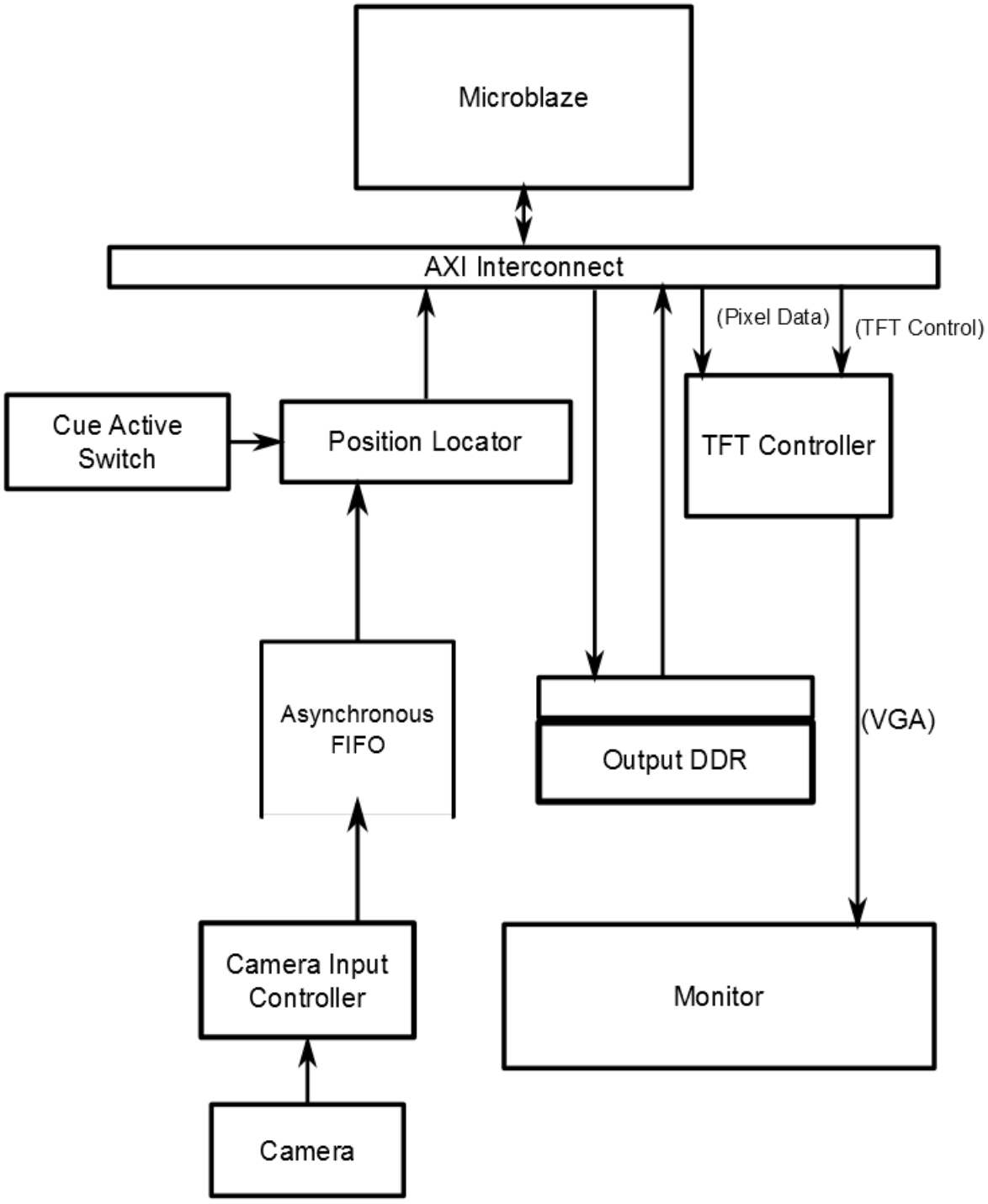

Figure 1 - Block diagram of all hardware components in the system.

1.4 Brief Description of IP Used

IP	Purpose	Source
Input Hardware		
OV7670 Camera	Looking down on the table, takes video of the screen and the red LED, representing the cue stick.	Produced by OmniVision. The Board to interface with the Nexys-4-DDR board was created by Digilent
Camera Input Controller	Configured the camera and received the frame data from it.	Based off of the source files in 532_pmod_camera_dist.zip, posted on the ECE532 2015 Piazza board. Modified by the group.
Asynchronous FIFO	Stored the pixels between in camera and the position locator	Xilinx IP
Position Locator	Processed the stream of pixels to determine the (x, y) position of the red LED on the screen.	Custom IP created by the group.
Processing Hardware		
AXI Interconnect	This interface coordinated transferring data between the hardware blocks.	Xilinx IP
Microblaze Soft Processor	Contained the code to process the input positions from the position locator, conduct physics simulations on the pool balls, and write the output frames to memory.	Xilinx IP
DDR Memory	Stored the output video frames.	External memory on the Digilent Nexys-4-DRR board
Processing Software		
Pool Cue Software Interface	Polled the position locator, and gave the cue ball its initial speed for the physics simulation.	Code written by the group
Physics and Game Library	Processes the movement and collision of the balls in the game.	Code written by the group
Drawing Library	Managed drawing the output video frames using data from the physics simulation. Also managed the TFT controller to implement double buffering, and drawing player UI information.	Code written by the group
Output Hardware		
TFT Controller	Reads the output frames from the DDR memory and drives the VGA signals to the monitor.	Xilinx IP

Monitor	Displayed the output image of the table.	Any 640×480 computer module with a VGA port.

2 Outcome

The overall goal of creating a simulated pool game on an FPGA was achieved.

There were three requirements of the system:

- Using the LED and position tracker to emulate the feel of using a pool cue
- Creating a realistic physics simulation of the balls moving and colliding during the game
- Draw the entire game in real time

2.1.1 Pool Cue Input

The pool cue's purpose was to be able to track the red light at the end of the cue, and, when it collides with the cue ball, give the ball an initial speed. The current system allows this basic action. The players can hit the cue ball hard or soft, depending on how fast they move the cue when they hit it, simulating one of the key aspects of making a shot in billiards.

However, there were several aspects of the performance that could be improved.

Firstly, the position tracking was not very precise. When players tried to hold the cue steady, the position read by the locator shook across an area several pixels wide. This meant that when hitting the cue ball, if they position was fluctuating, they would hit the ball from an angle they did not indent to. Making a straight shot is very difficult in the game. This may be able to be fixed by making a more sophisticated algorithm to determine the speed of the cue. Currently, the speed is averaged over a small number of points, which smoothed some of the fluctuations, but was not sufficient.

The position locator was also affected by noise from the outside environment. If there was a single pixel of red on the other side of the screen, the locator would pick it up and skew the average position calculated. Future designers may wish to implement a more sophisticated filter than the averaging filter. For example, a filter could consider a sliding window of multiple pixels, and only accept the position if all pixels in the window are red. This would require some extra memory to store extra pixels in a shift register and extra logic, but that was not the limiting factor in the design.

2.1.2 Physics Simulation

The initial goal of the physics engine was to create a high performance physics library that would not be the bottleneck of the system. The engine needs to be able to compute the collisions of a 16-ball game accurately without loss of frame rate. These goals were achieved by the final iteration of the physics engine.

The final iteration of the physics engine allows accurate collision detection of all 2-D collisions. The accuracy of the angle detection is 3%. The accuracy of the position and velocities is 1/128

pixels. The collision detection uses a circle collider check, comparing the straight-line distance between the centres of the balls against twice the radius of the balls.

Currently, the collisions are checked every frame. This causes collisions to sometimes be checked when the balls overlap. The overlapped state yields a different collision angle than what would be expected. To alleviate this, the collision library retroactively determines when the balls should have collided (between the frames), and shifts the balls back to the initial collision state. This reverse shift has an accuracy of 1/128 of a frame.

This retroactive collision detection mechanism is not operating a full effectiveness. Since the module is activated when two balls are first overlapped, it is assumed that the previous frame does not have the two balls touching. Therefore, a retroactive time step is expected to be 0 - 1 frames from the last frame. However, since collisions are only checked for balls in motion, and friction is applied as a constant (rather than a percentage of velocity), sometimes balls will end up overlapped but with zero velocity. This causes a subsequent check to determine an enormous time step is required to separate the balls (often resulting in a long long integer overflow). Bounding the value of this time step helps mitigate the problem, but still exposes the issue of this module sometimes being inaccurate.

Due to the lack of floating point division (floating point ALU was too large to fit in the design), divide operations were mimicked with bit-shifts. Friction, being a percentage, could not be adequately represented with a bit-shift operation. Instead, friction was implemented as a constant decrement of both x and y components of the velocity. At speeds with a dominant x or y component, this leads to an apparent curvature in the ball's trajectory.

2.1.3 Output Video

A majority of the proposed feature for the output video display was implemented. We were able to display all sixteen balls on screen with their respective colour, draw the position of the pockets and indicate whose turn it is as well as animate the end of the game. One feature that was limited by the speed of the TFT controller is the size of the balls which could be drawn. With sixteen balls on screen at the same time, the balls could only be 10 pixels in radius before there was detectable lag in the frame rate.

The implementation of the video output using the microblaze and TFT controller can be considered successful as it is able to display all relevant game information to the user and achieve acceptable frame rate.

One improvement which could be made if given more time is a more efficient drawing algorithm. Currently, all balls are erased and redrawn regardless if they are moving or not. In a more optimal implementation, only moving balls would be drawn which could potentially increase the performance of the video output. Another way to improve the quality of the video output is to use a more powerful video controller. The TFT controller used was only capable of displaying images in a 640x480 frame which results in a game which is not up to par with modern graphic standards.

3 Project Schedule

3.1 Original Proposed Schedule

- **Milestone 1 (Feb 10)**
  - Write drawing API for basic shapes, output static image to screen using TFT controller
  - Implement a tracking algorithm in C (for LED position)
  - C structure for the storage of the position and velocity of each ball
  - Create physics for ball-pool cue collisions (sourced velocity vector)
- **Milestone 2 (Feb 24)**
  - Create physics for single ball-ball collisions
  - Create physics for ball-pool cue collisions (derive velocity vector)
  - Create test frame images (for tracking)
  - Design tracking algorithm to work on FPGA
  - Read video data from camera into DDR. Read video into position tracker.
- **Milestone 3 (Mar 3)**
  - Create physics for ball-wall collisions
  - Create physics for ball collisions with pocket
  - Integrate: Generate video output from physics collisions
  - Implement double buffering in creating output video
  - Implement tracking algorithm on FPGA. Test using the same frames as the C version.
- **Milestone 4 (Mar 10)**
  - Integrate: Position tracker using the live video frames
  - Enable/Disable cue tracking when button depressed/released
  - Integrate: Create physics for deriving velocity vector from position tracker.
- **Milestone 5 (Mar 17)**
  - Implement double buffering in reading input camera video
  - Create physics for simultaneous collisions
  - Create game rule: Remove balls when they fall into the pocket
  - Create Game Rule: switching between player's turns
- **Milestone 6 (Mar 23)**
  - Create game rule: Striking opposing ball with cue ball
  - Create game rule: Losing when sinking the 8-ball inappropriately
  - Read gyroscope orientation. Implement backspin when striking cue ball.

3.2 Achieved Schedule

- **Milestone 1 (Feb 10)**
  - Created C++ structure for the storage of the position and velocity of each ball
  - Create architecture to track and update positions/velocities of all balls
  - Created box colliders. The distances between the ball's coordinates are compared directly to 2 x Radius.
  - Design and implemented averaging filter for position locator. (A C model was determined to be unnecessary for the algorithm.) Module currently untested.
- **Milestone 2 (Feb 24)**
  - Create physics for single ball-ball head on collisions.

    - Created angles.hpp library (trigonometric library)
    - Wrote drawing API for basic shapes of rectangles and circles, TFT and microblaze system built to be able to output static image to display.
    - Created simple testbench and simulated position locator module. Verified to be working for a 5 line frame. The testbench also simulated the FIFO signals to the module, to verify that interface.
    - Looked over the source from 532_pmod_camera_dist.zip to learn how to get pixels from the camera to the position locator.
- **Milestone 3 (Mar 3)**
    - Implemented all ball-ball collisions. This includes head on collisions, one ball stationary glancing collisions, and 2 moving ball collisions.
    - Created physics for ball-wall collisions. Balls simply bounce back from the wall. (invert x or y component of speed).
    - Created collision manager for all ball-ball collisions. Collision manager determines the appropriate calculations to be applied to the given scenario.
    - Created a friction mechanism. Decrement the speed per frame by a fixed amount.
    - Created an integer square root function. Square root can be calculated in one pass over the bits in the integer (constant time). This scales well.
    - Modified the code from 532_pmod_camera_dist to write the pixel data into a FIFO instead of BRAM.
    - Modified position locator module:
    - Use Xilinx divider generator IP to be able to run module at 100 MHz
    - Added a configuration register, so that the tolerances could be set in software
    - Integrated camera input and position locator. The position locator can process the camera pixels in real time.
    - Wrote simple microblaze polling code to interface microblaze with position locator.
- **Milestone 4 (Mar 10)**
    - Create physics for ball collisions with pocket.
    - Removed ball when ball is sunk into pocket
    - Created a list to track all balls in motion. Only check for collisions if the ball is in motion.
    - Integrated TFT system with physic collision system, balls are visible and moving on screen
    - Implemented double buffering on TFT camera for smoother frame transition
    - Modified camera input module to configure camera from 640x480 frames, instead of 320x240. Designed a test module that could read the 640x480 camera data but output only every other pixel to display 320x240 to the monitor.
    - Wrote software to poll the position locator for multiple positions, check for collision with cue ball, and convert coordinates from camera space to table space.
- **Milestone 5 (Mar 17)**
    - Created physics for multiple simultaneous collisions. Each pair of collisions is calculated separately and scaled by the appropriate conservation of momentum factor.
    - Created a list to track all colliding balls. Makes multiple simultaneous collisions easier to handle.

    - Created imminent collision detection module to determine exactly when the balls first touched (which could potentially be between the current frame and the last frame).
    - Implemented drawing of pockets
    - Added feature to TFT for displaying player turn and score
    - Built the frame to hold the camera above the monitor.
    - Integrated software with the camera/frame/screen system.
- **Milestone 6 (Mar 23)**
    - Migrated from 10-bit precision to 7-bit precision due to integer overflow.
    - Upgraded friction to only be applied every fourth frame. With a minimum friction of 1, and 7-bits of precision, this would have led to a maximum of 128 frames before the ball stopped.
    - Modified scoreboard for 8 ball rule
    - Added 8 ball scratch rules for switching for player turn
    - Built red LED wand to serve for the cue.
    - Integrated resent physics changes into the full system.
    - Cleaned up testing code in the software.
    - Using previously written cue ball functions, allowed the player to replace the ball anywhere on the table. Will be used in the game whenever an player scratches
- **Milestone 7 (Mar 31)**
    - Modified scratch rules to integrate with new collision prediction engine
    - Added extra phase (and indicators) to game where a player would place thecue ball if it has been sunk
    - Modified resistor in LED wand to have a higher resistance, so that the light is dimmer. (Previously, it saturated the camera and appeared white instead of red).
    - Cleaned up cue ball software after the switch from 10 bit precision to 7 bit precision in the physics library.

4 Description of Blocks

4.1 Physical Structure

4.1.1 Frame

In order to hold the camera above the screen, as simple wooden frame was constructed. The camera board was attached to the top piece, pointing down. The FPGA board, connected to the camera by a ribbon cable, also rested on top of the frame.

A pair of photographs of the frame is show in Figure 2 and Figure 3.

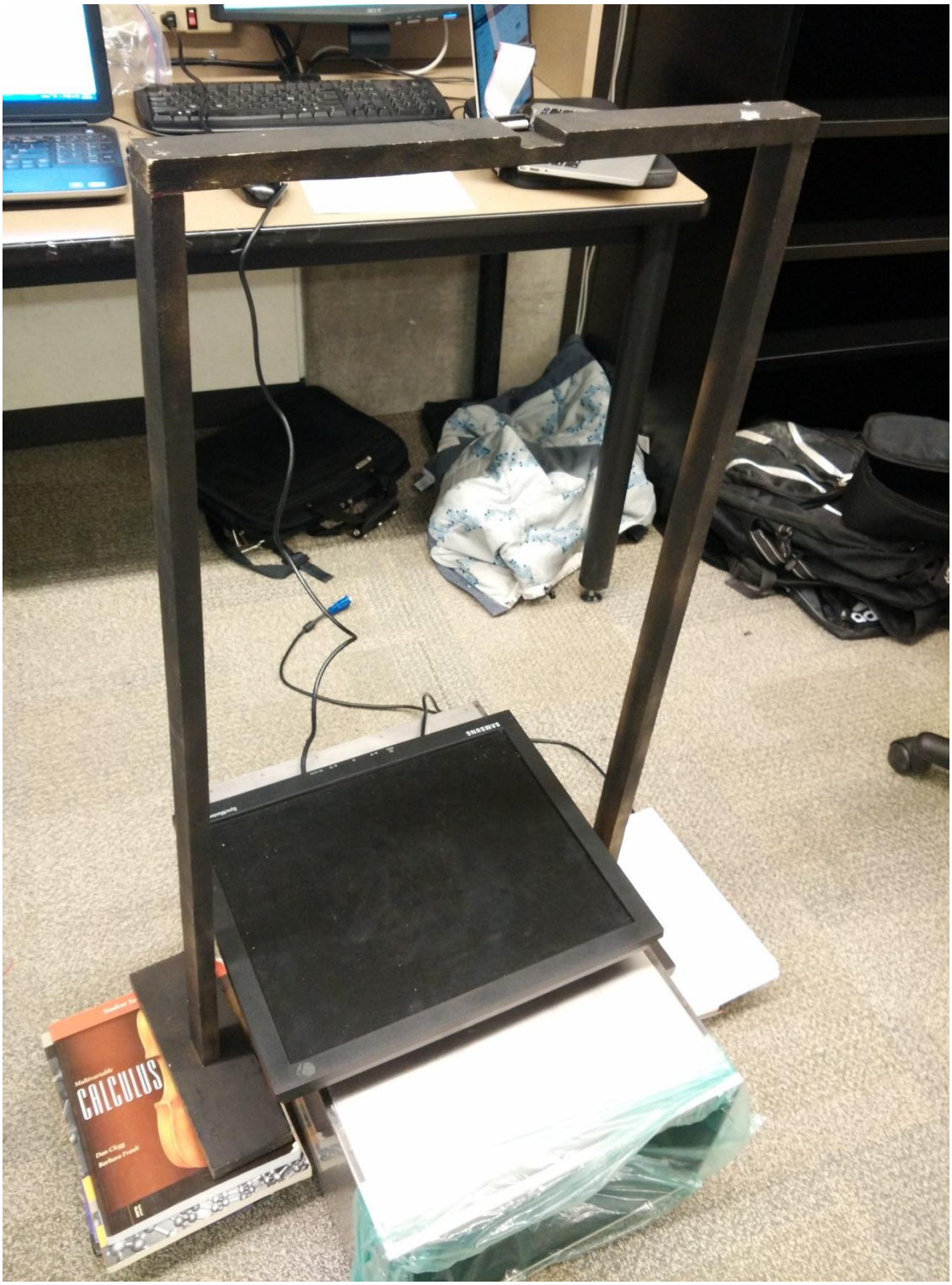

Figure 2 - Photograph of the frame. The camera is held on the frame, looking down.

Figure 3 - View of the game screen during play.

4.1.2 LED Cue Stick

Serving as the pool cue, a red LED was attached to the end of a wooden dowel. The red light will be held close to the monitor in from of the camera. The light is read from the position locator module as where the end of the cue is.

The following components were used:

- A red LED (nominal voltage = 2.2V, nominal current = 20mA)
- A 100Ω resistor
- Two AA batteries (producing a total of 3V)
- A battery holder with an on-off switch
- A piece of wooden dowel
- wires

A photograph of the stick is shown in Figure 4.

Figure 4 - Photograph of the LED wand, used for the cue.

4.2 Input Hardware

4.2.1 Camera

The camera used was the OV7670, produced by OmniVision. The camera was attached was attached via a ribbon cable to pmod terminal on the Artix-7 board.

4.2.2 532_pmod_camera_dist

This module is in charge of configuring the camera using the I2C interface to film 640x480 pixels using RGB444, and sending the pixel data in a stream to the position locator. It is primarily based off the sources downloaded in 532_pmod_camera_dist.zip from the ECE532 Piazza site.

This module was modified so that the output is provided in a stream. The original module stored the pixel data in block ram and sent it to a monitor using a VGA connection. The output VGA controller was removed, as the monitor display will be showing the pool game, not what the camera sees.

The block ram was also removed and replaced with control signals to drive a FIFO buffer. Each new pixel was written into the FIFO as it arrived. A special code (0x10000) was also written into the FIFO whenever the camera raised its vsync signal. This value is meant to signal to the position locator when last pixel in a frame is read.

4.2.3 FIFO

This FIFO is used to buffer the pixel stream coming from the camera to the position locator module. Version 12.0 of the Xilinx FIFO Generator IP included in the Vivado libraries was used.

The FIFO was set up to have separate write and read clocks. The signal came from the 532_pmod_camera_dist module, using the 25MHz pixel clock generated by the camera. The pixels were read from the position locator module, using the 100MHz system clock. Therefore, the FIFO also serves to separate the two clock domains.

The FIFO was implemented using distributed RAM and has a word size of 17 bits. The lower sixteen bits are used to represent the RGB pixel values. The highest bit is used to code the vsync signalling value. The FIFO is 512 words deep, the default for the FIFO generator. Since the system clock used to read is faster than the pixel clock used to write, and the reading position locator module is idle for at most around 30 clock cycles, only about 8 words would have been necessary. However, the FIFO still only used one block ram resource, and so the change was not deemed necessary.

4.2.4 Position locator

This module was custom written by the team for this system.

Its purpose is to process the stream of pixel values from the camera, read out of the FIFO, and determine where the red LED on the end of the cue is on the screen. This is achieved using an RGB colour threshold to determine which pixels are red. The average of all x and y positions are taken to find the centre of the red light.

The position locator can be enabled or disabled by the players by toggling SW15 on the FPGA board. When the position locator is disabled, it still reads the pixel values from the FIFO to prevent overflow, but the module will signal to the processor that there is no new valid data.

The states used in this module are described below:

- WAITING
    - The module checks the FIFO empty signal, waiting for new pixel data to arrive
    - If the empty signal becomes low, there is a new pixel and the next state is COMPUTING. If there are still no new pixels in the FIFO, the state remain WAITING.
- COMPUTING
    - The module checks most recently read pixel.
        - If the pixel is actual pixel data, then it checks if the red, blue, and green components are all within between the minimum and maximum values allowed. If they are, then the pixel is determined to be red. The current x and y positions of the pixel are added to the current accumulated totals.

After this step is done, then the current x position is updated for the next pixel. If this exceeds the length of the row, then the current y position is incremented.
        - If the pixel is actually the vsync code, then the module knows that this is the end of the frame.
    - If the vsync code was received, then the next state is CREATING OUTPUT. If there are still more pixels left in the frame, then the next state is COMPUTING again if there FIFO has more pixels to read, or WAITING if the pixel FIFO is empty.
- CREATING OUTPUT
    - During this state, the accumulated x and y positions are divided by the number of points counted to determine the average position. The divider has a latency of 26 clock cycles.
    - Once the division operation is complete, the state is changed to OUTPUT DONE.
- OUTPUT DONE
    - At this stage, the division is complete and the average position is available. The value is copied to the AXI slave register so that it can be read by the processor. In the corner case of no valid pixels being counted, an error code of 0xffffffff is written to the slave register.
    - After one clock cycle, the state is changed to RESETING.
- RESETING
    - The current x position, current y position, accumulated x position, accumulated y position, and valid pixels counted are all reset to 0.
    - After one clock cycle, the state returns to WAITING.

A block diagram of the data path is shown in Figure 5.

The integer dividers used in the module were created using the Xilinx Divider Generator IPs from the Vivado libraries. Version 5.1 of the IP was used. The dividers were set to divide 32-bit numbers by 24 bit numbers. The high radix algorithm option was chosen, due to the relatively large input numbers. This algorithm implements the divider by computing the reciprocal of the divisor and then multiplying using DSP slices to reduce latency. The dividers have a latency of 26 clock cycles.

The dividers can be configured to provide a fractional result from the integer division. A 5-bit fractional component was added to both the x and y positions. This was added to give extra precision to the physics software. The physics software uses the same fixed point fraction scheme (but with a different width of 7 bits), and so the conversion the processor does when reading the position values is simple bit shift.

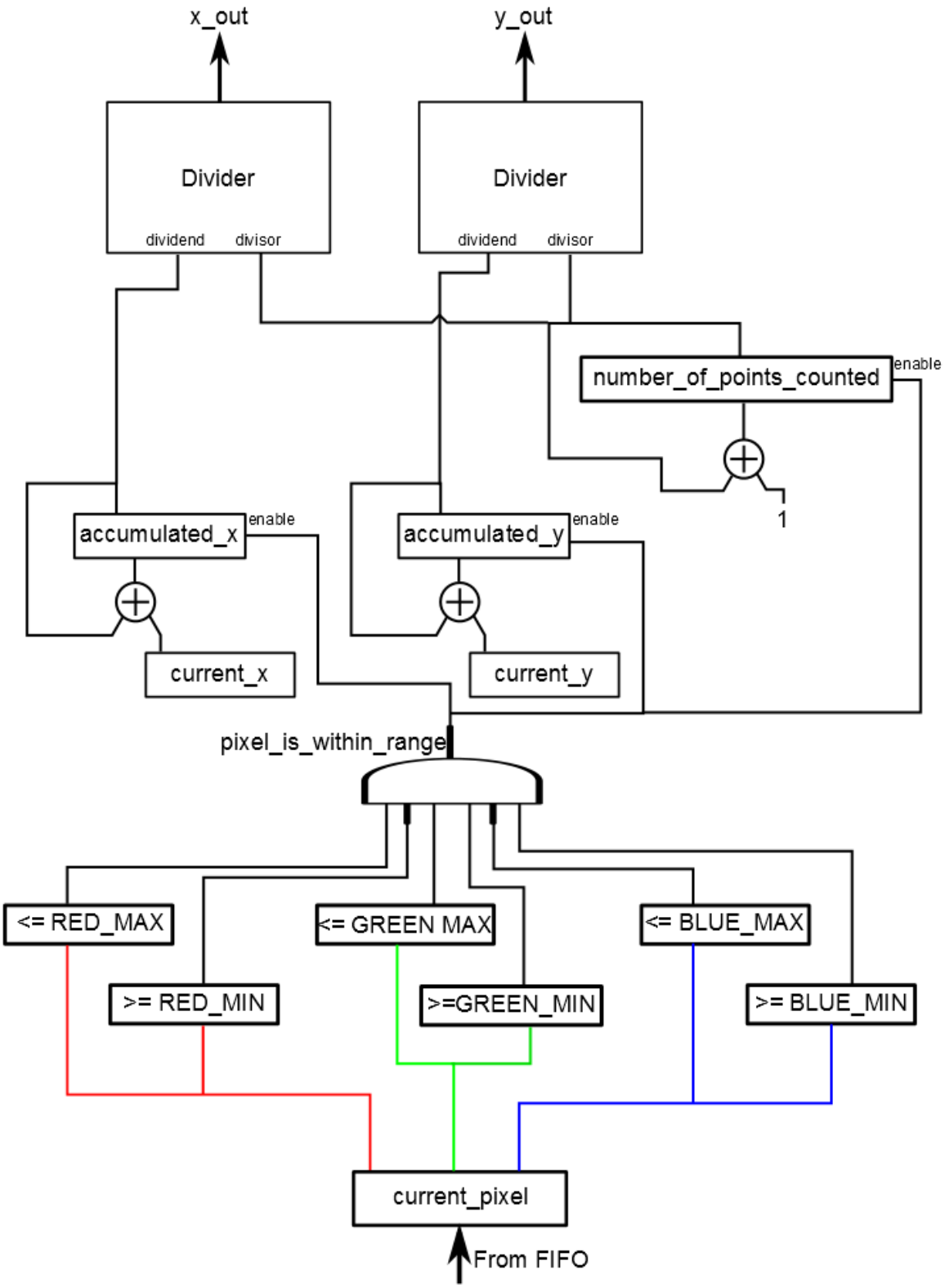

Figure 5 - Main datapath of position locator module.

The module is accessed as a slave by the microblaze processor using the AXI4 lite protocol. Two 32-bit registers are used.

- OUTPUT (address offset 0x0)
    - This read only register is polled to get the x and y positions of the cue LED.
        - Bits 15-0: Contains the x position. The lower 5 bits are used as fixed point decimal places. The uppers bits are the integer component of the position. The maximum position is 640.
        - Bits 30-16: Contains the y position. The lower 5 bits are used as fixed point decimal places. The uppers bits are the integer component of the position. The maximum position is 480.
        - Bit 31: A bit that indicates when new data is available. When the position locator writes a new position and SW15 is set to enable the position locator, this bit is set high. When the processor reads the register, the bit is cleared. This prevents the software from using the same value twice in a row.
- CONFIGURATION (address offset 0x4)
    - The processor writes to this register in order to configure the thresholds the position locator uses to determine what a red pixel is. This register only needs to be written to once at the beginning of the program, but can be changed on the fly if needed. A pixel is only counted if the pixel value is between these bound for all three colours.
        - Bits 3-0: Minimum blue value
        - Bits 7-4: Maximum blue value
        - Bits 11-8: Minimum green value
        - Bits 15-12: Maximum green value
        - Bits 19-16: Minimum red value
        - Bits 23-20: Maximum red value
        - Bits 31-24: Unused.

4.3 Software

4.3.1 Pool Cue Software Interface

The software components that managed the cue had two responsibilities. It polls the position locator to get where the cue is and gives the cue ball its initial speed once the cue collides with it. The first responsibility is contained within the source file “position_locator.cpp”. The second responsibility is contained within the “cue*” functions within “physics.cpp”.

All of the functions are accessed by the rest of the program through the function “cuePollPosition”. When this function is invoked, the following actions occur.

1. The OUTPUT register of the position locator is continuously polled until a high bit 31 is read, indicating that there is new data. (At this state in the game, all the balls are stationary, and no other action needs to be taken. Therefore, the polling is not wasting any processors cycles.)
2. The x and y position are separated out from the register value.

3. The x and y positions are checked to see if they are over the table. If the LED was not within the bound of the table, the program returns to step 1.
4. The x and y positions are rescaled so that (0, 0) is in the top left corner of the monitor and (640, 480) is in the bottom right corner.
5. The position is added to an array of previously read positions. 5 positions are stored. If there are more than 5, then the oldest value is pushed off. This array is used to compute the instantaneous speed of the cue. It is averaged over a few positions so that jitter from the person holding the cue can be reduced.
6. The most recent position is checked to see if it overlaps with the cue ball's position. If it does, and if 5 previous positions have been stored, then continue to step 7. Otherwise, return to step 1 to get a new position.
7. Take the average position change of the cue over the 5 cycles to compute the instantaneous velocity. This velocity is then directly applied to the cue ball.

After these steps are done, the function returns and the physics simulation can begin with only the cue ball moving. During the physics simulation, the position locator is never checked, so players cannot use the cue until all the balls stop moving and the next turn begins.

4.3.2 Physics and Game Library

In order to simulate the game of billiards, a software based (C++) physics engine was created. This engine runs on the Microblaze and maintains the information of the entire state of the game, including the player's turn, the number of balls in play, the position and velocities of each ball. A flow chart of the engine's operation can be seen in Figure 6.

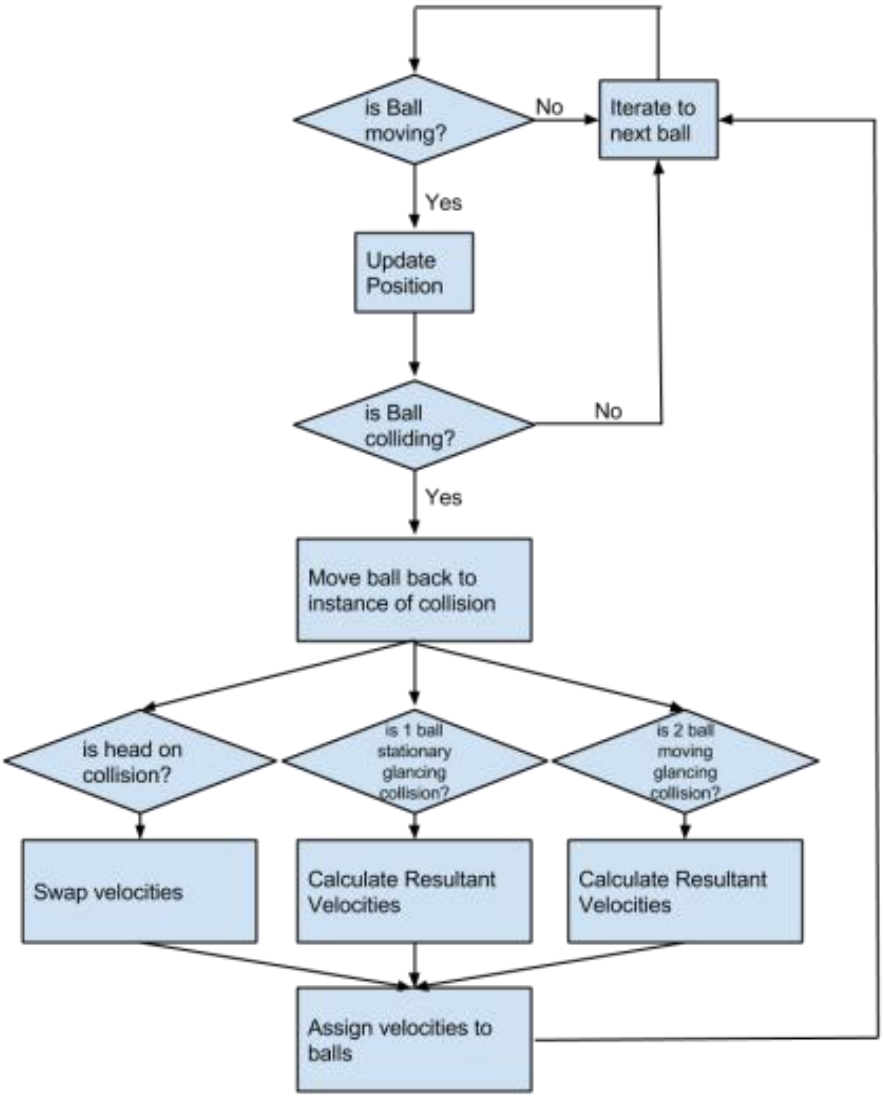

Figure 6 - Flow chart of physics simulation software.

4.3.2.1 Ball Object

Each pool ball is maintained as its own object in C++. Each ball contains the elements:

```
  // Physics properties
int   pos_x;
int   pos_y;
int   speed_x;
int   speed_y;
bool  exist        = true;
int   id;

  // Collision Handling
int   t          = -1;
int   col_id        = 0;  // Each bit's index represents a ball
int   prev_col_id   = 0;  // buffer previous frame's collisions

  // Multi-collisions
int   col_speed_x   = 0;
int   col_speed_y   = 0;
int   num_cols      = 0;  // use this to divide net momentum

int   friction_count= 0;  // used to apply friction
```

This includes a record of the ball's current position and velocity in Cartesian coordinates, as well as a unique identifier and a flag for whether the ball is currently in play or not. Additional fields are added to allow per-frame multi-ball collisions to be computed independently.

4.3.2.2 Trigonometric Library

For 2-D elastic collisions, the resultant velocity of a ball in a two-ball collision is described by:

$$v'_{1x} = \frac{v_1 \cos(\theta_1 - \varphi)(m_1 - m_2) + 2m_2 v_2 \cos(\theta_2 - \varphi)}{m_1 + m_2} \cos(\varphi) + v_1 \sin(\theta_1 - \varphi) \cos(\varphi + \frac{\pi}{2})$$

$$v'_{1y} = \frac{v_1 \cos(\theta_1 - \varphi)(m_1 - m_2) + 2m_2 v_2 \cos(\theta_2 - \varphi)}{m_1 + m_2} \sin(\varphi) + v_1 \sin(\theta_1 - \varphi) \sin(\varphi + \frac{\pi}{2})$$

To allow the use of sine and cosine, as well as determining the angle itself, a some trigonometric library is required. Trigonometric conversions are stored in a hash table, or in C++11, stored as a map. This can be found in `angles.hpp`. The arctan hash table is accessed via the closest multiple of 64 when checking the ratio of the rise over the run. The sine and cosine tables are created using only the angles supplied by the arctan hash. Angles beyond the range of 0-90 degrees are normalized to that region using trigonometric identities.

4.3.2.3 Collision Library

The collision library checks the position of two balls and determines if they are overlapped or not. If the balls are overlapped, the `imminentCollision` function determines at what point between the current frame and the last frame the two balls would have first collided. This yields a more accurate collision angle calculation, and thus a more accurate resulting collision.

The `collisionForce` function manages the calculation of the resultant velocities. This function gets the angle of the collision (and each balls velocity vectors) and determines the class of collision it belongs to. Then, the appropriate equations are evaluated to find the resultant velocity. Multiple simultaneous collisions are treated the same (sequentially), but the effective velocity is scaled in the interest of conservation of momentum.

4.3.2.4 Game State Object

The game state class was created as an object that always contained the snapshot of the current state of the entire program. The class definition is as follows:

```
  // Ball States
bool  done  = false;         // if game complete
int   num_balls;
Ball *ball[NUM_BALLS];       // List of all the balls
int   mov_ids[NUM_BALLS];    // List of moving balls
int   num_movs = 0;

// Player States
int   turn_id = 0;

// Cue stick states
int   cue_pos_x, cue_pos_y;
bool  cue_down = false;
int   cue_colour = 4;

// Turn Status
bool  scratch = false;
bool  scored = false;
bool cue_first_hit = false;
bool can_hit_eight_ball[2] = {false, false};
```

This class contains the pointers to all the balls in the game, and thus allows the state of the game to be easily passed between functions and modules. This is particularly useful for the TFT controller, where only one parameter needs to be passed to the drawing library.

4.3.2.5 Scratching

The scratch rule implemented is similar to the one found in real life eight-ball billiards with the exception that a scratch does not occue if the cue ball does not manage to hit any ball. A scratch occurs when the player fails to sink a ball of their respective colour, if the first hit of the cue is not their own ball or if they sink a ball of the other player or the eight ball before sinking all their

own balls. All scratches will result in a switching of the turn and the latter two scenarios will results in a replacing the cue ball. The game state variables cue_first_hit and can_hit_eight_ball are used to determine when a scratch has occurred while the game state variables scratch and scored are used to record scratches and ball sunk.

4.3.3 Drawing Library

4.3.3.1 Drawing

The DrawState function is the master function with draws the game state from the information it has obtained from the GameState object in physics.cpp. The DrawState function is called after every cycle of the simulation and draws all 16 balls is they exist as well as the 6 pockets. The three main draw functions DrawState utilizes are the DrawRectangle, DrawCircle and DrawNumber functions. The DrawRectangle function is a simple function which draws a filled rectangle of given length and height from a given starting point. The DrawNumber function draws a given number at a given x and y location using a similar algorithm as a Hex display. The DrawCircle function is a more involved function which draws a filled circle of a given radius at a given center point by using the midpoint circle algorithm. Along with drawing the game state, the DrawScore function inside the DrawState function indicates whose turn it is by drawing a small ball beside their player number with the colour of the ball indicating whether the play is hitting the cue ball, placing the cue ball after a scratch or is currently able to sink the eight ball. The drawBackground function is separate from the DrawState function as it is only called at the beginning of the game to create the green background for the game. It uses only the DrawRectangle function.

Lastly, when the game is won when the eight ball is sunk, the screen will appear all blue or cyan depending on which player has won the game.

4.3.3.2 Double buffering

The implementation for double buffering is also within the DrawState function where the values of memptr and memptr_alt are swapped after each frame with memptr indicating where the microblaze is writing to and memptr_alt being the location where the TFT is reading from. After swaping the memory pointers, it waits for the TFT to finish drawing the frame by pulling on bit zero of its status register to be high before existing the DrawState function.

4.3.3.3 Erasing

In order to achieve a high frame rate, we could not simply redraw the entire screen for each frame. Instead we needed to keep the background intact and erase the previous position of the ball before drawing the new one. This erasing is implemented in the DrawState function where tft_array_x and tft_array_y contain information on the previous position of the ball. Due to double buffering, the arrays needed to contain information about the position of the balls from the two previous frames. The erasing is done by a drawing green ball of the same size at the previous position of the ball making it the same colour as the background before drawing the ball's new position.

4.4 Output Hardware

4.4.1 TFT controller

The TFT controller reads in data starting from the memory location pointed to by tftptr[0], interprets its RGB value and displays this information on the monitor. The init_tft function sets the memory location of tftptr[0] and starts the TFT by writing a one to the TFT Display Enable bit.

5 Description of Design Tree

The project was submitted to GitHub. The URL is
https://github.com/davidmctavish/G6_FPGA_Billiards

The key directories are described below.

- docs: Contain the proposal and Final report for this group
- src:
    - Hardware:
        - PositionLocatorTest_2014_1: This hardware system is used to test and confiure the camera and position locator. The seven segment displays on the Nexys-4-DDR board can be used to test the configuration of the position locator.
        - project_1: The main project. Contains all the hardware needed to run the billiards game.
        - VivadoIP: Contains the IP modules, imported by the other projects
            - axi_to_7segDisplay_1.0: Used in the test hardware to output values to the seven segment display
            - pmod_input{2}_1.0: Configures the camera module and reads the pixel values. Version 1 reads 320x240 pixel frames, version 2 uses 640x480.
            - poisition_locator_1.0: Custom IP for locating the pool cue on screen.
    - PhysicsSoftware: All of the source files for the program to run on the processor.