Break Out: Implementing Old School Games on FPGAs

Brian Nakayama
Department of Computer Science
Iowa State University
Ames, Iowa 50011

Mahieddine Dellabani
Ensimag, School of Computer Engineering
Grenoble INP
Grenoble, France

***Abstract*—Implementing *Breakout* on an FPGA requires design elements found commonly in integrated circuits such as finite state machines and interesting examples of dataflow. As mentioned in the proposal, *Breakout* was originally implemented completely in digital hardware (with no processor or program). In our implementation, we broke down the game into three main modules: the keyboard, the game logic, and the VGA/screen. We further divided these modules into finer grained modules for both modularity and clarity. Though we have not completed an online tutorial for *Breakout* (as we focused on in our proposal), this document which specifies how our implementation works contains the beginnings of an eventually completed online lesson.**

I. Introduction

We have divided our guide to implementation into three sections comprising of the keyboard controller, the game logic, and the VGA controller. Following our implementation we have provided a list of open issues and lessons learned from our implementation of *Breakout*, as well as possible plans for the future. Figure 1 has the block diagram for our implementation of *Breakout* and should be referred to in order to understand the interactions between the different components.

II. Keyboard-Controller

A. Inputs and Outputs

The inputs for the controller are:

- PS2D
- PS2C

The outputs for the controller are:

- scan_ready
- scan_code

B. The Keyboard

For the communication between the keyboard and the FPGA board, the keyboard module (Kbd.vhd) implements a Device to host communication. Data sent from the keyboard has a 11-bits format: a start bit, 7 bits of data, a parity bit, and a stop bit.

The start and stop bits are always equal to 0 and 1 respectively and the parity bit corresponds to an odd parity. It is set if and only if there is an even number of one's in the n-bit data, otherwise it is cleared. This is used for error checking, so when an error occurs the bit packet is withdrawn.

The 8-bit data corresponds to either the scan code of the key pressed, the key up code (x"F0"), or the extend scan code for the extended keys (x"E0"). (See more about it in the controller section.) The keyboard writes the data bit per bit on the data line (PS2D) when clock (PS2C) is high, so they can be read by the host when the clock is low. During the sampling, the 8 bits data are gathered into a 1-byte vector using the following shifting operation:

1. scan_code_reg (6 downto 0) <= scan_code_reg (7 downto 1);
2. scan_code_reg (7) <= keyboard_data;

The new incoming keyboard_data bit is always received in the 7^{th} index after shifting the previous bits to the right. When the data is ready, the signal scan_ready is set to one for one clock cycle notifying the controller. Figure 2 (b) shows how the scan_ready is either set or cleared. Figure 2 (a) displays a high level view of the device to host communication.

C. The Controller

The controller takes the inputs:

- scan_ready
- scan_code

The controller then creates the outputs:

- control_en
- control_mode
- control_signal

The keyboard uses scan codes to communicate the key-pressed data. Each key has a particular scan code (1 byte). If a key is pressed, its scan code is sent and if it's held down, the scan code will be sent repeatedly once every 100ms(refer to the data sheet in the proposal). When released, the keyboard sends the key-up code (x"F0") followed by the scan code of the released key.

However, there are special keys in the keyboard, where the byte x"E0" is sent first ahead of the scan code and when pressed repeatedly the bytes x"E0" and the corresponding scan code are sent. The key-up bytes for those keys are x"E0_F0".

The controller take in consideration the following keys :

- escape: to end the game
- space: to pause the game

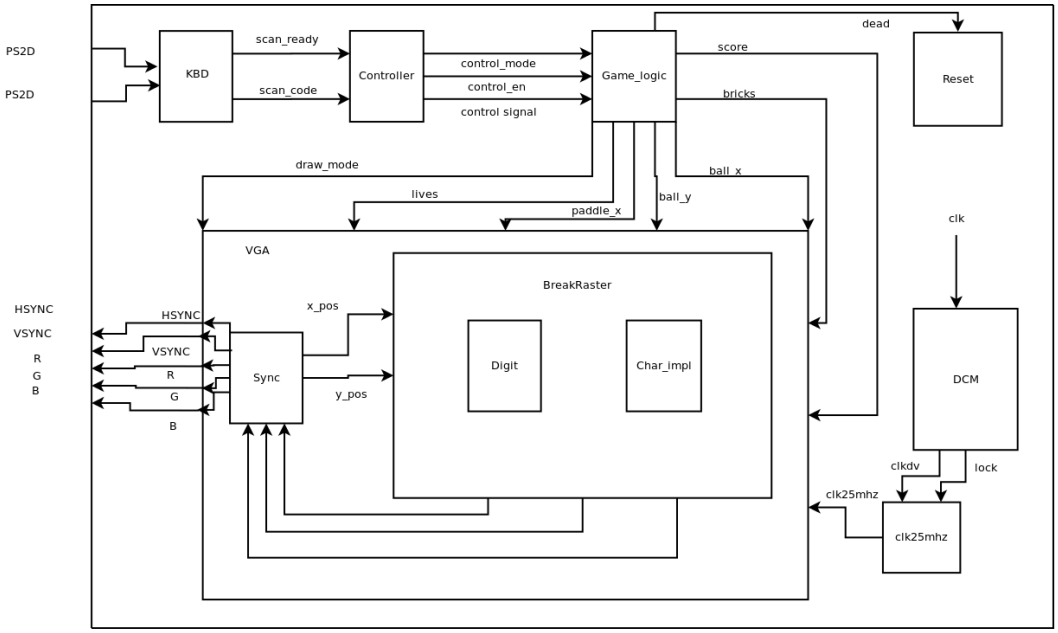

Fig. 1. The block diagram of the top module

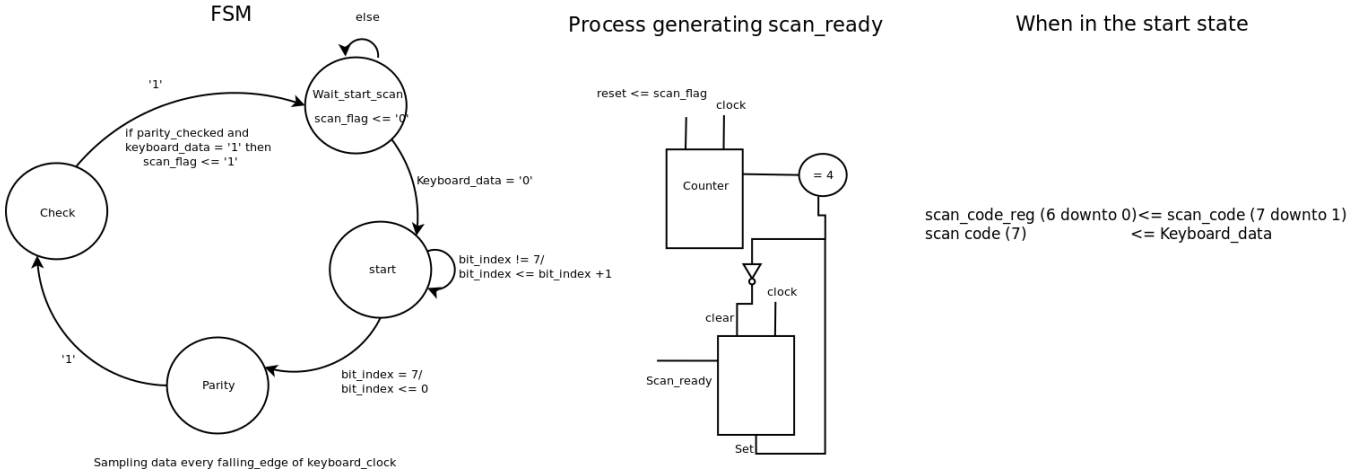

Fig. 2. The FSM and logic diagram for the keyboard

- enter: to launch the game
- the arrows (special keys): to control the paddle

The controller is implemented as a FSM (see Figure 3). It separates the received scan codes into two categories:

- 1-control codes : These represent the arrows' scan code, so when received the variable control_en will be set and the control_signal will have one of the following values:
  - go_left: when the left arrow is pressed
  - go_right: when the right arrow is pressed
- 2-mode codes: These refer to the control keys (space, launch, enter, escape). When received the signal control_mode is set and the control signal will have on of the following values:
  - end: when escape is pressed
  - pause: when space is pressed
  - launch: when enter is pressed

The implementation was spread into these two categories since the usage of the keys differs in order to have a smoother move of the paddle.

III. Video Gate Array

A. Inputs and Outputs

For communication with the VGA and rastering of the image, the top level VGA module required a 25mhz clock. In our design we included a wire for reset; however, one can raster images to the screen using almost exclusive dataflow, thus despite having a reset signal we did not use the reset wire for the VGA. Other than the clock and reset signals, all other input signals represented objects in the game. These signals

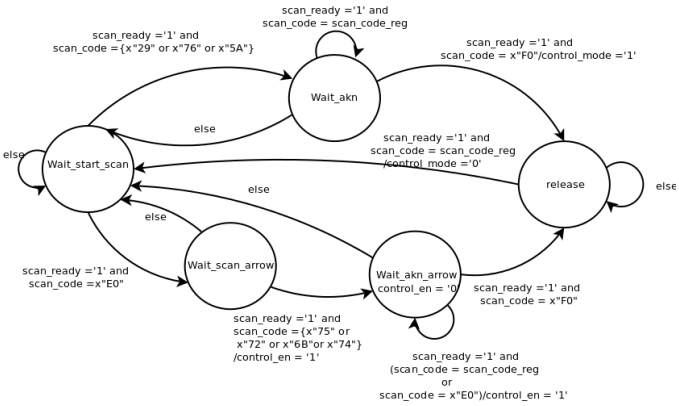

Fig. 3. The FSM for the controller

included the horizontal position of the paddle(12 bits), the horizontal(12 bits) and vertical(12 bits) position of the ball, the 128 bit array of bricks, the number of lives (4bits), the score(12 bits), and the four bit vector draw mode. All logic vectors have a width with a factor of four as a rule of thumb. This leads to a few unnecessary bits in each vector; however, keeping each vector a multiple of 4 increases the simplicity of the design with no measurable impact on performance.

The important outputs included the outgoing red, blue, and green signals(4 bits each) as well as the horizontal and vertical syncs(1 bit each) to control the VGA monitor. The Nexys2 board used for this project only required 8 bits of color information, thus the last bits of the red and green vectors as well as the last two bits of the blue vector were not used. See the top level design in Figure 1.

B. BreakRaster

The BreakRaster module within the VGA controller held all of the logic specific to drawing the pixels of the game. Since BreakRaster only drew pixels, the logic of the module is best thought of as a function $f : x, y, paddle\_x, ball\_x, ball\_y, bricks, score, lives, draw\_mode \rightarrow R, G, B$. Using *if* statements BreakRaster draws the screen (see Figure 7) according to the specifications shown in Figure 6. Some of the variables for rastering were parameterized for easy editing and good coding practices. Other elements in BreakRaster were fixed since they relied on tricks to be rendered. the logical vectors x and y are provided by the Sync discussed in detail below.

The digits and bricks used convenient size factors to simplify the logic behind rendering. Below is some example code for each:

Digits:

```
1.if(y <SCREEN_Y_BEGIN) then
2.   if(x < 64) then
3.        number <= score(11 downto 8);
4.        R <= rSymbol;
5.        G <= gSymbol;
6.        B <= bSymbol;
7.   elsif (x < 128) then
8.        number <= score(7 downto 4);
9.        R <= rSymbol;
10.       G <= gSymbol;
11.       B <= bSymbol;
12.          ...
13.xSymbol <= std_logic_vector(x(5 downto 3));
14.ySymbol <= std_logic_vector(y(5 downto 3));
```

Bricks:

```
15.elsif((SCREEN_BRICK_BEGIN <= y) and
         (y < SCREEN_BRICK_END)) then
16.    vx := std_logic_vector(x - SCREEN_X_BEGIN);
17.    vy := std_logic_vector(y - SCREEN_BRICK_BEGIN);
18.    vx := "00000" & vx(11 downto 5);
19.    vy := "000" & vy(11 downto 3);
20.    if(bricks(to_integer(unsigned(vy) *18+
             unsigned(vx))) = '1') then
21.       ... Set R,G,B to a colored brick pixel
```

For the digit code, the digit module in Figure 4 takes xSymbol and ySymbol vectors as an input and outputs rSymbol, gSymbol, and bSymbol vectors. Thus when we have a pixel in the area of a digit we set the outgoing R, G, and B signals to the rSymbol, gSymbol, and bSymbol vectors (lines 4-6 and 9-11). Notice that xSymbol and ySymbol (lines 13 and 14) take bits from the middle of the x and y vectors, dividing the coordinate space by 8 mod 8. This makes 8×8 pixels the same color for a given digit pixel, creating a retro-looking number on the screen.

The bricks used a similar trick for rendering. If the x and y coordinate is within the brick range (line 15), we divide by shifting the x and y coordinate (8 for height and 32 for width, lines 18 and 19). Then by multiplying the y value by the width of the bricks (see Figure 6), we can index the bricks bit array to see if the brick exists, 1, or is gone, 0. To color the bricks we use a multiplexer on the vy value to determine what line a brick is on.

C. Sync

The Sync is a general purpose module for rastering pixels on a 640×480 pixel screen at 25 mhz. Figure 5 shows the counters (A.) required to implement the VGA protocol. These counters are the only source of synchronous, behavioral logic relying on memory. Each indexes the screen along the horizontal axis (x) or the vertical axis (y).

The rest of the logic found within the sync relies on dataflow to implement the protocol for a 640×480 pixel screen. Note that a DCM created by the ISE Xilinx Wizard was required to generate a consistent 25 mhz clock from the board's default 50mhz to drive the index of the screen. Given these conditions, the Sync compares values from the counters with the numbers for the front porch, sync pulse, and back porch. Before the rendering of the screen's pixels, the VGA output toggles the sync pulse to off in the middle of the front and back porches.

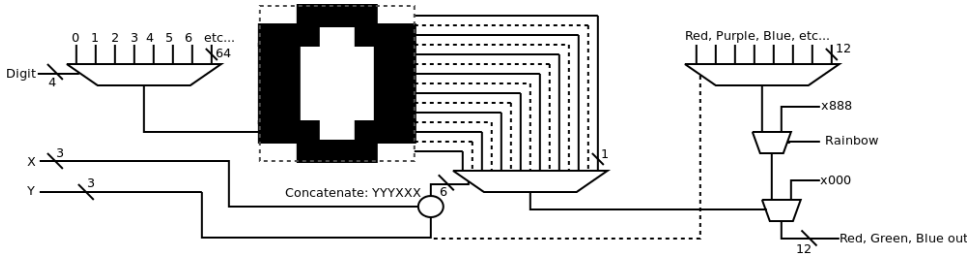

Fig. 4. The dataflow for a pixel of a digit

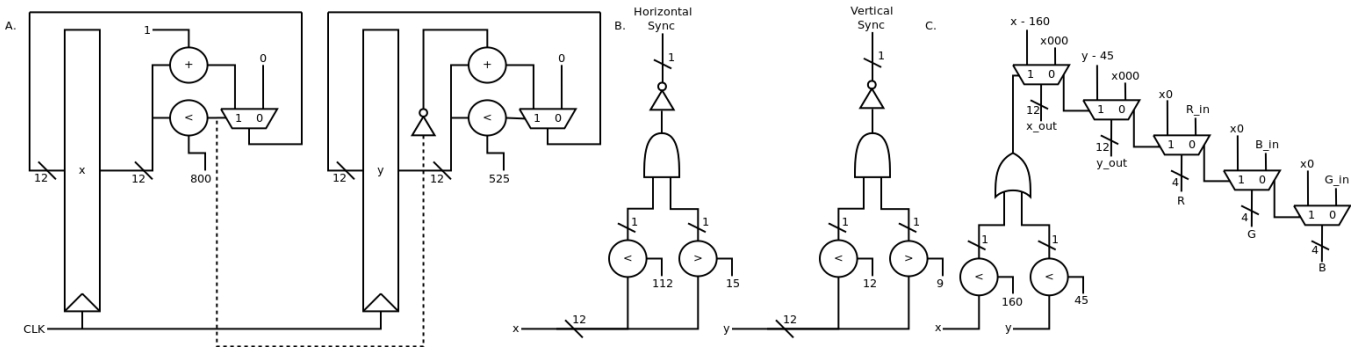

Fig. 5. The sync counters and corresponding dataflow logic

During the front porch, sync pulse, and back porch we keep the R, G and B signals low. After the back porch, we can send the x and y signals out and forward any received R, G, and B colors to the VGA output. Since this module only send out x, y coordinates, and colors without any rendering, it can be used in tandem with any hardware component. The x and y coordinates are hooked up to the x and y inputs of BreakRaster, while the R, G, and B outputs are forwarded from the R, G, and B outputs of BreakRaster. If we needed to make a new game, we would likely have to switch out BreakRaster; however, the Sync module would stay exactly the same.

IV. Game Logic

A. Inputs and Outputs

Much of the game logic depends on the specifications for the screen in Figure 6 examined in the previous section. We parameterized collisions with bricks, the paddle, and the size of the ball using these specification for ease of editing later in the process. The game logic module required signals for the clock and reset as well as signals from the keyboard controller: control_en, control_mode, control_signal.

All of the outputs feed into the VGA controller: paddle_x, ball_x, ball_y, bricks, lives, score, dead, and draw_mode. The dead signal was used in

In addition to the input and output signals, game logic featured several internal registers including:

- ball_x_dir: The horizontal direction of the ball, ‘1’ for right and ‘0’ for left.
- ball_y_dir: The vertical direction of the ball, ‘1’ for up and ‘0’ for down.

Fig. 7. The screen as implemented on the FPGA

- paddle_x_dir: The direction of the paddle, ‘1’ for right and ‘0’ for left.
- paddle_moving: An enable signal for whether the paddle should move.
- angle_reg: A register for the angle of the ball (i.e. low, med, and hi).
- speed_reg: A register for the speed of the ball (e.g. slow, normal, fast, etc.)
- clk50MhzCounter: A counter for the number of 50mhz cycles from the input clock.
- clk50hz: A “slow” clock for the updates of the game logic.

Figure 7 shows how the screen looks when rendered on the screen.

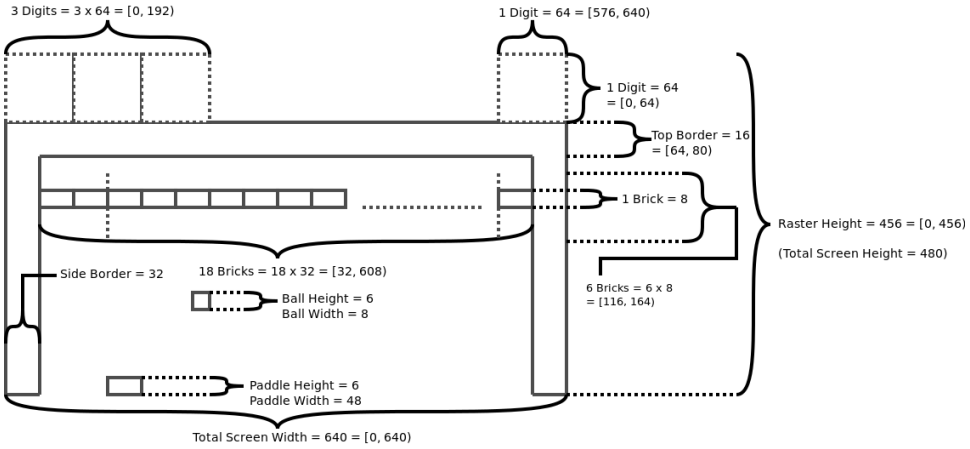

Fig. 6. The specifications for the screen

B. Speed, Lives, and Score

For the speed when the ball hits a brick, depending on the layer position of that brick, the ball will change its speed from slow to fastest. Here are the different layers with their corresponding speed:

- purple : slow
- blue : normal
- green : fast
- yellow : faster
- orange: fastest
- red: fastest

Each player has 3 lives, so each time the paddle misses the ball the lives are reduced by one and the game is restarted where it was left. If there is no lives left, the next miss will set a dead signal that will restart all the modules of the game. This is the score corresponding to that section:

```
1. if lives_reg > x“0” then
2.    lives_reg <= std_logic_vector(unsigned(lives_reg)-1);
3.    restart <= ‘1’;
4. else
5.    dead_reg <= ‘1’;
6. end if;
```

The lives also affect the draw mode. The draw mode is a signal that depending on the mode keys pressed tells the VGA which output mode it should display the images in: Begin, Play, Pause or Game_over. The FSM in Figure 9 reflects the behavior of this signal.

- Begin: displays a banner of Breakout
- Play: displays the game state (paddle, ball and bricks)
- Pause: displays a pause banner
- Game_over : displays a Game_over banner

Finally, each time the ball hits a brick, the score is incremented by one. The score output is a std_logic_vector from 11 downto 0. Each 4 bits represents a number using bits 0 through 9 (see Figure 8 (a)). This implementation was chosen for a matter of simplicity for the BreakRaster. If one of the

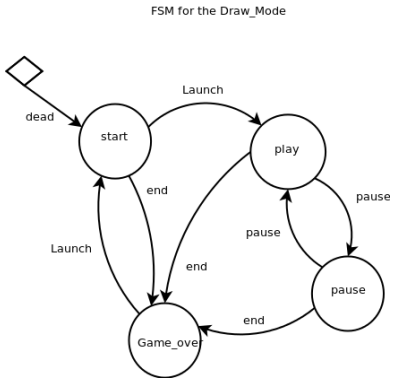

Fig. 9. The FSM for the draw mode

three numbers from 11 downto 8, 7 downto 4, or 3 downto 0 equals 9, it is cleared and the next number is incremented (see Figure 8 (b)). The following code shows this behavior:

```
1. if score_reg (3 downto 0) = x“9” then
2.    score_reg (3 downto 0) <= x“0”;
3.    if score_reg (7 downto 4) = x“9” then
4.       score_reg (7 downto 4) <= x“0”;
5.       score_reg (11 downto 8) <= std_logic_vector(
            unsigned(score_reg (11 downto 8)) + 1);
6.    else
7.       score_reg (7 downto 4) <= std_logic_vector(
            unsigned(score_reg (7 downto 4)) + 1);
8.    end if;
9. else
10.   score_reg (3 downto 0) <= std_logic_vector(
            unsigned(score_reg (3 downto 0)) + 1);
11. end if;
```

C. Collisions

Collisions between the ball with the side walls and the paddle were done using conditional comparators such as <

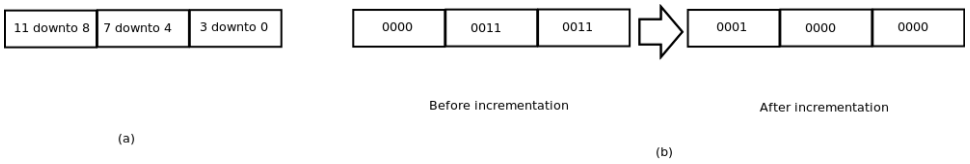

Fig. 8. The score representation for *Breakout*

or >=. Based off of the specifications of the screen, the ball would check if its location was out of the drawn boundaries. If so, it would adjust its position (either the ball_x or ball_y signals) accordingly and bounce by switching its horizontal or vertical direction. In the cases where the ball left the bounds of the bottom of the screen without colliding with the paddle, the player would lose a life. Below is some of the code for the movement and the collisions with the paddle:

Ball Movement:

```
1. case speed_reg is
2.     when slow =>
3.         case angle_reg is
4.             when low =>
5.                 if (ball_x_dir = '1') then
6.                     cx := to_signed(3, 4);
7.                 else
8.                     cx := to_signed(-3, 4);
9.                 end if;
10.                if (ball_y_dir = '1') then
11.                    cy := to_signed(1, 4);
12.                else
13.                    cy := to_signed(-1, 4);
14.                end if;
               ...
```

The above code uses some intermediate variables, cx and cy, which represent the values to change the x and y positions by respectively. These variables are also used to see where the ball will be in the next frame for the collision logic.

Collision with the bottom of the screen and possibly the paddle:

```
15. case ball_y_diris
16.     when '0' =>
17.         if(ball_y_reg <= SCREEN_Y_END) then
18.             ball_y_dir <= '1';
19.             ball_y_reg <= to_unsigned(SCREEN_Y_END,
                    12);
20.         end if;
21.     when '1' =>
22.         if(ball_y_reg > (SCREEN_PADDLE_BEGIN -
                BALL_HEIGHT)) then
23.             ball_y_reg <= to_unsigned((
                    SCREEN_PADDLE_BEGIN -
                    BALL_HEIGHT), 12);
24.                 if (ball_x_reg >= paddle_x_reg - 7 and
                        ball_x_reg < paddle_x_reg + 4) then
25.                     ball_x_dir <= '0';
26.                     ball_y_dir <= '0';
27.                     angle_reg <= low;
               ...
```

The collision of the ball with the bricks used a trick similar to the one for rendering the bricks to tell when the ball *would* collide with the bricks. By checking if the ball would collide in the next frame instead of the current frame, we prevented the ball from jumping into the middle of the bricks section, circumventing many possible bugs.

V. For the Future

A. Open Issues

One issue comes from increasing the speed of the ball. At the top speed, the ball sometimes skips over bricks. In a conventional processor this problem could elegantly be solved by some code; however, the quest for an equally elegant hardware solution has so far confounded the authors of this paper. However, there may also exist many clumsier "brute force" solutions that would be easier to implement, albeit a waste of reconfigurable logic.

B. Lessons Learned

As the project winded down the quality of the code reflected the energy of the authors. Though the code works, it has a bit of a tired, lazy feel to it, lacking comments or optimal clarity. For a tutorial the code may need some refactoring. The effort required to write code presentable to a learner of VHDL easily outweighs the effort of designing a working piece of hardware. More code reviews and ongoing, consistent commenting may help for future projects.

We also picked up some tips based off of many of our debugging experiences. Synchronization is also very important when dealing with multiple technologies such as the Keyboard, the FPGA, and the VGA. It also helps to use the LEDs for debugging in addition to Modelsim to verify one's logic. Lessons learned from implementing games in software apply equally to implementing games in hardware, though the programming may seem extremely different at first. Finally, building a whole project from scratch versus adding onto an already partially completed project was a very good experience.

C. Plans for the Future

I(Brian) will probably finish a tutorial for this and post it on my web site. In addition to being a cute resume builder, thinking of this from a teaching perspective creates much needed experience for developing curriculums.

VI. Conclusion

We have accomplished what we set out to do in our proposal to a satisfactory level.