Brick Breaker Final Report

Jemale Lockett and Jonathan Abbott | 6.111 Fall 2013

1 Overview

We made an interactive game based upon the classic game brick breaker. The object of brick breaker is to break the bricks that are distributed around the top of the game screen. The bricks are broken after coming in contact with a ball that bounces around the screen. At the bottom is a paddle that in the classic game moves based on user input. The user has to make sure the ball bounces off the paddle without going off the bottom of the screen. In our implementation, instead of using arrow keys to control the paddle's position, we use a camera to track the position of an actual Ping-Pong paddle held by the user. The control of the game paddle with an actual Ping-Pong paddle improves the user experience.

2 Overall Design

We chose the game brick breaker because it is already a fun game that we knew we could improve using an actual Ping-Pong paddle. The actual paddle made the game more interactive and allowed us to take the users' instinctual movements to allow for a better user experience, in much the same way that current generation video games are employing the technology (e.g. Microsoft Kinect, Nintendo Wii, PlayStation Move).

Our game was designed to be retro and fun. The overall game experience is shown below in Figure 1. At the very bottom left there is the 6.111 labkit that is performing the logic of the game. In the center is the visual game display. To the right is the actual paddle being tracked by the camera. To the right of the camera are the speakers that provide sound during the game.

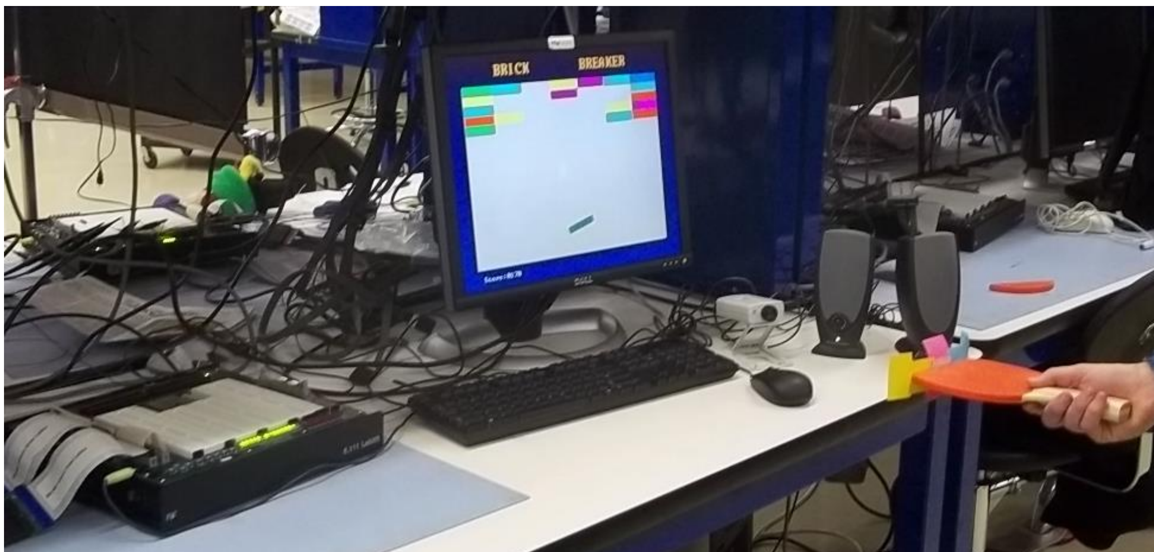

Figure 1: A view of the all essential components to the game. The game consists of a screen (center) that shows the gameplay. There is a camera pointed towards to the user who holds a paddle. To the left, there is the labkit with an FPGA that processes the game. There are also speakers here shown to the right that play background music and sound effects.

The game is great for this class because it has a modular design that can be split into discrete parts for each team member to work on. These modular parts are shown in Figure 2. These parts are the video processing, brick breaker logic, and sound output. The video processing and sound output was completed by Jemale and the brick breaker logic and visual output was completed by Jonathan. Each of these modular parts will be discussed individually.

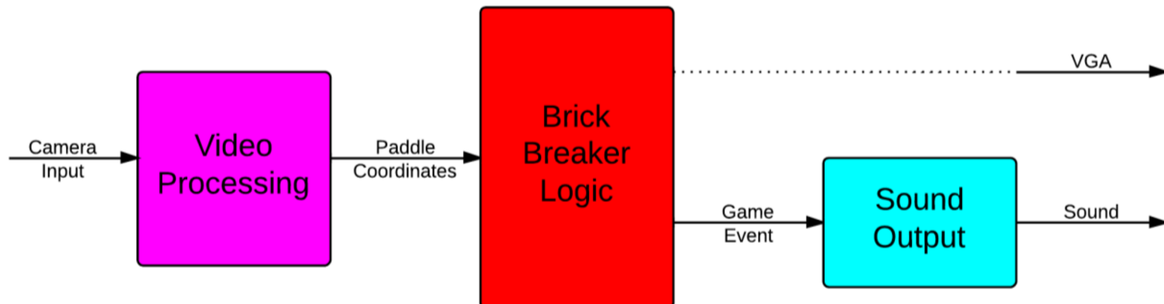

Figure 2: Block diagram of system. The video processing module takes in video and will give information about the paddle's location to the game logic. The game logic uses the paddle's location to determine what happens in the game and what should be displayed. These game events are sent to the sound output module to play the appropriate sound.

3 Camera Input Modules (Jemale)

The purpose of the camera module is to take in video input and produce output coordinates of the actual Ping-Pong paddle. "Camera Input Module" refers to the complete process of receiving a video stream and calculating paddle orientation. The module is implemented with several different subcomponents. The module takes in NTSC video data from the camera plugged into the component input jack, decodes the NTSC data, creates an RGB stream, and finally converts the RGB stream to HSV color space. Converting to HSV allows the module to better identify pixels corresponding to the paddle. These steps were mainly done through the modification of the zbt_6111 sample Verilog code provided by the staff. The module determines the central coordinates and the tilt of the paddle in the Paddle_Detect module. The module takes in an HSV pixel and determines if it matches any of the three detection colored pieces of paper that are shown attached to the paddle below in Figure 3.

Figure 3: The paddle with attached colorful pieces of paper. The red paper was used to track the center x and y position of the paddle. The blue and yellow paper was used to determine how the paddle was tilting. The center x and y position and the horizontal and vertical displacements of the "center of mass" of the blue and yellow regions were calculated and sent to the main game logic.

Two accumulators were created for each color to sum the horizontal and vertical position values that correspond to the matching pixels. There are three more accumulators which keep track of the total number of pixels that match each color. When the last camera pixel has been checked, we start each of three dividers generated using ISE's coregen that determine the center of mass of the detected pieces of paper. The tilt is computed by taking the leftmost and rightmost coordinates of the paddle, which correspond to the centers of mass for the yellow and blue pieces of paper. The range of the horizontal and vertical values of the pixels on the screen are greater than what is actually displayed as camera output, allowing time to complete the divisions before a new set of pixels starts to arrive. A block diagram of the process is shown below in Figure 4.

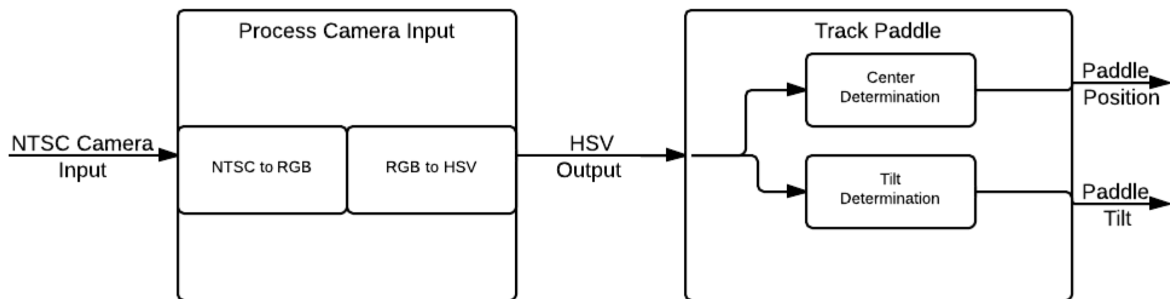

Figure 1: Video processing components. NTSC camera input is first decoded into an RGB stream. The RGB stream is converted to HSV and then sent for more processing. The paddle tracking component determines the center of mass of the paddle and its tilt.

This module was tested by displaying the calculated center of mass of the paddle on the screen against the camera's image. This is done through the use of a crosshairs module which outputs the HSV value of a particular pixel to the hex display. This allowed us to determine the necessary color range for paddle detection. In addition, when matching pixels were detected, the pixel values were changed such that they would be marked as brighter colors allowing for visual confirmation of correct detection. The output from the camera is shown in Figure 5.

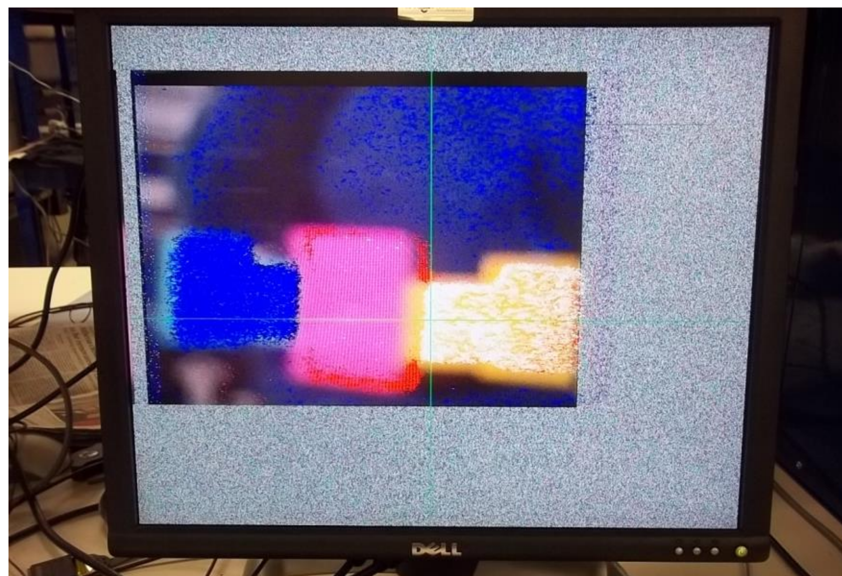

Figure 5: A view of the screen used to test and confirm paddle tracking. The camera's out-of-focus view is visible and the regions identified are lit up with bright pixels. The green crosshairs represent the estimated center of the paddle.

Figure 5 shows a transitional effect or noise that happens with the camera. To avoid the noise a simple averaging filter was added that made the gameplay much smoother.

Much of the time spent implementing the camera tracking was trying to determine the best color ranges. Determining the colors of the paddle was straightforward. However, it was also necessary to make sure that no other objects were being falsely detected; stricter bounds had to be used. Restricting the location of where the paddle would be tracked in the camera window was also tried. This improved tracking but was removed to allow the player more freedom of movement.

To increase the visible area of the paddle, Jonathan sawed off the front of the Ping-Pong paddle. This offered space to attach brightly colored paper in the future that was needed to increase trackable area and improve tracking. The paddle post-sawing is shown in Figure 6.

Figure 6: The paddle post-sawing. The paddle was cut along the front edge to provide a flat area that would be more easily be detectable.

The camera used for tracking was provided by the 6.111 staff. The camera outputs NTSC data through a composite video port. The camera focus can be adjusted to improve tracking. The camera was purposefully defocused significantly to act as a filter, or blur the image, and thereby decrease background noise. A picture of the camera is shown in Figure 7.

Figure 7: The camera used for tracking the paddle. It was helpful to make the image out of focus in order to avoid identifying the bright blue, pink, and yellow in the background.

4 Brick Breaker Logic Module (Jonathan)

The brick breaker logic module handles the basic gameplay. This module takes in the coordinates of the Ping-Pong paddle as input and determines what should be displayed and when game events happen. The logic module couples what is displayed to the internal logic as shown in Figure 8.

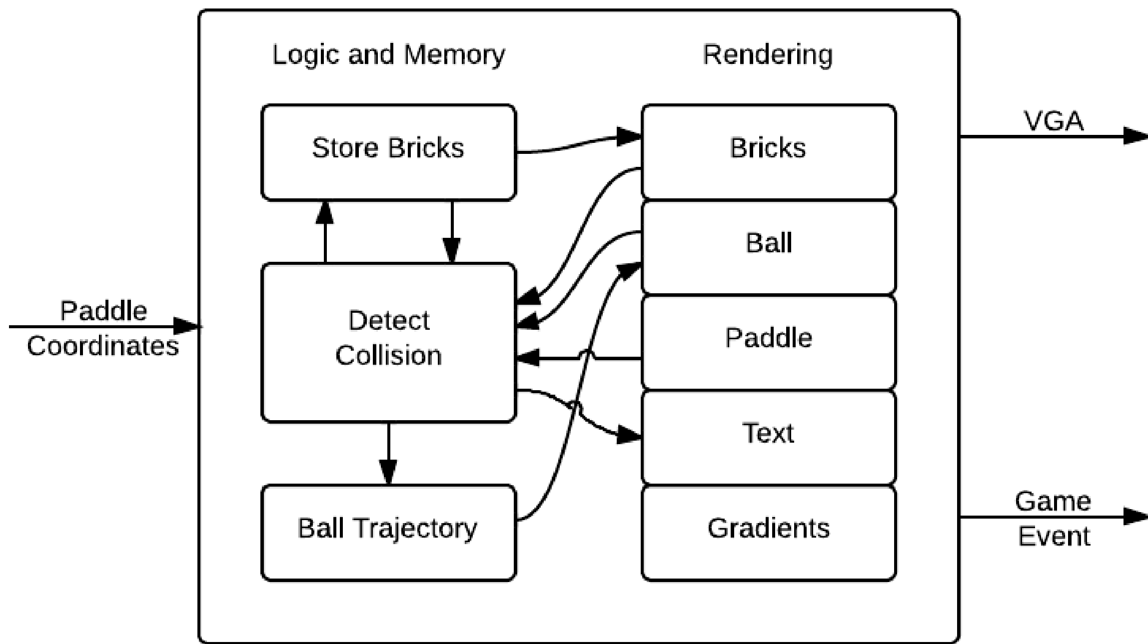

Figure 8: The internal block diagram of the logic module. The logic module takes in paddle coordinates and then closely couples the logic and memory and the rendering. The logic module then outputs a VGA signal and a signal for game events.

The logic and memory will be discussed in Section 4.1 and the rendering will be discussed in Section 4.2.

4.1 Logic and Memory

The logic and memory are the most basic implementation of the game consisting of 1) storing bricks, 2) detecting collisions, and 3) determining the trajectory for the ball.

4.1.1 Storing Bricks

The brick breaker game needed a way to conveniently store the existence, style, and placement of all the bricks. Since this was implemented in hardware (which tends to be static unlike dynamic programming languages) it made the most sense to have in memory a fixed number of brick memory placeholders. The brick memory placeholders - essentially a 2D array - simply held all the information needed to describe a brick and importantly whether the brick should "exist" or not. The brick memory placeholders were four bytes per brick and there was allocated space for 128 bricks although only 35 were used in the final implementation.

The brick array consisted of the following:

4 bits Exist	4 bits Style	9 bits Blank	10 bits Hor. Position	9 bits Vert. Position

The “exist” bits were used just as a zero (hide) or one (show) during the game implementation, but this was given multiple bits for extensibility. It was imagined that it would be useful to keep of special bricks that would have to be hit multiple times to fully break.

The “style” bits were used to label the color of all the bricks. The colors were determined by a case statement that corresponds to the following table. Ultimately, many more colors could have been used very easily, although the presence of just a few bold colors gave the game a strong retro feel.

0	Red
1	Blue
2	Green
3	Yellow
4	Pink
5	Teal
6	Darker Blue

The “blank” bits were a result trying to make the length 4 bytes and offer room to add more information later if needed. It was desirable to use multiples of whole bytes if it were later decided to use the labkit’s flash memory.

The “position” bits were to place the bricks on the screen. The full screen resolution was 1024 by 768 pixels, but the position of the bricks (as well as nearly all the game logic) was offset to match the inner gameplay area. The inner gameplay area was 896 x 604 pixels. The width of 896 pixels was selected because originally rendered block designs were 128 pixels wide and exactly seven bricks could fit horizontally in this space.

The bricks needed a way to be reappear when the game was reset. So the array that held the bricks was actually duplicated. The arrays were *bricksetupmemory* and *brickgameplaymemory*. The *bricksetupmemory* was intended to be fixed (unless changed through a level creator) whereas the *brickgameplaymemory* was intended to change each time of the bricks is broken.

Calculating whether or not a brick should show on the screen was more involved. To correctly register the brick color with the 65 MHz clock (with a clock cycle of about 15 ns) it was necessary to do some pipelining. Each brick had an instance of a module that calculated if current pipelined pixel position (called *hcount* and *vcount* based upon classical VGA output) was within the boundaries of the brick. The variable “*isbrickpixelactive*” determined if any of the bricks were active based upon the information from all the brick module instances and a special for loop was created that registered the “*activebrick*” index. The variables *isbrickpixelactive* and *activebrick* provided a way to determine 1) if there were a pixel at the pipelined *hcount* and *vcount* and 2) which brick that pixel belonged to. Ultimately, all the information needed for coloring and referencing the bricks was done in just one pipelined clock cycle.

4.1.2 Detect Collision

One challenging step was to detect and properly identify the collisions in the game. There were four sides of the gameplay area, a paddle that tilted, and 35 (or potentially 128 bricks) that all could have come into contact with the circular ball. The bricks each had four sides and four corners.

The fact that the ball was circular made it extra challenging. To compute the distance from the center of the ball would traditionally require pipelining as it requires too much arithmetic for one clock cycle. As a result, a clever solution was needed to determine collisions. The following algorithm was used:

1.) The first check was to see if the ball would pass one of the walls in the next clock cycle. Bouncing off wall was fairly trivial to calculate because the walls were all either vertical or horizontal.
2.) The second check was to see if the ball had hit any of the bricks. This overlap was determined by if one of the pixels of the ball overlapped with a brick. Potential overlap was checked each clock cycle and a flag as raised if there was any overlap and the *activebrick* number was registered for future processing for when the screen updated. The side of the brick where the collision occurred was also recorded. The position was stored as 0 for a top collision, 1 for top right, 2 for right, and so forth clockwise. The *activecollisionside* was determined based upon the location of the brick and the center of the ball.
3.) The third check was to see if the ball had hit the paddle. Similar to the bricks this was determined based upon if one of the pixels of the ball overlapped with one of the pixels of the paddle. If there was overlap, a flag was raised.

If there were a collision with one of the bricks the game event flag goes up to signal to the sound module to produce a collision sound. Additionally, the score increases by 10 for each brick broken.

Typically brick breaker games lose a life or have game over if the ball hits the bottom of the screen. However, the game was already challenging enough and so no penalties were added.

4.1.3 Ball Trajectory

After potential collisions had been detected, the next step was to determine the trajectory of the ball. The x and y positions for the ball were computed separately in parallel. The algorithm worked as follows:

1) If there were a wall collision, the ball would be placed adjacent to the wall and the velocity would change direction.
2) Else, if there were a brick collision, the ball would have to bounce appropriately. Bouncing off the sides of the brick was fairly trivial, but corners were much trickier. A simple approximation was used to either treat the collision as a side collision or a 45 degree angle collision.
3) Else, if there were a paddle collision, the ball would have its velocity directed upward and the paddle's velocity would be scaled and added to the ball.
4) Else, the ball would be in free space and acted under a constant pull of gravity.

The algorithm for the corners of bricks was to see if the ball's center was offset sufficiently far from the continuation of the edges of the brick. If the ball's center coordinates are close to the coordinate of the brick's edge it was treated as a purely horizontal or vertical bounce. Otherwise it was treated as a collision of 45 degrees. A parameter called *crit45* set the threshold for the regions. This is shown in Figure 9.

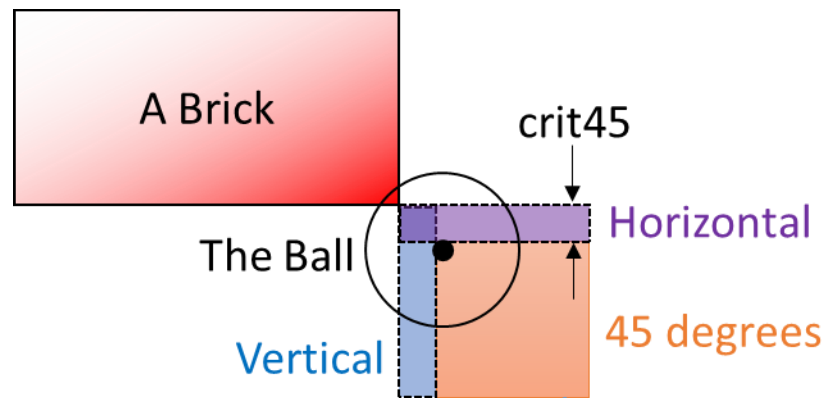

Figure 9: The brick corner collision algorithm. The position of the center of the ball determines the type of collision. If the ball's center is close to the right edge of the brick the collision is purely a vertical bounce. If the ball's center is close to the bottom of the brick it is purely a horizontal bounce. Otherwise, it bounces as if the brick had a surface at 45 degrees.

4.2 Rendering

There were five main layers to the rendering process that each were computed in parallel. The layers were the bricks, the ball, the paddle, the text, and the gradients.

The bricks were rendered based upon *isbrickpixelactive* and *activebrick*. Each brick module had its own 24 bit RGB output. Using the *activebrick* as an index, it was straightforward to show the correct color.

The ball was slightly challenging to display because it was circular. The calculations for ball were too much to do in one clock cycle. To fix this problem, the solution was to use pipelining. Here the horizontal and vertical displacements from the ball's center to a given pixel are squared in one clock cycle. The next clock cycle the products are then added together to use Pythagorean Theorem to determine if the pixel is within the range of the ball.

The paddle was also challenging because it was desired to be able to rotate the paddle. Originally, a pipelined solution was coded for computing the sines and cosines for the paddle's tilt. The input from the camera was simply the horizontal and vertical displacement of the blue and yellow regions on the paddle. Ultimately, it worked just fine to approximate the sine as a scaled version of the vertical displacement and the cosine as simply 1. The sine and cosine values were put into a rotation matrix to make the paddle on the screen correctly tilt with the actual paddle.

The layer for text was generated using a premade text writer. The memory required for coding text is fairly significant, so the actual module automatically doubles the size of the text which makes it slightly clunky. To make the title "BRICK BREAKER" larger, the size is again doubled. This makes the text very blocky, which fortunately goes well with the theme of brick breaker.

The last part to render was the background gradient. Originally, it was hoped to make an image and then simply display this image as part of the background. Unfortunately, the memory of the FPGA even with ROM was not sufficient to hold both the picture and the sound files. The rendered images had a background image with the gradients, a title, and some sample bricks as shown below in Figure 10. The bricks designs shown were actually converted into coefficient (.coe) files using a custom python script, and the files were ready to be uploaded to the ROM. Although the rendered bricks nor the background were never put on the ROM, the original appearance motivated the final design.

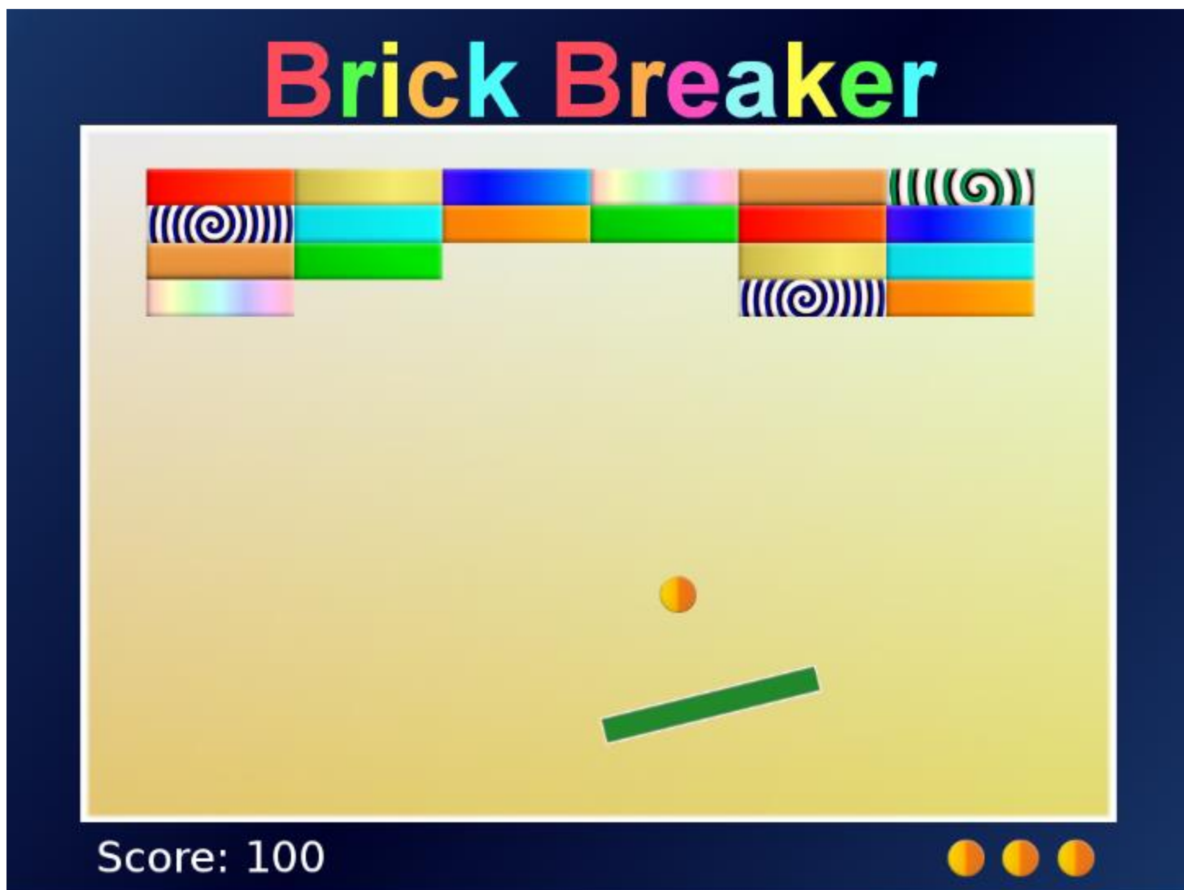

Figure 10: A view of the originally rendered game screen. There was not enough memory to actually display the exact background as an image, so the actual final image generated the gradients at runtime.

Jonathan very much wanted a similar final product with gradients. The final appearance of brick breaker with the gradients during the actual gameplay is shown below in Figure 11. The gradients are nearly identical to the original design. To speed up the calculation of the gradients, the ending RGB values was linearly interpolated such that the module could divide by 1024, which (since it is a multiple of 2) is reasonable to do in hardware.

The staff of 6.111 was greatly impressed by the gradient, and our project was said to be the only team that made a gradient in their project this year.

Figure 11: A view of the final screen. Notice the very gradual gradients for both the outside edge and the inner gameplay area. All of the display was created at runtime, and this constraint added to the retro theme of the game.

5 Sound Module (Jemale)

The sound module generates different sounds related to events that occur in the game. Currently, the module takes in an input which specifies whether or not there was a collision with the ball and a brick and whether the game is over. If it is the case of a ball/brick collision, the collision sound will be played. Once all the bricks have been broken a whistle sound is played to indicate the game is over. There is also a background theme that plays continuously throughout the game. The sample is a sixteen second clip taken from the sound library in Microsoft PowerPoint. Similarly, the collision and whistle sounds are also clip art.

The initial clips were in a WAV format, except for the theme that was MIDI. The clips were extracted from PowerPoint and then resampled to 48 kHz and converted to .coe files using staff provided MATLAB scripts. The MIDI file required the extra step of converting from MIDI to WAV format prior to .coe generation. Once the .coe files were obtained, Jemale then down-sampled to 6 kHz by taking every eighth line in the .coe files (using a python script) in order to save space. During playback the clips are up-sampled to 48 kHz. After the down-sampling, the .coe files were combined into a single .coe file and loaded on to the labkit by creating a ROM and specifying the single .coe file as the initial file. The ROM was a dual port ROM created using the FPGA BRAM bank. The ROM was made dual port because we wanted to always have the theme music playing and then play the other sounds on top of it. With a dual port ROM this is easy as there will be one address input that is constantly looping over the theme addresses and another address input which loops over an address range determined by the specific game event sent to the module.

Initially, this module was challenging because sounds did not appear to play when the respective signals were sent. Further testing showed that this was because the signals were like pulses (e.g. a short collision pulse) and thus the sound would only play for that short collision period. Jemale modified this such that once the sound begins to play it runs until completion. The sound is output from the labkit to speakers shown in Figure 12.

Figure 12: These were the speakers used in the final implementation. The speakers played a looped 16 second track that was noted by staff as being “retro” and fitting for this game. The speakers also played collisions upon impact with bricks and a whistle upon game over.

6 Conclusion

In all, Jemale and Jonathan had a lot of fun creating and playing Brick Breaker. Brick Breaker was a successful educational experience that solidified our understanding of Verilog and hardware electronics. Brick Breaker successfully ran at the end of the semester and provided a fun experience to those who stopped by the lab bench to try it out. Brick Breaker demonstrated our learning and new found expertise in Verilog coding.

7 Acknowledgements

Jemale and Jonathan would like to thank Prof. Gim Hom for his instruction and support throughout the semester, the TAs Devon Rosner, Rishi Naidu, Pranav Kaundinya, and Luis Fernandez, and the communication instructors Jessie Stickgold-Sarah and Michael Trice.