

Fpglappy Bird: A side-scrolling game

Wei Low, Nicholas McCoy, Julian Mendoza
6.111 Final Project Report, Fall 2015

1 Introduction

On February 10th, 2014, the creator of *Flappy Bird*, a popular side-scrolling game for mobile devices, removed the game from mobile application stores (Apple, Android, etc.). The reason for the removal was due to the game becoming “an addictive product” that had “become a problem” (Nyugen). Currently *Flappy Bird* is only available to those who downloaded the game before its deletion from application stores or through a fan-created website, flappybird.io.

This final project aims to implement this popular and exciting game in hardware. Previous implementations of this game, such as the original mobile app or fan-created online version, exist in software form. As a result, bringing this game into the hardware realm truly distinguishes our project from previous implementations of *Flappy Bird*. In the preexisting software-implemented versions of this game, a small bird must hop to avoid obstacles in the form of green pipes that scroll across the screen from right to left. The user taps the screen or clicks a button to make the bird jump up, but otherwise the bird is constantly falling. Due to the counter-intuitive control scheme, which required the user to remain vigilant, constantly monitoring the bird’s path of movement during the game, *Flappy Bird* became renowned for its difficulty and infuriating controls.

Our goal is to implement our own version of the game on an FPGA (Nexys 4 DDR) and make the game more interesting and difficult by incorporating a vision-tracking element that requires the player to jump in order to control the on-screen bird protagonist. Specifically, the vision-tracking component will look for a bright object placed over the player’s face (a “beak” in the form of a paper party hat), and then use the coordinates of this said object as input for the player coordinates to the game. Therefore, a large change in the player’s vertical position, as monitored by the vision-tracking component, will trigger a hop for the player’s on-screen character. By incorporating this vision-tracking specific control screen, player’s movements will be continually tracked, resulting in greater emphasis on the player’s attention and engagement within the the game. Additionally, during game play, the player’s face will be layered onto the bird sprite, allowing for a truly personalized experience.

Stretch goals of this project include implementing our project on two FPGAs connected via serial link to create a multiplayer experience and dynamic calculations that allow rotation of the bird sprite during jumps. The ideal outcome would be to deliver an aesthetically pleasing and physically engaging game that demonstrates successful implementation of the game integrated with the vision-tracking component.

2 Design

2.1 System Overview

The project can be visualized in four major blocks as shown in Figure 1: object tracking, gameplay logic, audio, and video display.

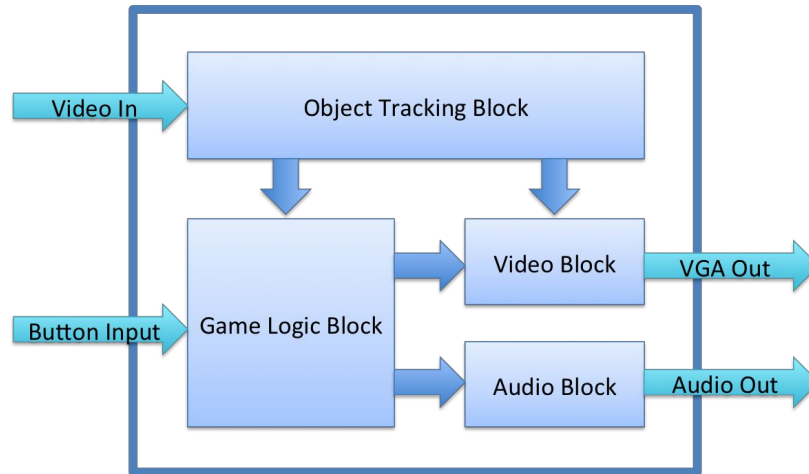


Figure 1: The high level design of the game

2.2 Design Decisions and Motivation

The primary motivation for this project is to make a well designed and physically engaging game that emphasizes and personalizes user experience. Completion of this project requires full functionality of every module. As a result, we require that each of the four critical blocks be independent of the others, allowing for streamlined parallel development and comprehensive testing of the modules. For object tracking purposes, the player is required to wear a “beak”, a brightly colored party hat over their nose and mouth, similar to a mask. The purpose of the physical “beak” is to provide an object of high contrast that allows us to easily determine the location of the player’s face (Figure 2 below). The vision-tracking component will look for a bright object placed over the player’s face (the “beak”), and then use the coordinates of this object as input of the player coordinates to the game. As a result, any large changes in the vertical coordinates of the player will be detected as a jump control sequence, which will cause the on-screen character to consequently jump within the game.



Figure 2: “Beak” on player face

At the highest level, the project will track the player’s face for vertical changes during gameplay and output sound effects (Audio Out) and video graphics (VGA out) where relevant. Depending on the player’s vertical movement, the game logic processing block will change and update the sound and graphics. The vision/object processing block takes input from the OV7670 camera and processes the

incoming video stream to determine the location of the “beak” and then sends the location coordinates to the game logic block. The game logic block uses an input button to start the game, and uses a combination of physics and saved previous player location states to determine when to jump. Additionally, the game logic block handles the location of new obstacles, and controls the movements of the bird. If the bird collides with an obstacle, it ends the game. The game logic then sends position data of objects, specifically the player face and obstacle location coordinates, to the video block, and also tells the audio block what sound effects to play. The video block takes input of the player face coordinates to be able to retrieve the player face sprite from BRAM and also takes in the object positions to convert them into image representations. All of this information is then sent over to VGA to be displayed. The audio block plays sound effects based on events determined by the game logic.

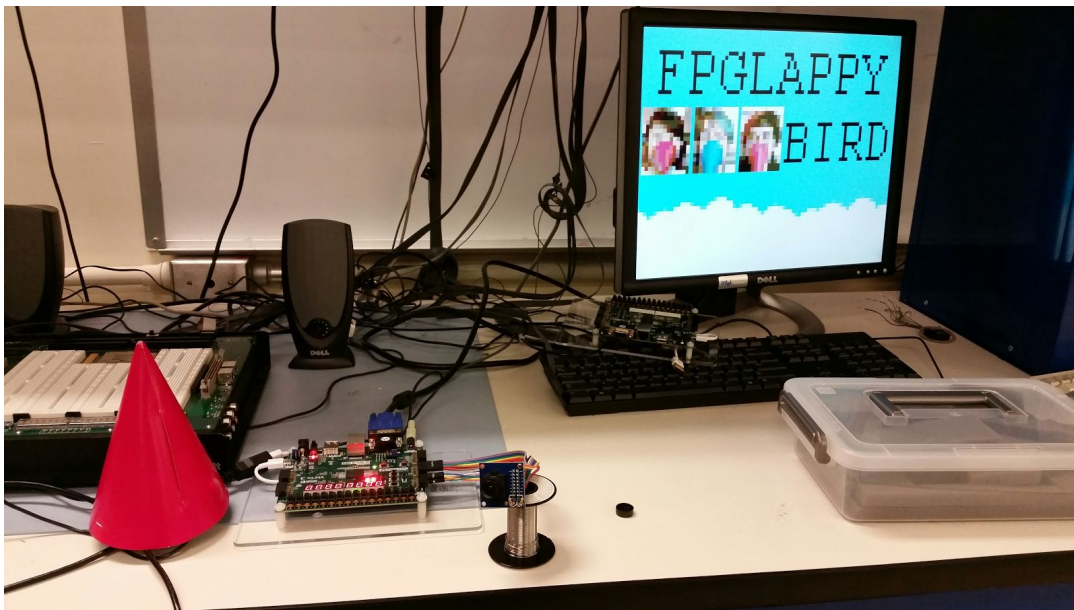


Figure 3: Setup Including FPGA, Camera, Beak, Speakers, and Monitor

3 Implementation

3.1 Vision (jwend)

Introduction What sets this Flappy Bird implementation apart from the original game is the use of motion tracking as the main method of control. The purpose of the vision module was to take the data stream from the camera and convert it to the position and velocity of the player. This data are then used by the game logic module to evaluate when the player is jumping and when they are not. In addition, the module will save the camera frames in BRAM to be used for the sprite of the player. The main goal of this module was to do all of this as efficiently as possible, specifically using as little memory as possible. We make the player wear a brightly colored beak in the form of a red paper party hat in order to reduce the problem complexity to an implementation of blob tracking. The initial approach for this module proved impractical, so it was replaced with one that was much more space and computationally efficient partway through the project.

Initial Approach/Flaws The original idea for the vision module consisted of working with the RGB16 data directly, converting to grayscale and post-processing it afterwards to pick out the color and find the center of mass (See Figure).

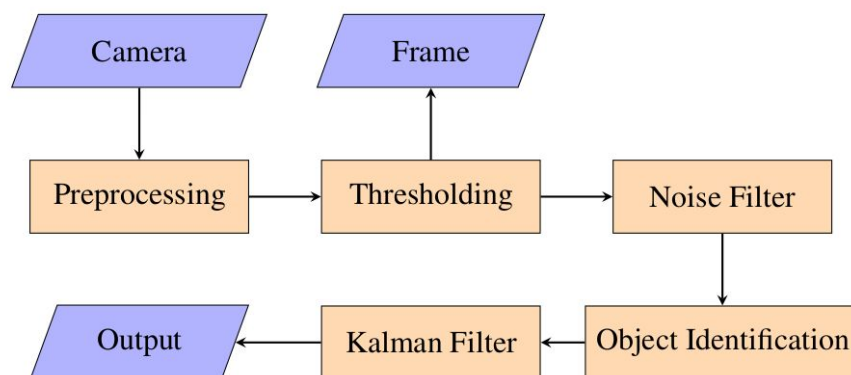


Figure 4: Original Design

There were two big issues with this approach, noise and computational complexity. Firstly, processing RGB data directly results in a lot of color noise which required post-processing to filter out. In the RGB color space, thresholding isn't as simple as thresholding the RGB values individually to figure out if a color matches, because an intensity shift in color causes diagonal movement over RGB (See figure). The best approach to figuring out if a color is close to the desired color involves taking the dot product of the two.

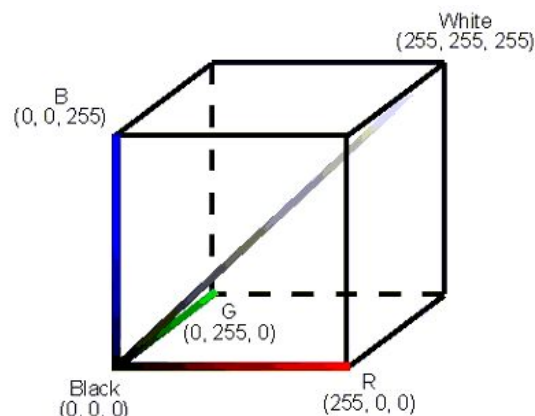


Figure 5: RGB Color Space

$$\frac{P \cdot D}{|P||D|} = \cos \theta$$

Where P is the vector of the current pixel and D is the desired vector (color). Even this approach produced a lot of noise, though. Since we required post-processing of the data, this also required saving and rereading from BRAM, as well as pipelining. Additional computational complexity of the problem came from the multiplication, division and square root, which would have required multiple clock cycles to calculate accurately enough.

New Approach The complexity of the first approach led to research into different approaches to the gray scale step, this led to the decision to change the color space from RGB to YUV. YUV is most commonly used to transmit video because it takes into account human perception and allows you to store data in less bits as a result. The key feature for us is that it separates luminance (intensity) and chrominance (colors). This means that changes in intensity will result in a 1-dimensional shift in the YUV color space, so simple thresholding can be used to pick out colors. In addition, this feature of the YUV color space means that there will be less false positives. So there will be a natural reduction.

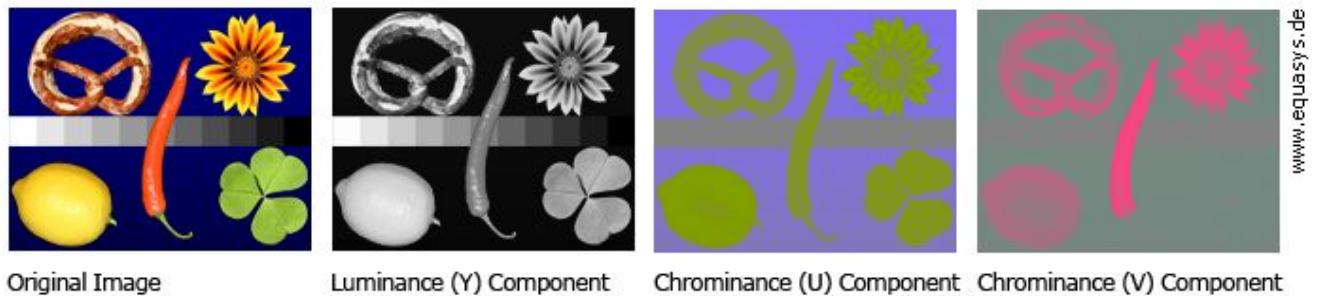


Figure 6: YUV Color Space Example

Implementation of New Approach With the new approach, the new goal was to process every pixel the same clock cycle it was streamed from the camera. This meant the actual vision tracking module would require no bram, other than saving the frame for the video module. Doing this involved a few optimizations.

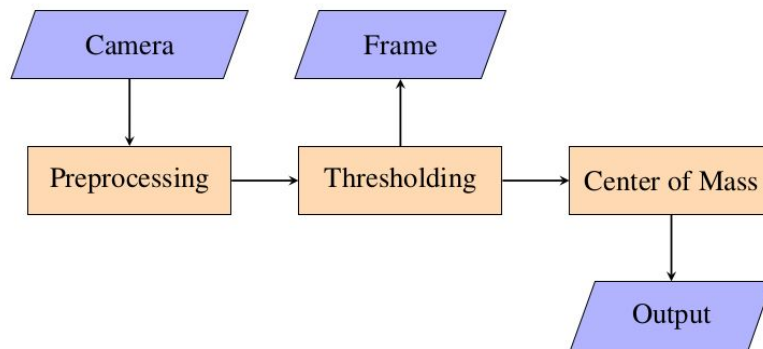


Figure 7: New Design

With the new approach, the preprocessing and thresholding steps happen together, and the center of mass calculation occurs at the end of each frame. We decided to keep the camera outputting 16-bit RGB values, and doing the conversions to YUV ourselves, because it would allow for more precision than what the camera's 8-bit YUV output would give. To greatly reduce noise, we decided to change the gains on the camera's output to make red (the tracked color) stand out more. The preprocessing module simply consisted of doing the RGB/YUV conversion.

$$\begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.500 \\ 0.500 & -0.419 & -0.081 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} Y \\ U \\ V \end{pmatrix}$$

Figure 8: RGB to YUV Conversion

The issue with the exact conversion was that it required floating point math, to get around this, we approximated the matrix to

$$\begin{pmatrix} 1/4 & 1/2 & 1/4 \\ 1/4 & -1/2 & 1/4 \\ 1/2 & 0 & -1/2 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} Y \\ U \\ V \end{pmatrix}$$

Figure 9: Approximated Conversion

so the conversion simply consisted of some addition and shifting, which was very possible in one clock cycle.

The center of mass calculation involved taking the mean of all the pixels which were part of the cone. To do this while streaming, whenever a pixel was within the threshold we added it's x and y coordinates to `sumX` and `sumY`. So if a pixel as at (320, 42) and was within threshold, then `sumX = sumX + 320` and `sumY = sumY + 42`. After a frame was done, we calculated the means of both by dividing by the total number of pixels which were within threshold.

Another key difference between this approach and the original one was the decision to leave out the Kalman Filter. With RGB processing, it would have been necessary to have a Kalman Filter because otherwise, the false positive pixels would have created substantial noise, enough so that a jump might be registered when a player was simply standing still. The Kalman Filter would have fixed this issue. After implementation, two things led me to drop the Kalman Filter all together. Firstly, the new implementation had virtually no noise, and few false positives. Since players could stand still and the calculated position would stay within +/- 1 pixel, it seemed fine. Additionally, this noise ended up being key, as the randomness of it was used in the game logic module as a random bit generator.

Simulation Examples To test out the algorithm, I implemented it in python using the `skimage`, `scipy` and `numpy` packages. This implementation was several orders of magnitude slower than the final implementation on the FPGA, but it's purpose was to confirm that the approach was valid, not to match

the efficiency of the parallelized FPGA. To test out the algorithm, I used a cone image (see figure), which I put through several iterations of the algorithm.

The first thing I attempted was to use my original, naive approach. This simulation disregarded the computational and space complexities of this approach. Additionally, since I was using a static image, I didn't include the Kalman Filter which was intended to be in the original approach. What I found from this, was that even using the dot product to calculate the "closeness" of two colors, there were still a lot of false positives. In this image, they wouldn't impact the center of mass calculations substantially, but in the real world there would be much more noise and the beak would be a much smaller portion of the overall screen, so the noise would be substantial. (See figures 11 and 12)



Figure 10: Original Cone Image

The next step was to simulate the YUV approach. Similar to the actual implementation, the simulation used only one loop through all of the pixels, calculating the YUV value, thresholding and doing the center of mass calculations all together. There was no noise reduction or additional passes to the pixels. There are two figures, one in which I made educated guesses on the thresholds. The second was a result of refining the thresholds for better results. One key feature of this image is that even with the expanded threshold values, the amount of noise was still substantially less than with RGB.

Future Goals If I were given more time to do this project, I'd focus on adding two things: support for more colors and replacing the current RGB to YUV conversion with one that is more accurate and uses floating point math. For adding new colors, the changes would just have to be in the parameters, so it would simply involve going into lab with different color beaks and trying out different configurations and thresholds to optimize for desired colors. In addition, this would require creating at least two modules to change the parameters on the camera and for the thresholds.

Replacing the RGB to YUV conversion would require pipelining because of the floating point math operations, but this would simply delay the output, and wouldn't require adding more steps to the process. Since the time between frames is substantial, there would be virtually no loss to tracking data, and this change would allow for more accurate color representations in YUV. The end result of the increased color fidelity is that picking out colors would become easier, and more robust. In addition, this would mean that the need for optimizing the camera's color output to the desired one becomes more unnecessary.



Figure 11: (RGB) Post Grayscale

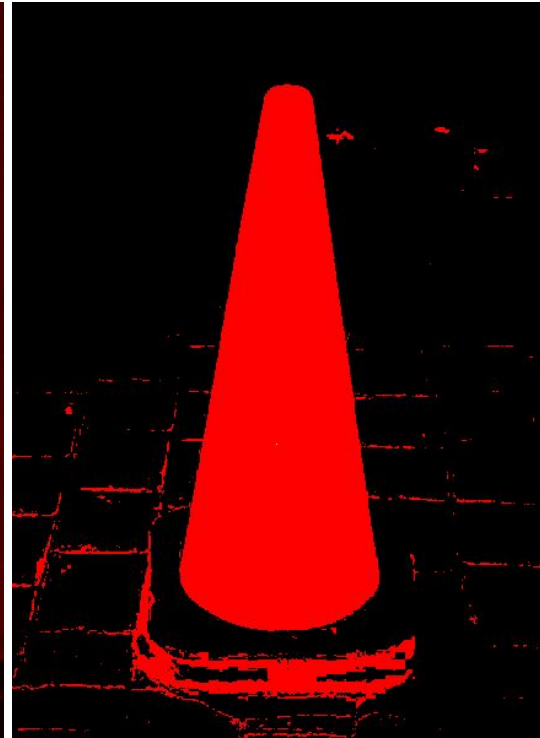


Figure 12: (RGB) Post Thresholding/
Noise Filtering

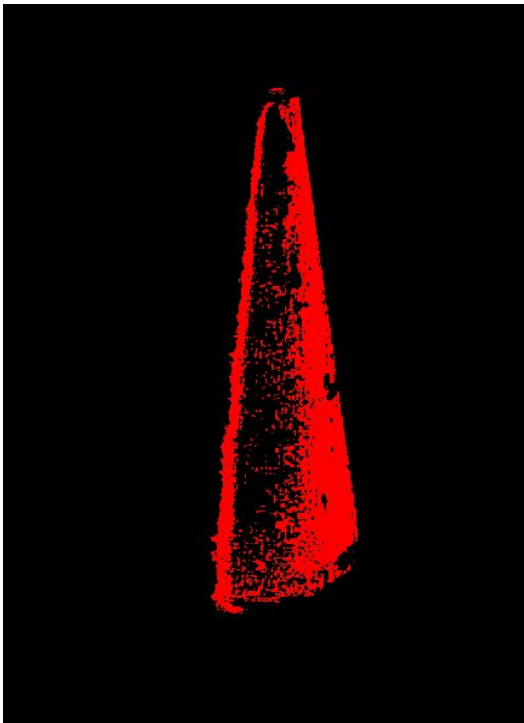


Figure 13: (YUV) Guessed Threshold

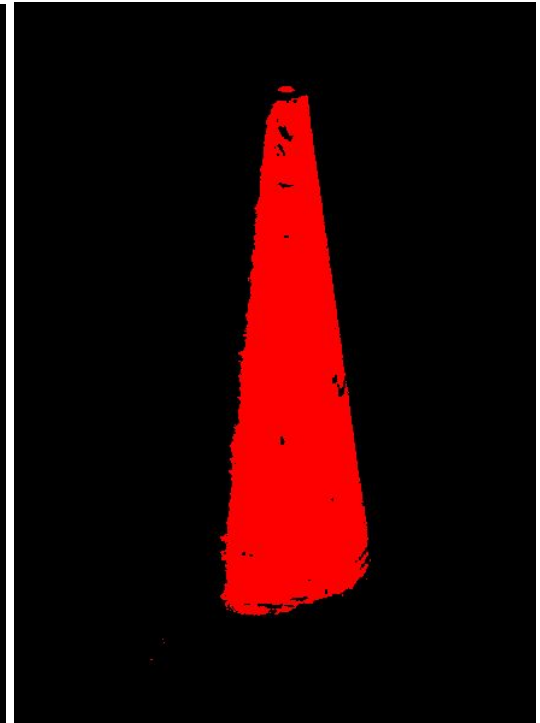


Figure 14: (YUV) Refined Threshold

3.2 Game Logic (weilow)

3.2.1 Introduction The gameplay will be similar to that of the original *Flappy Bird* in that the goal of the game is for the player to avoid colliding with generated onscreen obstacles. The purpose of the game logic block is to handle all aspects that make a game playable, such as the different state transitions and physics that occur during gameplay. Specifically, the game logic block handles the obstacle location generation, collision detection, movement physics and score tracking. This block will take in inputs in the form of player position (x, y coordinates) and signed velocity (change in player position) from the vision block to make the decision of whether or not the player has jumped. Depending on the state of the game, the game logic block outputs enable bits controlling various game sounds to the audio module. Additionally, the game logic block will output the locations of the generated obstacles and the location of the player bird sprite after physics manipulation to the video block.

3.2.2 The Process

In my initial approach, I accounted for only three modules: gamestate, physics and highscore. However, as I began implementation of these modules I realized that my original design was flawed in that it did not account for the complexities that come with randomly generating obstacles, different clock domains or the complexities needed to create smooth player (bird sprite protagonist) movement.

As a result, I needed to redistribute the functionality between modules and devise a different hierarchy that would allow me to use different clock domains for different functions, while ensuring that the overall game play is unaffected. For example, the player bird sprite location updates at a different rate than the obstacle location updates. In response to this need to update aspects of the game at different rates, two different clock domains were used for this project - the vga's outputted vsync domain, which was used in the obstacle_gen module and the default 25mhz clock, which was used for all other modules. The vision's VSYNC_C, one_hz signal and sixty_hz signals were used for the updating of previous player coordinates in the physics module, for the countdown (in seconds) in the timer module and for the computation of bird locations respectively and were therefore not utilized as clock domains.

Finally, because we are only interested in the high score in the current running instance of *Fpplappy Bird*, we need only store the top score achieved. Since the maximum score would be a max of 2 digits long, there was no need to retrieve/save the score by writing/reading to an SD card. Instead, each time the player loses the game, the score achieved is sent to the video block, which outputs the maximum of this currently achieved score and the previous score. By eliminating the need for an SD card, we eliminated a layer of processing that would come from reading/writing from the SD card, reducing the possibility of delays/lag time during gameplay. Since a high score specific module was no longer needed, I simply added scoring capabilities into the obstacle_gen module, since within that module I would already have access to all obstacle locations. As a result, the final breakdown of modules within the game logic block are as follows: gamestate, physics, collision_detection, obstacle_gen, timer, randombit, onehzstart and sixtyhzstart modules (see figure 15 on the next page). Additionally, a listing of all modules and their respective inputs and outputs are included in Table 1 on the following page.

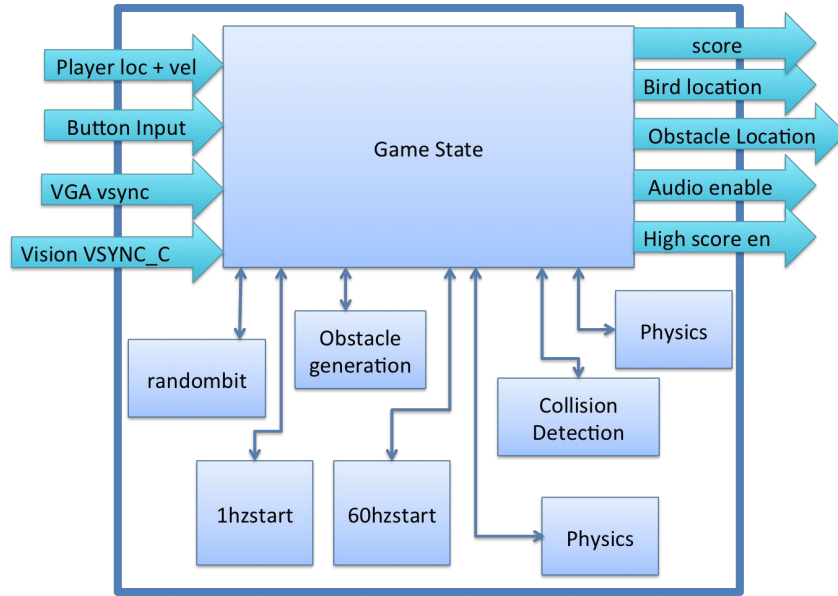


Figure 15: Updated Game Logic Block

Module	Inputs	Outputs
gamestate	clock (25mhz), start, reset, jump, collision, expired, one_hz	hs_enable, home_enable, updatepos, pause, reset_physics, reset_score, reset_collision, sound_collide, sound_jump, sound_background, start_timer [3:0] state
obstacle_gen	clock (vsync), updatepos, reset_score, reset_physics [3:0] randbit	obs1en, obs2en, obs3en [6:0] score [9:0] obs1x, obs1y, obs2x, obs2y, obs3x, obs3y
collision_detection	clock (25mhz), updatepos, reset_collision, obs1en, obs2en, obs3en [9:0] bird_x, bird_y, obs1x, obs1y, obs2x, obs2y, obs3x, obs3y	collision
physics	clock (25mhz), updatepos, reset_physics, sixty_hz, frameupdate, up, [9:0] player_x, player_y, signed [10:0] signed_y_vel	jump, prev_enable [19:0] diff, [9:0] bird_x, bird_y, prev_player_locx, prev_player_locy
timer	clock (25mhz), start_timer, one_hz	expired [2:0] countdown
randorbit	clock (25mhz) [9:0] player_x	[3:0] randbit
onehzstart	clock (25mhz)	one_hz_enable
sixtyhzstart	clock (25mhz)	sixty_hz_enable

Table 1: List of Modules, Inputs and Outputs for Game Logic Block

3.2.2.1 Game State Module

The game state module handles the different states within the game, which will be: START, PLAY, PAUSE, LOSE.

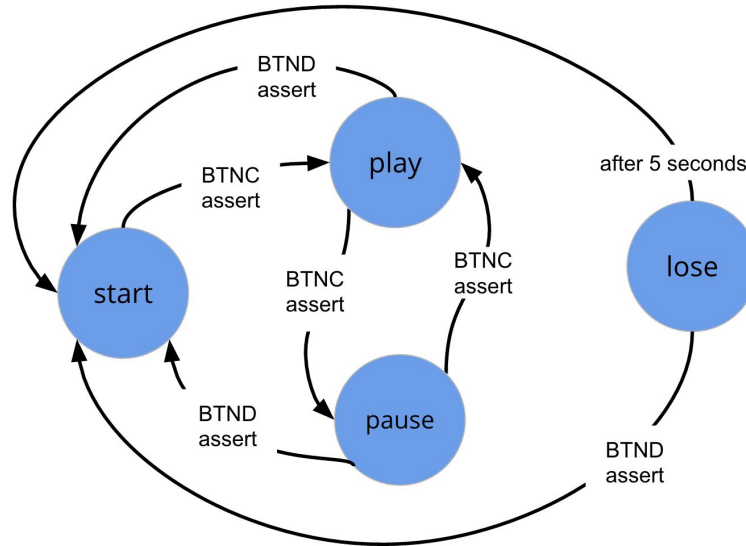


Figure 16: Game State diagram

Within the START state, the game’s default screen will load, bearing the game title. Upon pressing BTNC on the Nexys , the PLAY state will begin.

During the PLAY state, the player’s face will be displayed in sprite form and no obstacles will be displayed on the screen. However, once the player jumps, then the obstacles will begin generating on screen. At each jump that the player makes, a jump sound will be played. If at any point during the PLAY state, the player presses BTNC on the Nexys, then the screen will have a red filter effect and all objects will freeze in place. If at any point the player’s bird sprite hits an obstacle during the PLAY state, the game will transition to the LOSE state. When a collision occurs, the collision sound will also play. Finally, during the PLAY state, the player’s accumulated score will be displayed on the 7 segment display.

Within the PAUSE state, if the player presses BTNC again, then the game will pick off from where the player left off.

Within the LOSE state, the highest score obtained so far will be displayed on the screen. the game will wait 5 seconds before transitioning back to the START state or the player can simply press BTND on the Nexys to simply reset the game to the START state.

3.2.2.2 Physics Module

Since the game has a scrolling background and obstacles are generated on screen from right to left, the player bird sprite has a fixed horizontal coordinate, the player's jumps only impact the onscreen sprite's location in the vertical direction. Keeping this movement in only one dimension eliminates the need to perform rotations and smooth out the x-y translational motion via interpolation, reducing the complexity of calculations and ensuring that all needed calculations to produce the updated sprite position can occur within a single clock cycle.

My original concept for movement physics lead to a very staggered and choppy behavior. Before, on a valid input jump, I would update the player bird sprite location by subtracting 10 from the value of the previous `_player_y` coordinate each 25mhz clock cycle. If no jump occurred, then I would simply subtract 5 from the value of the previous `_player_y` coordinate. Since by convention the top left corner of the screen has coordinate (0,0), a vertical translation up results in a decrease in numerical value and a vertical translation down results in an increase in numerical value for the coordinate. This approach lead to a linear movement path, where the player bird sprite's location would either increase or decrease by some set value (See figure below).



Figure 17: Linear Movement Path

As a result, I had to change my approach by instead updating the `bird_y` coordinate based on first calculating a velocity value that updated on a 60hz clock cycle. By doing this, I could impact the translational change in the `bird_y` coordinate in a quadratic function, resulting in a better simulated movement path when jumping and falling. To do so, I used two constants, `VELOCITY_UP = 220` and `GRAVITY = -11`, where `VELOCITY_UP` represented the velocity to be applied in the event of a player jump and `GRAVITY` represented the gravitational constant affecting the player's movement and set the initial velocity to 7, a value that was arbitrarily picked. Finally, by only updating the `bird_y` coordinate every time the velocity value calculated reaches a multiple of 48, I was able to increase the precision of bird sprite movement, completely eliminating staggered changes in bird sprite position. Therefore, the sprite's movement no longer jumped from one spot to another in a linear fashion, but rather in a quadratic fashion (see figure 18 on the next page).

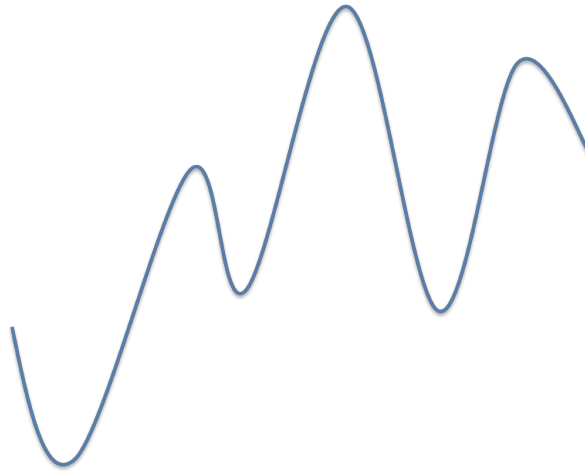


Figure 18: Quadratic Movement Path

In the presence of a jump, the formula for velocity was: $\text{velocity} = \text{VELOCITY_UP}$. In the presence of no jump, the formula for velocity was: $\text{velocity} = \text{velocity} + \text{GRAVITY}$. The formula for bird_y coordinate change was: $\text{bird_y} \leq \text{bird_y} - \text{velocity}/48$, in the event the bird did not reach the top or bottom edge of the screen. If the bird reached the top or bottom edge of the screen, then a collision would be detected and the player would lose the game, removing the need to continue to update the position of the bird sprite.

3.2.2.3 Collision Detection Module

For the collision detection module, the implementation was quite straightforward, essentially I needed to check if the bird ever went beyond the bounds of the output VGA screen or if it collided with an obstacle. Since the bird's movement is restrained to only the vertical axis, it will never surpass the left or right edge of the screen. When calculating collisions, the size of the bird must be taken into account. For *Fpplappy Bird*, the bird coordinate refers to the top left corner of the sprite. Since the size of the bird is 64x64 pixels, collision detection must take into account the entire size of the bird. Therefore, if $\text{bird_y} + 64 \geq 505$ or $\text{bird_y} \leq 42$, then the bird has surpassed the bounds of the screen and the collision bit is set to 1. The other time a collision occurs is if the bird collides with an obstacle. This is calculated by checking if the bird sprite fits within the gap contained within the pipe obstacles (refer to Figure 24 in the Video block section, 4.3). Since the obstacle coordinates are generated with the convention of the top left corner of the empty space between the pipes, which is essentially the rectangle between the top and bottom portion of the pipe obstacle in which the bird may pass through, we are able to check if the bird fits within the bounds. So, a collision also occurs if $(\text{bird_x} + 64 \geq \text{obstacle_x}$ and $\text{bird_x} < \text{obstacle_x} + \text{obstacle_width})$ and $(\text{bird_y} < \text{obstacle_y}$ or $\text{bird_y} + 64 > \text{obstacle_y} + \text{obstacle_height})$.

3.2.2.4 Randombit and Obstacle Generation Module

To generate pseudo-randomized obstacle locations, I originally designed the randombit module to output a number between 1 and 10 using the $\$random$ variable. However, the flaw in the utilization of this $\$random$ variable was that I quickly found out that $\$random$ only works in simulation, not in physical

hardware. I then had to devise an ulterior method of generating pseudorandom bits to use in the obstacle generation module. Since the player's location coordinates change once every camera frame (even if they are standing still, there are still natural slight movements that occur), it seemed reasonable for me to take the 4 least significant bits from the player_x position and use them as my pseudorandom bits, resulting in an output of randbit [3:0] from the randobit module.

For the obstacle generation module, there were two important tasks I needed to accomplish. The first was that there can be at most three obstacles onscreen at a time and that they should all be the same distance apart. That is, the obstacles should be spaced out evenly on the screen to give the player a reasonable buffer to change position their vertical position (e.g. if a player needs to go through an obstacle that has a low gap location and then immediately through an obstacle that that has a high gap location - see figure 24 in section 4.3, the Video block section). The convention for obstacle location generation was that the obstacle x, y coordinate corresponds to the top left corner of the valid rectangle between the top and bottom portion of the obstacle pipes, that the player bird sprite can pass through. As a result, the x position of the obstacles need to be generated at a constant rate and therefore cannot be randomized, since gameplay requires the player to be able to see obstacles moving from the right to left side of the screen in a predictable and steady rate. Therefore, in the obstacle generation module, the vsync clock domain from the Lab 4 vga module was used. This was to ease the updating of the x positions of obstacles and ensure they obstacles moved one pixel to the left at each positive edge of the vsync clock. The second component was the fact that the y position of the obstacles however, needed to be generated pseudorandomly, as the game would be quite boring if the player saw the same three obstacle gap heights at the same positions. Taking in the 4 bit wire randbit from the randobit module, my formulas for generating the y component of each obstacle were as follows:

```
obs1y <= (randbit[2]==1'b0)? 200+(randbit*10):300-(randbit*11);
obs2y <= (randbit[0]==1'b1)? 300-(randbit*7):125+(randbit*2);
obs3y <= (randbit[3]==1'b1)? 300: 50+(randbit*3);
```

Such that there were 16 different y coordinates that could be generated for obstacle 1 and 2 each and there were 9 different y coordinates that could be generated for obstacle 3.

Finally, the obstacle generator module was also responsible for keeping track of the player's score thus far. Since I would already have access to all of the obstacle locations, I could simply track if an obstacle's has moved far enough to the left past the bird_x coordinate that no collision would occur. Essentially, a +1 was added to the running score count whenever one of the obstacles had an x-coordinate equal to 135. This is due to the fact that the bird_x coordinate is at 200 and since the obstacle has a width of 64, the obstacle's x coordinate must be less than 200-64, which means it must be less than 136.

3.2.2.5 Timer Module, Onehzstart, Sixtyhzstart Module

The purpose of the timer module is to simply countdown 5 seconds when the player loses the game. This countdown serves as a delay between the player losing and then restarting the game itself, allowing the high score to be displayed onscreen. This timer module was an edited version of the one I developed for

Lab 4, where instead of having multiple possible timer delay values, there was only one preset one for 5 seconds.

The purpose of the onehzstart module was to serve as a clock that asserted every one second, which was necessary for the timer module. The purpose of the sixtyhzstart module was to assert 60 times a second, and serve as an input clock for the physics movement.

3.2.3 Testing Procedure

To test the various functionality of the timer, onehzstart, sixtyhzstart, randombit and collision modules I wrote testbenches (included in the gamelogic_tb.v file) that checked that modules worked as predicted. For testing the gamestate module, namely the transitions between the various states and to make sure the correct bits were enabled, I used the buttons on the Nexys to represent state transitions and utilized the LEDs to represent various output bits, such as the collision and jump sound enable bits. To test the physics module, I first had to integrate my physics module with Nick's video module. Then, I ran the code on the Nexys with a VGA monitor output to check that the jump and fall movements of the bird sprite looked smooth (as in the translational pixel updates occurred at a reasonable rate). To handle the case of whether or not obstacles were generated in the proper locations and moved at the correct rate, I made sure my obstacle generator module and Nick's video module were integrated. Then, I tested the obstacle generator module on the Nexys with the VGA monitor output to check that my generated obstacle locations were correct.

3.2.4 Improvements/Future Iterations

If I had more time, I would want to generate a true random number generator in hardware that would increase the randomness in the obstacle locations. Since I used several bits pulled from the player location given by the camera, I did not have as much range in variety as I would have hoped for. Although Vivado has a \$random variable, that works in simulation, it is not implemented in hardware, which I learned much later when I attempted to generate obstacles using the \$random variable. Another factor I would want to add would be varying levels of difficulty, selectable through the Nexys input switches. On higher difficulty levels, the speed at which obstacles move across the screen from right to left would increase, adding for added difficulty to the game. Overall, I was quite satisfied with the game logic block, including overall flow of gameplay.

3.3 Video (nmccoy)

The video block will perform all necessary operations for displaying the game's state. This should provide the user with an interface as intuitive and aesthetically pleasing as possible. To do this, objects are displayed on the screen, represented by custom sprites that are stored in memory. For the gameplay itself, a sprite pipeline takes in the locations of every object from the game logic module, then places sprite images at these locations, laying them over a scrolling background. In addition to the main gameplay, there are other screens involved. An opening title screen (Figure 19) is displayed at the start of the game, which is also stored in memory. A high score screen displays the game's highest score as a two-digit number (in case anyone ever manages to score two digits) after the player loses. A pause screen applies a red filter to the image and pauses the scrolling of the background to represent the paused state.



Figure 19: The Startup Screen

The inputs to this module are the object locations, vga hcount/vcount, enable bits for the various screens (startup, high score, pause), and an access port to the camera's memory. The output is the RGB value to be displayed at the current pixel.

The main sprite pipeline is formed around many sprites representing the various objects that can appear on the screen. As shown in Figure 20, it uses many different images, which are each stored in a synthesized IP ROM. These ROMs are preloaded with the values corresponding to the images they are meant to display (more on this later). Each sprite is passed its x/y location, as well as the h/v count from the VGA signal, and determines what value to make the pixel. If the current location does not overlap with this sprite, or if we are in a transparent area of the sprite, the pixel remains unchanged from the previous stage.

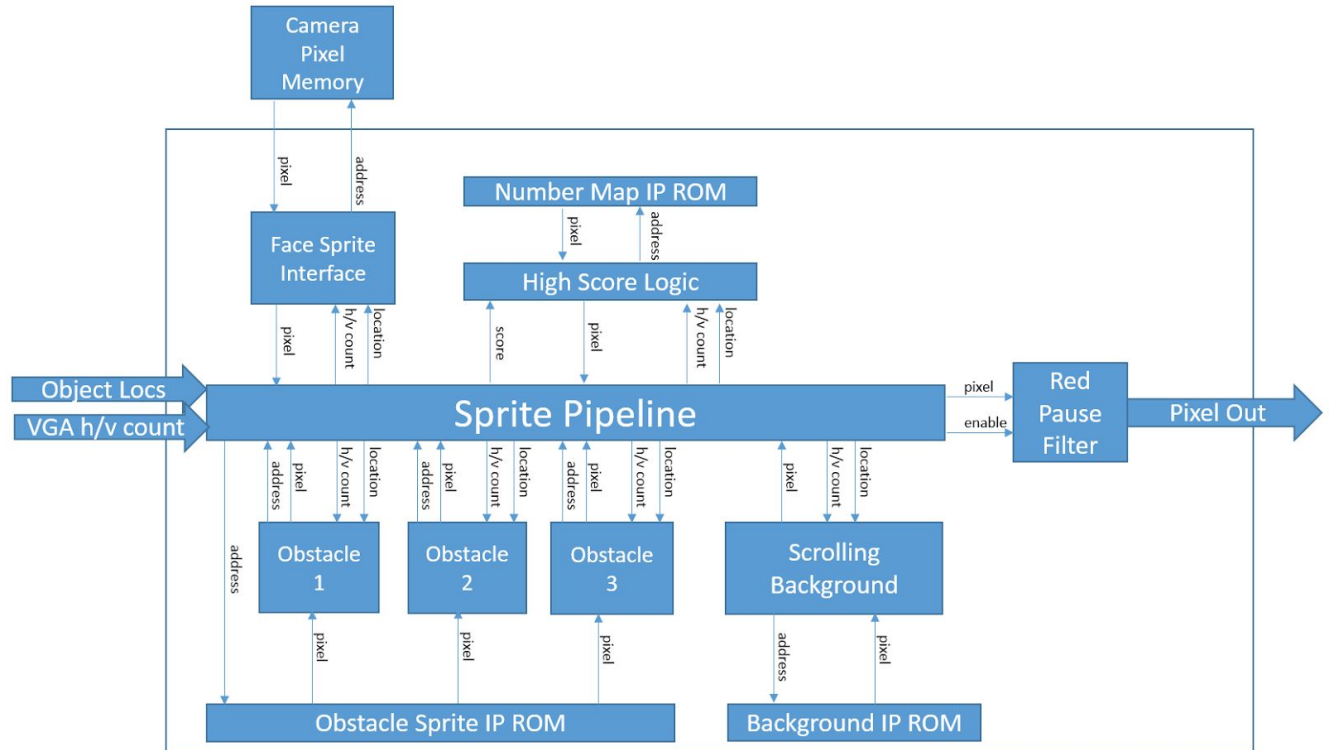


Figure 20: Video Module Block Diagram

The sprites were designed in Microsoft Paint, and saved as a PNG image. I modified a Matlab script to generate .coe files off of these images, before realizing that 6.111 provided a similar script on the website. These .coe files store an uncompressed bitmap of the image they represent, in 8-bit (3/3/2 r/g/b) color. In their synthesized ROMs, the address input is {ycoord,xcoord}, where each combination of input bits corresponds to exactly one pixel of the image, and the output is 8-bit pixel. The ROMs were configured with their specific .coe files, then synthesized in an out-of-context module so they would not have to be rebuilt every single time.

The obstacles had multiple measures added to minimize the amount of space they take up in the implementation. One measure that greatly decreased their size was storing the entire pipe as a single 64x32 pixel image (Figure 21). This contains the head of the pipe, and a small section of the straight part. The straight part of the image repeats the same line of colors for every row, so with a little logic sprite module can just reuse the last line and use a small fraction of the pipe. Additionally, the top sprite is just the mirror of the bottom sprite, so another bit of logic cut the memory requirement in half again. Finally, a single synthesized ROM could be used for all three possible obstacles. Although they can all be on the screen at once, they will never overlap, so we can use our current hcount/vcount to determine which obstacle sprite to grant access to the memory.



Figure 21: Stored Section of Pipe Sprite

The high score logic needs to convert a binary score into an hexadecimal score to display for the player. We can do this by using a special mapping file to convert a 4-bit number to a pixel mapping for display. I made a custom character map where each hex number 0-F gets an 8-bit wide section of the mapping image. It takes either the high or low 4 bits of the score, scales them, and adds them to the address to get the desired character for displaying. It shares the same character map file between the two digits in a similar way to how the pipe file is shared between three sprites.



Figure 22: Number Map in Action

The scrolling background comes from a stored sprite, but it is not displayed in a 1:1 ratio. Instead, it divides the the hcount and vcount by 4, increasing the size of the pixels but cutting the amount of memory used to 1/16. Not only does this save memory, but it also mirrors the aesthetic of the original Flappy Bird game. Saving memory is crucial, as the sprites all need to fit in the ROM of the FPGA in order to get single-cycle access for VGA display. Even with the pixel quadrupling, the sprite was still undersized, so it was designed to loop continuously horizontally. Vertically, the top pixel (blue sky) is extended upwards, similar to how the pipe sprites worked. For the scrolling itself, the vsync clock is divided, and the background shifts by one pixel for every 16 frames. The main location is shifted by one display pixel, rather than one image pixel, creating a smooth flowing effect (4 subdivision per image pixel) rather than a block jumping effect.



Figure 23: Background Sprite

The face sprite is a special case. Unlike everything else, it does not come from a generated IP ROM. Instead, it looks up pixels in a RAM that is used by the camera. The vision module updates it with the image every frame, so it will always contain the latest camera image. The sprite pipeline has access to the x and y coordinates of the player from the vision tracking, so it uses these to extract the sprite around the center of the player's face. I also added a scale factor parameter so we could choose how zoomed in/out the view of the player's face is.

In Figure 24, you can see the final product of the gameplay screen. The section of the camera image around the player's face is extracted and zoomed to create the game sprite. Multiple pipes appear on screen at once, using the same ROM. The background uses pixel quadrupling to save memory and create a similar pixelated background effect to the original Flappy Bird game.

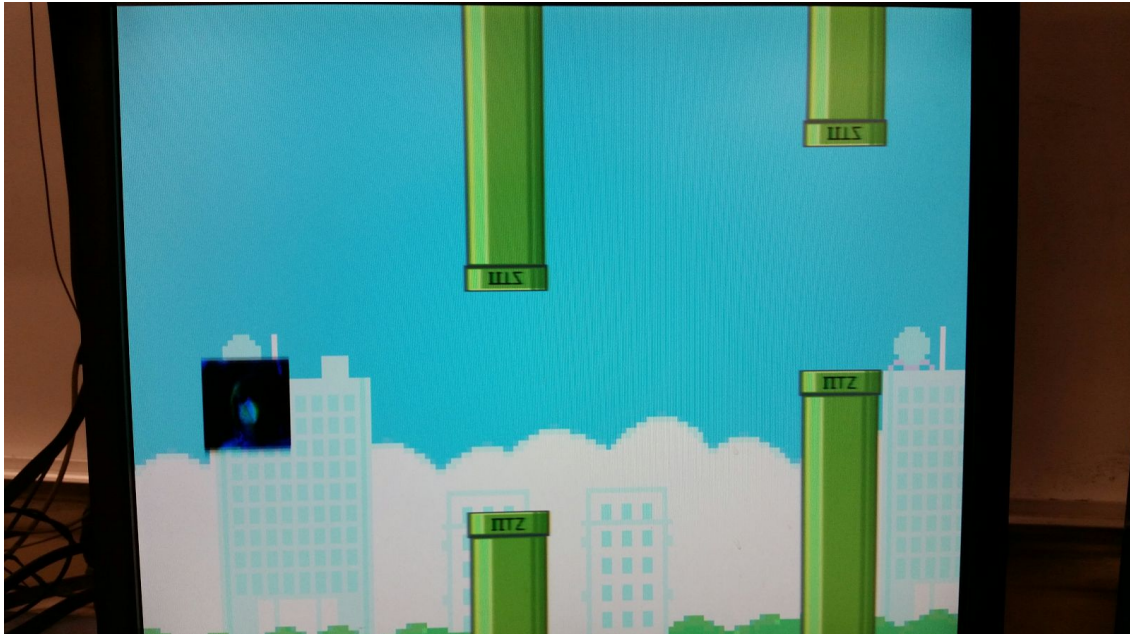


Figure 24: The Gameplay Screen

The Process

Initially, the plan was to first implement this module with simple shapes, and then later add the sprite textures. However, I wanted to figure out how generated IP ROMs work, and ended up getting it working with those first. Once the key elements of the sprite pipeline were completed, I could test each element by wiring the coordinates to the switches on the Nexys, then move them around to see if it responds as expected.

One element I had trouble with in this section was the getting the .coe files for the ROMs generated. Unfortunately I didn't realize there was already a Matlab script posted on the course website, so I spent a lot of time trying to get a different one working. This one converted PNG files to 8-bit .coe pixel maps. However, I eventually discovered that it only worked with pixel dimensions that went in powers of two. This was frustrating at first, but was easy to work around as most of the sprites went by powers of two anyway.

One frustrating problem to be noted for future projects is getting the vga to properly synchronize. At one point, I forgot to take into account the `at_display_area` bit. This bit, generated by the vga module, is 1 whenever the `hcount/vcount` corresponds to a valid part of the display, and 0 otherwise. If it is not used, and colors are output for the entire frame, it will not sync properly with the monitor, and positions/colors will shift in a seemingly random fashion. This was very frustrating to debug, and would be a good thing to keep in mind while debugging VGA signals.

3.4 Audio (nmccooy)

The audio block controls the audible components of game actions. In this case, we decided to implement jumping and crashing, as these are the two sounds from the original Flappy Bird game. In order to make this modules as simple as possible to the outside, taking on two bits to control. Each bit represents one of the sounds, and the module will play the sound entirely if the input is asserted for at least one clock cycle. The sound files themselves are stored in generated IP ROMs, and the current values of each sound are added together and sent to the DAC for output. The block diagram for this module is in Figure 25.

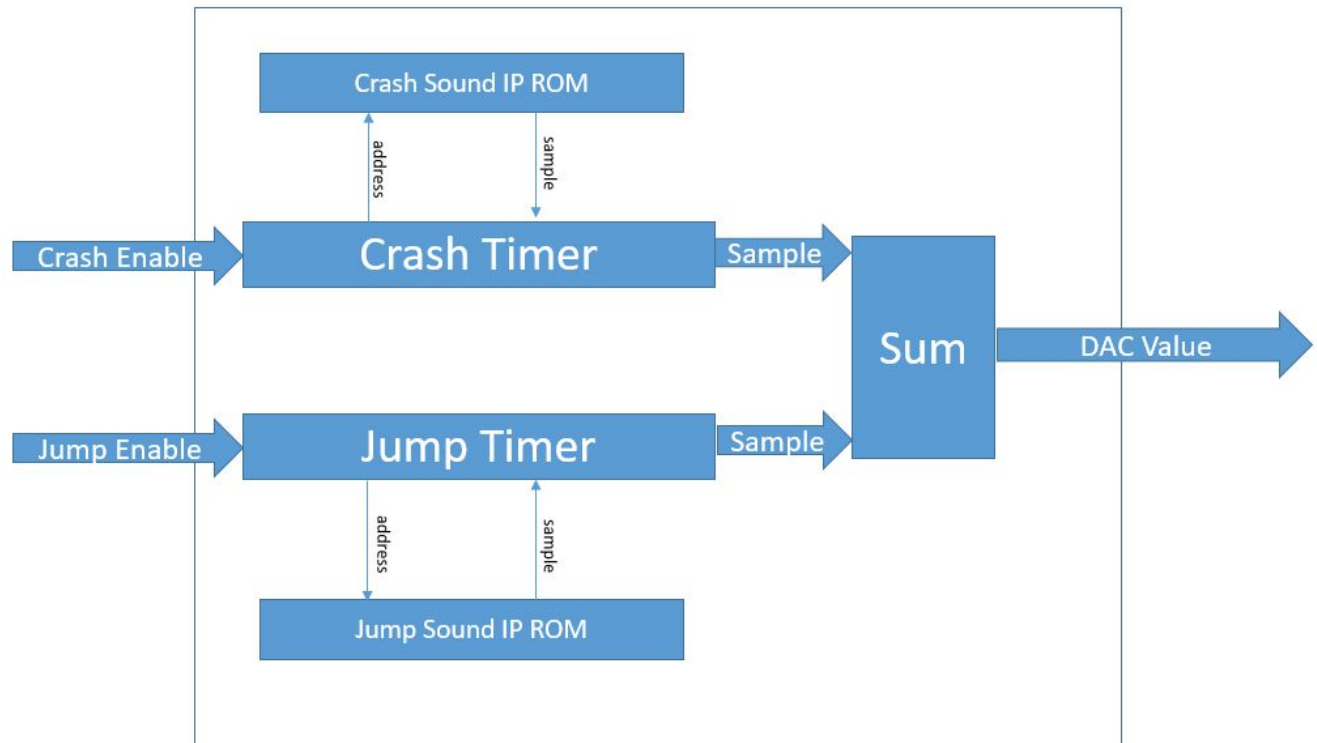


Figure 25: Audio Module Block Diagram

This module presented another challenge for optimizing memory usage. We did not want to use the SD card in order to avoid unnecessary complexity, as we knew we could fit everything in hardware if we optimized. I created some sound effects using existing sound samples, and experimented with them in Audacity. I realized that I could lower the sample rate of the sounds to 8khz and the bit depth to 4 bits, while still having them sound as desired. This greatly reduced the amount of space they take up in the ROM. I used the given Matlab script to convert these sounds to .coe files, and created ROMs for them where the address is the index of the sample and the output is the value for the DAC.

A timer module was created to interface with these ROMs. The module is effectively a wrapper for the ROM file itself. It takes in a parameter for the length of the sound file (in samples), and plays through it once when the input is asserted. A clock divider generates an 8khz clock that is shared between these two modules for progressing through the samples.

The Process

This section can easily be tested independently from the others, as it requires only two bits of inputs. The two enable bits were mapped to buttons on the Nexys so the sounds could be played on demand. I started with a simple sine-wave file loaded into the ROM. After confirming that the logic worked, the sine waves were replaced with the actual sound effects. Integration of this module with the others was fairly straightforward, as the interface is not complicated.

4 Conclusion

Implementing this game in hardware was an enjoyable project for all of us. It paired a fun final product with a challenging implementation, an ideal combination for a project in this class. The work was modular and divided well between three people, so we could optimize the division of labor. Our goals were achievable, and we accomplished what we wanted. Overall, this project helped all of us gain experience working with hardware implementation and interfacing with the outside world.

5 Resources

Besides the provided Nexys 4 FPGA board, OV7670 camera, and lab station complete with logic analyzer, we used Github to regulate version control since three people worked in parallel for this project. Additionally, our object of high contrast is in the form of a solid red conical party hat acquired from Amazon.

6 Citations

Nguyen, Lan Anh. "Exclusive: Flappy Bird Creator Dong Nguyen Says App 'Gone Forever' Because It Was 'An Addictive Product.'" *Forbes*. Forbes Magazine, 11 Feb. 2014. Web. 10 Nov. 2015.

7 Appendix

This section contains a listing of the Verilog files and modules generated for this project

- I. fpglappy.v (all)
- II. vision.v (jmend)
- III. gamelogic.v (weilow)
- IV. spriteline.v (nmccoy)
- V. audio.v (nmccoy)
- VI. gamelogic_tb.v (weilow)
- VII. vga.v (jmend - Lab 4)

Various .coe files used for audio and video ROM initialization (nmccoy)