Flying Pegasus Ground Attack

6.111 Final Project Report

Yini Qi | Tania Yu

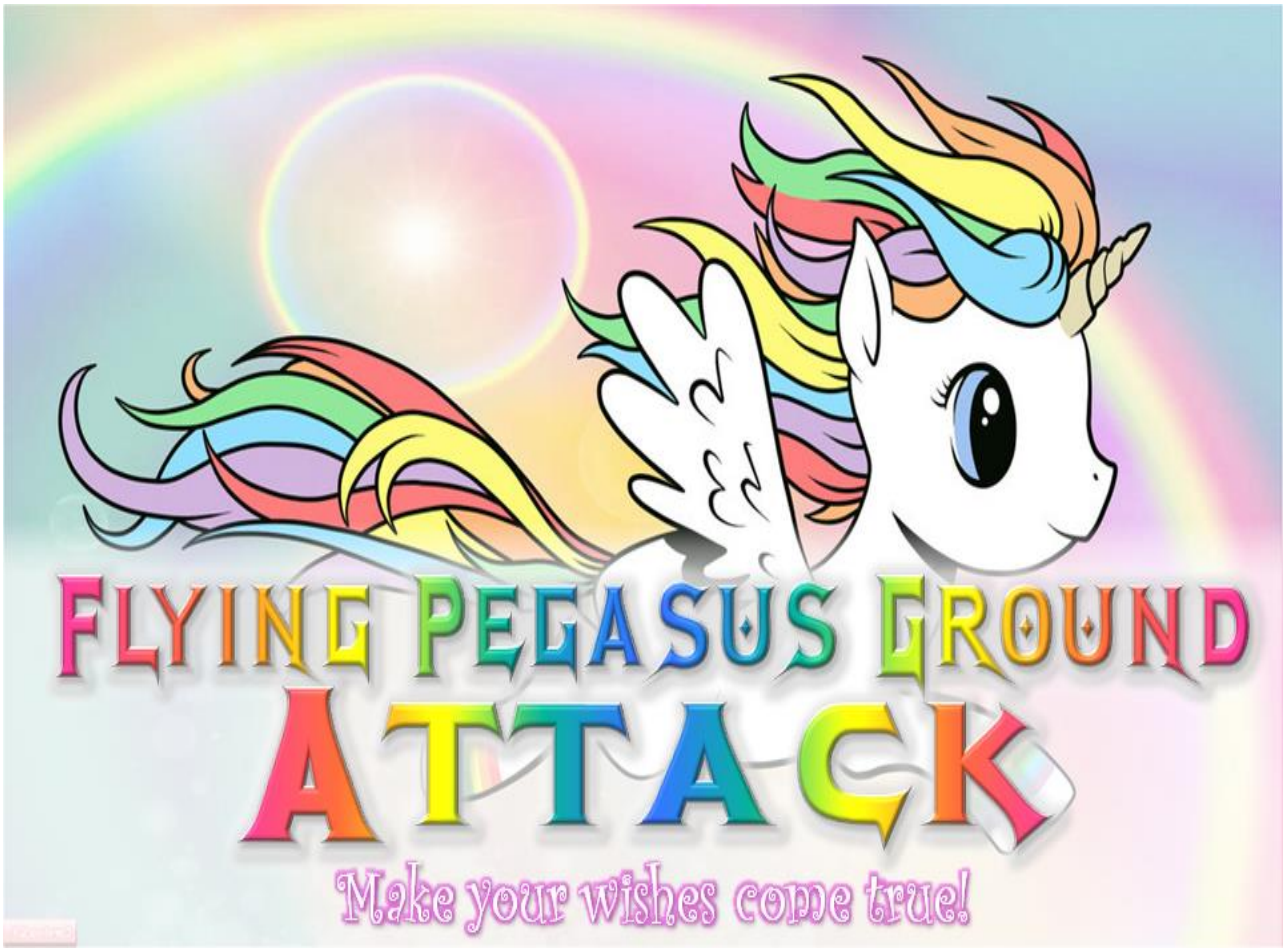

Table of Contents

1 Introduction

Robot Unicorn Attack is an “endless running” flash game in which the user controls the movement of a unicorn through space. The object of the game is to prolong gameplay without falling off the stage, crashing into platforms, or colliding with obstacles. In the original game, movements are controlled by keyboard inputs. This project aims to bring the classic game to life using a camera to detect a player’s motion to perform the corresponding actions in the game. Thus, the need for keyboard inputs is removed and the game relies solely on motion tracking. The player flaps her arms up and down to fly a Pegasus sprite, and the speed at which the flapping motion is performed controls its height. Additionally, the player can put her hands together in the center of her body to perform an “attacking” action on the obstacles to destroy them. These movements are predetermined and are the only recognized movements in the game. The player uses a remote with IR light emission to allow the camera to track the hand movements. The camera data is sent to the game logic in the FPGA on the Nexys 4 board to determine the state of the game and control the movement of the Pegasus. The basic implementation includes sprite images of the Pegasus and obstacles along with sound effects, including the traditional background music of Robot Unicorn Attack (Always by Erasure). The stretch goals included dynamically changing background images and complex obstacle movement, although these were ultimately not implemented.

2 Overview

The system consists of four major components: motion recognition, game logic, graphics rendering, and sound effects.

When a player turns on the FPGA, an opening splash screen displays on the monitor. Players will use a remote emitting an IR light as control. Once the game starts, the motion recognition module filters the light from the video stream, and determines the coordinates of the center of mass. Depending on the type of motion detected after a series of frames, the module sends the information of the state (flying or attacking) to the game logic. If the player is flying, the module calculates the speed at which the player flaps with the remote. If the player is attacking, the module returns the attack signal to the game logic.

All sprite graphics, including ones for the Pegasus and obstacles, are loaded into memory at the start of the game. During gameplay, these sprites are loaded from the appropriate memory blocks and painted pixel by pixel on each frame. In the basic implementation, sprites representing ground blocks and any other dynamically generated images in the background will also be pre-loaded in this fashion, but as one of the stretch goals, a large and complex background image will be loaded from SD card for display during the game.

The game logic also controls the interactions between the Pegasus and the game environment. Sound effects are included during the attacks and object destruction. Of course, no Robot Unicorn Game is complete without the theme song “Always” by Erasure, so the song is played in the background continuously on loop.

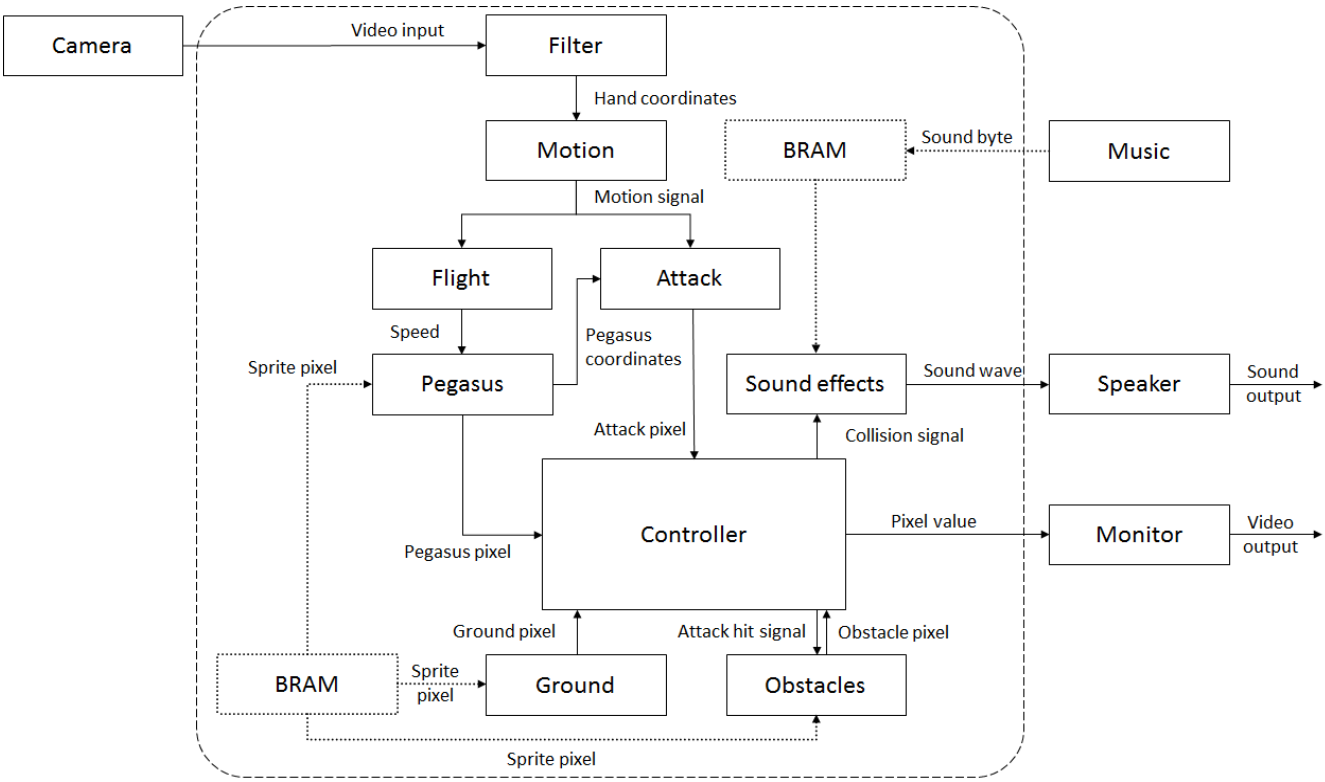

Figure 1: High-level block diagram

3 Design Decisions and Motivation

Setup

Figure 2 shows the hardware setup. The Nexys4 is mounted on top of the lab bench and taped to the logic analyzer to give the camera additional height so that it is shoulder level with a person of average height. The OV-7670 VGA camera is attached to the JA and JB pins and taped to the oscilloscope for support. The innards of a floppy disk covers the camera lens to act as an IR filter, which is described in further detail in section 4.1.1.

The speaker is attached to the audioPWM jack that is described in section 4.4.

Figure 2: Nexys4 board and camera setup

4 Implementation

The game components were divided such that Tania primarily worked on the game logic and graphics, and Yini worked on the motion tracking and sound modules.

4.1 Motion Tracking (Yini)

4.1.1 Camera

The OV-7670 camera used for motion tracking was more difficult to interface with than expected. The goal of the motion tracking module was to store incoming video data into BRAM, and then read it out to display to VGA. There existed a module to configure and read from the camera, but it was not well documented and we spent a long time debugging the various pin connections as it turned out that there was only one pin that could accept the camera clock. It is important to note that the pclk of the camera can only be connected to the JB[7] pin on the board.

The biggest challenge was storing and reading the video stream to BRAM. There was no sample code for this task and it was difficult to test because reading and writing must happen at the same time, so we did not know if it was the reading or writing that was the problem. When the camera output could finally be displayed, it was large nonsense due to BRAM addressing errors. The camera's video stream was 640x480 pixels at 16 bits each, but the BRAM only had 4,800Kbits of storage, so we needed to downsample the incoming video to 320x240. As a result, we only took every other pixel in every other row in each frame, and displayed it in the upper left quadrant (320x240) of the screen (Figure 3).

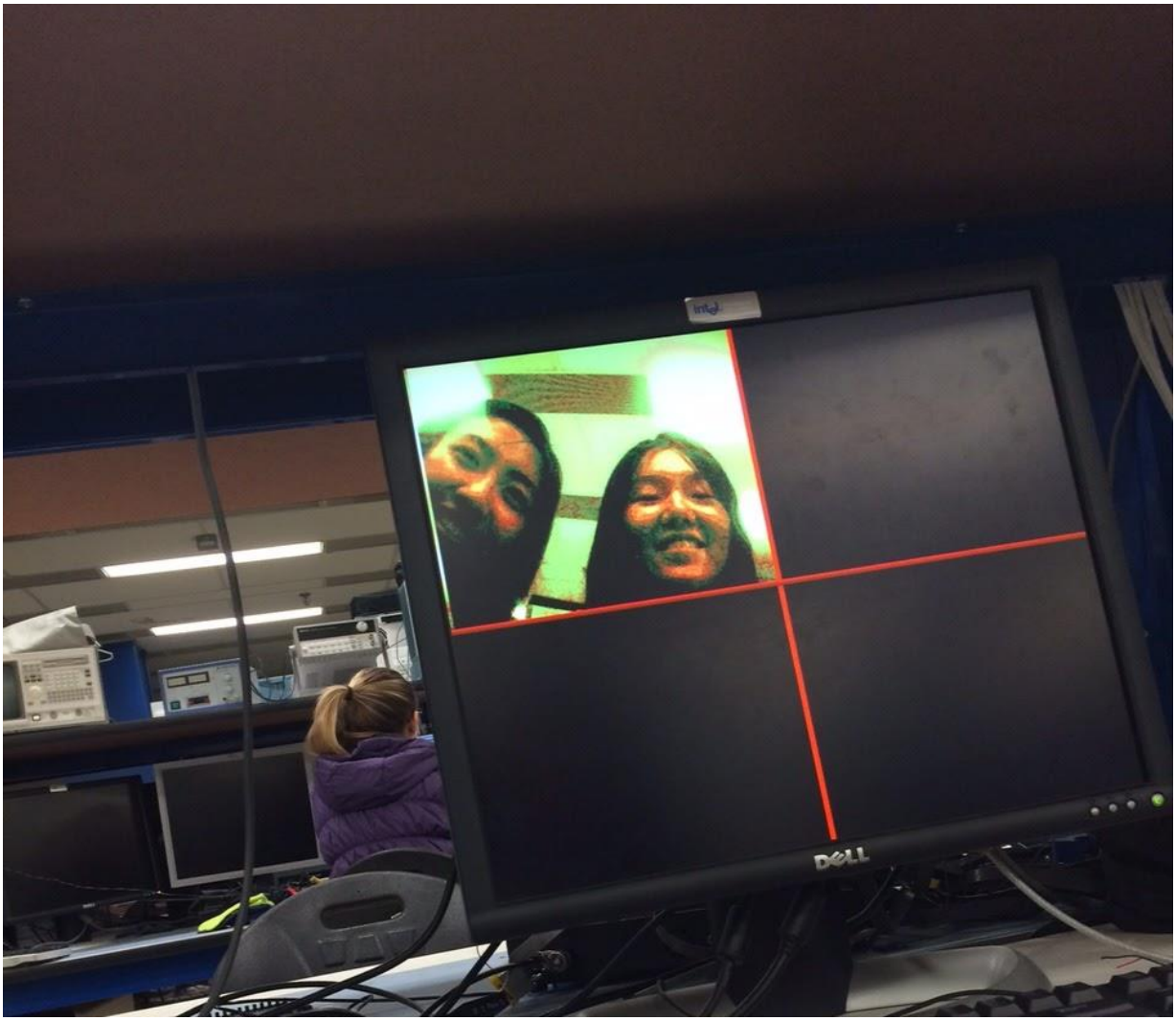

Figure 3: Camera output with 16-bit pixels

After properly displaying the camera video stream, we realized that the given configuration file did not support different colors very well. There existed a greenish tint over the whole frame so we could not accurately detect red and green (or any other color) gloves like we originally planned. Since the configuration registers were not documented at all, we decided to use a filter with an IR light instead of spending more time testing different configurations. We used a floppy

disk as the filter which worked well in blocking out everything except the IR light. Later, we realized that by using the light we did not need 16 bits of color, so we downscaled the video data again to only store 8 bits and save BRAM space which we used for sprites and sound (Figure 4).

Figure 4: Camera output with 8-bit pixels

4.1.2 Light Filtering and Tracking

To track the location of the IR light, the rest of the frame was filtered out. The light showed up on the camera as a white-ish color, so we used the RGB2HSV module to convert each RGB pixel to its HSV values in order to widen the available range of color detection. Since the background was a solid color with the filter on, the HSV values of the light pixels were much different from the background, making it easy to filter out the light. Then, we displayed the filtered light on the upper right quadrant of the screen for debugging purposes (Figure 5).

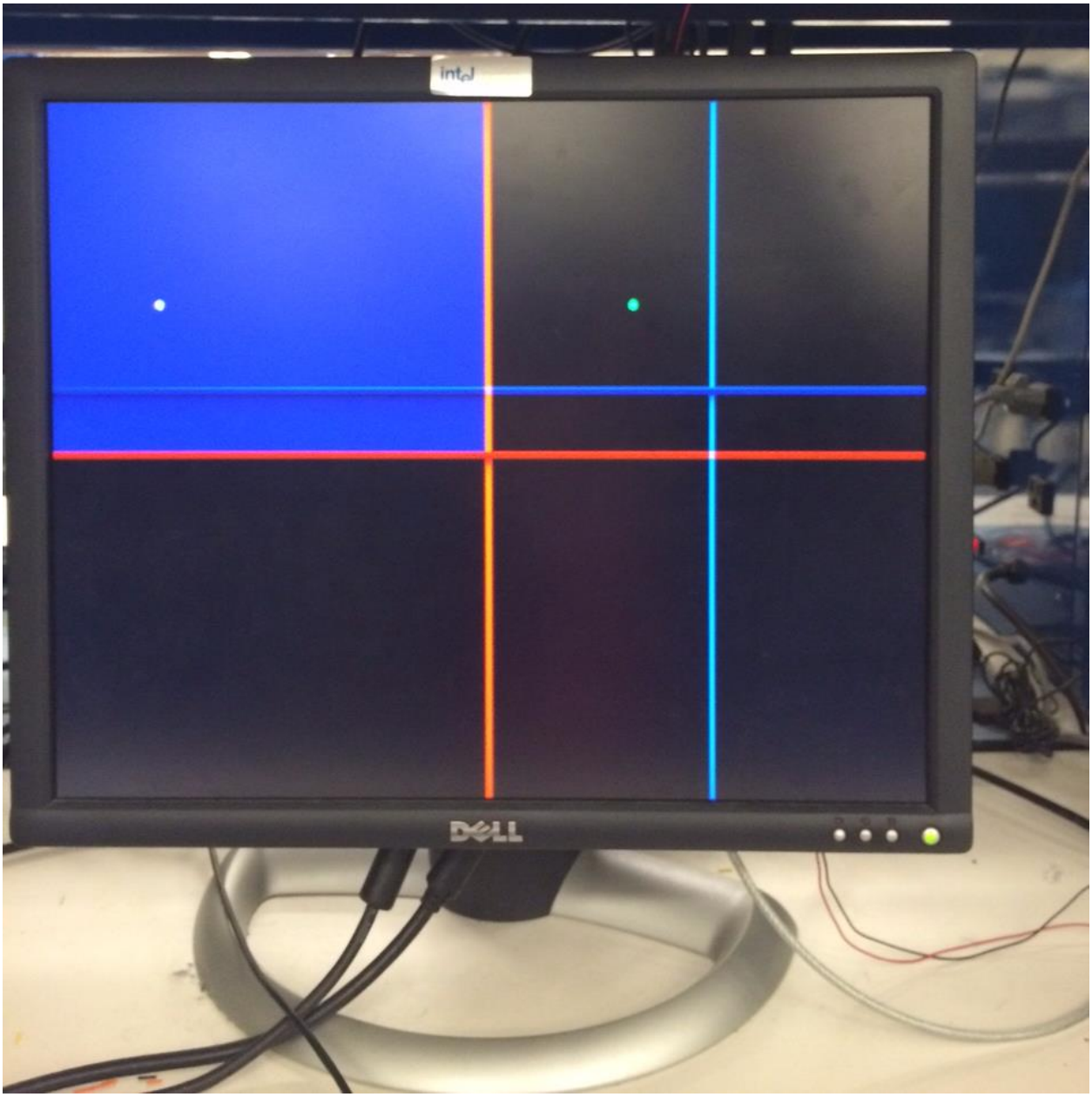

Figure 5: IR light-filtered camera output

4.1.3 Flight

We set a threshold on the bottom 1/5th of the screen, so that a flight signal would be activated whenever the player moves the remote below that threshold, imitating a flapping motion. The flight signal pulses for one cycle and does not turn on again until the player moves the remote back above the threshold.

4.1.4 Attack

We set the attack threshold in the middle of the screen, so that when the player moves the remote to the left side it will send an attack signal. The attack signal pulses for one cycle and does not turn on again until the player moves the remote back to the right side.

4.2 Game Logic (Tania)

Prior to integration with the motion tracking, the fundamental aspects of gameplay were developed such that the player could control sprite behavior using button and switch controls on the Nexys4 board. This required sufficient modularization of the game logic so that integration involved only connecting the output signals from motion tracking modules to the input signals of the game logic modules.

4.2.1 Physics

The physics of the game mimic real life behavior, but with different parameter values. A constant gravitational acceleration acts upon the Pegasus, providing a 1 pixel increase in downward velocity every eighth of a second. Taking the downward direction to be positive displacement, this is equivalent to an acceleration of 8 pixels/s^2. Each time the physics module receives a flight signal pulse, it is as if the Pegasus "flaps" once, so each pulse triggers an upward thrust that increases upward velocity by 2 pixels, or an instantaneous acceleration of -2 pixels/s^2.

After some testing, a "terminal velocity" of 3 pixels/frame, or 180 pixels/s with the 60Hz refresh rate was implemented. Without this limit, the Pegasus would fall extremely quickly and be nearly impossible to control. This also seemed to be a more realistic world, since objects also do not accelerate infinitely on earth. We arrived at all of these velocity and acceleration values through trial and error, since we wanted the game to be challenging but not impossible to play.

4.2.2 Controller

On every clock cycle, the controller performs calculations on the potential pixel values at a particular (hcount, vcount) coordinate. It receives these pixel values as input signals from all distinct game objects, which it uses to detect collisions and signal specific modules to react correspondingly. In particular, the controller determines when the Pegasus dies and when it successfully attacks an obstacle.

The Pegasus can die in one of two ways: falling through gaps in the ground or flying into an obstacle. Falling to death can be detected simply when the Pegasus sprite reaches the lowest row of pixels on the screen. An overlap of the Pegasus sprite and any one of the obstacle sprites would also trigger death. In either case, the controller module outputs a game-over signal in order to switch the display to the splash screen.

The other key collision is between the attack beam and an obstacle. The controller itself does not know where any of the obstacles or located and relies on the general obstacles module to provide the appropriate obstacle pixel values. Therefore, when a collision is detected, the module outputs a signal indicating this hit. It also outputs the identity of the obstacle that has been attacked, which it receives as an input signal on the same clock cycle.

4.2.3 Game Objects

Pegasus
The Pegasus is the object directly controlled by the player during the game. It moves only vertically on the screen, naturally falling due to gravity and sometimes flying upwards as a result of the player's flapping motions. Additionally, it can "see" when it is flying above a block of ground versus over a gap, i.e. it receives an indicator input signal, which allows it to land on top of the ground. This block was implemented simply with a basic blob, with the variant of displaying pixels from a sprite image rather than a uniform color.

Ground
Throughout the game, the ground moves 1 pixel to the left every frame refresh, or 60 pixels per second, and loops continuously over the screen. The block is shorter than the full 1024-pixel width of the screen, so gaps are effectively created between pieces of the ground. This module also determines whether the ground lies "below" the Pegasus, which is possible because the Pegasus object has a fixed x-axis location. This signal is outputted to Pegasus, as mentioned above. The implementation of this block required a small variation on the original blob module, since it loops around continuously.

Obstacles
The obstacles module holds a set of nine blobs, each representing an individual obstacle. These blobs are positioned in a 3x3 array that moves horizontally across the screen and scrolls in a similar manner as the ground does. Whenever a blob enters the display on the right side, it is randomly "turned on" or "off". If it is enabled, it becomes visible on the screen and can be interacted with, whether by colliding with the Pegasus or by being attacked. On the other hand, if it is disabled, it will not appear for the duration of the loop. In order to implement this enabling and disabling feature, however, instead of looping directly to the right edge of the screen, the array loops over a total width of 1536 pixels. This provides a buffer zone where the blobs can be

reset and prepared for the next loop. These blobs are another variation on the original blob module, with the added features of looping (like the ground) and being turned on or off.

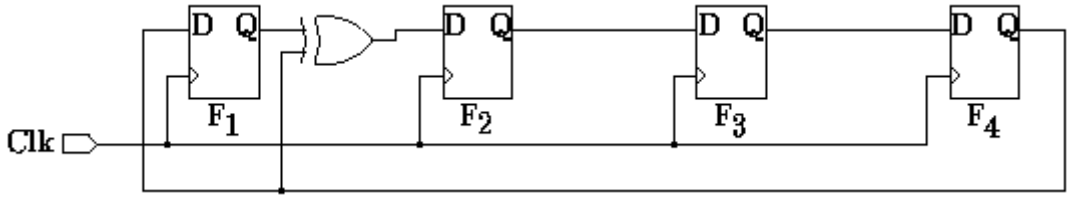

Figure 6: 4-bit linear-feedback shift register

For the random enabling of obstacles, we originally hoped to use the random number generating function built into Verilog, but soon realized that it can only be used in testbenches. Therefore, we implemented a pseudo-random number generator using a linear-feedback shift register (LFSR) instead. As seen in Figure 6, this 4-bit LFSR uses four flip-flops to propagate the bits and an XOR between the first and last bit values to pseudo-randomize. With a seed of 4'b1001, the LFSR cycles through all 16 possible values before repeating itself. Since we wanted to enable at most one of the three obstacles in a column of the array each loop, we needed only the lower two bits to determine which obstacle would be enabled each time. We could have implemented a higher bit LFSR for longer cycles, but we felt that 16 was sufficiently large that players would not recognize or remember the pattern of obstacle appearances.

4.3 Graphics (Tania/Yini)

Images were generated to replace the generic squares for game objects. JPG files were converted into COE files with RGB values with the MATLAB script from the course website. The images we incorporated were used in displaying the splash screen as well as sprites for the Pegasus and obstacles.

4.3.1 Splash Screen

The splash screen was displayed upon game start (Figure 7). Whenever the player dies by crashing into an obstacle or falling through the ground, the splash screen would display again until the game is reset. The original image we wanted to use was the poster we created for our presentation, which was at 1280x720 pixels with 16 bit color. Similarly to the camera video, the full image was too large to fit into BRAM. Thus, we reduced it to a 320x240 pixel image with 8 bits. However, we wanted to display the splash screen across the entire monitor, so we replicated each pixel four times, duplicating every other pixel and every row.

Figure 7: 640x480 upscaled splash screen

4.3.2 Sprites

The original "blobs" for the Pegasus and obstacle objects were replaced with sprites (Figure 8). All of the images were 8 bit color, with the Pegasus sprite at 100x64 pixel resolution and the obstacles at 150x100 pixel resolution. We wanted the obstacles to be larger than the Pegasus to pose a threat but not impossible to avoid.

We encountered a problem with drawing the sprites in that the left column of the pixels would be wrong due to the delay in reading from BRAM. To compensate for this delay, we delayed the read address calculation by five cycles because there was a two cycle delay each in reading the image pixel and corresponding color from the color table, and one cycle delay in determining whether that pixel should be on for the given hcount.

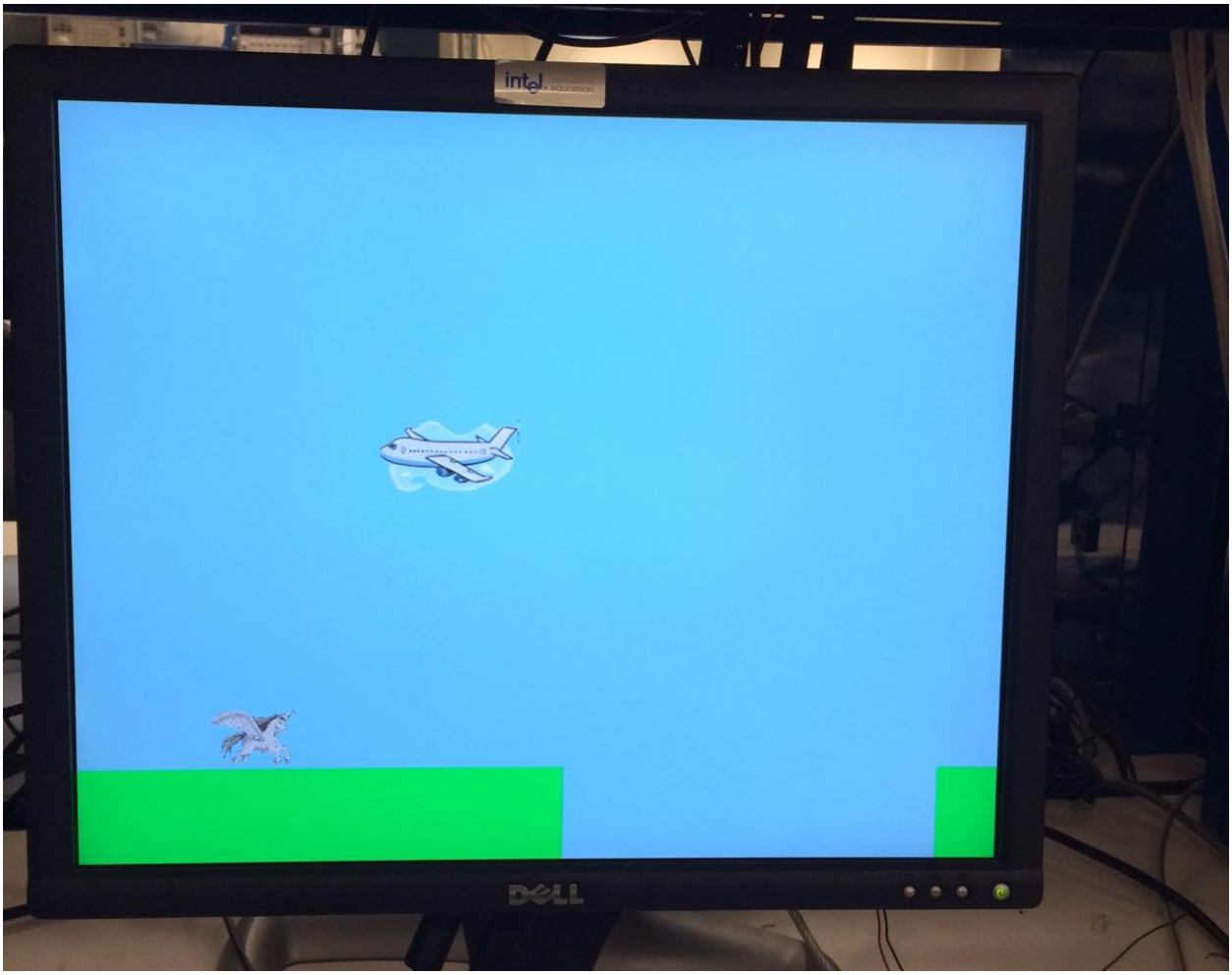

Figure 8: Gameplay with sprite images for Pegasus and obstacle objects

Another crucial issue with the sprites was memory allocation. Initially, we read the obstacle sprites from BRAM in the blob module, which needed to be instantiated once for each obstacle for a total of nine. As a result, Vivado created nine instances of the sprite BRAM even though they all stored the same data, causing the memory to overflow. We solved this issue by instantiating the BRAM read in the main obstacles module and developing logic to choose which address to read from. Each blob is responsible only for calculating the appropriate read address and determining whether it is present and enabled at the given pixel location. The obstacles module instead uses the output signals from all nine blobs to read and output the appropriate image pixel from memory. Thus, we were able to reduce memory usage to one BRAM per unique sprite like we originally planned and fit everything into memory, as seen in Figure 9.

Content	Percentage (%)
Camera video	14.07
30-second background music	47
Laser sound effect	4.81
Splash screen	14.07
Sprites	7.77
Total	**87.72**

Figure 9: BRAM allocation

4.4 Sound (Yini)

4.4.1 Background Music

Throughout the game, the audio module plays the background song from the Robot Unicorn Attack game -- “Always” by Erasure. Originally we wanted to use the SD card to store the whole song, but the SD controller module was quite complex so we decided to first play audio through BRAM, and use the SD card as a stretch goal. Since we did not have enough BRAM left to store the whole song, we downsampled the audio to 8KHz at 8 bits, which allowed us to store and play a 30 second clip of the chorus. We outputted the audio through the PWM jack on the Nexys4, and used the sample audioPWM module on the course website. Unlike the camera, this module was easy to use and we were able to get the audio playing quickly. However, there was the problem of noise overlaying the audio clip that we could not solve. We first theorized that the noise was due to the downsampling at 8KHz, but even after playing a test clip with the original 32KHz rate at 16 bits, the noise was still there. Ultimately, we did not have time to implement the stretch goal with the SD card, so the 30 second clip from BRAM was maintained.

4.4.2 Attack Sound Effect

Whenever the attack signal is sent to the sound effect module, a laser sounds. The sound effect lasts approximately 1.2 seconds, and is overlaid on top of the background music. We took the music data read out from BRAM from the sound effect and the music data read out from the background music and added the signals together. The signal is then sent to the audioPWM module which plays the total combined music signal. If another attack pulse is activated before the first laser sound finished player, the laser would restart from the beginning.

5 Review and Recommendation

We experienced many issues related to external devices such as the OV-7670 camera and sound outputs of the Nexys4 board. Since the Nexys4 board is relatively new, there did not exist much support for interfacing with these devices and as a result we spent a lot more time getting the basic camera to work than expected. The camera interfacing in particular took over a week longer than planned, which was a crucial hindrance to our project. Because of this delay and the bad camera quality, we had to simplify some of the modules in our initial design, such as using a remote with an IR light instead of gloves which would have provided a more realistic flying experience. We also changed the attack module to calculate signals based on one hand's movement instead of both hands.

In hindsight, we would have chosen to use the labkit instead as it provides more support for basic video and audio modules so we could have spent more time with more complex game logic instead. We originally chose the Nexys4 because it had an SD card slot, and one of our stretch goals was to include large color background images and audio. However, we did not end up implementing our stretch goal so the labkit's memory would have sufficed for the tasks we needed. There is also existing infrastructure for using the labkit's ZBT memory, which is much larger than the Nexys' BRAM.

6 Conclusion

The Flying Pegasus Ground Attack game successfully transformed the Robot Unicorn Attack game to a live interactive version. We faced many challenges including interfacing with the camera, noise in background music, and synchronizing between various BRAM reads. We were able to solve most of these challenges, save for the noise in background music. If we were to revisit the project, we would much like to figure out the audio problem, and find a way to play decent quality music. Although there are certainly improvements to be made, the current version of the game is fun and more challenging than the original.