Termanator
Design Document

Donald Eng
Rodrigo Ipince
Kevin Luu

Mentoring TA:
Brian Schmidt

December 12, 2007

Abstract

This project consists on the design and implementation of a first-person shooting game with a little twist. The player will be taking the role of the "Termanator," with the mission to kill as many creatures as possible. As the game progresses, creatures will be coming at the player at increasing speeds, incrementing the game's difficulty. To stay alive, the player must destroy each creature by shooting a constant stream of projectiles. Such stream is controlled by the Termanator through a specialized gun which uses a pointer and a shaker to respectively determine the target and the intensity of the stream. If a creature gets too close to the player, the game is over, and the score is recorded. Ultimately, the objective of the game is to obtain the highest score, for the purpose of eternal fame.

The system will consist of four main components: the devices, digital inputs, game engine, and graphics. The inputs are used to generate the power and direction of the stream, so that the game engine is able to calculate the respective game events. The game engine will incorporate all the core functionality of the game itself and generate a structured output for the graphics component, enabling the creatures and the stream to be displayed onto the screen.

Contents

1 Overview **4**

2 Component Descriptions **8**
- 2.1 Shaker . . . 8
  - 2.1.1 Shaker - mechanical device . . . 8
  - 2.1.2 Pulse Generator . . . 9
  - 2.1.3 Power Calculator . . . 9
  - 2.1.4 Power Bar Sprite . . . 9
- 2.2 Video Processing . . . 10
  - 2.2.1 Camera . . . 10
  - 2.2.2 Clock buffer . . . 11
  - 2.2.3 NTSC to ZBT . . . 11
  - 2.2.4 ZBT . . . 12
  - 2.2.5 Video Display . . . 12
- 2.3 Pixel Detection . . . 12
  - 2.3.1 Color Space Converter . . . 12
  - 2.3.2 Frame Filter . . . 13
  - 2.3.3 Pointer Calculator . . . 14
- 2.4 Game Engine . . . 14
  - 2.4.1 Creatures . . . 14
  - 2.4.2 Projectiles . . . 16
  - 2.4.3 Game Logic . . . 17
- 2.5 Graphics Block . . . 19
  - 2.5.1 Perspective Transformer . . . 19
  - 2.5.2 Sprites and Sprite Scaling . . . 21
  - 2.5.3 Sprite Display Pipeline . . . 21

3 Testing and Debugging **23**

4 Conclusion **26**

List of Figures

1 Screenshot of Nintendo Wii's Rayman Raving Rabbid . . . 4
2 Screenshot of Termanator. . . . 5
3 High level organization of Termanator. . . . 6
4 Screenshot of video processing with frame filter shown in green pixels. 7
5 Block diagram of shaker block. . . . 8
6 Anatomy of the Shaker device. . . . 9
7 Block diagram of video processing block. . . . 10
8 Video input connection to FPGA . . . 11
9 Block diagram of creatures module. . . . 15
10 Block diagram of projectiles module. . . . 16
11 Block diagram of game logic module. . . . 18

12 MATLAB-generated image of projectile perspective. 19
13 CAD representation of projectiles. 20
14 Scale factor, given a specific z (shown in x-axis above). 20
15 Representation of pipeline implementation for sprite layering. 22
16 Screenshot of XVGA display in HSV. 24
17 Screenshot of XVGA display, incorporating the pointer detection 25

List of Tables

1 Table of derivation for iterations 25

Acknowledgements

All accomplishments achieved for this project would not have been possible without the consistent help from the staff and fellow students of 6.111.

First, we would like to thank Professor Terman for his dedication to the class and the countless amounts of effort he invested into it. From the concepts to demonstrations, his lectures piqued our interest in the subject, and allowed us to let our imaginations run wild.

Secondly, Gim Hom was phenomenal in the amount of time he spent in the lab. His expertise in digital systems and quickness of thought allowed us to bounce many ideas off of him.

Also, we would like to thank the 6.111 TAs: Alessandro Yamhure, Roberto Carli, and Brian Schmidt. Each and every one of them were extremely helpful in every possible way, from staying late to give us more working time to enlightening us with their experiences of the class.

Finally, big thanks to the rest of the class for many days and nights of fun time. You were a great group of people to work with, and we enjoyed watching the accomplishments of your projects.

Also, thanks to Danny, Mike, and Timothy.

Overview

Over the past decades, video games have evolved drastically, from the Super Mario Bros. in the early 1990's to the high-resolution Halo 3 from Microsoft. Video games are enabling users to have a more interactive experience. For example, the recent Nintendo Wii offers a sensor-technology remote control to give the user a feel of what it is like to be in the actual game. In fact, the video game Rayman Raving Rabbid, shown in Figure 1, inspired this project to develop a game that adds a physical component to the feel.

Figure 1: Screenshot of Nintendo Wii's Rayman Raving Rabbid (taken from `http://ps2.ign.com/articles/721/721470p1.html`).

Termanator, shown in Figure 2, is an interactive 3-dimensional shooting game that provides the user a similar experience to the Nintendo Wii's Rayman Raving Rabbid. The basic structure of the game consists on creatures that will be coming towards the front of the screen, where the user stands. If a creature gets too close to the user, the game is over, and an appropriate message will be displayed based on how long the user lasts. To stay in the game, the user must destroy each creature with a special gun (the user's input).

In the game, the gun will generate a stream of projectiles, targeted at the creatures coming at the user. The gun consists of two devices, a pointer and a shaker, that are used to control such stream. The pointer determines the point of origin of the projectiles, while the shaker provides the corresponding power level of the gun; the higher the power level, the farther back the projectiles can travel. Since the pointer determines the initial position of a projectile, the user can aim high and shoot over a particular creature if he is targeting a creature in far sight. One note is that in order for the gun to operate at its full potential, the shaker needs to be constantly in motion, because the weapon power decreases constantly.

This project involves the design and implementations of four primary components: Devices, Inputs, Game Engine, and Graphics Engine. Figure 3 provides a brief overview of each block and their contributions to the overall game.

The two necessary hardware inputs to the game are the shaker device and the

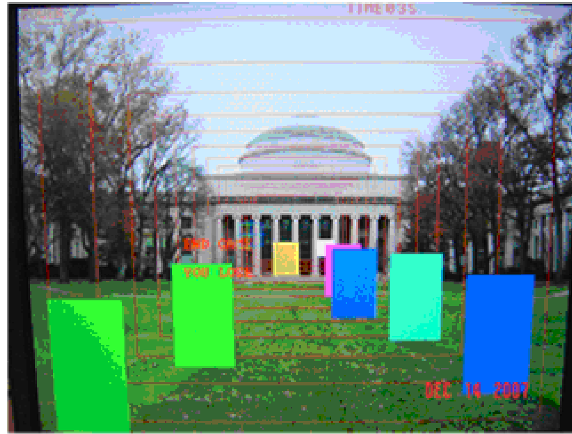

Figure 2: Screenshot of Termanator.

camera. The shaker device is used to determine the power that the user is generating for his gun in the game. By the speed that the user is shaking the device at, a power level is sent to the Game Engine, which will use that value to determine how far the stream of projectiles will travel in the z direction. Here, the z-coordinate encodes the depth in the screen; the higher the z-coordinate of an object, the farther back it is on the screen. If the user stops shaking the device at any given point during the game, the power level will decrease at a constant rate. This lowers the distance that the stream of projectiles will travel and, ultimately, constrains the user from being able to reach creatures at all. With this input added into the game, a physical aspect is incorporated, giving the user a more unique experience compared to traditional shooting entertainments.

The camera is used to detect the position of the pointer, which is a green tennis ball, held by the user. The position of the pointer in a 1024×768 XVGA display coordinate system is determined for each frame in the video display. Through color recognition, the pointer is detected by a defined threshold in a particular color space. For all pixels that pass the threshold test, the x and y position are accumulated, and the position of the pointer is taken to be the center of mass of these pixel positions. In addition to determining all pixels that pass the threshold test at every frame, a frame filter, shown in green in Figure 4, is incorporated to eliminate possible noises, allowing objects with similar color components to be near the pointer, but not be detected in the position calculation. After determining the center of mass for the pointer, a visual representation of the pointer is superimposed onto the game, allowing the user to see where they are aiming during game play.

The Game Engine is responsible for generating all the digital signals needed to represent the game abstractly. That is, it must be able to determine when the game starts, when it ends, and how the game evolves through time. The functionality of the component can be broken down into three main parts: creature motion, projectile motion, and game logic. The main task of this component amounts to creat-

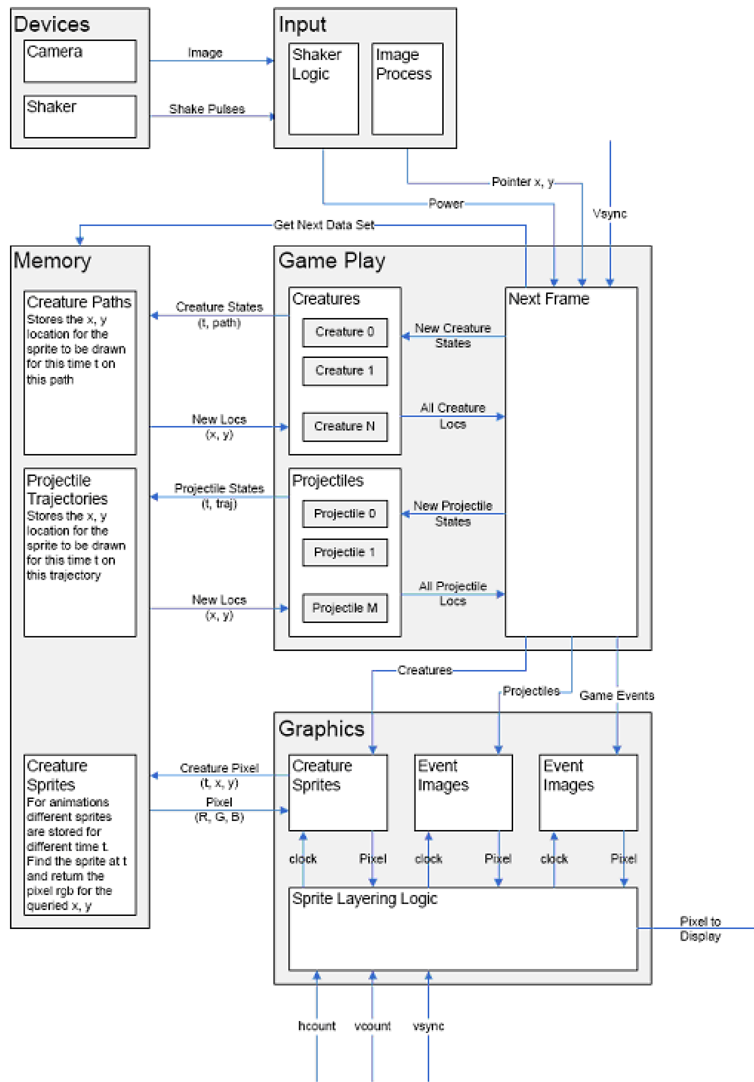

Figure 3: High level organization of Termanator.

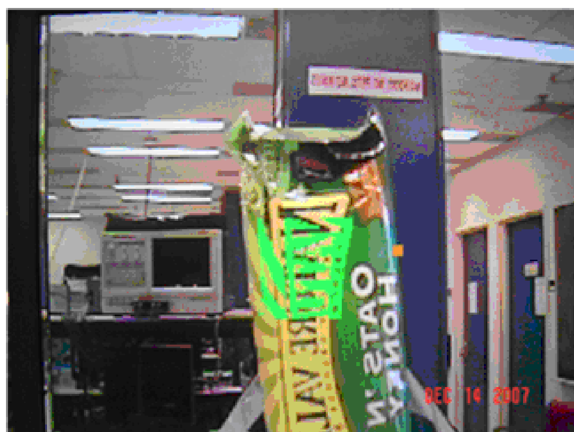

Figure 4: Screenshot of video processing with frame filter shown in green pixels.

ing a good model for the generation of creature and projectile positions. Given the 3-dimensional nature of the game, each position is represented with three coordinates, and the task of transforming this 3-dimensional model into a 2-dimensional one with the desired perspective is left to the graphics component. With this model, each object in the game has three coordinates describing its position, together with a frame field, which indicates how far along a path it is currently at.

Another essential component of the game is the Graphics Engine, which is responsible for displaying the game sprites appropriately, providing a unique interface for the game. The engine needs the x, y, z, and `frame` fields of each object in the game. The game exists in a square rectangular prism. Looking into the square, the x spans the horizontal axis and the y spans the vertical axis. This square would be the screen. The z dimension shrinks back into the screen and from a 2D perspective can be thought of as scaled concentric squares. Creatures and projectiles move into the screen and out of the background along the z dimension. Hence, projectiles have a constant x coordinate, a constant velocity in z (both determined when fired), and fall in y according to a parabolic path. Because `frame`s can be considered as units of time, each consecutive `frame` will propagate the object forward in time. Therefore, at every new `frame` (at every new z coordinate of an object), it is determined at which scaled concentric square of z the object is in. Then, scaling is performed on the x and y coordinates of the object at that instant in time by a scale factor associated with the current z, and the resulting x, y coordinates are normalized to the center of the screen. The new normalized $x' = nx$, $y' = ny$, are then the new sprite locations on the screen. Next, traditional sprite displaying is used with lookup tables for sequences of sprites at specific frames and sprite layering is implemented to handle depth.

Component Descriptions

Detailed descriptions of each of the four main components of the game are presented in this section.

2.1 Shaker

The Shaker device is used to provide power for the gun to shoot down creatures in the game. With the game being presented in 3-dimensional space, creatures that are farther away from the user require more power from the gun to produce a farther projectile. In other words, small amount of power limits the gun to reach creatures of short distance, while generating much power would allow the user to reach creatures that are farther away from the user. Power is generated by the user through how vigorously the user shakes the Shaker device. A slow shake would produce a gun with weak projectile, while a powerful shake gives the user the ability to shoot down more creatures. Figure 5 describes the general layout of each module that is necessary in the shaker block.

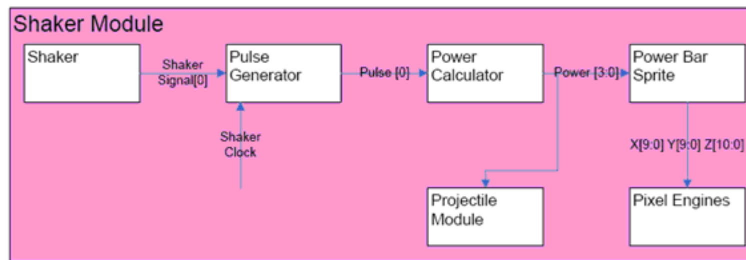

Figure 5: Block diagram of shaker block.

2.1.1 Shaker - mechanical device

The Shaker is a mechanical device constructed by the team. It consists of a cylindrical tube, with a level switch mounted to the bottom (see Figure 6). The Shaker device is designed to use a constructed Lego block that presses the level switch when contacts are made. The Shaker device is has two leads, one connected to the voltage signal and another connected to the user input of the FPGA. If the level switch is pressed, connection is made between the voltage signal and the user input, generating a level signal. However, another module, the Pulse Generator, converts the received level signal into a pulse. Because of this, the user needs to keep pressing the level switch to generate pulses which correspond to the power level of the gun in the game.

Figure 6: Anatomy of the Shaker device.

2.1.2 Pulse Generator - `posedge_detector.v`

With the physical shaker device sending level signals, this module is responsible for converting those signals into pulses. At every positive edge of the clock, the output signal is an AND combination between the current signal and the inverse of the last signal stored into a register. The result is a signal whose duration is only one clock cycle (1/65Mhz). Once the pulse signal is generated, it will be sent to the power calculator.

2.1.3 Power Calculator - `shaker.v`

The power calculator module computes the power value that the Game Engine will use to determine the z velocity of the projectiles. For every pulse that is received from the pulse generator, a predetermined value (5) is added to the ongoing sum that resembles the power value. Additionally, for every fraction of a second, the power value decrements by 1 to provide the effect that the power dissipates if the user is not shaking the device during game play. This design is analogous to a sink, where there is a constant rate of drainage, but the water level varies depending on the rate of water flowing into the sink. Once the power value is determined, it is sent to the Game Engine to be incorporated into the projectiles.

2.1.4 Power Bar Sprite - `power_sprite.v`

In addition to sending the power value to the Game Engine, the numerical figure is also used for a visual representation of the power level during the game play. A power bar sprite, whose width corresponding to the power level generated by the user, is displayed on the upper-left corner of the XVGA display. With a higher power value, a longer sprite will be displayed, and vice versa. Since the power value is only 8 bits wide, the power bar sprite has a finite width, and will not continue to grow even if the user keeps shaking the device vigorously.

2.2 Video Processing

The video processing block is an important component to the project. Its primary function is to take live feed from a camera and identify the presence and location of a designated marker, which the team decided it to be a green tennis ball, through color detection. Additionally, filtering techniques have been applied to improve the recognition of the marker used as a pointing device. After running the resulting collection of pixels through data processing, the output will be coordinate positions representing the location of the marker on the XVGA display. Figure 7 describes a general layout of individual modules and its contribution to the video processing block.

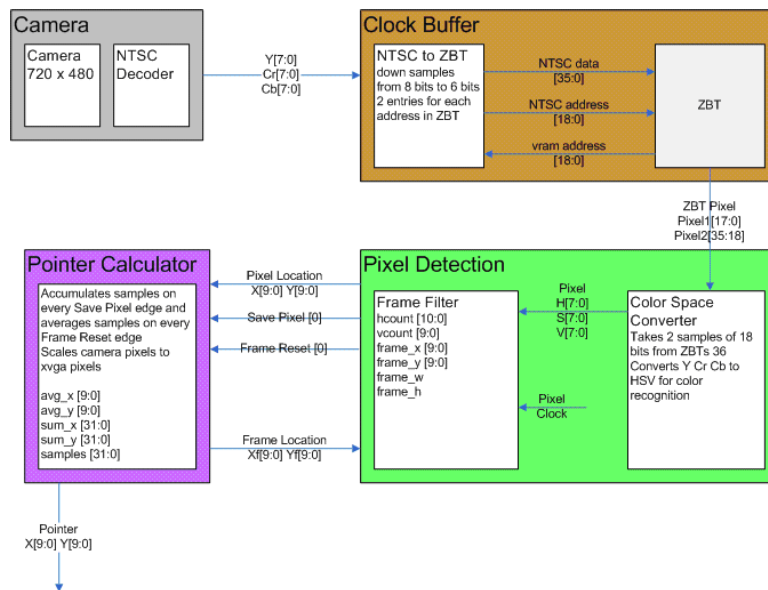

Figure 7: Block diagram of video processing block.

2.2.1 Camera - `video_decorder.v` (borrowed from 6.111 website)

The source of the video processing block is a camera based on NTSC-standard, provided courtesy of the 6.111 staff. Conveniently, the FPGA has a built-in composite video input port, as pictured in Figure 8. The camera sends input NTSC video data to the RCA phono jack on the right-hand side of the FPGA. From the camera, an interlaced NTSC signal is received. NTSC, an analog television standard in North America and several Asian countries, produces approximately 30 interlaced video frames per second, a rate sufficient in producing what appears to be a continuous motion capture for the human eye. The camera outputs an image that is 720 pixels wide and 480 pixels tall, at a clock rate of 60 Hz.

The module responsible for converting camera data into pixels to be displayed

Figure 8: Video input connection to FPGA (taken from `://web.mit.edu/6.111/www/f2006/projects/jburnham_Project_Final_Report.pdf`).

on the XVGA is the NTSC decoder. This module, originally written by Javier Castro in the Fall 2005, takes a stream of LLC data from the ADV7185 NTSC/PAL video decoder and generates 24-bits pixels, which are encoded within the stream, in YCrCb format. No modification was made to this particular module.

2.2.2 Clock buffer

The clock buffer block is responsible for synchronizing other blocks that run with different clock rate. With the camera running at 27 Mhz and the XVGA display running at 65 Mhz, the clock buffer centralizes both blocks so that the entire unit is running synchronously. This particular block consists of two primary modules: the NTSC to ZBT module and the ZBT module.

2.2.3 NTSC to ZBT - `ntsc2zbt.v` (modified from 6.111 website)

The basic functionality of this module was constructed by Professor Chuang. This module prepares the NTSC data to be loaded into the ZBT RAM for video display. Some modifications were made to efficiently utilize the fixed dimensions of the ZBT and to improve the resolution of the video images.

The data input is changed from an 8-bit value to 24-bit, representing the 24-bit YCrCb value. Originally, the data input was an 8-bit luminance, used to display a black and white image. In an effort to produce an image in YCrCb color space, the data input must increase its size to accommodate the two other color components. Since the output from the NTSC decoder is 30-bit, only the most significant 8 bits for each color components are necessary (in this case, `ycrcb[29:22]` for `Y`, `ycrcb[19:12]` for `Cr`, `ycrcb[9:2]` for `Cb`). Ultimately, all three components of

this particular color space will be used for color detection.

In addition to the change in the storage capacity of data input, the NTSC to ZBT module down-samples each 8-bit color into 6-bit value, total of 18-bit for the each pixel. This technique is used to efficiently utilize the space provided by the ZBT. Since each ZBT address is 36 bits long, 2 entries of 18-bit pixels can be stored into each address. In the end, the NTSC to ZBT modules output a 36-bit data for 2 pixel entries, along with 19-bit address, to the ZBT memory module.

2.2.4 ZBT - `zbt_6111.v` (borrowed from 6.111 website)

The ZBT RAM is the memory of choice used in the video processing block. This module is implemented by Professor Chuang as a simple ZBT driver. The ZBT memories have two cycle latencies on read and write, and need a longer data hold times around the positive edge of the clock to work properly. The ZBT module takes in five inputs: the system clock, the clock enable for gating ZBT cycles, the write enable, the 19-bit memory address, and finally, the 36-bit data to write (in this case, 2 entries of adjacent pixels). The data to write can be presented and clocked in immediately, while the actual writing to RAM will occur two cycles later. Unlike write, read requests are processed immediately, but the read data is not available until two cycles after the initial request. The primary output of this module is the read data, which are the two stored pixels of the current address. For the implementation of this project, no modifications have been made to this module.

2.2.5 Video Display - `vram_display.v` (modified from 6.111 website)

This module is responsible for generating display pixels from reading the ZBT RAM. This module is modified in response to the changes in resolution that is to be displayed. It is set to have the same corresponding number of bits with the NTSC to ZBT module, so that the number of pixels matches correctly.

2.3 Pixel Detection

The output of the camera is in YCrCb, a family of color space that is frequently used in video and digital photography systems. It utilizes luminance and chrominance to display its output. By knowing the YCrCb value of the marker that needs to be detected, it is possible to create filters with derived thresholds, using them to filter only pixels that correspond to the marker. With this, the center of mass of all pixels accumulated can be calculated, and can be used as the designated location of the marker in the game.

2.3.1 Color Space Converter - `ycrcb2rgb.v`, `rgb2hsv.v`

There have been some suggestions of working in a different color space, such as RGB, for easier implementation of the color threshold in the filters. When comparing the two different color spaces, RGB is a more intuitive standard, where a larger proportion of people are familiar with RGB than YCrCb.

With this in mind, an initial module developed by Xilinx is used to convert a 30-bit YCrCb value into a 24-bit RGB value. With constants hardcoded into the conversion, this particular module utilizes combination of YCrCb to produce the color scheme that most people are familiar with.

Much experimenting and testing has been done to detect pixels using the three specified color spaces. From the results of the team's trials, it appears that RGB color scheme performed poorly. From this, the team generalized that low robustness to change in the environment is a primary source of error. Because the RGB color space cannot differentiate brightness and color, changes in the lighting of the surround environment can dramatically impact the accuracy of the color detection. In other words, if the color value of the marker increases or decreases, the brightness will simultaneously change as well. Detection using YCrCb color space would alleviate this issue, but significant noises are still present in various circumstances based on the lighting of the surrounding environment and the angle that the camera is pointed. Testing has shown that certain procedures, such as blurring the camera so that it is out of focus, or tilting the camera to a particular position, would help in eliminating extraneous noises. Ultimately, using YCrCb color space would limit the robustness of the video processing block.

With this in mind, the team decided to experiment with HSV, which is an attempt to describe perpetual color relationships more accurately than RGB, while remaining computationally simple. HSV, representing hue, saturation, and value, describes colors as points in a cylinder, whose central axis range from black on the bottom to white on the top. Compared to the other two color spaces, HSV offers many advantages, but from the testing of the project, HSV is able to eliminate all noises in the background that could potentially affect the accuracy of the pointer. All in all, the HSV is the color space of choice in detecting pixels of the pointer.

2.3.2 Frame Filer - `pixel_detection.v`,`frame_filter.v`

The objective of the frame filter block is to find pixels corresponding to the color properties of the marker and eliminate any possible noises that may affect the calculation of the center of mass from these pixels. This particular block consists of two primary modules: the `pixel_detection.v` and `frame_filter.v`.

The first module, `pixel_detection.v`, is implemented to detect the marker in the HSV color space. This detection algorithm searches for pixels that falls within a predefined threshold combining H, S, and V. The threshold is determined through multiple trials in an effort to find the optimal value to use in detecting the depicted marker, the green ball. If a particular pixel fits within the defined threshold of HSV, its value will be replaced with a color value corresponding to white. Otherwise, a black pixel will be sent.

The second module, `frame_filter.v`, determines whether the detected pixel from the first module falls within the small window that resembles the location of the pointer. At the press of a button, the module's refresh window on the screen will detect any pixels that match the designated threshold levels for each color components. This module checks to see if the pixels detected falls within a window whose dimensions are given as inputs to the module. If the pixel is within the window, a

signal will be sent to the position calculator module to accumulate the x-position and y-position of that particular pixel. If the pixel detected is not located within the window, it will be discarded. When the entire XVGA screen is finished refreshing, a `frame_reset` signal will be sent, signaling the `position_calculator` module to compute the average x-position and y-position for the collection of pixels. The coordinate of that pixel will be the new center for the next frame filter cycle.

2.3.3 Pointer Calculator - `position_calculator.v`

At every frame reset, the module will begin accumulating the x-position and y-position of pixels that falls within the refresh window. If a `save_pixel` signal is received from the `frame_filter` module, this module will accumulate the corresponding x-position and y-position. On the other hand, the pointer calculator will discard coordinates that belong to pixels outside of the refresh window. Once the frame is completed refreshing, the pointer calculator will compute the average x-position and y-position and send the value as the determined location of the pointer off to the Game Engine block. At the same time, the same value will be sent back to the `frame_filter` block, and that figure will be the new position of the updated window used in the module. Through this procedure, the refresh window will be at the approximate location of the pointer at the next frame cycle. One assumption made here is that the pointer will not be physically moved at a rate faster than the update of the frame.

2.4 Game Engine - `game_engine.v`

The Game Engine block is the primary portion of the game; it is responsible for generating the signals needed to represent the game in an abstract manner. Below are descriptions of the three major attributes of the engine: the creatures, the projectiles, and the game logic.

2.4.1 Creatures - `creature.v`, `creatures.v`, `rng.v`

To simplify the control of the creature movements on the screen, there are only four allowable paths for the creatures to walk down through. Once a creature starts off in a path, it will stay in the same path, getting closer and closer to the front of the screen as time progresses. Also, to simplify the problem of spawning and destroying creatures, a set number of creatures will be used at all times (8). Each of these creatures can be in an active or inactive mode, which will determine whether they are displayed on the screen or not. Therefore, simply making these creatures loop around the paths (i.e., relocating each creature to the beginning of the path once it reaches the end) will achieve the desired effect. To avoid making the game monotonous, every time a creature reaches the end of the path, the path it will take during its next iteration will be determined randomly. Figure 9 shows a high level block diagram of the creatures component.

Basically, this module maintains information with each creature's current path, position, and frame (determines how far along the path is the creature at). It also

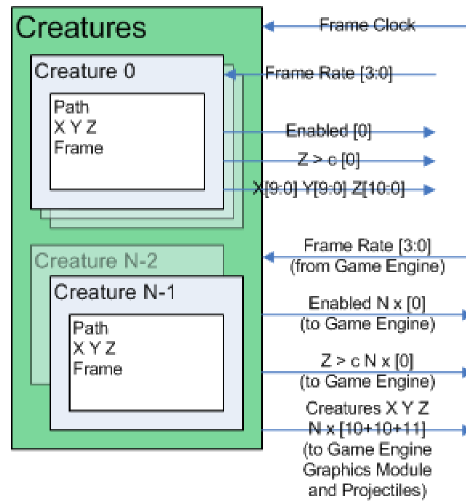

Figure 9: Block diagram of creatures module.

sends out a couple of signals that the game logic module needs. One of them is a signal that goes high whenever a creature is close to the front, and the other one is a pulse that indicates when a creatures is restarting on a new path.

This module was implemented by first dealing with a single creature (in `creature.v`), and then constructing many instances of these creatures, having them start evenly spaced along their paths. Since the creatures are just moving linearly in the z axis, it is quite easy and fast to determine their position. For any creature, its x coordinate (length on the screen, with the origin in the bottom left corner) is completely determined by the path in which it's in, its y coordinate is always 0, and its z coordinate (depth) is simply a linear function of its frame.

The team chose the z axis to span from 0 to 2047, and after some testing, it was determined that in order for the game to be playable, the creatures needed to move as slow as one z-unit per frame when starting the game. Thus, it was decided that the creature frame should also be able to go from 0 to 2047, and then the z location would just be determined by subtracting the frame from 2047. Next, the frame rate range was also calculated after some testing, and it is allowed to go from 1 (at the beginning of the game) to 15, at which point the creatures are moving so fast that it would be pratically impossible to beat the game.

These frame rates had to be taken into consideration when generating the other two signals that go to the game logic module. These are generated by simply comparing the z coordinate of the creature with some thresholds. These thresholds have to be large enough so that they will not be missed when the creatures are moving fast and loop around the paths. Basically, setting a window that is at least as large as the fastest speed a creature can achieve, guarantees that the signal will not be skipped.

Finally, choosing a new path for a creature was done via a random number generator. This was implemented in a very simple way. There are two counters, a large

one and a small one, that keep counting indefinitely. The 'random' number is then taken to be one of the bits of the large counter, when indexed by the small one. Of course, this will not exhibit truly random behavior, because both counts are periodic, but since the creatures are increasing their speeds, the times at which random numbers will be needed will vary, so the 'randomness' provided by this method is good enough for our purposes. Nevertheless, this method was only used for debugging purposes. A better and simpler method was used for the actual game. After seeing how much the pointer location flickered, we decided to use the lower order bits of the x and y locations of the pointer to code for the new random number (2 bits). This seemed to work out pretty good and the creatures seemed to appear in the paths in a truly random fashion.

2.4.2 Projectiles - `projectile.v`, `projectiles.v`

The projectiles are handled in the same general fashion as the creatures. There is a fixed number of projectiles (16), each of which can be active or inactive at any point in time. Again, this module maintains information about each projectile's position and frame. It also maintains an active or inactive state for each projectile, but it solely depends on whether the projectile has flown too high, too far, or if it has reached the floor. In other words, this module only concerns itself with projectile motion, and knows nothing about game logic. An high level block diagram of the projectiles module can be seen in Figure 10. Additionally, in this component, each projectile module takes in all the locations of all the creatures and generates an output which indicates if the projectile has collided with any creature, and if so, it specifies which creature it was.

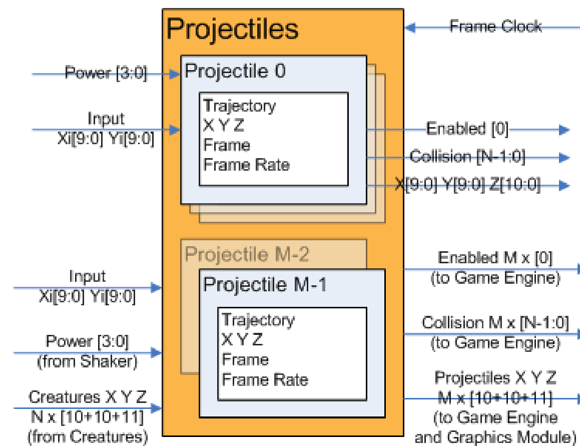

Figure 10: Block diagram of projectiles module.

The main difference with the creatures module is that projectiles have more than four possible trajectories, so their position is perhaps a bit harder to calculate. First, notice that the x coordinate will always remain the same, since it is assumed that

the projectiles only travel in one plane (that which is parallel to the plane spanned by the y and z axes). This x position is determined at the moment of launch, and is given by the x location of the pointer. Next, the y component can computed from the current frame with a simple kinematics equation:

$$y = y_{initial} + v_y f - \frac{g f^2}{2},$$

where f corresponds to the projectile frame, $y_{initial}$ is determined by the y location of the pointer at the time of launch, and v_y and g are constants that were determined (after a lot of testing) to be 7 and 1, respectively. The same can be done with the z component, as the projectile travels with constant velocity in this direction. The initial position in the z dimension is always 0, and the speed varies with the power given.

The trickiest part in this module was getting everything to fit together. First off, some parameters had to be estimated taking playability into consideration, such as the initial speed in the y direction. Once this was done, the maximum time of flight for a given projectile had to be calculated. With that in mind, it was necessary to have the frame be able to go high enough to enable to longest trajectory to be completed in time. The problem was that the more frames needed, the more spaced out the projectiles needed to be. This is not good, because the more spaced out the projectiles are, the less of a 'stream' effect we get, and it turned out that having 32 projectiles as initially planned was way too computationally intensive for the rest of the game (as the team found out with trial and error), so there was the added constraint of the number of projectiles. These issues were resolved by creating a solid testbed that allowed the tweaking of pretty much any parameter that was being used. After a lot of trial and error, parameters that offered the desired effects were found.

2.4.3 Game Logic - `game_logic.v`, `get_index.v`

This module is responsible for generating all the game event signals and determining how the game evolves. A high level block diagram of this module can be seen in Figure 11. The basic tasks of the game logic module can be summarized as the following.

Game start and pause The game logic modules needs to know when to start the game, and when to pause it. This is fairly simple to implement. A pulse that indicates when the game starts is used to reset all the registers to their initial status, including those in the creatures and projectiles modules. Next, the pause signal simply stops the game momentarily when it is high. To achieve this, the condition that the pause signal must be low was added to every `always` block in the entire game engine. With no problems, this achieved the desired functionality.

Creature frame rate The creature frame rate must increase as time marches on. Moreover, the game score is dependent on how long the use was able to stay

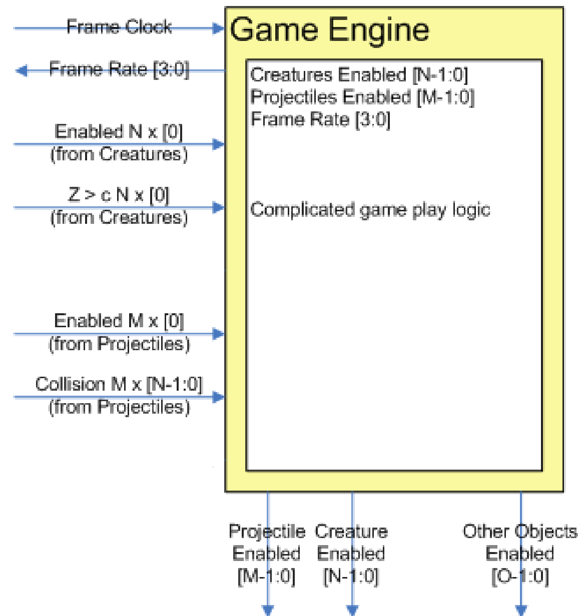

Figure 11: Block diagram of game logic module.

alive. Thus, the game logic module keeps a timer that starts every time a new game is started, and that only moves whenever the game is not paused. Using this simple timer, the creature frame rate is increased by 1 unit every time 16 seconds have elapsed. These numbers were again chosen empirically after playing a few games and determining what would be a good level for users to have fun.

Collisions This is perhaps the most important task of the game logic module, because it is used to determine which creatures and which projectiles will become inactive. The collisions bus coming from the projectiles module is used to determine which projectiles have collided with which creatures. This is done by passing the bus through an array of `get_index.v` modules, which determine which creature each projectile has hit. Once this is done, the game logic module just compares which of these possible collisions are actually valid, taking into consideration whether the creature *and* the projectile in question are currently active or not.

Game end Determine when the game ends is fairly easy. The game logic module simply looks at the close-to-front signal passed in from the creatures module. If any creature that is currently active is too close to the fron of the screen, then the game is over.

2.5 Graphics Block - `graphics03.v`

The graphics engine consists of two main components: the perspective transformer and the sprite pipeline. The Game Engine uses a 3D coordinate system to perform game logic, but the display uses a standard 1024×768 XVGA display, which is of course 2D. The perspective transformer is responsible for making the conversion from 3D to 2D to display sprites at the right location. In addition there is a perspective scaling algorithm that will properly scale the sprites according to their depth. The sprite pipeline takes these sprites and displays them in the proper layering according to their depths. Combining both of these modules is enough to create a pseudo 3D perspective representing the current state of the game.

2.5.1 Perspective Transformer - `3Dto2D_transformer.v`

Using the sprite's coordinates in 3D space, the perspective transformer displays this sprite at the correct 2D coordinate on the display. Rather than using a complicated matrix calculation a simple "tunnel perspective" is used with a point of origin located at the center of the screen and the objects radiating out. Depending on a sprite's z coordinate, its x and y will be normalized to the center by a specific factor. At depth 0 no normalization is required, but as z increases the x and y coordinates of the displayed sprite will pull in closer to the center of the screen. The result of this simple normalization is shown in the following plot produced in MATLAB. Each of the frames in blue represents a different coordinate in z. As illustrated, as z increases, the frame is shrinking but renormalized to the screen's center to produce the desired perspective.

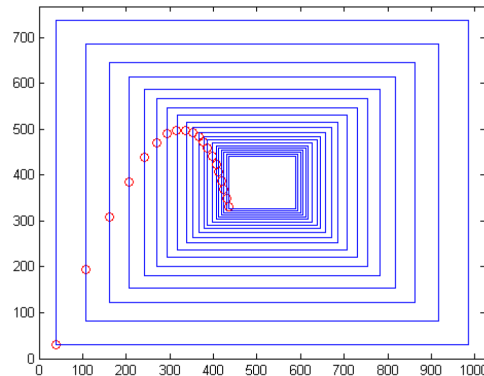

Figure 12: MATLAB-generated image of projectile perspective.

However, the scaling of the z coordinate is not linear. Data collection was performed on a sample image from a CAD program to determine at which rate images are scaled with increasing z. The image and the data collected are shown below.

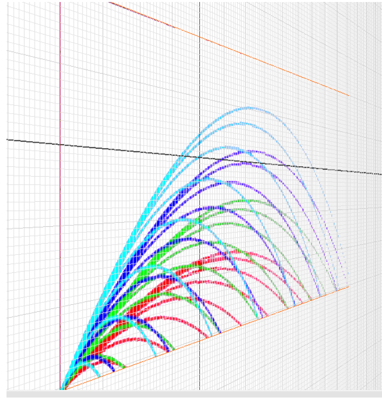

Figure 13: CAD representation of projectiles.

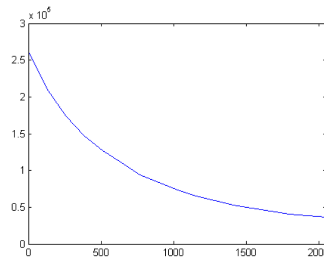

Figure 14: Scale factor, given a specific z (shown in x-axis above).

From this data the appropriate scaling ratios can be determined. Many methods were attempted to transfer the scaling relationship displayed in the graph into the Graphics Block. Originally, a polynomial fit was being used. However, the polynomial was of degree 4. A degree 2 polynomial was tried but produced undesirable results as significant deformation was observable. Piecewise linear functions were tried next. The linear approximations that needed to be made were becoming too many, and as a result linear interpolation was finally decided upon to estimate the scaling factors between data points. Not only is linear interpolation fast, but it preserves the original data producing the desired perspective transform.

The linear interpolation algorithm is as follows. Data was collected at some interval of z and the scale of z was stored for each interval. The data collected contained 17 data points resulting in 16 lines that connect the data. The intervals of z were such that the range of z was 0-2047. Now for a given input of z the scale factor for this input z can be determined from the linear interpolation of the line that this z is located on. The input z has the same range 0-2047. The most significant 4 bits were taken from the input z and used in a 4 bit case statement to determine which piece of the data set should be interpolated (recall that there are 16 lines that connect the data collected). Then the least significant 7 bits of z were used to inter-

polate the selected data. The result from the interpolation is then used to scale the x and y coordinate.

The FPGA requires a complicated dividing module if the scale ratios are used directly. An alternative to the dividing module is bit shifting. Hence the scale factor that is determined by z has a power of 2 maximum. Therefore, x and y can be multiplied by the scale factor of z and then bit shifted by the power of 2 maximum instead of divided. The scaled x and y are then normalized to the center of the screen to produce the 'concentric' frames for each z.

2.5.2 Sprites and Sprite Scaling - `game_sprites.v`

Sprites are stored in the FPGA's ROM. To generate a new sprite ROM, the *.bmp was analyzed by a simple MATLAB program and the RGB fields of the image were converted into binary to generate a *.coe file for this *.bmp. The *.coe file was then used to generate the sprite ROM. The ROM's width was the number of bits used to describe a pixel's RGB fields (8 bits for RGB makes a 24bit wide width). The number of addresses needed to be a minimum of the number of pixels in the image. Therefore the addresses needed to be $\lceil log(\text{height} \times \text{width}) \rceil$ bits wide. To save space, game sprites were mapped to a color space. In some instances only 5 colors were needed to describe the sprite, and then only 3 bits of color information needed to be stored for this sprite. The appropriate colors were then mapped in the verilog code keeping the ROM relatively small. For example, a sprite image might only contain the following colors: white, black, brown, red, and transparent. These colors are then mapped as follows: white = 000, black = 001, brown = 010, red = 011, and transparent = 100.

2.5.3 Sprite Display Pipeline - `single_pipe_4.v`, `sprite_pipeline_64.v`

The sprite pipeline is used to display the appropriate sprites at the right depth. It layers the sprites on top of one another to give the illusion of the 3D game. The algorithm for sprite layering depends on z and the sprite's transparency. For every pixel being displayed, all the sprites are queried for their pixel at that x y location. The sprite pipeline then determines which of these pixels is displayed by the layering convention. Pixels are prioritized on their z coordinate and on their transparency. A transparent pixel is always over written. But if two opaque pixels are being compared the closest opaque pixel is displayed. There were three pipeline stages and the end result is illustrated in the following diagram.

The above list shows the pixels being pipelined they are each located at the depth written on the pixel. The colored pixels are opaque and the white pixels are clear. In the first pipeline stage, all 16 pixels are evaluated in 4 groups. The "best" pixel for each group is contributed into the next stage of the pipeline. In the first group pixel A begins as the best because it is first. However, pixel 1 overwrites A as best because it is opaque. Pixel 4 is transparent and has not priority. And Pixel 6 is opaque but $1 < 6$ so pixel 1 is considered the best pixel for this grouping of 4 pixels. Pixel 1 is set into the next stage of the pipeline. The next stage of the pipeline performs the same evaluation and the very best pixel is then output 3 clocks behind. A simple image shift is

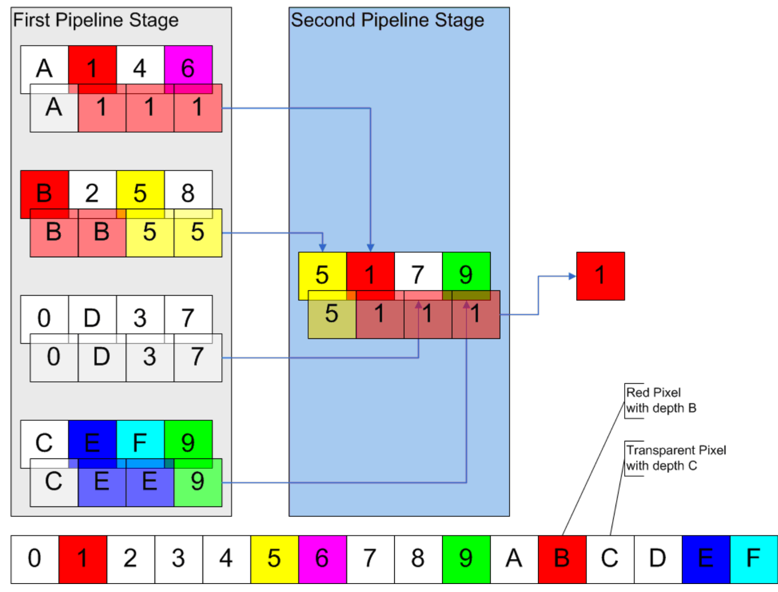

Figure 15: Representation of pipeline implementation for sprite layering.

then implemented to shift the entire output back 3 pixels. This is implemented with a sprites' clock that is jumped 3 edges ahead.

Testing and Debugging

To effectively test each and every component of the game, a debugging testbed is needed for each of the major blocks. The shaker will be displaying the corresponding power value that it sends to the Game Engine through the hex display and the width of the power bar sprite shown on the XVGA display.

In developing the shaker module, there were several changes to the design to optimize its performance. Initially, the shaker was designed to determine time differences between consecutive pulses, and that value will be translated into predefined power values used by the Game Engine. However, after implementing the physical devices and the module, the device was not working as expected. Occasionally, there appeared to be a sudden drop or spike in the power value, which could lead to other problems in the Game Engine and a flaw in the game. Much time was invested in trying to fix the problem. Through use of the logic analyzer and the oscilloscope, it appeared that, due to mechanical properties, the bouncing of metal contacts of the switch in the shaker result in multiple pulses in a very short period of time. This would translate to a sudden increase in power level, which is not physically feasible for the user to produce during the game. After debouncing the shaker's switch, the problem was not as severe, but not adequate enough to be used in the game. Finally, there have been suggestions to change the design a bit; instead of detecting the time difference between consecutive pulses, there should be a ongoing sum. For every pulse that was generated, a defined value was added to the overall power figure, while for every fraction of a second, the power value decrements. This is analogous to a sink, where there is a constant rate of drainage, but the water level can remain unchanged if the appropriate rate of water is flowing out of the running faucet. Luckily, the implementation of this design turned out to be successful, so this design was used in the shaker module.

Aside from the shaker, a few changes have been made to the initial design of the video processing unit of the project as well. Initially, the pixel detection module was designed to perform all of its detection in YCrCb, a color space that distinguish objects in terms of luminance and chrominance. After implementing the detection in YCrCb, it appears that its performance was decent enough to be used in the game. However, further testing shows that even with the user holding the pointer still, there was much noise in calculating the center of mass for the pointer, resulting in the pointer sprite "twitching" on the XVGA display. Additionally, the pointer was very unstable when it was near the edge and side of the screen. After much testing, it was decided by the team that detection in YCrCb was not accurate enough to make the project outstanding. The team hypothesized that this was due to the lack of concentrated pixels that were detected from the pointer object. Because YCrCb is sensitive towards curved objects, detection of a ball is not spectacular. Because of this, pixel detection in HSV was suggested and was implemented shortly afterwards. Research has shown that detection in HSV allows more pixels to be detected regardless of the object's shape, which would solve the problem that was discussed (See Figure 16).

However, the implementation of the pixel detection in HSV did not offer better performance; much noise from surrounding areas was detected in addition to the pointer, resulting in an unstable calculation of the pointer's center of mass. From

Figure 16: Screenshot of XVGA display in HSV.

this, a frame filter design was suggested to be incorporated to refine the pixel detection. The frame filter allows only pixels that falls within a designated frame to be included in the position calculation. Once the position is determined, that location will be used to determine the location of the new frame to be used during the next clock cycle. This implementation was very helpful to the block because it eliminates all noises in the surrounding area, allowing the object to be able to move freely to any location within the XVGA display without encountering significant noises.

To further the design of the pixel detection, a problem that can occur during the game is that the user might lose track of the pointer by moving the object outside of the camera's range. To compensate for this, there have been suggestions to design the module to detect if the pointer is lost. After much brainstorming, the team decided to incorporate a smaller frame into the current frame filter that will be used to see if a certain percentage of the pixel falls within that frame (See Figure 17).

If there is a high concentration of pixels inside the smaller frame, this implies that a device must have a large area of the same color that falls into the defined threshold, which would be the case of the pointer. Unfortunately, there have been consistent bugs in the implementation of this idea; there were many attempts in trying to find the optimal percentage to use as a threshold point to determine whether the pointer is lost. Due to time constraints for the project, the optimal threshold point was not found, so the implementation of the smaller frame was not included in the most recent release of the project.

For the Graphics Engine, it was eventually determined that scaling sprites would require too much computational time, and would have to be abandoned. However, the general algorithm that was used to scale the sprites was rather innovative to avoid performing any divisions. The scale factor for each sprite was determined by the perspective transformer described above. This scale factor would then determine which addresses needed to be read from the ROM. Two sets of registers were used to assist in this address calculating process. One set represents the last

Figure 17: Screenshot of XVGA display, incorporating the pointer detection. Note: the red pixels are used to determine if the pointer is at sight. In this case, the pixel concentration was not high enough, so the module interprets the pixel as "lost."

Table 1: Table of derivation for iterations

ROM Address	2	3	4	5	7	8	9	12
Scaled ROM Address	2	3	4	6	7	8	10	12
Spatial Address	1	2	3	4	5	6	7	8
ROM / Spatial	2.00	1.50	1.33	1.25	1.40	1.33	1.29	1.50
Scaled / Spatial	2.00	1.50	1.33	1.50	1.40	1.33	1.43	1.50
Scale Ratio	1.33	1.33	1.33	1.33	1.33	1.33	1.33	1.33

ROM address+1, the other is the last special address (where we physically are on the sprite). Taking this ratio as Scaled/Spatial = R we compare this to the scale factor Max Scale / Scale = S. If $R < S$ then we add extra to the scaled address and carry this over into our next iteration. This most closely preserves the ratio on each iteration. The iterations which required an update are highlighted below.

This scaling is performed on both the height and width of the sprite in order to produce the appropriate scaled sprite at a certain scale (which is all determined by the sprite's z coordinate). However, during the integration phase it was determined that the sprite scaling required too much computational time and would interfere with the camera's input clock and VRAM access time. In the end the sprite scaling was abandoned and simple blob scaling was implemented to reduce computational time between clock edges.

Conclusion

The integration phase of the project proved to be more difficult than expected, and required some of the original features of the project to be abandoned. The final product was a less glamorous version of the original, but included fully functional game play and interfacing between inputs, game play logic, and graphical display. More details for why the portions of the graphics module were abandoned are explained below.

There were a few in stream modifications made to each of the modules. Originally, the YCrCb color space was being implemented for color detection. However, it was determined that only the center region of the camera image could be detected reliably. To create a more robust color detection system, the HSV color space was implemented. This accounts for a wider range of lighting conditions. While the HSV color space could successfully pick out the colors, it created a large amount of noise in the entire image. Hence, when taking the center of mass on the entire screen, the average location was very susceptible to noise.

Another alternative approach was implemented to reduce the average noise created by HSV. Instead of a median filter to ensure that consecutive pixels were detected, a more robust frame filter was implemented to localize the noise. The frame filter is based on the fact that pointer movements are localized to the last frame (i.e. the pointer does not move dramatically between any two consecutive frames). Therefore, we can seed the frame on the location of the last pointer's average location using previous estimations.

The game play module had minor modifications. The original plan was to incorporate 16 creatures and 32 projectile objects. Unfortunately, the compile time for an integrated project with 16 creatures and 32 sprites pushed upwards of two hours. In addition, the processing time was too long per sprite and the camera image suffered when writing and reading to the ZBT. Instead, the creatures and projectiles were halved. The final project incorporated 8 creatures and 16 projectiles. Compile time was reduce to approximately 1 hour and the reduced number of creatures and projectiles meant that the graphics module could be reduced from handling 125 sprites down to handling 64 sprites. Now the three stage pipeline was modified from 125 to 25 to 5 to 1 to now 64 to 16 to 4 to 1. This also reduces compilation time with no loss to functionality.

The graphics engine had major modifications in order to compensate for the clock speed. The sprite scaling required too much computational time, and as a result interfered with the camera's clock either reading from the ZBT or writing. This was an easily identifiable problem because the camera image was a noisy image instead of the expected camera image. The noisy image arose directly after the sprite scaling was added to the integrated program. To correct this problem a simpler sprite was implemented to produce a pseudo-scaling of the sprite blobs according to their z axis location. Although this method did not produce the correct perspective scaling, it did produce a sprite scaling sufficient enough to trick the human eye. In addition, the pseudo scaling method was fast enough to not cause any problems writing to the ZBT or reading from the ZBT.

In the end, the project was a very challenging and rewarding experience. It shows

the importance of abstractions at times of integrating different modules of the code. In addition, timing was also a significant issue, and required much time invested to ensure that all pieces of the project were able to work together. Aside from the technical experience, this project has taught the team much in communication, time management, and teamwork dynamics. Much was accomplished in the process, and the experience gained will undoubtedly prove invaluable to the team in their future careers.