FlipFlop Team 18 Flappy Bird

James Bao
Dept. of Electrical, Computer, and Software Engineering
The University of Auckland
Auckland, New Zealand
jbao577@aucklanduni.ac.nz

Nicholas Wolf
Dept. of Electrical, Computer, and Software Engineering
The University of Auckland
Auckland, New Zealand
nwol626@aucklanduni.ac.nz

Marianne Healey
Dept. of Electrical, Computer, and Software Engineering
The University of Auckland
Auckland, New Zealand
mhea373@aucklanduni.ac.nz

***Abstract*—Flappy Bird is a side scroller game where the player controls a flapping bird whilst trying to fit through small gaps in pipes. As a team of three, we programmed a gate-level description of this game on a DE0-CV board using VHDL. In this report we will explain the fundamentals of our design, touch on the justification for our decisions, and discuss improvements and optimisations that should be made.**

***Index Terms*—fpga, vhdl, flappy bird, side scroller, game**

I. Design

A. Design Specifications

Flappy Bird is run on an Altera Cyclone V FGPA, provided to us as part of the DE0-CV development board. A PS/2 mouse is used to interface with the FPGA and enable the user to control the bird. Finally, the FPGA drives a 640x480 pixel Video Graphics Array (VGA) output signal for the game to be displayed.

B. Design Description

Flappy Bird is an implementation of a classic side-scroller video game, and satisfies the following description: "The player controls a bird, attempting to fly between rows of pipes without hitting them."

Our design of Flappy Bird features a bird that falls with gravity towards the floor, but flaps upwards when the player inputs a mouse click with the PS/2 mouse. The bird encounters a number of different objects as the world scrolls past—namely pipes and coins.

The pipes protrude from both the top and bottom of the screen leaving only a small space for the bird to fly through, and any contact with these obstacles results in a loss of life for the bird. These obstacles are equally spaced as the world scrolls past, but the exact vertical position of the gap is generated in a pseudorandom fashion.

While the bird weaves between pipes that seek to do it harm, it aims to collect as many coins as possible—with each coin collected incrementing the player's score by one. The overall aim of the game is to keep the bird alive for as long as possible, with a global high score persisted across multiple attempts as a target for the player to beat.

Depending on the player's chosen game mode, the speed at which the bird encounters these obstacles may also increase over time—increasing the game's difficulty and providing a greater challenge for the player.

The player may pause the game at any point by right-clicking with the PS/2 mouse.

C. Game Modes

Our Flappy Bird game begins at the menu screen (Fig. 1), where the player selects their desired game mode using the KEY3 and KEY2 push-buttons on the DE0-CV board. These push-buttons start Flappy Bird in TRAINING Mode and GAME Mode respectively—with both modes initially bringing the user to a hovering bird with no world movement.

Fig. 1. Screenshot of a menu screen waiting for player's input.

Both game modes begin with the bird flapping once the player inputs their first left-click with the mouse, with the bird flying in the vertical axis in the manner described previously. This continues until the player encounters the first obstacle, at which point the player must control the bird to fly through the pipe gap.

In TRAINING Mode, the bird is given three lives, with the number of lives remaining displayed to the player in the top-left corner of the screen as demonstrated in Fig. 2. A life is lost each time the bird collides with a pipe until the bird has no lives remaining, at which point the game ends.

In GAME Mode, the bird is instead afforded only one life—the game ends once it collides with a single pipe.

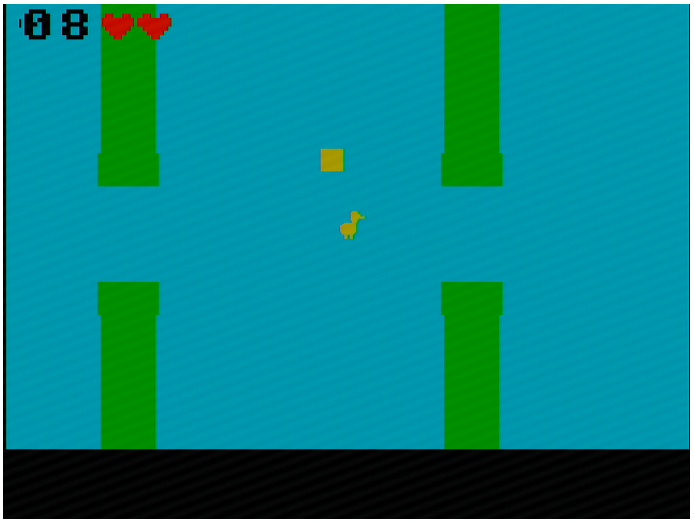

Fig. 2. Showcase of TRAINING Mode game display.

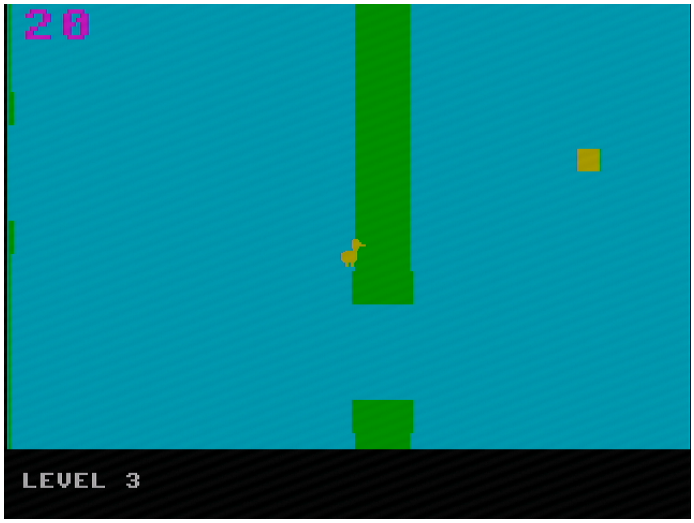

Fig. 3. Showcase of GAME Mode game display.

In both game modes, any collision with the world floor or ceiling results in the game ending, where the game then pauses, waiting for the player to input one final left-click of the mouse before returning them to the menu screen and displaying the saved high score.

The score increments in both game modes each time the bird passes through an obstacle, as well as each time a coin is collected. Each 10 accrued score points in GAME Mode progresses the player to the next level, where the scrolling speed of pipes and coins increments by a constant amount to increase game difficulty. In TRAINING Mode, the player remains in Level 1 until the game ends; allowing an opportunity to hone their skills at the lowest difficulty level.

Level 1 also features identical pipe gap generation between each game attempt to decrease the game difficulty further, with pseudorandom pipe generation applying from Level 2 onwards. The player's current level is displayed to them in the lower-left corner of the screen as seen in Fig. 3, in addition to the score changing colours.

D. Finite State Machine

1) Introduction: The finite state machine diagrammed in Fig. 4 is the core of our FPGA game console, and controls the different states the objects of our Flappy Bird game can be in. It is designed as a Moore type finite state machine, as the outputs predominantly rely on the current state value. The state machine itself is concerned about the bird's status and does not control scoring or other miscellaneous game mechanics.

The possible states the machine recognises are as follows:

1) DrawMenu - This state represents the games boot screen, where the game modes, game name, and high score are displayed. It is reached upon the start of the game, game reset, and on the condition that the bird died and the player would like to try again.
2) TrainingModeInit and HardModeInit - Upon reaching this state, the bird will begin to hover in the air, awaiting a player's click to start the movement of the pipes. A signal "difficulty" is used to store the current game mode, as that information is essential for the correct output in the case of collisions.
3) Gaming - This state is the most recognisable form of the game, where the bird is flapping and the objects are moving along the screen. This state is where the state machine decides whether the game is over or has been paused.
4) Paused - When the user would like a break, they may exit gaming mode and into Paused, where all movement is halted.
5) Dead - When the bird has hit the floor, lost all lives in TrainingMode, or hit an obstacle in HardMode, all movement will stop and the user has no choice but to go back to the menu screen to play again.

2) State Transitions:

1) DrawMenu -
   Set: If the global reset has been set, the current state is DrawMenu, or if the current state is Dead and the mouse's left button is clicked.
   Next: If the leftmost push button KEY3 has been pressed, the state will proceed to TrainingModeInit. If the push button KEY2 was pressed, the state will go to HardModeInit.
2) TrainingModeInit and HardModeInit -
   Set: If the current state is DrawMenu and the respective push button is pressed
   Next: If the mouse's left button is pressed, the state will proceed to Gaming.
3) Gaming -
   Set: If the current state is either game difficulty initialisation screen, and the left button of the mouse is clicked.
   Next: If the difficulty is training mode and "bird_died" is set to one, the state will move to Dead. "bird_died" is a signal that is shared among game difficulties. In training

mode, the signal will be set when the floor or ceiling has been touched, or when all lives have been depleted. If the difficulty is in hard mode, any type of collision that would warrant the loss of a life immediately forces the state to Dead. If the bird has not died and the right mouse button is pressed, the next state will be Paused.

4) Paused -
   Set: If the current state is Gaming and the right mouse button is clicked.
   Next: If the left mouse button is clicked while the game is paused, the next state will be Gaming.
5) Dead -
   Set: If the current state is Gaming and "bird_died" is set to 1.
   Next: If the left mouse button is clicked, the next state will be DrawMenu.

3) Inputs: The state machine's most important capabilities come from the inputs given to it by our top level entity. These inputs include three push buttons (two for menu navigation, and one for resetting), both mouse buttons (not for jumping, but for deciding to go back into DrawMenu or Paused), a clock (to synthesise functional storage elements like difficulty, and sync the states to an update time), and collision detectors (driven by the detection of both bird and object in a pixel, these decide when it is time to end the game or subtract a life).

4) Outputs: The state machine sends various outputs that control what is being displayed on the screen, and how objects are moving. The bulk of the outputs send a standard logic to main that dictates whether or not a display element should be shown on screen. The value of these outputs change per state, and vary from enabling the hearts, menu and menu fields. Other outputs such as "movement_enable" and "bird_hovering" are read by other components of our design to decide what the behaviour of the bird and other objects should be.

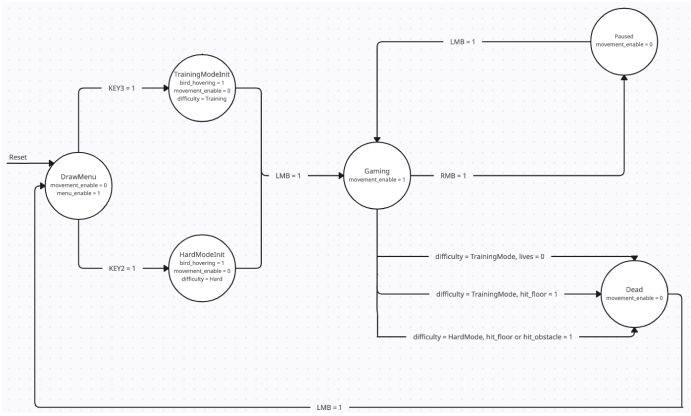

Fig. 4. Our high level state diagram.

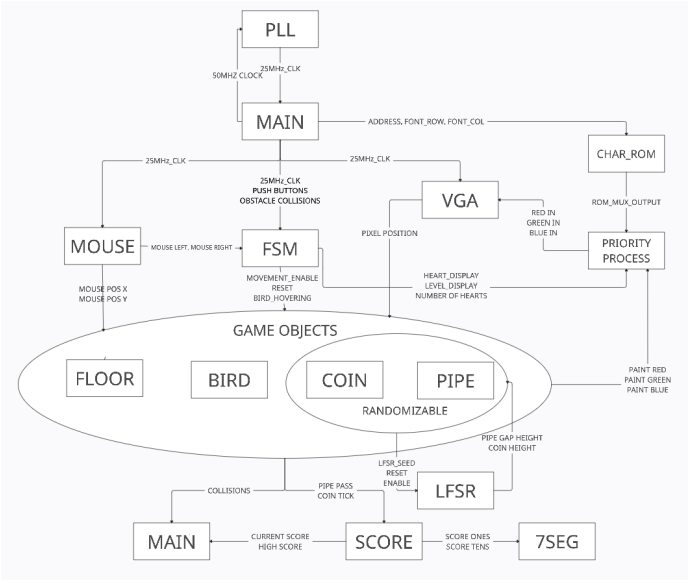

Fig. 5. Our high level interface diagram.

E. Implementation

F. Object Painting

We implemented our object painting algorithm with a fundamental concept of stack order, inspired by the "z-index" property of Cascading Style Sheets. This concept can similarly be thought of as the layers you have within a graphics editing program like Photoshop or GIMP, where higher layers are drawn on top of lower layers.

In Flappy Bird, we first clearly defined the intended stack order of every object to be drawn to the display (e.g. the bird, pipes, coins, score, hearts for lives remaining, etc.). We then implemented each of these objects as a unique entity that could be included as a component within our top-level entity, where each of these object entities produced four display-related signals for every given pixel:

1) red - the red component of the object's Red-Green-Blue (RGB) colour value,
2) green - the green component of the object's RGB colour value,
3) blue - the blue component of the object's RGB colour value, and
4) in_pixel - whether the object exists at that pixel.

The "in_pixel" signal can be thought as a binary implementation of an alpha (transparency) channel on the RGB colour value. It may alternatively be thought of as a form of 'display enable' signal output from the component; i.e. whether to draw the object at the given pixel.

With our individual entities for each type of desired game object, we then used the previously defined stack order to implement the "paint_screen" process in our top-level entity. This process simply implements a priority encoder, taking in the different object "in_pixel" signals as input, using an "if-elsif" conditional tree to find the object with the highest stack

order at the current pixel, and outputting that object's "red", "green", and "blue" signals to the "vga_sync" component.

G. Collision Detection

1) Introduction: Our collision detection is a process driven by the VGA component. We make use of the pixel x and y position output from "vga_sync" to check each pixel in a frame for a collision. If both the bird and an object is attempting to draw its sprite in the pixel, there is a collision. The boundaries for each object are within their personal entity. We define a region with a series of rectangles to represent the hitbox of an object. This region is compared against the current pixel position of the VGA process. In the case that the current pixel is within the defined region, output "in_pixel" is set to 1. In our collision detection process, we check if the the "in_pixel" output from the bird is set and check the status of "in_pixel" from other entities such as the floor and obstacles. If we detect a combination of two high "in_pixel" outputs, we set the corresponding collision signal.

The different types of collisions we implemented are as follows:

- "hit_floor" - This signal is set when the bird collides with the ground or the ceiling.
- "obs_one_hit"/"obs_two_hit" - This signal is set when the bird collides with the first or second obstacle, respectively.
- "coin_tick" - This signal is set when the bird touches a coin, and the score is prompted to increase by one.

2) Multiple Collisions in a Frame: Due to the way we implemented collisions, each frame frequently registers multiple collisions. As the pixel's current position proceeds column by column, row by row, it finds several positions at which both the bird and an object is requesting to be drawn. Hence, in each row the bird is found to be colliding with something, there will be a rising edge of the collision signal. The issue arises in the way we process collisions, most problematically in training mode where we have lives. We initially read the rising edge of the respective collision signal to execute logic for removing lives, but realised that many lives could be taken off at once. To work around this, we needed a way to halt collision processing in a frame once a collision has been detected. We created signals "obs_one_pass"/"obs_two_pass", which are set when the respective obstacle has reached a position near the left side of the screen. Once an obstacle has been hit, we turn off collisions until that obstacle pass signal is set. This way, when the bird loses a life, it is invincible for a short duration and prevents multiple lives from being taken away.

3) Additional Information: We programmed our object detection logic to be useful in multiple facets of the design. Specifically, the "in_pixel" output is not exclusively owned by collision detection; We use "in_pixel" to draw the objects and decide which ones get layered over each other. For example, when the bird is in the pixel, we send the colour yellow to be displayed. In contrast, the pipes are painted green and are a layer under the bird, meaning we never lose sight of the bird when it flies through a pipe. We determine the layer of the entity through cascading "elsif" statements. From top to bottom, the priorities go:

Menu, score, hearts, level, bird, floor, pipes, coins, screen.

This is vital to the gameplay of flappy bird, as we need the most important information to be visible on the screen.

II. Decisions and Trade-Offs

A. Game Clock

For the sake of simplicity, we used the "vert_sync" signal to provide a controlled update speed for the velocities of our objects. However, this meant that to achieve smooth, quick movement, we had to transport the position of the bird and pipes more than one pixel at a time. Initially, we intended for the bird to stop directly on the floor or obstacle it hit, but our design couldn't avoid the bird phasing into objects at least a few pixels. If we switched "vga_clk", we could have met our initial expectations, but it would require significant refactoring.

To be precise, the "vert_sync" signal controls movement in the game for pipes, the bird, and coins. We employ the "vga_clk" to draw our objects at a 60 Hertz refresh rate.

B. Object Painting

Our chosen implementation of stack ordering avoids the possibility of two separate objects mixing their colours, as may occur if we simply applied a superposition of the RGB outputs from each object entity. If we instead chose this approach, we would either expect all of our individual objects to blend together (such as the green pipe with RGB output #010 disappearing into the cyan #011 background), or poor, difficult-to-maintain logic trying to determine whether to paint an object using only its RGB output. Such a pursuit would be very difficult, as it would need to differentiate between an object having a black pixel (#000), and having no pixel at all (most sensibly #000, but this would not work).

A trade-off to acknowledge however is that we require an extra one-bit "in_pixel" signal for each object we wish to draw to the screen, requiring additional resources. Having acknowledged this however, we believe it is important to recognise that this additional one-bit signal is a very minor cost when considered relative to the other signals required by each entity, and the low number of objects within Flappy Bird. Furthermore, this "in_pixel" signal is also used to simplify our collision detection algorithm significantly.

C. Graphics

Initially, we planned to use the external SDRAM storage device mounted on the FPGA to store large amounts of high resolution image data. However, we found that this was not as simple as using embedded memory. Since there is significant delay involved with transporting data across the board, we would need to make our own SDRAM driver and compute the proper amount of bits to send synchronously. We would need to synthesise a buffer for the data that came in from the SDRAM to decide when to send data to the VGA display, and when to refresh its data. Unfortunately, this was out of scope for the project, but we managed to use MIF files to draw the

hearts on the screen. Currently, the majority of the game's graphics are done by directly colouring the calculated regions of our objects a solid colour.

D. Linear Feedback Shift Register

We chose to implement a 7-bit LFSR with a maximal loop length of 255, which we treated as a signed offset about the vertical middle of the screen. This gave us a signal varying between

$$-128 \leq \text{Generated Offset} \leq 127 \tag{1}$$

(note that the LFSR cannot generate a value of 0), which we added to the centre-row coordinate of 240. This produced a vertical coordinate for the centre of the pipes/coins varying between

$$240 - 128 \leq \text{Object Centre} \leq 240 + 127 \tag{2}$$

$$112 \leq \text{Object Centre} \leq 367 \tag{3}$$

This guaranteed us 112 pixels of padding between the randomised pipe/coin centre and the top/bottom edge of the screen, which sounded ideal towards ensuring the pipes/coins remained within a reasonable playable range.

We also chose to implement this as a Galois-type LFSR with taps placed between registers over a Fibonacci-type LFSR with taps placed on the feedback path, as the additional logic elements on the feedback path of the Fibonacci-type LFSR increases the maximum propagation delay of the circuit such that the maximum clocking frequency is impacted. Therefore, we decided to use a Galois-type LFSR to avoid any possible performance constraints.

Our LFSR has taps placed at 1, 2, 3, and 7 as recommended by the lecture slides, to ensure that the LFSR produces a maximum length sequence.

1) Game Mechanics: Most of our game logic worked exactly how we wanted it to from the beginning. We achieved smooth flapping through subpixel calculations, a dynamic obstacle speed, and perfect interaction between the bird and obstacles. However, we wanted the bird to be able to fly above the screen and die if it hit a pipe passing by. Since our collisions were driven by the detection of pixels that included both the bird and an object, they break entirely when operating outside of the VGA's area. Rather than re-designing our collision detection system entirely, we compromised by disallowing the player to fly above the top of the screen.

To go into detail, subpixel calculations work by virtually expanding the space the bird flies in to a region much larger than the VGA display. We can then gradually move the bird at a more precise speed along the subpixel plane, and divide that speed by 16 to return the computed change of position in the VGA plane. This allows us to increase the velocity of the bird in smaller increments rather than one extra pixel per "vert_sync". We did this to give the game a better flow, which was more enjoyable than more jumpy physics.

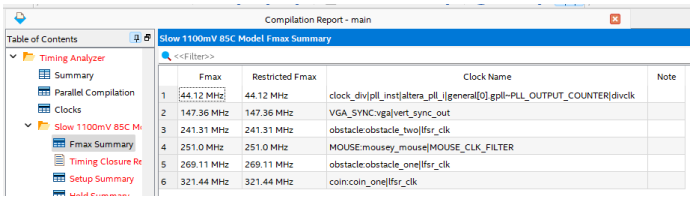

Fig. 6. Our high level interface diagram.

III. Resource Usage and Performance

A. Operating Frequency

The timing analysis in Fig. 6 shows the maximum clock frequency a component can output. Our lowest clock frequency is 44.12MHz, and the component is PLL. We expect an output of 25MHz from PLL to drive a 60Hz refresh rate 640x480 VGA monitor, which is well within the limitation. However, if we wanted to increase the size of the screen or the refresh rate, warranting a faster output from the PLL, we may reach the upper limit of this component's capability. To exceed the current maximum frequency, we would need to optimise our design to be more efficient in the way components use the output clock from PLL.

B. Logic Elements

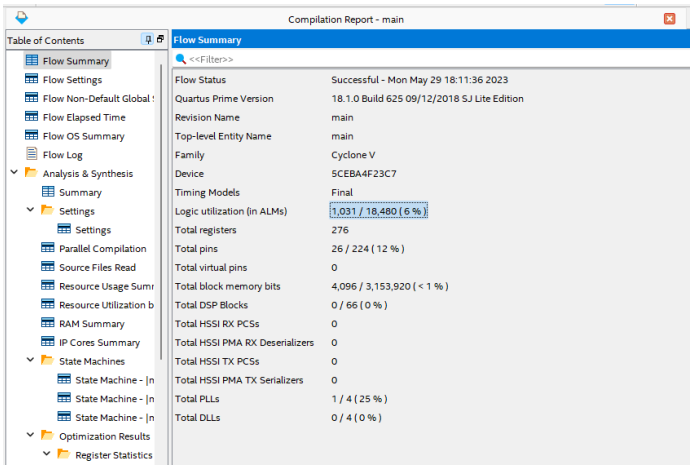

Fig. 7. Our high level interface diagram.

In our design, we tried to optimise the bit width of our signals so no logic elements were wasted. However, we found multiple instances of integers near the end of the project which could be simplified to the unsigned type. Additionally, some sections of the code are fairly redundant and could employ functions to reduce the large amount of similar signals and logic.

C. Registers

As shown in Fig. 7, we synthesise 276 registers in our gate-level description. The contrast in the amount of registers and logic elements is the effect of us responsibly avoiding clocked processes when possible. In clocked processes, we limited the

number of inferred flip flops by ensuring that signals used for transporting outputs were not included in the edge of a clock.

Similarly to the explanation above, some signals have bit widths that could be further optimised, and improving them would also reduce our register usage.

IV. Future Improvements

A. Fixing vga_sync Signal Widths

Currently, a known bug exists where our "pixel_column" output signal from "vga_sync" is a 10-bit signed(9 downto 0), but this is actually insufficient to properly encode the full horizontal axis of our display. Our FPGA outputs a 640x480 pixel VGA signal as outlined in the design specifications, i.e. 640 pixel columns across the screen. This is sufficiently fit inside a 10-bit unsigned number, as this data type accommodates a representation range of 0–1023.

However, as a signed data type, this 10-bit number instead supports a representation range of -512–511—where, as $511 < 640$, the value stored within "pixel_column" will overflow to -512 at the right-hand side of the screen. Although we intentionally wished for a signed type so that we could position objects off the left-hand edge of the screen (e.g. when implementing a smooth wrap-around of the pipes, it is critical that its x-coordinate can become negative), we neglected to increase the width of this signal to an 11-bit signed(10 downto 0) to maintain the same upper limit of 1023.

This is the reason that necessitates the concatenation of leading 0s in our spatial detection logic within our various game objects, as this implicitly treats the signed number as an unsigned number, and reverses any overflow—such that, for example, an overflowed value of -464 is properly treated as the intended 560. This works as the concatenated leading 0 effectively negates the sign-bit of the two's complement representation.

Unfortunately, we did not have enough time at the end of the project to update every component that took in the "pixel_row" signal as an input, but this would be a priority fix for future to reduce the logic complexity within our game objects.

B. Extra Life Gift and Object Sprites

This is a new feature that is almost fully implemented—these heart-shaped gifts are infrequently spawned instead of regular coins whilst in TRAINING Mode, and replenish one of the bird's lives upon collection. These heart gifts are implemented as a new component in a similar fashion to the existing coins, but with an internal "char_rom" component to retrieve the heart sprite from the .mif memory initialisation file.

The only task still to be completed is to notify the state machine upon collection of the gift so that the bird's lives can be incremented, which, although minor, has become a task that we have been unable to find time to complete.

The implementation of this component involved figuring out how to draw the sprite using the "char_rom", and how to move this sprite across this screen. This is already fully functional and can also be easily ported to our existing bird, obstacle, and coin objects to improve the visual appearance of the game's graphics.

Acknowledgment

A big thank you to our lecturers Dr. Morteza Biglari-Abhari and Dr. Maryam Hemmati; Teaching Assistants Ross Porter, Asher Butler, James Park, and Callum Iddon.

References

Coursebook information from COMPSYS 305 directed by Dr. Morteza Biglari-Abhari of the Department of Electrical, Computer, and Software Engineering at The University of Auckland.