Undertale on FPGA

Shun Yoshikawa

Department of Mechanical Engineering

Massachusetts Institute of Technology

Cambridge, MA

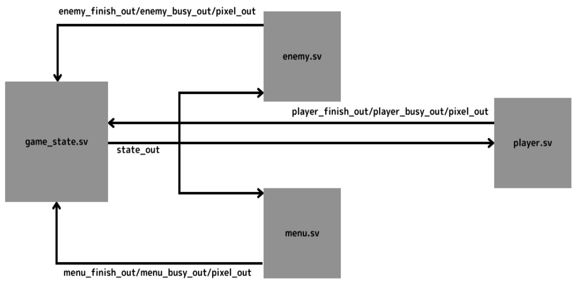
shuny@mie.edu

***Abstract*—I present a design for replicating one of the game scene from the video game, “Undertale” as well as the implementation for interaction between players and FPGA with the use of camera inputs. At the end of the project the system was developed to the extent it can give entertaining experience to the user. This project has shown a new approach to utilize fpga involved system as a game engine.**

I. Components

This Project is composed of three parts: FPGA, a VGA monitor, and a camera board. The majority of codes constituting the game mechanic is writtten in System Verilog, which is uploaded to FPGA. The FPGA is connected to the monitor with a VGA cable and the actual game play is displayed on it. The camera board is also attached to the FPGA in order to take in visual input from a player.

II. State Transition


enemy_finish_out/enemy_busy_out/pixel_out
enemy.sv
player_finish_out/player_busy_out/pixel_out
game_state.sv
state_out
player.sv
menu.sv
menu_finish_out/menu_busy_out/pixel_out

The actual game mainly consists of three states. They are connected in a definitive sequence, and the system continuously transits from one state to another. game_state.sv module is responsible for this phase transition as well as most of other functionality that are relevant to game mechanics in the system. The logic called state_out, which is 4 bits in size, is defined to store the value corresponding to the current state. For each state in the game, there are a set of logics to determine the current state of the game. They are:

- logic X_busy_out;
- logic X_finish_out,old_X_finish_out;
- logic[11:0] X_pixel_out;

(where X is replaced by a term representing each phase)

when one of the phases are active ,namely X_busy_out is set high, X_pixel_out is written to the output of the game_state.sv (pixel_out). X_finish_out goes high when the game exits its current active state. The game_state.sv module looks for the rising edge of this logic to detects the end of the state. Then it properly shift to the next state of the game by modifying the value of state_out.

A. Menu State

Whenever the game initiates, it goes into the menu state. During this state rectangular-shaped white frame appears in the middle of the display. Below that are four buttons indicating different commands from the original game which the player can choose one from. Those buttons are aligned horizontally, and only one button is highlighted at a time to show that it is being selected. The button highlight shift either to left or right depending on the controller input from the camera board (discussed in section III). The sub-module that dictates the menu state is menu.sv.When the value of state_out in the game_state.sv becomes 4’b0000, menu_busy_out is set to high and performs all the functionality described above unitl the player chooses one of the commands. To avoid unnecessary complication,however, not all of the four commands would be implemented in exactly the same way they work in the original game. The one that would be implemented thoroughly is the Fight command. When the player chooses to carrry out this command,menu_finished_out goes high, and the game exits the menu state and proceeds to the player state.

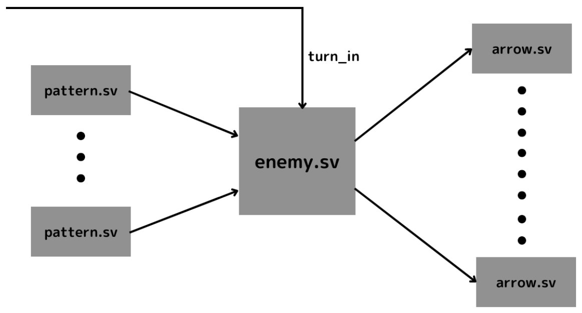
B. Player Phase

In this section, the state refered to as “player state” indicates the one which follows the menu state after the player chooses the Attack command out of four commands in the menu. The sub-module named player.sv controlls player state. The value of state_out which represents the enemy state is 4’b0001. During this state, a player is supposed to play a mini game to decide how much damage is dealt to the enemy for the round. As soon as the player state commences, a vertical bar appears on the right hand side of the rectangular frame (described in A.) . The vertical frame starts moving horizontally, and dissappears when it reaches the other side of the frame. The player can apply input signal (detail of which will be described in section III) to stop the bar while it is visible on the display. The damage the player deals to the enemy is determined by where the bar stopped inside the frame. In specific, the closer to the center of the frame the bar is, the more damage will be dealth to the enemy. When either the bar stops moving and the damage is dealt, or the bar dissapears from the screen, player_finish_out goes high to trigger the game_state.sv to move onto the next state.

C. Enemy Phase

The sub-module enemy.sv is responsible for the enemy state. Enemy_busy_out is asserted high when state_out is equal to 4'b1000. Conpared to menu.sv and player.sv, enemy.sv has considerable amounts of functionalities.

i. Properties of Arrows



During the enemy state, arrows appear from either of four sides (top,down,left right) of the screen, and move toward the center of the display, where there is a heart-shpaed sprite representing the player. The behavior of arrows follows one of the fixed patterns that are pre-defined, and the pattern to be used is switched every round. Each pattern information is stored in pattern.sv. Its properties are:

output logic[71:0] timing

output logic[71:0] speed

output logic[47:0] direction

output logic[23:0] inversed

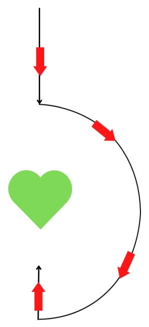
pattern.sv also has parameters. Their values are written into the logics shown above, respectively in a combinational logic.. That is, a pattern can be defined by making an instance of pattern.sv with proper parameter values assinged. Each logic has a bit size of a factor of 24. This is because the maximum number of arrows that appear during the enemy state is 24. Timing is a 72-bit logic, and each 3 bit define the interval between two arrow instantiation. Timing[2:0] represents the it takes the first arrow to appear, and timing[5:3] represents the time from when the first arrow appears to the second arrow is intantiated, and so on. The interval between two arrow instantiation are actually represented by the number of clock cycles enemy.sv holds the process.The unit number of clock cycles is 6500000 (theoretically 0.1 seconds since the frequency of the clock signal is 65MHz). The interval is acquired by multplying timing[i+2:i] by 6500000. The rest of logics of pattern.sv also store data in the same way as "timing". There are 10 instances of pattern.sv in enemy.sv and all the parameter values are stored into two-dimentional arrays called:

- logic[9:0] timings[71:0];
- logic[9:0] speeds[71:0];
- logic[9:0] directions[47:0];
- logic[9:0] inverseds[23:0];

Depending on the turn number, values from one of the pattern instances is called in sequence. The turn number is handed to the enemy.sv by game_state.sv via the logic turn_out, a 4-bit logic that ranges from 1 to 10.

ii. arrow.sv

The actual movement of each arrow is dictated by the module arrow.sv. As mentioned earlier, they are 24 instances of arrow.sv to accommodate the maximum number of arrows that could appear during one round. Inside arrow.sv logic x and y are modified at every frame. Arrows that take in inversed_in == 0 only move straight toward the heart in the center of the monitor. If inversed_in is set to high, on the other hand, arrows are supposed to change its moving direction halfway.



The actual trajectory of an arrow would look like what is shown on the picture of the left. The curving path is realized by using quaderatic equation. In the case of the picture, the value of y is incremented by a fixed number, and the value of x is found by the equation:

y = 607- (1/128)(y-384)^2

Inside arrow.sv there is a logic called inverse_state to controll the phase of the movement of an inversing arrow. It first moves straight (inverse_state == 0)and at a certain point it starts to follow the curving path (inverse_state == 1), and when it reaches the other side it starts to move toward the heart once again(inverse_state == 2).

Arrow.sv has also a functionality for collision detection. When an arrow reaches the center of the display i.e. it hits the heart sprite, arrow.sv asserts hit_player high, which is passed onto enemy.sv in order for it to determine whether it should modify the health bar on the screen. In addition, when an arrow passes where the shield can be positioned, arrow.sv checks if the shileld in the way of the arrow by comparing the value of direction[1:0] in arrow.sv and rotate_out[1:0] from shield.sv. When shield successfully block an arrow, the corresponding arrow.sv module stops rendering the arrow by setting one of the output logic valid_out to low. If an arrow hits the heart, it sets another output logic called is_player_hit to high for one clock cycle, which is passed onto health_bar.sv via game_state.sv to modify the visual of the health bar of the player. Either way a logic called is_hit is set to high whenever collision occurs. Enemy.sv detect the rising edge of is_hit from each instance of arrow.sv and counts the number of provoked edges at each round. When it reaches the number of arrows that are supposed to appear during the round, it indicates that all the arrows dissappeared either because they are blocked by a shileld or they reached the heart.

iii. Shield

A player uses a shield to block the moving arrows to protect the heart sprite in the center of the display. The shield has a narrow rectangular shape, and is located near the heart sprite facing one of the four directions.The player manipulates the position and the rotation of the shield to block either of the four paths arrows follow. The behavior of the shield is contolled by shield.sv. This module takes in rotate_in[1:0] as input, which determines the direction the shield has to face. The logic is wired to the input from the camera board via game_state.sv.

The game exits the enemy state when all the arrows for the round dissappear from the display (either by reaching the center or being blocked by the shield)

III. Camera Interaction

In this project, the fpga takes in visual input from the player by using camera board. Input extracted from camera.sv is stored into the BROM. Then it is scaled in a size of 360 by 480 pixels by mirror.sv and scale.sv. r_and_cr.sv extracts the r and cr value of camera input in order to detect location of the red sphere in the camera view. The player physically manipulates the location of this red sphere to send desired input to the system. The system basically identifies the input by which one of the four sections inside the 360x480 frame the red sphere is located. These sections are divided by two linear equations: $3x - 2y = 0$, $3x + 2y = 960$, where x and y represents the horizontal and vertical values of pixel coordicate, respectively. Controller.sv is responsible for actually outputting the value rotate_out[1:0] which indicates the section the red sphere is in.

A. Shield Movement

Taking the value of rotate_out[1:0], the shield in the enemy state changes its position and rotation in a way it faces either of the four directions.

B. Swipe

During the player state, the player is supposed to stop the bar inside the rectangular frame in order to give damage to the enemy. This will be done by making a “swipe” motion in from of the camera board. The system detects swipe when the red sphere moves from the upper section to the lower section. Player.sv looks for an edge case where the value of rotate_out[1:0] goes from 2’b00 (up) to 2’b01 (down).

C. Menu Select

Menu.sv also make use of the logic rotate_in. This module divides the camera board input into three vertical areas . The center area of the three is a base area where nothing happens when the center of mass is located inside it. When the center of mass shifts to either area next to it (one on the left or right), it asserts left or right key input depending on the area the center of mass moved to.

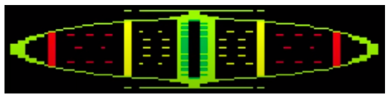
IV. Graphics

In order to draw objects on a monitor this project uses several methods to store pixel information to render objects.

A. Loading Sprites Images with BRAMs

Inside the projects there are two BRAMs that store the value of mem files for sprite images.

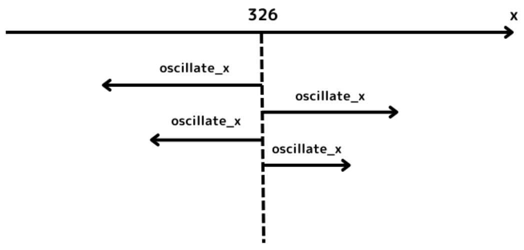
(i) undyne.png


(ii) attack_board.png

The sprites shown above are instantiated inside game_state.sv and player.sv, respectively.

i. undyne.png

undyne.png is a sprite image that appearts as the enemy of the game. The instance of image_sprite.sv for this sprite is created in game_state.sv, and it is shown throughout the game until the game enters the GameOver phase. There are two visual effects that are related to this image. One is to oscillate it when a player successfully gives damage to the enemy. When player.sv detects the player’s attack, output logic undyne_x sends value to game_state.sv to communicate where in the x coordinate the image should at a particular time to generate oscillating effect. The base x position of the image is 326, and undyne_x is a result of doing either adding or subtracting the value of logic oscillate_x, which is attenuated at every frame, to 326.



The process of attenuating oscillate_x is continued until the health bar of the enemy (which is described in a later section), finishes modifying its appearance.

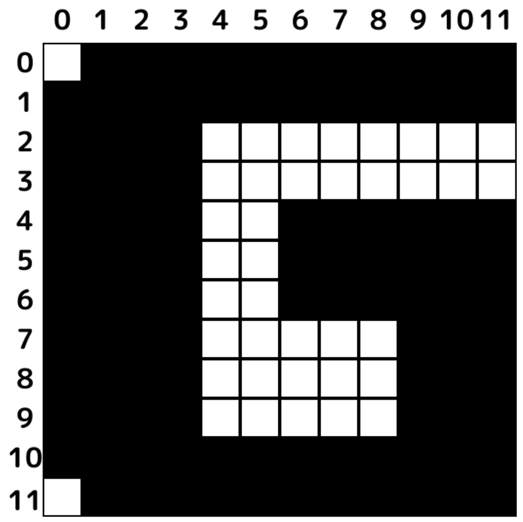
Another feature is alpha blending. During the Enemy phase, the image turns translucent . This is realized by separately dividing r,g, and v values of each pixel by 2.

ii. attack_board.png

This image is rendered during the Player phase. There is no additional visual effects to it. It only stays inside the white frame as a background image.

B. Fonts

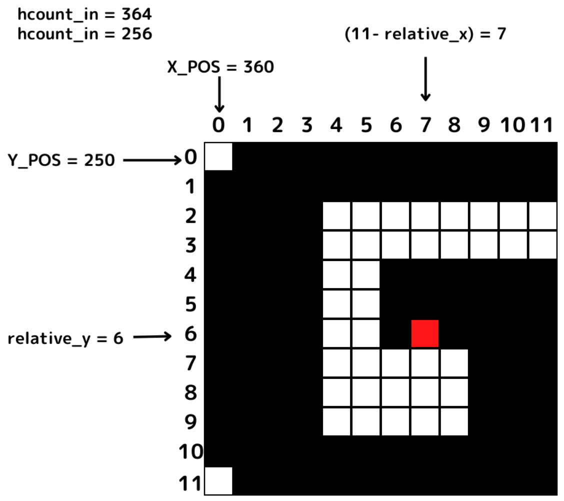
The original game of Undertale has a unique font of texts. I made an attempt to replicate this as similarly as possible by using 2d array to manually define each letter that appears in the game. The example array that indicates the structure of the letter "G" is show below.



Each letter is mostly expressed either with 12by12 ,12by8, or 8 by 8 array depending on the size of the letter. The module that stores all the representation is called fonts.sv, which has 2 parameters, 6 inputs, and 2 outputs:

- parameter X_POS = 128, Y_POS = 128
- input wire[10:0] hcount_in,
- input wire[9:0] vcount_in,
- input wire[5:0] letter_in,
- input wire[11:0] color_in,
- input wire[3:0] scale_in,
- output logic[11:0] pixel_out,
- output logic in_sprite

inside the module there is a big case statement that determines which letter to write its pixel value to pixel_out based on the value of letter_in. Each case condition in the module calculates the relative coordinate from the origin of the corresponding array (top-left corner). That is, we subtract X_POS and Y_POS from hcount_in and vcount_in, respectively, and assign the result to the internal logic, relative_x and relative_y. When the coordinate (relative_x, relative_y) is within the array, the value of the (11-relative_x)th bit of the (relative_y)th row of the corresponding array is written to in_sprite. The module writes the value of color_in to pixel_out when in_sprite is high, otherwise it writes 0 to pixel_out. When the relative coordinate is outside the range of the array in_sprite is also set to low.

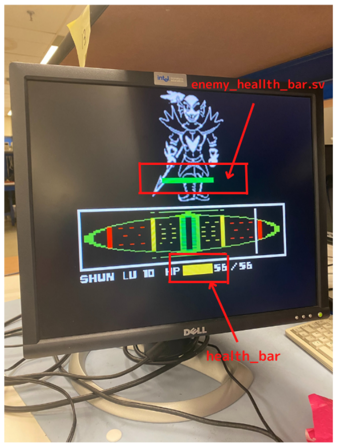


The 3-bit input scale_in indicates the rate of magnification of the letter. It is expressed by the number of bit shifts the module makes. If scale_in is equal to 1, meaning the size of the letter is scaled up by a factor 2, the size of the array is virtually expanded by 2. In the case of 12by12 array, it becomes 24by24 when checking if the relative coordinate (relative_x, relative_y) is inside the array. Because the values of relative_x and relative_y could be more than 11, and would not fit into the original size of the array, we remap them to the range of 0 to 11 by shifting left by 1. This way I could scale up the size of the letter by a factor of 2 to the power of scale_in. Instances of fonts.sv are created inside the module that need to draw texts.

C. Images without mem Files

Other images or visual effects are created either by using modules inheriting block_sprite.sv, or those that utilize the method used in fonts.sv.

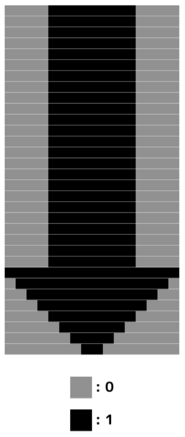
i) health_bar.sv / enemy_health_bar.sv



Examples of modules that inherit block_sprite.sv are health_ba.sv and enemy_health_bar.sv they have the same functionality of rendering a health bar in the game. They are made separately since each of them has other functionalities. Health_bar.sv, for instance, is also responsible for outputting numbers that indicate the current health of the player.
What they distinguish themselves from block_sprite.sv is that they will render a rectangle that consists of two sections each of which has a different color. They have an input logic called border_in, which provides them with the x value of where the border of two different color blocks is. By decreasing the value of border_x, the border of the two color sections shifts to the left, indicating that the health is decreased. If the value of border_x is modified at every frame, it will create an animation where the health decreases over time.

ii. arrow_sprite.sv

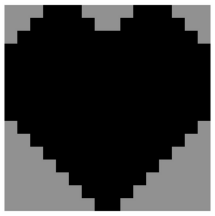
arrow_sprite.sv is one module that uses exactly the same method as fonts.sv for rendering an image of an arrow. This module is instantiated inside arrow.sv.



This pixel filling (represented by a unpacked 2d array, pixel_bram) shown on the left is used for rendering arrows facing all the four directions instead of making different one for each direction. This is realized by properly modifying the indices. In the case of rendering an arrow facing down, the indices would be pixel_bran[relative_y][relative_x]. When the arrow should be facing right, on the other hand, indices should be flipped, and pixel_bram[relative_x]-[relative_y] would give a disired output.
Adding to that, this module takes in the value of logic next_in and inversed_in from arrow.sv as inputs. When either of them is set to high, the color of the arrow changes from its normal color. The color of the arrow under each condition is defined by parameters.

iii. green_heart_sprite.sv

This is another module that utilizes the pixel filling method used in fonts.sv to render a heart on the monitor.



Hearts appear in serveral parts of the game. It is possible to change the color of the heart with a parameter. The position of the image is also defined by parameters, thus the heart will stay at the same place throughout the game.

V. GameOver

The GameOver phase is independent from the rest of the phases in the game (Menu, Player, Enemy). That is,the system enters the GameOver phase whenever the condition is satisfied. It indicates that the game is terminated by showing some animations and texts on the display.

A. Entering GameOver Phase

Whether the system should enter the GameOver phase is determined by an output logic of health_bar.sv, game_over_out. This logic is asserted high for a single clock cycle after the hp of the player reaches zero. The maximum hp of the player is 56, the player health decreases by 8 at every hit by an arrow. Inside health_bar.sv there is a logic called health_count which is reset to 0 at the initialization of the system. It is incremented every time the player gets hit by an arrow, and when it reaches 7, which means 7 arrows have hit the player in total, game_over_out is set to high.

B. Heart Sprite

Game_state.sv is responsible for rendering sprites and texts that appear during the game_over phase. When it detects the rising edge of game_over_out from health_bar.sv, it exits from the enemy_phase, and starts rendering for the GameOver phase. A 2-bit logic animation_phase is defined to control the process of the GameOver phase. The phase starts with animation_phase == 0. In this condition a red heart sprite appears in the middle of the game and it breaks in two after a short period of time. Green_heart_sprite.sv is used for this purpose. Then animation_phase is set to 1. When this happens the heart sprite dissappears from the monitor and it is followed by an animation of the heart falling apart. This animation is managed by heart_fall_apart.sv and heart_fall_apart_sprite.sv. The former module dictates the position of each piece of the heart that changes over time, and the latter module draws actual sprite of each piece of the heart. Heart_fall_apart.sv has logics for x and y values of all pieces, respectively, and they are modified at every frame. Heart_fall_apart_sprite uses almost the same method as fonts.sv to define two different sprites to draw as a piece of the heart, and it switches which sprite to draw into the monitor at every frame, which creates a visual effect of a small piece oscillating in the air.

C. Text Display

After the heart sprite breaks into parts, the text saying "GAMEOVER" starts to appeart on the display at the condition of animation_phase == 2. The text is made to have fade-in effect. A logic called timing_count is incremented at every frame, and when it reaches 2, the r, g, and b value of text color(represented by 12 bits) is incremented, respectively. This cycle is repeated 12 times for the pixel color to reach 12'FFF, which is complete white.

VI. Performance Discussion

Since this project entails rendering of a lot of visual effects, I anticipated that the number of BRAMs would be massive to the extent it conflicts with the memory limit. In fact, I could not load more than 1 undyne.png. I ended up using only two BRAMs for image_sprite.sv. For the rest of complicated images (including texts) to be displayed during the game, I used the method of storing all the data in a 2d packed array. This contributed to memory storage, and made it possible to include everything that is needed to draw the game scene. The system used 117 BRAM tiles output 135 available BRAMs(86.67%), and 100 out of 240 DSPs (41.67). Pipe-lining was incorporated into the camera interaction part to improve the accuracy of input the camera board takes in . Although there were no relevant, critical cases where timing issues break the game mechanics. There were a considerable amount of warning messages on the build log indicating that pipe-lining would improve the performance. There was also one apparent result that is thought to be caused by the timing issue is that arrows that appear on the monitor sometimes look like they jitter. According to the report the slack of the system was 0.883 nano seconds, and the skew -0.275 nano seconds. If I had had more time, I should have looked into the pipe-lining issues more, especially in the area that has to do with the game mechanics such as moving arrows.

VII. Reflection

My project has been developed to the extent it can be played as a game, which was the commitment of the game. A player can comfortably manipulates shields, select commands, and attack the enemy. The number of patterns of arrow movement are enough to give players excitement while they play the game. Graphics and animations look as similar as those from the original game. It would have been great I had had time to put music into the game. It took me unexpectedly a long time to implement and debug enemy.sv, one of the main modules of the system, and was not able to meet all of my stretch goals. Other than that I am satisfied with what has been achieved for my project.
One thing I could have done better is to think about the implementation and the structure of the entire system more thoroughly before I actually start write my codes. During the development there are so many cases where some functionalities that I did not consider are needed, and I ended up devicing a new module for them, leading me to restructure the system to conform to those new implementations. Anticipating and comprehending all the modules that would be needed to build the whole system would have spared me trouble of going back and forth during the development. This idea would be even more significant if I were to work as a team in the future.

VIII. Acknowledgments

I would like to express my sincere appreciation to our instructor Prof. Joe Steinmeyer, who provided me with sufficient help with completing my project. I am also grateful to Jay, Fischer and other TA/LAs who supported me throughout the semester.

VIII References

Source Code:
https://github.com/shuny42657/6.205_final_project.git

A.