HexaDodge

Clay Shubert, Zach Williams, and Robert Grady Williams

Introduction/Background

- Refresh:
  - Our final project was to design an infinite runner game using the VGA display on our BASYS3 board.
  - We named this game Hexadodge since the player model is a hexagon and the goal on the runner is to dodge the obstacles and avoid collisions.
  - We chose this project because we saw many challenging concepts that we wanted to attempt to solve. Namely, the VGA output port and Random Number Generation.

Design Details

- For our implementation:
- We were able to find various resources to help us understand how to drive the VGA display port on the BASYS3 board.
  - The driver code that we reference was created by Digikey and there were also references in other VHDL projects that involved the VGA display.
- RNG proved to be difficult, in the end we determined to pseudo randomly generate based off of the number of button presses.
  - The way this works is that we created a large array of 99 different obstacle sizes. The button press counter is an integer of range 0 to 99 that acts as the index to this array. The index is updated even if there is no button press that way the player can't just win without pressing a single button

Design Details Cont'd

- The player can control the player model with a single button press.
  - The center button for single button press to change up or down movement.
  - We attempted to implement two modes for controlling with two buttons for up and down however we were receiving some weird errors.
- We were able to find a function implementation for drawing strings on the VGA display. This allowed us to display "Game Over!" at the end of the game.
- We used the seven-segment display on the BASYS3 board to track the game score.
  - Unfortunately, we were unable to use the drawing strings function to display rapidly changing text, therefore the score is only visible on the seven-segment display, and we are unable to track the high scores on the screen.

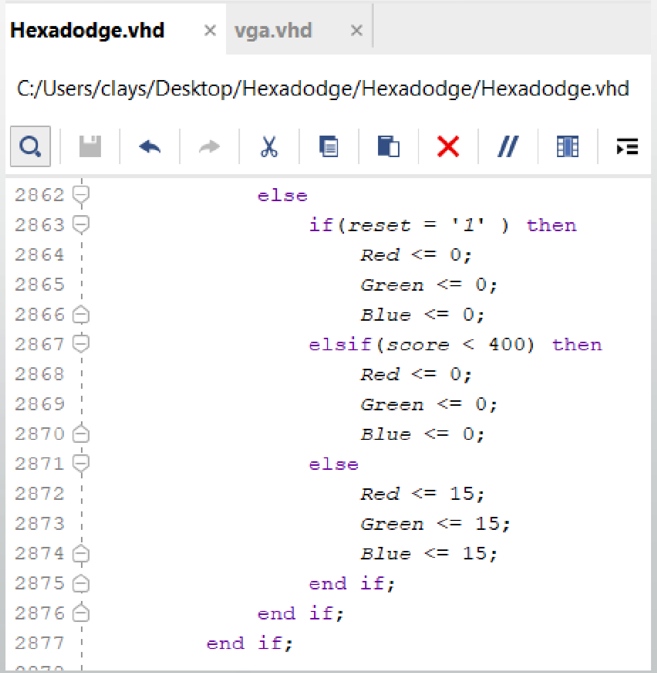
Experimental Results

- Our experimental results involved testing the game thoroughly by checking that the VGA display worked correctly as well as the randomness of the obstacles being generated.
- DEMO: We will now demo the game using the VGA display on the projector.

VGA

- For our VGA implementation, we were able to find reference driver software from GitHub user Dries007 (Copyright © 2016 Dries007 with MIT License (MIT)).
- This software included setting up the pixel clock.
- Our implementation still requires us to assign all the pixels, but the driver code essentially handles outputting our pixel assignments to the display.

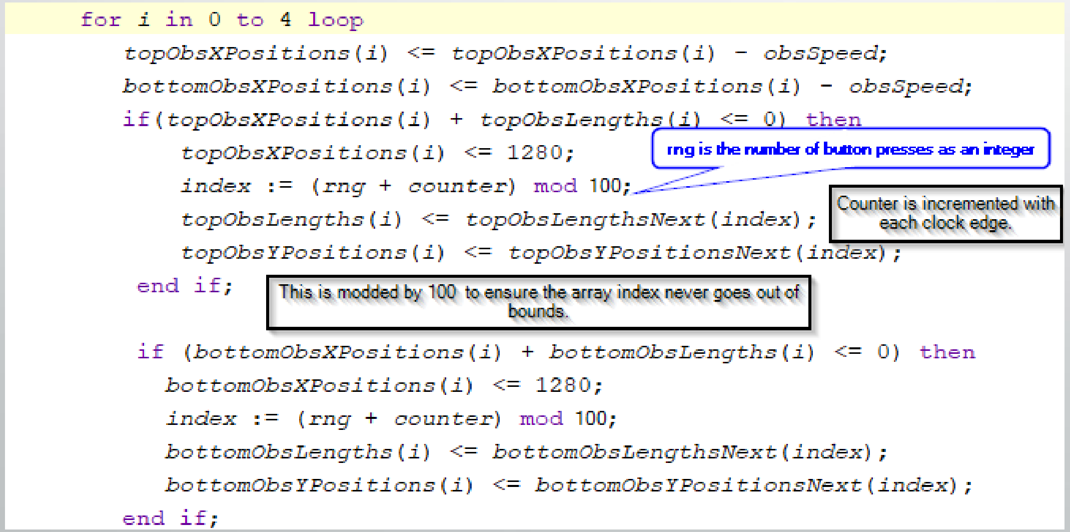
After checking for player model pixels and obstacles pixels, the rest of the screen is assumed to be background. This code segment demonstrates how we assign the colors for this region to be just the color black



```
            else
                if(reset = '1' ) then
                    Red <= 0;
                    Green <= 0;
                    Blue <= 0;
                elsif(score < 400) then
                    Red <= 0;
                    Green <= 0;
                    Blue <= 0;
                else
                    Red <= 15;
                    Green <= 15;
                    Blue <= 15;
                end if;
            end if;
        end if;
```

RNG

- Our implementation for RNG was based off the amount of player button presses and the clock.
- We created 2 arrays of 100 different Y positions for the obstacles on the top half and bottom half of the screen. The center of the screen is included in the top half array.
- The random number is generated by adding the counter (clock counter) and rng (number of button presses) together and then modding the total by 100 to ensure that array is always within bound



```
    for i in 0 to 4 loop
       topObsXPositions(i) <= topObsXPositions(i) - obsSpeed;
       bottomObsXPositions(i) <= bottomObsXPositions(i) - obsSpeed;
       if(topObsXPositions(i) + topObsLengths(i) <= 0) then
           topObsXPositions(i) <= 1280;
           index := (rng + counter) mod 100;
           topObsLengths(i) <= topObsLengthsNext(index);
           topObsYPositions(i) <= topObsYPositionsNext(index);
        end if;

        if (bottomObsXPositions(i) + bottomObsLengths(i) <= 0) then
          bottomObsXPositions(i) <= 1280;
          index := (rng + counter) mod 100;
          bottomObsLengths(i) <= bottomObsLengthsNext(index);
          bottomObsYPositions(i) <= bottomObsYPositionsNext(index);
       end if;
```

rng is the number of button presses as an integer

Counter is incremented with each clock edge.

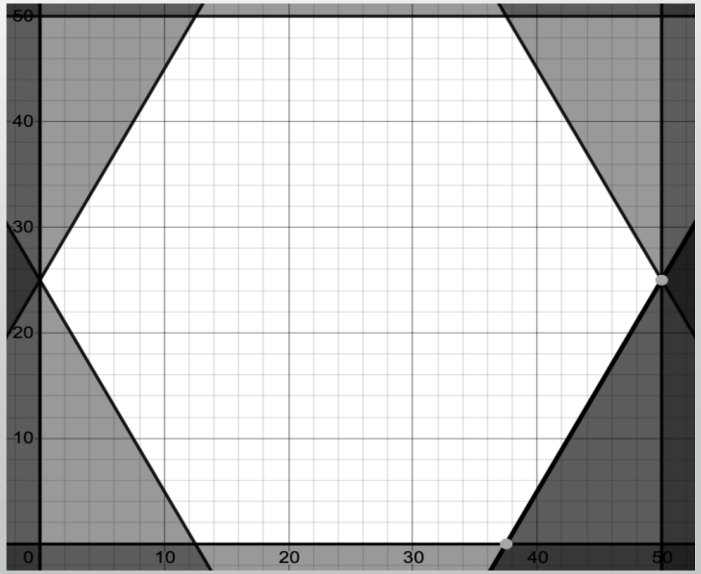
This is modded by 100 to ensure the array index never goes out of bounds.

Note: We considered the occasion where, knowing this, the player would try to cheat and not press any buttons. In this case, the clock will eventually cause there to be an obstacle in the center causing the player to press a button and thus starting their continuous motion.

Player/Obstacle Modeling

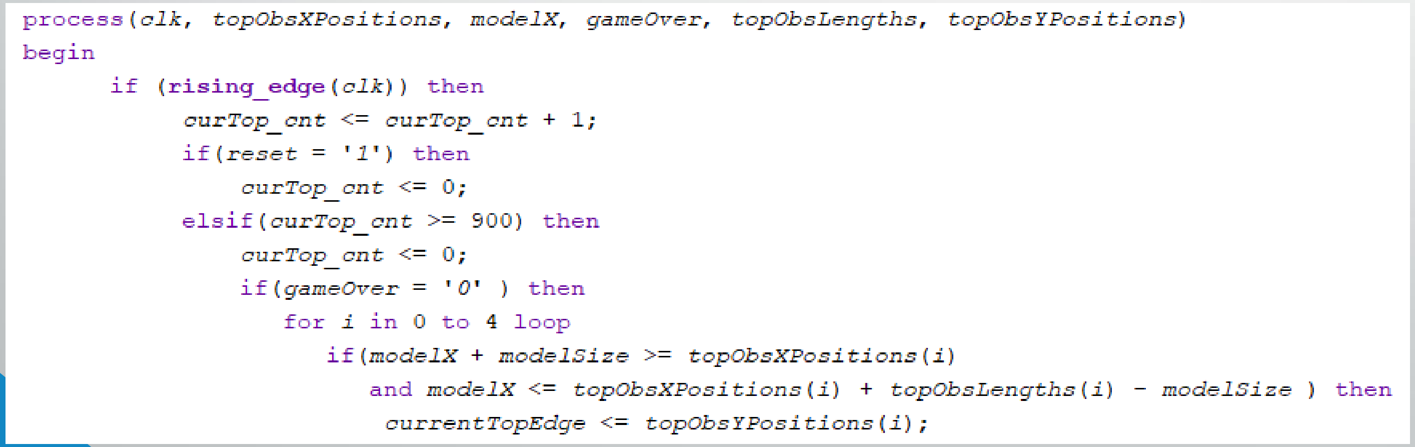
- For the obstacles, we defaulted all of the obstacles to have a height of 30 pixels and a length of 256 allowing us to make sure there are at most 5 on the screen at any given time for both top and bottom.
- For the player model, we were able to create a Hexagon by first creating a square.
- After this we used Desmos to create our model and determine the equations we needed to use to properly color in our player model on the display.
- The square was a 50x50 pixel model. Knowing this we can use a series of inequality expressions to "cutoff" the corners of the square and create a hexagon.
  - The equations below are included are wrapped in a conditional check to determine if the pixel scan from the VGA driver is within the player model bounds

(Y-modelY) < 25 – 2*(X-modelX)
(Y-modelY) > 25 + 2*(X-modelX)
(Y-modelY) < 2*(X-modelX) - 75
(Y-modelY) > 125 – 2*(X-modelX)



Movement Mechanics

- After each frame, the player is either moved up or down depending on their movement mode (up or down).
- In addition, after each frame, the obstacles are moved towards the left of the screen.
- This movement looks as though the player model is moving forwards when the obstacles are moving to meet the player model.



```
process(clk, topObsXPositions, modelX, gameOver, topObsLengths, topObsYPositions)
begin
      if (rising_edge(clk)) then
           curTop_cnt <= curTop_cnt + 1;
           if(reset = '1') then
               curTop_cnt <= 0;
           elsif(curTop_cnt >= 900) then
               curTop_cnt <= 0;
               if(gameOver = '0' ) then
                  for i in 0 to 4 loop
                     if(modelX + modelSize >= topObsXPositions(i)
                        and modelX <= topObsXPositions(i) + topObsLengths(i) - modelSize ) then
                         currentTopEdge <= topObsYPositions(i);
```

This is the process that we use to increment the top edges of the obstacles across the screen to the left.

Contributions

- Zach: I worked primarily on VGA display and integrating this into our implementation. I additionally assisted with obstacle and player modeling. This task required some math with pixels to create our shapes.
- Clay: I primarily worked on motion of the player model and obstacles. I also assisted with our RNG implementation
- Grady: I primarily was responsible for RNG and worked together with Zach and Clay on each of their individual parts.
- Our GitHub page for the project with our source code can be found at the link below
  - https://github.com/clayshubert/HexadodgeVHDLInfiniteRunner

References

- VGA Drawing Strings: https://github.com/Derek-X-Wang/VGA-Text-Generator
- Reference VGA: https://github.com/alisemi/fpga-projects/tree/master/Don'tHitTheBars