# EE 354 Final Project Report
# Eternal Maze Game

Name: Victor Hui (CECS) and Stephen Hornyak (CECS)
Semester: Spring 2021

## Abstract

For our EE354 project, we decided to challenge ourselves by creating a fun and challenging maze solving game by utilizing the VGA port on the Nexys4 FPGA board. Inspired by the classic Pokemon games and carnival stall activities, we reproduced their respective features in our maze game by using hardware only. We hope people who play this game will take up the challenge of solving all our mazes ranging from easy to difficult. Happy Solving!

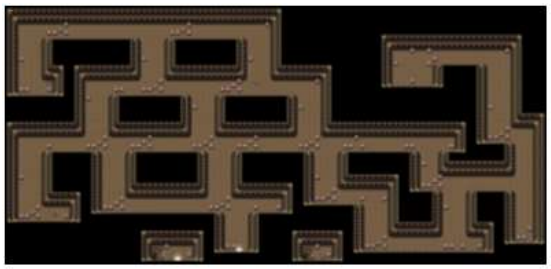


Figure 1: Maze Structure from Pokemon Games


Figure 2: Hidden Maze Mechanics from Pokemon


Figure 3: MegaWire Carnival Game, will buzz if the loop hits the metal wire

## Introduction and Background

For this project, we heavily relied on experience from 2 previous labs, the vga demo lab, and the vga moving block lab. Since our game used the VGA port of the Nexys4 FPGA board to display a user interface, we had to learn the board vga function extensively to fully utilize our possibilities, hence experimenting with the two lab demos were essential in aiding our understanding of how vga functioned and interacted with the monitor.

## The Design

The premise of our design came from our collective want to make a fun game, as we felt a game would be a more challenging and different path towards accomplishing the final project requirement for the class. In doing so, we also wanted to have the game be semi-simplistic and relatable, considering the deadline of 3 weeks out. Hence, we decided to design a maze game. Initially, we were wondering how we wanted to design the maze, we would either have a 2D array portray both the available spaces and the walls that are open from the current position or just store the available spots in coordinates where the player could travel to. We decided on the latter, as it would provide a smaller array and thus would allow for more mazes to be implemented while achieving the goal of being very simplistic by nature. Along with this design decision, we also decided to have some slight variance in the color scheme of the maze, making the startpoint/square a blue color, the endpoint a red color, the walls of the maze a black color, and the available path a neon green-like color so that there would be no confusion for the player. On top of this, the main player would also be a pink color to further distinguish between all the different colors and hues that are present in the maze. These color choices were chosen to boost the intuitiveness of the game so that anybody would be able to pick it up quickly.

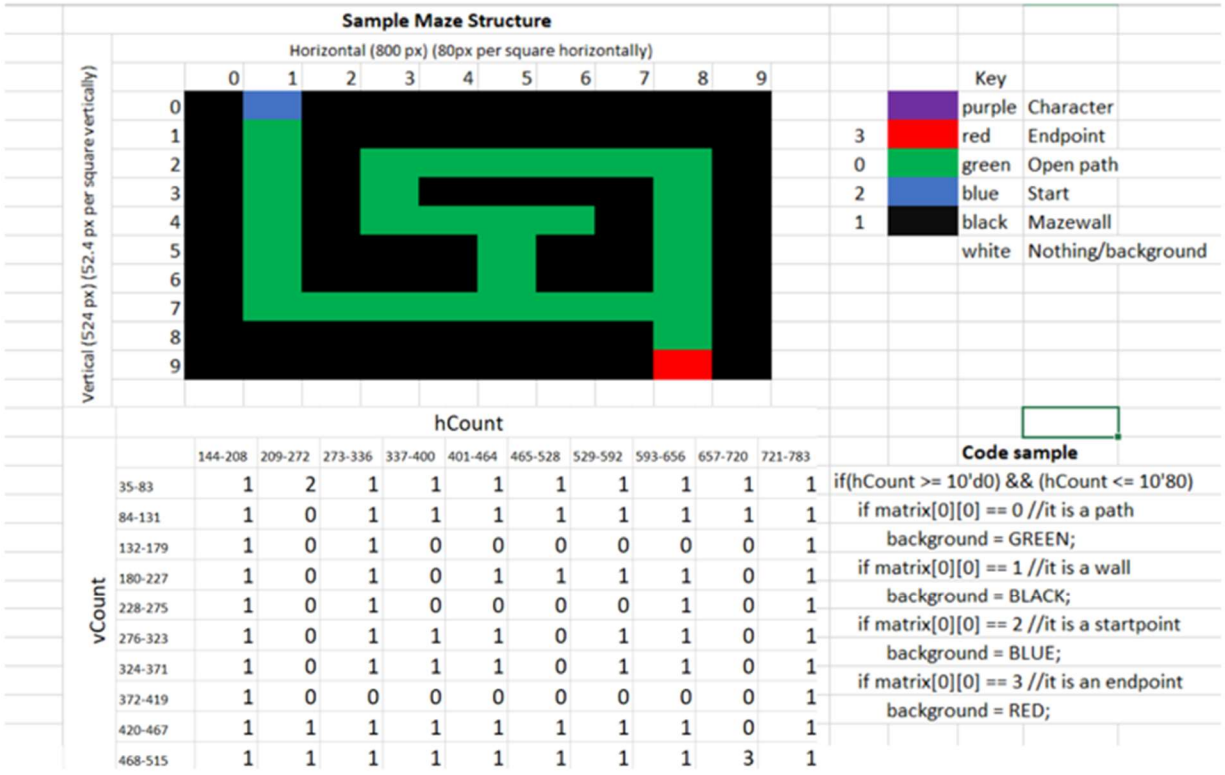



Figure 4: Graphic representation of the Maze + Key + sample maze matrix with coordinate range + pseudocode for displaying the maze

Along with the basics of the maze, we wanted to make the maze a little more complex. So, instead of just portraying the entire map to the player, we would limit the viewing field of the player, that way the player has much more of a challenge. Along with this limited view of the maze, we also implemented the

feature of not touching the walls or any of the obstacles at all. If the player were to run into a wall or obstacle, the player would be stuck there and must reset the game, where the player would restart from the beginning of the maze and have to reach the end of the same maze. Once the player has reached the end of the current maze, they would become stuck in the end zone. When they press the restart button, the player would be sent back to the beginning, but in a completely different maze, where the player would continue solving mazes that have already been generated but appear in a randomized fashion. In doing so, the player would continue on and on, with no end to the mazes. Thus, it's an eternal game with no end in sight as the mazes continually change once the end of the maze has been reached.

```
reg [1:0] x;

always @ *
begin
   if(maze_type == 0)begin
       if(((hCount >= 10'd144) && (hCount <= 10'd208)) && ((vCount >= 10'd35) && (vCount <= 10'd88)))
            x = maze1[0][0];
       else if (((hCount >= 10'd209) && (hCount <= 10'd272)) && ((vCount >= 10'd35) && (vCount <= 10'd88)))
            x = maze1[0][1];
       else if (((hCount >= 10'd273) && (hCount <= 10'd336)) && ((vCount >= 10'd35) && (vCount <= 10'd88)))
            x = maze1[0][2];
       else if (((hCount >= 10'd337) && (hCount <= 10'd400)) && ((vCount >= 10'd35) && (vCount <= 10'd88)))
            x = maze1[0][3];
       else if (((hCount >= 10'd401) && (hCount <= 10'd464)) && ((vCount >= 10'd35) && (vCount <= 10'd88)))
            x = maze1[0][4];
       else if (((hCount >= 10'd465) && (hCount <= 10'd528)) && ((vCount >= 10'd35) && (vCount <= 10'd88)))
            x = maze1[0][5];
       else if (((hCount >= 10'd529) && (hCount <= 10'd592)) && ((vCount >= 10'd35) && (vCount <= 10'd88)))
            x = maze1[0][6];
       else if (((hCount >= 10'd593) && (hCount <= 10'd656)) && ((vCount >= 10'd35) && (vCount <= 10'd88)))
            x = maze1[0][7];
       else if (((hCount >= 10'd657) && (hCount <= 10'd720)) && ((vCount >= 10'd35) && (vCount <= 10'd88)))
            x = maze1[0][8];
       else if (((hCount >= 10'd721) && (hCount <= 10'd784)) && ((vCount >= 10'd35) && (vCount <= 10'd88)))
            x = maze1[0][9];

       else if(((hCount >= 10'd144) && (hCount <= 10'd208)) && ((vCount >= 10'd84) && (vCount <= 10'd131)))
            x = maze1[1][0];
       else if (((hCount >= 10'd209) && (hCount <= 10'd272)) && ((vCount >= 10'd84) && (vCount <= 10'd131)))
            x = maze1[1][1];
       else if (((hCount >= 10'd273) && (hCount <= 10'd336)) && ((vCount >= 10'd84) && (vCount <= 10'd131)))
            x = maze1[1][2];
       else if (((hCount >= 10'd337) && (hCount <= 10'd400)) && ((vCount >= 10'd84) && (vCount <= 10'd131)))
            x = maze1[1][3];
       else if (((hCount >= 10'd401) && (hCount <= 10'd464)) && ((vCount >= 10'd84) && (vCount <= 10'd131)))
            x = maze1[1][4];
       else if (((hCount >= 10'd465) && (hCount <= 10'd528)) && ((vCount >= 10'd84) && (vCount <= 10'd131)))
            x = maze1[1][5];
       else if (((hCount >= 10'd529) && (hCount <= 10'd592)) && ((vCount >= 10'd84) && (vCount <= 10'd131)))
            x = maze1[1][6];
       else if (((hCount >= 10'd593) && (hCount <= 10'd656)) && ((vCount >= 10'd84) && (vCount <= 10'd131)))
            x = maze1[1][7];
       else if (((hCount >= 10'd657) && (hCount <= 10'd720)) && ((vCount >= 10'd84) && (vCount <= 10'd131)))
            x = maze1[1][8];
       else if (((hCount >= 10'd721) && (hCount <= 10'd784)) && ((vCount >= 10'd84) && (vCount <= 10'd131)))
            x = maze1[1][9];
```

Figure 5: The many lines to assign a cell to a range of horizontal and vertical pixels

Many conditional statements later…

```
       else if(((hCount >= 10'd144) && (hCount <= 10'd208)) && ((vCount >= 10'd468) && (vCount <= 10'd516)))
            x = maze4[9][0];
       else if (((hCount >= 10'd209) && (hCount <= 10'd272)) && ((vCount >= 10'd468) && (vCount <= 10'd516)))
            x = maze4[9][1];
       else if (((hCount >= 10'd273) && (hCount <= 10'd336)) && ((vCount >= 10'd468) && (vCount <= 10'd516)))
            x = maze4[9][2];
       else if (((hCount >= 10'd337) && (hCount <= 10'd400)) && ((vCount >= 10'd468) && (vCount <= 10'd516)))
            x = maze4[9][3];
       else if (((hCount >= 10'd401) && (hCount <= 10'd464)) && ((vCount >= 10'd468) && (vCount <= 10'd516)))
            x = maze4[9][4];
       else if (((hCount >= 10'd465) && (hCount <= 10'd528)) && ((vCount >= 10'd468) && (vCount <= 10'd516)))
            x = maze4[9][5];
       else if (((hCount >= 10'd529) && (hCount <= 10'd592)) && ((vCount >= 10'd468) && (vCount <= 10'd516)))
            x = maze4[9][6];
       else if (((hCount >= 10'd593) && (hCount <= 10'd656)) && ((vCount >= 10'd468) && (vCount <= 10'd516)))
            x = maze4[9][7];
       else if (((hCount >= 10'd657) && (hCount <= 10'd720)) && ((vCount >= 10'd468) && (vCount <= 10'd516)))
            x = maze4[9][8];
       else if (((hCount >= 10'd721) && (hCount <= 10'd784)) && ((vCount >= 10'd468) && (vCount <= 10'd516)))
            x = maze4[9][9];
       else
            x = 3;
   end

   if (( (hCount < (xpos - 100)) || (hCount > (xpos + 100)) || (vCount < (ypos - 100)) || (vCount > (ypos + 100 )) ))
       x = 1;

   if ((hCount == xpos) && (vCount == ypos)) begin
       if(x == 0)begin
           go = 1;
           end_of_maze <= 0;
       end
       else if (x == 3) begin
           go = 0;
           end_of_maze <= 1;
       end
       else
           go = 0;
   end
end

assign mazewall = x;
```

← Display black for pixels outside of the 100px limit for the dim maze effect

In terms of coding the maze, the general design was fairly straightforward with little deviance from an initial maze. The largest difficulty of coding the design was implementing it into VGA. The FPGA has a

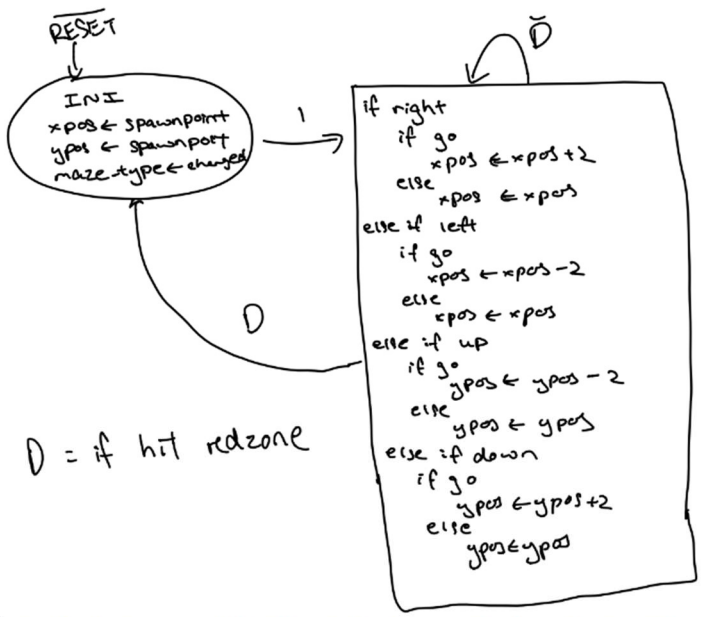
current state of the maze, which has the exact locations of where the correct path is and where the beginning and end are as well as where the player is stored in a 10 by 10 8 bit array. To portray that to the VGA, we had to check if the vCount and the hCount of the VGA display was over a specific area. If it was, then the code would reference that part of the array to decide to portray that area with a specific color. Therefore, a majority of our implemented code is successive conditional blocks (~500 lines) and initializations of each maze that is present in the game. In order to obtain the coordinates that each cell in the 2d Maze array would be accountable for, we also had to do substantial calculations outside of the Verilog code so that each cell would be referenced correctly.

```
parameter RED   = 12'b1111_0000_0000; //the end
parameter PURPLE = 12'b1111_0000_1111; //the character
parameter GREEN = 12'b0000_1111_0000; //open path
parameter BLUE = 12'b0000_0000_1111; //start
parameter BLACK = 12'b0000_0000_0000; //maze walls
parameter WHITE = 12'b1111_1111_1111; //background
parameter RANDO = 12'b0011_0000_1111; //walls2
```

```
always@ (*) begin
    if(~bright )    //force black if not inside the display area
        rgb = 12'b0000_0000_0000;
    else if (block_fill)
        rgb = PURPLE;
    else if (mazewall == 0)//path
        rgb = GREEN;
    else if (mazewall == 1)//wall
        rgb = BLACK;
    else if (mazewall == 2)//startpoint
        rgb = BLUE;
    else if (mazewall == 3)//endpoint
        rgb = RED;
    else if (mazewall == 4)//nothing
        rgb = WHITE;
    else
        rgb= background;
end
```

Figure 6: Setting the background color

Our state machine is simple and heavily references the lab vga_movingblock's state machine. On reset, or if the player finishes the current maze (i.e. lands in the red zone at the end of the maze), the code would respawn the player at the beginning of a fresh maze, of which the fresh maze is also randomized from our one of four preloaded mazes. The code has been designed so that adding new mazes into the game is simply the matter of importing new arrays in the code, this was done purposefully so that it would be easy to expand the game if we wanted to. The core of the state machine is simple, we have four states, up, down, left, right, and they are all responsible to move the player 2 pixels into their pointed direction. Though the state machine may appear simple in our game design, the bulk of our efforts went into designing how the mazes will be displayed, and different features of the game (as mentioned above) that would make this game more fun.




```
assign block_fill=vCount>=(ypos-5)&&vCount<=(ypos+5)&&hCount>=(xpos-5)&&hCount<=(xpos+5); //size of block player control

reg go; //1 bit register //for checking if the wall is hit
reg[1:0] maze_type = 2'b00;
reg end_of_maze;
reg[1:0] changed = 2'b00;

always@(posedge clk, posedge rst)
begin
    if((rst)) //if a wall is hit or rese|| (go != 1)
    begin

        //rough values for start of maze
        xpos <= 241;
        ypos <= 108;
        maze_type <= changed;

        if(changed[1] == 1'b1 && changed[0] == 1'b0) begin
            xpos <= 175;
            ypos <= 108;
        end
        else if (changed[1] == 1'b1 && changed[0] == 1'b1) begin
            xpos <= 752;
            ypos <= 203;
        end


    end
    else if (clk) begin

    /* Note that the top left of the screen does NOT correlate to vCount=0 and hCount=0. The display_controller.v file has
    the
        synchronizing pulses for both the horizontal sync and the vertical sync begin at vcount=0 and hcount=0. Recall that
        after
        the length of the pulse, there is also a short period called the back porch before the display area begins. So
        effectively,
        the top left corner corresponds to (hcount,vcount)~(144,35). Which means with a 640x480 resolution, the bottom right
        corner
        corresponds to ~(783,515).
    */
        if(right) begin
            if(go == 1)
                xpos<=xpos+2;
            else
                xpos<=xpos;
        end
        else if(left) begin
            if(go == 1)
                xpos<=xpos-2;
            else
                xpos<=xpos;
        end
        else if(up) begin
            if(go == 1)
                ypos<=ypos-2;
            else
                ypos<=ypos;
        end
        else if(down) begin
            if(go == 1)
                ypos<=ypos+2;
            else
                ypos<=ypos;
        end

        if(end_of_maze == 1) begin
            changed <= maze_type + 1;
        end
    end
end
```

Figure 7: State machine sketch + code

The user controls their character by the 5 buttons on the FPGA, pressing BtnU will cause the user character to move up, BtnL will cause the user character to move left, BtnR will cause the user character to move right, BtnD will cause the user character to move down, and BtnC will reset the maze game. Since our game relies on the VGA port to display the game UI, the user will also need to obtain a VGA cable and a VGA monitor so that they can be connected and the game can be displayed.

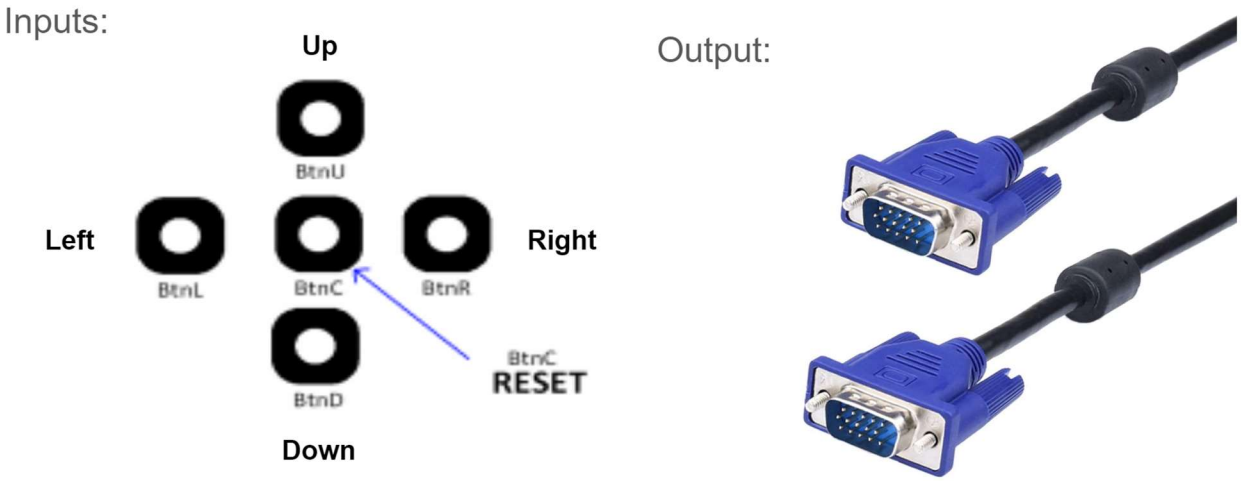



Figure 8: Input and Output

Although the code may seem simple by product, the game design, including brainstorming features, UI design and final project polishing took the most time. Designing each mazes to be progressively more difficult also took a significant amount of time.

## Test methodology

For our testing methodology, our philosophy behind the testing of the project was to catch any errors as quickly as possible and to not have to run through synthesis, implementation and creating a bitstream every time in order to properly debug the program. So, any errors that were related to the Verilog code were luckily caught after the synthesis period in terms of critical warnings. These warnings helped to serve as markers where the bitstream generation might fail and thus, we were able to efficiently debug those issues. In terms of the features that we were to debug, the only option was to run through the entire process mentioned above. Such process was used in implementing the limited view feature, the actual mazes themselves, and the varied mazes that appear. Another factor in our test methodology was trying to achieve the correct screen resolution in our testing. In order to provide a clear idea of whether the maze is functional was to try and adjust our monitors to the number of pixels being used in the program. In the program, we used an area of 783 x 515 pixels. In order to achieve the same results every time and the same sizing of images, we attempted to constantly reframe and resize our monitors to this aspect ratio.

## Conclusion and Future Work

In conclusion, we began with an idea of having a fun, yet semi-challenging maze game appear and over the course of 3 weeks, we were able to implement that while adding some features to give the maze

game a bit of a unique push to it. By allowing the player only a small portion of visibility, we were able to increase the difficulty of the mazes on top of the already complex mazes that were created. For future work, we'd like to implement more game-like features into our maze game, thus making it even more challenging and even more exciting. Some of the features we'd like to add are an actual randomly generated maze, more obstacles and helpful items for the players, larger/more mazes with greater variety, and better game mechanics, such as including a score counter, better designed player and more color variety. We fully enjoyed the labs throughout this semester, although at first we thought that 3 hour lab sessions would be extremely difficult, it turned out to be quite a nice change of pace from our other classes. To be able to just casually chat with our lab mate and lab TAs while working was nice, and we felt that the lab TAs were constantly trying to help out. Thank you for this semester.