ECE 385 Final Project

Spring 2023

Minecraft Theme
3D Arrow Shooting Game

Jerry Chang, Hsin Chen
HL/9:45-10:00
Honyi Li

Contents

1 Introduction

In our final project, we designed and built a semi-3D first person shooting game by implementing a raycasting [3] engine on the FPGA board. This semi-3D fps game is based on Minecraft [2] texture with basic Minecraft blocks that can be randomly generated through software during initialization. Semi-3D means that the player is actually still in a 2D space, in which Steve can only walk forward/ backward/ left/ right and rotate his head horizontally, however, the casting of walls and skeletons will make the visual appears like a 3D game. Player is able to shoot skeletons randomly generated across the map to accumulate scores. Every time Steve respawns, he will have 10 hearts indicating current health and whenever he gets hit by an arrow, one heart will be deducted.

2 Block Diagram

Below is the overall block diagram of our design.

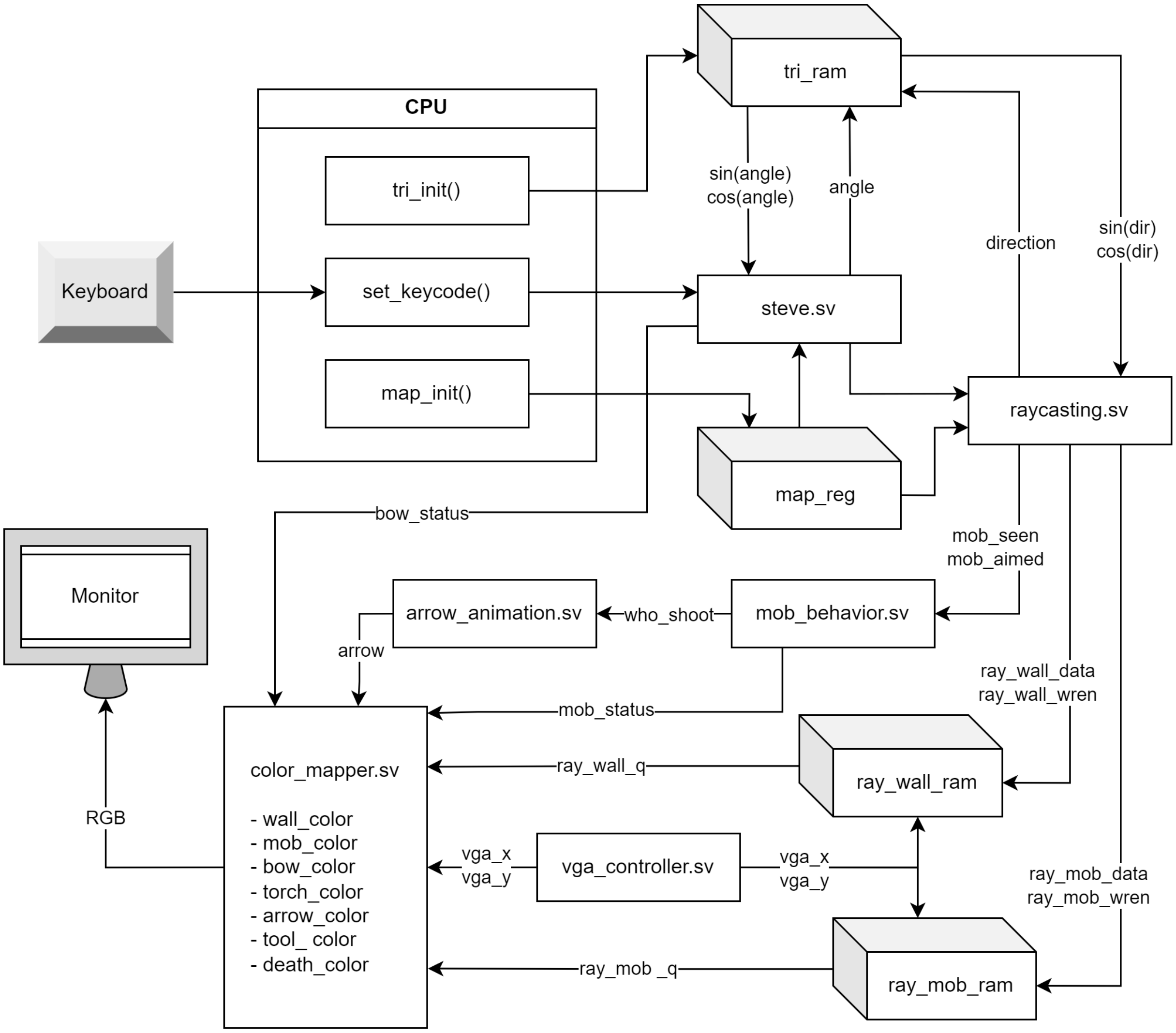


Figure 1: Block Diagram for Game Design

3 RTL Block Diagram

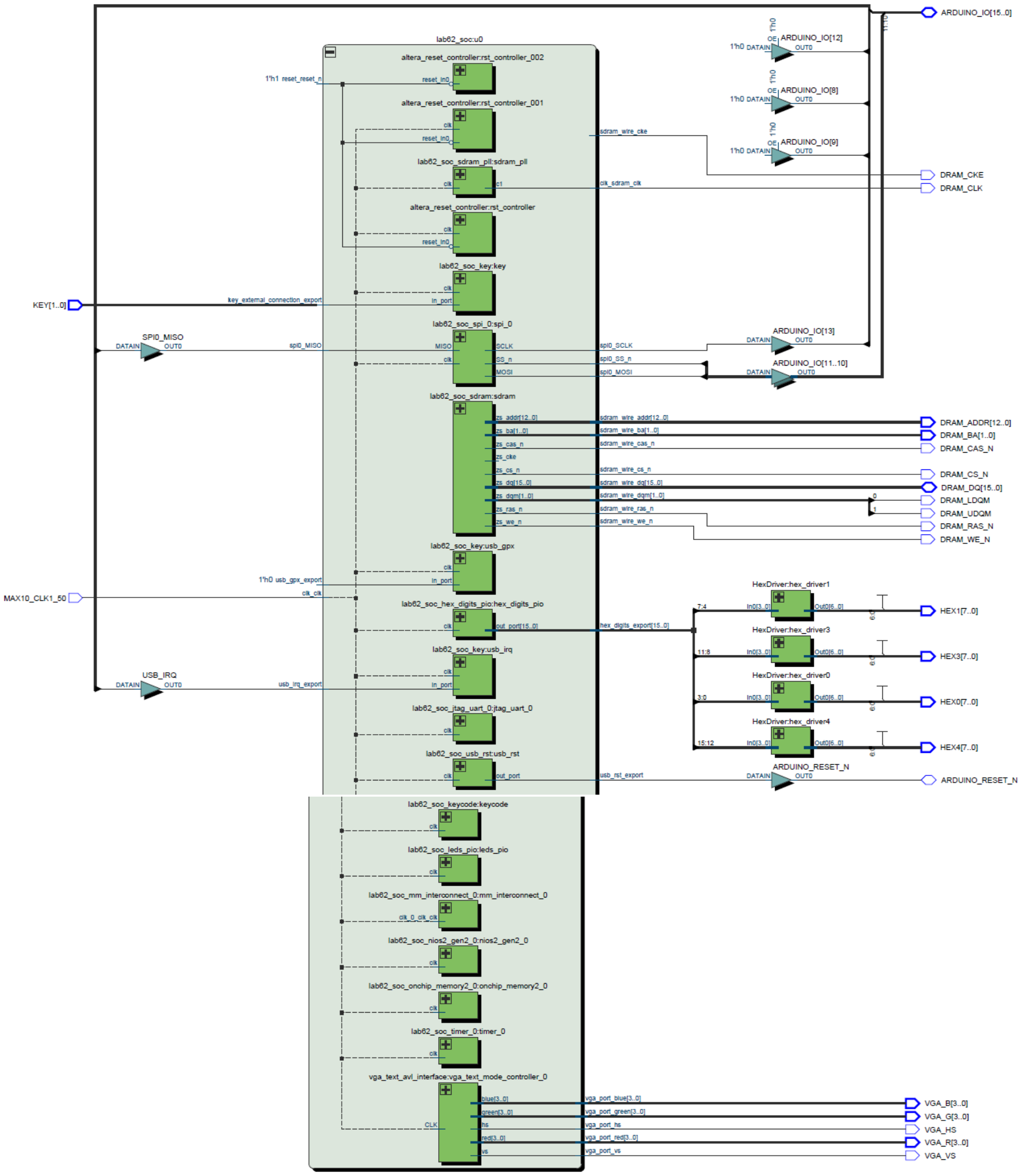


Figure 2: Block Diagram for Game Design

4 Design Features

4.1 Ray Casting

One of the most crucial part of this project is the implementation of the ray casting engine using SystemVerilog. Ray casting is a rendering technique to create a 3D perspective in a 2D map. The basic concept of ray casting is shooting a virtual light ray for every pixel columns of the monitor display and then calculating the distance the ray travels before hitting an object. We defined the small angle difference between each pixel light ray is 0.15 degree, and thus derived the player's angle of view a total of 0.15 * 640 = 96 degrees. However, it is impractical to sync the pixel signal from VGA interface and the execution of raycasting algorithm, which requires all the calculation to be finished within one VGA clock cycle (25Hz). Instead, the engine does 640 calculations sequentially, and then stores the results into a 640 word RAM; meanwhile, the color mapper module can access the result for each column from the RAM to further calculate for VGA color signals. The SystemVerilog code of our raycasting engine can be found in 8.1.

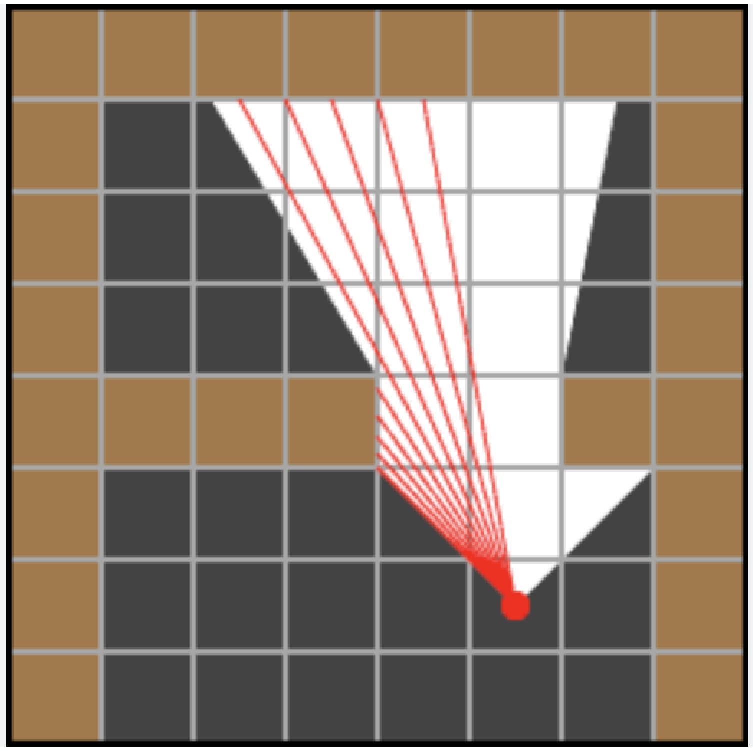


Figure 3: Ray Casting on 2D Map

4.2 Map

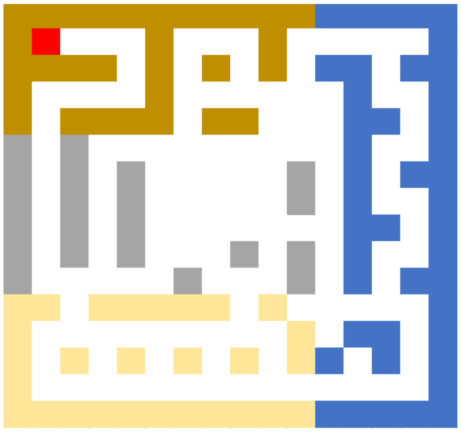


Figure 4: Map

The figure shown above is our 2D map. Currently, the map itself is not randomly generated. However, the blocks that end up appearing as walls are randomly generated by our C code running on Nios-II cpu. You can find the C code of how we generated the map in 8.2.

4.3 Texture blocks

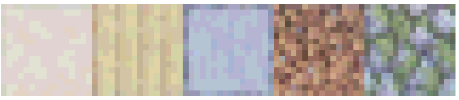


Figure 5: Texture Blocks

We implemented 4 different kinds of texture blocks to choose from. Each texture is stored as a 16*16*3 bit array. 16*16 corresponds to the 256 blocks on the texture block, and the 3 bits enables us to locate the correct rgb combination from another palette file. The texture blocks are created by Ian's Tool [1], which is a python script that turns a png file into compatible .sv and .mif files that could be used in SystemVerilog and synthesized on the board. However, knowing which texture block to display is not enough. For every pixel on the screen, we have to know which pixel within the texture block should be displayed. First, we have to calculate the height of the wall. This is calculated first through the distance between steve and the displayed wall $height_{wall} = \frac{122880}{distance_{wall}}$. The 122880 constant is just an experimental result that works well. Besides that, the raycasting.sv module outputs which column from 0 15 should be displayed. With that data known, we can then calculate the y pixel we need to access from the RAM. Since each wall's height is two blocks, we separate the calculations into 2 parts, one of which calculates when the displayed pixel is above half line of the screen, and the other is for pixels below. With those data, we can get to the x, y pixel that should be accessed from the RAM files. The full code could be found in 8.3.

4.4 Skeleton

The skeleton in our design is regarded as a cylinder, and always facing toward the player. The display of the mobs on the screen is implemented using the modified raycasting algorithm; we tried to figure out the distance from the mob's position to the line of the light ray as the horizontal offset from the center of the mob, and the distance difference from the mob to the original point when the ray hits the wall. For the simplicity and efficiency of calculation, we created a 256 * 4 bits array to hold mobs' statuses on the 16 by 16 map, and the 9 different states are:
$(0000)_2$: simply indicates that there is no mob in this square
$(1000)_2$: newly spawned mob
$(1001)_2$: mob appears in players POV and is close enough to detect the player
$(1010)_2$ to $(1011)_2$: bow is being charged until shot is made
$(1100)_2$ to $(1111)_2$: animation after mob is killed, and mob is again semi-randomly generated in an empty square 5 blocks away from Steve's location using a hash function.
The SystemVerilog code of mob behaviour handler can be found in 8.4.

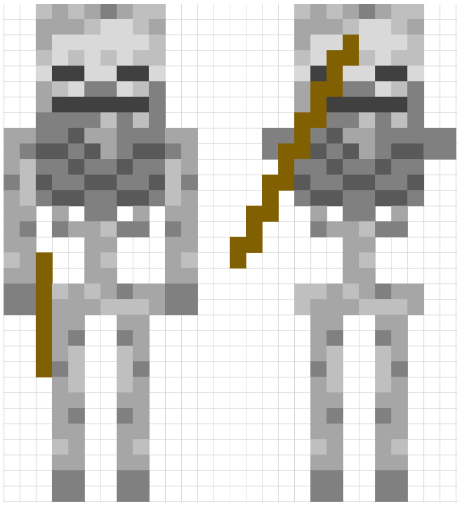


Figure 6: Skeletons at different states

4.5 Steve

Steve is the main character/user of our game. Users are able to input w/a/s/d for the control of translation and left_button/right_button for the rotation of vision. The I/O part is dealt with NIOS-II cpu, which sends keycode signal to fpga and later be used to change the position/angle of steve. Aside from controlling the movement, steve.sv also controls the states of the user (i.e. Alive/Dead) and the states of the bow to indicate which bow image should be displayed at the moment. The code for steve.sv could be found in 8.5

4.6 Bow and Arrow Animation

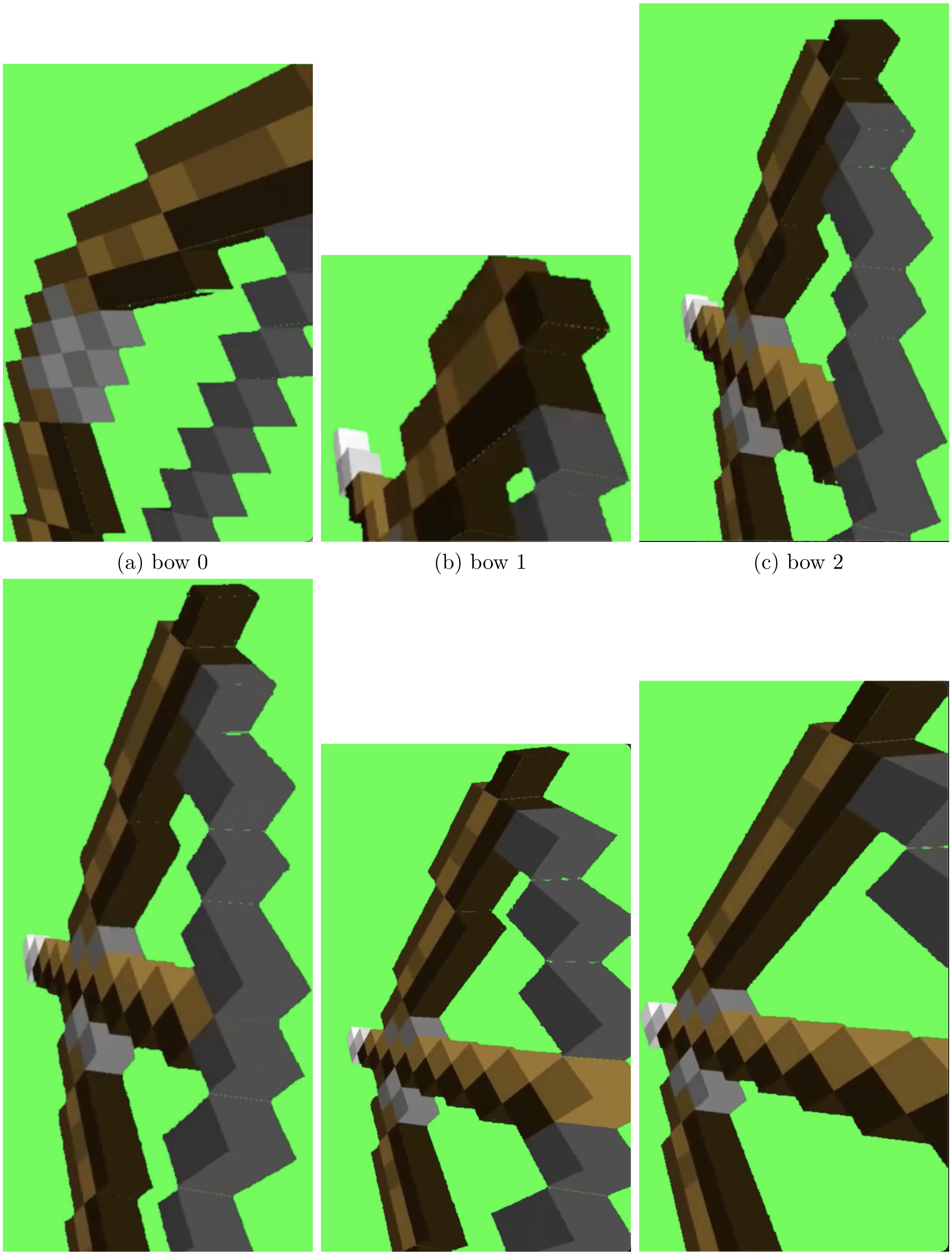
4.6.1 Bow

Since the bow should be interactive with user's inputs, for bow display, we created 5 different bow images and a bow state machine that indicates which bow image should be displayed at the moment. Whenever the user presses the space bar, the bow will transition from static bow image to a series of bow images that appear as an animation. In order to fit the bow images in our RAM, we designed it such that the first and the last bow image has higher resolution while the ones in between has lower resolution and is sort of stretched out. Each bow images are customized such that we're using the smallest amount of space and stretched and translated to the correct position on the VGA display. The bow state machine could be found in 8.5. The bow images could be found in Figure 7.

4.7 Arrow Shot Detection and Sidestep

The animation for the arrow shot by the user is simply two sizes of crosses that displays quickly such that the arrow appears to be shot to the facing direction. When the arrow is shot, the game will decide whether the mob will die or not by the current facing angle of the user. If the user currently faces a mob, the mob will die and the user will therefore receive a point.

The arrow shot detection from mobs to the user is also simple. We decide whether the user gets shot by calculating the user's position while the mob release the arrow and the user's position after the arrow arrives at the aimed spot. If the user is still in line with the arrow, the user gets shot, and vice versa. If the user gets shot, one heart will be deducted from the health bar. Bow display logic and the arrow shot detection logic could be found in 8.6.



(a) bow 0

(b) bow 1

(c) bow 2

(d) bow 3

(e) bow 4

(f) bow 5

Figure 7: Bow Images

5 Module Descriptions

5.1 Main Modules

Module: lab7.sv
Inputs: MAX10_CLK1_50, [1:0] KEY, [9:0] SW
Outputs: [9:0] LEDR, [7:0] HEX0, [7:0] HEX1, [7:0] HEX2, [7:0] HEX3, [7:0] HEX4, [7:0] HEX5, DRAM_CLK, DRAM_CKE, [12:0] DRAM_ADDR, [1:0] DRAM_BA, DRAM_LDQM, DRAM_UDQM, DRAM_CS_N, DRAM_WE_N, DRAM_CAS_N, DRAM_RAS_N, VGA_HS, VGA_VS, [3:0] VGA_R, [3:0] VGA_G, [3:0] VGA_B
Inouts: [15:0] DRAM_DQ, [15:0] ARDUINO_IO, ARDUINO_RESET_N
Description: Top level module for the project.
Purpose: This module is the main module that connects all the smaller modules in the processor.

Module: vga_text_avl_interface.sv
Inputs: CLK, RESET, AVL_READ, AVL_WRITE, AVL_CS, [3:0] AVL_BYTE_EN, [11:0] AVL_ADDR, [31:0] AVL_WRITEDATA
Outputs: [31:0] AVL_READDATA, [3:0] red, [3:0] green, [3:0] blue, hs, vs
Description: This is the interface module on the Platform Designer.
Purpose: This unit connects other smaller modules and has the Avalon-MM slave port which complete read and write operations requested by the Nios II processor.

Module: steve.sv
Inputs: Clk, Reset, [7:0] keycode, [255:0][4:0] map, [255:0][3:0] mob, [31:0] tri_center, [7:0] who_mem, hit
Outputs: [15:0] steve_x, [15:0] steve_y, [15:0] angle, [3:0] bow_state, shoot, [3:0] health, alive, respawn
Description: This unit continuously updates and outputs the information about the main character, Steve, including position, direction, and the hit points.
Purpose: Outputs of this unit such as position and direction are used in further raycasting computations, while hit points signal is wired to the color mapper module to be displayed on the screen.

Module: mob_behavior.sv
Inputs: Clk, alive, [255:0][4:0] map, [15:0] steve_x, [15:0] steve_y, [7:0] mob_seen, [7:0] mob_aimed, shoot, respawn
Outputs: [255:0][3:0] mob_map, [7:0] who_shoot, [6:0] score
Description: This module controls the behavior of the mobs such as spawning, shooting and death animation when getting shot.
Purpose: This unit outputs a map that stores the status of a skeleton on each square, where the most significant bit value being 1 indicates that a mob is spawned in this square.

Module: raycasting.sv
Inputs: Clk, Reset, [255:0][4:0] map, [255:0][3:0] mob, [15:0] steve_x, [15:0] steve_y, [31:0] tri_center, [31:0] tri_ray
Outputs: [9:0] ray_id, [31:0] ray_wall_data, ray_wall_wren, [31:0] ray_mob_data, ray_mob_wren, [7:0] mob_seen, [7:0] mob_aimed
Description: This module takes information of Steve and maps of the blocks and mobs as inputs, and stores the results of each ray into the RAM after computation.
Purpose: This module loops through 640 different directions of virtual light rays and calculates the total distance each ray travels before hitting a block or a mob.

Module: vga_controller.sv
Inputs: Clk, Reset
Outputs: hs, vs, pixel_clk, blank, sync, [9:0] DrawX, [9:0] DrawY
Description: This unit handles the signals needed for VGA display.
Purpose: This unit travels through every pixels on the screen, providing a specific position one at a time every pixel clock cycle, which is represented by DrawX and DrawY.

Module: arrow_animation.sv
Inputs: Clk, [7:0] who_shoot, [15:0] steve_x, [15:0] steve_y
Outputs: arrow, hit, [7:0] who_mem
Description: This unit handles arrow shot logic and arrow animation when steve is shot.
Purpose: This module takes in steve's position and decides whether he got shot by calculating the aimed position steve's latest position. It outputs a 1 bit signal that tells steve.sv whether he got shot or not. It also stores the mob that shot the arrow.

Module: Color_Mapper.sv
Inputs: Clk, vga_clk, vga_blank, [9:0] vga_x, [9:0] vga_y, [3:0] health, [6:0] score, alive, [31:0] ray_wall_q, [31:0] ray_mob_q, [3:0] bow_state, arrow_a
Outputs: [7:0] Red, [7:0] Green, [7:0] Blue
Description: This unit is a combinational logic circuit that takes information from RAMs as inputs and outputs RGB color.
Purpose: This unit decides the color for a single pixel according to the location on the screen.

5.2 Color Mapper Submodules

Module: wall_color.sv
Inputs: Clk, [9:0] vga_x, [9:0] vga_y, [31:0] ray_wall_q, [31:0] ray_mob_q
Outputs: [3:0] r, [3:0] g, [3:0] b
Description: This unit is a combinational logic circuit that takes the pixel currently displaying and decide what rgb values of either wall blocks or mobs to display. **Purpose**: This unit calculates the height of the wall based on the distance from steve to the wall and decides which pixel of different texture blocks should be displayed. This module is also responsible for the display of mobs since mobs are seen quite similarly as wall blocks.

Module: hp_color.sv
Inputs: [9:0] vga_x, [9:0] vga_y, [3:0] health
Outputs: [3:0] r, [3:0] g, [3:0] b, a
Description: This unit is a combinational logic circuit that takes the pixel currently displaying and decide what rgb values of the health bar to display. **Purpose**: This unit contains the data for small hearts in the bottom middle of the screen and decides how many of them should be displayed indicating current user's health.

Module: death_color.sv
Inputs: Clk, [9:0] vga_x, [9:0] vga_y, [6:0] score
Outputs: [3:0] r, [3:0] g, [3:0] b, a
Description: This unit is a combinational logic circuit that takes the pixel currently displaying and decide what rgb values to display when the user is dead. **Purpose**: This unit contains the data for the screen display when the user is dead, which consists of ultimate score, respawn button, a reddish screen, and the mob who killed steve.

6 Design Resources and Statistics

LUT	24467
DSP	220
Memory (BRAM)	1336320 bits
Flip-Flop	5623
Frequency	6.18 MHz
Static Power	98.25 mW
Dynamic Power	421.08 mW
Total Power	539.41 mW

Table 1: NIOS System Design Resources and Statistics Table

7 Conclusion

Through this project, we learned a lot about the process of designing and building a project we came up on our own. We successfully completed most of the bullet points we aimed for, except the wobbling walking animation of skeletons. One problem we encountered when during the project was determining the position of arrows and the actual display of arrow in air. This is hard due to the lack of inverse sin/cos function in systemverilog, thus we attempted the ask by calculating arrows positions using C code and pull the data from hardware, however, the process took too much time and the game will become laggy. Therefore we ended up not displaying the flying arrow. Regardless, we still finished a playable Minecraft themed maze game with 3D rendering through ray tracing, which is the most difficult achievment of this project.

References

[1] Ian Dailis. *Ian's Palettizer Tool*. URL: `https://github.com/iandailis/Palettizer`.

[2] Mojang Studios. *Minecraft*. 2023. URL: `http://www.minecraft.net`.

[3] Lode Vandevenne. *Lode's Computer Graphic Tutorial - Raycasting*. URL: `https://lodev.org/cgtutor/raycasting.html`.

8 Appendix

8.1 Appendix A

```
module raycasting (
    input Clk, Reset,
    input [255:0][4:0] map,
    input [255:0][3:0] mob,
    input [15:0] steve_x, steve_y,
    input [31:0] tri_center,
    input [31:0] tri_ray,
    output logic [9:0] ray_id = 10'h0,
    output logic [31:0] ray_wall_data,
    output logic ray_wall_wren,
    output logic [31:0] ray_mob_data,
    output logic ray_mob_wren,
    output logic [7:0] mob_seen,
    output logic [7:0] mob_aimed
);

    logic refresh_clk;
    clk_divider clk1 (Clk, 2, refresh_clk);

    int sin, cos, rx, ry;
    int sum_dx, sum_dy, sum_sx, sum_sy;
    logic [3:0] row_id, col_id;

    int dx, dy, sx, sy;
    int refsin, refcos, shadow, distx, disty;
    int mob_distx, mob_disty, mob_sx, mob_sy, deltax, deltay;

    logic who, hit_wall, hit_mob;
    logic [5:0] state = 6'h0;

    always_comb begin
        if (sin == 0) begin
            dx = 2147483647;
            dy = (tri_ray[15])? -rx: rx;
        end
        else if (cos == 0) begin
            dx = (tri_ray[31])? -ry: ry;
            dy = 2147483647;
        end
        else begin
            dx = (ry << 8) / sin;
            dy = (rx << 8) / cos;
        end

        sx = (dx * cos) >>> 8;
        sy = (dy * sin) >>> 8;

        refsin = {{16{tri_center[31]}}, tri_center[31:16]};
        refcos = {{16{tri_center[15]}}, tri_center[15:0]};
        shadow = (refcos*cos + refsin*sin) >> 8;
        distx = sum_dx * shadow;
        disty = sum_dy * shadow;

        if (tri_ray[31]) begin
            mob_sx = (sin + cos << 15) - {16'h0, sum_sx[15:0]} * sin;
            deltax = (sum_dx << 8)
                    + (cos - sin << 15) - {16'h0, sum_sx[15:0]} * cos;
            mob_distx = (deltax >> 8) * shadow;
        end
        else begin
            mob_sx = (sin - cos << 15) - {16'h0, sum_sx[15:0]} * sin;
            deltax = (sum_dx << 8) +
                (sin + cos << 15) - {16'h0, sum_sx[15:0]} * cos;
            mob_distx = (deltax >> 8) * shadow;
        end

        if (tri_ray[15]) begin
            mob_sy = - (sin + cos << 15) +
                {16'h0, ~sum_sy[15:0]+16'h1} * cos;
            deltay = (sum_dy << 8) +
                (sin - cos << 15) -
                {16'h0, ~sum_sy[15:0]+16'h1} * sin;
            mob_disty = (deltay >> 8) * shadow;
        end
        else begin
            mob_sy = (sin - cos << 15) +
                {16'h0, ~sum_sy[15:0]+16'h1} * cos;
            deltay = (sum_dy << 8) +
                (sin + cos << 15) -
                {16'h0, ~sum_sy[15:0]+16'h1} * sin;
            mob_disty = (deltay >> 8) * shadow;
        end
    end

    always_ff @ (posedge refresh_clk) begin
        if (Reset) begin
```

```
        state <= 6'h0;
        ray_id <= 10'b0;
    end
    else begin
        state <= state + 6'b1;
        if (state == 6'd63) begin
            if (ray_mob_data[31:24] < 8'h4)
                mob_seen <= ray_mob_data[15:8];
            else
                mob_seen <= 8'h0;

            if (ray_id == 10'd639)
                ray_id <= 10'b0;
            else
                ray_id <= ray_id + 10'b1;

            if (ray_id == 10'd319)
                mob_aimed <= ray_mob_data[15:8];
        end
    end
end

int lx, ly;
assign lx = {16'h0, steve_x[7:0], 8'h0};
assign ly = {{16'h0, ~steve_y[7:0]}+8'h1, 8'h0};

always_ff @ (posedge refresh_clk) begin
    ray_wall_wren <= 1'b0;
    ray_mob_wren <= 1'b0;

    if (state == 6'd0) begin
        sin <= {{16{tri_ray[31]}}, tri_ray[31:16]};
        cos <= {{16{tri_ray[15]}}, tri_ray[15:0]};
        row_id <= steve_y[11:8];
        col_id <= steve_x[11:8];

        if (tri_ray[15])
            rx <= -lx;
        else
            rx <= {{16'h0, ~steve_x[7:0]}+8'h1, 8'h0};
        if (tri_ray[31])
            ry <= -ly;
        else
            ry <= {16'h0, steve_y[7:0], 8'h0};
    sum_dx <= 0;
    sum_dy <= 0;
    sum_sx <= {16'h0, steve_x[7:0], 8'h0};
    sum_sy <= {16'h0, steve_y[7:0], 8'h0};
    hit_wall <= 1'b0;
    hit_mob <= 1'b0;
end
else if (state[0]) begin
    if (sum_dx+dx < sum_dy+dy) begin
        sum_dx <= sum_dx + dx;
        sum_sx <= sum_sx + sx;
        ry <= (tri_ray[31])? -65536: 65536;
        row_id <= (tri_ray[31])? row_id + 4'h1: row_id - 4'h1;
        who <= 1'b0;
    end
    else begin
        sum_dy <= sum_dy + dy;
        sum_sy <= sum_sy - sy;
        rx <= (tri_ray[15])? -65536: 65536;
        col_id <= (tri_ray[15])? col_id - 4'h1: col_id + 4'h1;
        who <= 1'b1;
    end

    if (ray_mob_data[7:5] > 0 && ray_mob_data[7:5] < 7)
        hit_mob <= 1'b0;
end
else begin
    if (~hit_wall & map[{row_id, col_id}][4]) begin
        hit_wall <= 1'b1;
        if (who) begin
            ray_wall_data <=
                {disty[31:16], 8'h0,
                sum_sy[15:12], map[{row_id, col_id}][3:0]};
            ray_wall_wren <= 1'b1;
        end
        else begin
            ray_wall_data <=
                {distx[31:16], 8'h0,
                sum_sx[15:12], map[{row_id, col_id}][3:0]};
            ray_wall_wren <= 1'b1;
        end
    end
    else if (~hit_wall & ~hit_mob & mob[{row_id, col_id}][3])
        begin
        hit_mob <= 1'b1;
```

```
                if (who) begin
                    ray_mob_data <= {mob_disty[31:16], {row_id, col_id},
                    mob_sy[24:19], mob[{row_id, col_id}][2:1]};
                    ray_mob_wren <= 1'b1;
                end
                else begin
                    ray_mob_data <= {mob_distx[31:16], {row_id, col_id},
                    mob_sx[24:19], mob[{row_id, col_id}][2:1]};
                    ray_mob_wren <= 1'b1;
                end
            end
        end

        if (state == 6'd62) begin
            if (~hit_wall) begin
                ray_wall_data <= 32'hffff0000;
                ray_wall_wren <= 1'b1;
            end
            if (~hit_mob) begin
                ray_mob_data <= 32'hffff0000;
                ray_mob_wren <= 1'b1;
            end
        end
    end

endmodule
```

8.2 Appendix B

```
void map_init(){
    int map[16][16] = {
        {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
        {1,0,0,0,0,1,0,0,0,1,0,0,0,0,0,1},
        {1,1,1,1,0,1,0,1,0,1,0,1,1,0,1,1},
        {1,0,0,0,0,1,0,0,0,0,0,0,1,0,0,1},
        {1,0,1,1,1,1,0,1,1,0,0,0,1,1,0,1},
        {1,0,1,0,0,0,0,0,0,0,0,0,1,0,0,1},
        {1,0,1,0,1,0,0,0,0,0,1,0,1,0,1,1},
        {1,0,1,0,1,0,0,0,0,0,1,0,1,0,0,1},
        {1,0,1,0,1,0,0,0,0,0,0,0,1,1,0,1},
        {1,0,1,0,1,0,0,0,1,0,1,0,1,0,0,1},
        {1,0,0,0,0,0,1,0,0,0,1,0,1,0,1,1},
        {1,1,0,1,1,1,1,1,0,1,0,0,0,0,0,1},
        {1,0,0,0,0,0,0,0,0,0,1,0,1,1,0,1},
        {1,0,1,0,1,0,1,0,1,0,1,1,0,1,0,1},
        {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
        {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}
    };

    for(int i=0; i<16; i++){
                for(int j=0; j<16; j++){
                  if(map[i][j] == 1){
                               if(j>10)
                                       ram->MAP_RAM[i*16+j] =
                           16 + (rand()%2) * 4 + (rand()%2);
                               else if (i>10)
                                       ram->MAP_RAM[i*16+j] =
                           16 + (rand()%2) * 8 + (rand()%2) * 2;
                               else if (i>4)
                                       ram->MAP_RAM[i*16+j] =
                           21 + (rand()%2) * 8 + (rand()%2) * 2;
                               else
                                       ram->MAP_RAM[i*16+j] =
                           26 + (rand()%2) * 4 + (rand()%2);
                  }else
                      ram->MAP_RAM[i*16+j] = 0x00;
        }
    }
}
```

8.3 Appendix C

```
module wall_color (
        input Clk,
   input [9:0] vga_x, vga_y,
   input [31:0] ray_wall_q, ray_mob_q,
   output [3:0] r, g, b
);

int unsigned dist_wall, h_wall;
int unsigned pxl_wall_y, pxl_mob_y;
int unsigned dist_mob, h_head, h_body;
logic [1:0] txr_sel_wall;
logic [3:0] pxl_mob_x;
logic [3:0] r0, g0, b0, r1, g1, b1;
logic a0, a1;

int unsigned rw, gw, bw, rm, gm , bm;
logic [9:0] rc, gc, bc, rf, gf, bf;

always_comb begin
    dist_wall = {16'h0, ray_wall_q[31:16]};
    h_wall = 122880 / dist_wall;

    if (vga_y > 239) begin
        pxl_wall_y = {18'h0, vga_y - 10'd240, 4'h0} / h_wall;
        txr_sel_wall = ray_wall_q[3:2];
    end
    else begin
        pxl_wall_y = 15 - {18'h0, 10'd239 - vga_y, 4'h0} / h_wall;
        txr_sel_wall = ray_wall_q[1:0];
    end

    dist_mob = {16'h0, ray_mob_q[31:16]};
    h_head = 40960 / dist_mob;
    h_body = 122880 / dist_mob;

    if (ray_mob_q[15:12] > 4'h0) begin
        pxl_mob_x = ~ray_mob_q[5:2] + 4'h8;
        pxl_mob_y = (vga_y + h_head - 240 << 3) / h_head;
    end
    else begin
        pxl_mob_x = 4'h0;
        pxl_mob_y = 0;
    end
```

```
end

always_comb begin
    if (h_wall > 159) begin
        rw = {28'h0, r0};
        gw = {28'h0, g0};
        bw = {28'h0, b0};
    end
    else if (h_wall > 32) begin
        rw = {28'h0, r0} * (h_wall - 32) >> 7;
        gw = {28'h0, g0} * (h_wall - 32) >> 7;
        bw = {28'h0, b0} * (h_wall - 32) >> 7;
    end
    else begin
        rw = 0;
        gw = 0;
        bw = 0;
    end

    if (vga_y < 80) begin
        rc = 10'h4;
        gc = 10'h4;
        bc = 10'h4;
    end
    else if (vga_y < 208) begin
        rc = (10'd207 - vga_y) >> 5;
        gc = (10'd207 - vga_y) >> 5;
        bc = (10'd207 - vga_y) >> 5;
    end
    else begin
        rc = 0;
        gc = 0;
        bc = 0;
    end

    if (vga_y > 399) begin
        rf = 10'h4;
        gf = 10'h4;
        bf = 10'h4;
    end
    else if (vga_y > 10'd271) begin
        rf = (vga_y - 10'd272) >> 5;
        gf = (vga_y - 10'd272) >> 5;
        bf = (vga_y - 10'd272) >> 5;
    end
    else begin
        rf = 0;
        gf = 0;
        bf = 0;
    end

    if (h_body > 159) begin
        rm = {28'h0, r1};
        gm = {28'h0, g1};
        bm = {28'h0, b1};
    end
    else if (h_body > 32) begin
        rm = {28'h0, r1} * (h_body - 32) >> 7;
        gm = {28'h0, g1} * (h_body - 32) >> 7;
        bm = {28'h0, b1} * (h_body - 32) >> 7;
    end
    else begin
        rm = 0;
        gm = 0;
        bm = 0;
    end

    if (vga_y + h_head > 239 && vga_y < 240 + h_body)
        a0 = 1'b1;
    else
        a0 = 1'b0;
end

texture_pack texture (
    Clk,
    {6'b0, ray_wall_q[7:4]},
    {6'b0, pxl_wall_y[3:0]},
    1'b1,
    txr_sel_wall,
    r0,
    g0,
    b0);
```

```
mob_texture mob (
    ~Clk,
    pxl_mob_x,
    pxl_mob_y,
    ray_mob_q[1:0],
    r1,
    g1,
    b1,
    a1);

always_comb begin
    if (a0 & a1) begin
        r = rm[3:0];
        g = gm[3:0];
        b = bm[3:0];
    end
    else if (vga_y > 240 + h_wall) begin
        r = rf[3:0];
        g = gf[3:0];
        b = bf[3:0];
    end
    else if (h_wall + vga_y > 239) begin
        r = rw[3:0];
        g = gw[3:0];
        b = bw[3:0];
    end
    else begin
        r = rc[3:0];
        g = gc[3:0];
        b = bc[3:0];
    end
end

endmodule
```

8.4 Appendix D

8.4.1 Mob Texture

```
module mob_texture (
    input Clk,
    input [3:0] pxl_x,
    input int pxl_y,
    input [1:0] texture_sel,
    output logic [3:0] r, g, b,
    output logic a
);

localparam [0:31][0:15][2:0] mob0 = {
    {0,0,0,0,2,3,2,3,4,3,2,2,0,0,0,0},
    {0,0,0,0,3,3,3,2,1,1,2,3,0,0,0,0},
    {0,0,0,0,3,1,1,1,1,1,1,2,0,0,0,0},
    {0,0,0,0,2,1,2,1,2,1,2,2,0,0,0,0},
    {0,0,0,0,1,6,6,1,1,6,6,1,0,0,0,0},
    {0,0,0,0,3,2,1,4,4,1,2,4,0,0,0,0},
    {0,0,0,0,3,6,6,6,6,6,6,4,0,0,0,0},
    {0,0,0,0,4,4,4,4,2,4,2,4,0,0,0,0},
    {0,0,3,3,4,4,5,3,3,4,4,4,3,3,0,0},
    {0,0,3,4,5,5,4,5,3,4,5,5,4,3,0,0},
    {0,0,3,2,4,4,5,4,4,5,4,4,2,3,0,0},
    {0,0,4,2,5,4,4,5,5,4,4,5,2,4,0,0},
    {0,0,3,2,4,5,5,4,4,5,5,4,2,3,0,0},
    {0,0,3,3,0,3,0,4,4,0,3,0,3,3,0,0},
    {0,0,3,4,0,4,3,3,2,4,4,0,4,3,0,0},
    {0,0,3,3,0,0,0,3,3,0,0,0,3,3,0,0},
    {0,0,2,3,7,0,0,3,2,0,0,0,3,2,0,0},
    {0,0,3,3,7,0,0,3,3,0,0,0,3,3,0,0},
    {0,0,4,4,7,2,3,2,3,4,3,3,4,4,0,0},
    {0,0,4,4,7,3,3,3,2,2,2,3,4,4,0,0},
    {0,0,0,0,7,3,3,0,0,3,3,0,0,0,0,0},
    {0,0,0,0,7,3,4,0,0,4,3,0,0,0,0,0},
    {0,0,0,0,7,3,2,0,0,2,3,0,0,0,0,0},
    {0,0,0,0,7,4,2,0,0,2,4,0,0,0,0,0},
    {0,0,0,0,0,3,2,0,0,2,3,0,0,0,0,0},
    {0,0,0,0,0,3,3,0,0,3,3,0,0,0,0,0},
    {0,0,0,0,0,3,4,0,0,4,3,0,0,0,0,0},
    {0,0,0,0,0,3,3,0,0,3,3,0,0,0,0,0},
    {0,0,0,0,0,2,3,0,0,3,2,0,0,0,0,0},
    {0,0,0,0,0,3,3,0,0,3,3,0,0,0,0,0},
    {0,0,0,0,0,4,4,0,0,4,4,0,0,0,0,0},
    {0,0,0,0,0,4,4,0,0,4,4,0,0,0,0,0}
};

localparam [0:31][0:15][2:0] mob1 = {
    {0,0,0,0,2,3,2,3,4,3,2,2,0,0,0,0},
    {0,0,0,0,3,3,3,2,1,1,2,3,0,0,0,0},
    {0,0,0,0,3,1,1,7,1,1,1,2,0,0,0,0},
    {0,0,0,0,2,1,7,7,2,1,2,2,0,0,0,0},
    {0,0,0,0,1,6,7,1,1,6,6,1,0,0,0,0},
    {0,0,0,0,3,7,7,4,4,1,2,4,0,0,0,0},
    {0,0,0,0,3,7,6,6,6,6,6,4,0,0,0,0},
    {0,0,0,0,7,7,4,4,2,4,2,4,0,0,0,0},
    {0,0,4,4,7,4,5,3,3,4,4,4,4,4,0,0},
    {0,0,4,7,7,5,4,5,3,4,5,5,4,4,0,0},
    {0,0,0,7,4,4,5,4,4,5,4,4,0,0,0,0},
    {0,0,7,7,5,4,4,5,5,4,4,5,0,0,0,0},
    {0,0,7,0,4,5,5,4,4,5,5,4,0,0,0,0},
    {0,7,7,0,0,3,0,4,4,0,3,0,0,0,0,0},
    {0,7,0,0,0,4,3,3,2,4,4,0,0,0,0,0},
    {0,7,0,0,0,0,0,3,3,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,3,2,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,3,3,0,0,0,0,0,0,0},
    {0,0,0,0,2,2,3,2,3,4,3,3,0,0,0,0},
    {0,0,0,0,2,3,3,3,2,2,2,3,0,0,0,0},
    {0,0,0,0,0,3,3,0,0,3,3,0,0,0,0,0},
    {0,0,0,0,0,3,4,0,0,4,3,0,0,0,0,0},
    {0,0,0,0,0,3,2,0,0,2,3,0,0,0,0,0},
    {0,0,0,0,0,4,2,0,0,2,4,0,0,0,0,0},
    {0,0,0,0,0,3,2,0,0,2,3,0,0,0,0,0},
    {0,0,0,0,0,3,3,0,0,3,3,0,0,0,0,0},
    {0,0,0,0,0,3,4,0,0,4,3,0,0,0,0,0},
    {0,0,0,0,0,3,3,0,0,3,3,0,0,0,0,0},
    {0,0,0,0,0,2,3,0,0,3,2,0,0,0,0,0},
    {0,0,0,0,0,3,3,0,0,3,3,0,0,0,0,0},
    {0,0,0,0,0,4,4,0,0,4,4,0,0,0,0,0},
    {0,0,0,0,0,4,4,0,0,4,4,0,0,0,0,0}
};

localparam [0:31][0:15][2:0] mob2 = {
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,2,3,4,3,2,0,0,0,0,0,0},
    {0,0,0,0,0,3,2,1,1,2,3,0,0,0,0,0},
    {0,0,0,0,1,1,1,1,1,1,2,0,0,0,0,0},
    {0,0,0,2,1,2,1,2,1,2,2,0,0,0,0,0},
    {0,0,0,1,6,6,1,1,6,6,1,0,0,0,0,0},
    {0,0,0,3,2,1,4,4,1,2,4,0,0,0,0,0},
    {0,0,0,3,6,6,6,6,6,6,4,0,0,0,0,0},
    {0,0,0,4,4,4,4,2,4,2,4,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,7,2,0,0,0,0,0,0,0,0,4,4,4,0},
    {0,0,7,0,2,0,0,0,0,0,5,3,4,5,5,0},
    {0,0,0,7,0,2,0,0,4,5,4,4,5,4,4,0},
    {0,0,0,7,0,0,2,5,4,4,5,5,4,4,5,0},
    {0,0,0,7,0,0,0,2,5,5,4,4,5,5,4,0},
    {0,0,0,0,7,0,4,4,2,0,4,4,0,3,0,0},
    {0,0,0,0,7,4,4,0,4,2,3,2,4,4,0,0},
    {0,0,0,0,0,7,4,0,0,0,2,0,0,0,0,0},
    {0,0,0,0,0,0,7,7,0,0,0,2,0,0,0,0},
    {0,0,0,0,0,0,0,0,7,7,7,0,2,0,0,0},
    {0,0,0,0,0,0,0,3,4,0,0,7,7,7,0,0},
    {0,0,0,0,0,0,3,3,0,0,3,3,0,0,0,0},
    {0,0,0,0,0,2,3,0,0,3,2,0,0,0,0,0},
    {0,0,0,0,3,3,0,0,3,3,0,0,0,0,0,0},
    {0,0,0,4,4,0,0,4,4,0,0,0,0,0,0,0},
    {0,0,4,4,0,0,4,4,0,0,0,0,0,0,0,0}
};

localparam [0:31][0:15][2:0] mob3 = {
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,3,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,3,0,0,0,0,0,0,0,0,3,0,0,0,0},
    {0,0,0,2,2,0,0,0,0,2,0,0,0,0,0,0},
    {0,0,2,2,2,0,0,0,0,0,2,2,0,0,0,0},
    {0,0,0,2,0,0,0,0,0,2,2,2,0,0,0,0},
    {0,0,0,0,0,2,2,0,0,0,2,0,2,0,0,0},
    {0,0,0,2,0,0,2,2,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,2,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,2,0,0},
    {0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0},
    {0,0,0,2,0,0,0,2,0,0,0,3,0,0,0,0},
    {0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0},
    {0,0,0,0,1,0,0,1,1,1,1,1,0,0,0,0},
    {0,0,0,0,0,1,1,1,1,0,1,1,0,0,0,0},
    {0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0},
    {0,2,0,0,0,0,1,0,1,0,0,0,0,0,0,0},
    {0,0,2,0,0,0,1,1,1,0,2,0,0,2,0,0},
    {0,0,0,0,0,0,0,1,1,0,0,0,2,2,0,0},
    {0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0},
    {0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0},
    {0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,1,1,1,0,1,1,0,0},
    {0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}
};

localparam [0:7][11:0] palette = {
    12'h000,
    12'hccc,
    12'hbbb,
    12'h999,
    12'h777,
    12'h555,
    12'h333,
    12'h642
};

logic [3:0] q;

always_comb begin
    unique case (texture_sel)
            2'd0: q <= mob0[pxl_y][pxl_x];
            2'd1: q <= mob1[pxl_y][pxl_x];
            2'd2: q <= mob2[pxl_y][pxl_x];
            2'd3: q <= mob3[pxl_y][pxl_x];
    endcase
end
```

```
always_ff @ (posedge Clk) begin
    if (q == 3'b0)
        a <= 1'b0;
    else
        a <= 1'b1;

    if (texture_sel == 2'd2) begin
        r <= palette[q][11:8] + 4'd3;
        g <= palette[q][7:4] - 4'd2;
        b <= palette[q][3:0] - 4'd2;
    end
    else begin
        r <= palette[q][11:8];
        g <= palette[q][7:4];
        b <= palette[q][3:0];
    end
end

endmodule
```

8.4.2 Mob Behavior

```
module mob_behavior (
    input Clk, alive,
    input [255:0][4:0] map,
    input [15:0] steve_x, steve_y,
    input [7:0] mob_seen, mob_aimed,
    input shoot, respawn,
    output logic [255:0][3:0] mob_map,
    output logic [7:0] who_shoot,
    output logic [6:0] score = 7'h0
);

int counter = 0;
int mob_cnt, j;
logic [7:0] i, loc;

always_ff @ (posedge Clk) begin
    who_shoot <= 8'h0;
    if (respawn) begin
        score <= 7'h0;
        mob_cnt <= 0;
        for(j=0; j<256; j=j+1)
            mob_map[j] <= 4'b0;
    end
    else if (alive) begin
        if (counter == 30000000) begin
            i <= i + 1;
            unique case (mob_map[i])
                4'b1001: mob_map[i] <= 4'b1010;
                4'b1010: mob_map[i] <= 4'b1011;
                4'b1011: begin
                    mob_map[i] <= 4'b1000;
                    who_shoot <= i;
                end
                4'b1100: mob_map[i] <= 4'b1101;
                4'b1101: mob_map[i] <= 4'b1110;
                4'b1110: mob_map[i] <= 4'b1111;
                4'b1111: mob_map[i] <= 4'b0000;
                default: ;
            endcase
            if (i == 8'd255)
                counter <= 0;
        end
        else begin
            counter <= counter + 1;
            if (shoot && mob_map[mob_aimed][3:2] == 2'b10) begin
                mob_map[mob_aimed] <= 4'b1100;
                mob_cnt <= mob_cnt - 1;
                score <= (score < 7'd99)? score + 1: 7'd99;
            end
            else if (mob_map[mob_seen] == 4'b1000)
                mob_map[mob_seen] <= 4'b1001;
            else begin
                loc <= loc + 8'd73;
                if ((~map[loc][4] & ~mob_map[loc][3])
                    && mob_cnt < 7
                    && (steve_x[15:8]-loc[3:0])
                    *(steve_x[15:8]-loc[3:0]-8'h1)
                    + (steve_y[15:8]-loc[7:4])
                    *(steve_y[15:8]-loc[7:4]-8'h1) > 20) begin
                    mob_map[loc] <= 4'b1000;
                    mob_cnt <= mob_cnt + 1;
                end
            end
        end
    end
end
endmodule
```

8.5 Appendix E

```
module steve (
    input Clk, Reset,
    input [7:0] keycode,
    input [255:0][4:0] map,
    input [255:0][3:0] mob,
    input [31:0] tri_center,
    input [7:0] who_mem,
    input hit,
    output [15:0] steve_x,
    output [15:0] steve_y,
    output logic [15:0] angle = 16'h0,
    output logic [3:0] bow_state,
    output logic shoot,
    output logic [3:0] health = 4'd10,
    output logic alive = 1'b1,
    output logic respawn
);

// keycode = {x, space, ->, <-, D, S, A, W}

logic frame_clk;
logic blk_u, blk_d, blk_r, blk_l;
int x = 32'h18000;
int y = 32'h18000;
int tmp_x, tmp_y;
int next_x, next_y;
int sin, cos, op, op_mem;

assign sin = {{16{tri_center[31]}},tri_center[31:16]};
assign cos = {{16{tri_center[15]}},tri_center[15:0]};
assign op = cos * (y - {12'h0, who_mem[7:4], 16'h8000})
           + sin * (x - {12'h0, who_mem[3:0], 16'h8000});

assign steve_x = x[23:8];
assign steve_y = y[23:8];

assign blk_u = mob[{y[19:16]-4'b1, x[19:16]}][3]
             | map[{y[19:16]-4'b1, x[19:16]}][4];
assign blk_d = mob[{y[19:16]+4'b1, x[19:16]}][3]
             | map[{y[19:16]+4'b1, x[19:16]}][4];
assign blk_r = mob[{y[19:16], x[19:16]+4'b1}][3]
             | map[{y[19:16], x[19:16]+4'b1}][4];
assign blk_l = mob[{y[19:16], x[19:16]-4'b1}][3]
```

```
            | map[{y[19:16], x[19:16]-4'b1}][4];

clk_divider clk0 (Clk, 32'd200000, frame_clk);  //125Hz

always_ff @ (posedge frame_clk) begin
    if (respawn | Reset)
        angle <= 16'h0;
    else if (alive) begin
        if (keycode[6]) begin
            if (keycode[5] & ~keycode[4]) begin
                if (angle < 16'd1)
                    angle <= angle + 16'd2399;
                else
                    angle <= angle - 16'd1;
            end
            else if (keycode[4] & ~keycode[5]) begin
                if (angle > 16'd2399)
                    angle <= angle - 16'd2399;
                else
                    angle <= angle + 16'd1;
            end
        end
        else begin
            if (keycode[5] & ~keycode[4]) begin
                if (angle < 16'd4)
                    angle <= angle + 16'd2396;
                else
                    angle <= angle - 16'd4;
            end
            else if (keycode[4] & ~keycode[5]) begin
                if (angle > 16'd2396)
                    angle <= angle - 16'd2396;
                else
                    angle <= angle + 16'd4;
            end
        end
    end
    else begin
        if (op_mem < 0 && op < 0) begin
            if (angle < 16'd4)
                angle <= angle + 16'd2396;
            else
                angle <= angle - 16'd4;
        end
        else if (op_mem > 0 && op > 0) begin
```

```
            if (angle > 16'd2396)
                angle <= angle - 16'd2396;
            else
                angle <= angle + 16'd4;
        end
    end
end

always_comb begin
    if (keycode[0] & ~keycode[2] & ~hit_back) begin
        if (keycode[6]) begin
            tmp_x = x + cos;
            tmp_y = y - sin;
        end
        else begin
            tmp_x = x + (cos << 2);
            tmp_y = y - (sin << 2);
        end
    end
    else if (keycode[2] & ~keycode[0] | hit_back) begin
        if (keycode[6]) begin
            tmp_x = x - cos;
            tmp_y = y + sin;
        end
        else begin
            tmp_x = x - (cos << 2);
            tmp_y = y + (sin << 2);
        end
    end
    else begin
        tmp_x = x;
        tmp_y = y;
    end
end

always_comb begin
    if (keycode[1] & ~keycode[3]) begin
        if (keycode[6]) begin
            next_x = tmp_x - sin;
            next_y = tmp_y - cos;
        end
        else begin
            next_x = tmp_x - (sin << 2);
            next_y = tmp_y - (cos << 2);
        end
    end
    else if (keycode[3] & ~keycode[1]) begin
        if (keycode[6]) begin
            next_x = tmp_x + sin;
            next_y = tmp_y + cos;
        end
        else begin
            next_x = tmp_x + (sin << 2);
            next_y = tmp_y + (cos << 2);
        end
    end
    else begin
        next_x = tmp_x;
        next_y = tmp_y;
    end
end

always_ff @ (posedge frame_clk) begin
    if (respawn | Reset) begin
        x <= 32'h18000;
        y <= 32'h18000;
    end
    else if (alive) begin
        if (blk_l && next_x < {x[31:16], 16'h4000})
            x <= {x[31:16], 16'h4000};
        else if (blk_r && next_x > {x[31:16], 16'hc000})
            x <= {x[31:16], 16'hc000};
        else
            x <= next_x;

        if (blk_u && next_y < {y[31:16], 16'h4000})
            y <= {y[31:16], 16'h4000};
        else if (blk_d && next_y > {y[31:16], 16'hc000})
            y <= {y[31:16], 16'hc000};
        else
            y <= next_y;
    end
    else begin
        if (((x >> 8) - {20'h0, who_mem[3:0], 8'h80})
            * ((x >> 8) - {20'h0, who_mem[3:0], 8'h80})
            + ((y >> 8) - {20'h0, who_mem[7:4], 8'h80})
            * ((y >> 8) - {20'h0, who_mem[7:4], 8'h80})
            > 32'h10000) begin
            x <= (x * 31 >> 5) + {17'h0, who_mem[3:0], 11'h400};
            y <= (y * 31 >> 5) + {17'h0, who_mem[7:4], 11'h400};
        end
    end
end
// Bow State Machine

int bow_counter = 0;

always_ff @ (posedge frame_clk) begin
    shoot <= 1'b0;
    if (respawn | Reset) begin
        bow_state <= 4'b0000;
        bow_counter <= 0;
    end
    else if (keycode[6] && bow_counter < 25) begin
        bow_state <= 4'b0001;
        bow_counter <= bow_counter + 1;
    end
    else if (keycode[6] && bow_counter < 50) begin
        bow_state <= 4'b0010;
        bow_counter <= bow_counter + 1;
    end
    else if (keycode[6] && bow_counter < 75) begin
        bow_state <= 4'b0011;
        bow_counter <= bow_counter + 1;
    end
    else if (keycode[6] && bow_counter < 100) begin
        bow_state <= 4'b0100;
        bow_counter <= bow_counter + 1;
    end
    else if (bow_counter == 100) begin
        bow_state <= 4'b0101;
        if (~keycode[6]) begin
            bow_counter <= bow_counter + 1;
            shoot <= 1'b1;
        end
    end
    else if (bow_counter > 100 && bow_counter < 107) begin
        bow_state <= 4'b0110;
        bow_counter <= bow_counter + 1;
    end
    else if (bow_counter >= 107  && bow_counter < 117) begin
        bow_state <= 4'b0111;
        bow_counter <= bow_counter + 1;
    end
    else begin
```

```
        bow_state <= 4'b0000;
        bow_counter <= 0;
    end
end
// health
int hp_counter = 0;
logic hit_state;
logic [3:0] hp_mem;
logic hit_back;
always_ff @ (posedge Clk) begin
    if (respawn | Reset) begin
        health <= 4'd10;
        alive <= 1'b1;
        hit_back <= 1'b0;
    end
    else if (hit && hp_counter == 0) begin
        hp_counter <= 1;
        hit_back <= 1'b1;
        hp_mem <= health;
        op_mem <= cos * (y - {12'h0, who_mem[7:4], 16'h8000}) + sin * (x - {1
    end
    if (hp_counter == 30000000) begin
        health <= hp_mem - 1;
        hp_counter <= 0;
        hit_back <= 1'b0;
    end
    else if (hp_counter > 0 && hp_counter < 30000000) begin
        hit_back <= 1'b1;
        hp_counter <= hp_counter + 1;

        if (hp_counter[23])
            health <= hp_mem - 1;
        else
            health <= hp_mem;
    end
    if (health == 4'd0)
        alive <= 1'b0;
end
always_ff @ (posedge frame_clk) begin
    respawn <= 1'b0;
    if (~alive & keycode[7])
        respawn <= 1'b1;
end
endmodule
```

8.6 Appendix F

8.6.1 Bow Color

```
`define STATE_NUM 6

module bow_color (
    input Clk,
    input [9:0] vga_x, vga_y,
    input [3:0] bow_state,
    output [3:0] bow_r, bow_g, bow_b,
    output bow_a
);

logic [3:0] ri [`STATE_NUM];
logic [3:0] gi [`STATE_NUM];
logic [3:0] bi [`STATE_NUM];
logic ai [`STATE_NUM];


bow_0_example bow0 (Clk, vga_x, vga_y, 1'b1, ri[0], gi[0], bi[0], ai[0]);
bow_1_example bow1 (Clk, vga_x, vga_y, 1'b1, ri[1], gi[1], bi[1], ai[1]);
bow_2_example bow2 (Clk, vga_x, vga_y, 1'b1, ri[2], gi[2], bi[2], ai[2]);
bow_3_example bow3 (Clk, vga_x, vga_y, 1'b1, ri[3], gi[3], bi[3], ai[3]);
bow_4_example bow4 (Clk, vga_x, vga_y, 1'b1, ri[4], gi[4], bi[4], ai[4]);
bow_5_example bow5 (Clk, vga_x, vga_y, 1'b1, ri[5], gi[5], bi[5], ai[5]);

localparam [0:6][0:6][1:0] cross_color = {
    {2'd1,2'd0,2'd0,2'd0,2'd0,2'd0,2'd1},
    {2'd0,2'd1,2'd2,2'd0,2'd2,2'd1,2'd0},
    {2'd0,2'd2,2'd1,2'd2,2'd1,2'd2,2'd0},
    {2'd0,2'd0,2'd2,2'd1,2'd2,2'd0,2'd0},
    {2'd0,2'd2,2'd1,2'd2,2'd1,2'd2,2'd0},
    {2'd0,2'd1,2'd2,2'd0,2'd2,2'd1,2'd0},
    {2'd1,2'd0,2'd0,2'd0,2'd0,2'd0,2'd1},
};

localparam [0:2][11:0] cross_palette = {
    {4'h0, 4'h0, 4'h0},
    {4'hc, 4'ha, 4'h6},
    {4'ha, 4'h8, 4'h6},
};

logic[1:0] cross_0, cross_1;
```

```
always_comb begin
    if (vga_x >= 215 && vga_y >= 109 && vga_x < 439 && vga_y < 333)
        cross_0 = cross_color[(vga_x-215)>>5][(vga_y-109)>>5];
    else
        cross_0 = 2'd0;

    if (vga_x >= 309 && vga_y >= 226 && vga_x < 337 && vga_y < 254)
        cross_1 = cross_color[(vga_x-309)>>2][(vga_y-226)>>2];
    else
        cross_1 = 2'd0;
end

always_comb begin
    if( bow_state == 4'b0110 && cross_0 > 0) begin
        { bow_r, bow_g, bow_b, bow_a } = { cross_palette[cross_0] , 1'b1};
    end else if ( bow_state == 4'b0111 && cross_1 > 0) begin
        { bow_r, bow_g, bow_b, bow_a } = { cross_palette[cross_1] , 1'b1};
    end else if ( bow_state >= 4'b0110 ) begin
        bow_r = ri[0];
        bow_g = gi[0];
        bow_b = bi[0];
        bow_a = ai[0];
    end else begin
        bow_r = ri[bow_state];
        bow_g = gi[bow_state];
        bow_b = bi[bow_state];
        bow_a = ai[bow_state];
    end
end

endmodule
```

8.6.2 Arrow Animation

```
module arrow_animation(
    input Clk,
    input logic [7:0] who_shoot,
    input [15:0] steve_x,
    input [15:0] steve_y,
    output logic arrow, hit,
    output logic [7:0] who_mem
);

int frame_counter = 0;
```

```
logic [7:0] aim_x, aim_y;

always_ff @(posedge Clk) begin
    if(who_shoot > 8'h0 && frame_counter == 0) begin
        aim_x <= steve_x[15:8];
        aim_y <= steve_y[15:8];
        frame_counter <= 1;
        who_mem <= who_shoot;
    end

    else if (frame_counter > 0 && frame_counter < 15000000)
        frame_counter <= frame_counter + 1;

    else if (frame_counter == 15000000) begin
        if(steve_x[15:8] == aim_x && steve_y[15:8] == aim_y) begin
            hit <= 1'b1;
            arrow <= 1'b1;
        end
        frame_counter <= frame_counter + 1;
    end

    else if (frame_counter > 15000000 && frame_counter < 25000000)
        frame_counter <= frame_counter + 1;

    else if (frame_counter >= 25000000 && frame_counter < 27000000)
    begin
        frame_counter <= frame_counter + 1;
        hit <= 1'b0;
    end

    else if (frame_counter == 27000000) begin
        arrow <= 1'b0;
        frame_counter <= 0;
    end
end

endmodule
```