DUCK HUNT '22

Final Report


Kristen Shaker (kls2243)


Alex Yao (awy2108)


Bryce Natter (bdn2113)

Introduction

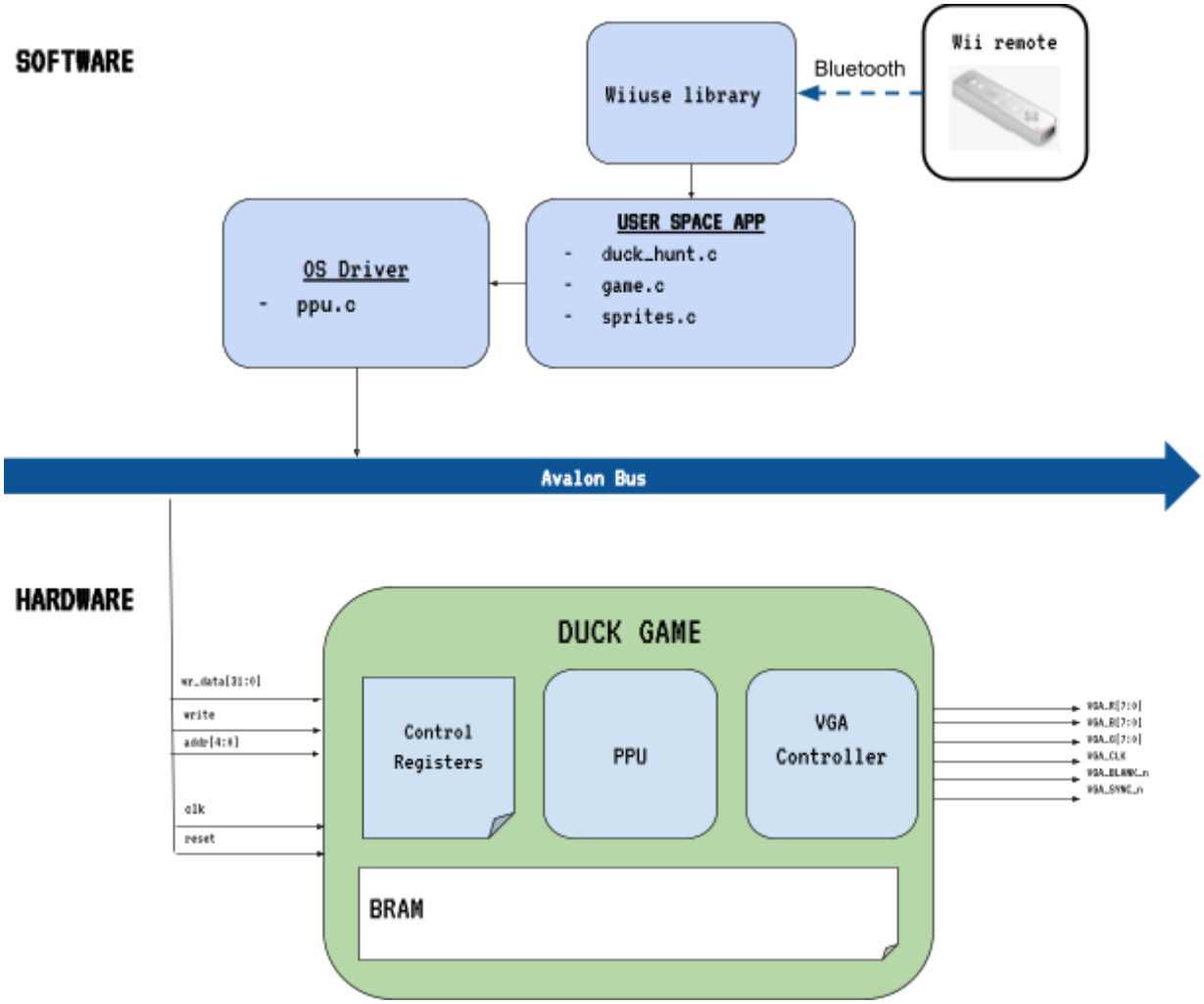
We recreate the classic NES game “Duck Hunt” on the DE1-SoC board using a Wii remote. The premise of the game is that ducks are introduced from the bottom of the screen and fly in a randomized pattern. The player has three shots to hit the ducks. After shooting all three bullets, the state resets and new ducks are introduced. We use a Wii remote to control the position of a crosshair on the screen displayed by the VGA. When the player clicks the trigger (button B), a shot is registered and the visuals are updated. There are a total of 8 rounds in a single game instance, with each round increasing in difficulty. Ducks in later rounds are worth more points, but also move faster.

We use tile and sprite graphics to display our visuals, and the game logic and input are handled by software. The software communicates coordinates and game state to the hardware for display on a screen, as well as manages inputs from a Bluetooth connection to a Wii remote.

In this report, we provide the design details of our implementation as well as a code appendix for non-generated files in both hardware and software. For the complete project code, see our zip archive. Note that we require the FPGA Linux kernel to be compiled with Bluetooth support to connect to the Wii controller.

System Architecture

In this section, we present an overview of the system architecture used for our project. On the software side, we have a user space program which interacts with the Wiiuse library to communicate with a Wii controller over Bluetooth. This application serves as the main control point for the game, managing all game state and sprite coordination. The application communicates through our kernel module, ppu.ko, with the FPGA hardware. The hardware implements an NES inspired sprite controller design. All sprites are loaded at program startup from the software, allowing our underlying hardware to be used for any custom game application.



Hardware: Picture Processing Unit (PPU)

We implemented a PPU on the FPGA fabric to display our sprites. For the design of the PPU, we took inspiration from the original NES's PPU and adapted it to output at a resolution of 640x480.

Hardware Interface

Our design included a 64 entry sprite-attribute table, a 1200 entry pattern table, a 16 entry color table, and a 256 entry sprite table. These tables offered similar functionality to those in the NES's PPU.

Internally, these tables were implemented using the FPGA's BRAM, stored 32bit words, and addressable using a 16 bit address space. Actual table sizes are as follows: 64 32bit words in the attribute table, 2056 32bit words in the pattern table, 64 32bit entries in the color table, and 4096 32bit words in the sprite table. Externally, for writing data, the tables are word addressable. Externally reading from the tables is not possible - reading from the tables only occurs internally, starting at table entry-size offsets. Table entries for each table are listed below.

Valid Address Ranges	Table
0x0000 -> 0x03F	Attribute Table
0x1000 -> 0x102F	Color Table
0x2000 -> 0x24BA	Pattern Table
0x3000 -> 0x3FFF	Sprite Table

Attribute:

31 Color ID 28	27 Sprite ID 20	19 X Location 10	9 Y Location 0

Pattern:

31 Unused 12	11 Color ID 8	7 Sprite ID 0

Sprite:

(ID) n	31 Pixel 30	29 2bit Pixels	0
n+1	31 Pixel	29 2bit Pixels	0
	31 Pixel 30	29 2bit Pixels	0
n+15	31 Pixel 30	29 2bit Pixels	0

Color Palette:

(ID) n	31	Unused	24	23	RED	16	15	GREEN	8	7	BLUE	0
n+1	31	Unused	24	23	RED	16	15	GREEN	8	7	BLUE	0
n+2	31	Unused	24	23	RED	16	15	GREEN	8	7	BLUE	0
n+3	31	Unused	24	23	RED	16	15	GREEN	8	7	BLUE	0

VGA Picture Output

A state machine is used internally to access tables, load pixel down counters and shifters, and display pixels. The state machine consists of two main stages: outputting pixels and loading the pixel-gen registers. Loading the pixel-gen registers occurs between hcount 1280 and 1599, when the VGA signal is resetting for the next horizontal line. Outputting occurs during hcount 0 and 1279, when the module is outputting the color signals to the display.

Loading Stage:
The loading stage is broken down into 5 states: A_INDEX, CHECK, S_INDEX, SET, and IDLE. A_INDEX state refers to having the index into the attribute table set, meaning its data will be available during the next cycle while in state CHECK. CHECK state is used to check the attribute's parameters, if the attribute's y_value is within vcount to vcount + 15, the index into the sprite table is set next cycle and the state is changed to S_INDEX, otherwise set next attribute index and set the state to A_INDEX. S_INDEX state is similar to A_INDEX, but with the additional step of setting the load triggers for the shift and down counters, so that the sprite output can be loaded in during the next cycle when its outputted from the sprite table. The IDLE state is used to hold the state to prepare sprite generation and pattern generation during the output stage.

Output Stage:
The output stage is broken down into two parts, one to handle sprite shifters output and another to handle pattern shifter loading and outputting. Outputting a pixel takes 4 cycles: reading shifter outputs and setting the color table index, waiting for color table output, reading color table output and setting output registers, and finally displaying the pixel. Although this takes 4 cycles, it is pipelined such that a pixel is output every cycle. For the patterns, their sprites are read and loaded into shifters every 32 cycles. A ping pong design is used so that while one shifter is outputting the sprite, the other shifter is loaded from pattern and sprite tables.

Software

In this section, we describe the various software components in our system including the game logic, Wii controller interaction, and auxiliary scripts used to create sprites.

Game Logic

The core game logic and game state are updated through control algorithms on the software side. The main states we keep track of during gameplay are:

- Game state (score, round, bullets remaining, ducks hit)
- Duck states (position, velocity vector, sprite option)
- Crosshair position

Game state - The main game loop is responsible for tracking the gameplay in one instance. On game over, the loop will reset all states for a new playthrough. The game state we track during gameplay are the scores (increasing by duck values after shot), bullets remaining (from 3), and round (increasing after each set of ducks).

Specifically, the game state object is:

```
typedef struct {
	unsigned char bullets, score, round;
	int spawned_ducks;
	int visible_ducks;
} game_config_t;
```

These parameters control the displayed game state (bullets, score, round) as well as hidden state to determine rounds ending and resetting ducks on screen.

Duck states - Each duck state encapsulates the information that should be communicated to the hardware during each frame (position, sprite option) as well as some meta information about the velocity of the duck. This provides sufficient information to determine duck position at the next frame. E.g. If the duck has been shot, decrease Y position and display “dead” sprite.

Specifically, the duck state object is:

```
typedef struct {
	// How many points the duck is worth in the game.
	int value;
	// Where the duck currently is on the screen.
	coord_t coord;
	// Direction on x plane.
	enum direction x_direction;
	// Direction on y plane.
```

```
        enum direction y_direction;
        // Angle in degrees that ducks will move at an angle on the y plane to make
game_play more interesting.
        // an angle of 45 degrees denotes 1 unit of movement in the y plane for every
one unit of movement in the x plane.
        double angle;
        // Velocity multiplier
        double velocity;
        // unique_id associated with each duck.
        int id;
        enum duck_state state;
        // when the duck was created.
        time_t spawn_time;
} duck_t;
```

Each time a duck is spawned, the value and velocity are generated based on the current round of the game. The angle is initially set to a random value between 0 and 45 degrees, and each time the duck hits an edge of a screen or at random time intervals, a new angle is generated. This simulates random pathing of the duck during gameplay.

Crosshair position - This information is polled from the Wii remote controller during each pass and checked for collision with ducks that are not dead or inactive. To avoid jittering, this information is polled every cycle in the inner game loop regardless of gameplay state.

Using these states, we have a continuous game loop which covers game play.

1. Sync Wii controller and enable IR motion detection
2. Press A to begin game
3. Deploy 2 ducks with generated parameters
4. User has 5 seconds to shoot ducks before they fly away
5. Increase round, loop back to 3
6. After 8 rounds, end game and loop back to 2

Wii Controller

The main peripheral for this project is a Wii controller which is used to determine the X and Y coordinates of the crosshair and take button input to start the game and shoot. We use an existing userspace library, Wiiuse, to interpret the Wii controller inputs.

Citation: https://github.com/wiiuse/wiiuse

Wiiuse is a C library which connects to multiple Wii remotes using the Bluetooth protocol. The main functionality Wiiuse provides is a set of abstractions for polling the Wii controller for data and interpreting the output by populating a struct and matching with a set of macros.

There are a few key obstacles which had to be overcome to incorporate this library into the FPGA software:

1. Cross compilation of kernel to support Bluetooth. The FPGA 4.19 kernel as given does not support any external modules. Bluetooth is a fairly standard protocol which is supported in a kernel module included in many linux kernels. We recompile the FPGA kernel using a new configuration file which includes external modules and specifically enables Bluetooth. The corresponding zImage was loaded onto the FPGA.
2. External USB Bluetooth receiver. The FPGA does not have built-in Bluetooth, so we purchased an external receiver and connected it to a USB slot on the FPGA.
3. Installing Bluetooth libraries. There are a number of existing packages for using Bluetooth. The necessary ones which we installed were bluetooth and blueman.

After setup, we use the Wiiuse library to iterate over all available Bluetooth devices and connect only to the Wii controllers. Then, we send over some configuration data such as enabling IR detection. Each game tick, we poll the Wii controller for pointer location data and button presses. We manually scale the returned pointer location based on multiplicative constants determined by visual inspection.

Sprite Generation

One of the challenging problems with custom hardware is producing sprites that are visually appealing and true to the original game. To approach this problem, we create a custom script which converts image files into our required format using Python and Jinja2. We divide the images into 16x16 pixel chunks. Since our color maps only index 3 colors, we determine the most relevant colors of each block, then render both the color maps and sprite data.

The script incorporates the following steps:

1. Convert image file into an array of RGBA channel data
2. Determine the most frequent 3 RGB values
3. Convert all lower frequency colors to the most frequent RGB color
4. Divide the image into 16x16 tiles
5. Render outputs using templates to generate color maps and individual tile sprite information

The rendering stage loads both a generic color map template and sprite output template.

The color template format the PPU accepts follows this layout:

```
{
        .id = {{id}},
        .color = {
                [0] = {.r = 0, .g = 0, .b = 0},
{% for color in colors %}
                 [{{loop.index}}] = {.r = {{color[0]}}, .g = {{color[1]}}, .b = {{color[2]}} },
```

```
{% endfor %}
        },
};
```

Each color is written out to a continuous block in a struct. For sprite data, the PPU accepts sixteen bit integers, written out as:

```
{
        .id  = {{ id }},
        .line = {
{% for line in lines %}
                {{ line }},
{% endfor %}
        },
};
```

Each line is an individual integer row of data, while id is used to give each sprite a unique id which can be indexed.

Resource Budget

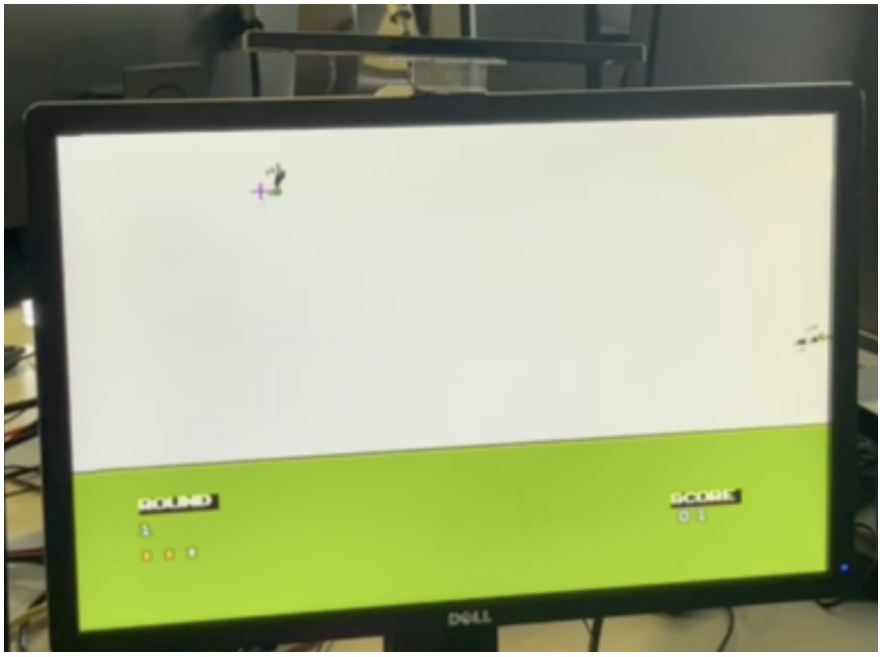
Category	Image	Size (pixels)	Variants	Total Bits
Duck		32 x 32	4 (up, down, dead, flying away)	4 * 32 * 32 * 2 = 8,192 bits
Bullet		16 x 16	1	1 * 16 * 16 * 2 = 512 bits
Background		16 x 16	8	8* 16 * 16 * 2 = 4096 bits
Score + Round (numbers)		16 x 16	10	10 * 16 * 16 * 2 = 10240 bits
Crosshair		16 x 16	1	1 * 16 * 16 * 2 =
Color Pallet		4 x 32 (bits not pixels)	8	4 * 32 = 128 bits

Total Budget Required: 23,680 bits = 2,960 bytes = 2.89 KB

Game Screenshots

Screenshot 1: Initial starting game state, 0 score no ducks, press A to begin
Wii remote IR bar is located on top of monitor

Screenshot 2: Two ducks are deployed, flying in random directions at different speeds. The crosshair location is from the Wii controller input.



Screenshot 3: A quick muzzle flash occurs when the user pulls the trigger (presses B) along with controller rumble. The duck animation changes if it is shot and falls down, bullets decrease, and score increases.

Screenshot 4: As the game progresses, the ducks are worth more points and also fly faster. The color changes at certain rounds to indicate new duck types.

Work Distribution and Lessons

All members contributed equally to the final project. The general boundaries of work distribution can be summarized as:

- Bryce - PPU hardware, test bench, pattern tiling and sprites
- Kristen - Core game logic, unit tests, HW communication
- Alex - Wii controller, sprite generation, end-to-end integration

There are a few key lessons we learned throughout this process, both technical and non-technical, which we think would be valuable to future groups.

1. There's always more work to be done! Especially for creating a game, there's essentially an endless amount of features that could be integrated. The important part is prioritizing the minimum amount necessary to demonstrate the key components of the game. In our case, this was focusing on the shooting mechanics and making some ad-hoc workarounds in the hardware to generate a sky-like background.
2. Have multiple threads of work at any given time. Testing hardware can be a slow process, with compilation and rebooting eating up hours of time. We found the most efficient distribution was to have a list of tasks to accomplish and being able to switch between compiling and implementing quick features in other areas.
3. Go to TA meetings with very specific questions! The teaching staff are all managing many different projects and cannot give very specific feedback on certain issues. The best approach is to frame a minimum set of information and tried methods for a question, then ask for feedback on a set of proposed possible solutions. This helped a lot with connecting the Wii controller for example, where we understood the general issues but not how to specifically cross compile the right kernel version.

Code Appendix

HARDWARE

```
`include "down_counter.sv"
`include "shift.sv"
`include "memory.sv"
`include "hex7seg.sv"

/* verilator lint_off DECLFILENAME */
/* verilator lint_off WIDTH */
/* verilator lint_off UNUSED */
module ppu
       #(parameter
       VISIBLE_SPRITES = 8,
       SPRITE_ATTRS = 64
       )
       (input logic              clk,
       input logic         reset,
       input logic [31:0]        writedata,
       input logic         write,
       input               chipselect,
       input logic [15:0]        address,

       output logic [6:0]  HEX0, HEX1, HEX2, HEX3, HEX4, HEX5,
       output logic [7:0]  VGA_R, VGA_G, VGA_B,
       output logic        VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n,
       output logic        VGA_SYNC_n);


       logic [7:0]         background_r, background_g, background_b;
        // which table in memroy to write to (if any).
       // which horizontal pixel we are currently on.
       logic [10:0]        hcount;
       logic [9:0]         vcount;

       // down counter enable and load variables used for (sprite,color) -> pixel
color
       logic [VISIBLE_SPRITES-1:0]      dc_en, dc_ld, dc_done, dc_r;
       logic [VISIBLE_SPRITES-1:0]      sh_en, sh_ld, sh_r;
       logic [1:0]               sh_out [VISIBLE_SPRITES - 1: 0];
       logic [5:0]               color [VISIBLE_SPRITES - 1: 0];


       // attribute count variable to loop through attribute table entries.
       // vc variable to keep track of how many attributes are actually
```

```
	// visible on screen.
	logic [5:0]		ac, vc;
	logic [9:0]		tx; // ty;

	//Address and Data to write to a table
	logic [31:0]		w_data;
	logic [15:0]		w_addr;

	//Address used to index into each table
		logic [5:0]		a_addr;
	logic [10:0]		p_addr;
	logic [11:0]		s_addr;
		logic [5:0]		c_addr;

	//Address used to read a table entry
	 logic [5:0]			ar_addr;
	logic [10:0]		pr_addr;
	logic [11:0]		sr_addr;
	logic [5:0]		cr_addr;

	//Outputs of each table
	logic [31:0]		sprite_attr, sprite, pattern, color_out;


	logic [3:0] mem_write;
	assign a_addr = mem_write[0] ? w_addr[5:0]: ar_addr[5:0];
	assign c_addr = mem_write[1] ? w_addr[5:0]: cr_addr;
	assign p_addr = mem_write[2] ? w_addr[11:0]: (state == OUTPUT ? (
(vcount[9:4] * 40) + hcount[10:5]) : vcount[9:4] * 40);
	assign s_addr = mem_write[3] ? w_addr[11:0]: (state == OUTPUT ? (
(pattern[7:0] << 4) + vcount[3:0]): sr_addr);


	vga_counters		counters(.clk50(clk), .*);
	memory #(32,  64, 6)	attr_table  (.clk(clk),  .we(mem_write[0]),
.addr(a_addr[5:0]), .data_in(w_data), .data_out(sprite_attr));
	memory #(32,  64, 6)	color_table (.clk(clk),  .we(mem_write[1]),
.addr(c_addr[5:0]), .data_in(w_data), .data_out(color_out));
	memory #(32, 2048, 11)	pattern_table(.clk(clk), .we(mem_write[2]),
.addr(p_addr[10:0]), .data_in(w_data), .data_out(pattern));
	memory #(32, 4096, 12)	sprite_table(.clk(clk),  .we(mem_write[3]),
.addr(s_addr[11:0]), .data_in(w_data), .data_out(sprite));

	genvar k;
	generate
	for(k = 0; k <= VISIBLE_SPRITES - 1; k = k+1) begin : pixelgen
		down_counter dc(.clk(clk), .en(dc_en[k]), .ld(dc_ld[k]),
```

```
.reset(dc_r[k]), .data_in(tx), .done(dc_done[k]));
            shift sh(.clk(clk), .en(sh_en[k]), .ld(sh_ld[k]), .reset(sh_r[k]),
.data_in(sprite), .data_out(sh_out[k]));
      end
      endgenerate
      assign sh_en = dc_done;


      logic ping_pong;
      logic [1:0] pp_sh_en, pp_sh_ld, pp_sh_reset;
       logic [1:0] pp_sh_out [1:0];
       logic [31:0] pp_color [1:0];
      shift pong_sh(.clk(clk), .en(pp_sh_en[0]), .ld(pp_sh_ld[0]),
.reset(pp_sh_reset[0]), .data_in(sprite), .data_out(pp_sh_out[0]));
      shift ping_sh(.clk(clk), .en(pp_sh_en[1]), .ld(pp_sh_ld[1]),
.reset(pp_sh_reset[1]), .data_in(sprite), .data_out(pp_sh_out[1]));



      always_ff @(posedge clk) begin
            mem_write <= 3'b0;
            if (chipselect && write) begin
                  // The first two bits of the address tell us which
                  // table we are writing to.
                  // 0X0000 Sprite Attribute Table
                  // 0x1000 Color Table
                  // 0x2000 Pattern Table
                  // 0x3000+ Sprite Table
                  case(address[15:12])
                        4'b0000: mem_write[0]     <= 1'b1;
                        4'b0001: mem_write[1]     <= 1'b1;
                        4'b0010: mem_write[2]     <= 1'b1;
                        default: mem_write[3]     <= 1'b1;
                  endcase
                  w_addr <= address;
                  w_data <= writedata;
            end
      end

      enum logic [3:0] {A_INDEX, CHECK, S_INDEX, SET, IDLE, OUTPUT} state;
      enum logic [1:0] {PP_P_INDEX, PP_P_CHECK, PP_S_READ} pp_state;

      always_ff @(posedge clk) begin
            dc_en <= {VISIBLE_SPRITES{1'b0}}; dc_ld <={VISIBLE_SPRITES{1'b0}};
dc_r <= {VISIBLE_SPRITES{1'b0}};
            sh_ld <= {VISIBLE_SPRITES{1'b0}}; sh_r <= {VISIBLE_SPRITES{1'b0}};
            pp_sh_en <= 2'b0; pp_sh_ld <= 2'b0; pp_sh_reset <= 2'b0;
```

```
            case (state)
                  A_INDEX: begin //Attr table index set, data available next
cycle
                        state <= CHECK;
                  end
                  CHECK: begin //Attr table output ready, check y
                        if (ac == SPRITE_ATTRS - 1'b1 || vc == VISIBLE_SPRITES -
1'b1 || hcount == 11'd1598) state <= IDLE;
                        else if (vcount <= sprite_attr[9:0] + 15 && vcount >=
sprite_attr[9:0]) begin
                              tx    <= sprite_attr[19:10];
                              sr_addr     <= (sprite_attr[27:20] << 4) +
(vcount - sprite_attr[9:0]);
                              color[vc]   <= sprite_attr[31:28] << 2;
                              state       <= S_INDEX;

                        end else begin
                              ar_addr     <= {2'b0, ac + 4'b1};
                              ac    <= ac + 1'b1;
                              state  <= A_INDEX;
                        end

                  end
                  S_INDEX: begin //Sprite table index set, data available next
cycle
                        //Prep pixel_gen modules
                        dc_ld[vc]   <= 1'b1;
                        sh_ld[vc]   <= 1'b1;

                        //We can use SET state to skip A_INDEX state
                        ar_addr           <= {2'b0, ac + 4'b1};
                        ac          <= ac + 1'b1;
                        state       <= SET;
                  end
                  SET: begin //Sprite table output ready, just wait
                        if (ac == SPRITE_ATTRS - 1'b1 || hcount == 11'd1598)
state <= IDLE;
                        else state  <= CHECK;
                        vc    <= vc + 1'b1;
                  end
                  IDLE: begin
                        if (hcount == 11'd1597) sr_addr <= (pattern[7:0] << 4) +
vcount[3:0];
                        if (hcount == 11'd1598) pp_sh_ld[ping_pong] <= 1'b1;
                        if (hcount == 11'd1599) begin
```

```
                        state <= OUTPUT;
                        dc_en <= {VISIBLE_SPRITES{1'b1}};
                        pp_sh_en[ping_pong] <= 1'b1;

                    end
                end
                OUTPUT: begin

                    dc_en <= hcount[0] ? {VISIBLE_SPRITES{1'b0}} :
{VISIBLE_SPRITES{1'b1}};
                    if (hcount[4:0] == 5'b11111) ping_pong <= ping_pong ? 0 :
1;
                    pp_sh_en[ping_pong] <= hcount[0];

                    //Start loading !ping_pong
                    case (pp_state)
                        PP_P_INDEX: begin //Pattern table addr has been
set
                            pp_state <= PP_P_CHECK;
                        end
                        PP_P_CHECK: begin //Patterntable outputing sprite
index
                            pp_color[!ping_pong] <= (pattern[11:8] <<
2); //Save color;
                            pp_sh_ld[!ping_pong] <= 1; //Set !ping_pong
shifter to load next cycle
                            pp_state <= PP_S_READ; //Sprite table
should be outputting sprite next cycle
                        end
                        PP_S_READ: begin //Sprite table outputting pattern
sprite, just wait...
                            if(hcount[4:0] == 5'b11111) begin
                                pp_state <= PP_P_INDEX;
                                pp_sh_en[!ping_pong] <= 1'b1; //en
set high on 11110, outputs 11111
                            end
                        end
                        default:pp_state <= PP_P_INDEX;

                    endcase


                    cr_addr <= pp_color[ping_pong] + {3'b0,
pp_sh_out[ping_pong]};
                    if(sh_out[0] != 2'b0)        cr_addr <= color[0] +
{3'b0, sh_out[0]};
                    else if(sh_out[1] != 2'b0) cr_addr <= color[1] + {3'b0,
```

```
sh_out[1]};
                        else if(sh_out[2] != 2'b0) cr_addr <= color[2] + {3'b0,
sh_out[2]};
                        else if(sh_out[3] != 2'b0) cr_addr <= color[3] + {3'b0,
sh_out[3]};
                        else if(sh_out[4] != 2'b0) cr_addr <= color[4] + {3'b0,
sh_out[4]};
                        else if(sh_out[5] != 2'b0) cr_addr <= color[5] + {3'b0,
sh_out[5]};
                        else if(sh_out[6] != 2'b0) cr_addr <= color[6] + {3'b0,
sh_out[6]};
                        else if(sh_out[7] != 2'b0) cr_addr <= color[7] + {3'b0,
sh_out[7]};


                        background_r <= color_out[7:0];
                        background_g <= color_out[15:8];
                        background_b <= color_out[23:16];

                        if (hcount == 11'd1279) begin
                              dc_r   <= {VISIBLE_SPRITES{1'b1}};
                              sh_r   <= {VISIBLE_SPRITES{1'b1}};
                              state  <= A_INDEX;
                              ar_addr      <= 8'b0;
                              ac     <= 4'b0;
                              vc     <= 4'b0;

                              //reset ping_pong;
                              ping_pong <= 1'b0;
                              pp_sh_reset  <= 2'b11;
                        end
                  end
                  default: state <= CHECK;

            endcase //case
      end //always_ff



      //Write background color to VGA
      always_comb begin
            if (VGA_BLANK_n )
                  {VGA_R, VGA_G, VGA_B} = {background_r, background_g,
background_b};
            else
                  {VGA_R, VGA_G, VGA_B} = {8'h0, 8'h0, 8'h0};
      end
```

```
	//Hex Display debug
	logic [15:0]		haddr;
	always_ff @(posedge clk) begin
		if (address) haddr <= address;
	end

	hex7seg h0(haddr[3:0],	HEX0);
	hex7seg h1(haddr[7:4],	HEX1);

	hex7seg h2(haddr[11:8],	HEX2);
	hex7seg h3(haddr[15:12],HEX3);

	hex7seg h4({3'b0, ping_pong},	HEX4);
	hex7seg h5(state[3:0],	HEX5);

endmodule


module shift
	(input logic		clk,
	 input logic		reset,
	 input logic		en,
	 input logic		ld,
	 input logic [31:0]		data_in,
	 output logic [1:0]		data_out);

	 // 16 memory locations each 2 bits in size.
	 logic [1:0] mem[15:0];

	always_ff @(posedge clk) begin
		if (reset) begin
			mem[0]		<= 2'b0;
			mem[1]		<= 2'b0;
			mem[2]		<= 2'b0;
			mem[3]		<= 2'b0;
			mem[4]		<= 2'b0;
			mem[5]		<= 2'b0;
			mem[6]		<= 2'b0;
			mem[7]		<= 2'b0;
			mem[8]		<= 2'b0;
			mem[9]		<= 2'b0;
			mem[10]	<= 2'b0;
			mem[11]	<= 2'b0;
			mem[12]	<= 2'b0;
			mem[13]	<= 2'b0;
			mem[14]	<= 2'b0;
			mem[15]	<= 2'b0;
```

```
                data_out <= 2'b0;
            end
            else if (en) begin
                mem[0]       <= 2'b0;
                mem[1]       <= mem[0];
                mem[2]       <= mem[1];
                mem[3]       <= mem[2];
                mem[4]       <= mem[3];
                mem[5]       <= mem[4];
                mem[6]       <= mem[5];
                mem[7]       <= mem[6];
                mem[8]       <= mem[7];
                mem[9]       <= mem[8];
                mem[10] <= mem[9];
                mem[11] <= mem[10];
                mem[12] <= mem[11];
                mem[13] <= mem[12];
                mem[14] <= mem[13];
                mem[15] <= mem[14];
                data_out <= mem[15];
            end
            else if (ld) begin
                mem[0]       <= data_in[31:30];
                mem[1]       <= data_in[29:28];
                mem[2]       <= data_in[27:26];
                mem[3]       <= data_in[25:24];
                mem[4]       <= data_in[23:22];
                mem[5]       <= data_in[21:20];
                mem[6]       <= data_in[19:18];
                mem[7]       <= data_in[17:16];
                mem[8]       <= data_in[15:14];
                mem[9]       <= data_in[13:12];
                mem[10] <= data_in[11:10];
                mem[11] <= data_in[9:8];
                mem[12] <= data_in[7:6];
                mem[13] <= data_in[5:4];
                mem[14] <= data_in[3:2];
                mem[15] <= data_in[1:0];
            end
            else data_out <= 2'b0;
        end
        //assign data_out = mem[15];
    endmodule

module memory
        #(parameter
        WORD_SIZE = 32,                                 // Size of RAM words
```

```
	NUM_WORDS = 16,                                 // Number of words to store in
RAM
	ADDR_BITS = 4)                                  // Number of RAM address bits
	(input logic                clk,    // Clock
	 input logic                we,           // Write enable
	 input logic [ADDR_BITS -1 :0]   addr,         // Address to read/write
	 input logic [WORD_SIZE-1:0]     data_in,      // Data to write
	 output logic [WORD_SIZE-1:0]    data_out);    // Data to read


	// Addresses are 4 bit addressible ( 0 - 15 ) and each address has 32
	// bits of space.
	logic [WORD_SIZE - 1:0]  mem[NUM_WORDS];

	always_ff @(posedge clk) begin
		if (we) mem[addr] <= data_in;
		data_out <= mem[addr];
	end

endmodule

module down_counter
	#(parameter
	N = 10)
	(input logic        clk,
	 input logic        reset,
	 input logic        en,
	 input logic        ld,
	 input logic [N-1:0]      data_in,
	 output logic             done);


	// How many clock cycles we should wait.
	logic [N-1:0] count;

	always_ff @(posedge clk) begin
		if (reset) begin
			count <= 0;
			done  <= 0;
		end
		else if (ld) begin
			count <= data_in;
			done  <= 0;
		end
		else if (en) begin
			if (count == 0) done <= 1;
			else count <= count - 1'b1;
```

```
            end
            else done <= 0;
        end
endmodule
```

```
#include <stdint.h>
#include <iostream>
#include "Vppu.h"
#include <verilated.h>
#include <verilated_vcd_c.h>
#include "../../sw/sprites.h"
#include <bitset>

#define OBJ_SPRITE_OFFSET 20
#define OBJ_X_COORD_OFFSET 10
#define OBJ_Y_COORD_OFFSET 0
#define OBJ_COLOR_OFFSET 28

#define ATTR_TABLE_MEMORY_OFFSET 0x0000
#define COLOR_TABLE_MEMORY_OFFSET 0x1000
#define PATTERN_TABLE_MEMORY_OFFSET 0x2000
#define SPRITE_TABLE_MEMORY_OFFSET 0x3000


int32_t attr_to_int(attr_table_entry_t *attr)
{
	int data = 0x00000000;
	data = data | (attr->coord.x      << OBJ_X_COORD_OFFSET);
	data = data | (attr->coord.y      << OBJ_Y_COORD_OFFSET);
	data = data | (attr->sprite       << OBJ_SPRITE_OFFSET );
	data = data | (attr->color_table << OBJ_COLOR_OFFSET  );
	return data;
}

int32_t pattern_to_int( pattern_table_entry_t *pat)
{
	int data = 0x00000000;
	data = data | (pat->sprite << PAT_SPRITE_OFFSET);
	data = data | (pat->color_table << PAT_COLOR_OFFSET);
	return data;
}

int32_t color_to_int( color_t *color)
{
	int data = 0x00000000;
	data = data | (color->r << RED_OFFSET);
```

```
	data = data | (color->b << BLUE_OFFSET);
	data = data | (color->g << GREEN_OFFSET);
	return data;
}

int main(int argc, const char ** argv, const char ** env) {

	Verilated::commandArgs(argc, argv);

	Vppu * dut = new Vppu;
	Verilated::traceEverOn(true);
	VerilatedVcdC * tfp = new VerilatedVcdC;
	dut->trace(tfp, 99);
	tfp->open("ppu.vcd");

	//PPU Inputs:
	dut->clk     = 0;
	dut->reset   = 0;
	dut->writedata      = 0;
	dut->write   = 0;
	dut->chipselect = 0;
	dut->address = 0;

	//To write:
	dut->write   = 1;
	dut->chipselect = 1;

	//Attr Table Entry
	attr_table_entry_t attr = {
			.coord = {
				.x = 50,
				.y = 20,
			},
			.sprite         = 0x0,
			.color_table    = 0x0,
			.id             = 0x0,
		};

	//Sprite Table Entry
	sprite_table_entry_t sprite = {
		.id  = 0x0,
		.line = {
			0x55555555, 0x55555555, 0x55555555, 0x55555555,
			0xAAAAAAAA, 0xAAAAAAAA, 0xAAAAAAAA, 0xAAAAAAAA,
			0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
			0x55555555, 0x55555555, 0x55555555, 0x55555555,
		},
```

```
};

//Color Table Entry:
color_table_entry_t color_palette = {
	.id = 0x0,
	.color = {
		[0] = {.r = 0,   .g = 0,   .b = 0  },
		[1] = {.r = 0,   .g = 100, .b = 100},
		[2] = {.r = 100, .g = 100, .b = 0  },
		[3] = {.r = 100, .g = 0,   .b = 100},
	},
};

int time = 0;
int clock = 0;

for (int i = 0 ; i < 10; i++)
{
	dut->clk = 1;
	clock++;
	time += 10;
	dut->eval();
	tfp->dump( time );
	dut->clk = 0;
	time += 10;

	dut->eval();
	tfp->dump( time );

}

pattern_table_entry_t patt = {
	.id = 0,
	.sprite = 0,
	.color_table = 0,
};
for (int i = 0; i < 1200; i++)
{

	//CLOCK HIGH
	dut->clk = 1;
	clock++;
	time += 10;

	dut->writedata = pattern_to_int(&patt);
	dut->address = PATTERN_TABLE_MEMORY_OFFSET + i;
```

```
            dut->eval();
            tfp->dump( time );

            //CLOCK LOW
            dut->clk = 0;
            time += 10;

            dut->eval();
            tfp->dump( time );

      }

      // Write all of the sprites to the Sprite Attribute Table. Will take
NUM_SPRITES clock cycles.
      for(int i = 0; i < 16; i++) {
            //CLOCK HIGH
            dut->clk = 1;
            clock++;
            time += 10;

            std::cout << "Writing sprite attr table data" << std::endl;
            std::cout << std::bitset<32> (attr_to_int(&attr)) << std::endl;
            dut->writedata = attr_to_int(&attr);
            dut->address = ATTR_TABLE_MEMORY_OFFSET + i;

            dut->eval();
            tfp->dump( time );

            //CLOCK LOW
            dut->clk = 0;
            time += 10;

            dut->eval();
            tfp->dump( time );
            attr.coord.y  += 5;

      }

      // Read all of the sprites attribute entries from the table. Will take 16
clock cycles.
      // Compare them to what was input


      for(int i = 0; i < 16; i++) {
            for (int c = 0; c < 16; c++) {
                  //CLOCK HIGH
```

```
                    dut->clk = 1;
                    clock++;
                    time += 10;

                    //std::cout << attr_to_int(&attr) << std::endl;
                    dut->writedata = sprite.line[c];
                    dut->address = SPRITE_TABLE_MEMORY_OFFSET +  c +  i *
SPRITE_TABLE_ENTRY_SIZE;

                    dut->eval();
                    tfp->dump( time );


                    //CLOCK LOW
                    dut->clk = 0;
                    time += 10;

                    dut->eval();
                    tfp->dump( time );
             }
      }


      for(int i = 0; i < 16; i++) {
             for (int c = 0; c < 4; c++) {
                    //CLOCK HIGH
                    dut->clk = 1;
                    clock++;
                    time += 10;

                    std::cout << color_to_int(&color_palette.color[c]) << "@" <<
COLOR_TABLE_MEMORY_OFFSET + c + i <<std::endl;
                    dut->writedata = color_to_int(&(color_palette.color[c]));
                    dut->address = COLOR_TABLE_MEMORY_OFFSET + c + i *
COLOR_TABLE_ENTRY_SIZE;

                    dut->eval();
                    tfp->dump( time );


                    //CLOCK LOW
                    dut->clk = 0;
                    time += 10;

                    dut->eval();
                    tfp->dump( time );
             }
```

```
	}
	dut->chipselect = 0;
	dut->write = 0;
	while (clock < 512000) {
		//CLOCK HIGH
		dut->clk = 1;
		clock++;
		time += 10;
		dut->eval();
		tfp->dump( time );


		//CLOCK LOW
		dut->clk = 0;
		time += 10;

		dut->eval();
		tfp->dump( time );
	}



	//
	//	for(int i = 0; i < 16; ++i){
	//		dut->clk = 1;
	//		dut->write_data = kColorTableData[i].colors;
	//
	//		dut->address = COLOR_TABLE_MEMORY_OFFSET +
COLOR_TABLE_ENTRY_SIZE * kColorTableData[i].id;
	//		dut->eval();
	//		// TODO(kristenshaker): verify that this entry in the memory
table was set correctly.
	//		dut->clk=0;
	//		dut->eval();
	//	}
	//

	tfp->close();
	delete tfp;

	dut->final();
	delete dut;
	return 0;

}

#include <iostream>
```

```
#include "Vmemory.h"
#include <verilated.h>
#include <verilated_vcd_c.h>
#include <cassert>

int main(int argc, const char ** argv, const char ** env) {
  Verilated::commandArgs(argc, argv);

  int n;
  if (argc > 1 && argv[1][0] != '+')
    n = atoi(argv[1]);
  else
    n = 7;

  Vmemory * dut = new Vmemory;

  Verilated::traceEverOn(true);
  VerilatedVcdC * tfp = new VerilatedVcdC;
  dut->trace(tfp, 99);
  tfp->open("memory.vcd");

  dut->clk = 0;
  dut->data_in = 0;

  int time = 0;

  //Write to memory
  for (int i = 0 ; i < 16 ; i++) {
    dut->clk = 0;
    dut->we = 1;
    dut->addr = i;
    dut->data_in = i;
    time += 10;

    dut->eval();
    tfp->dump( time );

    dut->clk = 1;
    time += 10;

    dut->eval();
    tfp->dump( time );

    std::cout << "(" << i << ") Writing "<< dut->data_in << " to address " <<
dut->addr  << std::endl;
  }
```

```
  dut->we = 0;
  //Read from  memory
  for (int i = 0 ; i < 16 ; i++) {
    dut->clk = 0;
    dut->addr = i;
    time += 10;

    dut->eval();
    tfp->dump( time );

    // on the pos edge we are always reading the memory at addr even if the we bit
is not enabled.
    dut->clk = 1;
    time += 10;

    dut->eval();
    tfp->dump( time );
    std::cout << i << ' ' << dut->data_out << std::endl;
    assert(i == dut->data_out);

  }
  tfp->close();
  delete tfp;

  dut->final();
  delete dut;

  return 0;
}

#include <stdint.h>
#include <iostream>
#include "Vshift.h"
#include <verilated.h>
#include <verilated_vcd_c.h>
#include <bitset>

int main(int argc, const char ** argv, const char ** env) {
  Verilated::commandArgs(argc, argv);

  // TODO (ask bryce this is greater than 32 bits?)
  int64_t d_in = 0x53977053977;
  // In binary this is 0101001110010111011100000101001110010111011111.
  // Broken into 2 bit pieces it is
  std::bitset<32> d_in_bits (d_in);
```

```
  Vshift * dut = new Vshift;
  Verilated::traceEverOn(true);
  VerilatedVcdC * tfp = new VerilatedVcdC;
  dut->trace(tfp, 99);
  // TODO Bryce what is this trace and why do we have to open .vcd?
  tfp->open("shift.vcd");

  dut->clk = 0;
  dut->data_in = 0;

  int time = 0;

  dut->clk = 0;
  dut->ld = 1;
  std::cout << "Writing "<< d_in <<  std::endl;

  dut->data_in = d_in;
  // TODO Does it need 16 clock cycles to read in the data? Doesn't it just need
one? Are we doing this 16 times superfluously?
  for (int i = 0 ; i < 16 ; i++) {
    time += 10;

    dut->eval();
    tfp->dump( time );

    dut->clk = 1;
    time += 10;

    dut->eval();
    tfp->dump( time );
    dut->clk = 0;

  }

  dut->ld = 0;
  dut->en = 1;
  //Read from  memory. Every cycle we will read out 2 bits.
  for (int i = 0 ; i < 16 ; i++) {
    dut->clk = 0;
    time += 10;

    dut->eval();
    tfp->dump( time );

    dut->clk = 1;
    time += 10;
```

```
    dut->eval();
    tfp->dump( time );
    std::cout << "datout: " << std::bitset<3>(dut->data_out) << std::endl;

  }
  tfp->close();
  delete tfp;

  dut->final();
  delete dut;

  return 0;
}

#include <stdint.h>
#include <iostream>
#include "Vdown_counter.h"
#include <verilated.h>
#include <verilated_vcd_c.h>
#include <bitset>

int main(int argc, const char ** argv, const char ** env) {
  Verilated::commandArgs(argc, argv);

  int n;
  if (argc > 1 && argv[1][0] != '+')
    n = atoi(argv[1]);
  else
    n = 7;

  Vdown_counter * dut = new Vdown_counter;
  Verilated::traceEverOn(true);
  VerilatedVcdC * tfp = new VerilatedVcdC;
  dut->trace(tfp, 99);
  tfp->open("down_counter.vcd");

  dut->clk = 0;
  dut->data_in = 0;

  int time = 0;

  dut->clk = 0;
  dut->ld = 1;
  std::cout << "Writing "<< n <<  std::endl;

  dut->data_in = n;
  for (int i = 0 ; i < 16 ; i++) {
```

```
    time += 10;

    dut->eval();
    tfp->dump( time );

    dut->clk = 1;
    time += 10;

    dut->eval();
    tfp->dump( time );
    dut->clk = 0;

  }

  dut->ld = 0;
  //Read from  memory
  int count = 0;
  while(!dut->done){
    dut->clk = 0;
    dut->en = 1;
    time += 10;

    dut->eval();
    tfp->dump( time );

    dut->clk = 1;
    time += 10;

    dut->eval();
    tfp->dump( time );
    std::cout << "(" << count++ << ") done: " << std::bitset<1>(dut->done) <<
std::endl;

  }
  std::cout << "(" << count++ << ") done: " << std::bitset<1>(dut->done) <<
std::endl;
  tfp->close();
  delete tfp;

  dut->final();
  delete dut;

  return 0;
}

/*
```

DRIVER AND USERSPACE CODE

```
This module is responsible for pulling data from userspace and writing it to FPGA
memory using
iowrite calls.

*/
#ifndef _PPU_H_
#define _PPU_H_

#include <linux/ioctl.h>

#ifndef __KERNEL__
#include <stdint.h>
#else
#include <linux/types.h>
#endif

// sizes of each entry in various tables in memory
#define SPRITE_TABLE_ENTRY_SIZE (16)
#define COLOR_TABLE_ENTRY_SIZE  (4)
#define ATTR_TABLE_ENTRY_SIZE   (1)

// How the bits for each attr table entry are laid out.
#define OBJ_Y_COORD_OFFSET 0
#define OBJ_X_COORD_OFFSET 10
#define OBJ_SPRITE_OFFSET 20
#define OBJ_COLOR_OFFSET 28

#define ATTR_TABLE_MEMORY_OFFSET    (0x0000 * 4)
#define COLOR_TABLE_MEMORY_OFFSET   (0x1000 * 4)
#define PATTERN_TABLE_MEMORY_OFFSET (0x2000 * 4)
#define SPRITE_TABLE_MEMORY_OFFSET  (0x3000 * 4)

// first argument is dev.base, second argument is distance from table offset.
#define ATTR_WRITE(x, y)  (x + ATTR_TABLE_MEMORY_OFFSET    + (y * 4))
#define SPRITE_WRITE(x,y) (x + SPRITE_TABLE_MEMORY_OFFSET  + (y * 4))
#define PATTERN_WRITE(x,y)(x + PATTERN_TABLE_MEMORY_OFFSET + (y * 4))
#define COLOR_WRITE(x, y) (x + COLOR_TABLE_MEMORY_OFFSET   + (y * 4))

//Write to address structure - used for debugging.
struct wta {
    int addr;
    int value;
};

typedef struct {
```

```
	// signed so that we can go negative when sprite is partially on the screen.
	int x,y;
} sprite_coord_t;

// An entry in the sprite attribution table.
typedef struct {
	// Location of this attr table entry on the VGA monitor.
	sprite_coord_t coord;
	// Sprite table offset.
	char sprite;
	// Color table offset with RBG values for whichever sprite this attr table
entry represents.
	char color_table;
	// Unique id associated with an attr table entry. Offset in the attr table
will
	// be computed by multiplying id * ATTR_TABLE_ENTRY_SIZE.
	int id;
} attr_table_entry_t;

#define PAT_ID_OFFSET 0
#define PAT_SPRITE_OFFSET 0
#define PAT_COLOR_OFFSET 8
typedef struct {
    uint32_t id;
    char     sprite;
    char     color_table;
} pattern_table_entry_t;


typedef struct {
    uint32_t id;
    uint32_t line[16];
} sprite_table_entry_t;

#define RED_OFFSET 0
#define GREEN_OFFSET 8
#define BLUE_OFFSET 16
typedef struct {
    int r, g, b;
} color_t;

typedef struct {
    int id;
    color_t color[4]; //, color1, color2, color3;
} color_table_entry_t;

/* ioctls and their arguments */
```

```
#define PPU_MAGIC 'p'
#define ATTR_TABLE_WRITE_DATA     _IOW(PPU_MAGIC, 1, attr_table_entry_t *)
#define PATTERN_TABLE_WRITE_DATA  _IOW(PPU_MAGIC, 2, color_table_entry_t *)
#define SPRITE_TABLE_WRITE_DATA   _IOW(PPU_MAGIC, 3, sprite_table_entry_t *)
#define COLOR_TABLE_WRITE_DATA    _IOW(PPU_MAGIC, 4, color_table_entry_t *)
#define WRITE_TO_ADDRESS          _IOW(PPU_MAGIC, 5, struct wta *)

#endif
#include "ppu.h"
#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>

#define DRIVER_NAME "ppu"

/*
 * Information about our device
 */
struct ppu_dev {
       struct resource res; /* Resource: our registers */
       void __iomem *virtbase; /* Where registers can be accessed in memory */
} dev;

static void write_to_address(struct wta *wv)
{
    iowrite32(wv->value, dev.virtbase + wv->addr);
}

// Write to the attribution table sprite related information. Addr will vary based
on which sprite we are updating.
static void write_to_attr_table(attr_table_entry_t *attr)
{

       int data = 0x0;

       data = data | (attr->coord.x << OBJ_X_COORD_OFFSET);
       data = data | (attr->coord.y << OBJ_Y_COORD_OFFSET);
```

```
        data = data | (attr->sprite          << OBJ_SPRITE_OFFSET);
        data = data | (attr->color_table << OBJ_COLOR_OFFSET);

        iowrite32(data, ATTR_WRITE(dev.virtbase, attr->id * ATTR_TABLE_ENTRY_SIZE));

    printk(KERN_INFO "Wrote attribute(%d) to %x\n", attr->id,
            (unsigned int) ATTR_WRITE(dev.virtbase, (attr->id *
ATTR_TABLE_ENTRY_SIZE)));

}

static void write_to_pattern_table(pattern_table_entry_t *pat)
{

        int data = 0x0;
        data = data | (pat->sprite << PAT_SPRITE_OFFSET);
        data = data | (pat->color_table << PAT_COLOR_OFFSET);

        iowrite32(data, PATTERN_WRITE(dev.virtbase, pat->id));

        printk(KERN_INFO "Wrote pattern(%d) to %x\n", pat->id,
        (unsigned int) PATTERN_WRITE(dev.virtbase, pat->id));

}


static void write_to_color_table(color_table_entry_t *color_palette)
{
    // All color tables take up COLOR_TABLE_ENTRY_SIZE 'rows' in the 32 bit FPGA
memory. Each call to this function represents one color
    // table being written.
    int color = 0x0;
    int i = 0;

    for (; i < COLOR_TABLE_ENTRY_SIZE; i++) {
        color = 0x0;
        color = color | (color_palette->color[i].r << RED_OFFSET);
        color = color | (color_palette->color[i].b << BLUE_OFFSET);
        color = color | (color_palette->color[i].g << GREEN_OFFSET);
        iowrite32(color, COLOR_WRITE(dev.virtbase , (color_palette->id *
COLOR_TABLE_ENTRY_SIZE * 4) + i));
    }
    printk(KERN_INFO "Wrote color palette(%d) to %x->%x\n", color_palette->id,
            (unsigned int) COLOR_WRITE(dev.virtbase, (color_palette->id *
COLOR_TABLE_ENTRY_SIZE * 4)),
            (unsigned int) COLOR_WRITE(dev.virtbase, (color_palette->id *
```

```
COLOR_TABLE_ENTRY_SIZE * 4) + 3));
}

// Called at program startup to initialize all of the sprites. This data will not
change throughout the lifetime of the program.
static void write_to_sprite_table(sprite_table_entry_t * sprite)
{
       int i = 0;
       for(; i < SPRITE_TABLE_ENTRY_SIZE; i++){
         iowrite32(sprite->line[i], SPRITE_WRITE(dev.virtbase , (sprite->id *
SPRITE_TABLE_ENTRY_SIZE * 4) + i));
       }

    printk(KERN_INFO "Wrote sprite(%d) to %x->%x\n", sprite->id,
            (unsigned int) SPRITE_WRITE(dev.virtbase, (sprite->id *
SPRITE_TABLE_ENTRY_SIZE * 4)),
            (unsigned int) SPRITE_WRITE(dev.virtbase, (sprite->id *
SPRITE_TABLE_ENTRY_SIZE * 4) +
            SPRITE_TABLE_ENTRY_SIZE - 1));
}

/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
 */
static long ppu_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
       attr_table_entry_t      attr_table_entry;
       sprite_table_entry_t    sprite;
       color_table_entry_t     color_palette;
       pattern_table_entry_t   pattern;
       struct wta av;

       switch (cmd) {
              case ATTR_TABLE_WRITE_DATA:
                     if (copy_from_user(&attr_table_entry, (attr_table_entry_t *)
arg,
                                                sizeof(attr_table_entry_t)))
                            return -EACCES;
                     write_to_attr_table(&attr_table_entry);
                     break;

              case SPRITE_TABLE_WRITE_DATA:
                     if (copy_from_user(&sprite, (sprite_table_entry_t*) arg,
sizeof(sprite_table_entry_t)))
                            return -EACCES;
```

```
                write_to_sprite_table(&sprite);
                break;

        case PATTERN_TABLE_WRITE_DATA:
                if (copy_from_user(&pattern, (pattern_table_entry_t*) arg,
                                   sizeof(pattern_table_entry_t)))
                        return -EACCES;
                write_to_pattern_table(&pattern);
                break;

        case COLOR_TABLE_WRITE_DATA:
                if (copy_from_user(&color_palette, (color_table_entry_t*) arg,
                                   sizeof(color_table_entry_t)))
                        return -EACCES;
                write_to_color_table(&color_palette);
                break;

     case WRITE_TO_ADDRESS:
                if (copy_from_user(&av, (struct wta*) arg, sizeof(struct wta)))
            return -EACCES;

        write_to_address(&av);
        break;
         default:
                return -EINVAL;
    }

    return 0;
}

/* The operations our device knows how to do */
static const struct file_operations ppu_fops = {
    .owner      = THIS_MODULE,
    .unlocked_ioctl = ppu_ioctl,
};

/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice ppu_misc_device = {
    .minor      = MISC_DYNAMIC_MINOR,
    .name       = DRIVER_NAME,
    .fops       = &ppu_fops,
};

/*
 * Initialization code: get resources (registers).
 */
static int __init ppu_probe(struct platform_device *pdev)
```

```
{
	int ret;

	/* Register ourselves as a misc device: creates /dev/vga_ball */
	ret = misc_register(&ppu_misc_device);

	/* Get the address of our registers from the device tree */
	ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
	if (ret) {
		ret = -ENOENT;
		goto out_deregister;
	}

	/* Make sure we can use these registers */
	if (request_mem_region(dev.res.start, resource_size(&dev.res),
			       DRIVER_NAME) == NULL) {
		ret = -EBUSY;
		goto out_deregister;
	}

	/* Arrange access to our registers */
	dev.virtbase = of_iomap(pdev->dev.of_node, 0);
	if (dev.virtbase == NULL) {
		ret = -ENOMEM;
		goto out_release_mem_region;
	}

	return 0;

out_release_mem_region:
	release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
	misc_deregister(&ppu_misc_device);
	return ret;
}

/* Clean-up code: release resources */
static int ppu_remove(struct platform_device *pdev)
{
	iounmap(dev.virtbase);
	release_mem_region(dev.res.start, resource_size(&dev.res));
	misc_deregister(&ppu_misc_device);
	return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
```

```
static const struct of_device_id ppu_of_match[] = {
	// set this in the hardware program thingy
	{ .compatible = "csee4840,ppu-1.0" },
	{},
};
MODULE_DEVICE_TABLE(of, ppu_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver ppu_driver = {
	.driver	= {
		.name  = DRIVER_NAME,
		.owner = THIS_MODULE,
		.of_match_table = of_match_ptr(ppu_of_match),
	},
	.remove	= __exit_p(ppu_remove),
};

/* Called when the module is loaded: set things up */
static int __init ppu_init(void)
{
	printk(KERN_INFO DRIVER_NAME ": init\n");
	return platform_driver_probe(&ppu_driver, ppu_probe);
}

/* Calball when the module is unloaded: release resources */
static void __exit ppu_exit(void)
{
	platform_driver_unregister(&ppu_driver);
	printk(KERN_INFO DRIVER_NAME ": exit\n");
}

module_init(ppu_init);
module_exit(ppu_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Kristen Shaker");
MODULE_DESCRIPTION("PPU driver");
/*
   Handles all sprite related data.
*/

#ifndef _SPRITES_H
#define _SPRITES_H

#include "ppu.h"
#include "game/game.h"
```

```
#ifndef __KERNEL__
#include <stdint.h>
#else
#include <linux/types.h>
#endif

#define NUM_SPRITES 64
#define NUM_COLOR_TABLE_ENTRIES 7
#define NUM_ATTR_TABLE_ENTRIES 64
#define NUM_SPRITES_PER_DUCK 4

#define NUM_BULLETS 3
#define NUM_SCORE_DIGITS 2

// Offsets of various classes of sprites in the sprite table.
// The 0th offset is reserved for sprites we do not want to see on the screen.
#define DUCK_DOWN_EAST_SPRITE_OFFSET 1
#define DUCK_UP_EAST_SPRITE_OFFSET 5
#define DUCK_DOWN_WEST_SPRITE_OFFSET 13
#define DUCK_UP_WEST_SPRITE_OFFSET 17
#define DUCK_DEAD_SPRITE_OFFSET 29
#define DUCK_FLYING_AWAY_SPRITE_OFFSET 33

#define BULLET_SPRITE_OFFSET 37
// Shaded and non shaded bullets will share the same sprite but point
// to different color tables.
#define NUMBER_SPRITE_OFFSET 38
// 6 + 10 digits = 16
#define CROSSHAIR_SPRITE_OFFSET 48

// Offsets of various classes of entries in the sprite attribution table.
// The order in which these entries are laid out is an implementation decision.
#define DUCK_ATTR_TABLE_OFFSET 0
#define BULLET_ATTR_TABLE_OFFSET NUM_DUCKS_PER_ROUND * NUM_SPRITES_PER_DUCK
#define SCORE_ATTR_TABLE_OFFSET BULLET_ATTR_TABLE_OFFSET + NUM_BULLETS
#define ROUND_ATTR_TABLE_OFFSET SCORE_ATTR_TABLE_OFFSET + NUM_SCORE_DIGITS
#define CROSSHAIR_ATTR_TABLE_OFFSET ROUND_ATTR_TABLE_OFFSET + 1

#define DUCK_COLOR_TABLE_OFFSET 0
#define SHADED_BULLET_COLOR_TABLE_OFFSET 1
#define UNSHADED_BULLET_COLOR_TABLE_OFFSET 2
#define NUMBER_LETTER_COLOR_TABLE_OFFSET 3

// Populates attr_table_entry_t array with all of the attr table entries. Because
of the number of entries in this table and how much data is stored in each entry,
it is more readible to populate the entries programtically than by initializing a
```

```
global array.
int build_sprite_attr_table(attr_table_entry_t * entries);

// Writes all entries in the attribution table to FPGA memory using ioctl calls.
Returns 1 if writes succesful, 0 on failure.
int write_sprite_attr_table(int fd);

// Writes the all data in kSpriteTableData to FPGA memory using ioctl calls.
Returns 1 if writes succesful, 0 on failure.
int write_sprite_table(int fd);

// Write the color table to FPGA memory using ioctl calls. Returns 1 if writes
succesful, 0 on failure.
int write_color_table(int fd);

// Updates score and number of bullets remaining every round.
int update_game_state_attrs(int fd, int num_bullets, int score, int round);

// Updates the duck attr for the duck corresponding to duck_id in the attr table
every round.
int update_duck_attr(int fd, duck_t* duck);

// Updates the crosshair attribute for display.
int update_crosshair_attr(int fd, int x_coord, int y_coord);
int write_pattern_table(int fd, int back_c);
#endif
#include "sprites.h"
#include <string.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <stdio.h>
#include <unistd.h>

const int kBulletSpriteXLoc = 5;
const int kScoreSpriteXLoc = 500;
const int kLowerGraphicYLoc = 400;
const int kBulletSpaceApart = 2;
const int kScoreSpaceApart = 0;
const int kSmallGraphicWidth = 16;
const int kSmallGraphicHeight = 16;

attr_table_entry_t attr_table[NUM_SPRITES] = {};

int build_sprite_attr_table(attr_table_entry_t * entries){

	int num_entries = 0;
	int i = 0;
```

```
	for(;i < NUM_DUCKS_PER_ROUND * NUM_SPRITES_PER_DUCK; ++i){
		attr_table_entry_t duck = {
			.coord = { .x = 0, .y = 0 },
			// duck starts off screen;
			.sprite = 0x0,
			.id = num_entries,
			.color_table = DUCK_COLOR_TABLE_OFFSET,
		};
		entries[num_entries] = duck;
		++(num_entries);
	}

	i = 0;
	for(;i < NUM_BULLETS; ++i){
		attr_table_entry_t bullet = {
			.coord =  {
				.x = kBulletSpriteXLoc + i * kSmallGraphicWidth + i *
kBulletSpaceApart,
				.y = kLowerGraphicYLoc
			},
			// all bullets start off shaded and on screen.
			.sprite = BULLET_SPRITE_OFFSET,
			.id = num_entries,
			.color_table = SHADED_BULLET_COLOR_TABLE_OFFSET
		};
		entries[num_entries] = bullet;
		++(num_entries);
	}

	i = 0;
	for(;i< NUM_SCORE_DIGITS; ++i){
		attr_table_entry_t score = {
			.coord = { .x = kScoreSpriteXLoc + i * kSmallGraphicWidth+ i*
kScoreSpaceApart, .y = kLowerGraphicYLoc },
			// score starts off 0 0
			.sprite = NUMBER_SPRITE_OFFSET,
			.id = num_entries,
			//TODO(WHY WON"T THIS WORK WITH THE COLOR TABLE OFFSET OF 4?)
			// this should be SHADED_BULLET_COLOR_TABLE_OFFSET
			.color_table = NUMBER_LETTER_COLOR_TABLE_OFFSET
		};
		entries[num_entries] = score;
		++(num_entries);
	}

	attr_table_entry_t round = {
		// Round sprite is right above bullets.
```

```
            .coord = {.x = kBulletSpriteXLoc, .y = kLowerGraphicYLoc -
kSmallGraphicHeight - 10 },
            // round starts at 0
            .sprite = NUMBER_SPRITE_OFFSET,
            .id = num_entries,
            .color_table = NUMBER_LETTER_COLOR_TABLE_OFFSET
      };

      entries[num_entries] = round;
      ++(num_entries);

      attr_table_entry_t crosshair;
      // TODO(kristenshaker): change these coords
      printf("Crosshair at: %d \n", num_entries);
      crosshair.coord.x = 600;
      crosshair.coord.y = 10;
      crosshair.sprite = CROSSHAIR_SPRITE_OFFSET;
      crosshair.id = num_entries;
      crosshair.color_table = 0x1;

      entries[num_entries]=crosshair;

      return 1;
}


int write_sprite_table(int fd){

      // This data is read into FPGA memory once at start up time. It is not
required
      // anywhere else, so it will be kept a local variable.
      // TODO(kristenshaker): populate this table with the rest of the sprites and
accurate sprite data.
      sprite_table_entry_t sprites[NUM_SPRITES] = {
            // Sprite not on the screen.
            [0] = {
                  .id  = 0x0,
                  .line = {
                        0xF0F0F0F0, 0xF0F0F0F0, 0xF0F0F0F0, 0xF0F0F0F0,
                        0xF00FF00F, 0xF00FF00F, 0xF00FF00F, 0xF00FF00F,
                        0xAAAAAAAA, 0xAAAAAAAA, 0xAAAAAAAA, 0xAAAAAAAA,
                        0x55555555, 0x55555555, 0x55555555, 0x55555555,
                  },
            },
            // Duck Flap Down Top Left
            [1] = {
                  .id  = 0x1,
```

```
        .line = {
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x15000000,
                0x55550000,
                0x55555400,
                0x55555554,
                0x55555555,
                0x55555558,
                0x55555560,
                0x55555580,
                0x55565a80,
                0x5556aa00,
                0x555aa800,
        },
},
// Duck Flap Down Top Right
[2] = {
        .id  = 2,
        .line = {
                0x555ac000,
                0x5553ff00,
                0x5560ff00,
                0x55603f00,
                0x55800c00,
                0x59800000,
                0x9a000000,
                0xa0000000,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
        },
},
// Duck Flap Down Bottom Left
[3] = {
        .id  = 3,
        .line = {
                0x0,
                0x3bc000,
```

```
                0xeaf000,
                0xdafc00,
                0x3daff00,
                0x3feaff90,
                0x3ffbff95,
                0x3fffff95,
                0x3ffe95,
                0xe55,
                0x55,
                0x5,
                0x2,
                0x0,
                0x0,
                0x0,
        },
},
// Duck Flap Down Bottom Right
[4] = {
        .id  = 4,
        .line = {
                0x1,
                0x1,
                0x1,
                0x1,
                0x5,
                0x5,
                0x5,
                0x15,
                0x16,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
        },
},
// Duck Flap Up Top Left
[5] = {
        .id  = 5,
        .line = {
                0x0,
                0x50000000,
                0x64000000,
                0xa9800000,
                0xaa900000,
```

```
                0xaa900000,
                0xaaa00000,
                0xaaa00000,
                0xaaa00000,
                0xaaa80000,
                0x6aa80000,
                0x2aa80000,
                0x2aa80000,
                0x6aa94000,
                0x5aa95500,
                0x5aa95554,
        },
},
// Duck Flap Up Top Right
[6] = {
        .id  = 6,
        .line = {
                0x5aa95555,
                0xaaa95558,
                0xaaa95560,
                0xaaaa5680,
                0x2aaaaa80,
                0x2aaaa00,
                0x2aa800,
                0x2c000,
                0xff00,
                0x3f00,
                0xf00,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
        },
},
// Duck Flap Up Bottom Left
[7] = {
        .id  = 7,
        .line = {
                0x14,
                0x19,
                0x1a,
                0x6,
                0x6,
                0x2,
                0x1,
                0x0,
```

```
                0xbc000,
                0x2af000,
                0x1afc00,
                0xdaff00,
                0x3feaff90,
                0x3ffbff95,
                0x3fffff95,
                0xffe95,
        },
},
// Duck Flap Up Bottom Right
[8] = {
        .id  = 8,
        .line = {
                0x255,
                0x15,
                0x2,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
        },
},
// Duck Flap Mid Top Left
[9] = {
        .id  = 9,
        .line = {
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
```

```
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
        },
},
// Duck Flap Mid Top Right
[10] = {
        .id  = 10,
        .line = {
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
        },
},
// Duck Flap Mid Bottom Left
[11] = {
        .id  = 11,
        .line = {
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
```

```
                0x0,
                0x0,
        },
},
// Duck Flap Mid Bottom Right
[12] = {
        .id  = 12,
        .line = {
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
        },
},

// LEFT FACING DUCK
// Duck Flap Down Top Left
[15] = {
        .id  = 15,
        .line = {
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x54,
                0x5555,
                0x155555,
                0x15555555,
                0x55555555,
                0x25555555,
                0x9555555,
                0x2555555,
                0x2a59555,
                0xaa9555,
```

```
                0x2aa555,
        },
},
// Duck Flap Down Top Right
[16] = {
        .id  = 16,
        .line = {
                0x3a555,
                0xffc555,
                0xff0955,
                0xfc0955,
                0x300255,
                0x265,
                0xa6,
                0xa,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
        },
},
// Duck Flap Down Bottom Left
[13] = {
        .id  = 13,
        .line = {
                0x0,
                0x3ec00,
                0xfab00,
                0x3fa700,
                0xffa7c0,
                0x6ffabfc,
                0x56ffeffc,
                0x56fffffc,
                0x56bffc00,
                0x55b00000,
                0x55000000,
                0x50000000,
                0x80000000,
                0x0,
                0x0,
                0x0,
        },
},
```

```
// Duck Flap Down Bottom Right
[14] = {
	.id  = 14,
	.line = {
		0x40000000,
		0x40000000,
		0x40000000,
		0x40000000,
		0x50000000,
		0x50000000,
		0x50000000,
		0x54000000,
		0x94000000,
		0x0,
		0x0,
		0x0,
		0x0,
		0x0,
		0x0,
		0x0,
	},
},
// Duck Flap Up Top Left
[19] = {
	.id  = 19,
	.line = {
		0x0,
		0x5,
		0x19,
		0x26a,
		0x6aa,
		0x6aa,
		0xaaa,
		0xaaa,
		0xaaa,
		0x2aaa,
		0x2aa9,
		0x2aa8,
		0x2aa8,
		0x16aa9,
		0x556aa5,
		0x15556aa5,
	},
},
// Duck Flap Up Top Right
[20] = {
	.id  = 20,
```

```
        .line = {
                0x55556aa5,
                0x25556aaa,
                0x9556aaa,
                0x295aaaa,
                0x2aaaaa8,
                0xaaaa80,
                0x2aa800,
                0x38000,
                0xff0000,
                0xfc0000,
                0xf00000,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
        },
},
// Duck Flap Up Bottom Left
[17] = {
        .id  = 17,
        .line = {
                0x14000000,
                0x64000000,
                0xa4000000,
                0x90000000,
                0x90000000,
                0x80000000,
                0x40000000,
                0x0,
                0x3e000,
                0xfa800,
                0x3fa400,
                0xffa700,
                0x6ffabfc,
                0x56ffeffc,
                0x56fffffc,
                0x56bff000,
        },
},
// Duck Flap Up Bottom Right
[18] = {
        .id  = 18,
        .line = {
                0x55800000,
                0x54000000,
```

```
                0x80000000,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
        },
},
// Duck Flap Mid Top Left
[21] = {
        .id  = 21,
        .line = {
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
        },
},
// Duck Flap Mid Top Right
[22] = {
        .id  = 22,
        .line = {
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
```

```
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
        },
},
// Duck Flap Mid Bottom Left
[23] = {
        .id  = 23,
        .line = {
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
        },
},
// Duck Flap Mid Bottom Right
[24] = {
        .id  = 24,
        .line = {
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
```

```
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
        },
},

// Duck Just Shot Top Left
[25] = {
        .id  = 25,
        .line = {
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
        },
},
// Duck Just Shot Top Right
[26] = {
        .id  = 26,
        .line = {
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
```

```
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
        },
},
// Duck Just Shot Bottom Left
[27] = {
        .id  = 27,
        .line = {
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
        },
},
// Duck Just Shot Bottom Right
[28] = {
        .id  = 28,
        .line = {
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
```

```
                0x0,
                0x0,
                0x0,
        },
},


// Duck Dead Top
[29] = {
        .id  = 29,
        .line = {
                0x0,
                0x110000,
                0x150000,
                0x58000,
                0x6a300,
                0x2896af20,
                0x2896bf2c,
                0x2656bf7f,
                0x1596bfaf,
                0x1565afaf,
                0x1565aaaf,
                0x1559aaac,
                0x1555aa00,
                0x15556a00,
                0x5556a00,
                0x1556a00,
        },
},
// Duck Dead Bottom
[30] = {
        .id  = 30,
        .line = {
                0x556800,
                0x556800,
                0x156400,
                0x155000,
                0x55000,
                0x54000,
                0x54000,
                0xa8000,
                0xfc000,
                0x3ffc00,
                0x3fff30,
                0xffaff0,
                0xfe6bc0,
                0xfd57fc,
```

```
                0x3d54ff,
                0x0,
        },
},
// Duck Dead Empty
[31] = {
        .id  = 31,
        .line = {
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
        },
},
// Duck Dead Empty 2
[32] = {
        .id  = 32,
        .line = {
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
                0x0,
        },
```

```
},
// Duck Flying Away Top Left
[33] = {
	.id  = 33,
	.line = {
		0x0,
		0x0,
		0x80000000,
		0xa0000000,
		0xf8000000,
		0xfc000000,
		0xff000000,
		0xff000000,
		0xff000000,
		0xff000000,
		0xac000000,
		0x60000000,
		0x50000000,
		0x50000000,
		0x50015500,
		0x54155550,
	},

},
// Duck Flying Away Top Right
[34] = {
	.id  = 34,
	.line = {
		0x55555555,
		0x55555556,
		0x55555552,
		0x55556968,
		0x55568280,
		0x55280000,
		0x55400000,
		0x55400000,
		0x55400000,
		0x95800000,
		0xa6800000,
		0xaa000000,
		0xab000000,
		0xafc00000,
		0xfc00000,
		0x0,
	},

},
```

```
// Duck Flying Away Bottom Left
[35] = {
	.id  = 35,
	.line = {
		0x0,
		0x0,
		0x2,
		0xa,
		0x2f,
		0x3f,
		0xff,
		0xff,
		0xff,
		0xff,
		0x3a,
		0x9,
		0x5,
		0x5,
		0x554005,
		0x5555415,
	},

},
// Duck Flying Away Bottom Right
[36] = {
	.id  = 36,
	.line = {
		0x15555555,
		0x15555555,
		0x5555555,
		0x29695555,
		0x2829555,
		0x2855,
		0x155,
		0x155,
		0x155,
		0x256,
		0x29a,
		0xaa,
		0xea,
		0x3fa,
		0x3f0,
		0x0,
	},

},
// Bullet Shaded & Non Shaded
```

```
[37] = {
       .id  = 37,
       .line = {
              0x0,
              0x0,
              0x155400,
              0x555500,
              0x5aa500,
              0x5aa500,
              0x5eb500,
              0x5eb500,
              0x5eb500,
              0x5eb500,
              0x5aa500,
              0x5aa500,
              0x555500,
              0x155400,
              0x0,
              0x0,
       },

},
// Number Sprites
// 0
[38] = {
       .id  = 38,
       .line = {
              0x0,
              0x555500,
              0x1555540,
              0x56aa950,
              0x5aaaa50,
              0x5a55a50,
              0x5a55a50,
              0x5a55a50,
              0x5a55a50,
              0x5a55a50,
              0x5a55a50,
              0x5aaaa50,
              0x56aa950,
              0x1555540,
              0x555500,
              0x0,
       },

},
// 1
```

```
[39] = {
       .id  = 39,
       .line = {
              0x0,
              0x15540,
              0x55540,
              0x15a940,
              0x16a940,
              0x169540,
              0x169540,
              0x169400,
              0x169400,
              0x1569540,
              0x1569540,
              0x16aa940,
              0x16aa940,
              0x1555540,
              0x1555540,
              0x0,
       },

},
// 2
[40] = {
       .id = 40,
       .line = {
              0x0,
              0x555550,
              0x1555550,
              0x56aaa50,
              0x5aaaa50,
              0x5a55550,
              0x5a55540,
              0x5aaa950,
              0x56aaa50,
              0x1555a50,
              0x5555a50,
              0x5aaaa50,
              0x5aaaa50,
              0x5555550,
              0x5555550,
              0x0,
       }
},
// 3
[41] = {
       .id = 41,
```

```
        .line = {
                0x0,
                0x555550,
                0x1555550,
                0x56aaa50,
                0x5aaaa50,
                0x5a55550,
                0x5a55550,
                0x5aaaa50,
                0x5aaaa50,
                0x5a55550,
                0x5a55550,
                0x5aaaa50,
                0x56aaa50,
                0x1555550,
                0x555550,
                0x0,
        }
},
// 4
[42] = {
        .id = 42,
        .line = {
                0x0,
                0x5555550,
                0x5555550,
                0x5a55a50,
                0x5a55a50,
                0x5a55a50,
                0x5a55a50,
                0x5aaaa50,
                0x5aaaa50,
                0x5a55550,
                0x5a55550,
                0x5a50000,
                0x5a50000,
                0x5550000,
                0x5550000,
                0x0,
        }
},
// 5
[43] = {
        .id = 43,
        .line = {
                0x0,
                0x5555550,
```

```
                0x5555550,
                0x5aaaa50,
                0x5aaaa50,
                0x5555a50,
                0x1555a50,
                0x56aaa50,
                0x5aaa950,
                0x5a55540,
                0x5a55550,
                0x5aaaa50,
                0x56aaa50,
                0x1555550,
                0x555550,
                0x0,
        }
},
// 6
[44] = {
        .id = 44,
        .line = {
                0x0,
                0x5550,
                0x5550,
                0x5a50,
                0x5a50,
                0x555a50,
                0x1555a50,
                0x56aaa50,
                0x5aaaa50,
                0x5a55a50,
                0x5a55a50,
                0x5aaaa50,
                0x56aa950,
                0x1555540,
                0x555500,
                0x0,
        }
},
// 7
[45] = {
        .id = 45,
        .line = {
                0x0,
                0x555550,
                0x1555550,
                0x56aaa50,
                0x5aaaa50,
```

```
                0x5a55550,
                0x5a55550,
                0x5a50000,
                0x5a50000,
                0x5a50000,
                0x5a50000,
                0x5a50000,
                0x5a50000,
                0x5550000,
                0x5550000,
                0x0,
        }
},
// 8
[46] = {
        .id = 46,
        .line = {
                0x0,
                0x555500,
                0x1555540,
                0x56aa950,
                0x5aaaa50,
                0x5a55a50,
                0x5a55a50,
                0x5aaaa50,
                0x5aaaa50,
                0x5a55a50,
                0x5a55a50,
                0x5aaaa50,
                0x56aa950,
                0x1555540,
                0x555500,
                0x0,
        }
},
// 9
[47] = {
        .id = 47,
        .line = {
                0x0,
                0x555500,
                0x1555540,
                0x56aa950,
                0x5aaaa50,
                0x5a55a50,
                0x5a55a50,
                0x5aaaa50,
```

```
                0x5aaa950,
                0x5a55540,
                0x5a55500,
                0x5a50000,
                0x5a50000,
                0x5550000,
                0x5550000,
                0x0,
        }
},
// Crosshair
[48] = {
        .id = 48,
        .line = {
                0x0,
                0x0,
                0x0,
                0x0,
                0xfc3f00,
                0xffff00,
                0xffff00,
                0x3ffc00,
                0x3ffc00,
                0xffff00,
                0xffff00,
                0xfc3f00,
                0x0,
                0x0,
                0x0,
                0x0,
        }
},
// Grass Body
[49] = {
        .id = 49,
        .line = {
                0x55555555,
                0x55555555,
                0x55555555,
                0x55555555,
                0x55555555,
                0x55555555,
                0x55555555,
                0x55555555,
                0x55555555,
                0x55555555,
                0x55555555,
```

```
                0x55555555,
                0x55555555,
                0x55555555,
                0x55555555,
                0x55555555,
        }
},
// Grass Top
[50] = {
        .id = 50,
        .line = {
                0xffffffff,
                0xffffffff,
                0xffffffff,
                0x5aaaaaaa,
                0xaaaaaaaa,
                0x55555555,
                0x95555555,
                0x55655555,
                0x56555555,
                0x55555555,
                0x55555555,
                0x55555555,
                0x55555555,
                0x55555555,
                0x55555555,
                0x55555555,
        }
},
// Sky
[51] = {
        .id = 51,
        .line = {
                0xaaaaaaaa,
                0xaaaaaaaa,
                0xaaaaaaaa,
                0xaaaaaaaa,
                0xaaaaaaaa,
                0xaaaaaaaa,
                0xaaaaaaaa,
                0xaaaaaaaa,
                0xaaaaaaaa,
                0xaaaaaaaa,
                0xaaaaaaaa,
                0xaaaaaaaa,
                0xaaaaaaaa,
                0xaaaaaaaa,
                0xaaaaaaaa,
```

```
                        0xaaaaaaaa,
                        0xaaaaaaaa,
                    }
                },
        };

        int i = 0;
        for(; i < NUM_SPRITES; ++i){
                if (ioctl(fd, SPRITE_TABLE_WRITE_DATA, &sprites[i])) {
                        perror("ioctl(SPRITE_TABLE_WRITE_DATA) failed");
                        return 0;
                }
        }
        return 1;
}

int write_color_table(int fd){

        color_table_entry_t color_tables[NUM_SPRITES] =
        {
                // Duck
                [0] = {
                        .id = 0x0,
                        .color = {
                                [0] = {.r = 0, .g = 0, .b = 0},
                                [1] = {.r = 0, .g = 0, .b = 168 },
                                [2] = {.r = 252, .g = 252, .b = 252 },
                                [3] = {.r = 188, .g = 0, .b = 188 },
                        },
                },
                // Shaded Bullet
                [1] =
                {
                        .id = 0x1,
                        .color = {
                                [0] = {.r = 0,  .g = 0,   .b = 0  },
                                [1] = {.r = 111, .g = 62, .b = 67 },
                                [2] = {.r = 244, .g = 180, .b = 27 },
                                [3] = {.r = 254, .g = 228, .b = 129 },
                        },
                },
                // Unshaded Bullet
                [2] =
                {
                        .id = 0x2,
                        .color = {
                                [0] = {.r = 0,  .g = 0,   .b = 0  },
```

```
                [1] = {.r = 67, .g = 74, .b = 95 },
                [2] = {.r = 223, .g = 246, .b = 245 },
                [3] = {.r = 223, .g = 246, .b = 245 },
        },
},
// Numbers and Letters
[3] =
{
        .id = 0x3,
        .color = {
                [0] = {.r = 0, .g = 0, .b = 0},
                [1] = {.r = 67, .g = 74, .b = 95 },
                [2] = {.r = 223, .g = 246, .b = 245 },
                [3] = {.r = 0,  .g = 0,   .b = 255},
        },
},
// Grass Background
[4] =
{
        .id = 0x4,
        .color = {
                [0] = {.r = 0,  .g = 0,   .b = 0  },
                [1] = {.r = 113, .g = 170, .b = 52 },
                [2] = {.r = 182, .g = 213, .b = 60 },
                [3] = {.r = 52, .g = 85, .b = 81 },
        },
},
// Sky Background
[5] =
{
        .id = 5,
        .color = {
                [0] = {.r = 0,  .g = 0,   .b = 0  },
                [1] = {.r = 0,  .g = 0,   .b = 0  },
                [2] = {.r = 135, .g = 206, .b = 235 },
                [3] = {.r = 0,  .g = 0,   .b = 0  },
        },
},
// Flash Background
[6] =
{
        .id = 6,
        .color = {
                [0] = {.r = 0,  .g = 0,   .b = 0  },
                [1] = {.r = 1, .g = 1, .b = 1 },
                [2] = {.r = 255,  .g = 255,   .b = 255  },
                [3] = {.r = 0,  .g = 200,   .b = 0  },
```

```
                        },
                },
        };

        int i = 0;
        for(;i< NUM_COLOR_TABLE_ENTRIES; ++i){
                if (ioctl(fd, COLOR_TABLE_WRITE_DATA, &color_tables[i])) {
                        perror("ioctl(COLOR_TABLE_WRITE_DATA) failed");
                        return 0;
                }
        }
}

int write_sprite_attr_table(int fd){
        if(!build_sprite_attr_table(attr_table)){

                return 0;
        }
        int i = 0;
        for(; i < NUM_ATTR_TABLE_ENTRIES; ++i){
                if (ioctl(fd, ATTR_TABLE_WRITE_DATA, &attr_table[i])) {
                        perror("ioctl(ATTR_TABLE_WRITE_DATA) failed");
                        return 0;
                }
        }
}

int write_pattern_table(int fd, int back_c){
        pattern_table_entry_t pattern = {0};
        pattern.sprite = 51;
        pattern.color_table = back_c;
        int i = 0;
        for(; i < 1200; ++i){
                pattern.id = i;
                if (i > 840 - 1 ) {
                        pattern.sprite = 49;
                        pattern.color_table = 4;
                }
                else if (i > 800 - 1 ) {
                        pattern.sprite = 50;
                        pattern.color_table = 4;
                }

                if (ioctl(fd, PATTERN_TABLE_WRITE_DATA, &pattern)) {
                        perror("ioctl(PATTERN_TABLE_WRITE_DATA) failed");
                        return 0;
                }
```

```
	}
}

int update_game_state_attrs(int fd, int num_bullets, int score, int round){


	attr_table[ROUND_ATTR_TABLE_OFFSET].sprite = NUMBER_SPRITE_OFFSET + round;
	if (ioctl(fd, ATTR_TABLE_WRITE_DATA, &attr_table[ROUND_ATTR_TABLE_OFFSET]))
{
		perror("ioctl(ATTR_TABLE_WRITE_DATA) failed");
		return 0;
	}

	for(int i = 0; i < NUM_BULLETS; ++i){
		if(num_bullets > i ){
			attr_table[BULLET_ATTR_TABLE_OFFSET + i].color_table =
SHADED_BULLET_COLOR_TABLE_OFFSET;
		}
		else {
			attr_table[BULLET_ATTR_TABLE_OFFSET + i].color_table =
UNSHADED_BULLET_COLOR_TABLE_OFFSET;
		}
		if (ioctl(fd, ATTR_TABLE_WRITE_DATA,
&attr_table[BULLET_ATTR_TABLE_OFFSET +i])) {
			perror("ioctl(ATTR_TABLE_WRITE_DATA) failed");
			return 0;
		}
	}

	// work from lowest significant digit to highest significant digit.
	for(int i = NUM_SCORE_DIGITS; i > 0; --i) {
		attr_table[SCORE_ATTR_TABLE_OFFSET + i - 1].sprite =
NUMBER_SPRITE_OFFSET + score % 10;
		score = score / 10;
		if (ioctl(fd, ATTR_TABLE_WRITE_DATA,
&attr_table[SCORE_ATTR_TABLE_OFFSET + i - 1])) {
			perror("ioctl(ATTR_TABLE_WRITE_DATA) failed");
			return 0;
		}
	}

	return 1;
}

int update_duck_attr(int fd, duck_t * duck) {

	int i = 0;
```

```
	for(; i < NUM_SPRITES_PER_DUCK; ++i){

		int attr_table_entry = DUCK_ATTR_TABLE_OFFSET + duck->id *
NUM_SPRITES_PER_DUCK + i;
		attr_table[attr_table_entry].coord.x = duck->coord.x +
SPRITE_TABLE_ENTRY_SIZE * (i / 2);
		attr_table[attr_table_entry].coord.y = duck->coord.y +
SPRITE_TABLE_ENTRY_SIZE * (i % 2) ;
		if (duck->velocity > 3) {
			attr_table[attr_table_entry].color_table  = 0 ;
		} else {
			attr_table[attr_table_entry].color_table  = 6 ;
		}
		if(duck->state == flap_up && duck->x_direction == east){
			attr_table[attr_table_entry].sprite  =
DUCK_UP_EAST_SPRITE_OFFSET  + i ;
		}
		if(duck->state == flap_up && duck->x_direction == west){
			attr_table[attr_table_entry].sprite  =
DUCK_UP_WEST_SPRITE_OFFSET  + i ;
		}
		if(duck->state == flap_down && duck->x_direction == east){
			attr_table[attr_table_entry].sprite  =
DUCK_DOWN_EAST_SPRITE_OFFSET  + i ;
		}
		if(duck->state == flap_down && duck->x_direction == west){
			attr_table[attr_table_entry].sprite  =
DUCK_DOWN_WEST_SPRITE_OFFSET  + i ;
		}
		if(duck->state == dead) {
			attr_table[attr_table_entry].sprite  = DUCK_DEAD_SPRITE_OFFSET
+ i ;
		}
		if(duck->state == flying_away) {
			attr_table[attr_table_entry].sprite  =
DUCK_FLYING_AWAY_SPRITE_OFFSET  + i ;
		}
		if (ioctl(fd, ATTR_TABLE_WRITE_DATA, &attr_table[attr_table_entry])) {
			perror("ioctl(ATTR_TABLE_WRITE_DATA) failed");
			return 0;
		}
	}


	return 1;
}
```

```c
int update_crosshair_attr(int fd, int x_coord, int y_coord) {
	int attr_table_entry = CROSSHAIR_ATTR_TABLE_OFFSET;
	attr_table[attr_table_entry].coord.x = x_coord;
	attr_table[attr_table_entry].coord.y = y_coord;
	attr_table[attr_table_entry].color_table = 0;

	if (ioctl(fd, ATTR_TABLE_WRITE_DATA, &attr_table[attr_table_entry])) {
		perror("ioctl(ATTR_TABLE_WRITE_DATA) failed");
		return 0;
	}
	return 1;
}
/*
 * Userspace program that communicates with the ppu device driver
 * through ioctls
 *
 */

#include <stdio.h>
#include "sprites.h"
#include "game/game.h"
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <time.h>


#define SECONDS_BETWEEN_SPAWNS 1

int duck_hunt_fd;

void play_game(){

	while(1){

		// Set inintial game state.
		game_config_t game_data = {
			.bullets = kBulletsPerRound,
			.score = 0,
			.round = 0,
			.spawned_ducks = 0,
			.visible_ducks = 0
		};
```

```
		// Set initial duck state. Value initialize array to all member
variables are zero initialized. There will only ever be ducksperround ducks on the
screen.
		duck_t ducks[NUM_DUCKS_PER_ROUND] = {};
		for(int i = 0; i < NUM_DUCKS_PER_ROUND; ++i){
			ducks[i].id = i;
			ducks[i].state = inactive;
		}

		coord_t cross_hair = { .x = 0, .y = 0};

		time_t last_spawned_time = time(0);
		printf("TIME %ld\n", last_spawned_time);

		while(!is_game_over(&game_data)){
			usleep(10000);

			// Introduce a duck every 5 seconds if there are fewer than 2
ducks on the screen.
			time_t now = time(0);
			if(now - last_spawned_time > SECONDS_BETWEEN_SPAWNS
				&& game_data.visible_ducks < NUM_DUCKS_PER_ROUND &&
game_data.spawned_ducks < NUM_DUCKS_PER_ROUND) {

				//printf("trying to spawn duck\n");
				last_spawned_time = now;
				// spawn the first duck that is currently not visible.
				for(int i = 0; i < NUM_DUCKS_PER_ROUND; ++i){
					if(ducks[i].state == inactive){
						spawn_duck(&ducks[i], &game_data);
						break;
					}
				}

			}
			move_ducks(ducks, NUM_DUCKS_PER_ROUND, &game_data);
			for(int i = 0; i < NUM_DUCKS_PER_ROUND; ++i){
				if(ducks[i].state != inactive){
					update_duck_attr(duck_hunt_fd, &ducks[i]);
				}
			}
			if(is_round_over(&game_data)){
				start_new_round(&game_data);
				update_game_state_attrs(duck_hunt_fd, game_data.bullets,
game_data.score, game_data.round);
				usleep(2000000);
			}
```

```
                    update_game_state_attrs(duck_hunt_fd, game_data.bullets,
game_data.score, game_data.round);
            }
            printf("GAME_OVER");

      }
}

int main()
{
      static const char filename[] = "/dev/ppu";

      printf("Duck Hunt userspace program started\n");

      if ( (duck_hunt_fd = open(filename, O_RDWR)) == -1) {
            fprintf(stderr, "could not open %s\n", filename);
            return -1;
      }

      write_sprite_table(duck_hunt_fd);
      write_color_table(duck_hunt_fd);
      write_sprite_attr_table(duck_hunt_fd);
      write_pattern_table(duck_hunt_fd, 4);

      play_game();

      printf("VGA BALL Userspace program terminating\n");
      return 0;
}
#include "game.h"
#include "stdio.h"
#include <math.h>
#include <stdlib.h>
#include <time.h>

const int kVerticalScreenSize = 300;
const int kHorizontalScreenSize = 640;
const double kPI = 3.14159;
const int kMaxDuckTimeSeconds = 6;
const int kRoundsPerGame = 6;
const int kCrossHairSquareSize = 40;
const int kGraphicSize = 32;
const int kBulletsPerRound = 3;

coord_t get_center_of_graphic(coord_t* top_left) {
      return (coord_t){.x = top_left->x + kGraphicSize/2, .y = top_left->y +
kGraphicSize/2};
```

```
}

int calculate_hit(duck_t * duck, coord_t cross_hair){
	// coord_t cross_hair_center = get_center_of_graphic(&cross_hair);
	coord_t cross_hair_center = (coord_t) {.x = cross_hair.x + 8, .y =
cross_hair.y + 8};
	coord_t duck_center = get_center_of_graphic(&duck->coord);

	return duck->state != inactive && duck->state != flying_away && duck->state
!= dead && cross_hair_center.x < (duck_center.x + kCrossHairSquareSize) &&
cross_hair_center.x > (duck_center.x - kCrossHairSquareSize)
		&& cross_hair_center.y < (duck_center.y + kCrossHairSquareSize) &&
cross_hair_center.y > (duck_center.y - kCrossHairSquareSize);
}

int move_ducks(duck_t* ducks, int num_ducks, game_config_t * game_config){
	for(int i = 0; i < num_ducks; ++i){
		move_duck(&ducks[i], game_config);
	}
}

int move_duck(duck_t * duck, game_config_t * game_config){

	if(duck->state == inactive) return 1;
	// A dead duck should not move at all on the x plane. It should drop down
where it was shot. Dropping in our coordinate system means adding to the y coord.
	if(duck->state == dead ){
		if(duck->coord.y < kVerticalScreenSize){
			++duck->coord.y;
		} else {
			printf("duck dead\n");
			duck->state = inactive;
			// --game_config->visible_ducks;
			printf("visible ducks:%d\n", game_config->visible_ducks);
		}
		return 1;
	}
	// A flying away duck should not move on the x plane. It should leave the
screen by flyinga directly upward.
	if(duck->state == flying_away ){
		if(duck->coord.y > 0 - kGraphicSize){
			duck->coord.y--;
		}
		else {
			printf("flew off screen\n");
			duck->state = inactive;
			--game_config->visible_ducks;
```

```
                printf("visible ducks:%d\n", game_config->visible_ducks);
            }
            return 1;
        }
        if(duck->coord.x <= 0 || (rand() % 256 == 0)) {
            duck->x_direction = east;
            duck->y_angle = rand() % 45;
        }
        // right edge of the graphic at the edge of the screen
        else if(duck->coord.x >= kHorizontalScreenSize - kGraphicSize - 1 || (rand()
% 256 == 0)){
            duck->x_direction = west;
            duck->y_angle = rand() % 45;
        }
        // at the top of the screen
        if(duck->coord.y <= 0 || (rand() % 256 == 0)) {
            duck->y_direction = south;
            duck->state = flap_down;
            duck->y_angle = rand() % 45;
        }
        else if(duck->coord.y >= kVerticalScreenSize - kGraphicSize - 1 || (rand() %
256 == 0)) {
            duck->y_direction = north;
            duck->y_angle = rand() % 45;
        }
        // Otherwise a duck should continue moving in the x and y direction it was
previously.
        if(duck->x_direction == east){
            duck->coord.x += ceil(duck->velocity * cos(duck->y_angle*kPI/180.0));
        }
        else if(duck->x_direction == west){
            duck->coord.x -= ceil(duck->velocity * cos(duck->y_angle*kPI/180.0));
        }
        if(duck->y_direction == north){
            // convert to radians and calculate the change in y given a change of
1 unit in the x direction.
            // calculations need to take place as doubles and then be ceiled so
they will make sense as ints.
            duck->coord.y -= ceil(duck->velocity * sin(duck->y_angle*kPI/180.0));
            //printf("angle %f\n", duck->y_angle);
            //printf("tan: %f\n", tan(duck->y_angle*kPI/180.0));
        }
        else if(duck->y_direction == south){
            // convert to radians and calculate the change in y given a change of
1 unit in the x direction.
            duck->coord.y += ceil(duck->velocity * sin(duck->y_angle*kPI/180.0));
        }
```

```
	if(time(0) - duck->spawn_time  > kMaxDuckTimeSeconds){
		duck->state = flying_away;
	}
	if(duck->y_direction == north){
		duck->state = (time(0) % 2) ? flap_up : flap_down;
	}
	return 1;
}

void kill_duck_update_config(duck_t * duck, game_config_t* config){
	duck->state = dead;
	config->score += duck->value;
	config->visible_ducks--;
}

int shoot_at_ducks(duck_t* ducks, int num_ducks, coord_t cross_hair, game_config_t*
config){

	for(int i = 0; i < num_ducks; i++){
		int hit = calculate_hit(&ducks[i], cross_hair);
		if(hit){
			kill_duck_update_config(&ducks[i], config);
		}
	}

	--config->bullets;

	return 1;
}

// The round is over only if we have seen all the allowed ducks per round
int is_round_over(game_config_t * config){
	if (config->bullets == 0)
		printf("ROUND OVER BECAUSE BULLETS \n");
	if (config->spawned_ducks == NUM_DUCKS_PER_ROUND && config->visible_ducks ==
0)
		printf("DUCKS DEAD FAULT\n");
	return (config->bullets == 0) || (config->spawned_ducks ==
NUM_DUCKS_PER_ROUND && config->visible_ducks == 0);
}

int is_game_over(game_config_t * config){
	return config->round == kRoundsPerGame + 1;
}

int start_new_round(game_config_t * config){
	++config->round;
```

```
        config->bullets = kBulletsPerRound;
        config->visible_ducks = 0;
        config->spawned_ducks = 0;
}

int spawn_duck(duck_t * duck, game_config_t * config){

        duck->coord.x = 100 + rand() % 300;
        duck->coord.y = kVerticalScreenSize;
        duck->spawn_time = time(0);
        duck->value = 1 + config->round/2;
        duck->state = flap_up;
        duck->velocity = 2 + config->round/2;
        // Randomly pick whether the duck starts moving east or west.
        duck->x_direction = rand() % 2;
        // Duck always starts moving upward since it is coming out of the grass.
        duck->y_direction = north;

        // config->visible_ducks++;
        // config->spawned_ducks++;

}
/* This module contains all of the game specific logic for Duck Hunt.
*/

#ifndef _GAME_H
#define _GAME_H
#include <time.h>


// Constants exposed for testing purposes.
extern const int kHorizontalScreenSize;
// how much vertical room the ducks have to move around ( how much sky there is on
the screen ).
extern const int kVerticalScreenSize;
// Used to caluclate the bounding box for a cross hair. Essentially, how much
// leeway to give users in the x and way directions when computing if they
successfully shot a duck.
extern const int kCrossHairSquareSize;
// How big the ducks and the crosshair are.
extern const int kGraphicSize;
// Number of moves a duck can make on screen before it flies away.
extern const int kMaxNumDuckMoves;
// How many ducks to give the player an opportunity to shoot before we end the
game.

#define NUM_DUCKS_PER_ROUND 2
```

```
extern const int kBulletsPerRound;

enum duck_state { flap_down, flap_up,  dead, flying_away, inactive };
// east and west denote movement on the x plane. north and south denote movement
// on the y plane.
enum direction { east, west, north, south };

typedef struct {
	// signed so that we can go negative when sprite is partially on the screen.
	int x,y;
} coord_t;

typedef struct {
	// How many points the duck is worth in the game.
	int value;
	// Where the duck currently is on the screen.
	coord_t coord;
	// Direction on x plane.
	enum direction x_direction;
	// Direction on y plane.
	enum direction y_direction;
	// Angle in degrees that ducks will move at an angle on the y plane to make
game_play more interesting.
	// an angle of 45 degrees denotes 1 unit of movement in the y plane for
every one unit of movement in the x plane.
	double y_angle;
	// Velocity multiplier
	double velocity;
	// unique_id associated with each duck.
	int id;
	enum duck_state state;
	// when the duck was created.
	time_t spawn_time;
} duck_t;

typedef struct {
	unsigned char bullets, score, round;
	int spawned_ducks;
	int visible_ducks;
} game_config_t;


// Checks to see if a given shot has hit any of the ducks.
int shoot_at_ducks(duck_t* ducks, int num_ducks, coord_t cross_hair, game_config_t*
config);
```

```
// Caclulates hit square associated with x,y position of the duck. Returns 1 if
cross_hair is in the hit square. Returns 0 if cross_hair is outside of the hit
square.
int calculate_hit(duck_t * duck, coord_t cross_hair);

// Deducts from bullet count, adds ducks value to the total score.
void kill_duck_update_config(duck_t * duck, game_config_t* config);

// Moves a single duck across the x plane. (y plane if the duck is dead).
Increments the num_ducks_seen in the game
// config if the duck enters the flying away state.
int move_duck(duck_t * duck, game_config_t * game_config);

// moves all of the ducks.
int move_ducks(duck_t* ducks, int num_ducks, game_config_t * game_config);

int is_round_over(game_config_t * config);
int start_new_round(game_config_t * config);
// game is over if we are out of bullets or if we've seen a set number of ducks.
int is_game_over(game_config_t * config);
int spawn_duck(duck_t * duck, game_config_t * config);




#endif
#include <stdio.h>
#include <assert.h>
#include "game.h"

int test_calculate_hit(){

	duck_t duck = {
		.coord = {
			.x = 10,
			.y = 16
		}
	};
	coord_t cross_hair = { .x = 0, .y = 0};

	// x and y are dead on
	cross_hair.x = 10;
	cross_hair.y = 16;
	assert(calculate_hit(&duck, cross_hair) == 1);

	// y is in range but x is out of range in either direction
	cross_hair.x = duck.coord.x + kCrossHairSquareSize;
```

```
        assert(calculate_hit(&duck, cross_hair) == 0);

        cross_hair.x = duck.coord.x - kCrossHairSquareSize;
        assert(calculate_hit(&duck, cross_hair) == 0);

        // x is just in range in either direction. Should hit.
        cross_hair.x = duck.coord.x + kCrossHairSquareSize -1;
        assert(calculate_hit(&duck, cross_hair) == 1);

        cross_hair.x = duck.coord.x - kCrossHairSquareSize +1;
        assert(calculate_hit(&duck, cross_hair) == 1);

        // x is in range but y is out of range.
        cross_hair.x = 10;
        cross_hair.y = duck.coord.y + kCrossHairSquareSize;
        assert(calculate_hit(&duck, cross_hair) == 0);

        cross_hair.y = duck.coord.y - kCrossHairSquareSize;
        assert(calculate_hit(&duck, cross_hair) == 0);

        // y is just in range in either direction. should hit
        cross_hair.y = duck.coord.y + kCrossHairSquareSize -1;
        assert(calculate_hit(&duck, cross_hair) == 1);

        cross_hair.y = duck.coord.y - kCrossHairSquareSize + 1;
        assert(calculate_hit(&duck, cross_hair) == 1);

        // both x and y are wildly out of range
        cross_hair.x = 100;
        cross_hair.y = 100;
        assert(calculate_hit(&duck, cross_hair) == 0);

        // ducks that are dead or flying away cannot be hit.
        duck.state = flying_away;
        assert(calculate_hit(&duck, cross_hair) == 0);

        duck.state = dead;
        assert(calculate_hit(&duck, cross_hair) == 0);

}

void test_move_duck(){

        duck_t duck = {
                .coord = { .x = 200, .y = 200},
                .x_direction = east,
                .y_direction = north,
```

```
        .y_angle = 45
    };

    game_config_t config = {
        .num_ducks_seen = 0
    };
    // Basic test. assert that moving the duck starting at 0, 0 with a 45 degree
angle moves the duck 1 unit in the
    // x direction and 1 unit in the y direction.
    int success = move_duck(&duck, &config);
    assert(success == 1);
    assert(duck.coord.x == 201);
    assert(duck.x_direction == east);
    //printf("%d\n", duck.coord.y);
    assert(duck.coord.y == 199);
    assert(duck.y_direction == north);

    duck.coord.x = 1;
    duck.y_angle = 0;

    // Should move duck all the way to the edge of the screen.
    // Starting at 1 since 0 is a special case.
    for(int i = 1; i < kHorizontalScreenSize - kGraphicSize - 1; ++i){
        assert(duck.coord.x == i);
        int success = move_duck(&duck, &config);
        assert(success == 1);
        assert(duck.coord.x == i + 1 );
        assert(duck.coord.y == 199);
        assert(duck.x_direction == east);
    }

    assert(duck.coord.x == kHorizontalScreenSize - kGraphicSize - 1);

    duck.coord.x = kHorizontalScreenSize - kGraphicSize;
    for(int i = duck.coord.x ; i > 0; --i){
        assert(duck.coord.x == i);
        int success = move_duck(&duck, &config);
        assert(success == 1);
        assert(duck.coord.x == i-1);
        assert(duck.x_direction == west);
    }

    assert(duck.coord.x == 0);

    success = move_duck(&duck, &config);
    assert(success == 1);
    assert(duck.coord.x == 1);
```

```
	assert(duck.x_direction == east);

	// dead ducks don't move on the x plane and fall to the ground.
	duck_t dead_duck;
	dead_duck.state = dead;
	dead_duck.coord.x = 52;
	dead_duck.coord.y = 20;
	for(int i = 0; i < 300; i++){
		move_duck(&dead_duck, &config);
	}

	printf("%d\n", dead_duck.coord.x);
	printf("%d\n", dead_duck.coord.y);
	assert(dead_duck.coord.x == 52);
	assert(dead_duck.coord.y == kVerticalScreenSize + kGraphicSize );

	duck_t flies_away = {
		.coord = {
			.x = 200,
			.y = 200
		},
	};

	for(int i = 0; i <= kMaxNumDuckMoves; ++i){
		move_duck(&flies_away, &config);
	}

	assert(flies_away.state == flying_away);
	assert(config.num_ducks_seen == 1);
}


void test_kill_duck_update_score(){
	game_config_t config = {

		.bullets = 3,
		.score = 0,
		.round = 0
	};

	duck_t ducks[2] = {};
	ducks[0].coord.x = 5;
	ducks[0].coord.y = 10;
	ducks[0].value = 5;

	ducks[1].coord.x = 200;
	ducks[1].coord.y = 200;
```

```
        ducks[1].value = 5;

        // We should hit duck one
        coord_t cross_hair;
        cross_hair.x = 5;
        cross_hair.y = 10;

        shoot_at_ducks(ducks, 2, cross_hair, &config);
        assert(config.score == 5);
        assert(ducks[0].state == dead);
        assert(ducks[1].state !=dead);
        assert(config.bullets == 2);
}

void test_game_over(){
        game_config_t config = {
        .bullets = 3,
        .score = 0,
        .round = 0,
        .num_ducks_seen = 0
        };

        assert(is_game_over(&config) == 0);
        config.bullets = 0;
        assert(is_game_over(&config) == 1);
        config.bullets = 3;
        config.num_ducks_seen = kMaxDucksPerGame;
        assert(is_game_over(&config) == 1);
}


int main() {
        test_calculate_hit();
        test_move_duck();
        test_kill_duck_update_score();
        test_game_over();
}
```

SCRIPTS

```
from PIL import Image
from collections import defaultdict
from jinja2 import Environment
from jinja2 import PackageLoader

# Format:
# image_name : [tl x, tl y, br x, br y]
SPRITES = {
        "DUCK_RIGHT_1" : [0, 120, 35, 145],
}

if __name__ == "__main__":
    im = Image.open('media/duckhunt_various_sheet.png').convert('RGBA')
    pix = im.load()

    width, height = im.size


    print(im.size)
    print(pix[96,88])
    print(im.mode)
    loader      = PackageLoader("scripts", "templates")
    env         = Environment(loader=loader)
    sprite_temp    = env.get_template('sprite.j2')
    color_temp     = env.get_template('colormap.j2')

    for sprite, coords in SPRITES.items():
        tx, ty, bx, by = coords[0], coords[1], coords[2], coords[3]

        colorcount = defaultdict(lambda: 0)
        for i in range(tx, bx):
            for j in range(ty, by):
                colorcount[pix[i,j]] += 1

        max_colors = sorted([(v, k) for (k, v) in
colorcount.items()])[::-1]
        print(colorcount)
```

```
        print(max_colors)


        out_colors = []
        cm_ind = 1
        for cnt, col in max_colors:
            if col[3] == 0:
                colorcount[col] = 0
            else:
                if cm_ind <= 3:
                    out_colors.append(col)
                colorcount[col] = min(cm_ind, 3)
                cm_ind += 1

        print(out_colors)

        # Write out color map
        variables   = dict(id = 1, colors = out_colors)
        output_from_parsed_template = color_temp.render(**variables)
        print(output_from_parsed_template)

        for i in range(tx, bx, 16):
            for j in range(ty, by, 16):
                # Generate hex numbers
                nums = ['0x0' for _ in range(16)]
                for q in range(j, min(j + 16, by)):
                    num = 0
                    for p in range(i, min(i + 16, bx)):
                        # num = num * 4 + colorcount[pix[p,q]]
                        num ^= (colorcount[pix[p,q]] << (2 * (p - i)))
                    nums[q - j] = str(hex(num))
                    # nums[q - j] = str(hex(rev))

                print(nums)

                variables   = dict(id = 1, lines = nums)
                output_from_parsed_template =
sprite_temp.render(**variables)
                print(output_from_parsed_template)
```