Monkey Madness Project Report

Kyle Edwards (kje2115), William Freedman (wrf2107), Sadie Freisthler (srf2156),
Madeline Skeel (mgs2189), Jake Torres (jrt2170)

May 14, 2025

1 Overview

This project is a simplified recreation of Marble Madness, an NES game that utilizes a trackball as its main input source. The goal of the player is to move a marble through a maze while avoiding obstacles. A fun aspect of this game is that it utilizes the full range of motion of the trackball, pictured in Figure 1, without requiring 3D graphics. Marble Madness does this by representing the game map isometrically, creating the perception of a 3D field. See an example level in Figure 2. In order to accurately represent the motion of the ball, isometric projection is required to convert from the 3D physics game map to the 2D screen coordinates. This projection creates an interesting twist on top of traditional video game rendering of this time.

In our implementation, the graphics rendering is handled by the FPGA. The position of the marble and the tiles to display are communicated to the hardware through the device driver. The marble physics is handled by software and intended to function completely independently of the hardware. More in depth technical implementation details are described below.

2 Technical Implementation

2.1 Trackball Peripheral

The user will provide input to the game through a Kensington Orbit Optical Trackball mouse, which provides a 360 trackball, as well as two buttons. The mouse connects to the FPGA via a USB port, and its input can be read through the /dev/input/event0 file. Inputs are sent and read in the form of the input event struct, which allows our userspace program to read the mouses input. Since we are only using the mouse for directional input, we only store packets where the .type field is set to EV REL, in which case we read the .value field, which will either be an x or y offset, depending on the value of ev.code (REL X or REL Y, respectively).

Since the trackball will be sending events far more often than frames will be generated, we will have a thread constantly reading and processing input events. This thread will store and update

Figure 1: Trackball peripheral

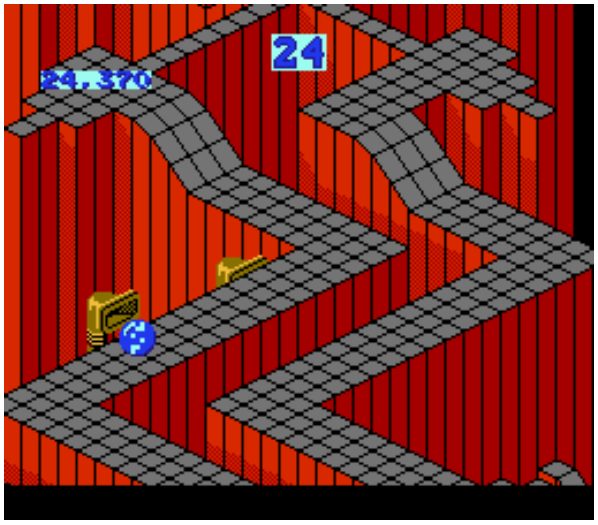


Figure 2: Example Marble Madness Level

net dx and net dy values, corresponding to the accumulated change in the balls position since dx and dy were last read by the main thread. This thread will also use appropriate locking to ensure that the main thread can read dx and dy correctly. The main trackball logic is defined in `trackball.c`.

`setupReader()`
This function is called at the start of each level to perform any necessary setup for the trackball-reading thread (discussed below).

`readTrackball()`
This function returns a struct storing the net dx and dy input by the trackball since the last time it was called. These values are then used to calculate the impulse values in the main game loop.

2.2 Software Physics Engine

Our software physics engine creates a 3D model of the game map. We simulate how the marble moves through and interacts with this 3D space to inform where the marble is rendered in hardware. The main obstacles we have to handle are walls, ramps, and cliffs. More in depth discussion of the different files and functions is outlined below.

main.c
The central game loop is located in `main.c`. First, the levels are loaded into memory using a function called `initialize_levels()`. This function takes in the current marble state so that the initial position can be updated based on the loaded game map. Next, we initialize the vga marble device driver as well as the trackball by calling `setupReader()`. After these initialization steps, we enter an infinite while loop that runs until game termination.

In this loop, we first poll the trackball to get the impulse vector by calling `getAccumulatedInput()` this impulse is parsed into a 3D vector and used to update the position and velocity of the ball using `apply_impuse()`.

With this updated position, we check boundary conditions using `check_boundaries()` to check if we need to handle any wall, fall, or ramp conditions. This function also returns macros `WON`, `LOST`, or `CONTINUE` to communicate the current game state. If `WON` or `LOST` states are returned we handle them accordingly. Otherwise, we continue looping.

types.h
This file defines the main types used in our program. The `Tile` struct contains the x, y, and z indices and the tile type. The macros for different tile types are defined in `game_map.h`. Every level is stored in memory as a 2D array of `Tile` structs. There is also a `MarbleState3D` struct that stores 3D vectors

for position and velocity. 3D and 2D vector structs are defined in this file as well.

physics.c physics.h
`physics.c` contains the core of the software logic. `apply_impulse()` applies the impulse polled from the trackball and updates the position and velocity of the global `MarbleState3D` struct. The `check_boundaries()` function computes the current tile based on the marbles 3D coordinates. It then checks if the marble is on an empty tile or on the win tile by calling `check_game_state()`. If we are not in a win or lose state, we check if the marble is colliding with a wall, rolling down a ramp, or falling off a ledge and handle that accordingly. These functions are called `check_wall()`, `check_fall()`, `handle_wall()`, `handle_fall()`, `check_ramp()`, `handle_ramp()`.

`check_wall()` checks whether the marble is about to encounter a wall based on its current tile and velocity. If the tile the marble is about to move to has a lower z value, there is a wall and the function returns the side of the current tile that the wall is on. `check_fall()` also checks the height of the tile the marble is about to encounter based on the marble's position and velocity. If the z value of the neighboring tile is higher than the current tile, then the ball should fall, so the function returns 1. Otherwise it returns 0.

`handle_wall()` reverses the marbles velocity in the x direction if the wall is on the left or the right sides of the tile, and reverses the marbles velocity in the y direction if the wall is on the top or bottom sides of the tile. `handle_fall()` applies a gravitational force to the marble in the z direction. This only occurs in the case where the marble is falling onto another lower part of the map rather than falling off the map. In the case where the marble is falling off the map, the function returns LOST.

`check_ramp()` returns the type of ramp if the marble is on a ramp tile. Otherwise it returns 0. `handle_ramp()` checks the direction of the ramp and applies a gravitational force in the corresponding direction.

game_map.c game_map.h
`game_map.c` defines the `initialize_levels()` and `check_position()` functions. `check_position()` returns the `Tile` struct associated with the marbles x,y, and z coordinates. `game_map.h` defines all of the different tile types and global variables for levels and `current_level`.

level_reader.c level_reader.h
The `read_level()function` reads in csv data from preconstructed levels. Each csv square has two integers. The first integer is the tile height and the second defines the tile type. This function reads in this information, converts it to `Tile` structs and stores them in memory in a 2D array. The starting position is also set in this function. A `Tile**` pointer to this array is returned.

2.3 Software Graphics Rendering

We had to use the art of illusion in order to draw a 3D looking game map in 2D space. Starting from the 3D game map used for physics modeling we isometrically project these coordinates into 2D space to be displayed on screen. See the derivation for this projection in Figure 3. The equations for the 3D-2D isometric projection are defined below.

$$x = 2x' + 2y'$$
$$y = y' - x' + 2z'$$

In order to obtain the tile set to render levels in hardware we first had to create a software rendering. This software rendering is then subdivided into a grid of 8x8 tiles to be used as the tile set.
First, we created a CSV file that represents a top-down view of the level, with every tile assigned a height and tile type. We then used Python to read the CSV and parse it into a matrix of pairs, where each index had a Z height and a 16x16 tile to display. We iterate through the matrix and draw the associated subdivided isometric tile based on the integer tile type. The images were drawn to the screen using the Python library Pygame. Pygame allowed us to load in images, create a display the

same size as our screen that we will be using, and draw the images to the screen at a display position x and y.
In Figure 4 we show the algorithm we used to parse the matrix and the order in which the tiles are rendered. To ensure the appropriate layer ordering, the matrix must be parsed diagonally and from back to front. Figure 4 also has arrows to illustrate this parsing. In addition, in our program, we draw pillars from each tile to the bottom of the screen. In order to do this, we check if we are drawing a tile at a given index in the matrix, if so, from the bottom of the screen up, we draw pillar tiles until we are at the same level as our currently tile we want to draw. See an example-level CSV file and associated software rendering in Figures 5 and 6 respectively.
After using our software representation to render our levels, we then subdivided the screen into 8x8 tiles that will be used in rendering the final level. We took each 8x8 tile and added all of the colors used to a set of colors - this is our color palette - and created a new 8x8 matrix of offsets into our color palette. We then added each of these 8x8 tiles to the set of textures used in the levels and created a tilemap that stores offsets into this texture map for tiles at each position on the screen. We created a visualization of how all this is all connected in Figure 8. After creating the final tilemap, texture set and palette color set, we wrote them to binary files with 1 byte headers indicating the length of the two sets, or 2 bytes for the tilemap which has the width and height followed by the data associated with each structure. These binary files are later read by the games main function at boot and immediately written into hardware.

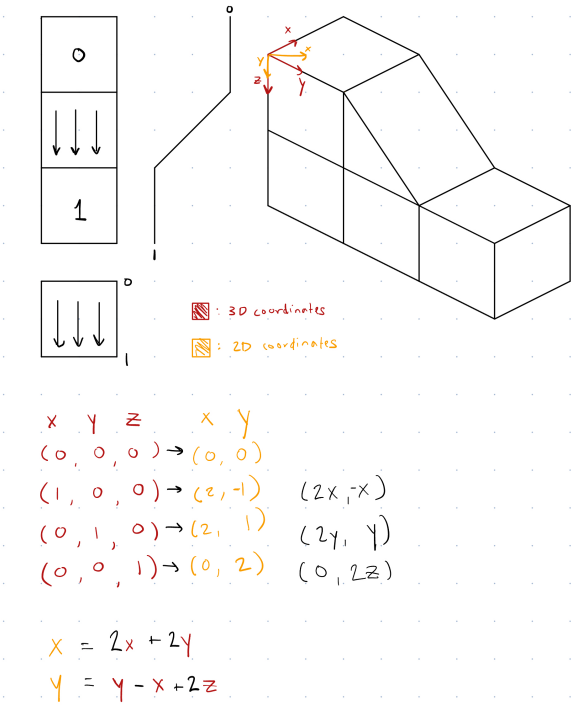


Figure 3: Isometric Projection Derivation

2.4 Device Driver

Our device driver handles the communication between software and hardware as a kernel module, similarly to Lab 3. Like in Lab 3, our kernel module has an ioctl function allowing userspace programs to write to our hardware memory. Using our memory regions and offsets we will later discuss in the Hardware section, we have multiple write functions that userspace functions can call: VGA_GPU_WRITE_TEXTURE, VGA_GPU_WRITE_TILE, VGA_GPU_WRITE_SPRITE, VGA_GPU_WRITE_PALETTE, and VGA_GPU_WRITE_CTRL.

VGA_GPU_WRITE_TEXTURE allows users to write a texture to an index in the texture memory

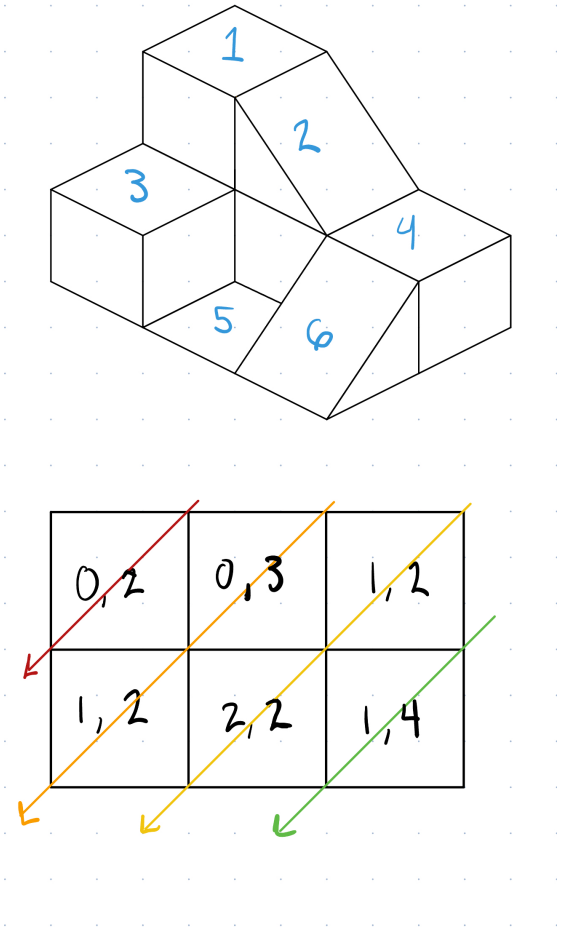


Figure 4: Software Rendering Algorithm

0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
0,0	0,0	0,0	0,0	0,2	0,3	1,2	0,0	0,0	0,0	0,0
0,0	0,0	0,0	0,0	1,2	2,2	1,4	0,0	0,0	0,0	0,0
0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0

Figure 5: Example Level CSV

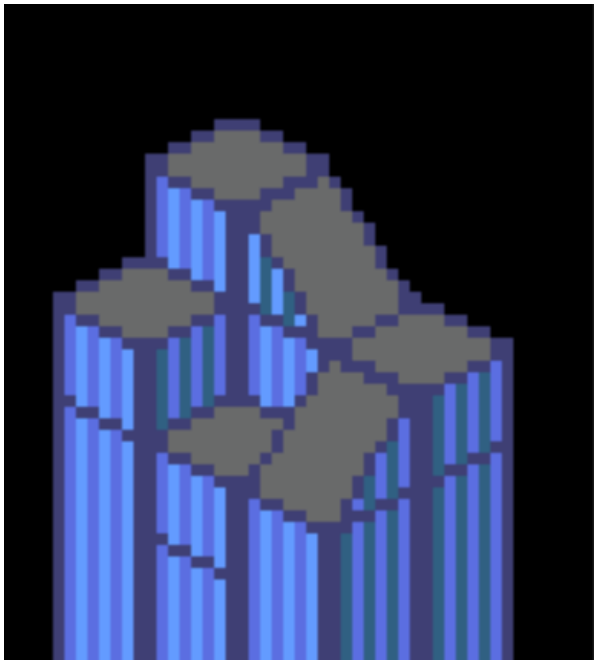


Figure 6: Example Level Software Rendering

region with an matrix 8x8 of offsets into a palette.

VGA_GPU_WRITE_TILE allows users to write a tile to an x and y position in the tilemap region of memory. We additionally have a layer field allowing support for multiple layers of tilemaps if we want to use that. Each tile has an 10 bit index into the texture region for what texture to use for the tile. It additionally has a index into the palette region of memory for what palette to use. It lastly has optional bits for priority over sprites, horizontal flipping, or vertical flipping.

VGA_GPU_WRITE_SPRITE allows users to write a sprite to an x and y position in the sprites region of memory. It behaves similarly to writing tiles and has all of the same fields.

VGA_GPU_WRITE_PALETTE allows the user to write a color to a color palette at an index in the palette region of memory at an offset into the palette. Each palette can store up to 16 colors and there are 8 available palettes in memory. The user must pass in r,g,b and a values for the color.

VGA_GPU_WRITE_CTRL allows the user to change the overall behavior of the GPU. As of now, the only option we have is to enable and disable blanking, though in the future we may add more.

2.5 Hardware

The hardware's job is to use the data stored in its VRAM to output a valid VGA signal. This is done with four main components: a large piece of VRAM, two components that are used to fetch palette information from VRAM for sprites and tiles respectively, and a compositor that takes the palette information given to it by the sprite and tile components and outputs a VGA signal. All of these components are controlled by a memory-mapped Avalon agent, which is how software eventually communicates with hardware.

VRAM itself is split up into multiple different smaller sections of RAM, each of which being dual port and byte-enabled. This allows the other components to read from VRAM while the agent performs operations of its own, which is especially useful for accessing VRAM without having to wait blanking. Furthermore, each section of VRAM is given its own separate line, allowing texture, tile, sprite, and palette data to be read in a single clock cycle, thus allowing pipelining.

The sprite and tile components vary internally, but overall perform the same pipelined operations in

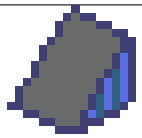
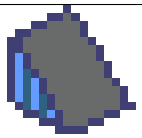
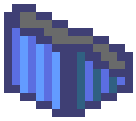
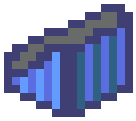
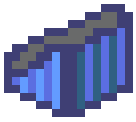
Name	Graphic	Size (bits)	csv value
Banana Ball		8x8	N/A
Floor Tile		16x16	2
Ramp Left Tile		16x16	4
Ramp Right Tile		16x16	3
Back Ramp Right Tile		16x16	6
Back Ramp Left Tile		16x16	7
Win Tile		16x16	8

Figure 7: Software Rendering Tiles - Note: These tiles are further subdivded into 8x8 tiles that are actually drawn to the screen.

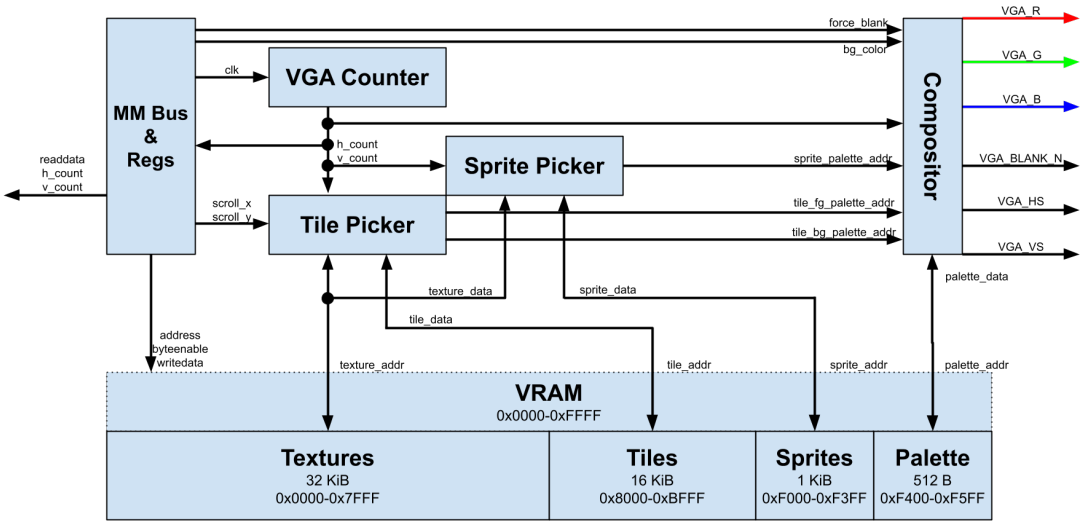


Figure 8: General hardware architecture

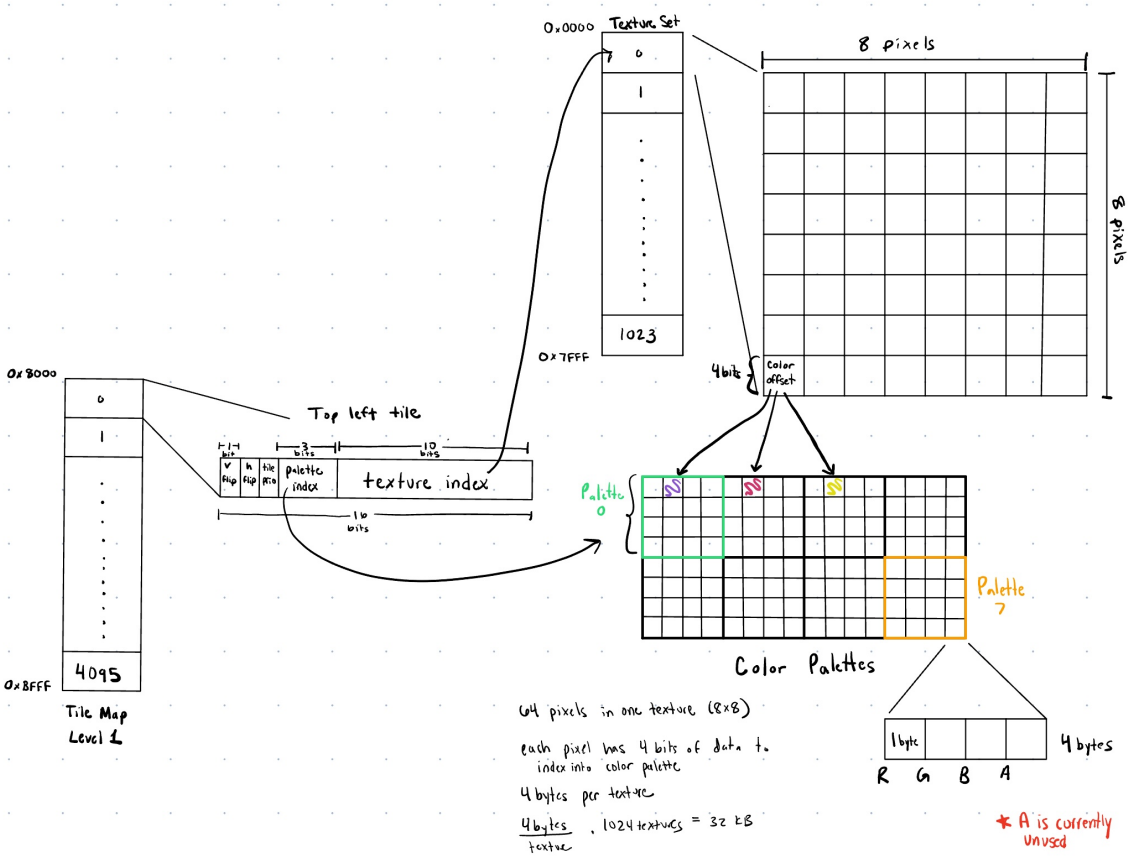


Figure 9: Memory Diagram

the following order: fetch a descriptor based on the current values of the VGA counters, fetch texture data using that descriptor, and finally write the texture data to an internal pixel buffer. Sprites and tiles differ during the last operation: The sprite component has a line buffer that takes up the entire screen, while the tile component has a much smaller buffer that can only store 16 pixels ahead. Both components are Mealy-style.

The compositor is the simplest of the four components, being a simple Mealy-style picker that reads palette data based on the palette addresses given to it by the previous two components. More specifically, the compositor is responsible for making sure that pixels are drawn in the correct order (e.g. ensuring that, if the current sprite pixel is not transparent, it is drawn instead of the current background pixel).

3 Contributions and Lessons Learned

3.1 Division of Labor

We split up the work of our project evenly. Kyle worked on the hardware and device driver, Jake worked on the artwork, software representation, and device driver, Madeline, Sadie and William worked on the game logic, physics system, and trackball interface. A lot of the project was a collaborative process with group members working together on different parts, especially during the design stages of the project and end, through pair programming or group discussion such as the device driver, game logic and physics.

Each of the parts was worked on concurrently and combined towards the end of our time working on the project.

3.2 Jake's Takeaways

Throughout the project I got to work on a lot of different aspects of the game, from coming up with some of the initial design ideas, to isometrically rendering the game in software and then working on the hardware-software interface with the device driver. I am especially glad I got to work on the device driver as I had felt shakey about my understanding of this after completing Lab 3 but I can confidently say I understand how the device communication works now and how the interface functions.

Additionally, I got to have some fun working with Pygame, a Python library I had not used prior to working on this project, which I used for another project concurrently because I was so excited about it. Also utilizing some of the concepts I learned in taking and TAing Operating Systems while working on the project was a good throwback and once again proved how useful these classes are.

It was really awesome seeing everything come together in the end. Some takeaways I have is the importance of everyone having a good understanding of how the hardware works and how everything is stored in memory. Knowledge about the hardware proved very important as having discussions about the device driver with other group members while I was working on it was difficult without a collective understanding of it. Another difficulty I experienced was drawing the artwork and rendering ramps facing away from the camera in an isometric view. We realized why there was a limited use of these in the Marble Madness game.

My last takeaway is the complexity of rendering an isometric view without tools that simplify the process. As a game developer that uses the Godot Engine, Godot has made rendering of tiles insanely easy with tilemap tools built-in to the engine allow for a variety of tilesets, including isometric tilesets. Manually rendering the isometric tiles proved to be a much more difficult challenge that provided a lot of learning opportunities!

3.3 Madeline's Takeaways

This project was a valuable learning experience in hardware design, hardware software interfacing, and group work. I thought it was really interesting to learn about how video game consoles like the

NES worked and how games were built under constraints.I also enjoyed learning about how isometric projects can be used to create the illusion of a 3-dimensional game.

One takeaway I had was in the division of work that we used. In some past projects Ive worked on, everyone has worked on all parts of the project. That has its benefits, but I really saw the value of a divide and conquer approach here. It allowed us to move faster and work on the parts we are most interested in. It also made me realize how important communication and documentation are.

Another part of the project I enjoyed was learning about tile based systems for computer graphics. As a computer graphics TA, I have spent a lot of time learning about ray tracing and pipeline rendering techniques, so this was an interesting new way of thinking about how graphics can be displayed.

3.4 Sadie's Takeaways

I had a lot of fun working on this project! It was a great exercise in planning, implementing, and integrating a larger scale project which I don't often get to do in academic settings. I learned a lot about isometric projection and graphics tile rendering that I had not been exposed to earlier. I enjoyed working on the physics system and modeling a 3D "world" that our user gets to interact with. My experience as an AP TA and with writing C code was crucial to quickly building the physics system.

There are a lot of lessons to take away from this project. I think most importantly, the value of having a detailed plan early on. This way everyone on the team is on the same page. I think we also learned the importance of scope as we had to scale back our project to better fit within the bounds of project timeline and group expertise.

3.5 William's Takeaways

I really enjoyed this project. It required working with a ton of distinct but interconnected components, all of which we needed to have at least a working understanding of to maximize our productivity. This kind of scale and integration, especially when starting from a blank project, is a challenge I haven't had a lot of practice in, so this was a fun challenge.

I predominantly worked on the software and physics side of the project, including the game logic and physics engine itself, as well as the conversion process from a level csv to an actual software representation. I also wrote the code that we used to interface with the trackball, which ended up being much simpler than we initially thought it would be, since our software was able to just read from /dev/event0.

The challenge in this kind of project is that it seems (at least to our group) most efficient to divide it up by person, with dedicated people working on hardware, software, device drivers, hardware, etc. This means I wasn't able to work in depth on our hardware and rendering, which would have been interesting, but challenging. However, I still think the way we divided up the project was the right call.

3.6 Kyle's Takeaways

This project was a lot of fun, but I wish I had spent more time perfecting things. The section of the project that I focused most on, the hardware, was more ambitious than the example code given to us in class, and as such I spent extra time with re-discovering many things. Most surprisingly, most of my time was spent on design decisions rather than on actual coding, a fact largely due to using Verilator and software VGA simulation. We have attached the script that we used for this in 'gpu.cpp', which hopefully other groups can use in the future.

With hardware, the main challenge I had with some of the design was figuring out the exact capabilities of the FPGA. Initially, I treated the FPGA as having a similar level of power as the Nintendo Entertainment System, but upon further examination,, it became evident that it was much closer to the Super Nintendo Entertainment System in terms of hardware capabilities, if not surpassing it. Not only did this make programming a bit more complex, it also meant that I had to spend time learning about the internals of the SNES (and other consoles with similar hardware, such as the Neo Geo and some Atari arcade cabinets), which likely doubled the total development time. On one hand, I am happy that we were able to implement more ambitious things such as scrolling and layered backgrounds. On the other hand, had we stuck to the NES, a console that although underpowered I

understood inside and out, it may have let us focus more on the software side. Regardless, I think the scope we ended up landing on hardware-wise was perfect in its overall level of complexity.

4 Files

Hardware

gpu.sv

```
/*
 * Avalon memory-mapped peripheral that generates VGA
 */

module gpu #(
    parameter int DATA_WIDTH = 32,
    parameter int ADDR_WIDTH = 14
) (
    input logic clk,
    input logic reset,
    input chipselect,
    input logic [DATA_WIDTH-1:0] writedata, // Write at most 32 bits at once
    input logic read,
    input logic write,
    input logic [ADDR_WIDTH-1:0] address,
    input logic [3:0] byteenable,

    output logic [DATA_WIDTH-1:0] readdata,

    output logic [7:0] VGA_R,
    VGA_G,
    VGA_B,
    output logic VGA_CLK,
    VGA_HS,
    VGA_VS,
    VGA_BLANK_N,
    VGA_SYNC_N
);

  /* VGA Controller setup */
  // Input from the VGA counter
  logic [9:0] h_count, v_count;
  // The actual location on the screen
  // logic [8:0] screen_x, screen_y;

  // next_next during attribute fetch, next during texture fetch
  // logic next_next_hs, next_hs;
  // logic next_next_vs, next_vs;
  // logic next_next_blank_n, next_blank_n;

  // For both, divide by 2 (VGA is 640x480, our resolution is 320x240)
  // assign screen_x = h_count[9:1];
  // assign screen_y = v_count[9:1];

  localparam logic [9:0] HACTIVE = 10'd640,
                         HFRONT_PORCH = 10'd16,
                         HSYNC = 10'd96,
                         HBACK_PORCH = 10'd48,
                         VACTIVE = 10'd480,
                         VFRONT_PORCH = 10'd10,
                         VSYNC = 10'd2,
                         VBACK_PORCH = 10'd33;
```

```
53
54      // localparam logic [9:0] HACTIVE = 10'd1,
55      // HFRONT_PORCH = 10'd1,
56      // HSYNC = 10'd1,
57      // HBACK_PORCH = 10'd1,
58      // VACTIVE = 10'd1,
59      // VFRONT_PORCH = 10'd1,
60      // VSYNC = 10'd1,
61      // VBACK_PORCH = 10'd1;
62      // localparam int SCREEN_WIDTH = HACTIVE >> 1, SCREEN_HEIGHT = VACTIVE >> 1;
63
64      localparam logic [9:0] HTOTAL = HACTIVE + HFRONT_PORCH + HSYNC + HBACK_PORCH,
65                             VTOTAL = VACTIVE + VFRONT_PORCH + VSYNC + VBACK_PORCH;
66
67      // force_blanking should be in registers, but it makes more sense to keep it
68      // here...
69      logic blank_n, force_blanking;
70
71      vga_counter #(
72          .HACTIVE(HACTIVE),
73          .HFRONT_PORCH(HFRONT_PORCH),
74          .HSYNC(HSYNC),
75          .HBACK_PORCH(HBACK_PORCH),
76          .VACTIVE(VACTIVE),
77          .VFRONT_PORCH(VFRONT_PORCH),
78          .VSYNC(VSYNC),
79          .VBACK_PORCH(VBACK_PORCH)
80      ) counter (
81          .clk50 (clk),
82          .reset (reset),
83          .h_count(h_count),
84          .v_count(v_count),
85          .vga_clk(VGA_CLK),
86          .h_sync (VGA_HS),
87          .v_sync (VGA_VS),
88          .blank_n(blank_n)
89      );
90
91      // TODO: Should we use this in the future?
92      assign VGA_SYNC_N = 1'b0;
93      assign VGA_BLANK_N = blank_n & ~force_blanking;
94
95      /* VRAM */
96
97      logic [31:0] vram_dout;
98
99      // FIXME uncomment
100     logic [12:0] tile_texture_addr, sprite_texture_addr, texture_addr; // 32 KiB == 13+2 bit address
101     logic [31:0] texture_data;
102
103     logic [12:0] tile_addr; // 16 KiB == 12+2 bit address
104     logic [15:0] tile_data;
105
106     logic [ 7:0] sprite_addr; // 1 KiB == 8+2 bit address
107     logic [31:0] sprite_data;
108
109     logic [6:0]
110         fg_palette_addr,
111         bg_palette_addr,
112         sprite_palette_addr,
113         palette_addr; // 512 B == 7+2 bit address
114     logic [ 3:0] palette_off;
```

```
115     logic [31:0] palette_data;
116
117     logic [7:0] pixel_r, pixel_g, pixel_b;
118
119     /* verilator lint_off UNUSEDSIGNAL */
120     logic [7:0] pixel_a;
121     /* verilator lint_on UNUSEDSIGNAL */
122
123     // always_comb begin
124     // if (|sprite_palette_addr)
125     // $write("[%0d][%0d] sprite_palette_addr = [%04X]\n", h_count, v_count, tile_texture_addr);
126     // if (|tile_palette_addr)
127     // $write("[%0d][%0d] tile_palette_addr = [%04X]\n", h_count, v_count, sprite_texture_addr);
128     // end
129
130     // If everything works as intended, they will never both be active at the
131     // same time!
132     assign texture_addr = tile_texture_addr | sprite_texture_addr;
133     // assign texture_addr = (h_count < H_ACTIVE || h_count >= H_TOTAL - 32) ? tile_texture_addr : sprite_texture_addr;
134     // assign texture_addr = tile_texture_addr;
135
136     assign palette_off = palette_addr[3:0];
137     assign pixel_r = palette_data[31:24];
138     assign pixel_g = palette_data[23:16];
139     assign pixel_b = palette_data[15:8];
140     assign pixel_a = palette_data[7:0];
141
142     vram #(
143         .BYTES_PER_WORD(4),
144         .DATA_WIDTH(DATA_WIDTH)
145     ) mem (
146         .clk (clk),
147         .addr(address),
148         .din (writedata),
149         .be (byteenable),
150         .we (chipselect & write),
151         .dout(vram_dout),
152
153         .texture_addr(texture_addr), // 32 KiB == 13 bits
154         .texture_data(texture_data),
155         .tile_addr(tile_addr), // 8 KiB == 11 bits
156         .tile_data(tile_data),
157         .sprite_addr(sprite_addr), // 1 KiB == 8 bits
158         .sprite_data(sprite_data),
159         .palette_addr(palette_addr), // 512 B == 7 bits
160         .palette_data(palette_data)
161     );
162
163     /* Registers */
164
165     logic [31:0] reg_dout;
166
167     // Forced blanking (basically, turn the screen off)
168     // logic force_blank;
169
170     // Scroll
171     logic [8:0] scroll_x, scroll_y;
172
173     // Background color (For now, it's just black)
174     logic [23:0] bg_color;
175     logic [7:0] bg_r, bg_g, bg_b;
176
```

```
177   assign bg_r = bg_color[23:16];
178   assign bg_g = bg_color[15:8];
179   assign bg_b = bg_color[7:0];
180
181   always_ff @(posedge clk or posedge reset)
182     if (reset) begin
183       bg_color <= 24'h008000;
184       scroll_x <= 0;
185       scroll_y <= 0;
186       force_blanking <= 1;
187       reg_dout <= 32'h0;
188     end else begin
189       if (chipselect)
190         if (write)
191           unique case ({
192             address, 2'b0
193           })
194             16'hFF00: begin
195               if (byteenable[0]) force_blanking <= writedata[0];
196             end
197             16'hFF04: begin
198               if (byteenable[0]) scroll_x[7:0] <= writedata[7:0];
199               if (byteenable[1]) scroll_x[8] <= writedata[8];
200               if (byteenable[2]) scroll_y[7:0] <= writedata[23:16];
201               if (byteenable[3]) scroll_y[8] <= writedata[24];
202             end
203             16'hFF08: begin
204               if (byteenable[1]) bg_color[23:16] <= writedata[15:8];
205               if (byteenable[2]) bg_color[15:8] <= writedata[23:16];
206               if (byteenable[3]) bg_color[7:0] <= writedata[31:24];
207             end
208             default: ;
209           endcase
210         else if (read)
211           unique case ({
212             address, 2'b0
213           })
214             16'hFF0C: begin
215               reg_dout[15:0] <= {6'b0, h_count};
216               reg_dout[31:16] <= {6'b0, v_count};
217               // $write("[%04X] -> [%08X]\n", {address, 2'b0}, readdata);
218             end
219             default: ;
220           endcase
221     end
222
223   /* General logic */
224
225   // These should never interfere
226   assign readdata = read ? vram_dout | reg_dout : 0;
227
228   // Used to index into the tile and sprite palette offsets; determines the
229   // offset into the shift registers
230   // assign fine_scroll_x = scroll_x[2:0];
231
232   /* Tile logic */
233
234   tile_gpu #(
235       .HACTIVE(HACTIVE),
236       .VACTIVE(VACTIVE),
237       .HTOTAL (HTOTAL),
238       .VTOTAL (VTOTAL)
```

```
239     ) tile (
240         .clk(clk),
241         .vga_clk(VGA_CLK),
242         .blank_n(VGA_BLANK_N),
243         .reset(reset),
244         .h_count(h_count),
245         .v_count(v_count),
246         .next_scroll_x(scroll_x), // TODO: Set these for scrolling!
247         .next_scroll_y(scroll_y),
248         .tile_data(tile_data),
249         .texture_data(texture_data), // Texture data from previous fetch
250 
251         .tile_addr(tile_addr), // Which tile address to fetch
252         .texture_addr(tile_texture_addr), // Which texture sliver to fetch
253         .fg_palette_addr(fg_palette_addr), // Which palette address to use
254         .bg_palette_addr(bg_palette_addr) // Which palette address to use
255     );
256 
257     /* Sprite logic */
258 
259     sprite_gpu #(
260         .HACTIVE(HACTIVE),
261         .VTOTAL (VTOTAL)
262     ) sprite (
263         .clk(clk),
264         .vga_clk(VGA_CLK),
265         .blank_n(VGA_BLANK_N),
266         .reset(reset),
267         .h_count(h_count),
268         .v_count(v_count),
269         .sprite_data(sprite_data),
270         .texture_data(texture_data),
271 
272         .sprite_addr (sprite_addr),
273         .texture_addr(sprite_texture_addr),
274         .palette_addr(sprite_palette_addr)
275     );
276 
277     // Compositor
278     always_comb begin
279       // When blanking, only draw black!
280       {VGA_R, VGA_G, VGA_B} = {8'd0, 8'd0, 8'd0};
281       palette_addr = 0;
282 
283       if (VGA_BLANK_N) begin
284         if (|fg_palette_addr) palette_addr = fg_palette_addr;
285         else if (|sprite_palette_addr) palette_addr = sprite_palette_addr;
286         else palette_addr = bg_palette_addr;
287 
288         // If the final palette offset is nonzero, draw the pixel
289         // Otherwise, draw the background color
290         if (|palette_off) {VGA_R, VGA_G, VGA_B} = {pixel_r, pixel_g, pixel_b};
291         else {VGA_R, VGA_G, VGA_B} = {bg_r, bg_g, bg_b};
292       end
293 
294       // if (VGA_BLANK_N && VGA_CLK && h_count == 0)
295       // $write("[%0d][%0d] drawing [%02X%02X%02X]\n", h_count, v_count, VGA_R, VGA_G, VGA_B);
296     end
297 endmodule
```

vga_counter.sv

```
module vga_counter #(

    parameter logic [9:0] HACTIVE = 10'd640,
    parameter logic [9:0] HFRONT_PORCH = 10'd16,
    parameter logic [9:0] HSYNC = 10'd96,
    parameter logic [9:0] HBACK_PORCH = 10'd48,
    parameter logic [9:0] VACTIVE = 10'd480,
    parameter logic [9:0] VFRONT_PORCH = 10'd10,
    parameter logic [9:0] VSYNC = 10'd2,
    parameter logic [9:0] VBACK_PORCH = 10'd33
) (
    input clk50,
    reset,
    output wire [9:0] h_count, // Pixel column
    output logic [9:0] v_count, // Pixel row
    output logic vga_clk,
    h_sync, // 0 on sync, 1 otherwise
    v_sync,
    blank_n // High when not blanking
);

  localparam logic [9:0] HTOTAL = HACTIVE + HFRONT_PORCH + HSYNC + HBACK_PORCH; // 800
  localparam logic [9:0] VTOTAL = VACTIVE + VFRONT_PORCH + VSYNC + VBACK_PORCH; // 525

  // Parameters for hcount
  // parameter logic [9:0] HACTIVE = 10'd 640,
  // HFRONT_PORCH = 10'd 16,
  // HSYNC = 10'd 96,
  // HBACK_PORCH = 10'd 48,
  // HTOTAL = HACTIVE + HFRONT_PORCH + HSYNC + HBACK_PORCH; // 800
  // parameter logic [10:0] HACTIVE = 11'd1280,
  // HFRONT_PORCH = 11'd32,
  // HSYNC = 11'd192,
  // HBACK_PORCH = 11'd96,
  // HTOTAL = HACTIVE + HFRONT_PORCH + HSYNC + HBACK_PORCH, // 1600

  logic [10:0] h_count_raw;

  /*
  * 640 X 480 VGA timing for a 50 MHz clock: one pixel every other cycle
  *
  * HCOUNT 1599 0 1279 1599 0
  * _____________ _______
  * ___________| Video |___________| Video
  *
  *
  * |SYNC| BP |<-- HACTIVE -->|FP|SYNC| BP |<-- HACTIVE
  * _____________________ ____________
  * |____| VGA_HS |____|
  */

  /* vga_clk is 25 mhz
     * __ __ __
     * clk50 __| |__| |__|
     *
     * _____ __
     * hcount[0]__| |_____|
     */
  assign vga_clk = h_count_raw[0];
```

```
60  assign h_count = h_count_raw[10:1];
61
62  always_ff @(posedge clk50 or posedge reset) begin
63    // if (vga_clk) begin
64    // end
65    if (reset) begin
66      h_count_raw <= 11'd0;
67      v_count <= 10'd0;
68    end else if (vga_clk && h_count == HTOTAL - 1) begin
69      h_count_raw <= 11'd0;
70      v_count <= (v_count == VTOTAL - 1) ? 10'd0 : v_count + 10'd1;
71    end else h_count_raw <= h_count_raw + 11'd1;
72  end
73
74  // Hopefully this is abstracted away...
75  assign h_sync = (h_count < HACTIVE + HFRONT_PORCH) || (h_count >= HACTIVE + HFRONT_PORCH + HSYNC);
76  assign v_sync = (v_count < VACTIVE + VFRONT_PORCH) || (v_count >= VACTIVE + VFRONT_PORCH + VSYNC);
77  assign blank_n = (h_count < HACTIVE) && (v_count < VACTIVE);
78
79  // logic endOfLine;
80  //
81  // always_ff @(posedge clk50 or posedge reset)
82  // if (reset) hcount <= 0;
83  // else if (endOfLine) hcount <= 0;
84  // else hcount <= hcount + 11'd1;
85  //
86  // assign endOfLine = hcount == HTOTAL - 1;
87  //
88  // logic endOfField;
89  //
90  // always_ff @(posedge clk50 or posedge reset)
91  // if (reset) vcount <= 0;
92  // else if (endOfLine)
93  // if (endOfField) vcount <= 0;
94  // else vcount <= vcount + 10'd1;
95  //
96  // assign endOfField = vcount == VTOTAL - 1;
97
98  // Horizontal sync: from 0x520 to 0x5DF (0x57F)
99  // 101 0010 0000 to 101 1101 1111
100 // assign VGA_HS = !((hcount[10:8] == 3'b101) & !(hcount[7:5] == 3'b111));
101 // assign VGA_HS = (hcount[10:8] != 3'b101) | (hcount[7:5] == 3'b111);
102 // assign VGA_HS =
103 // assign VGA_VS = !(vcount[9:1] == (VACTIVE + VFRONT_PORCH) / 2);
104 // assign VGA_VS = vcount[9:1] != ((VACTIVE + VFRONT_PORCH) >> 1);
105
106 // assign VGA_SYNC_n = 1'b0; // For putting sync on the green signal; unused
107 // assign vsync = 1'b0;
108 // assign hsync = 1'b0;
109
110 // Horizontal active: 0 to 1279 Vertical active: 0 to 479
111 // 101 0000 0000 1280 01 1110 0000 480
112 // 110 0011 1111 1599 10 0000 1100 524
113 // assign VGA_BLANK_n = !( hcount[10] & (hcount[9] | hcount[8]) ) &
114 // !( vcount[9] | (vcount[8:5] == 4'b1111) );
115 // assign VGA_BLANK_n = !((hcount[10] & |hcount[9:8])) |
116 // (vcount[9] | (vcount[8:5] == 4'b1111)));
117 // assign h_blank =
118
119 /* vga_clk is 25 mhz
120    * __ __ __
121    * clk50 __| |__| |__|
```

```
122       *
123       * _____ __
124       * hcount[0]__| |_____|
125       */
126     // assign VGA_CLK = h_count[0]; // 25 MHz clock: rising edge sensitive
127
128 endmodule
```

vram.sv

```
1   /*
2   * Memory map (16-bit address space)
3   * MSB is used to determine which area to read from
4   * 0x0000 - 0x7FFF Texture data (32-bit)
5   * 0xDDDDDDDD Texture row data/palette offset (4bpp)
6   * 0x8000 - 0xBFFF Tile data (16-bit)
7   * 0bVHOPPPTT_TTTTTTTT
8   * ||||||| ||||||||
9   * ||||||++-+++++++++- Texture index
10  * |||+++------------ Palette index
11  * ||+---------------- Tile priority
12  * |+----------------- Horizontal flip
13  * +------------------ Vertical flip
14  * 0xF000 - 0xF3FF Sprite data (32-bit)
15  * 0bVHPPPTTT_TTTTTTTY_YYYYYYYX_XXXXXXXX
16  * |||||||| |||||||| |||||||| ||||||||
17  * |||||||| |||||||| |||||||+-+++++++++- X position
18  * |||||||| |||||||+-+++++++------------ Y position
19  * |||||+++-+++++++--------------------- Texture index
20  * ||+++-------------------------------- Palette index
21  * |+---------------------------------- Horizontal flip
22  * +----------------------------------- Vertical flip
23  *
24  * 0xF400 - 0F5FF Palette data
25  * 0bRRRRRRRR_GGGGGGGG_BBBBBBBB_AAAAAAAA Color data (A is unused for now)
26  *
27  * ====Registers====
28  * 0xF000 Control (W32)
29  * 0b0000_0000_0000_000F
30  * |
31  * +- Force blanking
32  * 0xF004 X scroll (W16)
33  * 0b0000_000X_XXXX_XXXX
34  * | |||| ||||
35  * +-++++-++++- X value
36  * 0xF006 Y scroll (W16)
37  * 0b0000_000Y_YYYY_YYYY
38  * | |||| ||||
39  * +-++++-++++- Y value
40  * 0xF008 Background color (W32)
41  * 0bRRRRRRRR_GGGGGGGG_BBBBBBBB_AAAAAAAA
42  * 0xF00c Horizontal count (R16)
43  * 0b0000_00HH_HHHH_HHHH
44  * || |||| ||||
45  * ++-++++-++++- Horizontal count
46  * 0xF010 Vertical count (R16)
47  * 0b0000_00VV_VVVV_VVVV
48  * || |||| ||||
49  * ++-++++-++++- Vertical count
50  */
```

```
51  module vram #(
52      // Bytes per word
53      parameter int BYTES_PER_WORD = 4,
54      parameter int ADDR_WIDTH = 14,
55      // # of bits read at once
56      parameter int DATA_WIDTH = 32,
57
58      parameter int TEXTURE_ADDR_WIDTH = 13,
59      parameter int TILE_ADDR_WIDTH = 13,
60      parameter int SPRITE_ADDR_WIDTH = 8,
61      parameter int PALETTE_ADDR_WIDTH = 7,
62
63      parameter int TEXTURE_DATA_WIDTH = 32,
64      parameter int TILE_DATA_WIDTH = 16,
65      parameter int SPRITE_DATA_WIDTH = 32,
66      parameter int PALETTE_DATA_WIDTH = 32
67  ) (
68      input clk,
69      // Wire these to access from the outside
70      input [ADDR_WIDTH-1:0] addr, // Always 14-bit (2**16 bytes == 2**14 words)
71      input [DATA_WIDTH-1:0] din,
72      input [BYTES_PER_WORD-1:0] be,
73      input we,
74      output logic [DATA_WIDTH-1:0] dout,
75
76      // These can all be accessed simultaneously
77      input logic [TEXTURE_ADDR_WIDTH-1:0] texture_addr, // 32 KiB == 13 bits
78      output logic [TEXTURE_DATA_WIDTH-1:0] texture_data,
79      input logic [TILE_ADDR_WIDTH-1:0] tile_addr, // 16 KiB == 12 bits
80
81      output logic [TILE_DATA_WIDTH-1:0] tile_data,
82      input logic [SPRITE_ADDR_WIDTH-1:0] sprite_addr, // 1 KiB == 8 bits
83      output logic [SPRITE_DATA_WIDTH-1:0] sprite_data,
84      input logic [PALETTE_ADDR_WIDTH-1:0] palette_addr, // 512 B == 7 bits
85      output logic [PALETTE_DATA_WIDTH-1:0] palette_data
86  );
87
88    logic texture_select, tile_select, sprite_select, palette_select;
89    logic [DATA_WIDTH-1:0] texture_dout, tile_dout, sprite_dout, palette_dout;
90    logic prev_tile_data_select;
91
92    logic [DATA_WIDTH-1:0] tile_data_raw;
93
94    assign texture_select = ~addr[13]; // 0x0000 - 0x7FFF
95    assign tile_select = addr[13] & ~addr[12]; // 0x8000 - 0xBFFF
96    assign sprite_select = &addr[13:10] & ~|addr[9:8]; // 0xF000 - 0xF3FF
97    assign palette_select = {addr, 2'b0} >= 16'hF400 && {addr, 2'b0} < 16'hF5FF;
98    // assign palette_select = &addr[13:10] & addr[9] & ~addr[8]; // 0xF400 - 0xF5FF
99
100   dual_port_be_ram #(
101       .ADDR_WIDTH(TEXTURE_ADDR_WIDTH) // 32 KiB
102   ) textures (
103       .clk(clk),
104
105       .addr_a(addr[TEXTURE_ADDR_WIDTH-1:0]),
106       .be_a (be),
107       .din_a (din),
108       .we_a (we & texture_select),
109       .dout_a(texture_dout),
110
111       .addr_b(texture_addr),
112       .be_b (0),
```

```
113         .din_b (0),
114         .we_b (0),
115         .dout_b(texture_data)
116     );
117
118     dual_port_be_ram #(
119         .ADDR_WIDTH(TILE_ADDR_WIDTH - 1) // 16 KiB
120     ) bg_tiles (
121         .clk(clk),
122
123         .addr_a(addr[TILE_ADDR_WIDTH-2:0]),
124         .be_a (be),
125         .din_a (din),
126         .we_a (we & tile_select),
127         .dout_a(tile_dout),
128
129         .addr_b(tile_addr[TILE_ADDR_WIDTH-1:1]),
130         .be_b (0),
131         .din_b (0),
132         .we_b (0),
133         .dout_b(tile_data_raw)
134     );
135     // assign tile_data = tile_data_raw[15:0];
136     assign tile_data = prev_tile_data_select ? tile_data_raw[31:16] : tile_data_raw[15:0];
137     always_ff @(posedge clk) prev_tile_data_select <= tile_addr[0];
138
139
140     dual_port_be_ram #(
141         .ADDR_WIDTH(SPRITE_ADDR_WIDTH) // 1 KiB
142     ) sprites (
143         .clk(clk),
144
145         .addr_a(addr[SPRITE_ADDR_WIDTH-1:0]),
146         .be_a (be),
147         .din_a (din),
148         .we_a (we & sprite_select),
149         .dout_a(sprite_dout),
150
151         .addr_b(sprite_addr),
152         .be_b (0),
153         .din_b (0),
154         .we_b (0),
155         .dout_b(sprite_data)
156     );
157
158     dual_port_be_ram #(
159         .ADDR_WIDTH(PALETTE_ADDR_WIDTH) // 512 B
160     ) palette (
161         .clk(clk),
162
163         .addr_a(addr[PALETTE_ADDR_WIDTH-1:0]),
164         .be_a (be),
165         .din_a (din),
166         .we_a (we & palette_select),
167         .dout_a(palette_dout),
168
169         .addr_b(palette_addr),
170         .be_b (0),
171         .din_b (0),
172         .we_b (0),
173         .dout_b(palette_data)
174     );
```

```
175
176     always_comb begin
177         dout = 0;
178
179         if (texture_select) dout = texture_dout;
180         else if (tile_select) dout = tile_dout;
181         else if (sprite_select) dout = sprite_dout;
182         else if (palette_select) dout = palette_dout;
183
184         // if (clk & tile_select)
185         // $write("Addr %04X (byte enable %04b) given value %04X\n", {addr, 2'b0}, be, din);
186     end
187
188 endmodule : vram
```

dual_port_be_ram.sv

```
1   // Quartus Prime SystemVerilog Template
2   //
3   // True Dual-Port RAM with different read/write addresses and single read/write clock
4   // and with a control for writing single bytes into the memory word; byte enable
5
6   // Read during write produces old data on ports A and B and old data on mixed ports
7   // For device families that do not support this mode (e.g. Stratix V) the ram is not inferred
8   module dual_port_be_ram #(
9       parameter int BYTES_PER_WORD = 4,
10      parameter int DATA_WIDTH = 32,
11      parameter int ADDR_WIDTH
12  ) (
13      input [ADDR_WIDTH-1:0] addr_a,
14      input [ADDR_WIDTH-1:0] addr_b,
15      input [BYTES_PER_WORD-1:0] be_a,
16      input [BYTES_PER_WORD-1:0] be_b,
17      input [DATA_WIDTH-1:0] din_a,
18      input [DATA_WIDTH-1:0] din_b,
19      input we_a,
20      we_b,
21      clk,
22      output [DATA_WIDTH-1:0] dout_a,
23      output [DATA_WIDTH-1:0] dout_b
24  );
25      localparam int BITS_PER_BYTE = 8;
26      localparam int NUM_WORDS = 1 << ADDR_WIDTH;
27
28      // model the RAM with two dimensional packed array
29      logic [BYTES_PER_WORD-1:0][BITS_PER_BYTE-1:0] ram[NUM_WORDS];
30
31      reg [DATA_WIDTH-1:0] reg_a, reg_b;
32
33      // port A
34      always @(posedge clk) begin
35          if (we_a) begin
36              if (be_a[0]) ram[addr_a][0] <= din_a[7:0];
37              if (be_a[1]) ram[addr_a][1] <= din_a[15:8];
38              if (be_a[2]) ram[addr_a][2] <= din_a[23:16];
39              if (be_a[3]) ram[addr_a][3] <= din_a[31:24];
40
41              // $write("VRAM: Writing [%08X][%04b] to rel. addr. [%04X]/byte addr. [%04X]\n", din_a, be_a,
42              // addr_a, {addr_a, 2'b0});
43              // $write("VRAM: ram now holds [%08X]\n", ram[addr_a]);
```

```
44
45          // for (int i = 0; i < BYTES_PER_WORD; i++)
46          // if (be_a[i]) ram[addr_a][i] <= din_a[(i*BITS_PER_BYTE)+:BITS_PER_BYTE];
47          // for (int i = 0; i < BYTES_PER_WORD; i++) begin
48          // if (be_a[i]) ram[addr_a][i] <= din_a[(i*BYTES_PER_WORD)+:BITS_PER_BYTE];
49          // end
50        end
51        reg_a <= ram[addr_a];
52      end
53
54      assign dout_a = reg_a;
55
56      // port B
57      always @(posedge clk) begin
58        if (we_b) begin
59          if (be_b[0]) ram[addr_b][0] <= din_b[7:0];
60          if (be_b[1]) ram[addr_b][1] <= din_b[15:8];
61          if (be_b[2]) ram[addr_b][2] <= din_b[23:16];
62          if (be_b[3]) ram[addr_b][3] <= din_b[31:24];
63
64          // for (int i = 0; i < BYTES_PER_WORD; i++)
65          // if (be_b[i]) ram[addr_b][i] <= din_b[(i*BITS_PER_BYTE)+:BITS_PER_BYTE];
66        end
67        reg_b <= ram[addr_b];
68      end
69
70      assign dout_b = reg_b;
71
72  endmodule : dual_port_be_ram
```

tile_gpu.sv

```
1   module tile_gpu #(
2       parameter int NUM_BG_LAYERS = 2,
3       parameter int TILE_ADDR_WIDTH = 12 + $clog2(NUM_BG_LAYERS),
4       parameter int TEXTURE_ADDR_WIDTH = 13,
5       parameter int PALETTE_ADDR_WIDTH = 7,
6       parameter int TILE_DATA_WIDTH = 16,
7       parameter int TEXTURE_DATA_WIDTH = 32,
8       parameter int BPP = 4, // Bits per pixel
9
10      parameter int VGA_COUNTER_WIDTH = 10,
11      parameter logic [VGA_COUNTER_WIDTH:0] CYCLES_PER_SLIVER = 32,
12      parameter logic [VGA_COUNTER_WIDTH-1:0] HACTIVE,
13      parameter logic [VGA_COUNTER_WIDTH-1:0] VACTIVE,
14      parameter logic [VGA_COUNTER_WIDTH-1:0] HTOTAL,
15      parameter logic [VGA_COUNTER_WIDTH-1:0] VTOTAL
16  ) (
17      input clk,
18      vga_clk,
19      blank_n,
20      reset,
21      input [VGA_COUNTER_WIDTH-1:0] h_count,
22      v_count,
23      input [VGA_COUNTER_WIDTH-2:0] next_scroll_x,
24      next_scroll_y,
25      input [TILE_DATA_WIDTH-1:0] tile_data, // Tile data from previous fetch
26      input [TEXTURE_DATA_WIDTH-1:0] texture_data, // Texture data from previous fetch
27
28      output logic [TILE_ADDR_WIDTH-1:0] tile_addr, // Which tile address to fetch
```

```
29      output logic [TEXTURE_ADDR_WIDTH-1:0] texture_addr, // Which texture sliver to fetch
30      // Palette address to draw in front of sprites
31      output logic [PALETTE_ADDR_WIDTH-1:0] fg_palette_addr,
32      // Palette address to draw behind sprites
33      output logic [PALETTE_ADDR_WIDTH-1:0] bg_palette_addr
34  );
35    localparam logic [VGA_COUNTER_WIDTH:0] HACTIVE_RAW = {HACTIVE, 1'b0};
36    localparam logic [VGA_COUNTER_WIDTH:0] HTOTAL_RAW = {HTOTAL, 1'b0};
37    localparam int TILE_CYCLE_WIDTH = $clog2(CYCLES_PER_SLIVER);
38
39    localparam logic [TILE_CYCLE_WIDTH-1:0] FETCH_ATTR_START = 0;
40    localparam logic [TILE_CYCLE_WIDTH-1:0] FETCH_ATTR_END = NUM_BG_LAYERS[TILE_CYCLE_WIDTH-1:0];
41    localparam logic [TILE_CYCLE_WIDTH-1:0] FETCH_TEXTURE_START = FETCH_ATTR_START + 1;
42    localparam logic [TILE_CYCLE_WIDTH-1:0] FETCH_TEXTURE_END = FETCH_ATTR_END + 1;
43    localparam logic [TILE_CYCLE_WIDTH-1:0] DRAW_SLIVER_START = FETCH_TEXTURE_START + 1;
44    localparam logic [TILE_CYCLE_WIDTH-1:0] DRAW_SLIVER_END = FETCH_TEXTURE_END + 1;
45
46    // We want to use the 50MHz clock in its entirety
47    logic [VGA_COUNTER_WIDTH:0] h_count_raw;
48    logic [VGA_COUNTER_WIDTH-2:0] screen_x, screen_y;
49    logic [TILE_CYCLE_WIDTH-1:0] tile_render_cycle;
50    assign h_count_raw = {h_count, vga_clk};
51    assign screen_x = h_count[VGA_COUNTER_WIDTH-1:1];
52    assign screen_y = v_count[VGA_COUNTER_WIDTH-1:1];
53    // assign screen_x_abs = screen_x + scroll_x;
54    // assign screen_y_abs = screen_y + scroll_y;
55    assign tile_render_cycle = h_count_raw[TILE_CYCLE_WIDTH-1:0];
56
57    logic fetch_attr, fetch_texture, draw_sliver;
58    assign fetch_attr = tile_render_cycle < FETCH_ATTR_END;
59    assign fetch_texture = tile_render_cycle >= FETCH_TEXTURE_START
60                           && tile_render_cycle < FETCH_TEXTURE_END;
61    assign draw_sliver = tile_render_cycle >= DRAW_SLIVER_START
62                           && tile_render_cycle < DRAW_SLIVER_END;
63
64    logic on_pre_render_line, before_next_line;
65    logic draw_first_two, draw_rest, draw_any;
66    // Are we on the line right before we turn on blanking?
67    assign on_pre_render_line = v_count == VTOTAL - 1;
68    // Are we two tiles before we start drawing again?
69    assign before_next_line = h_count_raw >= HTOTAL_RAW - (CYCLES_PER_SLIVER << 1);
70
71    // Draw the first two tiles of the next scanline
72    assign draw_first_two = (on_pre_render_line || (v_count < VACTIVE - 1)) && before_next_line;
73    // Draw two tiles ahead of the tile currently being drawn; draw the rest of
74    // the tiles on this scanline
75    assign draw_rest = blank_n && h_count_raw < HACTIVE_RAW - CYCLES_PER_SLIVER;
76    assign draw_any = draw_first_two | draw_rest;
77
78    // Scrolling is only updated once per frame, to avoid jittering
79    logic [VGA_COUNTER_WIDTH-2:0] scroll_x, scroll_y;
80    always_ff @(posedge clk or posedge reset) begin
81      if (reset) begin
82        scroll_x <= next_scroll_x;
83        scroll_y <= next_scroll_y;
84      end else if (on_pre_render_line && ~|h_count_raw) begin
85        scroll_x <= next_scroll_x;
86        scroll_y <= next_scroll_y;
87      end
88    end
89
90    /* verilator lint_off UNUSEDSIGNAL */
```

```
91    logic [VGA_COUNTER_WIDTH-2:0] tile_pos_x;
92    /* verilator lint_on UNUSEDSIGNAL */
93    logic [VGA_COUNTER_WIDTH-2:0] tile_pos_y;
94
95    logic [5:0] tile_ind_x, tile_ind_y;
96
97    assign tile_ind_x = tile_pos_x[VGA_COUNTER_WIDTH-2:3];
98    assign tile_ind_y = tile_pos_y[VGA_COUNTER_WIDTH-2:3];
99
100   always_comb begin
101     tile_pos_x = 0;
102     tile_pos_y = 0;
103
104     if (draw_first_two) begin
105       // Equivalent to scroll_x + screen_x - (HTOTAL - 2 * CYCLES_PER_TILE)
106       tile_pos_x = scroll_x + {4'b0, screen_x[TILE_CYCLE_WIDTH-1:0]};
107
108       if (on_pre_render_line) tile_pos_y = scroll_y;
109       else tile_pos_y = screen_y + scroll_y + {8'b0, v_count[0]};
110
111     end else if (draw_rest) begin
112       tile_pos_y = scroll_y + screen_y;
113       tile_pos_x = scroll_x + screen_x + 16;
114     end
115
116     // if (vga_clk && draw_any)
117     // $write("[%0d][%0d] Drawing [%0d][%0d]\n", h_count, v_count, tile_draw_x, tile_draw_y);
118   end
119
120   // We also need to save a couple of values
121   logic [2:0] prev_tile_off_y;
122   always_ff @(posedge clk) begin
123     prev_tile_off_y <= tile_pos_y[2:0];
124   end
125
126   /* verilator lint_off UNUSEDSIGNAL */
127   logic [TEXTURE_ADDR_WIDTH - 4:0] tile_texture_ind;
128   logic [PALETTE_ADDR_WIDTH - 5:0] next_tile_palette_ind, tile_palette_ind;
129
130   // TODO: Add tile priority and flipping
131   logic tile_flip_v, tile_flip_h;
132   logic next_tile_priority, tile_priority;
133   /* verilator lint_on UNUSEDSIGNAL */
134
135   assign tile_flip_v = tile_data[15];
136   assign tile_flip_h = tile_data[14];
137   assign next_tile_priority = tile_data[13];
138   assign next_tile_palette_ind = tile_data[12:10];
139   assign tile_texture_ind = tile_data[9:0];
140
141   always_ff @(posedge clk) begin
142     tile_palette_ind <= next_tile_palette_ind;
143     tile_priority <= next_tile_priority;
144   end
145
146   logic [TEXTURE_DATA_WIDTH-1:0] next_fg_palette_offs;
147   logic [TEXTURE_DATA_WIDTH-1:0] next_bg_palette_offs;
148   logic [TEXTURE_DATA_WIDTH-1:0] next_fg_palette_inds;
149   logic [TEXTURE_DATA_WIDTH-1:0] next_bg_palette_inds;
150
151   logic [TEXTURE_DATA_WIDTH * 2 - 1:0] fg_palette_offs;
152   logic [TEXTURE_DATA_WIDTH * 2 - 1:0] bg_palette_offs;
```

```
153     logic [TEXTURE_DATA_WIDTH * 2 - 1:0] fg_palette_inds;
154     logic [TEXTURE_DATA_WIDTH * 2 - 1:0] bg_palette_inds;
155
156     always_ff @(posedge clk or posedge reset)
157       if (reset) begin
158         next_fg_palette_offs <= 0;
159         next_bg_palette_offs <= 0;
160         next_fg_palette_inds <= 0;
161         next_bg_palette_inds <= 0;
162         fg_palette_offs <= 0;
163         fg_palette_inds <= 0;
164         bg_palette_offs <= 0;
165         bg_palette_inds <= 0;
166       end else begin
167         // if (blank_n && h_count == 0)
168         // $write(
169         // "[%0d][%0d]\t[%08X][%08X][%08X][%08X]\t[%0X][%0X]\n",
170         // h_count_raw,
171         // v_count,
172         // fg_palette_inds[31:0],
173         // fg_palette_offs[31:0],
174         // bg_palette_inds[31:0],
175         // bg_palette_offs[31:0],
176         // fg_palette_addr,
177         // bg_palette_addr
178         // );
179         // if (draw_any && v_count < 8) begin
180         // if (fetch_attr)
181         // $write(
182         // "[%0d][%0d]: Fetching tile Layer:[%0t] X:[%0d] Y:[%0d]\tAddress:[%0X]\n",
183         // h_count_raw,
184         // v_count,
185         // tile_addr[12],
186         // tile_ind_x,
187         // tile_ind_y,
188         // tile_addr
189         // );
190         // if (fetch_texture)
191         // $write(
192         // "[%0d][%0d]: Tile has data [%04X] palette [%0d] texture [%0d] (offset %0d)\t Address [%04X]\n",
193         // h_count_raw - 1,
194         // v_count,
195         // tile_data,
196         // next_tile_palette_ind,
197         // tile_texture_ind,
198         // prev_tile_off_y,
199         // {
200         // texture_addr, 2'b0
201         // }
202         // );
203         // if (draw_sliver)
204         // $write("[%0d][%0d]: Tile has sliver [%08X]\n", h_count_raw - 2, v_count, texture_data);
205         // end
206
207         // Draw slivers in increasing importance, making sure not to draw
208         // tranparent
209         // if (draw_sliver && draw_any) begin
210         // next_bg_palette_offs <= texture_data;
211         // next_bg_palette_inds <= {8{1'b0, tile_palette_ind}};
212         // end
213
214         if (draw_sliver && draw_any)
```

```
215           for (int i = 0; i < TEXTURE_DATA_WIDTH; i += BPP)
216           if (|texture_data[i+:BPP])
217             if (tile_priority) begin
218               next_fg_palette_offs[i+:BPP] <= texture_data[i+:BPP];
219               next_fg_palette_inds[i+:BPP-1] <= tile_palette_ind;
220             end else begin
221               next_bg_palette_offs[i+:BPP] <= texture_data[i+:BPP];
222               next_bg_palette_inds[i+:BPP-1] <= tile_palette_ind;
223             end
224
225         // At the end of each sliver, load in the next one
226         if (&tile_render_cycle) begin
227           // $write("[%0d][%0d] Next bg offs: [%08X]\n",
228           // h_count_raw[VGA_COUNTER_WIDTH:TILE_CYCLE_WIDTH], v_count, next_bg_palette_offs);
229
230           if (draw_any) begin
231             // Write the next buffers
232             fg_palette_offs <= {
233               next_fg_palette_offs, fg_palette_offs[(TEXTURE_DATA_WIDTH-1)+BPP : BPP]
234             };
235             fg_palette_inds <= {
236               next_fg_palette_inds, fg_palette_inds[(TEXTURE_DATA_WIDTH-1)+BPP : BPP]
237             };
238             bg_palette_offs <= {
239               next_bg_palette_offs, bg_palette_offs[(TEXTURE_DATA_WIDTH-1)+BPP : BPP]
240             };
241             bg_palette_inds <= {
242               next_bg_palette_inds, bg_palette_inds[(TEXTURE_DATA_WIDTH-1)+BPP : BPP]
243             };
244           end
245
246           // Clear the next buffers
247           next_fg_palette_offs <= 0;
248           next_fg_palette_inds <= 0;
249           next_bg_palette_offs <= 0;
250           next_bg_palette_inds <= 0;
251         end else if (&tile_render_cycle[1:0]) begin
252           // Still ensure that things are shifted
253           fg_palette_offs <= fg_palette_offs >> BPP;
254           bg_palette_offs <= bg_palette_offs >> BPP;
255           fg_palette_inds <= fg_palette_inds >> BPP;
256           bg_palette_inds <= bg_palette_inds >> BPP;
257         end
258
259       end
260
261     assign fg_palette_addr = {
262       fg_palette_inds[{1'b0, scroll_x[2:0], 2'b0}+:(BPP-1)],
263       fg_palette_offs[{1'b0, scroll_x[2:0], 2'b0}+:BPP]
264     };
265     assign bg_palette_addr = {
266       bg_palette_inds[{1'b0, scroll_x[2:0], 2'b0}+:(BPP-1)],
267       bg_palette_offs[{1'b0, scroll_x[2:0], 2'b0}+:BPP]
268     };
269
270     always_comb begin
271       tile_addr = 0;
272       texture_addr = 0;
273
274       if (draw_any) begin
275         // TODO: Add more background layers
276         if (fetch_attr) tile_addr = {tile_render_cycle[0], tile_ind_y, tile_ind_x};
```

```
277         if (fetch_texture) texture_addr = {tile_texture_ind, prev_tile_off_y};
278       end
279
280     end
281
282
283 endmodule : tile_gpu
```

sprite_gpu.sv

```
1  // While !HACTIVE, we have ~340 cycles to load the data of all sprites. This
2  // can be done through pipelining: when loading sprite data for sprite 1,
3  // fetch texture data for sprite 0. Doing so, we only need 258 cycles to
4  // buffer all sprites for the next line!
5  //
6  // Cycle 1280: clear line buffer data; set it all to 0x0
7  // It's a good thing we can use 0, otherwise I'm guessing this would suck
8  //
9  // Cycle 1280—1537: load all 256 (!) sprites into buffer
10 //
11 // 0x500 | 0x501 | 0x502 | ... | 0x5FF | 0x600 | 0x601
12 // data 0 | data 1| data 2| | data 255 | |
13 // | tex. 0| tex. 1| | tex. 254 | tex. 255|
14 // | | draw 0| | draw 253 | draw 254| draw 255
15
16 module sprite_gpu #(
17     parameter int SPRITE_ADDR_WIDTH = 8,
18     parameter int TEXTURE_ADDR_WIDTH = 13,
19     parameter int PALETTE_ADDR_WIDTH = 7,
20     parameter int SPRITE_DATA_WIDTH = 32,
21     parameter int TEXTURE_DATA_WIDTH = 32,
22     parameter int VGA_COUNTER_WIDTH = 10,
23
24     parameter logic [VGA_COUNTER_WIDTH—1:0] HACTIVE,
25     parameter logic [VGA_COUNTER_WIDTH—1:0] VTOTAL
26 ) (
27     input clk,
28     vga_clk,
29     blank_n,
30     reset,
31     input [VGA_COUNTER_WIDTH—1:0] h_count,
32     v_count,
33     input [SPRITE_DATA_WIDTH—1:0] sprite_data, // Sprite data from previous fetch
34     input [TEXTURE_DATA_WIDTH—1:0] texture_data, // Texture data from previous fetch
35
36     output logic [ SPRITE_ADDR_WIDTH—1:0] sprite_addr, // Which sprite address to fetch
37     output logic [TEXTURE_ADDR_WIDTH—1:0] texture_addr, // Which texture sliver to fetch
38     output logic [PALETTE_ADDR_WIDTH—1:0] palette_addr // Which palette address to use
39 );
40
41
42   localparam logic [VGA_COUNTER_WIDTH:0] NUM_SPRITES = 1 << SPRITE_ADDR_WIDTH;
43
44   // Timing intervals
45   localparam logic [VGA_COUNTER_WIDTH:0] CLEAR = {HACTIVE, 1'b0};
46   localparam logic [VGA_COUNTER_WIDTH:0] FETCH_ATTR_BEGIN = CLEAR;
47   localparam logic [VGA_COUNTER_WIDTH:0] FETCH_ATTR_END = FETCH_ATTR_BEGIN + NUM_SPRITES;
48   localparam logic [VGA_COUNTER_WIDTH:0] FETCH_TEXTURE_BEGIN = FETCH_ATTR_BEGIN + 1;
49   localparam logic [VGA_COUNTER_WIDTH:0] FETCH_TEXTURE_END = FETCH_TEXTURE_BEGIN + NUM_SPRITES;
50   localparam logic [VGA_COUNTER_WIDTH:0] DRAW_SLIVER_BEGIN = FETCH_TEXTURE_BEGIN + 1;
```

```
51  localparam logic [VGA_COUNTER_WIDTH:0] DRAW_SLIVER_END = DRAW_SLIVER_BEGIN + NUM_SPRITES;
52
53  // So we can take full advantage of the 50MHz clock
54  logic [VGA_COUNTER_WIDTH:0] h_count_raw;
55  assign h_count_raw = {h_count, vga_clk};
56
57  logic [VGA_COUNTER_WIDTH-2:0] screen_x;
58  logic [VGA_COUNTER_WIDTH-3:0] screen_y;
59  assign screen_x = h_count[VGA_COUNTER_WIDTH-1:1];
60  // Topmost bit is never used, divide v_count by 2
61  assign screen_y = v_count == (VTOTAL - 1) ? 0 : v_count[VGA_COUNTER_WIDTH-2:1] + 1;
62
63  // Reading is simple enough
64  logic unused;
65  logic [2:0] palette_ind;
66  logic [3:0] palette_offset;
67
68  // Data related to the sprites
69  logic [8:0] next_sprite_x, sprite_x;
70  logic [7:0] next_sprite_y, sprite_y;
71  logic [9:0] sprite_texture_ind;
72  logic [3:0] next_sprite_palette_ind, sprite_palette_ind;
73  assign next_sprite_x = sprite_data[8:0];
74  assign next_sprite_y = sprite_data[16:9];
75  assign sprite_texture_ind = sprite_data[26:17];
76  assign next_sprite_palette_ind = {1'b0, sprite_data[29:27]};
77
78  // TODO: Add flipping
79  /* verilator lint_off UNUSEDSIGNAL */
80  logic next_sprite_flip_v, next_sprite_flip_h;
81  assign next_sprite_flip_v = sprite_data[31];
82  assign next_sprite_flip_h = sprite_data[30];
83
84  // Difference in sprite's position and screen's position
85  logic [7:0] sprite_y_offset;
86
87  /* verilator lint_on UNUSEDSIGNAL */
88  assign sprite_y_offset = screen_y - next_sprite_y;
89
90  // When should we do each thing
91  logic update_line_buffers, clear, fetch_attr, fetch_texture, draw_sliver;
92  // Since each pixel is 2x2 pixels displayed, only perform actions on odd
93  // scanlines
94  assign update_line_buffers = ((!blank_n && v_count[0]) || v_count == VTOTAL - 1)
95                    && h_count_raw >= CLEAR && h_count_raw < DRAW_SLIVER_END;
96
97  assign clear = reset | (update_line_buffers && h_count_raw == CLEAR);
98  assign fetch_attr = h_count_raw >= FETCH_ATTR_BEGIN && h_count_raw < FETCH_ATTR_END;
99  assign fetch_texture = h_count_raw >= FETCH_TEXTURE_BEGIN && h_count_raw < FETCH_TEXTURE_END;
100 // Not only do we need to be at the right time, the current sprite must also
101 // be visible somewhere
102 assign draw_sliver = update_line_buffers && h_count_raw >= DRAW_SLIVER_BEGIN
103                           && h_count_raw < DRAW_SLIVER_END
104                           && screen_y >= sprite_y
105                           && screen_y < sprite_y + 8;
106
107 // Store the indicies and offsets separately to make loading easier
108 linebuffer #(
109     .DATA_WIDTH_R(4)
110 ) palette_inds_buffer (
111     .clk(clk),
112     .we(draw_sliver),
```

```
113         .clear(clear),
114         .addr_r(screen_x),
115         .addr_w(sprite_x),
116         .data_r({unused, palette_ind}),
117         // TODO: Rewrite this later to be less stupid
118         .data_w({8{sprite_palette_ind}})
119
120     );
121
122     linebuffer #(
123         .DATA_WIDTH_R(4)
124     ) palette_offs_buffer (
125         .clk(clk),
126         .we(draw_sliver),
127         .clear(clear),
128         .addr_r(screen_x),
129         .addr_w(sprite_x),
130         .data_r(palette_offset),
131         .data_w(texture_data)
132     );
133
134     always_ff @(posedge clk) begin
135       // Pipe these all to so that they're visible on the next
136       sprite_x <= next_sprite_x;
137       sprite_y <= next_sprite_y;
138       sprite_palette_ind <= next_sprite_palette_ind;
139     end
140
141     always_comb begin
142       sprite_addr = 0;
143       texture_addr = 0;
144       palette_addr = 0;
145
146       // Only draw the buffer when not blanking
147       if (blank_n) palette_addr = {palette_ind[2:0], palette_offset};
148
149       if (update_line_buffers) begin
150         // if (clear) $write("[%02X][%03X]: Clearing\n", v_count, h_count_raw);
151
152         // Technically this should be h_count_raw - HACTIVE, but HACTIVE is
153         // a multiple of 256, so this also works!
154         if (fetch_attr) sprite_addr = h_count_raw[7:0];
155         // Use the texture index fetched last cycle, using the sprite's relative
156         // y position to choose the sliver
157         if (fetch_texture) texture_addr = {sprite_texture_ind, sprite_y_offset[2:0]};
158
159         // if (fetch_attr)
160         // $write("[%0d][%0d]: Fetching sprite [%02X]\n", h_count_raw, v_count, sprite_addr);
161         //
162         // if (fetch_texture) begin
163         // $write("[%0d][%0d]: Fetched sprite [%0d] with values [%0t][%0t][%01X][%02X][%02X][%03X]\n",
164         // h_count_raw, v_count, h_count_raw[7:0] - 1, next_sprite_flip_v, next_sprite_flip_h,
165         // next_sprite_palette_ind, texture_addr[9:3], next_sprite_y, next_sprite_x);
166         // end
167
168         // if (draw_sliver)
169         // $write(
170         // "[%0d][%0d]: Drawing sprite [%02X] at [%0d][%0d]\n",
171         // h_count_raw,
172         // v_count,
173         // h_count_raw[7:0] - 2,
174         // sprite_x,
```

```
175         // sprite_y
176         // );
177       end
178
179     end
180
181 endmodule : sprite_gpu
```

linebuffer.sv

```
1  module linebuffer #(
2      parameter int ADDR_WIDTH = 9, // 512 == 2 ** 9
3      parameter int DATA_WIDTH_R = 4, // Bits per pixel
4      parameter int DATA_WIDTH_W = 32 // Bits per sliver
5  ) (
6      input [ADDR_WIDTH-1:0] addr_r,
7      input [ADDR_WIDTH-1:0] addr_w,
8      input [DATA_WIDTH_W-1:0] data_w,
9      input we,
10     clk,
11     clear,
12     output reg [DATA_WIDTH_R-1:0] data_r
13 );
14   localparam int RATIO = DATA_WIDTH_W / DATA_WIDTH_R;
15   localparam int DEPTH = 1 << ADDR_WIDTH;
16   // Use a multi-dimensional packed array to model the different read/ram width
17   reg [DATA_WIDTH_R-1:0] ram[DEPTH];
18   reg [DATA_WIDTH_R-1:0] data_reg_r;
19
20   always @(posedge clk) begin
21     if (clear) for (int i = 0; i < DEPTH; i++) ram[i] <= 0;
22     else if (we)
23       for (int i = 0; i < RATIO; i++)
24       if (|data_w[i*DATA_WIDTH_R+:DATA_WIDTH_R]) begin
25         // $write("Drawing [%02X] to [%03X]\n", data_w[(i*DATA_WIDTH_R)+:DATA_WIDTH_R],
26         // addr_w + i[ADDR_WIDTH-1:0]);
27         ram[addr_w+i[ADDR_WIDTH-1:0]] <= data_w[i*DATA_WIDTH_R+:DATA_WIDTH_R];
28       end
29     data_reg_r <= ram[addr_r];
30   end
31
32   assign data_r = data_reg_r;
33
34 endmodule : linebuffer
```

Software

gpu.cpp

```
1  // https://zipcpu.com/blog/2017/06/21/looking-at-verilator.html
2
3  #include <SDL3/SDL.h>
4  #include <SDL3/SDL_hints.h>
5  #include <SDL3/SDL_main.h>
6  #include <SDL3/SDL_pixels.h>
7  #include <SDL3/SDL_render.h>
8  #include <cassert>
9  #include <cstdlib>
10 #include <cstring>
```

```
11 #include <iostream>
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <verilated.h>
15
16 #include "Vgpu.h"
17
18 static inline void SDL_Die(const char *msg) {
19   SDL_Log("%s (%s)\n", msg, SDL_GetError());
20   exit(EXIT_FAILURE);
21 }
22
23 using u8 = uint8_t;
24 using u16 = uint16_t;
25 using u32 = uint32_t;
26 using u64 = uint64_t;
27
28 constexpr int VGA_H_ACTIVE = 640;
29 constexpr int VGA_H_BACK_PORCH = 48;
30 constexpr int VGA_H_SYNC = 96;
31 constexpr int VGA_H_FRONT_PORCH = 16;
32 // constexpr int VGA_H_ACTIVE = 1;
33 // constexpr int VGA_H_BACK_PORCH = 1;
34 // constexpr int VGA_H_SYNC = 1;
35 // constexpr int VGA_H_FRONT_PORCH = 1;
36 constexpr int VGA_H_TOTAL =
37     VGA_H_ACTIVE + VGA_H_BACK_PORCH + VGA_H_SYNC + VGA_H_FRONT_PORCH;
38
39 constexpr int VGA_V_ACTIVE = 480;
40 constexpr int VGA_V_BACK_PORCH = 33;
41 constexpr int VGA_V_SYNC = 2;
42 constexpr int VGA_V_FRONT_PORCH = 10;
43 // constexpr int VGA_V_ACTIVE = 1;
44 // constexpr int VGA_V_BACK_PORCH = 1;
45 // constexpr int VGA_V_SYNC = 1;
46 // constexpr int VGA_V_FRONT_PORCH = 1;
47 constexpr int VGA_V_TOTAL =
48     VGA_V_ACTIVE + VGA_V_BACK_PORCH + VGA_V_SYNC + VGA_H_FRONT_PORCH;
49
50 typedef struct {
51   uint8_t b, g, r, a;
52 } Color;
53
54 typedef struct {
55   u32 x : 9;
56   u32 y : 8;
57   u32 tile_ind : 10;
58   u32 palette_ind : 3;
59   u32 flip_h : 1;
60   u32 flip_v : 1;
61 } Sprite;
62
63 typedef struct {
64   u16 texture_ind : 10;
65   u16 palette_ind : 3;
66   u16 priority : 1;
67   u16 flip_h : 1;
68   u16 flip_v : 1;
69 } Tile;
70
71 typedef struct {
72   u32 slivers[8];
```

```
73  } Texture;
74
75  typedef struct {
76    uint16_t addr;
77    uint32_t data;
78    uint8_t be;
79  } WriteCmd;
80
81  template <class M> class Testbench {
82  protected:
83    uint64_t ticks;
84    M *module;
85
86  public:
87    Testbench(void) {
88      module = new M;
89      ticks = 0l;
90    }
91
92    virtual ~Testbench(void) {
93      module->final();
94      delete module;
95      module = NULL;
96    }
97
98    virtual void reset(void) {
99      module->reset = 1;
100     // Make sure any inheritance gets applied
101     this->tick();
102     module->reset = 0;
103   }
104
105   virtual void tick(void) {
106     // Increment our own internal time reference
107     ticks++;
108
109     // Make sure any combinatorial logic depending upon
110     // inputs that may have changed before we called tick()
111     // has settled before the rising edge of the clock.
112     module->clk = 0;
113     module->eval();
114
115     // Toggle the clock
116
117     // Rising edge
118     module->clk = 1;
119     module->eval();
120
121     // Falling edge
122     module->clk = 0;
123     module->eval();
124   }
125
126   virtual bool done(void) { return (Verilated::gotFinish()); }
127 };
128
129 constexpr u32 TEXTURE_ADDR_START = 0x0000;
130 constexpr u32 TILE_ADDR_START = 0x8000;
131 constexpr u32 PALETTE_ADDR_START = 0xF400;
132 constexpr u32 SPRITE_ADDR_START = 0xF000;
133
134 class GPUTestBench : public Testbench<Vgpu> {
```

```
135     SDL_Window *window = nullptr;
136     SDL_Renderer *renderer = nullptr;
137     SDL_Texture *texture = nullptr;
138
139     Color *screen_buffer = nullptr;
140
141     // Negative edge == reset
142     bool prev_hs = true, prev_vs = true;
143     int h_count = 0, v_count = 0;
144     uint64_t vga_ticks = 0;
145
146   public:
147     GPUTestBench() : Testbench<Vgpu>() {
148       if (!SDL_CreateWindowAndRenderer("GPU Verilator Testing", VGA_H_ACTIVE * 3,
149                                        VGA_V_ACTIVE * 3, 0, &window, &renderer))
150         SDL_Die("SDL_CreateWindowAndRenderer()");
151
152       if ((texture = SDL_CreateTexture(renderer, SDL_PIXELFORMAT_ARGB8888,
153                                        SDL_TEXTUREACCESS_STATIC, VGA_H_ACTIVE,
154                                        VGA_V_ACTIVE)) == nullptr)
155         SDL_Die("SDL_CreateTexture()");
156
157       screen_buffer = new Color[VGA_H_ACTIVE * VGA_V_ACTIVE];
158
159       SDL_Log("Finished initializing SDL\n");
160     }
161
162     ~GPUTestBench() {
163
164       delete screen_buffer;
165
166       SDL_DestroyRenderer(renderer);
167       renderer = nullptr;
168       SDL_DestroyWindow(window);
169       window = nullptr;
170     }
171
172     void iowrite32(u16 addr, u32 data) {
173       module->address = addr >> 2;
174       module->writedata = data;
175       module->byteenable = 0x0F;
176       module->chipselect = 1;
177       module->write = 1;
178
179       tick();
180
181       module->chipselect = 0;
182       module->write = 0;
183     }
184     void iowrite16(u16 addr, u16 data) {
185       module->address = addr >> 2;
186       module->writedata = ((u32)data) << ((addr & 0b10) * 8);
187       module->byteenable = 0b0011 << (addr & 0b10);
188       module->chipselect = 1;
189       module->write = 1;
190
191       tick();
192
193       module->chipselect = 0;
194       module->write = 0;
195     }
196     u16 ioread16(u16 addr) {
```

```
197       module->address = addr >> 2;
198       module->chipselect = 1;
199       module->read = 1;
200
201       tick();
202
203       module->chipselect = 0;
204       module->read = 0;
205
206       return (addr & 0b10) ? module->readdata >> 16 : module->readdata & 0xFFFF;
207     }
208     void iowrite8(u16 addr, u8 data) {
209       module->address = addr >> 2;
210       module->writedata = ((u32)data) << ((addr & 0b11) * 8);
211       module->byteenable = 0b0001 << (addr & 0b11);
212       module->chipselect = 1;
213       module->write = 1;
214
215       tick();
216
217       module->chipselect = 0;
218       module->write = 0;
219     }
220
221     virtual void reset(void) {
222       Testbench<Vgpu>::reset();
223       h_count = 0;
224       v_count = 0;
225       prev_hs = true;
226       prev_vs = true;
227     }
228
229     virtual void tick(void) {
230
231       // Model ticking, based on the component's current state
232       if (module->VGA_CLK) {
233         // printf("%d %d\n", h_count, v_count);
234         // Only render when not blanking!
235
236         if (!module->VGA_VS && prev_vs)
237           v_count = -(VGA_V_BACK_PORCH + VGA_V_SYNC);
238
239         if (!module->VGA_HS && prev_hs) {
240           h_count = -(VGA_H_BACK_PORCH + VGA_H_SYNC);
241           v_count++;
242         }
243         h_count++;
244
245         prev_hs = module->VGA_HS;
246         prev_vs = module->VGA_VS;
247       }
248
249       // printf("\nTick %8ld (Reset = [%d], VGA_CLK = [%d])\n", ticks,
250       // module->reset, module->VGA_CLK);
251
252       // Assert that timings are correct
253
254       // Tick the actual component
255       Testbench<Vgpu>::tick();
256
257       if (module->VGA_CLK) {
258         if (module->VGA_BLANK_N) {
```

```
259             assert(h_count >= 0);
260             assert(h_count < VGA_H_ACTIVE);
261             assert(v_count >= 0);
262             assert(v_count < VGA_V_ACTIVE);
263 
264             screen_buffer[v_count * VGA_H_ACTIVE + h_count] = {
265                 module->VGA_B, module->VGA_G, module->VGA_R, 0xFF};
266           } else {
267             assert(module->VGA_R == 0);
268             assert(module->VGA_G == 0);
269             assert(module->VGA_B == 0);
270           }
271 
272           if (!module->VGA_VS && prev_vs) {
273             // Render once per frame, just for performance
274             SDL_UpdateTexture(texture, NULL, screen_buffer,
275                               VGA_H_ACTIVE * sizeof(Color));
276             SDL_RenderClear(renderer);
277             SDL_RenderTexture(renderer, texture, NULL, NULL);
278             SDL_RenderPresent(renderer);
279           }
280         }
281       }
282     };
283 
284     int main(int argc, char **argv) {
285       Verilated::commandArgs(argc, argv);
286 
287       if (!SDL_Init(SDL_INIT_VIDEO))
288         SDL_Die("SDL_Init()");
289 
290       GPUTestBench *tb = new GPUTestBench();
291       bool running = true;
292       SDL_Event e;
293       SDL_zero(e);
294 
295       std::cout << "Starting simulation...\n";
296 
297       tb->reset();
298 
299       // Move all the unused OAM sprites offscreen
300       Sprite empty_sprite = {320, 240, 0, 0, 0, 0};
301       for (u32 i = 0; i < 256; i++) {
302         tb->iowrite32(SPRITE_ADDR_START + (i << 2), *(u32 *)&empty_sprite);
303         tb->tick();
304       }
305 
306       // Palette 0 will be various shades of gray
307       for (u32 i = 0; i < 16; i++)
308         tb->iowrite32(PALETTE_ADDR_START + (i << 2), 0x0F0F0F00 * i);
309 
310       // Palette 1 has purple at offset 1
311       tb->iowrite32(PALETTE_ADDR_START + 64 + 4, 0xFF00FF00);
312 
313       // A gradient for texture 1
314       for (u32 i = 0; i < 8; i++)
315         tb->iowrite32(TEXTURE_ADDR_START + 32 + i * 4,
316                       0x123456789ABCDEF >> (i * 4));
317 
318       // And just color 1 for texture 2
319       for (u32 i = 0; i < 8; i++)
320         tb->iowrite32(TEXTURE_ADDR_START + 64 + i * 4, 0x11111111);
```

```
321
322     // for (u32 ind = 3; ind < 1024; ind++)
323     // for (u32 i = 0; i < 8; i++)
324     // tb->iowrite32(TEXTURE_ADDR_START + ind * 32 + i * 4,
325     // 0x123456789ABCDEF >> (i * 4));
326
327     // Draw a checkerboard
328
329     Tile tile = {};
330     // printf("Printing tile\n");
331     // tb->iowrite32(TILE_ADDR_START + 0x2000, 0xFFFFFFFF);
332     // tb->iowrite32(TILE_ADDR_START + 0x2000, 0xFFFFFFFF);
333     // tb->iowrite16(TILE_ADDR_START + 0x2000, *(u16 *)&tile);
334     for (u32 y = 0; y < 64; y++) {
335       for (u32 x = 0; x < 64; x++) {
336         tile.texture_ind = ((x & 0b1) ^ (y & 0b1)) + 1;
337         tile.palette_ind = (x & 0b1) ^ (y & 0b1);
338         // printf("%d %d %d\n", y, x, tile.texture_ind);
339         tb->iowrite16(TILE_ADDR_START + (y << 7) + (x << 1), *(u16 *)&tile);
340         // tb->iowrite16(TILE_ADDR_START + (0x2000) + (y << 7) + (x << 1),
341         // *(u16 *)&tile);
342       }
343     }
344
345     // Draw a sprite
346     Sprite test_sprite = {20, 20, 1, 0, 0, 0};
347     tb->iowrite32(SPRITE_ADDR_START + 255 * 4, *(u32 *)&test_sprite);
348
349     // Finally, disable forced blanking
350     tb->iowrite8(0xFF00, 0);
351
352     u64 frame = 0;
353     bool blank = false;
354
355     for (u32 i = 0; !tb->done() && running; i++) {
356       u16 v_count = tb->ioread16(0xFF0E);
357       if (!blank && v_count == 0)
358         blank = true;
359       else if (blank && v_count == 480) {
360         blank = false;
361         frame++;
362       }
363       else if (!blank && v_count == 481) {
364         tb->iowrite16(0xFF04, frame);
365         tb->iowrite16(0xFF06, frame);
366       }
367
368       while (SDL_PollEvent(&e))
369         if (e.type == SDL_EVENT_QUIT)
370           running = false;
371     }
372
373     delete tb;
374
375     SDL_Quit();
376
377     return EXIT_SUCCESS;
378 }
```

4.0.1 game_map.h

```
#ifndef GAME_MAP_H
#define GAME_MAP_H

#include "types.h"
#include "level_reader.h"

#define LEVEL_COUNT 1
#define LEVEL_0_ROWS 37
#define LEVEL_0_COLS 26



// Attention for ramp naming convention
// UP_X_RAMP means that x and z are increasing
// DOWN_X_RAMP means that z is decreasing as x is increasing
enum TileType {
    NO_TILE = 0,
    START_TILE = 1,
    FLAT = 2,
    UP_Y_RAMP = 3,
    UP_X_RAMP = 4,
    DOWN_X_RAMP = 6,
    DOWN_Y_RAMP = 7,
    WIN_TILE = 8
};

Tile check_position(MarbleState3D *marble_state3D);
void free_levels();
void initialize_levels(MarbleState3D *marble_state3D);

extern Tile*** levels;
extern int current_level;
extern int current_level_cols;
extern int current_level_rows;


#endif
```

4.0.2 game_map.c

```
#include "game_map.h"
#include <stdlib.h>
#include <stdio.h>

int current_level = 0;
int current_level_rows = LEVEL_0_ROWS;
int current_level_cols = LEVEL_0_COLS;
Tile*** levels = NULL;

void initialize_levels(MarbleState3D *marble_state3D) {

    levels = malloc(LEVEL_COUNT * sizeof(Tile**));

   Tile** level0 = read_level("levels/level4.csv",37,26, marble_state3D);
   levels[0] = level0;

   /*
   for (int i = 0; i < 37; i++) {
    for (int j = 0; j < 26; j++) {
        printf("%d ", level0[i][j].z_idx);
```

```
21          }
22          printf("\n");
23        }
24          */
25
26
27      printf("%d, %d, %d\n", level0[31][14].type, level0[31][14].x_idx, level0[31][14].y_idx);
28
29  }
30
31  void free_levels() {
32      for (int L = 0; L < LEVEL_COUNT; ++L) {
33          Tile **lvl = levels[L];
34          for (int i = 0; i < LEVEL_0_ROWS; ++i) {
35              free(lvl[i]);
36          }
37          free(lvl);
38      }
39      free(levels);
40  }
41
42
43  Tile check_position(MarbleState3D *marble_state3D) {
44          int x_idx = (int)marble_state3D->pos3D.x;
45          int y_idx = (int)marble_state3D->pos3D.y;
46
47      if (x_idx < 0) {
48          x_idx = 0;
49          marble_state3D->pos3D.x = 0;
50      }
51      if (y_idx < 0) {
52          y_idx = 0;
53          marble_state3D->pos3D.y = 0;
54      }
55
56      // printf("%d\n", levels[current_level][y_idx][x_idx].type);
57
58          return levels[current_level][y_idx][x_idx];
59  }
```

4.0.3 level_reader.h

```
1   #ifndef LEVEL_READER_H
2   #define LEVEL_READER_H
3
4   #include "types.h"
5   #include "game_map.h"
6
7   Tile** read_level(const char* filename, int height, int width, MarbleState3D *marble_state3D);
8
9   typedef struct {
10      unsigned char red, green, blue, alpha;
11  } ColorRGBA;
12
13  typedef struct {
14      char palette_indicies[8][8];
15  } Texture;
16
17  typedef struct {
18      char *tiles;
```

```
19      char height;
20      char width;
21  } Tilemap;
22
23  #endif
```

4.0.4 level_reader.c

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4   #include "level_reader.h"
5
6   /**
7    * Reads a level from a CSV where each cell is formatted as "z,type"
8    * and returns a heightwidth Tile** array.
9    *
10   * @param filename path to the CSV file
11   * @param height number of rows in the level
12   * @param width number of columns in the level
13   * @return pointer to an array of Tile* (length = height), each pointing
14   * to an array of Tiles (length = width). NULL on failure.
15   */
16  Tile** read_level(const char* filename, int height, int width, MarbleState3D *marble_state3D) {
17      FILE* fp = fopen(filename, "r");
18      if (!fp) {
19          perror("fopen");
20          return NULL;
21      }
22
23      // Allocate the array of row pointers
24      Tile** level = malloc(height * sizeof(Tile*));
25      if (!level) {
26          perror("malloc");
27          fclose(fp);
28          return NULL;
29      }
30
31      // Allocate each row
32      for (int y = 0; y < height; y++) {
33          level[y] = malloc(width * sizeof(Tile));
34          if (!level[y]) {
35              perror("malloc");
36              // clean up previously allocated rows
37              for (int j = 0; j < y; j++) free(level[j]);
38              free(level);
39              fclose(fp);
40              return NULL;
41          }
42      }
43
44      // Buffer to hold each line
45      size_t bufsize = 4 * width * 8; // rough estimate
46      char* line = malloc(bufsize);
47      if (!line) {
48          perror("malloc");
49          // clean up
50          for (int y = 0; y < height; ++y) free(level[y]);
51          free(level);
52          fclose(fp);
```

```
53          return NULL;
54      }
55
56      // Read each row
57      for (int y = 0; y < height; y++) {
58          if (!fgets(line, bufsize, fp)) {
59              fprintf(stderr, "Premature end of file at row %d\n", y);
60              break;
61          }
62          char* p = line;
63          for (int x = 0; x < width; ++x) {
64              // Find opening quote
65              p = strchr(p, '"');
66              if (!p) {
67                  fprintf(stderr, "Parse error at row %d, col %d\n", y, x);
68                  goto fail;
69              }
70              p++; // skip the '"'
71
72              // Extract up to closing quote
73              char cell[32];
74              char* endq = strchr(p, '"');
75              if (!endq) {
76                  fprintf(stderr, "Unterminated quote at row %d, col %d\n", y, x);
77                  goto fail;
78              }
79              int len = endq - p;
80              if (len >= (int)sizeof(cell)) len = sizeof(cell) - 1;
81              memcpy(cell, p, len);
82              cell[len] = '\0';
83
84              // Parse z,type
85              int z, type;
86              if (sscanf(cell, "%d,%d", &z, &type) != 2) {
87                  fprintf(stderr, "Invalid cell \"%s\" at row %d, col %d\n", cell, y, x);
88                  goto fail;
89              }
90
91              // Store into struct
92              level[y][x].x_idx = x;
93              level[y][x].y_idx = y;
94              level[y][x].z_idx = z;
95              level[y][x].type = type;
96
97              if (level[y][x].type == START_TILE) {
98                                      marble_state3D->pos3D.x = x;
99                                      marble_state3D->pos3D.y = y;
100                                     marble_state3D->pos3D.z = z;
101             printf("start: %f,%f\n",marble_state3D->pos3D.x,marble_state3D->pos3D.y);
102                             }
103
104             // Move pointer past this cell's closing quote
105             p = endq + 1;
106         }
107     }
108
109     free(line);
110     fclose(fp);
111     return level;
112
113 fail:
114     free(line);
```

```
115     for (int j = 0; j < height; ++j) free(level[j]);
116     free(level);
117     fclose(fp);
118     return NULL;
119 }
```

4.0.5 physics.h

```
1  #ifndef PHYSICS_H_
2  #define PHYSICS_H_
3
4  #include "types.h"
5
6
7  #define GRAVITY 0.1
8  #define BALL_RADIUS 0.5
9  #define UP 1
10 #define DOWN 2
11 #define LEFT 3
12 #define RIGHT 4
13
14 #define TRACKBALL_SENSITIVITY 0.0005
15
16 extern MarbleState3D marble_state3D;
17
18 void apply_impulse(Vec3 impulse, double dt);
19
20 void handle_ramp(int direction, double dt);
21
22 int check_boundaries();
23
24 int check_game_state(Tile cur_tile);
25
26 void end_game();
27
28 void handle_wall(int wall_direction);
29
30 int check_wall(Tile cur_tile);
31 int check_fall(Tile cur_tile);
32
33 int handle_fall(double dt);
34
35 int check_win(Tile cur_tile);
36
37 int check_ramp(Tile cur_tile);
38
39
40 #endif
```

4.0.6 physics.c

```
1  #include "physics.h"
2  #include "vga_marble.h"
3  #include "game_map.h"
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  MarbleState3D marble_state3D = {
8          .pos3D = {0, 0, 0},
```

```
9           .velocity = {0, 0, 0}
10  };
11
12  int on_win()
13  {
14      return 1;
15  }
16
17  void apply_impulse(Vec3 impulse, double dt)
18  {
19      // Update velocity based on impulse an time
20      marble_state3D.velocity.x += impulse.x*TRACKBALL_SENSITIVITY;
21      marble_state3D.velocity.y += impulse.y*TRACKBALL_SENSITIVITY;
22      marble_state3D.velocity.z += impulse.z*TRACKBALL_SENSITIVITY;
23
24
25      // Update position based on new velocity and time
26      marble_state3D.pos3D.x += marble_state3D.velocity.x*TRACKBALL_SENSITIVITY;
27      marble_state3D.pos3D.y += marble_state3D.velocity.y*TRACKBALL_SENSITIVITY;
28      marble_state3D.pos3D.z += marble_state3D.velocity.z*TRACKBALL_SENSITIVITY;
29
30      if(impulse.x != 0 || impulse.y != 0){
31          //printf("Impulse: x: %f, y: %f, z: f\n", impulse.x, impulse.y, impulse.z);
32          //printf("Updated Position: x %f y %f \n", marble_state3D.pos3D.x, marble_state3D.pos3D.y);
33      }
34  }
35
36
37
38  void handle_ramp(int direction, double dt) {
39      printf("THERE IS A RAMP!\n");
40      if (direction == DOWN_X_RAMP) {
41          printf("DOWN X");
42          Vec3 impulse = {0.2*GRAVITY, 0, 0.5*GRAVITY};
43          apply_impulse(impulse,dt);
44      } else if (direction == UP_X_RAMP) {
45          printf("UP X");
46          Vec3 impulse = {−0.2*GRAVITY, 0, 0.5*GRAVITY};
47          apply_impulse(impulse,dt);
48      } else if (direction == DOWN_Y_RAMP) {
49          printf("DOWN Y");
50          Vec3 impulse = {0, 0.2*GRAVITY, 0.5*GRAVITY};
51          apply_impulse(impulse,dt);
52      } else if (direction == UP_Y_RAMP) {
53          printf("UP Y");
54          Vec3 impulse = {0, −0.2*GRAVITY, 0.5*GRAVITY};
55          apply_impulse(impulse,dt);
56      }
57  }
58
59
60  int check_wall(Tile cur_tile) {
61
62      if ((cur_tile.x_idx + BALL_RADIUS > marble_state3D.pos3D.x) && cur_tile.x_idx != 0) {
63          Tile next_tile = levels[current_level][cur_tile.y_idx][cur_tile.x_idx − 1];
64
65          if (next_tile.type <= 7 && next_tile.type >= 3) {
66              return 0;
67          }
68
69          if (next_tile.z_idx < cur_tile.z_idx && marble_state3D.velocity.x < 0) {
70              printf("Left Wall\n");
```

```
71              return LEFT;
72          }
73      } else if ((cur_tile.x_idx + 1 - BALL_RADIUS < marble_state3D.pos3D.x) && cur_tile.x_idx != current_level_rows - 1) {
74          Tile next_tile = levels[current_level][cur_tile.y_idx][cur_tile.x_idx + 1];
75
76          if (next_tile.type <= 7 && next_tile.type >= 3) {
77              return 0;
78          }
79
80          //printf("Cur Pos: %f, %f\n", marble_state3D.pos3D.x, marble_state3D.pos3D.y);
81          //printf("Next Tile: %d, Current Tile: %d\n", next_tile.z_idx, cur_tile.z_idx);
82          //printf("Next Tile: %d %d, Current Tile: %d %d\n", next_tile.x_idx, next_tile.y_idx, cur_tile.x_idx, cur_tile.y_idx);
83          if (next_tile.z_idx < cur_tile.z_idx && marble_state3D.velocity.x > 0) {
84              printf("Right Wall\n");
85              return RIGHT;
86          }
87      } else if ((cur_tile.y_idx + BALL_RADIUS > marble_state3D.pos3D.y) && cur_tile.y_idx != 0) {
88          Tile next_tile = levels[current_level][cur_tile.y_idx - 1][cur_tile.x_idx];
89
90          if (next_tile.type <= 7 && next_tile.type >= 3) {
91              return 0;
92          }
93
94          if (next_tile.z_idx < cur_tile.z_idx && marble_state3D.velocity.y > 0) {
95              printf("Up Wall\n");
96              return UP;
97          }
98
99      } else if ((cur_tile.y_idx + 1 - BALL_RADIUS < marble_state3D.pos3D.y) && cur_tile.y_idx != current_level_cols - 1) {
100         Tile next_tile = levels[current_level][cur_tile.y_idx + 1][cur_tile.x_idx];
101
102         if (next_tile.type <= 7 && next_tile.type >= 3) {
103             return 0;
104         }
105
106         if (next_tile.z_idx < cur_tile.z_idx && marble_state3D.velocity.y < 0) {
107             printf("Down Wall\n");
108             return DOWN;
109         }
110     }
111
112     return 0;
113 }
114
115 int check_fall(Tile cur_tile) {
116
117     if ((cur_tile.x_idx + BALL_RADIUS > marble_state3D.pos3D.x) && cur_tile.x_idx != 0) {
118         Tile next_tile = levels[current_level][cur_tile.y_idx][cur_tile.x_idx - 1];
119
120         if (next_tile.z_idx > cur_tile.z_idx) {
121             printf("Left Fall\n");
122             return 1;
123         }
124     } else if ((cur_tile.x_idx + 1 - BALL_RADIUS < marble_state3D.pos3D.x) && cur_tile.x_idx != current_level_rows - 1) {
125         Tile next_tile = levels[current_level][cur_tile.y_idx][cur_tile.x_idx + 1];
126
127         if (next_tile.z_idx > cur_tile.z_idx) {
128             printf("Right Fall\n");
129             return 1;
130         }
131     } else if ((cur_tile.y_idx + BALL_RADIUS > marble_state3D.pos3D.y) && cur_tile.y_idx != 0) {
132         Tile next_tile = levels[current_level][cur_tile.y_idx-1][cur_tile.x_idx];
```

```
133
134             if (next_tile.z_idx > cur_tile.z_idx) {
135                 printf("Up Fall\n");
136                 return 1;
137             }
138         } else if ((cur_tile.y_idx + 1 - BALL_RADIUS < marble_state3D.pos3D.y) && cur_tile.y_idx != current_level_cols - 1) {
139             Tile next_tile = levels[current_level][cur_tile.y_idx + 1][cur_tile.x_idx];
140             if (next_tile.z_idx > cur_tile.z_idx) {
141                 printf("Down Fall\n");
142                 return 1;
143             }
144         }
145
146         return 0;
147     }
148
149
150
151     int check_game_state(Tile cur_tile){
152         if(cur_tile.type == NO_TILE) {
153             printf("%f, %f, %f\n", marble_state3D.pos3D.x, marble_state3D.pos3D.y, marble_state3D.pos3D.z);
154             printf("On empty tile\n");
155             return LOST;
156         } else if (cur_tile.type == WIN_TILE) {
157             return WON;
158         }
159         return CONTINUE;
160     }
161
162
163     //1 if you have lost, 0 otherwise
164     // int check_game_state(Tile cur_tile) {
165
166     // if (cur_tile.x_idx == 0 && (cur_tile.x_idx + BALL_RADIUS > marble_state3D.pos3D.x)) {
167     // return LOST;
168     // } else if ((cur_tile.x_idx == current_level_cols - 1) && (cur_tile.x_idx + 1 - BALL_RADIUS < marble_state3D.pos3D.x)) {
169     // return LOST;
170     // } else if (cur_tile.y_idx == 0 && (cur_tile.y_idx + BALL_RADIUS > marble_state3D.pos3D.y)) {
171     // return LOST;
172     // } else if ((cur_tile.y_idx == current_level_rows - 1) && (cur_tile.y_idx + 1 - BALL_RADIUS < marble_state3D.pos3D.y)) {
173     // return LOST;
174     // }
175
176     // if (check_win(cur_tile))
177     // {
178     // return WON;
179     // }
180
181     // return CONTINUE;
182     // }
183
184     void handle_wall(int wall_direction) {
185         printf("handling wall\n");
186         if (wall_direction == LEFT || wall_direction == RIGHT)
187         {
188             marble_state3D.velocity.x *= -1;
189         }
190         if (wall_direction == UP || wall_direction == DOWN)
191         {
192             marble_state3D.velocity.y *= -1;
193         }
194     }
```

```
195
196 int check_win(Tile cur_tile)
197 {
198     return cur_tile.type == WIN_TILE;
199 }
200
201 //returns LOST if you lost, WON if you won, CONTINUE otherwise
202 int check_boundaries(double dt) {
203     // fetch current tile
204     Tile cur_tile = check_position(&marble_state3D);
205
206     //printf("current tile type: %d\n", cur_tile.type);
207
208     int game_state = check_game_state(cur_tile);
209     if(game_state == LOST) {
210         //we lost
211         return LOST;
212     }
213     else if (game_state == WON)
214     {
215         return WON;
216     }
217
218     int wall_direction = check_wall(cur_tile);
219
220     if (wall_direction) {
221         handle_wall(wall_direction);
222     }
223
224     int fall = check_fall(cur_tile);
225
226     if (fall) {
227
228         int fall_result = handle_fall(dt);
229
230         if (fall_result == LOST)
231         {
232             printf("fall lost\n");
233             return LOST;
234         }
235     }
236
237     int ramp = check_ramp(cur_tile);
238
239     if (ramp)
240     {
241         handle_ramp(ramp,dt);
242     }
243     return CONTINUE;
244
245 }
246
247 int check_ramp(Tile cur_tile)
248 {
249     if (cur_tile.type == UP_X_RAMP)
250     {
251         return cur_tile.type;
252     }
253     if (cur_tile.type == UP_Y_RAMP)
254     {
255         return cur_tile.type;
256     }
```

```
257     if (cur_tile.type == DOWN_X_RAMP)
258     {
259         return cur_tile.type;
260     }
261     if (cur_tile.type == DOWN_Y_RAMP)
262     {
263         return cur_tile.type;
264     }
265     return 0;
266 }
267
268 int handle_fall(double dt)
269 {
270     printf("In handle fall\n");
271     Tile cur_tile = check_position(&marble_state3D);
272
273     if (cur_tile.type == NO_TILE)
274     {
275         printf("%f, %f, %f\n", marble_state3D.pos3D.x, marble_state3D.pos3D.y, marble_state3D.pos3D.z);
276         printf("on empty tile\n");
277         return LOST;
278     }
279
280     Vec3 impulse = {0, 0, GRAVITY};
281     apply_impulse(impulse, dt);
282     return CONTINUE;
283 }
```

4.0.7 types.h

```
1  #ifndef TYPES_H_
2  #define TYPES_H_
3
4  #define CONTINUE 0
5  #define WON 1
6  #define LOST 2
7
8  typedef struct {
9      double x;
10     double y;
11     double z;
12 } Vec3;
13
14 typedef struct {
15     double x;
16     double y;
17 } Vec2;
18
19 typedef struct {
20     Vec3 pos3D;
21     Vec3 velocity;
22 } MarbleState3D;
23
24 typedef struct {
25     int x_idx;
26     int y_idx;
27     int z_idx;
28     int type;
29 } Tile;
30
```

```
#endif
```

4.0.8 main.c

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <time.h>
#include <unistd.h>

#include "vga_gpu.h"

#include "../trackball/trackball.h"
#include "game_map.h"
#include "level_reader.h"
#include "physics.h"
#include "vga_gpu.h"


/* Read the palette file and return size of the palette! */
void read_and_write_palette(const char *path, int vga_gpu_fd) {
        GPUPaletteArgs palette;

        //printf("Reading palette!\n");
        FILE *fp = fopen(path, "rb");
        if (!fp) return;

        char palette_size;
        if (!fread(&palette_size, 1, 1, fp)) return;

        //printf("Read palette size: %d\n", palette_size);

        ColorRGBA colors;
        for (int i = 0; i < palette_size; i++) {
                palette.ind = i;
                for (int j = 0; j < 16; j++) {
                        if (!fread(&colors, sizeof(ColorRGBA), 1, fp))
                        return;

                        palette.off = j + 1;
                        palette.r = colors.red;
                        palette.g = colors.green;
                        palette.b = colors.blue;
                        palette.a = 0xFF;

                        ioctl(vga_gpu_fd, VGA_GPU_WRITE_PALETTE, &palette);
                }
        }

        //printf("Read & wrote all palette colors\n");

        fclose(fp);
}

void read_and_write_textures(const char*path, int vga_gpu_fd) {
        GPUTextureArgs textures;
        Texture tex;

        //printf("Reading textures!\n");
```

```
57          FILE *fp = fopen(path, "rb");
58          if (!fp) return;
59
60          char num_textures;
61          if (!fread(&num_textures, 1, 1, fp)) return;
62
63          //printf("Read %d tiles!\n", num_textures);
64
65
66          for (int i = 0; i < num_textures; i++) {
67                  if (!fread(&tex, sizeof(Texture), 1, fp)) return;
68
69                  for (int j = 0; j < 8; j++) {
70                          for (int k = 0; k < 8; k++) {
71                                  if(tex.palette_indicies[j][k] == 0 && i == num_textures-1) textures.palette_offs[j][k] = tex.palette_in
72                                  else textures.palette_offs[j][k] = tex.palette_indicies[j][k] + 1;
73                          }
74                  }
75
76                  textures.ind = i;
77
78
79                          //printf("Texture at index i: %d\n", textures.ind);
80                          for(int j = 0; j < 8; j++) {
81                                  for (int k = 0; k < 8; k++) {
82                                          //printf("%d ", textures.palette_offs[j][k]);
83                                  }
84                                  //printf("\n");
85                          }
86
87
88
89
90                  ioctl(vga_gpu_fd, VGA_GPU_WRITE_TEXTURE, &textures);
91          }
92
93          //printf("Read & wrote all textures\n");
94
95          fclose(fp);
96  }
97
98  void read_and_write_tilemap(const char *path, int vga_gpu_fd) {
99          GPUTileArgs map_args;
100
101         //printf("Reading tilemap!\n");
102         FILE *fp = fopen(path, "rb");
103         if (!fp) return;
104
105         char height;
106         char width;
107         if (!fread(&height, 1, 1, fp)) return;
108         if (!fread(&width, 1, 1, fp)) return;
109
110         //printf("Reading tilemap with width: %d and height: %d\n", width, height);
111         char texture_index;
112         for(int y = 0; y < height; y++) {
113                 for(int x = 0; x < width; x++) {
114                         if (!fread(&texture_index, 1, 1, fp)) return;
115
116                         map_args.h_flip = 0;
117                         map_args.v_flip = 0;
118                         map_args.priority = 0;
```

```
119                             map_args.palette_ind = 0;
120                             map_args.texture_ind = texture_index;
121                             //map_args.texture_ind = 0;
122                             map_args.layer = 1;
123                             map_args.x = x;
124                             map_args.y = y;
125
126
127
128                             //printf("Texture Index: %d at x = %d and y = %d\n", map_args.texture_ind, map_args.x, map_args.y);
129
130                             ioctl(vga_gpu_fd, VGA_GPU_WRITE_TILE, &map_args);
131                     }
132             }
133
134             //printf("Finished writing tilemap\n");
135             fclose(fp);
136     }
137
138     char read_ball_sprite(const char *path, int vga_gpu_fd) {
139
140             //printf("Reading Sprites!\n");
141             FILE *fp = fopen(path, "rb");
142             if (!fp) return 0;
143
144             char height;
145             char width;
146             if (!fread(&height, 1, 1, fp)) return 0;
147             if (!fread(&width, 1, 1, fp)) return 0;
148
149             //printf("Reading sprites with width: %d and height: %d\n", width, height);
150             char texture_index;
151             fread(&texture_index, 1, 1, fp);
152
153             fclose(fp);
154
155             //printf("Texture Index: %c", texture_index);
156             return texture_index;
157
158     }
159
160     Vec2 project_3D_to_2D(Vec3 pos3D) {
161             Vec2 pos2D;
162             //printf("Vec3, %f, %f, %f\n", pos3D.x, pos3D.y,pos3D.z);
163             pos2D.x = -(2 * pos3D.y - 2 * pos3D.x) * 4;
164             pos2D.y = (pos3D.x + pos3D.y + 2 * pos3D.z) * 4;
165
166             pos2D.x += 120.0;
167             pos2D.y += 50.0;
168             //printf("here\n");
169             //printf("projected: %f,%f\n",pos2D.x,pos2D.y);
170             return pos2D;
171     }
172
173     Vec3 unproject_2D_to3D(Vec2 pos2D) {
174         Vec3 pos3D;
175
176         float x = (pos2D.x - 136.0) / 2.0;
177         float y = (pos2D.y - 50.0) / 4.0;
178
179         float sum_xy = x / 2.0;
180         float sum_xyz = y;
```

```
181
182     pos3D.z = (sum_xyz − sum_xy) / 2.0;
183
184     // Even split between x and y
185     pos3D.x = sum_xy / 2.0;
186     pos3D.y = sum_xy / 2.0;
187
188     return pos3D;
189 }
190
191 void write_sprite_location(Vec2 state, int vga_gpu_fd, char texture_index) {
192
193         GPUSpriteArgs sprite_args;
194
195         //printf("Reading sprite!\n");
196         sprite_args.x = state.x;
197         sprite_args.y = state.y;
198         sprite_args.ind = 0;
199         sprite_args.texture_ind = texture_index;
200         sprite_args.h_flip = 0;
201         sprite_args.v_flip = 0;
202         sprite_args.palette_ind = 1;
203
204         ioctl(vga_gpu_fd, VGA_GPU_WRITE_SPRITE, &sprite_args);
205
206         //printf("Finished writing sprite\n");
207 }
208
209 void clear_sprite_location(int vga_gpu_fd) {
210
211         GPUSpriteArgs sprite_args;
212
213         //printf("Reading sprite!\n");
214         sprite_args.x = 0;
215         sprite_args.y = 0;
216         sprite_args.ind = 0;
217         sprite_args.texture_ind = 0;
218         sprite_args.h_flip = 0;
219         sprite_args.v_flip = 0;
220         sprite_args.palette_ind = 0;
221
222         ioctl(vga_gpu_fd, VGA_GPU_WRITE_SPRITE, &sprite_args);
223
224         //printf("Finished writing sprite\n");
225 }
226
227 void clear_all_sprites(int vga_gpu_fd) {
228         GPUSpriteArgs sprite = {};
229         sprite.x = 320;
230         for (int i = 0; i < 256; i++)
231                 ioctl(vga_gpu_fd, VGA_GPU_WRITE_SPRITE, &sprite);
232 }
233
234
235
236 // int fd;
237 // void set_gpu_position(Vec2 pos) {
238 // vga_gpu_pos_arg_t pos_arg;
239 // pos_arg.pos.x = pos.x;
240 // pos_arg.pos.y = pos.y;
241
242 // if (ioctl(fd, VGA_gpu_WRITE_POSITION, &pos_arg) < 0) {
```

```
243 // perror("ioctl");
244 // close(fd);
245 // }
246 // }
247
248 // void handle_win() { // jake help
249 // printf("You Won!\n");
250 // }
251
252 // void handle_lose() { printf("You Lost!\n"); }
253
254 // Vec2 project_3D_to_2D(Vec3 pos3D) {
255 // Vec2 pos2D;
256 // pos2D.x = 2 * pos3D.x + 2 * pos3D.y;
257 // pos2D.y = -pos3D.x + pos3D.y + 2 * pos3D.z;
258 // return pos2D;
259 // }
260
261
262 void handle_win()
263 {
264         printf("You Won!\n");
265 }
266
267
268 void handle_lose()
269 {
270         printf("Died on x: %f, y: %f, z: %f",marble_state3D.pos3D.x,marble_state3D.pos3D.y,marble_state3D.pos3D.z);
271         printf("You Lost!\n");
272 }
273
274 int main() {
275         //initialize_levels(&gpu_state3D);
276         //printf("%f, %f, %f\n", gpu_state3D.pos3D.x, gpu_state3D.pos3D.y,
277         //  gpu_state3D.pos3D.z);
278
279         static const char filename[] = "/dev/vga_gpu";
280         static const char palette_file[] = "./levels/level4-palette.bin";
281         static const char textures_file[] = "./levels/level4-textures.bin";
282         static const char tilemap_file[] = "./levels/level4-tilemap.bin";
283         static const char sprite_file[] = "./levels/level4-sprites.bin";
284
285         //static const char palette_file[] = "./levels/level1-palette.bin";
286         //static const char textures_file[] = "./levels/level1-textures.bin";
287         //static const char tilemap_file[] = "./levels/level1-tilemap.bin";
288         //static const char sprite_file[] = "./levels/level1-sprites.bin";
289
290
291         GPUCtrlArgs ctrl;
292
293         int vga_gpu_fd;
294
295         printf("VGA GPU userspace program started\n");
296
297         if ((vga_gpu_fd = open(filename, O_RDWR)) == -1) {
298                 fprintf(stderr, "could not open %s\n", filename);
299                 return -1;
300         }
301
302         ctrl.force_blank = 0;
303         ioctl(vga_gpu_fd, VGA_GPU_WRITE_CTRL, &ctrl);
304
```

```
305         read_and_write_palette(palette_file, vga_gpu_fd);
306         read_and_write_textures(textures_file, vga_gpu_fd);
307         read_and_write_tilemap(tilemap_file, vga_gpu_fd);
308         clear_all_sprites(vga_gpu_fd);
309         //clear_sprite_location(vga_gpu_fd);
310         char texture_index = read_ball_sprite(sprite_file, vga_gpu_fd);
311
312         initialize_levels(&marble_state3D);
313         printf("starting position: %f,%f\n",marble_state3D.pos3D.x,marble_state3D.pos3D.y);
314         Vec2 initial_position_proj = project_3D_to_2D(marble_state3D.pos3D);
315         Vec3 unproject = unproject_2D_to3D(initial_position_proj);
316         printf("%f, %f, %f\n", unproject.x, unproject.y, unproject.z);
317
318
319         write_sprite_location(initial_position_proj, vga_gpu_fd, texture_index);
320
321         if (setupReader() == -1)
322         {
323                 return -1;
324         }
325
326         double dt = 0;
327
328         ctrl.force_blank = 0;
329         ioctl(vga_gpu_fd, VGA_GPU_WRITE_CTRL, &ctrl);
330
331         while(1)
332         {
333                 //add ball read
334                 struct ball_input input = getAccumulatedInput();
335
336                 Vec3 impulse = {input.dx, input.dy, 0};
337                 apply_impulse(impulse, dt);
338
339                 Vec2 current_position_proj = project_3D_to_2D(marble_state3D.pos3D);
340                 write_sprite_location(current_position_proj, vga_gpu_fd, texture_index);
341
342                 int state = check_boundaries(dt);
343
344                 if (state == WON)
345                 {
346                         handle_win();
347                         break;
348                 }
349
350                 if (state == LOST)
351                 {
352                         handle_lose();
353                         break;
354                 }
355
356         }
357
358         //write_sprite_location(vga_gpu_fd);
359
360 // if (setupReader() == -1) {
361 // close(fd);
362 // return -1;
363 // }
364 // time_t start = 0, end = 0;
365 // double dt;
366 // // Vec2 initial_position_proj = project_3D_to_2D(gpu_state3D.pos3D);
```

```
367 // // set_gpu_position(initial_position_proj);
368 // while (1) {
369 // // calculte time of while loop (dt)
370 // // dt = difftime(end, start);
371
372 // // // need trackball input here to define F
373 // // struct ball_input input = getAccumulatedInput();
374
375 // // Vec3 impulse = {input.dx, input.dy, 0};
376 // // apply_impulse(impulse, dt);
377
378 // // // handle game logic here
379
380 // // int state = check_boundaries(dt);
381
382 // // if (state == WON) {
383 // // handle_win();
384 // // break;
385 // // }
386
387 // // if (state == LOST) {
388 // // handle_lose();
389 // // break;
390 // // }
391
392 // // convert 3D state to 2D state to send to hardware
393 // // send ball state to hardware
394 // // Vec2 pos2D = project_3D_to_2D(gpu_state3D.pos3D);
395 // // set_gpu_position(pos2D);
396
397 // int vsync_done = read_vsync();
398 // while (!vsync_done) {
399 // continue;
400 // }
401 // }
402         printf("VGA GPU userspace program terminating\n");
403         //free_levels();
404         close(vga_gpu_fd);
405         return EXIT_SUCCESS;
406 }
407
408 // int main() {
409 // GPUTextureArgs texture;
410 // GPUTileArgs tile;
411 // GPUSpriteArgs sprite;
412 // GPUPaletteArgs palette;
413 // GPUCtrlArgs ctrl;
414
415 // int vga_gpu_fd;
416 // int i, j;
417
418 // static const char filename[] = "/dev/vga_gpu";
419 // static const u8 colors[][4] = {
420 // {0x00, 0x00, 0x00, 0x00}, /* Black */
421 // {0xff, 0x00, 0x00, 0x00}, /* Red */
422 // {0x00, 0xff, 0x00, 0x00}, /* Green */
423 // {0x00, 0x00, 0xff, 0x00}, /* Blue */
424 // {0xff, 0xff, 0x00, 0x00}, /* Yellow */
425 // {0x00, 0xff, 0xff, 0x00}, /* Cyan */
426 // {0xff, 0x00, 0xff, 0x00}, /* Magenta */
427 // {0x80, 0x80, 0x80, 0x00}, /* Gray */
428 // {0xff, 0xff, 0xff, 0x00} /* White */
```

```
// };
// //
// // #define NUM_COLORS 9
// // #define SCREEN_WIDTH 640
// // #define SCREEN_HEIGHT 480

// printf("VGA GPU userspace program started\n");

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

// for (i = 0; i < 8; i++) {
// palette.r = colors[i][0];
// palette.g = colors[i][1];
// palette.b = colors[i][2];
// palette.a = colors[i][3];
// palette.ind = 0;
// palette.off = i;
// ioctl(vga_gpu_fd, VGA_GPU_WRITE_PALETTE, &palette);
// }

// unsigned long long gradient = 0x1234567812345678;
// for (i = 0; i < 8; i++)
// for (j = 0; j < 8; j++)
// texture.palette_offs[i][j] = (gradient >> ((i + j) * 4)) & 0b1111;
// texture.ind = 1;
// ioctl(vga_gpu_fd, VGA_GPU_WRITE_TEXTURE, &texture);

// sprite.ind = 0;
// sprite.texture_ind = 1;
// sprite.palette_ind = 0;
// sprite.x = 4;
// sprite.y = 4;
// ioctl(vga_gpu_fd, VGA_GPU_WRITE_SPRITE, &sprite);

// ctrl.force_blank = 0;
// ioctl(vga_gpu_fd, VGA_GPU_WRITE_CTRL, &ctrl);

// printf("VGA GPU userspace program terminating\n");
// close(vga_gpu_fd);
// return EXIT_SUCCESS;
// }
```

4.1 software_renderer.py

```
import csv
import pygame
import numpy as np
import struct

# level name of csv
LEVEL_NAME = "level1"

 # Offset from the left of the screen
x_offset = 120
y_offset = 50


```

```
14  # Load Images into Pygame
15  def load_image(path, scale = 2):
16      image_tiles = []
17      image_tiles.append(pygame.image.load("./assets/" + path + "_ul.png").convert_alpha())
18      image_tiles.append(pygame.image.load("./assets/" + path + "_ur.png").convert_alpha())
19      image_tiles.append(pygame.image.load("./assets/" + path + "_bl.png").convert_alpha())
20      image_tiles.append(pygame.image.load("./assets/" + path + "_br.png").convert_alpha())
21
22      for i in range(len(image_tiles)):
23          image = image_tiles[i]
24          image = pygame.transform.scale(image, (image.get_width() * scale, image.get_height() * scale))
25          image_tiles[i] = image
26      return image_tiles
27
28  # load the marble sprite
29  def load_sprite(path, scale = 2):
30      marble = pygame.image.load("./assets/" + path).convert_alpha()
31      marble = pygame.transform.scale(marble, (marble.get_width() * scale, marble.get_height() * scale))
32
33      return marble
34
35  # Read the CSV and return a level matrix
36  def read_csv_to_matrix(file_path):
37      matrix = []
38      with open(file_path, newline='') as csvfile:
39          reader = csv.reader(csvfile)
40          for row in reader:
41              matrix.append(row) # Each row is a list of strings
42      return matrix
43
44  # Draw a game tile to the screen with each of its respective tiles
45  def draw_image(screen, image_tiles, position):
46      x, y = position
47      width = image_tiles[0].get_width()
48      for i in range(2):
49          for j in range(2):
50              screen.blit(image_tiles[i*2 + j], (x + j * width, y + i * width))
51
52  # Load the tiles from the folder
53  def load_tiles(scale):
54      block = load_image("tile2", scale)
55      ramp_south_west = load_image("tile3", scale)
56      ramp_south_east = load_image("tile4", scale)
57      pillar = load_image("tile5", scale)
58      ramp_north_east = load_image("tile6", scale)
59      ramp_north_west = load_image("tile7", scale)
60      start = load_image("tile1", scale)
61      end = load_image("tile8", scale)
62
63
64      return [start, block, ramp_south_west, ramp_south_east, pillar, ramp_north_east, ramp_north_west, end], pillar
65
66  # Load the tiles from the folder
67  def load_marble_surface(scale):
68      marble_surface = load_sprite("ball_8.png", scale)
69      return marble_surface
70
71  # Write palette to file
72  '''
73  Header —
74  Byte 1 — Number of 16 color palettes
75  Groups of 4 byte RGBA colors
```

```
76  '''
77  def write_palettes(palettes, path):
78      with open(path, 'wb') as f:
79          f.write(struct.pack('B', len(palettes)))
80          for palette in palettes:
81              write_palette(f, palette)
82
83  def write_palette(f, palette):
84      counter = 0
85      for r, g, b in palette:
86          f.write(struct.pack('BBBB', r, g, b, 0)) # 4 bytes per color
87          counter += 1
88      while counter < 16:
89          f.write(struct.pack('BBBB', 0, 0, 0, 0)) # 4 bytes per color
90          counter += 1
91
92  # Write tilemap to file
93  '''
94  Header—
95  Byte 1 — Height of tilemap
96  Byte 2 — Width of tilemap
97
98  Rest is writing 1 byte per tile index thats an index into the unique_tiles set
99  '''
100 def write_tilemap(tilemap, path):
101     with open(path, 'wb') as f:
102         height = len(tilemap)
103         width = len(tilemap[0]) if height > 0 else 0
104         f.write(struct.pack('B', height))
105         f.write(struct.pack('B', width))
106         for row in tilemap:
107             for item in row:
108                 f.write(struct.pack('B', item))
109
110 # Write tiles to file
111 '''
112 Header —
113 Byte 1 — Number of 8x8 tiles
114
115 The rest of file is chunks of 64 byte chunks 8x8 tiles where each value is an index into the color palette
116
117 '''
118 def write_texture(textures, path):
119     with open(path, 'wb') as f:
120         tiles_len = len(textures)
121         f.write(struct.pack('B', tiles_len)) # header: number of tiles
122         for texture in textures:
123             for row in texture:
124                 for item in row:
125                     f.write(struct.pack('B', item)) # write each palette index as 1 byte
126
127 # convert a surface to a np array of pixels
128 def surface_to_pixel_matrix(surface):
129     surface = surface.convert(24)
130     pixel_array = pygame.surfarray.array3d(surface)
131     return np.transpose(pixel_array, (1, 0, 2))
132
133 # hash the np arrays
134 def hash_array(arr):
135     return hash(arr.tobytes())
136
137 # take in array of pixels (tile) and change it to reference colors in a palette
```

```
138 def quantize_pixels(pixel_array, palette):
139     height, width, _ = pixel_array.shape
140     index_array = np.zeros((height, width), dtype=np.uint8)
141 
142     for y in range(height):
143         for x in range(width):
144             color = tuple(pixel_array[y, x])
145             if color not in palette:
146                 palette.append(color)
147             index_array[y, x] = palette.index(color)
148 
149     return index_array, palette
150 
151 # tile mapify
152 def tile_mapify(screen_surface, tile_width, tile_height, textures):
153     rows = screen_surface.get_height() // tile_height
154     cols = screen_surface.get_width() // tile_width
155 
156     tilemap = []
157     hash_to_index = {}
158     palette = []
159 
160     for y in range(rows):
161         row = []
162         for x in range(cols):
163             # gets a 8x8 subsurface in the screen
164             tile_surface = screen_surface.subsurface(
165                 (x * tile_width, y * tile_height, tile_width, tile_height)
166             )
167 
168             # converts surface to numpy array
169             pixels = surface_to_pixel_matrix(tile_surface)
170             pixels, palette = quantize_pixels(pixels, palette)
171 
172             # hash of the pixels array
173             h = hash_array(pixels)
174 
175             # if the tile is unique add it to the set
176             if h not in textures:
177                 index = len(textures)
178                 textures[h] = pixels.copy()
179                 hash_to_index[h] = index
180 
181             # add index to
182             row.append(hash_to_index[h])
183         # add the row to the tile map
184         tilemap.append(row)
185 
186     # return the tile map and list of unique tiles
187     return tilemap, textures, palette
188 
189 # same thing as tile_mapify but slightly different - last minute addition
190 def marble_mapify(screen_surface, textures):
191     rows = screen_surface.get_height() // 8
192     cols = screen_surface.get_width() // 8
193 
194     tilemap = []
195     hash_to_index = {}
196     palette = []
197 
198     for y in range(rows):
199         row = []
```

```
200             for x in range(cols):
201                 # gets a 8x8 subsurface in the screen
202                 tile_surface = screen_surface.subsurface(
203                     (x * 8, y * 8, 8, 8)
204                 )
205 
206                 # converts surface to numpy array
207                 pixels = surface_to_pixel_matrix(tile_surface)
208                 pixels, palette = quantize_pixels(pixels, palette)
209 
210                 print(pixels)
211 
212                 # hash of the pixels array
213                 h = hash_array(pixels)
214 
215                 # if the tile is unique add it to the set
216                 if h not in textures:
217                     index = len(textures)
218                     textures[h] = pixels.copy()
219                     hash_to_index[h] = index
220 
221                 # add index to
222                 row.append(hash_to_index[h])
223             # add the row to the tile map
224             tilemap.append(row)
225 
226         # return the tile map and list of unique tiles
227         return tilemap, textures, palette
228 
229 
230 def main():
231         # Pygame stuff
232         pygame.init()
233         screen = pygame.display.set_mode((320, 240*2))
234         clock = pygame.time.Clock()
235         running = True
236 
237         # Load the matrix and tiles used for rendering
238         matrix = read_csv_to_matrix("./levels/" + LEVEL_NAME + ".csv")
239         tiles, pillar = load_tiles(1)
240         marble_surface = load_marble_surface(1)
241 
242         # Width and height of each tile to use for math
243         tile_width = tiles[0][0].get_width() * 2
244         tile_height = tiles[0][0].get_height() * 2 // 2
245 
246         one_iter = False
247         # Game Loop
248         while(running and not one_iter):
249             for event in pygame.event.get():
250                 if event.type == pygame.QUIT:
251                     running = False
252 
253             # Fill the background
254             screen.fill("black")
255 
256             rows = len(matrix)
257             cols = len(matrix[0])
258 
259             # Render loop diagonally
260             for diagonal in range(rows+cols):
261                 for i in reversed(range(rows)):
```

```
                    j = diagonal - i

                    # Check the i and j are within the bounds of the matrix
                    if i >= len(matrix) or j >= len(matrix[i]) or j < 0:
                        continue

                    # Each Item is a pair with the z coordinate and the tile number
                    pair = matrix[i][j].split(",")
                    if len(pair) > 1:
                        z = int(pair[0])
                        tile_num = int(pair[1])

                        # if the tile num is any of these, we can skip rendering it
                        if tile_num == 0:
                            continue

                        # Math for transforming the i and j in the matrix to isometric screen coordinates
                        screen_x = (2 * j - 2 * i) * (tile_width // 4) + x_offset
                        screen_y = (i + j + 2 * z) * (tile_height // 2) + y_offset

                        # Draw the columns below the tile we are drawing first from bottom up
                        number_to_draw = (480-screen_y) // (tile_height)
                        for k in reversed(range(number_to_draw)):
                            if k == 0:
                                break
                            pillar_x = screen_x
                            pillar_y = screen_y+k*tile_height
                            draw_image(screen, pillar, (pillar_x, pillar_y))

                        # Drawing the tile
                        draw_image(screen, tiles[tile_num-1], (screen_x, screen_y))

        # Draw to the screen and time 60 fps
        textures = {}
        tilemap, textures, palette = tile_mapify(screen, 8, 8, textures)
        marble_map, textures, marble_palette = marble_mapify(marble_surface, textures)

        # list the final textures
        textures = list(textures.values())

        # write everything to binary files
        write_texture(textures, LEVEL_NAME+"-textures.bin")
        write_palettes([palette, marble_palette], LEVEL_NAME+"-palette.bin")
        write_tilemap(tilemap, LEVEL_NAME+"-tilemap.bin")
        write_tilemap(marble_map, LEVEL_NAME+"-sprites.bin")

        # only one loop iter
        one_iter = True

        pygame.display.flip()
        clock.tick(60)

    # Clean up
    pygame.quit()

if __name__ == "__main__":
    main()
```

References