

# Final Report

Natalie Hughes, Qian Zhao, Shiyang Wang

May 2024

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Game Overview . . . . .	2
1.2	System Architecture . . . . .	2
<b>2</b>	<b>Hardware</b>	<b>4</b>
2.1	Graphics . . . . .	4
2.2	Memory . . . . .	6
2.3	Audio . . . . .	7
<b>3</b>	<b>Software</b>	<b>7</b>
3.1	User Input . . . . .	7
3.2	Game Logic . . . . .	7
3.2.1	Player Movement . . . . .	9
3.2.2	Bomb Placement and Explosion . . . . .	9
3.2.3	Power-Ups . . . . .	9
3.2.4	Collision Detection . . . . .	11
3.2.5	End Game . . . . .	11
3.3	Hardware-Software Interfacing . . . . .	11
<b>4</b>	<b>Discussion</b>	<b>12</b>
4.1	Challenges . . . . .	12
4.2	Lessons Learned . . . . .	13
4.3	Who did what section? . . . . .	13
<b>5</b>	<b>References</b>	<b>13</b>
<b>A</b>	<b>vga_ball.sv</b>	<b>14</b>
<b>B</b>	<b>vga.c</b>	<b>24</b>
<b>C</b>	<b>hello.h</b>	<b>29</b>
<b>D</b>	<b>controller.h</b>	<b>29</b>
<b>E</b>	<b>bomberman.h</b>	<b>30</b>
<b>F</b>	<b>controller.c</b>	<b>32</b>
<b>G</b>	<b>bomberman.c</b>	<b>36</b>

H main.c 62

I Makefile 64

# 1 Introduction

## 1.1 Game Overview

”Bomberman” is an iconic 2D multiplayer game where two players face off in explosive battles set within intricate grid-based mazes, developed by Hudson Soft. Each player controls their own Bomberman character, tasked with strategically placing bombs to blast through destructible obstacles and outmaneuver their opponent. The maze features both destructible and indestructible wall blocks, adding strategic depth as players navigate the maze and plan their attacks carefully. As bombs detonate, they unleash explosions that can trap or eliminate the opposing player. The game ends when one player successfully eliminates the other by trapping them with bomb explosions or other means.

## 1.2 System Architecture

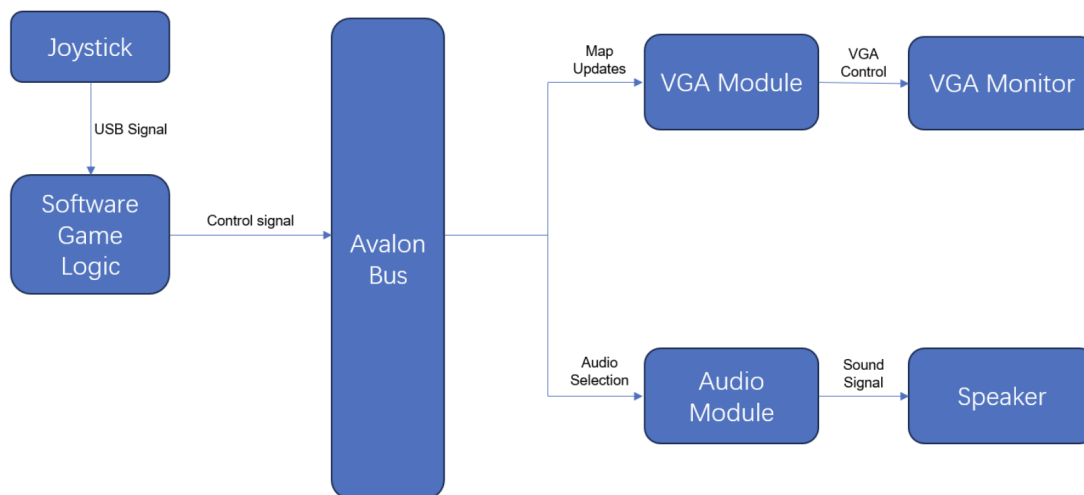


Figure 1: System Diagram

In the Bomberman game implementation on an FPGA platform, players use USB controllers to manage the movement of their respective character. These controllers interface with the game logic software via the USB protocol, utilizing the libusb library for communication. The primary game logic is responsible for handling gameplay elements such as player movement, bomb placement, explosions, and scoring. The ‘controller.c’ file manages the initialization and recognition of inputs from the USB controllers, ensuring they are properly translated into actions within the game.

The software communicates with the FPGA hardware through the ‘vga.c’ device driver. This driver uses the Avalon bus interface to relay control signals to the FPGA, which then updates the graphics on the VGA monitor. The FPGA hardware setup includes on-chip memory ROMs for storing sprite data necessary for the game’s graphics. The ‘vga display.sv’ retrieves this sprite data based on the game’s logic and outputs it to the VGA monitor, ensuring that the graphics displayed are accurate

and timely. Additionally, the audio cues are managed through the WM8731 CODEC, which outputs sound to connected earbuds or speakers.

This setup involves a seamless interaction between software and hardware components. The software processes USB controller inputs, updates the game state, and sends corresponding control signals to the FPGA. The FPGA, in turn, processes these signals to manage the display output and audio cues. Registers within the FPGA, updated by the game logic, track player status, explosions, bomb placements, and destroyed walls. These registers ensure that the game rules, such as collisions and movements, are consistently applied. This integration allows for a smooth and interactive gaming experience on the FPGA platform, effectively combining user inputs, game logic, and hardware processing.

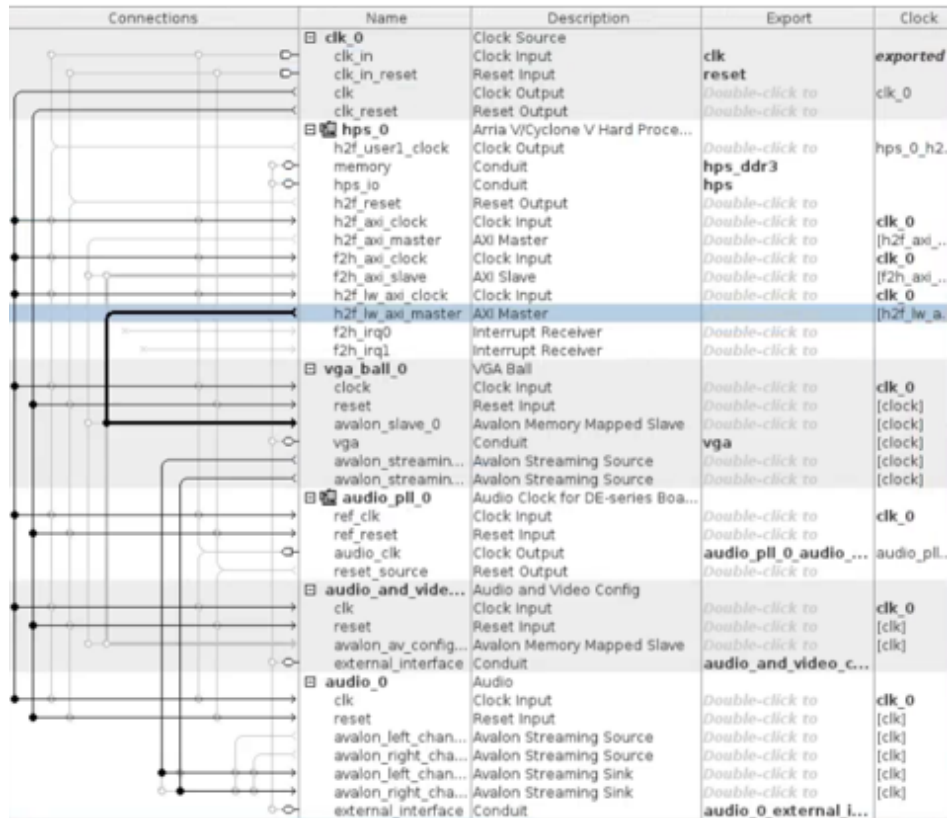


Figure 2: qsys-connection

In detail, the qsys connection is shown in Fig.2. The audio\_0 is a default Intel FPGA IP core to feed data to codec and the task of the vga\_ball is to give the correct data to audio\_0 through the streaming source signal. The streaming source signal outputs a valid bit and a 16-bit data signal and its input is a ready signal. The audio\_pll part is an Intel IP Core which outputs a 12.288MHz clock signal with 50MHz clock as reference clock. This clock is used to drive the codec. The audio\_and\_video\_config is an Intel IP core which includes several IP signal to configure the codec, including telling the codec the data width, frequency and encoding. The vga\_ball module will also be in charge of the video output.

## 2 Hardware

### 2.1 Graphics

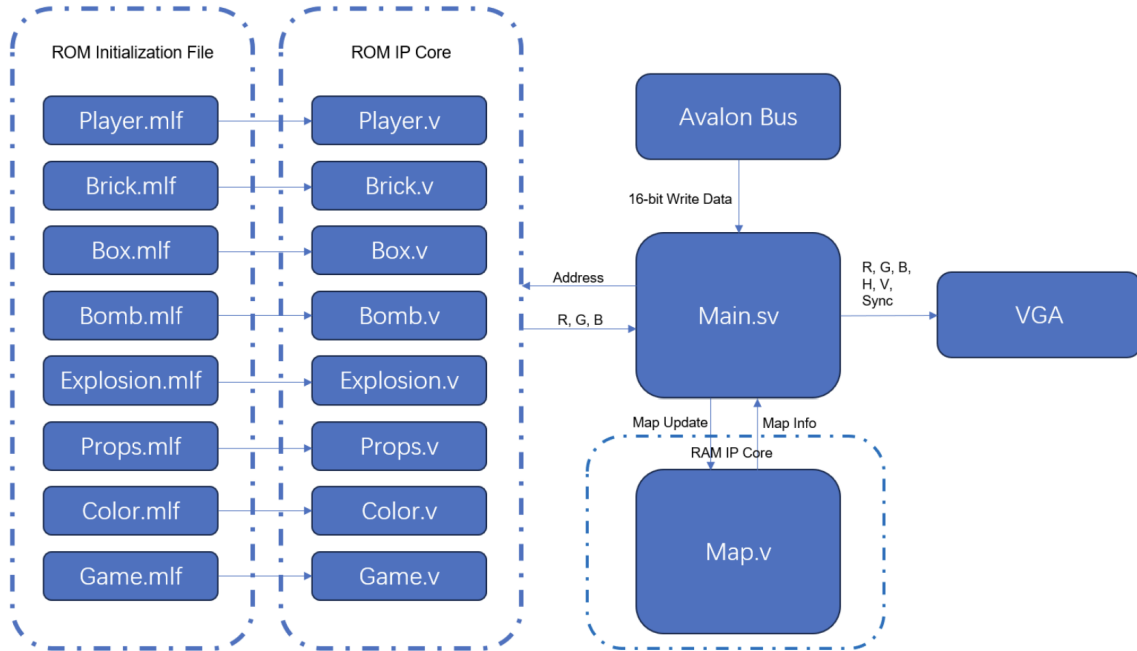


Figure 3: VGA Module Diagram

The ‘vga display.sv’ implements a VGA display module for an FPGA, driving the display output and handling game graphics rendering for a Bomberman-like game. It utilizes various ROM modules to fetch sprite data for players, background, stones, and boxes, which are then mapped to appropriate pixel positions on the screen based on the current horizontal (‘hcount’) and vertical (‘vcount’) counters. The display module also manages player positions and their respective sprites, ensuring accurate rendering of moving and stationary players. The ‘vga counter’ submodule generates synchronization signals for the VGA display, controlling the timing of pixel updates. The module handles game logic and interactions, updating map changes and player states based on input signals, ultimately outputting the final VGA signals for display.

Figure 3 illustrates the block diagram of the VGA module. The map is represented in the ‘map.v’ file, which corresponds to a 1200-row by 8-bit RAM block (only 3 of 8 are used). The 1200 rows indicate that the map consists of 40 x 30 blocks, each sized at 16 x 16 pixels, and fitting within a 640 x 480 resolution. The 3-bit width allows up to eight different elements to be represented in the map, such as background, non-removable bricks, removable bricks, bombs, explosion effects, and other props.

These blocks are stored in ROM files where each block is graphically represented by 16 x 16 x 2 bits, which allows for a maximum of four colors per block. The game supports up to 256 colors in total, so we have a color map of 4 x 8 bits. First, the VGA module translates the row and column numbers of each pixel to determine the block type and specific position within the block. It then uses the block type to locate the relevant ROM and the position information to index into the block’s graphical data so it can retrieve the 2-bit color. Simultaneously, this 2-bit color is translated into an 8-bit color using the color map in the same ROM. Finally, the 8-bit color is used to index into the color ROM to obtain the corresponding RGB values for display.

Rendering the player follows a similar process but uses the player's central pixel location. Each player has four facing directions: four sets of 16 x 16 x 2-bit graphical representations and four sets of 8 x 2-bit color maps, for both players. The VGA module manages two display layers: the top layer for players and the bottom layer for the map.

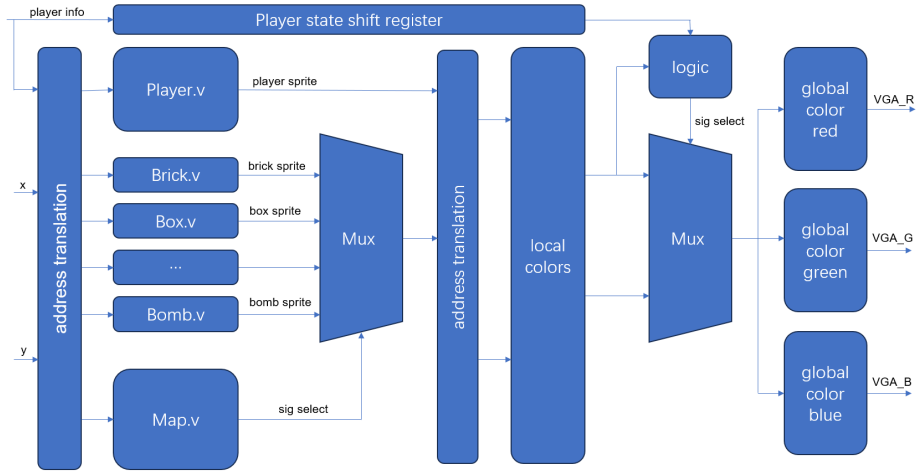


Figure 4: Detailed Connection

The detail memory connection for the vga display part is shown in Figure 4. It can be seen that the display section works in a pipeline structure. The x and y axis are translated into relative position within one block and used as input to index into the tile ROMs as well as the map RAM. In the next stage, the outputs of tile ROMs are connected to a MUX, with the map output as the selection signal. The similar procedure happens to the player, but the player did not need a selection signal and a MUX. After that the tile outputs are translated into the ROM address again and indexed into the local color map. The local color map needs two ports, one for the player and the other for the background tile. The outputs are then selected by the player states and the players local color. If the player's local color is "white", which means transparent and the pixel is within the collision box of a player, it will display the background. If the pixel is within the collision box of one player and the player and not transparent it will display the player pixel. Otherwise it will display the background anyway. After that the output of the MUX is indexed into the RGB ROM to finally get the RGB color. Note that the game rule does not allow the two players to overlap and a software collision logic is used to detect that. If the overlapping does happen, the hardware will take display player1 on the top and the display is not totally correct.

## 2.2 Memory







Category	Graphic	Size(bits)	Num Images	Total(bits)
Box		16*16	1	256
Stone		16*16	1	256
Map	N/A	40*30	1	1200
Player (x2)		16*16	7	3584
Bomb		16*16	1	256
Explosion		16*16	4	1024
Props		16*16	3	768
Audio	N/A	12109*16	1	193744
Total				201,088

Figure 5: Memory Table

Figure 5 includes the components and specifications breakdown for the Bomberman game assets, including graphics for the player characters, bombs, explosions, power-ups, and audio, displaying their bit size and quantity. The FPGA has a total of 4450 Kbits of memory, and we are well within that range. The map is comprised of a single background image of grass, and the obstacles are made up of stone blocks and box blocks. The players are comprised of 7 images, for walking and facing each direction. There is 1 image for the bomb, and the 4 for the explosion, including when an explosion breaks a box. There are 3 props for powerups. Lastly, our audio took up most of the memory. Overall, we used about 201 Kbits of memory.

## 2.3 Audio

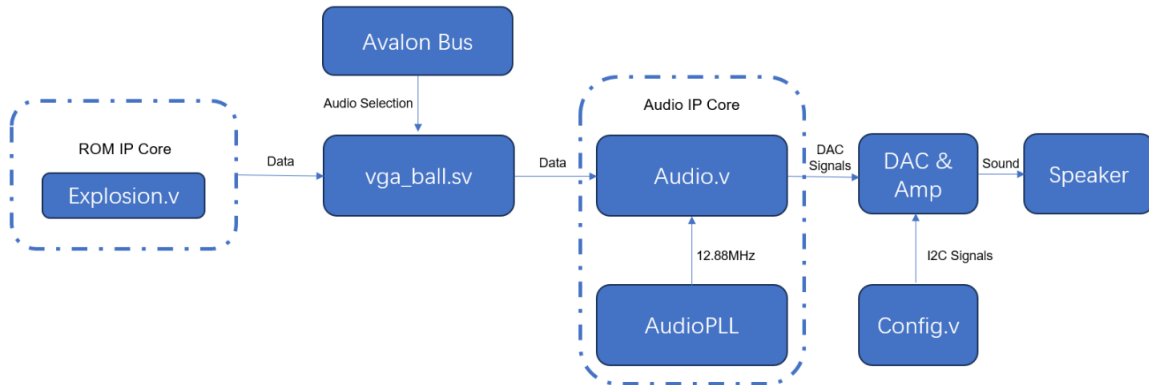


Figure 6: Audio

The connection of the audio part is introduced in the qsys-connection part above. When a audio event triggers, it will set the 17th register to 1, which will start a counter, which validate the data output of the streaming sources and start reading from the ROM.

## 3 Software

### 3.1 User Input

The code uses USB game controllers identified by an idProduct of 17, each featuring a single interface. Players control their character using the gamepad arrows and place bombs using the A button. Counter variables are implemented to debounce switches, ensuring only single presses are registered. Both controllers are handled in a single loop for efficient input processing without requiring separate threads. Two controllers are accommodated with 'libusb interrupt transfer' sequentially reading their 7-byte protocol messages into structures. These structures are later processed in the game loop to discern player movement and bomb placement. The code 'controller.c' is designed to manage USB game controllers using the 'libusb' library. It consists of several key functions that collectively handle the detection and interaction with game controllers, as well as the interpretation of button presses. The code initializes and searches for USB controllers with a specific product ID. It then reads input from the controllers to detect player actions, such as movement and bomb placement. This input processing occurs in a loop, enabling real-time responsiveness to controller inputs. The program also manages the connection and configuration of the controllers, ensuring proper communication for input detection.

### 3.2 Game Logic

After setting up the game environment, the main game loop continuously processes player movement, bomb placement, power-up collection, and updates the game state accordingly. It also handles bomb detonation, explosion propagation, and synchronization of map changes with the display hardware. The loop continues until one of the players dies, at which point it displays the game over screen and cleans up resources.



Figure 7: Controller

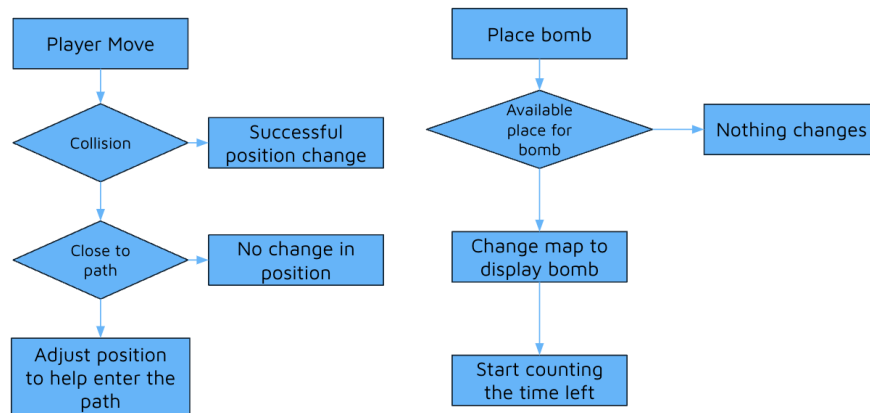


Figure 8: Game Logic



### 3.2.1 Player Movement

Most of the player movement is implemented in the 'handle player movement' function in 'bomberman.c'. It first retrieves the current virtual positions of both players, then calculates the virtual positions based on the players' directions and speeds. The function checks for obstacles and collisions surrounding each player's attempted position, and adjusts the movement to avoid collisions with any blocks. It also detects collisions with explosions and determines the players' face-direction based on their movement direction. If a player attempts to move beyond the game boundaries or collides with the opposing player, their movement is restricted, and their pose is updated to match their facing direction.

### 3.2.2 Bomb Placement and Explosion

There are several bomb-handling functions that manage their behavior within the game environment. When a bomb is placed by a player, it is added to a linked list of bombs, each bomb having a countdown timer indicating its time until explosion. The 'handle bomb explode' function iterates through this list, decrementing the timer for each bomb. Once a bomb's timer reaches zero, it triggers an explosion. If the bomb is the head of the list, its explosion is started, and the bomb is removed from the list. Otherwise, the previous bomb's 'next' pointer is adjusted to skip the exploded bomb, and the bomb is removed from the list. Additionally, the function updates the player's remaining bomb count. The 'free bombs' function is responsible for freeing the memory allocated for the bombs once they have exploded or been removed from the game.

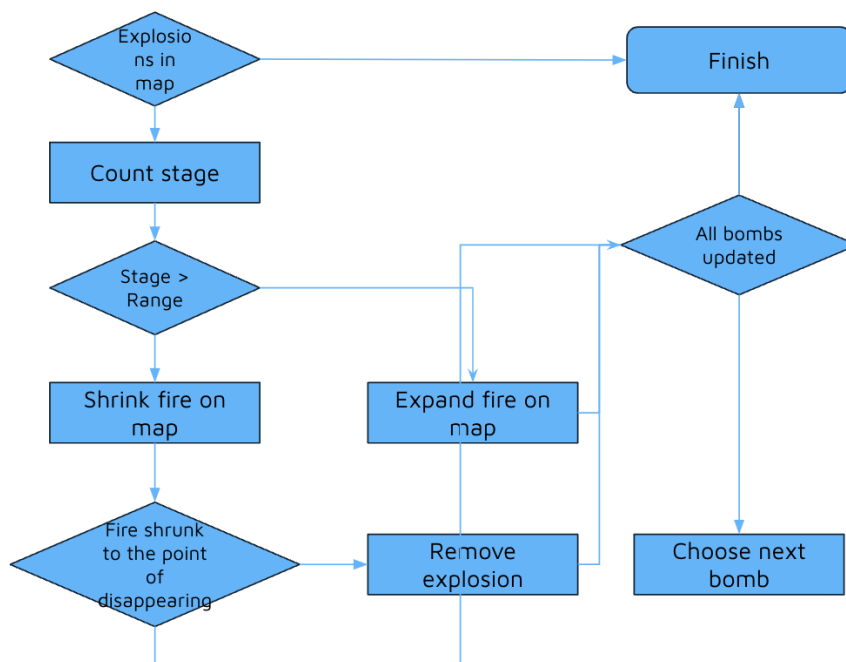


Figure 9: Explosion Game Logic

### 3.2.3 Power-Ups

The 'player get prop' function in bomberman.c is responsible for checking if a player is positioned on a tile containing a power-up. It first converts the player's position from pixel coordinates to

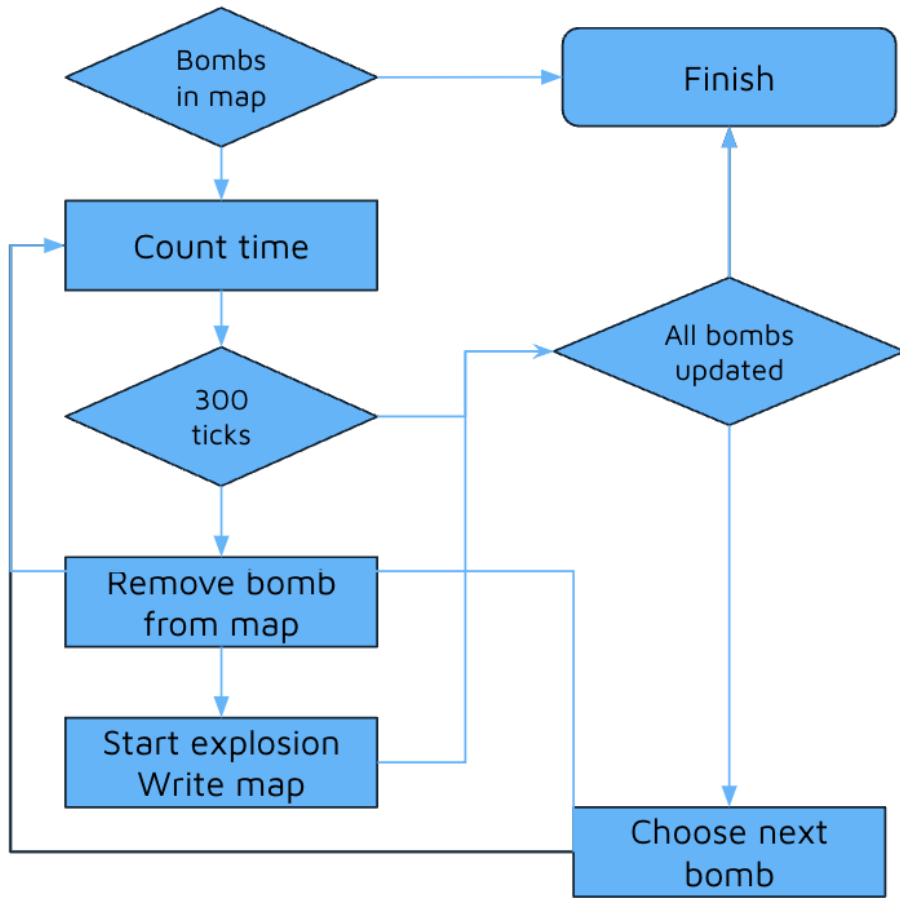


Figure 10: Bomb Game Logic

tile coordinates and then calculates the index of the tile on which the player is positioned. If the tile contains a power-up represented by specific values in the map array, the function processes the corresponding power-up effect. The power-ups include increasing the maximum number of bombs a player can place (max bombs), extending the blast range of bombs (bomb range), and increasing the player's vertical speed (vspeed). After applying the power-up effect, the tile is updated to a standard ground tile (0).

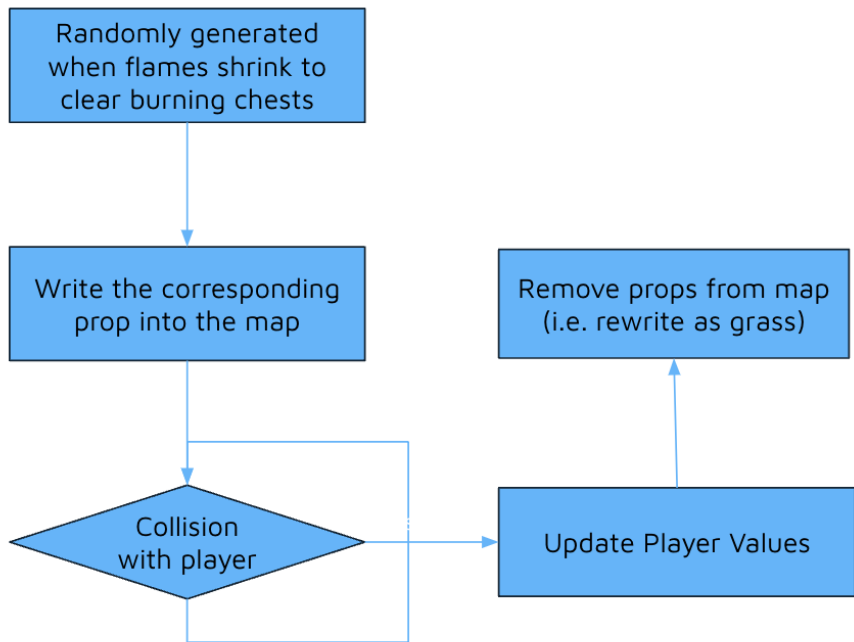


Figure 11: Power-Up Logic

### 3.2.4 Collision Detection

To determine collisions between a player and a bomb explosion, it calculates the grid coordinates of the corners of the explosion based on the position of the player, then checks the tiles at those coordinates on the map. If those coordinates match up, then there has been a player collision with an explosion and that player is eliminated.

### 3.2.5 End Game

The game ends when a player collides with the flame and is eliminated, and the other player wins. Players dying at the same time is a tie.

## 3.3 Hardware-Software Interfacing

There are 18 16-bit registers used to handle graphics and audio. The interface is shown in figure 11.

Register	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	P0_pose					P0_moving	P0_x										
1						P0_facing		P0_y									
2	P1_pose					P1_moving	P1_x										
3						P1_facing		P1_y									
4	en	Tile_type				Position											
5	en	Tile_type				Position											
6	en	Tile_type				Position											
7	en	Tile_type				Position											
8	en	Tile_type				Position											
9	en	Tile_type				Position											
10	en	Tile_type				Position											
11	en	Tile_type				Position											
12	en	Tile_type				Position											
13	en	Tile_type				Position											
14	en	Tile_type				Position											
15	en	Tile_type				Position											
16	en	Tile_type				Position											
17																	Audio_play

Figure 12: Register Interface

Player Registers (0-3): These registers manage the state, position, and movement details of two players. For each player, there are fields to track their pose, movement status, X and Y coordinates, and facing direction.

Tile Registers (4-16): These registers are used to handle environmental tiles within the game. Each tile has an enable flag, a type, and a position which collectively describe its characteristics and location in the game world.

Audio Register (17): This register includes a control bit to manage audio playback within the system.

## 4 Discussion

### 4.1 Challenges

For hardware, the video part includes the use of IP core, for example, the ROM IP core. The ROM is a synchronous ROM, which needs two cycles to actually output the data, so the pipeline should be designed carefully.

Generation of the sprites and tiles is time consuming, a lot of simulation should be done to ensure the correctness and the expected effect of display.

The audio part needs to be carefully configured in qsys for the audio to actually play.

## 4.2 Lessons Learned

A careful design can make the life much easier. Always try to solve the problem in the design instead of trying to solve it during implementation.

Try to split the project into independent parts. This can shorten the testing cycle. For example, compiling the hardware is quite time consuming, so, separating the test of hardware and software will make the debugging smoother.

## 4.3 Who did what section?

Qian Zhao: Hardware, part of the control and game logic.

Shiyan Wang: Collision detection and path fitting, debugging.

Natalie Hughes: Controller logic, player movement logic and documentation.

## 5 References

Part of audio code and part of the software code are from tank project Spring 2023

## A vga\_ball.sv

```
1      /*
2      * Avalon memory-mapped peripheral that generates VGA
3      *
4      * Stephen A. Edwards
5      * Columbia University
6      */
7
8      module vga_ball(input logic    clk,
9                    input logic    reset,
10                   input logic [15:0] writedata,
11                   input logic    write,
12                   input          chipselect,
13                   input logic [5:0] address,
14
15                   output logic [7:0] VGA_R, VGA_G, VGA_B,
16                   output logic    VGA_CLK, VGA_HS, VGA_VS,
17                   output logic    VGA_BLANK_n,
18                   output logic    VGA_SYNC_n,
19                   input L_READY,
20                   input R_READY,
21                   output logic [15:0] L_DATA,
22                   output logic [15:0] R_DATA,
23                   output logic L_VALID,
24                   output logic R_VALID
25                );
26
27    logic [7:0] local_pixel;
28    logic [7:0] idle_down_sprite_raw;
29    logic [7:0] background_sprite_raw;
30    logic [7:0] stone_sprite_raw;
31    logic [7:0] box_sprite_raw;
32    logic [7:0] bomb_sprite_raw;
33    logic [7:0] flamed_box_sprite_raw;
34    logic [7:0] flame_middle_sprite_raw;
35    logic [7:0] flame_h_sprite_raw;
36    logic [7:0] flame_v_sprite_raw;
37    logic [7:0] pickup_bi_sprite_raw;
38    logic [7:0] pickup_fi_sprite_raw;
39
40
41
42    logic [7:0] map_sig_select_raw;
43    logic [3:0] map_sig_select;
44
45    logic [1:0] background_sprite;
46    logic [1:0] stone_sprite;
47    logic [1:0] box_sprite;
48    logic [1:0] bomb_sprite;
49    logic [1:0] flamed_box_sprite;
50    logic [1:0] flame_middle_sprite;
51    logic [1:0] flame_h_sprite;
52    logic [1:0] flame_v_sprite;
53    logic [1:0] pickup_bi_sprite;
54    logic [1:0] pickup_fi_sprite;
```

```

55     logic [1:0] pickup_su_sprite;
56
57
58     logic [1:0] map_sprite;
59     logic [10:0] hcount;
60     logic [9:0] vcount;
61     logic [1:0] sprite_offset_temp; //Decides which 2 bits in the 1 byte sprite_info
62     logic [1:0] sprite_offset;
63     logic [7:0] map_local_color_addr;
64     logic [7:0] global_color_addr;
65     logic [10:0] map_read_addr;
66
67     /*Registers for player0*/
68     logic [15:0] player_info_00; // moving[10] + xinfo[9:0]
69     logic [15:0] player_info_01; // facing[10:9] + yinfo[8:0]
70     /*Registers for player1*/
71     logic [15:0] player_info_10; // same format
72     logic [15:0] player_info_11;
73     logic [15:0] map_change_0;
74     logic [15:0] map_change_1;
75     logic [15:0] map_change_2;
76     logic [15:0] map_change_3;
77     logic [15:0] map_change_4;
78     logic [15:0] map_change_5;
79     logic [15:0] map_change_6;
80     logic [15:0] map_change_7;
81     logic [15:0] map_change_8;
82     logic [15:0] map_change_9;
83     logic [15:0] map_change_10;
84     logic [15:0] map_change_11;
85     logic [15:0] audio_play_0;
86
87
88     /*The variable to determine whether the pixel belongs to player 0 and
89         corresponding shift register*/
90     logic is_player_0;
91     logic is_player_1;
92     logic is_player;
93     logic [3:0] is_player_sr;
94
95     /*Whether the player is moving*/
96     logic moving;
97
98     /*shift register for facing*/
99     logic [1:0] facing;
100
101     logic [10:0] player_addr_player;
102     logic [10:0] player_addr_moving;
103     logic [10:0] player_addr_facing;
104     logic [10:0] player_addr_sprite;
105     logic [10:0] player_addr_local;
106     logic [10:0] player_addr;
107
108     logic [7:0] player_sprite_raw;
109     logic [1:0] player_sprite;
110     logic [3:0] player_offset_sr;

```

```

110 logic [7:0] player_color_addr;
111 logic [15:0] player_color_addr_sr;
112 logic [7:0] player_local_color_addr;
113
114 logic [7:0] upper_layer_local_color_addr;
115 logic [7:0] lower_layer_local_color_addr;
116 logic [7:0] upper_layer_global_color_addr;
117 logic [7:0] lower_layer_global_color_addr;
118
119 logic map_write;
120 logic [10:0] map_write_addr;
121 logic [7:0] map_write_data;
122
123 logic [13:0] audio_explosion_addr;
124 logic [13:0] audio_divider;
125 logic [15:0] audio_out;
126 logic [1:0] audio_valid_sr;
127 logic audio_start;
128
129
130
131
132 parameter PLAYER_OFFSET = 11'd640,
133             MOVING_OFFSET = 11'd192,
134             IDLE_SIDE_OFFSET = 11'd64,
135             IDLE_UP_OFFSET = 11'd128,
136             MOVING_SIDE_OFFSET = 11'd128,
137             MOVING_UP_OFFSET = 11'd320,
138             PLAYER_COLOR_OFFSET = 8'd44;
139
140 assign R_DATA = audio_out;
141 assign L_DATA = audio_out;
142 assign L_VALID = audio_valid_sr[1];
143 assign R_VALID = audio_valid_sr[1];
144
145
146 assign player_addr = player_addr_player + player_addr_moving +
147                    player_addr_facing + player_addr_sprite + player_addr_local[7:2];
148
149 assign is_player_0 = (player_info_00[9:0] != 10'd0) && (hcount[10:1] >=
150                    (player_info_00[9:0] - 10'd7)) && (hcount[10:1] <= (player_info_00[9:0] +
151                    10'd8))
152                    && (vcount[8:0] >= (player_info_01[8:0] - 9'd7)) &&
153                    (vcount[8:0] <= (player_info_01[8:0] + 9'd8));
154
155 assign is_player_1 = (player_info_10[9:0] != 10'd0) && (hcount[10:1] >=
156                    (player_info_10[9:0] - 10'd7)) && (hcount[10:1] <= (player_info_10[9:0] +
157                    10'd8))
158                    && (vcount[8:0] >= (player_info_11[8:0] - 9'd7)) &&
159                    (vcount[8:0] <= (player_info_11[8:0] + 9'd8));
160
161 assign is_player = is_player_0 || is_player_1;
162 assign upper_layer_local_color_addr[7:0] = player_local_color_addr[7:0];
163 assign lower_layer_local_color_addr[7:0] = map_local_color_addr[7:0];

```



```

159
160 assign map_write_data[7:4] = 4'd0;
161
162
163 vga_counters count(.clk50(clk),
164                   .hcount(hcount),
165                   .vcount(vcount),
166                   .VGA_CLK(VGA_CLK),
167                   .VGA_HS(VGA_HS),
168                   .VGA_VS(VGA_VS),
169                   .VGA_BLANK_n(VGA_BLANK_n),
170                   .VGA_SYNC_n(VGA_SYNC_n));
171
172 //Decide which pixel to look at
173 assign local_pixel[7:4] = vcount[3:0];
174 assign local_pixel[3:0] = hcount[4:1];
175
176 ram_empty_map empty_map_ram(.clock(clk),
177                             .data(map_write_data),
178                             .rdaddress(map_read_addr),
179                             .wraddress(map_write_addr),
180                             .wren(map_write),
181                             .q(map_sig_select_raw));
182
183
184
185 rom_players players_rom(.address(player_addr),
186                         .clock(clk),
187                         .q(player_sprite_raw));
188
189 rom_background background_rom(.address(local_pixel[7:2]),
190                               .clock(clk),
191                               .q(background_sprite_raw));
192
193 rom_stone stone_rom(.address(local_pixel[7:2]),
194                    .clock(clk),
195                    .q(stone_sprite_raw));
196
197 rom_box box_rom(.address(local_pixel[7:2]),
198                .clock(clk),
199                .q(box_sprite_raw));
200
201 rom_bomb bomb_rom(.address(local_pixel[7:2]),
202                  .clock(clk),
203                  .q(bomb_sprite_raw));
204
205 rom_flamed_box flamed_box_rom(.address(local_pixel[7:2]),
206                               .clock(clk),
207                               .q(flamed_box_sprite_raw));
208
209 rom_flame_middle flame_middle_rom(.address(local_pixel[7:2]),
210                                   .clock(clk),
211                                   .q(flame_middle_sprite_raw));
212
213 rom_flame_h flame_h_rom(.address(local_pixel[7:2]),
214                          .clock(clk),

```

```

215         .q(flame_h_sprite_raw));
216
217 rom_flame_v flame_v_rom(.address(local_pixel[7:2]),
218         .clock(clk),
219         .q(flame_v_sprite_raw));
220
221 rom_pickup_bi pickup_bi_rom(.address(local_pixel[7:2]),
222         .clock(clk),
223         .q(pickup_bi_sprite_raw));
224
225
226 rom_pickup_fi pickup_fi_rom(.address(local_pixel[7:2]),
227         .clock(clk),
228         .q(pickup_fi_sprite_raw));
229
230 rom_pickup_su pickup_su_rom(.address(local_pixel[7:2]),
231         .clock(clk),
232         .q(pickup_su_sprite_raw));
233
234 audio_tank_explosion
235     audio_explosion_rom(.address(audio_explosion_addr), .clock(clk), .q(audio_out));
236
237 assign map_sig_select[3:0] = map_sig_select_raw[3:0];
238
239 always_comb begin
240     case (sprite_offset)
241     2'd0 : begin
242         background_sprite[1:0] = background_sprite_raw[1:0];
243         stone_sprite[1:0] = stone_sprite_raw[1:0];
244         box_sprite[1:0] = box_sprite_raw[1:0];
245         bomb_sprite[1:0] = bomb_sprite_raw[1:0];
246         flamed_box_sprite[1:0] = flamed_box_sprite_raw[1:0];
247         flame_middle_sprite[1:0] = flame_middle_sprite_raw[1:0];
248         flame_h_sprite[1:0] = flame_h_sprite_raw[1:0];
249         flame_v_sprite[1:0] = flame_v_sprite_raw[1:0];
250         pickup_bi_sprite[1:0] = pickup_bi_sprite_raw[1:0];
251         pickup_fi_sprite[1:0] = pickup_fi_sprite_raw[1:0];
252         pickup_su_sprite[1:0] = pickup_su_sprite_raw[1:0];
253     end
254     2'd1 : begin
255         background_sprite[1:0] = background_sprite_raw[3:2];
256         stone_sprite[1:0] = stone_sprite_raw[3:2];
257         box_sprite[1:0] = box_sprite_raw[3:2];
258         bomb_sprite[1:0] = bomb_sprite_raw[3:2];
259         flamed_box_sprite[1:0] = flamed_box_sprite_raw[3:2];
260         flame_middle_sprite[1:0] = flame_middle_sprite_raw[3:2];
261         flame_h_sprite[1:0] = flame_h_sprite_raw[3:2];
262         flame_v_sprite[1:0] = flame_v_sprite_raw[3:2];
263         pickup_bi_sprite[1:0] = pickup_bi_sprite_raw[3:2];
264         pickup_fi_sprite[1:0] = pickup_fi_sprite_raw[3:2];
265         pickup_su_sprite[1:0] = pickup_su_sprite_raw[3:2];
266     end
267     2'd2 : begin
268         background_sprite[1:0] = background_sprite_raw[5:4];
269         stone_sprite[1:0] = stone_sprite_raw[5:4];
270         box_sprite[1:0] = box_sprite_raw[5:4];

```

```

270     bomb_sprite[1:0] = bomb_sprite_raw[5:4];
271     flamed_box_sprite[1:0] = flamed_box_sprite_raw[5:4];
272     flame_middle_sprite[1:0] = flame_middle_sprite_raw[5:4];
273     flame_h_sprite[1:0] = flame_h_sprite_raw[5:4];
274     flame_v_sprite[1:0] = flame_v_sprite_raw[5:4];
275     pickup_bi_sprite[1:0] = pickup_bi_sprite_raw[5:4];
276     pickup_fi_sprite[1:0] = pickup_fi_sprite_raw[5:4];
277     pickup_su_sprite[1:0] = pickup_su_sprite_raw[5:4];
278     end
279     2'd3 : begin
280         background_sprite[1:0] = background_sprite_raw[7:6];
281         stone_sprite[1:0] = stone_sprite_raw[7:6];
282         box_sprite[1:0] = box_sprite_raw[7:6];
283         bomb_sprite[1:0] = bomb_sprite_raw[7:6];
284         flamed_box_sprite[1:0] = flamed_box_sprite_raw[7:6];
285         flame_middle_sprite[1:0] = flame_middle_sprite_raw[7:6];
286         flame_h_sprite[1:0] = flame_h_sprite_raw[7:6];
287         flame_v_sprite[1:0] = flame_v_sprite_raw[7:6];
288         pickup_bi_sprite[1:0] = pickup_bi_sprite_raw[7:6];
289         pickup_fi_sprite[1:0] = pickup_fi_sprite_raw[7:6];
290         pickup_su_sprite[1:0] = pickup_su_sprite_raw[7:6];
291     end
292 endcase
293 map_read_addr = (vcount[9:4] * 40 + hcount[10:5]);
294 player_sprite = (player_sprite_raw >> (2 * player_offset_sr[3:2])) % 4;
295 player_color_addr = player_addr[10:6] * 4 + PLAYER_COLOR_OFFSET;
296 player_local_color_addr[7:0] = player_color_addr_sr[15:8] + player_sprite;
297
298 case (map_sig_select)
299     4'd0 : map_sprite = background_sprite;
300     4'd1 : map_sprite = stone_sprite;
301     4'd2 : map_sprite = box_sprite;
302     4'd3 : map_sprite = bomb_sprite;
303     4'd4 : map_sprite = flamed_box_sprite;
304     4'd5 : map_sprite = flame_middle_sprite;
305     4'd6 : map_sprite = flame_h_sprite;
306     4'd7 : map_sprite = flame_v_sprite;
307     4'd8 : map_sprite = pickup_bi_sprite;
308     4'd9 : map_sprite = pickup_fi_sprite;
309     4'd10 : map_sprite = pickup_su_sprite;
310     default : map_sprite = background_sprite;
311 endcase
312 map_local_color_addr = map_sig_select * 4 + map_sprite;
313 if (is_player_1) begin
314     player_addr_player = PLAYER_OFFSET;
315     facing = player_info_11[10:9];
316     moving = player_info_10[10];
317     player_addr_sprite = player_info_10 [12:11] * 64;
318     if (facing == 2'd1)
319         player_addr_local = (vcount - player_info_11[8:0] + 7) * 16 + 15 -
320             (hcount[10:1] - player_info_10[9:0] + 7);
321     else
322         player_addr_local = (vcount - player_info_11[8:0] + 7) * 16 +
323             (hcount[10:1] - player_info_10[9:0] + 7);
324 end else if (is_player_0) begin
325     player_addr_player = 11'd0;

```

```

324     facing = player_info_01[10:9];
325     moving = player_info_00[10];
326     player_addr_sprite = player_info_00 [12:11] * 64;
327     if (facing == 2'd1)
328         player_addr_local = (vcount - player_info_01[8:0] + 7) * 16 + 15 -
            (hcount[10:1] - player_info_00[9:0] + 7);
329     else
330         player_addr_local = (vcount - player_info_01[8:0] + 7) * 16 +
            (hcount[10:1] - player_info_00[9:0] + 7);
331 end else begin
332     player_addr_player = 11'd0;
333     facing = 2'd0;
334     moving = 1'd0;
335     player_addr_sprite = 11'd0;
336     player_addr_local = 11'd0;
337 end
338 if (moving) begin
339     player_addr_moving = MOVING_OFFSET;
340     case (facing)
341         2'd0 : player_addr_facing = 11'd0;
342         2'd1 : player_addr_facing = MOVING_SIDE_OFFSET;
343         2'd2 : player_addr_facing = MOVING_UP_OFFSET;
344         2'd3 : player_addr_facing = MOVING_SIDE_OFFSET;
345     endcase
346 end else begin
347     player_addr_moving = 11'd0;
348     case (facing)
349         2'd0 : player_addr_facing = 11'd0;
350         2'd1 : player_addr_facing = IDLE_SIDE_OFFSET;
351         2'd2 : player_addr_facing = IDLE_UP_OFFSET;
352         2'd3 : player_addr_facing = IDLE_SIDE_OFFSET;
353     endcase
354 end
355 if (is_player_sr[3]) begin
356     if (upper_layer_global_color_addr == 8'd0)
357         global_color_addr = lower_layer_global_color_addr;
358     else
359         global_color_addr = upper_layer_global_color_addr;
360 end else begin
361     global_color_addr = lower_layer_global_color_addr;
362 end
363 case(address)
364     5'd4 : begin
365         map_write_addr[10:0] = map_change_0[10:0];
366         map_write_data[3:0] = map_change_0[14:11];
367         map_write = map_change_0[15];
368     end
369     5'd5 : begin
370         map_write_addr[10:0] = map_change_1[10:0];
371         map_write_data[3:0] = map_change_1[14:11];
372         map_write = map_change_1[15];
373     end
374     5'd6 : begin
375         map_write_addr[10:0] = map_change_2[10:0];
376         map_write_data[3:0] = map_change_2[14:11];
377         map_write = map_change_2[15];

```

```

378     end
379     5'd7 : begin
380         map_write_addr[10:0] = map_change_3[10:0];
381         map_write_data[3:0] = map_change_3[14:11];
382         map_write = map_change_3[15];
383     end
384     5'd8 : begin
385         map_write_addr[10:0] = map_change_4[10:0];
386         map_write_data[3:0] = map_change_4[14:11];
387         map_write = map_change_4[15];
388     end
389     5'd9 : begin
390         map_write_addr[10:0] = map_change_5[10:0];
391         map_write_data[3:0] = map_change_5[14:11];
392         map_write = map_change_5[15];
393     end
394     5'd10 : begin
395         map_write_addr[10:0] = map_change_6[10:0];
396         map_write_data[3:0] = map_change_6[14:11];
397         map_write = map_change_6[15];
398     end
399     5'd11 : begin
400         map_write_addr[10:0] = map_change_7[10:0];
401         map_write_data[3:0] = map_change_7[14:11];
402         map_write = map_change_7[15];
403     end
404     5'd12 : begin
405         map_write_addr[10:0] = map_change_8[10:0];
406         map_write_data[3:0] = map_change_8[14:11];
407         map_write = map_change_8[15];
408     end
409     5'd13 : begin
410         map_write_addr[10:0] = map_change_9[10:0];
411         map_write_data[3:0] = map_change_9[14:11];
412         map_write = map_change_9[15];
413     end
414     5'd14 : begin
415         map_write_addr[10:0] = map_change_10[10:0];
416         map_write_data[3:0] = map_change_10[14:11];
417         map_write = map_change_10[15];
418     end
419     5'd15 : begin
420         map_write_addr[10:0] = map_change_11[10:0];
421         map_write_data[3:0] = map_change_11[14:11];
422         map_write = map_change_11[15];
423     end
424     default : begin
425         map_write_addr[10:0] = 11'd0;
426         map_write_data[3:0] = 4'd0;
427         map_write = 0;
428     end
429 endcase
430
431 end
432
433

```

```

434
435     rom_local_color local_colors_rom(.address_a(upper_layer_local_color_addr[6:0]),
436                                     .address_b(lower_layer_local_color_addr[6:0]),
437                                     .clock(clk),
438                                     .q_a(upper_layer_global_color_addr),
439                                     .q_b(lower_layer_global_color_addr));
440
441     rom_global_color_r global_color_r(.address(global_color_addr[5:0]),
442                                       .clock(clk),
443                                       .q(VGA_R));
444
445     rom_global_color_g global_color_g(.address(global_color_addr[5:0]),
446                                       .clock(clk),
447                                       .q(VGA_G));
448
449
450     rom_global_color_b global_color_b(.address(global_color_addr[5:0]),
451                                       .clock(clk),
452                                       .q(VGA_B));
453
454
455
456     always_ff @(posedge clk) begin
457         sprite_offset[1:0] <= sprite_offset_temp[1:0];
458         sprite_offset_temp[1:0] <= local_pixel[1:0];
459         player_offset_sr[1:0] <= player_addr_local[1:0];
460         player_offset_sr[3:2] <= player_offset_sr[1:0];
461         player_color_addr_sr[7:0] <= player_color_addr[7:0];
462         player_color_addr_sr[15:8] <= player_color_addr_sr[7:0];
463         is_player_sr[0] <= is_player;
464         is_player_sr[1] <= is_player_sr[0];
465         is_player_sr[2] <= is_player_sr[1];
466         is_player_sr[3] <= is_player_sr[2];
467         audio_valid_sr[1] <= audio_valid_sr[0];
468
469         if (chipselct && write) begin
470             case(address)
471                 5'd0 : player_info_00[15:0] <= writedata[15:0];
472                 5'd1 : player_info_01[15:0] <= writedata[15:0];
473                 5'd2 : player_info_10[15:0] <= writedata[15:0];
474                 5'd3 : player_info_11[15:0] <= writedata[15:0];
475                 5'd4 : map_change_0 [15:0] <= writedata[15:0];
476                 5'd5 : map_change_1 [15:0] <= writedata[15:0];
477                 5'd6 : map_change_2 [15:0] <= writedata[15:0];
478                 5'd7 : map_change_3 [15:0] <= writedata[15:0];
479                 5'd8 : map_change_4 [15:0] <= writedata[15:0];
480                 5'd9 : map_change_5 [15:0] <= writedata[15:0];
481                 5'd10 : map_change_6 [15:0] <= writedata[15:0];
482                 5'd11 : map_change_7 [15:0] <= writedata[15:0];
483                 5'd12 : map_change_8 [15:0] <= writedata[15:0];
484                 5'd13 : map_change_9 [15:0] <= writedata[15:0];
485                 5'd14 : map_change_10 [15:0] <= writedata[15:0];
486                 5'd15 : map_change_11 [15:0] <= writedata[15:0];
487                 5'd16 : audio_play_0 [15:0] <= writedata[15:0];
488             endcase
489         end

```

```

490
491     if (L_READY && R_READY) begin
492         if (audio_play_0 == 1'd1) begin
493             audio_start <= 1'd1;
494             audio_explosion_addr <= 14'd0;
495             audio_valid_sr[0] <= 1'd0;
496         end
497         else if (audio_divider == 14'b0 && audio_start == 1'd1) begin
498             if (audio_explosion_addr == 14'd12109) begin
499                 audio_explosion_addr <= 14'd0;
500                 audio_start <= 1'd0;
501             end
502             else begin
503                 audio_explosion_addr <= audio_explosion_addr + 14'd1;
504                 audio_start <= 1'd1;
505             end
506             audio_valid_sr[0] <= 1;
507         end
508         else
509             audio_valid_sr[0] <= 0;
510
511         if (audio_divider == 14'd3125)
512             audio_divider <= 14'd0;
513         else
514             audio_divider <= audio_divider + 14'd1;
515     end
516 end
517
518
519
520 endmodule
521
522 module vga_counters(
523     input logic      clk50, reset,
524     output logic [10:0] hcount, // hcount[10:1] is pixel column
525     output logic [9:0] vcount, // vcount[9:0] is pixel row
526     output logic     VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n, VGA_SYNC_n);
527
528 /*
529 * 640 X 480 VGA timing for a 50 MHz clock: one pixel every other cycle
530 *
531 * HCOUNT 1599 0          1279          1599 0
532 *
533 * -----| Video      |-----| Video
534 *
535 *
536 * |SYNC| BP |<-- HACTIVE -->|FP|SYNC| BP |<-- HACTIVE
537 *
538 * |____|      VGA_HS      |____|
539 */
540 // Parameters for hcount
541 parameter HACTIVE      = 11'd 1280,
542           HFRONT_PORCH = 11'd 32,
543           HSYNC        = 11'd 192,
544           HBACK_PORCH  = 11'd 96,
545           HTOTAL       = HACTIVE + HFRONT_PORCH + HSYNC +

```

```

546                                     HBACK_PORCH; // 1600
547
548 // Parameters for vcount
549 parameter VACTIVE    = 10'd 480,
550         VFRONT_PORCH = 10'd 10,
551         VSYNC        = 10'd 2,
552         VBACK_PORCH  = 10'd 33,
553         VTOTAL       = VACTIVE + VFRONT_PORCH + VSYNC +
554         VBACK_PORCH; // 525
555
556 logic endOfLine;
557 logic [10:0] effective_hcount;
558 logic [9:0]  effective_vcount;
559
560 count_sr sr_count_ram(.clock(clk50),
561         .shiftin({hcount[10:0], vcount[9:0]}),
562         .shiftout({effective_hcount[10:0], effective_vcount[9:0]}));
563
564 always_ff @(posedge clk50)
565     if (endOfLine)
566         hcount <= 0;
567     else
568         hcount <= hcount + 11'd 1;
569
570 assign endOfLine = hcount == HTOTAL - 1;
571
572 logic endOfField;
573
574 always_ff @(posedge clk50)
575     if (endOfLine)
576         if (endOfField)
577             vcount <= 0;
578         else
579             vcount <= vcount + 10'd 1;
580
581 assign endOfField = vcount == VTOTAL - 1;
582 assign VGA_CLK = hcount[0];
583 assign VGA_HS = !( (effective_hcount[10:8] == 3'b101) &
584         !(effective_hcount[7:5] == 3'b111));
585 assign VGA_VS = !( effective_vcount[9:1] == (VACTIVE + VFRONT_PORCH) / 2);
586
587 assign VGA_SYNC_n = 1'b0; // For putting sync on the green signal; unused
588
589 // Horizontal active: 0 to 1279 Vertical active: 0 to 479
590 // 101 0000 0000 1280      01 1110 0000 480
591 // 110 0011 1111 1599      10 0000 1100 524
592 assign VGA_BLANK_n = !( effective_hcount[10] & (effective_hcount[9] |
593         effective_hcount[8]) ) &
594         !( effective_vcount[9] | (effective_vcount[8:5] == 4'b1111) );
595
596 endmodule

```

## B vga.c



```

1      /* * Device driver for the VGA video generator
2      *
3      * A Platform device implemented using the misc subsystem
4      *
5      * Stephen A. Edwards
6      * Columbia University
7      *
8      * References:
9      * Linux source: Documentation/driver-model/platform.txt
10     *      drivers/misc/arm-charlcd.c
11     * http://www.linuxforu.com/tag/linux-device-drivers/
12     * http://free-electrons.com/docs/
13     *
14     * "make" to build
15     * insmod vga_ball.ko
16     *
17     * Check code style with
18     * checkpatch.pl --file --no-tree vga_ball.c
19     */
20
21 #include <linux/module.h>
22 #include <linux/init.h>
23 #include <linux/errno.h>
24 #include <linux/version.h>
25 #include <linux/kernel.h>
26 #include <linux/platform_device.h>
27 #include <linux/miscdevice.h>
28 #include <linux/slab.h>
29 #include <linux/io.h>
30 #include <linux/of.h>
31 #include <linux/of_address.h>
32 #include <linux/fs.h>
33 #include <linux/uaccess.h>
34 #include "hello.h"
35
36 #define DRIVER_NAME "vga_ball"
37
38 /* Device registers */
39 #define PLAYER1X(x) (x)
40 #define PLAYER1Y(x) ((x)+2)
41 #define TANK1_P(x) ((x)+4)
42 #define TANK1_D(x) ((x)+6)
43 #define ENEMY1_P(x) ((x)+8)
44 #define ENEMY1_D(x) ((x)+10)
45 #define ENEMY2_P(x) ((x)+12)
46 #define ENEMY2_D(x) ((x)+14)
47 #define ENEMY3_P(x) ((x)+16)
48 #define ENEMY3_D(x) ((x)+18)
49 #define ENEMY4_P(x) ((x)+20)
50 #define ENEMY4_D(x) ((x)+22)
51 #define BULLET1(x) ((x)+24)
52 #define BULLET2(x) ((x)+26)
53 #define BULLET3(x) ((x)+28)
54 #define BULLET4(x) ((x)+30)
55 #define BULLET5(x) ((x)+32)

```

```

56 #define RESERVE4(x) ((x)+34)
57 /*
58  * Information about our device
59  */
60 struct vga_ball_dev {
61     struct resource res; /* Resource: our registers */
62     void __iomem *virtbase; /* Where registers can be accessed in memory */
63     game_info_t background;
64 } dev;
65
66 /*
67  * Write segments of a single digit
68  * Assumes digit is in range and the device information has been set up
69  */
70 static void write_background(game_info_t *background)
71 {
72     iowrite16(background->playerinfo00, PLAYER1X(dev.virtbase) ); //display player
73     iowrite16(background->playerinfo01, PLAYER1Y(dev.virtbase) );
74     iowrite16(background->playerinfo10, TANK1_P(dev.virtbase) );
75     iowrite16(background->playerinfo11, TANK1_D(dev.virtbase) );
76     iowrite16(background->map_change_0, ENEMY1_P(dev.virtbase) );
77     iowrite16(background->map_change_1, ENEMY1_D(dev.virtbase) );
78     iowrite16(background->map_change_2, ENEMY2_P(dev.virtbase) );
79     iowrite16(background->map_change_3, ENEMY2_D(dev.virtbase) );
80     iowrite16(background->map_change_4, ENEMY3_P(dev.virtbase) );
81     iowrite16(background->map_change_5, ENEMY3_D(dev.virtbase) );
82     iowrite16(background->map_change_6, ENEMY4_P(dev.virtbase) );
83     iowrite16(background->map_change_7, ENEMY4_D(dev.virtbase) );
84     iowrite16(background->map_change_8, BULLET1(dev.virtbase) );
85     iowrite16(background->map_change_9, BULLET2(dev.virtbase) );
86     iowrite16(background->map_change_10, BULLET3(dev.virtbase) );
87     iowrite16(background->map_change_11, BULLET4(dev.virtbase) );
88     iowrite16(background->reserved_0, BULLET5(dev.virtbase) );
89     iowrite16(background->reserved_1, RESERVE4(dev.virtbase) );
90
91
92     //iowrite16(0x7320, TANK1_D(dev.virtbase) );
93     dev.background = *background;
94 }
95
96 /*
97  * Handle ioctl() calls from userspace:
98  * Read or write the segments on single digits.
99  * Note extensive error checking of arguments
100  */
101 static long vga_ball_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
102 {
103     vga_ball_arg_t vla;
104
105     switch (cmd) {
106     case VGA BALL WRITE BACKGROUND:
107         if (copy_from_user(&vla, (vga_ball_arg_t *) arg,
108             sizeof(vga_ball_arg_t)))
109             return -EACCES;
110         write_background(&vla.background);
111         break;

```

```

112
113     case VGA BALL_READ_BACKGROUND:
114         vla.background = dev.background;
115         if (copy_to_user((vga_ball_arg_t *) arg, &vla,
116             sizeof(vga_ball_arg_t)))
117             return -EACCES;
118         break;
119
120     default:
121         return -EINVAL;
122     }
123
124     return 0;
125 }
126
127 /* The operations our device knows how to do */
128 static const struct file_operations vga_ball_fops = {
129     .owner      = THIS_MODULE,
130     .unlocked_ioctl = vga_ball_ioctl,
131 };
132
133 /* Information about our device for the "misc" framework -- like a char dev */
134 static struct miscdevice vga_ball_misc_device = {
135     .minor      = MISC_DYNAMIC_MINOR,
136     .name       = DRIVER_NAME,
137     .fops       = &vga_ball_fops,
138 };
139
140 /*
141  * Initialization code: get resources (registers) and display
142  * a welcome message
143  */
144 static int __init vga_ball_probe(struct platform_device *pdev)
145 {
146     game_info_t beige = {0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
147         0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
148         0x0000,0x0000};
149     int ret;
150
151     /* Register ourselves as a misc device: creates /dev/vga_ball */
152     ret = misc_register(&vga_ball_misc_device);
153
154     /* Get the address of our registers from the device tree */
155     ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
156     if (ret) {
157         ret = -ENOENT;
158         goto out_deregister;
159     }
160
161     /* Make sure we can use these registers */
162     if (request_mem_region(dev.res.start, resource_size(&dev.res),
163         DRIVER_NAME) == NULL) {
164         ret = -EBUSY;
165         goto out_deregister;
166     }
167

```

```

168     /* Arrange access to our registers */
169     dev.virtbase = of_iomap(pdev->dev.of_node, 0);
170     if (dev.virtbase == NULL) {
171         ret = -ENOMEM;
172         goto out_release_mem_region;
173     }
174
175     /* Set an initial color */
176     write_background(&beige);
177
178
179     return 0;
180
181 out_release_mem_region:
182     release_mem_region(dev.res.start, resource_size(&dev.res));
183 out_deregister:
184     misc_deregister(&vga_ball_misc_device);
185     return ret;
186 }
187
188 /* Clean-up code: release resources */
189 static int vga_ball_remove(struct platform_device *pdev)
190 {
191     iounmap(dev.virtbase);
192     release_mem_region(dev.res.start, resource_size(&dev.res));
193     misc_deregister(&vga_ball_misc_device);
194     return 0;
195 }
196
197 /* Which "compatible" string(s) to search for in the Device Tree */
198 #ifdef CONFIG_OF
199 static const struct of_device_id vga_ball_of_match[] = {
200     { .compatible = "csee4840,vga_ball-1.0" },
201     {}},
202 };
203 MODULE_DEVICE_TABLE(of, vga_ball_of_match);
204 #endif
205
206 /* Information for registering ourselves as a "platform" driver */
207 static struct platform_driver vga_ball_driver = {
208     .driver = {
209         .name = DRIVER_NAME,
210         .owner = THIS_MODULE,
211         .of_match_table = of_match_ptr(vga_ball_of_match),
212     },
213     .remove = __exit_p(vga_ball_remove),
214 };
215
216 /* Called when the module is loaded: set things up */
217 static int __init vga_ball_init(void)
218 {
219     pr_info(DRIVER_NAME ": init\n");
220     return platform_driver_probe(&vga_ball_driver, vga_ball_probe);
221 }
222
223 /* Calball when the module is unloaded: release resources */

```

```

224 static void __exit vga_ball_exit(void)
225 {
226     platform_driver_unregister(&vga_ball_driver);
227     pr_info(DRIVER_NAME ": exit\n");
228 }
229
230 module_init(vga_ball_init);
231 module_exit(vga_ball_exit);
232
233 MODULE_LICENSE("GPL");
234 MODULE_AUTHOR("Stephen A. Edwards, Columbia University");
235 MODULE_DESCRIPTION("VGA ball driver");

```

## C hello.h

```

1     #ifndef _VGA BALL_H
2 #define _VGA BALL_H
3
4 #include <linux/ioctl.h>
5
6 typedef struct {
7     unsigned short playerinfo00, playerinfo01, playerinfo10, playerinfo11,
8     map_change_0, map_change_1, map_change_2, map_change_3, map_change_4,
9     map_change_5, map_change_6, map_change_7, map_change_8, map_change_9,
10    map_change_10, map_change_11, reserved_0, reserved_1;
11 } game_info_t;
12
13 typedef struct {
14     game_info_t background;
15 } vga_ball_arg_t;
16
17 #define VGA BALL_MAGIC 'q'
18
19 /* ioctls and their arguments */
20 #define VGA BALL_WRITE_BACKGROUND _IOW(VGA BALL_MAGIC, 1, vga_ball_arg_t *)
21 #define VGA BALL_READ_BACKGROUND _IOR(VGA BALL_MAGIC, 2, vga_ball_arg_t *)
22 #endif

```

## D controller.h

```

1     #ifndef _CONTROLLER_H
2 #define _CONTROLLER_H
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <libusb-1.0/libusb.h>
7 #include "bomberman.h"
8
9 struct controller_list {

```

```

10
11     struct libusb_device_handle *device1;
12     struct libusb_device_handle *device2;
13     uint8_t device1_addr;
14     uint8_t device2_addr;
15
16 };
17
18 struct controller_pkt {
19
20     uint8_t codes[7];
21
22 };
23
24 struct args_list {
25
26     struct controller_list devices;
27     char buttons[11];
28     int mode;
29     int print;
30     control_info_t *control_info;
31 };
32
33 struct controller_list open_controllers();
34 void *listen_controllers(void *arg);
35
36 #endif

```

## E bomberman.h

```

1     #ifndef _BOMBERMAN_H
2     #define _BOMBERMAN_H
3     #include <stdint.h>
4     #include "hello.h"
5     #define PLAYER_X_UPPER_LIM (uint16_t) 631
6     #define PLAYER_X_LOWER_LIM (uint16_t) 8
7     #define PLAYER_Y_UPPER_LIM (uint16_t) 471
8     #define PLAYER_Y_LOWER_LIM (uint16_t) 8
9     #define SET_BIT(x,y) x |= ((uint16_t) 1 << y)
10    #define SET_BITS(x,y,z) x |= ((uint16_t) z << y)
11    #define CLEAR_BIT(x,y) x &= ~(uint16_t) 1 << y)
12    #define MAP_SIZE 1200
13    #define PLAYERO_INIT_X (uint16_t) 168
14    #define PLAYERO_INIT_Y (uint16_t) 167
15    #define PLAYER1_INIT_X (uint16_t) 488
16    #define PLAYER1_INIT_Y (uint16_t) 327
17    #define abs_diff(x, y) ((x < y) ? (y - x) : (x - y))
18
19
20    enum FACING {DOWN, LEFT, UP, RIGHT};
21    enum STAT {STATIC, MOVING};
22    enum POSE {IDLE, SIDE0, SIDE1, SIDE2, DOWN0, DOWN1, UP0, UP1};
23    enum PLAYER {PLAYERO, PLAYER1};
24

```

```

25 typedef struct {
26     uint16_t vxpos;
27     uint16_t vypos;
28     enum FACING facing;
29     enum STAT status;
30     enum POSE pose;
31     uint16_t pos_tick;
32     uint16_t vspeed;
33     uint16_t bomb_range;
34     uint16_t max_bombs;
35     uint16_t bombs_left;
36     uint16_t bomb_colddown;
37     uint16_t dead;
38 } player_info_t;
39
40 typedef struct {
41     enum FACING direction0;
42     enum FACING direction1;
43     unsigned long long press_tick0;
44     unsigned long long press_tick1;
45     int attempt_place_bomb_0;
46     int attempt_place_bomb_1;
47     int idle0;
48     int idle1;
49 } control_info_t;
50
51 struct Bomb {
52     int time_left;
53     enum PLAYER player;
54     uint16_t pos;
55     struct Bomb *next;
56     uint16_t range;
57 };
58
59 struct Explosion {
60     int stage;
61     uint16_t range;
62     int ends[4]; /*Down, Left, Up, Right*/
63     int stage_time_left;
64     struct Explosion *next;
65 };
66 void init_explosion_sound(void);
67 void set_player_pos(uint16_t *pos, player_info_t *info);
68 void display_game_over(void);
69 void set_player_status (uint16_t vxpos, uint16_t vypos, enum FACING facing, enum
    STAT status, enum POSE pose, uint16_t pos_tick, uint16_t vspeed, uint16_t
    bomb_range, uint16_t max_bombs, uint16_t bombs_left, uint16_t bomb_colddown,
    uint16_t dead, enum PLAYER player);
70 void write_player_info(void);
71 void pass_game_info(void);
72 void generate_software_map(void);
73 void init_players(void);
74 int is_player_moving(player_info_t *info);
75 enum FACING get_player_facing(player_info_t *info);
76 void get_player_vpos(uint16_t *pos, player_info_t *info);
77 uint16_t get_player_vspeed(player_info_t *info);

```

```

78 void handle_player_movement(void);
79 void reset_map_change_list(void);
80 void sync_hw_map_change(void);
81 void handle_player_place_bomb(void);
82 void handle_bomb_explode(void);
83 int map_change_list_append_tile(uint16_t pos, uint16_t tile);
84 void handle_explosion(void);
85 void handle_player_prop_get(void);
86 void free_explosion(void);
87 void free_bombs(void);
88 #endif

```

## F controller.c

```

1   #include "controller.h"
2   #include <libusb-1.0/libusb.h>
3   #include <stdio.h>
4   #include <stdlib.h>
5   #include <string.h>
6   #include <unistd.h>
7   #include <sys/time.h>
8   #include "controller.h"
9
10  struct controller_list open_controllers() {
11
12      printf("Searching for USB connections...\n");
13
14      uint8_t endpoint_address = 0;
15      struct controller_list devices;
16      libusb_device **devs;
17      struct libusb_device_descriptor desc;
18      struct libusb_device_handle *controller = NULL;
19      ssize_t num_devs;
20
21
22
23      // Boot libusb library
24      if (libusb_init(NULL) != 0) {
25          printf("\nERROR: libusb failed to boot");
26          exit(1);
27      }
28
29      if ((num_devs = libusb_get_device_list(NULL, &devs)) < 0) {
30          printf("\nERROR: no controllers found");
31          exit(1);
32      }
33
34      //printf("Detected %d devices...\n", num_devs);
35      int connection_count = 0;
36      for (int i = 0; i < num_devs; i++) {
37
38          libusb_device *dev = devs[i];
39
40          if (libusb_get_device_descriptor(dev, &desc) < 0) {

```



```

41         printf("\nERROR: bad device descriptor.");
42         exit(1);
43     }
44
45     // Our controllers have idProduct of 17
46     if (desc.idProduct == 0xe401) {
47
48         //printf("FOUND: idProduct-%d ", desc.idProduct);
49         struct libusb_config_descriptor *config;
50         if ((libusb_get_config_descriptor(dev, 0, &config) < 0) {
51             printf("\nERROR: bad config descriptor.");
52             exit(1);
53         }
54         //printf("interfaces-%d\n", config->bNumInterfaces);
55
56         // Our controllers only have a single interface, no need for
           looping
57         // This interface also only has one .num_altsetting, no need
           for looping
58
59         int r;
60         const struct libusb_interface_descriptor *inter =
           config->interface[0].altsetting;
61         if ((r = libusb_open(dev, &controller)) != 0) {
62             printf("\nERROR: couldn't open controller");
63             exit(1);
64         }
65         if (libusb_kernel_driver_active(controller, 0)) {
66             libusb_detach_kernel_driver(controller, 0);
67         }
68         libusb_set_auto_detach_kernel_driver(controller, 0);
69         if ((r = libusb_claim_interface(controller, 0)) != 0) {
70             printf("\nERROR: couldn't claim controller.");
71             exit(1);
72         }
73
74         endpoint_address = inter->endpoint[0].bEndpointAddress;
75         connection_count++;
76
77         if (connection_count == 1) {
78             devices.device1 = controller;
79             devices.device1_addr = endpoint_address;
80         } else {
81             devices.device2 = controller;
82             devices.device2_addr = endpoint_address;
83             //printf("%d:%d,%d:%d\n",devices.device1,devices.device1_addr,devices.device2,d
           goto found;
84         }
85     }
86 }
87
88 if (connection_count < 2) {
89     printf("ERROR: couldn't find 2 controllers.");
90     exit(1);
91 }
92

```

```

93     found:
94         printf("Connected %d controllers!\n", connection_count);
95         libusb_free_device_list(devs, 1);
96
97         return devices;
98     }
99
100 void detect_presses(struct controller_pkt pkt1, struct controller_pkt pkt2,
101                    control_info_t *control_info, int print) {
102
103     // Choose whether you want human-readable or binary output
104     if (pkt1.codes[0] == 0x7f && pkt1.codes[1] == 0xff && pkt1.codes[2] == 0x00)
105     { //Player0 press Down direction
106         control_info -> idle0 = 0;
107         if (control_info -> direction0 == DOWN)
108             control_info -> press_tick0 ++;
109
110         else {
111             control_info -> direction0 = DOWN;
112             control_info -> press_tick0 = 0;
113         }
114         if (print)
115             printf("Player 0: Down\n");
116     } else if (pkt1.codes[0] == 0x00 && pkt1.codes[1] == 0x7f && pkt1.codes[2] ==
117     0x00) {
118         control_info -> idle0 = 0;
119         if (control_info -> direction0 == LEFT)
120             control_info -> press_tick0 ++;
121         else {
122             control_info -> direction0 = LEFT;
123             control_info -> press_tick0 = 0;
124         }
125         if (print)
126             printf("Player 0: Left\n");
127     } else if (pkt1.codes[0] == 0x7f && pkt1.codes[1] == 0x00 && pkt1.codes[2] ==
128     0x00) {
129         control_info -> idle0 = 0;
130         if (control_info -> direction0 == UP)
131             control_info -> press_tick0 ++;
132         else {
133             control_info -> direction0 = UP;
134             control_info -> press_tick0 = 0;
135         }
136         if (print)
137             printf("Player 0: Up\n");
138     } else if (pkt1.codes[0] == 0xff && pkt1.codes[1] == 0x7f && pkt1.codes[2] ==
139     0x00) {
140         control_info -> idle0 = 0;
141         if (control_info -> direction0 == RIGHT)
142             control_info -> press_tick0 ++;
143         else {
144             control_info -> direction0 = RIGHT;
145             control_info -> press_tick0 = 0;
146         }
147         if (print)
148             printf("Player 0: Right\n");

```

```

144 } else {
145     control_info -> press_tick0 = 0;
146     control_info -> idle0 = 1;
147 }
148 if (pkt2.codes[0] == 0x7f && pkt2.codes[1] == 0xff && pkt2.codes[2] == 0x00)
149     { //Player1 press Down direction
150     control_info -> idle1 = 0;
151     if (control_info -> direction1 == DOWN)
152         control_info -> press_tick1 ++;
153
154     else {
155         control_info -> direction1 = DOWN;
156         control_info -> press_tick1 = 0;
157     }
158     if (print)
159         printf("Player 1: Down\n");
160 } else if (pkt2.codes[0] == 0x00 && pkt2.codes[1] == 0x7f && pkt2.codes[2] ==
161 0x00) {
162     control_info -> idle1 = 0;
163     if (control_info -> direction1 == LEFT)
164         control_info -> press_tick1 ++;
165     else {
166         control_info -> direction1 = LEFT;
167         control_info -> press_tick1 = 0;
168     }
169     if (print)
170         printf("Player 1: Left\n");
171 } else if (pkt2.codes[0] == 0x7f && pkt2.codes[1] == 0x00 && pkt2.codes[2] ==
172 0x00) {
173     control_info -> idle1 = 0;
174     if (control_info -> direction1 == UP)
175         control_info -> press_tick1 ++;
176     else {
177         control_info -> direction1 = UP;
178         control_info -> press_tick1 = 0;
179     }
180     if (print)
181         printf("Player 1: Up\n");
182 } else if (pkt2.codes[0] == 0xff && pkt2.codes[1] == 0x7f && pkt2.codes[2] ==
183 0x00) {
184     control_info -> idle1 = 0;
185     if (control_info -> direction1 == RIGHT)
186         control_info -> press_tick1 ++;
187     else {
188         control_info -> direction1 = RIGHT;
189         control_info -> press_tick1 = 0;
190     }
191     if (print)
192         printf("Player 1: Right\n");
193 } else {
194     control_info -> press_tick1 = 0;
195     control_info -> idle1 = 1;
196 }
197 if (pkt1.codes[5] == 0x2f) {
198     control_info -> attempt_place_bomb_0 = 1;
199     if (print)

```

```

196         printf("Player 0: Place bomb!\n");
197     } else
198         control_info -> attempt_place_bomb_0 = 0;
199     if (pkt2.codes[5] == 0x2f) {
200         control_info -> attempt_place_bomb_1 = 1;
201         if (print)
202             printf("Player 1: Place bomb!\n");
203     } else
204         control_info -> attempt_place_bomb_1 = 0;
205
206
207 }
208
209 void *listen_controllers(void *arg) {
210
211     struct args_list *args_p = arg;
212     struct args_list args = *args_p;
213     struct controller_list devices = args.devices;
214
215     struct controller_pkt pkt1, pkt2;
216     int fields1, fields2;
217     int size1 = sizeof(pkt1);
218     int size2 = sizeof(pkt2);
219     char buttons1[] = "-----";
220     char buttons2[] = "-----";
221     struct timeval stop, start;
222     unsigned long interval[10];
223     int count = 0;
224     gettimeofday(&start, NULL);
225     for (;;) {
226
227         libusb_interrupt_transfer(devices.device1, devices.device1_addr,
228             (unsigned char *) &pkt1, size1, &fields1, 0);
229         libusb_interrupt_transfer(devices.device2, devices.device2_addr,
230             (unsigned char *) &pkt2, size2, &fields2, 0);
231
232         // 7 fields should be transferred for each packet
233         if (fields1 == 7 && fields2 == 7) {
234             detect_presses(pkt1, pkt2, args.control_info, args.print);
235         }
236     }
237 }

```

## G bomberman.c

```

1     #include <unistd.h>
2     #include <stdio.h>
3     #include <sys/ioctl.h>
4     #include <sys/types.h>
5     #include <sys/stat.h>
6     #include <fcntl.h>
7     #include <string.h>

```







```

44
45 int start_explosion(struct Bomb *bomb)
46 {
47     if (map_change_list_next >= 12)
48         return -1;
49     start_explosion_sound();
50     printf("Start explosion at %d\n", bomb->pos);
51     map_change_list_append_tile(bomb->pos, 5);
52     map[(bomb->pos)] = 5;
53     struct Explosion *new = (struct Explosion *) malloc(sizeof(struct Explosion));
54     new->stage = 0;
55     new->range = bomb->range;
56     int pos = bomb -> pos;
57     printf("map[%d]=%d\n", pos, map[pos]);
58     new->stage_time_left = 50 / ((bomb->range) + 2);
59     for (int i = 0; i < 4; i++)
60         new->ends[i] = pos;
61     if (explosion_head == NULL){
62         new->next = NULL;
63         explosion_head = new;
64     }
65     else{
66         new->next = explosion_head;
67         explosion_head = new;
68     }
69     return 0;
70 }
71
72 void free_explosion(void)
73 {
74     struct Explosion *curr;
75     while (curr) {
76         struct Explosion *temp = curr->next;
77         free(temp);
78         curr = temp;
79     }
80 }
81
82 int is_flame(uint16_t tile) {
83     return ((tile >= 5) && (tile <= 7));
84 }
85
86 int is_flame_obstacle(uint16_t tile) {
87     return ((tile > 0) && (tile < 4) && (tile != 2));
88 }
89
90
91 int can_expand_flame(int from, int to) {
92     uint16_t from_tile = map[from];
93     if (is_flame(from_tile))
94         if (!is_flame_obstacle(map[to]))
95             return 1;
96     return 0;
97 }
98 /*Modify ends, hw, map*/
99 int expand_flame(int ends[])

```



```

100 {
101     /*Get number of registers needed for the map change*/
102     int needs_modify[4]; /*A structure to denote which end needs to be expanded*/
103     printf("Start expanding flame, ends = [%d %d %d %d]\n", ends[0], ends[1],
104           ends[2], ends[3]);
105     for (int i = 0; i < 4; i++) {
106         int pos = ends[i];
107         printf("map[%d] = %d\n", pos, map[pos]);
108         int pos_r = pos / 40;
109         int pos_c = pos % 40;
110         int to_r;
111         int to_c;
112         switch (i) {
113             case 0:
114                 to_r = pos_r + 1;
115                 to_c = pos_c;
116                 break;
117             case 1:
118                 to_r = pos_r;
119                 to_c = pos_c - 1;
120                 break;
121             case 2:
122                 to_r = pos_r - 1;
123                 to_c = pos_c;
124                 break;
125             case 3:
126                 to_r = pos_r;
127                 to_c = pos_c + 1;
128                 break;
129         }
130         if ((to_r >= 0 && to_r < 30) && (to_c >= 0 && to_c < 40)) {
131             int to_pos = to_r * 40 + to_c;
132             if (can_expand_flame(pos, to_pos))
133                 needs_modify[i] = 1;
134             else
135                 needs_modify[i] = 0;
136         } else {
137             needs_modify[i] = 0;
138         }
139     }
140     int num_reg_needed = 0;
141     for (int i = 0; i < 4; i++)
142         num_reg_needed += needs_modify[i];
143     if (map_change_list_next + num_reg_needed > 12)
144         return -1;
145     printf("Needs modify [%d, %d, %d, %d]\n", needs_modify[0], needs_modify[1],
146           needs_modify[2], needs_modify[3]);
147     for (int i = 0; i < 4; i++) {
148         if(needs_modify[i]) {
149             int pos = ends[i];
150             int pos_r = pos / 40;
151             int pos_c = pos % 40;
152             int to_r;
153             int to_c;
154             switch (i) {
155                 case 0:

```

```

154         to_r = pos_r + 1;
155         to_c = pos_c;
156         break;
157     case 1:
158         to_r = pos_r;
159         to_c = pos_c - 1;
160         break;
161     case 2:
162         to_r = pos_r - 1;
163         to_c = pos_c;
164         break;
165     case 3:
166         to_r = pos_r;
167         to_c = pos_c + 1;
168         break;
169     }
170     uint16_t target_flame;
171     if (i % 2 == 0)
172         target_flame = 6;
173     else
174         target_flame = 7;
175     int to_pos = to_r * 40 + to_c;
176     uint16_t original_tile = map[to_pos];
177     uint16_t target_tile;
178     if (is_flame(original_tile)) {
179         if (target_flame != original_tile)
180             target_tile = 5;
181         else {
182             target_tile = target_flame;
183         }
184     }
185     }
186     else if (original_tile == 2)
187         target_tile = 4;
188     else {
189         target_tile = target_flame;
190     }
191     /*Change the hw*/
192     map_change_list_append_tile(to_pos, target_tile);
193     map[to_pos] = target_tile;
194     ends[i] = to_pos;
195     printf("Write flame to row %d col %d\n", to_r, to_c);
196 }
197 }
198
199 return 0;
200 }
201
202 int shrink_flame(int ends[], int center) {
203     /*get number of registers to change*/
204     printf("Start shrinking flame, ends = [%d, %d, %d, %d]\n", ends[0], ends[1],
205         ends[2], ends[3]);
206     int needs_modify[4];
207     for (int i = 0; i < 4; i++) {
208         if ((is_flame(map[ends[i]]) || map[ends[i]] == 4) && ends[i] != center)
209             needs_modify[i] = 1;

```

```

209     else if (ends[i] != center) {
210         int pos = ends[i];
211         int pos_r = pos / 40;
212         int pos_c = pos % 40;
213         int to_r;
214         int to_c;
215         switch (i) {
216             case 0:
217                 to_r = pos_r - 1;
218                 to_c = pos_c;
219                 break;
220             case 1:
221                 to_r = pos_r;
222                 to_c = pos_c + 1;
223                 break;
224             case 2:
225                 to_r = pos_r + 1;
226                 to_c = pos_c;
227                 break;
228             case 3:
229                 to_r = pos_r;
230                 to_c = pos_c - 1;
231                 break;
232         }
233         int to_pos = to_r * 40 + to_c;
234         ends[i] = to_pos;
235         needs_modify[i] = 0;
236     }
237     else {
238         needs_modify[i] = 0;
239     }
240 }
241 int num_reg_needed = 0;
242 for (int i = 0; i < 4; i++)
243     num_reg_needed += needs_modify[i];
244 if (map_change_list_next + num_reg_needed > 12)
245     return -1;
246 for (int i = 0; i < 4; i++) {
247     if (needs_modify[i]) {
248         int pos = ends[i];
249         int pos_r = pos / 40;
250         int pos_c = pos % 40;
251         int to_r;
252         int to_c;
253         switch (i) {
254             case 0:
255                 to_r = pos_r - 1;
256                 to_c = pos_c;
257                 break;
258             case 1:
259                 to_r = pos_r;
260                 to_c = pos_c + 1;
261                 break;
262             case 2:
263                 to_r = pos_r + 1;
264                 to_c = pos_c;

```

```

265         break;
266     case 3:
267         to_r = pos_r;
268         to_c = pos_c - 1;
269         break;
270     }
271     if (map[ends[i]] != 5) {
272         if (map[ends[i]] != 4) {
273             map_change_list_append_tile(ends[i], (uint16_t) 0);
274             map[ends[i]] = 0;
275             printf("delete flame at %d\n", ends[i]);
276         }
277         else {
278             uint16_t prop_id = rand() % 10;
279             if (prop_id < 3) {
280                 map_change_list_append_tile(ends[i], (uint16_t) prop_id + 8);
281                 map[ends[i]] = prop_id + 8;
282                 printf("place prop %d at %d\n", prop_id, ends[i]);
283             }
284             else {
285                 map_change_list_append_tile(ends[i], (uint16_t) 0);
286                 map[ends[i]] = 0;
287                 printf("delete flame at %d\n", ends[i]);
288             }
289         }
290     }
291     else {
292         if((i % 2) == 0) {
293             map_change_list_append_tile(ends[i], (uint16_t) 7);
294             map[ends[i]] = 7;
295         }
296         else {
297             map_change_list_append_tile(ends[i], (uint16_t) 6);
298             map[ends[i]] = 6;
299         }
300     }
301     int to_pos = to_r * 40 + to_c;
302     ends[i] = to_pos;
303 }
304 }
305 return 0;
306 }
307
308 void handle_explosion(void)
309 {
310     struct Explosion *curr = explosion_head;
311     struct Explosion *prev = explosion_head;
312     while(curr){
313         if(curr->stage_time_left <= 0){
314             int stage = curr->stage;
315             int range = (int) curr->range;
316             printf("stage = %d, range = %d\n", stage, range);
317             if (stage < range) {
318                 if (expand_flame(curr->ends)) {
319                     printf("Warning: Need enough registers to do the flame
320                             expansion!\n");

```

```

320         continue;
321     }
322 }
323 else if ((stage > range) && (stage <= range * 2)) {
324     int center_c = curr->ends[0] % 40;
325     int center_r = curr->ends[1] / 40;
326     int center = center_r * 40 + center_c;
327     if (shrink_flame(curr->ends, center)) {
328         printf("Warning: Need enough registers to do the flame shrink!\n");
329         continue;
330     }
331 }
332 else if (stage > range * 2){
333     int center_c = curr->ends[0] % 40;
334     int center_r = curr->ends[1] / 40;
335     int center = center_r * 40 + center_c;
336     for (int i = 0; i < 4; i++) {
337         if (curr->ends[i] != center)
338             printf("Warning: explosion has not been shrunk to 1\n");
339     }
340     if (map_change_list_append_tile(center, (uint16_t) 0)) {
341         printf("Warning: Need enough registers to do the flame
342             deletion!\n");
343         continue;
344     }
345     map[center] = 0;
346     if (curr == explosion_head) {
347         explosion_head = curr->next;
348         free(curr);
349         curr = explosion_head;
350         continue;
351     }
352     else {
353         prev->next = curr->next;
354         free(curr);
355         curr = prev->next;
356         continue;
357     }
358 }
359 curr->stage ++;
360 curr->stage_time_left = 50 / (curr->range + 2);
361 prev = curr;
362 curr = curr->next;
363 }
364 else {
365     curr->stage_time_left --;
366     prev = curr;
367     curr = curr->next;
368 }
369 }
370 }
371
372 void insert_bomb(enum PLAYER player, uint16_t pos) {
373     struct Bomb *new = (struct Bomb *) malloc(sizeof(struct Bomb));
374     new->time_left = 300;

```

```

375     new->player = player;
376     new->pos = pos;
377     switch (player) {
378         case PLAYER0:
379             new->range = player0_info.bomb_range;
380             break;
381         case PLAYER1:
382             new->range = player1_info.bomb_range;
383             break;
384     }
385     if (bombs_head == NULL) {
386         bombs_head = new;
387         bombs_head->next = NULL;
388     }
389     else {
390         new->next = bombs_head;
391         bombs_head = new;
392     }
393 }
394
395 void player_get_prop(uint16_t pos, player_info_t *info) {
396     if (map[pos] == 8) {
397         if (!map_change_list_append_tile(pos, (uint16_t) 0)) {
398             map[pos] = 0;
399             if (info->max_bombs < 6) {
400                 info->max_bombs ++;
401                 info->bombs_left ++;
402                 info->bomb_colddown = 50;
403             }
404         }
405     }
406     else if (map[pos] == 9) {
407         if (!map_change_list_append_tile(pos, (uint16_t) 0)) {
408             map[pos] = 0;
409             if (info->bomb_range < 5) {
410                 info->bomb_range ++;
411             }
412         }
413     }
414     else if (map[pos] == 10) {
415         if (!map_change_list_append_tile(pos, (uint16_t) 0)) {
416             map[pos] = 0;
417             if (info->vspeed < 4) {
418                 info->vspeed ++;
419             }
420         }
421     }
422 }
423 void handle_player_prop_get()
424 {
425     uint16_t vpos[2];
426     get_player_vpos(vpos, &player0_info);
427     uint16_t pos[2];
428     pos[0] = vpos[0] / 3;
429     pos[1] = vpos[1] / 3;
430     int xpos = (int) pos[0];

```

```

431     int ypos = (int) pos[1];
432     int blocktlx = (xpos - 6) / 16;
433     int blocktly = (ypos - 6) / 16;
434     player_get_prop(blocktly * 40 + blocktlx, &player0_info);
435     int blocktrx = (xpos + 7) / 16;
436     int blocktry = (ypos - 6) / 16;
437     player_get_prop(blocktry * 40 + blocktrx, &player0_info);
438     int blockblx = (xpos - 6) / 16;
439     int blockbly = (ypos + 7) / 16;
440     player_get_prop(blockbly * 40 + blockblx, &player0_info);
441     int blockbrx = (xpos + 7) / 16;
442     int blockbry = (ypos + 7) / 16;
443     player_get_prop(blockbry * 40 + blockbrx, &player0_info);
444     get_player_vpos(vpos, &player1_info);
445     pos[0] = vpos[0] / 3;
446     pos[1] = vpos[1] / 3;
447     xpos = (int) pos[0];
448     ypos = (int) pos[1];
449     blocktlx = (xpos - 6) / 16;
450     blocktly = (ypos - 6) / 16;
451     player_get_prop(blocktly * 40 + blocktlx, &player1_info);
452     blocktrx = (xpos + 7) / 16;
453     blocktry = (ypos - 6) / 16;
454     player_get_prop(blocktry * 40 + blocktrx, &player1_info);
455     blockblx = (xpos - 6) / 16;
456     blockbly = (ypos + 7) / 16;
457     player_get_prop(blockbly * 40 + blockblx, &player1_info);
458     blockbrx = (xpos + 7) / 16;
459     blockbry = (ypos + 7) / 16;
460     player_get_prop(blockbry * 40 + blockbrx, &player1_info);
461 }
462 void handle_bomb_explode(void) {
463     struct Bomb *curr = bombs_head;
464     struct Bomb *prev = bombs_head;
465     while(curr) {
466         curr->time_left--;
467         if (curr->time_left <= 0) {
468             if (curr == bombs_head) {
469                 if (start_explosion(curr)) {
470                     prev = curr;
471                     curr = curr->next;
472                     continue;
473                 }
474                 bombs_head = curr->next;
475                 switch(curr->player) {
476                     case PLAYER0:
477                         player0_info.bombs_left++;
478                         break;
479                     case PLAYER1:
480                         player1_info.bombs_left++;
481                         break;
482                 }
483                 struct Bomb* temp = curr->next;
484                 free(curr);
485                 curr = temp;
486             }

```

```

487     else {
488         if (start_explosion(curr)) {
489             prev = curr;
490             curr = curr->next;
491             continue;
492         }
493         prev->next = curr->next;
494         switch(curr->player) {
495             case PLAYER0:
496                 player0_info.bombs_left ++;
497                 break;
498             case PLAYER1:
499                 player1_info.bombs_left ++;
500                 break;
501         }
502         struct Bomb *temp = curr->next;
503         free(curr);
504         curr = temp;
505     }
506 }
507 else {
508     prev = curr;
509     curr = curr->next;
510 }
511 }
512 }
513 }
514
515 void free_bombs(void) {
516     struct Bomb* curr = bombs_head;
517     while(curr) {
518         struct Bomb *temp = curr->next;
519         free(curr);
520         curr = temp;
521     }
522 }
523
524 void reset_map_change_list(void) {
525     memset((void *) map_change_list, 0, 12 * sizeof(short));
526     map_change_list_next = 0;
527 }
528
529 void sync_hw_map_change() {
530     memcpy(&global_info.map_change_0, &map_change_list[0], sizeof(short));
531     memcpy(&global_info.map_change_1, &map_change_list[1], sizeof(short));
532     memcpy(&global_info.map_change_2, &map_change_list[2], sizeof(short));
533     memcpy(&global_info.map_change_3, &map_change_list[3], sizeof(short));
534     memcpy(&global_info.map_change_4, &map_change_list[4], sizeof(short));
535     memcpy(&global_info.map_change_5, &map_change_list[5], sizeof(short));
536     memcpy(&global_info.map_change_6, &map_change_list[6], sizeof(short));
537     memcpy(&global_info.map_change_7, &map_change_list[7], sizeof(short));
538     memcpy(&global_info.map_change_8, &map_change_list[8], sizeof(short));
539     memcpy(&global_info.map_change_9, &map_change_list[9], sizeof(short));
540     memcpy(&global_info.map_change_10, &map_change_list[10], sizeof(short));
541     memcpy(&global_info.map_change_11, &map_change_list[11], sizeof(short));
542 }

```



```

543 void display_game_over()
544 {
545     if (player0_info.dead && !player1_info.dead) {
546         uint16_t pos[2] = {21, 21};
547         uint16_t pos2[2] = {21, 42};
548         set_player_pos(pos, &player0_info);
549         set_player_pos(pos2, &player1_info);
550         for(int i = 0; i < 100; i++) {
551             write_tile(i * 12, p2_win_arr[i * 12], &global_info.map_change_0);
552             write_tile(i * 12 + 1, p2_win_arr[i * 12 + 1], &global_info.map_change_1);
553             write_tile(i * 12 + 2, p2_win_arr[i * 12 + 2], &global_info.map_change_2);
554             write_tile(i * 12 + 3, p2_win_arr[i * 12 + 3], &global_info.map_change_3);
555             write_tile(i * 12 + 4, p2_win_arr[i * 12 + 4], &global_info.map_change_4);
556             write_tile(i * 12 + 5, p2_win_arr[i * 12 + 5], &global_info.map_change_5);
557             write_tile(i * 12 + 6, p2_win_arr[i * 12 + 6], &global_info.map_change_6);
558             write_tile(i * 12 + 7, p2_win_arr[i * 12 + 7], &global_info.map_change_7);
559             write_tile(i * 12 + 8, p2_win_arr[i * 12 + 8], &global_info.map_change_8);
560             write_tile(i * 12 + 9, p2_win_arr[i * 12 + 9], &global_info.map_change_9);
561             write_tile(i * 12 + 10, p2_win_arr[i * 12 + 10],
562                 &global_info.map_change_10);
563             write_tile(i * 12 + 11, p2_win_arr[i * 12 + 11],
564                 &global_info.map_change_11);
565             write_player_info();
566             pass_game_info();
567             usleep(2000);
568         }
569         printf("player1 wins!\n");
570     }
571     else if (player1_info.dead && !player0_info.dead){
572         uint16_t pos[2] = {21, 21};
573         uint16_t pos2[2] = {21, 42};
574         set_player_pos(pos, &player0_info);
575         set_player_pos(pos2, &player1_info);
576         for(int i = 0; i < 100; i++) {
577             write_tile(i * 12, p1_win_arr[i * 12], &global_info.map_change_0);
578             write_tile(i * 12 + 1, p1_win_arr[i * 12 + 1], &global_info.map_change_1);
579             write_tile(i * 12 + 2, p1_win_arr[i * 12 + 2], &global_info.map_change_2);
580             write_tile(i * 12 + 3, p1_win_arr[i * 12 + 3], &global_info.map_change_3);
581             write_tile(i * 12 + 4, p1_win_arr[i * 12 + 4], &global_info.map_change_4);
582             write_tile(i * 12 + 5, p1_win_arr[i * 12 + 5], &global_info.map_change_5);
583             write_tile(i * 12 + 6, p1_win_arr[i * 12 + 6], &global_info.map_change_6);
584             write_tile(i * 12 + 7, p1_win_arr[i * 12 + 7], &global_info.map_change_7);
585             write_tile(i * 12 + 8, p1_win_arr[i * 12 + 8], &global_info.map_change_8);
586             write_tile(i * 12 + 9, p1_win_arr[i * 12 + 9], &global_info.map_change_9);
587             write_tile(i * 12 + 10, p1_win_arr[i * 12 + 10],
588                 &global_info.map_change_10);
589             write_tile(i * 12 + 11, p1_win_arr[i * 12 + 11],
590                 &global_info.map_change_11);
591             write_player_info();
592             pass_game_info();
593             usleep(2000);
594         }
595         printf("player0 wins!\n");
596     }
597     else {
598         uint16_t pos[2] = {21, 21};

```

```

595     uint16_t pos2[2] = {21, 42};
596     set_player_pos(pos, &player0_info);
597     set_player_pos(pos2, &player1_info);
598     for(int i = 0; i < 100; i++) {
599         write_tile(i * 12, tie_arr[i * 12], &global_info.map_change_0);
600         write_tile(i * 12 + 1, tie_arr[i * 12 + 1], &global_info.map_change_1);
601         write_tile(i * 12 + 2, tie_arr[i * 12 + 2], &global_info.map_change_2);
602         write_tile(i * 12 + 3, tie_arr[i * 12 + 3], &global_info.map_change_3);
603         write_tile(i * 12 + 4, tie_arr[i * 12 + 4], &global_info.map_change_4);
604         write_tile(i * 12 + 5, tie_arr[i * 12 + 5], &global_info.map_change_5);
605         write_tile(i * 12 + 6, tie_arr[i * 12 + 6], &global_info.map_change_6);
606         write_tile(i * 12 + 7, tie_arr[i * 12 + 7], &global_info.map_change_7);
607         write_tile(i * 12 + 8, tie_arr[i * 12 + 8], &global_info.map_change_8);
608         write_tile(i * 12 + 9, tie_arr[i * 12 + 9], &global_info.map_change_9);
609         write_tile(i * 12 + 10, tie_arr[i * 12 + 10], &global_info.map_change_10);
610         write_tile(i * 12 + 11, tie_arr[i * 12 + 11], &global_info.map_change_11);
611         write_player_info();
612         pass_game_info();
613         usleep(2000);
614     }
615     printf("It's a tie!\n");
616 }
617 }
618 void set_player_status (uint16_t vxpos, uint16_t vypos, enum FACING facing, enum
STAT status, enum POSE pose, uint16_t pos_tick, uint16_t vspeed, uint16_t
bomb_range, uint16_t max_bombs, uint16_t bombs_left, uint16_t bomb_colddown,
uint16_t dead, enum PLAYER player)
619 {
620     player_info_t *target_info;
621     if (player == PLAYER0)
622         target_info = &player0_info;
623     else
624         target_info = &player1_info;
625     target_info->vxpos = vxpos;
626     target_info->vypos = vypos;
627     target_info->facing = facing;
628     target_info->status = status;
629     target_info->pose = pose;
630     target_info->pos_tick = pos_tick;
631     target_info->vspeed = vspeed;
632     target_info->bomb_range = bomb_range;
633     target_info->max_bombs = max_bombs;
634     target_info->bombs_left = bombs_left;
635     target_info->bomb_colddown = bomb_colddown;
636     target_info->dead = dead;
637 }
638
639 void init_players()
640 {
641     set_player_status (PLAYER0_INIT_X * 3, PLAYER0_INIT_Y * 3, DOWN, STATIC, IDLE,
0, 1, 1, 1, 1, 0, 0, PLAYER0);
642     set_player_status (PLAYER1_INIT_X * 3, PLAYER1_INIT_Y * 3, DOWN, STATIC, IDLE,
0, 1, 1, 1, 1, 0, 0, PLAYER1);
643 }
644
645

```

```

646 void write_player_info()
647 {
648     uint16_t pos = 0;
649     global_info.playerinfo00 = player0_info.vxpos / 3;
650     if (player0_info.status == MOVING)
651         SET_BIT(global_info.playerinfo00, 10);
652     if (player0_info.pose == SIDE1 || player0_info.pose == DOWN1 ||
        player0_info.pose == UP1)
653         pos = 1;
654     else if (player0_info.pose == SIDE2)
655         pos = 2;
656     SET_BITS(global_info.playerinfo00, 11, pos);
657     global_info.playerinfo01 = player0_info.vypos / 3;
658     SET_BITS(global_info.playerinfo01, 9, player0_info.facing);
659
660
661     global_info.playerinfo10 = player1_info.vxpos / 3;
662     if (player1_info.status == MOVING)
663         SET_BIT(global_info.playerinfo10, 10);
664     pos = 0;
665     if (player1_info.pose == SIDE1 || player1_info.pose == DOWN1 ||
        player1_info.pose == UP1)
666         pos = 1;
667     else if (player1_info.pose == SIDE2)
668         pos = 2;
669     SET_BITS(global_info.playerinfo10, 11, pos);
670     global_info.playerinfo11 = player1_info.vypos / 3;
671     SET_BITS(global_info.playerinfo11, 9, player1_info.facing);
672 }
673
674 void write_tile(uint16_t tile_pos, uint16_t tile_type, unsigned short *pos)
675 {
676     *pos = tile_pos;
677     SET_BITS(*pos, 11, tile_type);
678     SET_BIT(*pos, 15);
679 }
680
681 void pass_game_info()
682 {
683     vga_ball_arg_t vla;
684     vla.background = global_info;
685     if (ioctl(vga_fd, VGA BALL_WRITE_BACKGROUND, &vla)) {
686         perror("ioctl(VGA BALL_SET_BACKGROUND) failed");
687         return;
688     }
689 }
690
691 uint16_t manhattan_distance(uint16_t x_0, uint16_t y_0, uint16_t x_1, uint16_t y_1)
692 {
693     uint16_t x_diff;
694     if (x_0 < x_1)
695         x_diff = x_1 - x_0;
696     else
697         x_diff = x_0 - x_1;
698     uint16_t y_diff;
699     if (y_0 < y_1)

```

```

700     y_diff = y_1 - y_0;
701     else
702         y_diff = y_0 - y_1;
703     return x_diff + y_diff;
704
705 }
706
707 uint16_t is_obstacle(uint16_t tile)
708 {
709     return ((tile > 0) && (tile < 8));
710 }
711
712 void generate_software_map()
713 {
714     time_t t;
715     srand((unsigned) time(&t));
716     write_player_info();
717     pass_game_info();
718     map = (uint16_t *) malloc (MAP_SIZE * sizeof(uint16_t));
719     uint16_t row_p0 = 10;
720     uint16_t col_p0 = 10;
721     uint16_t row_p1 = 20;
722     uint16_t col_p1 = 30;
723     for (uint16_t i = 0; i < MAP_SIZE; i ++) {
724         uint16_t row = i / 40;
725         uint16_t col = i % 40;
726         if (row % 2 == 1 && col % 2 == 1) {
727             map[i] = 1;
728             continue;
729         }
730         if (rand() % 10 < 2 && manhattan_distance(row, col, row_p0, col_p0) > 10 &&
731             manhattan_distance(row, col, row_p1, col_p1) > 10) {
732             map[i] = 2;
733             continue;
734         }
735         map[i] = 0;
736     }
737     for (int i = 0; i < 100; i++) {
738         write_tile(i * 12, map[i * 12], &global_info.map_change_0);
739         write_tile(i * 12 + 1, map[i * 12 + 1], &global_info.map_change_1);
740         write_tile(i * 12 + 2, map[i * 12 + 2], &global_info.map_change_2);
741         write_tile(i * 12 + 3, map[i * 12 + 3], &global_info.map_change_3);
742         write_tile(i * 12 + 4, map[i * 12 + 4], &global_info.map_change_4);
743         write_tile(i * 12 + 5, map[i * 12 + 5], &global_info.map_change_5);
744         write_tile(i * 12 + 6, map[i * 12 + 6], &global_info.map_change_6);
745         write_tile(i * 12 + 7, map[i * 12 + 7], &global_info.map_change_7);
746         write_tile(i * 12 + 8, map[i * 12 + 8], &global_info.map_change_8);
747         write_tile(i * 12 + 9, map[i * 12 + 9], &global_info.map_change_9);
748         write_tile(i * 12 + 10, map[i * 12 + 10], &global_info.map_change_10);
749         write_tile(i * 12 + 11, map[i * 12 + 11], &global_info.map_change_11);
750         pass_game_info();
751         usleep(2000);
752     }
753 }
754 int is_player_moving(player_info_t *info)

```

```

755 {
756     if (info -> status == STATIC)
757         return 0;
758     else
759         return 1;
760 }
761
762 enum FACING get_player_facing(player_info_t *info)
763 {
764     return info -> facing;
765 }
766
767 void get_player_vpos(uint16_t *pos, player_info_t *info)
768 {
769     pos[0] = info -> vxpos;
770     pos[1] = info -> vypos;
771 }
772
773 uint16_t get_player_vspeed(player_info_t *info)
774 {
775     return info -> vspeed;
776 }
777
778 void set_player_moving (int moving, player_info_t *info)
779 {
780     if (moving)
781         info -> status = MOVING;
782     else
783         info -> status = STATIC;
784 }
785
786 void set_player_pos (uint16_t *pos, player_info_t *info)
787 {
788     info -> vxpos = pos[0];
789     info -> vypos = pos[1];
790 }
791
792 void set_player_sprite(enum POSE pos, player_info_t *info)
793 {
794     info -> pose = pos;
795 }
796
797 void set_player_facing(enum FACING facing, player_info_t *info)
798 {
799     info -> facing = facing;
800 }
801
802 int detect_static_flame_collision(uint16_t *pos) {
803     int xpos = (int) pos[0];
804     int ypos = (int) pos[1];
805     int blocktlx = (xpos - 6) / 16;
806     int blocktly = (ypos - 6) / 16;
807     int blocktrx = (xpos + 7) / 16;
808     int blocktry = (ypos - 6) / 16;
809     int blockblx = (xpos - 6) / 16;
810     int blockbly = (ypos + 7) / 16;

```

```

811     int blockbrx = (xpos + 7) / 16;
812     int blockbry = (ypos + 7) / 16;
813     uint16_t tile_tl = map[blocktly * 40 + blocktlx];
814     uint16_t tile_tr = map[blocktry * 40 + blocktrx];
815     uint16_t tile_bl = map[blockbly * 40 + blockblx];
816     uint16_t tile_br = map[blockbry * 40 + blockbrx];
817     return (is_flame(tile_tl) || is_flame(tile_tr) || is_flame(tile_br) ||
            is_flame(tile_bl));
818 }
819 void handle_player_movement()
820 {
821     uint16_t player0_curr_vpos[2];
822     uint16_t player1_curr_vpos[2];
823     uint16_t player0_attempt_vpos[2];
824     uint16_t player1_attempt_vpos[2];
825     get_player_vpos(player0_curr_vpos, &player0_info);
826     get_player_vpos(player1_curr_vpos, &player1_info);
827     memcpy((void *) player0_attempt_vpos, (void *) player0_curr_vpos, 2 *
            sizeof(uint16_t));
828     memcpy((void *) player1_attempt_vpos, (void *) player1_curr_vpos, 2 *
            sizeof(uint16_t));
829     if (control_info.direction0 == get_player_facing(&player0_info) &&
        control_info.press_tick0 > 10 && control_info.idle0 == 0) {
830         uint16_t player0_vs = get_player_vspeed(&player0_info);
831         switch (control_info.direction0) {
832             case DOWN:
833                 player0_attempt_vpos[1] += player0_vs;
834                 break;
835             case LEFT:
836                 player0_attempt_vpos[0] -= player0_vs;
837                 break;
838             case UP:
839                 player0_attempt_vpos[1] -= player0_vs;
840                 break;
841             case RIGHT:
842                 player0_attempt_vpos[0] += player0_vs;
843                 break;
844         }
845         uint16_t player0_attempt_xpos = player0_attempt_vpos[0] / 3;
846         uint16_t player0_attempt_ypos = player0_attempt_vpos[1] / 3;
847         int num_corner = 0;
848         int fit = 0;
849         int blocktlx = (player0_attempt_xpos - 6) / 16;
850         int blocktly = (player0_attempt_ypos - 6) / 16;
851         if (is_obstacle(map[blocktly * 40 + blocktlx])){
852             if (is_flame(map[blocktly * 40 + blocktlx]))
853                 player0_info.dead = 1;
854             num_corner++;
855             if (get_player_facing(&player0_info) == LEFT){
856                 fit = 0; //move down
857             }
858             else{
859                 fit = 1; //move right
860             }
861         }
862     }

```

```

863     int blocktrx = (player0_attempt_xpos + 7) / 16;
864     int blocktry = (player0_attempt_ypos - 6) / 16;
865     if (is_obstacle(map[blocktry * 40 + blocktrx])){
866         if (is_flame(map[blocktry * 40 + blocktrx]))
867             player0_info.dead = 1;
868         num_corner++;
869         if (get_player_facing(&player0_info) == RIGHT){
870             fit = 0; //move down
871         }
872         else{
873             fit = 2; //move left
874         }
875     }
876
877     int blockblx = (player0_attempt_xpos - 6) / 16;
878     int blockbly = (player0_attempt_ypos + 7) / 16;
879     if (is_obstacle(map[blockbly * 40 + blockblx])){
880         if (is_flame(map[blockbly * 40 + blockblx]))
881             player0_info.dead = 1;
882         num_corner++;
883         if (get_player_facing(&player0_info) == LEFT){
884             fit = 3; //move up
885         }
886         else{
887             fit = 1; //move right
888         }
889     }
890
891     int blockbrx = (player0_attempt_xpos + 7) / 16;
892     int blockbry = (player0_attempt_ypos + 7) / 16;
893     if (is_obstacle(map[blockbry * 40 + blockbrx])){
894         if (is_flame(map[blockbry * 40 + blockbrx]))
895             player0_info.dead = 1;
896         num_corner++;
897         if (get_player_facing(&player0_info) == RIGHT){
898             fit = 3; //move up
899         }
900         else{
901             fit = 2; //move left
902         }
903     }
904     if (num_corner == 1){
905         memcpy((void *) player0_attempt_vpos, (void *) player0_curr_vpos, 2 *
906             sizeof(uint16_t));
907         switch(fit){
908             case 0:
909                 player0_attempt_vpos[1] += 1;
910                 break;
911             case 1:
912                 player0_attempt_vpos[0] += 1;
913                 break;
914             case 2:
915                 player0_attempt_vpos[0] -= 1;
916                 break;
917             case 3:
918                 player0_attempt_vpos[1] -= 1;

```

```

918         break;
919     }
920 }
921 if (num_corner > 1){
922     memcpy((void *) player0_attempt_vpos, (void *) player0_curr_vpos, 2 *
           sizeof(uint16_t));
923 }
924 if (player0_attempt_xpos > PLAYER_X_UPPER_LIM || player0_attempt_xpos <
     PLAYER_X_LOWER_LIM || player0_attempt_ypos > PLAYER_Y_UPPER_LIM ||
     player0_attempt_ypos < PLAYER_Y_LOWER_LIM)
925     memcpy((void *) player0_attempt_vpos, (void *) player0_curr_vpos, 2 *
           sizeof(uint16_t));
926 if (abs_diff(player0_attempt_xpos, player1_curr_vpos[0] / 3) < 16 &&
     abs_diff(player0_attempt_ypos, player1_curr_vpos[1] / 3) < 16)
927     memcpy((void *) player0_attempt_vpos, (void *) player0_curr_vpos, 2 *
           sizeof(uint16_t));
928 set_player_pos(player0_attempt_vpos, &player0_info);
929 printf("player0_attempt_vpos = (%d, %d)\n", player0_attempt_xpos,
     player0_attempt_ypos);
930 set_player_moving(1, &player0_info);
931 /*Determine the pose*/
932 uint16_t pos_tick = ++ player0_info.pos_tick;
933 if (pos_tick > 1000)
934     pos_tick = 0;
935 player0_info.pos_tick = pos_tick;
936 enum POSE pos;
937 switch (control_info.direction0) {
938     case DOWN:
939         if ((pos_tick / 20) % 2 == 1) pos = DOWN1;
940         else pos = DOWN0;
941         break;
942     case LEFT:
943         if ((pos_tick / 20) % 3 == 1) pos = SIDE1;
944         else if ((pos_tick / 20) % 3 == 2) pos = SIDE2;
945         else pos = SIDE0;
946         break;
947     case UP:
948         if ((pos_tick / 20) % 2 == 1) pos = UP1;
949         else pos = UP0;
950         break;
951     case RIGHT:
952         if ((pos_tick / 20) % 3 == 1) pos = SIDE1;
953         else if ((pos_tick / 20) % 3 == 2) pos = SIDE2;
954         else pos = SIDE0;
955         break;
956 }
957 set_player_sprite(pos, &player0_info);
958
959
960 }
961 else {
962     /*Turn*/
963     uint16_t player0_vpos[2];
964     uint16_t player0_pos[2];
965     get_player_vpos(player0_vpos, &player0_info);
966     player0_pos[0] = player0_vpos[0] / 3;

```



```

967     player0_pos[1] = player0_vpos[1] / 3;
968     if (detect_static_flame_collision(player0_pos))
969         player0_info.dead = 1;
970     set_player_facing(control_info.direction0, &player0_info);
971     set_player_moving(0, &player0_info);
972     player0_info.pos_tick = 0;
973     set_player_sprite(IDLE, &player0_info);
974 }
975 /*Update p0 pos*/
976 get_player_vpos(player0_curr_vpos, &player0_info);
977
978 /*Player1*/
979 if (control_info.direction1 == get_player_facing(&player1_info) &&
    control_info.press_tick1 > 10 && control_info.idle1 == 0) {
980     uint16_t player1_vs = get_player_vspeek(&player1_info);
981     switch (control_info.direction1) {
982         case DOWN:
983             player1_attempt_vpos[1] += player1_vs;
984             break;
985         case LEFT:
986             player1_attempt_vpos[0] -= player1_vs;
987             break;
988         case UP:
989             player1_attempt_vpos[1] -= player1_vs;
990             break;
991         case RIGHT:
992             player1_attempt_vpos[0] += player1_vs;
993             break;
994     }
995
996     uint16_t player1_attempt_xpos = player1_attempt_vpos[0] / 3;
997     uint16_t player1_attempt_ypos = player1_attempt_vpos[1] / 3;
998
999     int num_corner = 0;
1000    int fit = 0;
1001    int blocktlx = (player1_attempt_xpos - 6) / 16;
1002    int blocktly = (player1_attempt_ypos - 6) / 16;
1003    if (is_obstacle(map[blocktly * 40 + blocktlx])){
1004        if (is_flame(map[blocktly * 40 + blocktlx]))
1005            player1_info.dead = 1;
1006        num_corner++;
1007        if (get_player_facing(&player1_info) == LEFT){
1008            fit = 0; //move down
1009        }
1010        else{
1011            fit = 1; //move right
1012        }
1013    }
1014
1015    int blocktrx = (player1_attempt_xpos + 7) / 16;
1016    int blocktry = (player1_attempt_ypos - 6) / 16;
1017    if (is_obstacle(map[blocktry * 40 + blocktrx])){
1018        if (is_flame(map[blocktry * 40 + blocktrx]))
1019            player1_info.dead = 1;
1020        num_corner++;
1021        if (get_player_facing(&player1_info) == RIGHT){

```

```

1022         fit = 0; //move down
1023     }
1024     else{
1025         fit = 2; //move left
1026     }
1027 }
1028
1029 int blockblx = (player1_attempt_xpos - 6) / 16;
1030 int blockbly = (player1_attempt_ypos + 7) / 16;
1031 if (is_obstacle(map[blockbly * 40 + blockblx])){
1032     if (is_flame(map[blockbly * 40 + blockblx]))
1033         player1_info.dead = 1;
1034     num_corner++;
1035     if (get_player_facing(&player1_info) == LEFT){
1036         fit = 3; //move up
1037     }
1038     else{
1039         fit = 1; //move right
1040     }
1041 }
1042
1043 int blockbrx = (player1_attempt_xpos + 7) / 16;
1044 int blockbry = (player1_attempt_ypos + 7) / 16;
1045 if (is_obstacle(map[blockbry * 40 + blockbrx])){
1046     if (is_flame(map[blockbry * 40 + blockbrx]))
1047         player1_info.dead = 1;
1048     num_corner++;
1049     if (get_player_facing(&player1_info) == RIGHT){
1050         fit = 3; //move up
1051     }
1052     else{
1053         fit = 2; //move left
1054     }
1055 }
1056 if (num_corner == 1){
1057     memcpy((void *) player1_attempt_vpos, (void *) player1_curr_vpos, 2 *
1058         sizeof(uint16_t));
1059     switch(fit){
1060     case 0:
1061         player1_attempt_vpos[1] += 1;
1062         break;
1063     case 1:
1064         player1_attempt_vpos[0] += 1;
1065         break;
1066     case 2:
1067         player1_attempt_vpos[0] -= 1;
1068         break;
1069     case 3:
1070         player1_attempt_vpos[1] -= 1;
1071         break;
1072     }
1073 }
1074 if (num_corner > 1){
1075     memcpy((void *) player1_attempt_vpos, (void *) player1_curr_vpos, 2 *
1076         sizeof(uint16_t));

```

```

1076     if (player1_attempt_xpos > PLAYER_X_UPPER_LIM || player1_attempt_xpos <
        PLAYER_X_LOWER_LIM || player1_attempt_ypos > PLAYER_Y_UPPER_LIM ||
        player1_attempt_ypos < PLAYER_Y_LOWER_LIM)
1077         memcpy((void *) player1_attempt_vpos, (void *) player1_curr_vpos, 2 *
                sizeof(uint16_t));
1078     if ((abs_diff(player1_attempt_xpos, (player0_curr_vpos[0] / (uint16_t) 3)) <
        (uint16_t) 16) && (abs_diff(player1_attempt_ypos, (player0_curr_vpos[1]
        / (uint16_t) 3)) < (uint16_t) 16))
1079         memcpy((void *) player1_attempt_vpos, (void *) player1_curr_vpos, 2 *
                sizeof(uint16_t));
1080     set_player_pos(player1_attempt_vpos, &player1_info);
1081     printf("player1_attempt_vpos = (%d, %d)\n", player1_attempt_xpos,
            player1_attempt_ypos);
1082     set_player_moving(1, &player1_info);
1083     /*Determine the pose*/
1084     uint16_t pos_tick = ++ player1_info.pos_tick;
1085     if (pos_tick > 1000)
1086         pos_tick = 0;
1087     player1_info.pos_tick = pos_tick;
1088     enum POSE pos;
1089     switch (control_info.direction1) {
1090     case DOWN:
1091         if ((pos_tick / 20) % 2 == 1) pos = DOWN1;
1092         else pos = DOWN0;
1093         break;
1094     case LEFT:
1095         if ((pos_tick / 20) % 3 == 1) pos = SIDE1;
1096         else if ((pos_tick / 20) % 3 == 2) pos = SIDE2;
1097         else pos = SIDE0;
1098         break;
1099     case UP:
1100         if ((pos_tick / 20) % 2 == 1) pos = UP1;
1101         else pos = UP0;
1102         break;
1103     case RIGHT:
1104         if ((pos_tick / 20) % 3 == 1) pos = SIDE1;
1105         else if ((pos_tick / 20) % 3 == 2) pos = SIDE2;
1106         else pos = SIDE0;
1107         break;
1108     }
1109     set_player_sprite(pos, &player1_info);
1110
1111
1112 }
1113 else {
1114     uint16_t player1_vpos[2];
1115     uint16_t player1_pos[2];
1116     get_player_vpos(player1_vpos, &player1_info);
1117     player1_pos[0] = player1_vpos[0] / 3;
1118     player1_pos[1] = player1_vpos[1] / 3;
1119     if (detect_static_flame_collision(player1_pos))
1120         player1_info.dead = 1;
1121     /*Turn*/
1122     set_player_facing(control_info.direction1, &player1_info);
1123     set_player_moving(0, &player1_info);
1124     player1_info.pos_tick = 0;

```

```

1125     set_player_sprite(IDLE, &player1_info);
1126 }
1127
1128 }
1129
1130 int can_player_place_bomb(player_info_t *info) {
1131     return (info->bomb_colddown == 0) && (info->bombs_left > 0);
1132 }
1133 /*Change the memory*/
1134 int map_change_list_append_tile(uint16_t pos, uint16_t tile)
1135 {
1136     if (map_change_list_next >= 12)
1137         return -1;
1138     else {
1139         write_tile(pos, tile, &map_change_list[map_change_list_next]);
1140         map_change_list_next ++;
1141         return 0;
1142     }
1143 }
1144
1145
1146 void handle_player_place_bomb ()
1147 {
1148     int can_player0_place_bomb = can_player_place_bomb(&player0_info);
1149     if (control_info.attempt_place_bomb_0 || control_info.attempt_place_bomb_1) {
1150         printf("Player 0: bombs left %d, bomb colddown %d\n",
1151             player0_info.bombs_left, player0_info.bomb_colddown);
1152         printf("Player 1: bombs left %d, bomb colddown %d\n",
1153             player1_info.bombs_left, player1_info.bomb_colddown);
1154     }
1155     if (can_player0_place_bomb && control_info.attempt_place_bomb_0) {
1156         enum FACING facing = get_player_facing(&player0_info);
1157         uint16_t curr_vpos[2];
1158         get_player_vpos(curr_vpos, &player0_info);
1159         int player_x = (int) curr_vpos[0] / 3;
1160         int player_y = (int) curr_vpos[1] / 3;
1161         int attempt_pos_r;
1162         int attempt_pos_c;
1163         int player_r = player_y / 16;
1164         int player_c = player_x / 16;
1165         switch (facing) {
1166             case DOWN:
1167                 attempt_pos_r = player_r + 1;
1168                 attempt_pos_c = player_c;
1169                 break;
1170             case LEFT:
1171                 attempt_pos_r = player_r;
1172                 attempt_pos_c = player_c - 1;
1173                 break;
1174             case UP:
1175                 attempt_pos_r = player_r - 1;
1176                 attempt_pos_c = player_c;
1177                 break;
1178             case RIGHT:
1179                 attempt_pos_r = player_r;
1180                 attempt_pos_c = player_c + 1;

```

```

1179         break;
1180     }
1181     if (attempt_pos_r >= 0 && attempt_pos_r < 30 && attempt_pos_c >= 0 &&
1182         attempt_pos_c < 40) {
1183         int attempt_center_x = attempt_pos_c * 16 + 7;
1184         int attempt_center_y = attempt_pos_r * 16 + 7;
1185         int md = abs_diff(player_x, attempt_center_x) + abs_diff(player_y,
1186             attempt_center_y);
1187         int attempt_pos = attempt_pos_c + attempt_pos_r * 40;
1188         if (map[attempt_pos] == 0 && md > 12) {
1189             if (map[attempt_pos] == 0) {
1190                 if (map_change_list_append_tile((uint16_t) attempt_pos, (uint16_t)
1191                     3) == 0) {
1192                     map[attempt_pos] = 3;
1193                     if (md <= 16) {
1194                         uint16_t player_next_vpos[2];
1195                         player_next_vpos[0] = (uint16_t) (player_c * 16 + 8)* 3;
1196                         player_next_vpos[1] = (uint16_t) (player_r * 16 + 8)* 3;
1197                         set_player_pos(player_next_vpos, &player0_info);
1198                     }
1199                     insert_bomb(PAYER0, attempt_pos);
1200                     player0_info.bombs_left --;
1201                     player0_info.bomb_colddown = 50;
1202                 }
1203             }
1204         }
1205     }
1206     else {
1207         if (player0_info.bomb_colddown > 0)
1208             player0_info.bomb_colddown --;
1209     }
1210     int can_player1_place_bomb = can_player_place_bomb(&player1_info);
1211     if (can_player1_place_bomb && control_info.attempt_place_bomb_1) {
1212         enum FACING facing = get_player_facing(&player1_info);
1213         uint16_t curr_vpos[2];
1214         get_player_vpos(curr_vpos, &player1_info);
1215         int player_x = (int) curr_vpos[0] / 3;
1216         int player_y = (int) curr_vpos[1] / 3;
1217         int attempt_pos_r;
1218         int attempt_pos_c;
1219         int player_r = player_y / 16;
1220         int player_c = player_x / 16;
1221         switch (facing) {
1222             case DOWN:
1223                 attempt_pos_r = player_r + 1;
1224                 attempt_pos_c = player_c;
1225                 break;
1226             case LEFT:
1227                 attempt_pos_r = player_r;
1228                 attempt_pos_c = player_c - 1;
1229                 break;
1230             case UP:
1231                 attempt_pos_r = player_r - 1;
1232                 attempt_pos_c = player_c;

```

```

1232         break;
1233     case RIGHT:
1234         attempt_pos_r = player_r;
1235         attempt_pos_c = player_c + 1;
1236         break;
1237     }
1238     if (attempt_pos_r >= 0 && attempt_pos_r < 30 && attempt_pos_c >= 0 &&
        attempt_pos_c < 40) {
1239
1240         int attempt_center_x = attempt_pos_c * 16 + 7;
1241         int attempt_center_y = attempt_pos_r * 16 + 7;
1242         int md = abs_diff(player_x, attempt_center_x) + abs_diff(player_y,
            attempt_center_y);
1243         int attempt_pos = attempt_pos_c + attempt_pos_r * 40;
1244         if (map[attempt_pos] == 0 && md > 12) {
1245             if (map[attempt_pos] == 0) {
1246                 if (map_change_list_append_tile((uint16_t) attempt_pos, (uint16_t)
                    3) == 0) {
1247                     map[attempt_pos] = 3;
1248                     if (md <= 16) {
1249                         uint16_t player_next_vpos[2];
1250                         player_next_vpos[0] = (uint16_t) (player_c * 16 + 8)* 3;
1251                         player_next_vpos[1] = (uint16_t) (player_r * 16 + 8)* 3;
1252                         set_player_pos(player_next_vpos, &player1_info);
1253
1254                     }
1255                     insert_bomb(PPLAYER1, attempt_pos);
1256                     player1_info.bombs_left --;
1257                     player1_info.bomb_colddown = 50;
1258                 }
1259             }
1260         }
1261     }
1262 }
1263 else {
1264     if (player1_info.bomb_colddown > 0)
1265         player1_info.bomb_colddown --;
1266 }
1267 }

```

## H main.c

```

1     #include <stdio.h>
2     #include <sys/ioctl.h>
3     #include <sys/types.h>
4     #include <sys/stat.h>
5     #include <fcntl.h>
6     #include <string.h>
7     #include <unistd.h>
8     #include <stdlib.h>
9     #include <time.h>
10    #include <pthread.h>
11    #include "controller.h"
12    #include "bomberman.h"

```

```

13 extern int vga_fd;
14 extern game_info_t global_info;
15 extern player_info_t player0_info;
16 extern player_info_t player1_info;
17 extern control_info_t control_info;
18 extern uint16_t *map;
19 extern struct controller_list controllers;
20 extern struct args_list c_args_list;
21 extern struct Bomb* bombs_head;
22 extern struct Explosion *explosion_head;
23
24 int main()
25 {
26     controllers = open_controllers();
27     pthread_t control_thread;
28     memset((void *) &control_info, 0, sizeof(control_info_t));
29     c_args_list.mode = 0;
30     c_args_list.print = 1;
31     c_args_list.devices = controllers;
32     c_args_list.control_info = &control_info;
33
34
35     if (pthread_create(&control_thread, NULL, &listen_controllers, (void *)
36         &c_args_list)) {
37         fprintf(stderr, "Could not create controller thread\n");
38         return -1;
39     }
40     static const char filename[] = "/dev/vga_ball";
41     printf("VGA ball Userspace program started\n");
42
43     if ( (vga_fd = open(filename, O_RDWR)) == -1) {
44         fprintf(stderr, "could not open %s\n", filename);
45         return -1;
46     }
47     memset((void *) &global_info, 0, sizeof(game_info_t));
48     generate_software_map();
49     init_players();
50     /*for (int i = 0; i < 10000; i++){
51         xpos0 += v0;
52         xpos1 += v1;
53         pos_counter ++;
54         pos_counter = pos_counter % 18;
55         if (xpos0 > PLAYER_X_UPPER_LIM) {
56             xpos0 = PLAYER_X_UPPER_LIM;
57             facing0 = LEFT;
58             v0 = -1;
59         }
60         if (xpos0 < PLAYER_X_LOWER_LIM) {
61             xpos0 = PLAYER_X_LOWER_LIM;
62             facing0 = RIGHT;
63             v0 = 1;
64         }
65         if (xpos1 > PLAYER_X_UPPER_LIM) {
66             xpos1 = PLAYER_X_UPPER_LIM;
67             facing1 = LEFT;
68             v1 = -1;

```

```

68     }
69     if (xpos1 < PLAYER_X_LOWER_LIM) {
70         xpos1 = PLAYER_X_LOWER_LIM;
71         facing1 = RIGHT;
72         v1 = 1;
73     }
74     if (pos_counter / 6 == 0)
75         pos = SIDE0;
76     else if (pos_counter / 6 == 1)
77         pos = SIDE1;
78     else pos = SIDE2;
79
80     set_player_status(xpos0, 136, facing0, MOVING, pos, PLAYER0);
81     set_player_status(xpos1, 400, facing1, MOVING, pos, PLAYER1);
82     write_player_info();
83     pass_game_info();
84     usleep(20000);
85 }*/
86 bombs_head = NULL;
87 explosion_head = NULL;
88 for(;;) {
89     init_explosion_sound();
90     reset_map_change_list();
91     handle_explosion();
92     handle_bomb_explode();
93     handle_player_movement();
94     if (player0_info.dead || player1_info.dead)
95         break;
96     handle_player_place_bomb();
97     handle_player_prop_get();
98     write_player_info();
99     sync_hw_map_change();
100    pass_game_info();
101    usleep(10000);
102 }
103 display_game_over();
104 pthread_cancel(control_thread);
105 pthread_join(control_thread, NULL);
106 free(map);
107 free_bombs();
108 free_explosion();
109 return 0;
110 }

```

## I Makefile

```

1     ifneq (${KERNELRELEASE},)
2
3 # KERNELRELEASE defined: we are being compiled as part of the Kernel
4     obj-m := vga.o
5
6 else
7
8 # We are being compiled as a module: use the Kernel build system

```



```
9
10     KERNEL_SOURCE := /usr/src/linux-headers-$(shell uname -r)
11     PWD := $(shell pwd)
12
13 default: module main
14
15 module:
16     ${MAKE} -C ${KERNEL_SOURCE} SUBDIRS=${PWD} modules
17 main: controller.o bomberman.o main.o
18     gcc -o software_moving controller.o bomberman.o main.o -lusb-1.0 -lpthread
19 main.o: main.c
20     gcc -o main.o -c main.c -lusb-1.0 -lpthread
21 bomberman.o: bomberman.c
22     gcc -o bomberman.o -c bomberman.c -lusb-1.0 -lpthread
23 controller.o: controller.c
24     gcc -o controller.o -c controller.c -lusb-1.0 -lpthread
25 clean:
26     ${MAKE} -C ${KERNEL_SOURCE} SUBDIRS=${PWD} clean
27     rm *.o
28 TARFILES = Makefile README hello.h vga.c software_moving.c
29 TARFILE = lab3-sw.tar.gz
30 .PHONY : tar
31 tar : $(TARFILE)
32
33 $(TARFILE) : $(TARFILES)
34     tar zcfc $(TARFILE) .. $(TARFILES:%=lab3-sw/%)
35 endif
```