Final Report for Crazy Arcade Game

Yuqi Lin, Yelin Mao, Hongkuan Yu
UNI: yl5334, ym3000, hy2819

Spring 2024

Contents

1 Overview

1.1 Short Introduction

We develop a "Crazy Arcade" game (shown in Figure 1 below) on the DE1-SoC board, leveraging SystemVerilog for hardware implementation and C for the game's algorithm. The game features two players competing in real-time, navigating through a maze, and placing bombs to destroy another player. Bombs will explode after a certain time, with the affected distance as a "+" (up, down, left, and right).

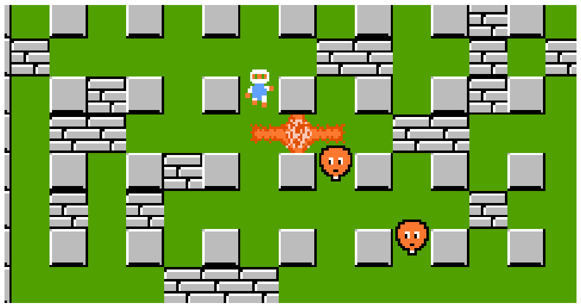


Figure 1: Crazy Arcade game

1.2 Game Rules

At the beginning of each game, players will spawn on opposite corners of the screen, surrounded by randomly generated obstacles. There are fixed obstacles and temporary obstacles. Players can reach each other through different paths and place bombs, exploding after a set of times, to block another's path and destroy temporary obstacles. The bomb has an affected distance, and within this distance, the explosion will kill nearby players. So it is obvious that if obstacles and a bomb block the path of a player, then he has no chance but only waits for the explosion and gets killed! If one player is killed, the surviving player wins the game. After each game concludes, players respawn at their starting positions with a new randomly generated map.

2 System Block Diagram

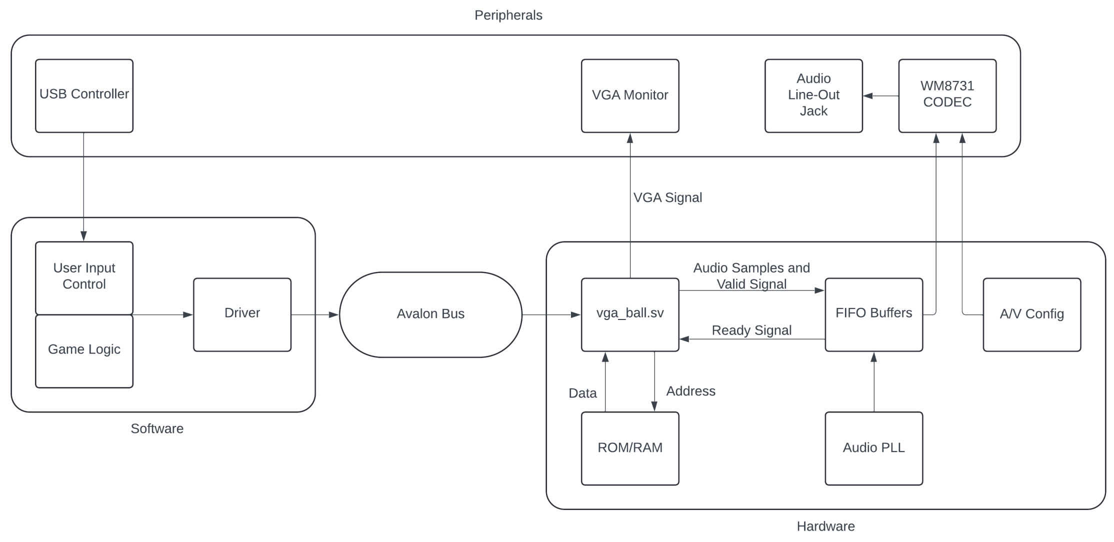


Figure 2: System Block Diagram

Players control characters using gaming controllers provided by Professor Edwards, which interface with the game's software via USB protocol. The software communicates with the Avalon Bus via a device driver; the software is responsible for passing the correct tile and pixel coordinates to the hardware and for triggering audio.

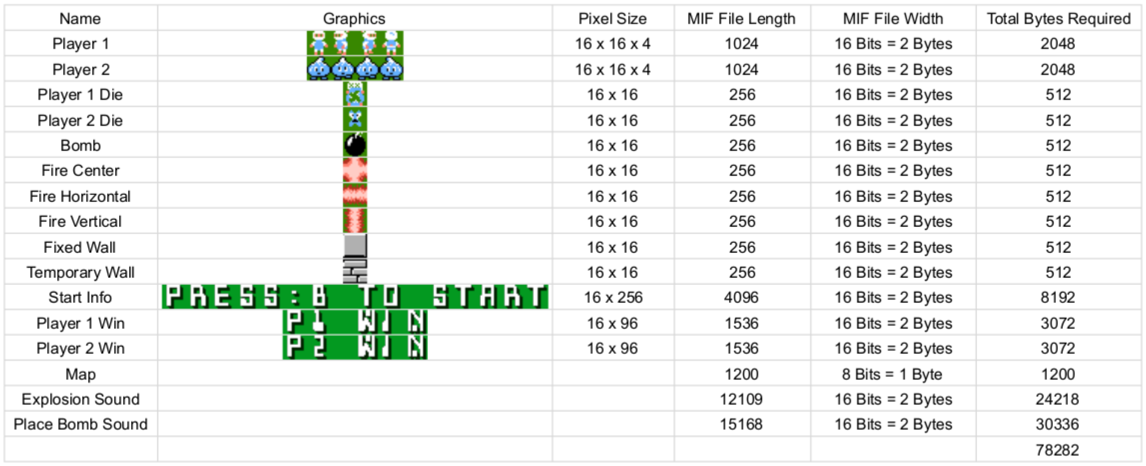
The primary game logic is in the "main.c" file. The "controller.c" file identifies inputs from gaming controllers and enables the execution of game logic through USB protocol and the *libusb* library. The device driver in the "vga_ball.c" file communicates with the "vga_ball.sv" via the Avalon bus to refresh graphics on the VGA monitor according to game logic.

Peripherals are gaming controllers and the VGA monitor. On-chip memory ROMs store sprite data and audio, and "vga_ball.sv" decodes addresses to the ROMs and then gets the corresponding data back. The data are fed to the VGA Monitor or Audio Jack accordingly.

3 Hardware

3.1 Total Memory Usage

Based on the specification sheet of DE1-SoC, we know that the FPGA includes 556,250 bytes of embedded memory. The sprites and audio required for our project are listed in Figure 3.

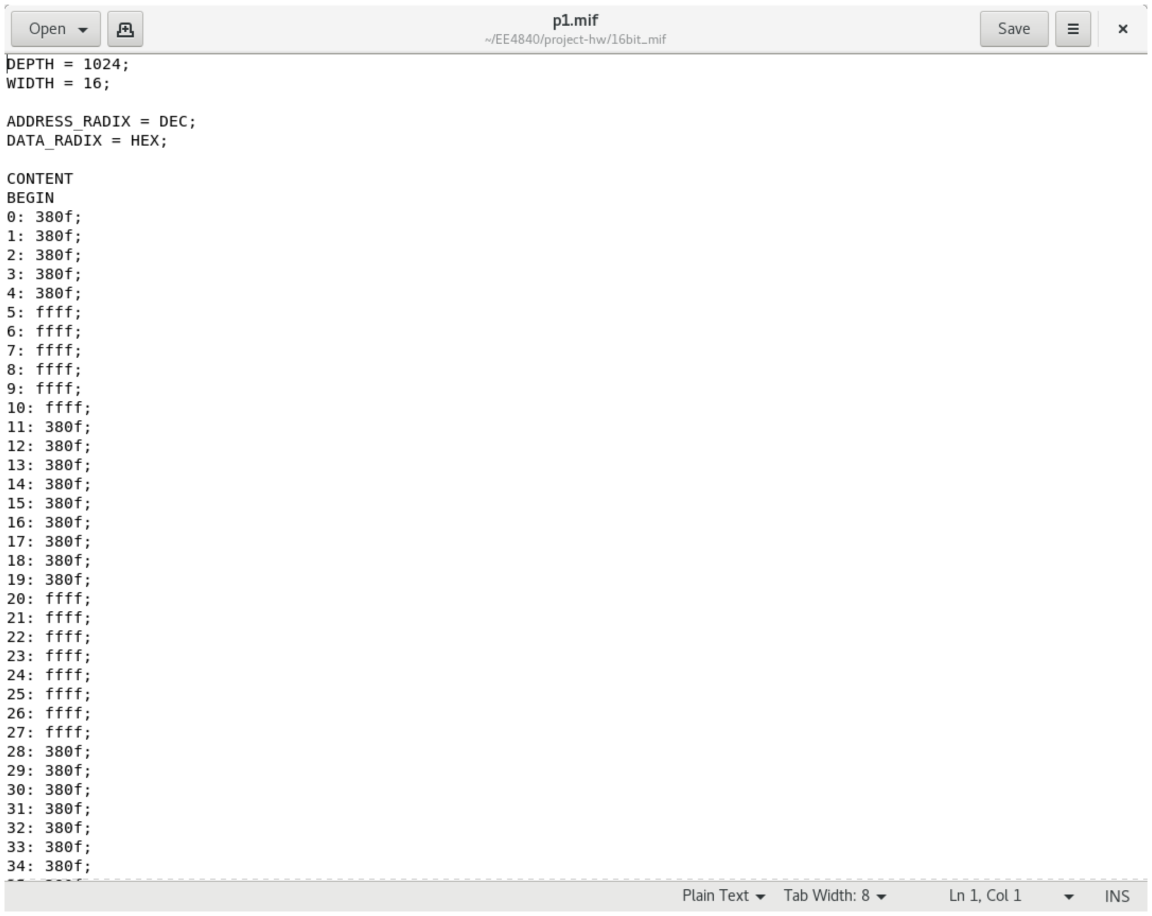


Name	Graphics	Pixel Size	MIF File Length	MIF File Width	Total Bytes Required
Player 1		16 x 16 x 4	1024	16 Bits = 2 Bytes	2048
Player 2		16 x 16 x 4	1024	16 Bits = 2 Bytes	2048
Player 1 Die		16 x 16	256	16 Bits = 2 Bytes	512
Player 2 Die		16 x 16	256	16 Bits = 2 Bytes	512
Bomb		16 x 16	256	16 Bits = 2 Bytes	512
Fire Center		16 x 16	256	16 Bits = 2 Bytes	512
Fire Horizontal		16 x 16	256	16 Bits = 2 Bytes	512
Fire Vertical		16 x 16	256	16 Bits = 2 Bytes	512
Fixed Wall		16 x 16	256	16 Bits = 2 Bytes	512
Temporary Wall		16 x 16	256	16 Bits = 2 Bytes	512
Start Info	PRESS:B TO START	16 x 256	4096	16 Bits = 2 Bytes	8192
Player 1 Win	P1 WIN	16 x 96	1536	16 Bits = 2 Bytes	3072
Player 2 Win	P2 WIN	16 x 96	1536	16 Bits = 2 Bytes	3072
Map			1200	8 Bits = 1 Byte	1200
Explosion Sound			12109	16 Bits = 2 Bytes	24218
Place Bomb Sound			15168	16 Bits = 2 Bytes	30336
					78282

Figure 3: Total Memory Usage

3.2 Graphics

The main SystemVerilog of hardware focuses on the logic required to render graphics. We collected sprites in .png format from the website "The Spriters Resource" and converted them to .mif files for the FPGA. Figure 4 below is the .mif file of the sprite of Player 1 as an example.

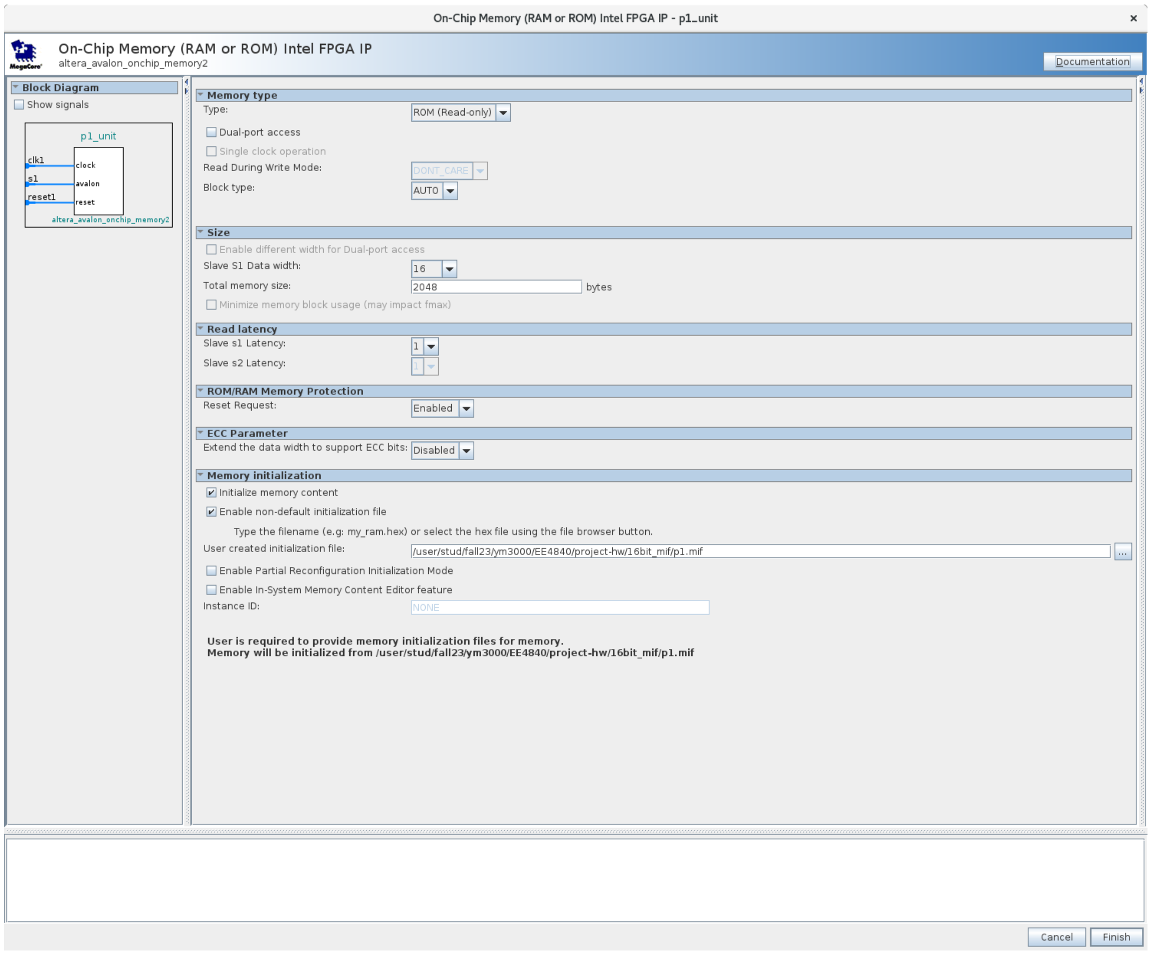


```
DEPTH = 1024;
WIDTH = 16;

ADDRESS_RADIX = DEC;
DATA_RADIX = HEX;

CONTENT
BEGIN
0: 380f;
1: 380f;
2: 380f;
3: 380f;
4: 380f;
5: ffff;
6: ffff;
7: ffff;
8: ffff;
9: ffff;
10: ffff;
11: 380f;
12: 380f;
13: 380f;
14: 380f;
15: 380f;
16: 380f;
17: 380f;
18: 380f;
19: 380f;
20: ffff;
21: ffff;
22: ffff;
23: ffff;
24: ffff;
25: ffff;
26: ffff;
27: ffff;
28: 380f;
29: 380f;
30: 380f;
31: 380f;
32: 380f;
33: 380f;
34: 380f;
```

Figure 4: The .mif file of Player 1

As shown in Figure 4, the addresses of the .mif file of each sprite are in decimal form, which makes it easier for us to write the SystemVerilog of hardware. The data of each address is 16-bit in hexadecimal, with the first 4-bit (first hex) counting from left representing degrees in Red, the second 4-bit (second hex) representing degrees in Green, and the third 4-bit (first hex) representing degrees in Blue. In SystemVerilog of hardware, we sliced the data based on this pattern and fed it to the VGA output.

The .mif files are stored in On-Chip Memory ROMs as an initialization that is designed and set up using the System Memory IP blocks in Platform Designer. According to the Total Memory Usage shown in the above sub-section, we configured them separately. The .mif file for the map uses RAM for us to randomly generate the layout of the map. The Figure 5 below is the configuration of Player 1 as an example.


On-Chip Memory (RAM or ROM) Intel FPGA IP - p1_unit
On-Chip Memory (RAM or ROM) Intel FPGA IP
altera_avalon_onchip_memory2
Documentation
Block Diagram
Show signals
p1_unit
clk1
s1
reset1
clock
avalon
reset
altera_avalon_onchip_memory2
Cancel
Finish

Memory type
Type: ROM (Read-only)
☐ Dual-port access
☐ Single clock operation
Read During Write Mode: DONT_CARE
Block type: AUTO

Size
☐ Enable different width for Dual-port access
Slave S1 Data width: 16
Total memory size: 2048 bytes
☐ Minimize memory block usage (may impact fmax)

Read latency
Slave s1 Latency: 1
Slave s2 Latency: 1

ROM/RAM Memory Protection
Reset Request: Enabled

ECC Parameter
Extend the data width to support ECC bits: Disabled

Memory initialization
☑ Initialize memory content
☑ Enable non-default initialization file
Type the filename (e.g: my_ram.hex) or select the hex file using the file browser button.
User created initialization file: /user/stud/fall23/ym3000/EE4840/project-hw/16bit_mif/p1.mif
☐ Enable Partial Reconfiguration Initialization Mode
☐ Enable In-System Memory Content Editor feature
Instance ID: NONE

User is required to provide memory initialization files for memory.
Memory will be initialized from /user/stud/fall23/ym3000/EE4840/project-hw/16bit_mif/p1.mif

Figure 5: ROM Configuration of Player 1

The main SystemVerilog of hardware is written in ”vga_ball.sv”. It contains submodules (generated by Platform Designer) for calculating screen position and assigning RGB pixel coordinates to the VGA output. Data is transferred from the software to ”vga_ball.sv” in 32-bit through the Avalon bus interface through the device driver, specifying when and where the corresponding sprite should be displayed. Using these inputs and the hcount and vcount coordinates, ”vga_ball.sv” sends addresses to the corresponding On-Chip Memory ROM of sprite and gets the 16-bit returned data. Then, ”vga_ball.sv” slices the returned data and determines the pixel color as described above. The Figure 6 below illustrates the architecture of graphics.

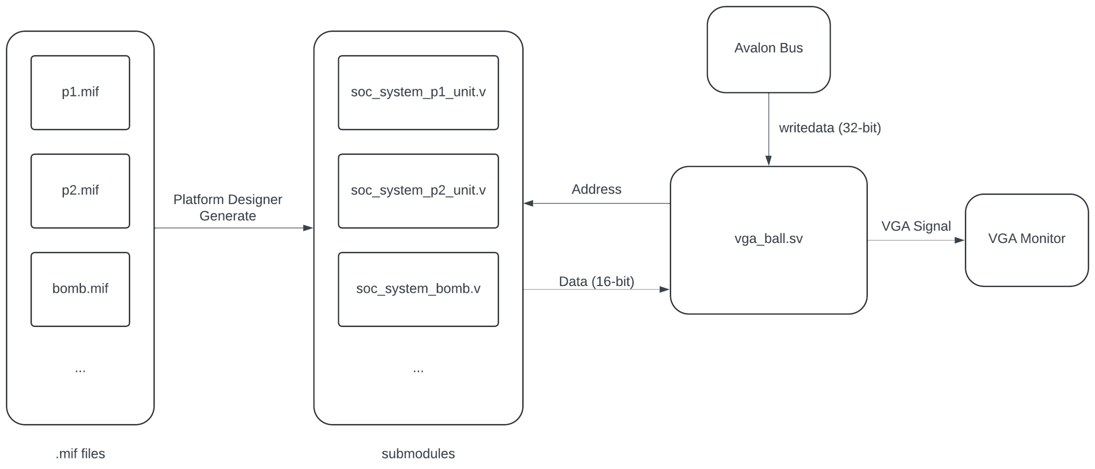


Figure 6: Graphics Architecture

3.3 Audio

We obtained the .mif file of our bomb placing and explosion sound in .wav format and converted them to .mif file for the FPGA. They are 8 kHz samples. The .mif file loads sound data into the On-Chip Memory ROM. We then adjusted the .qsys connection to ensure that the audio sample was correctly routed to the Wolfson WM8731 CODEC. The system involves three main components, as discussed in the following.

altera_up_avalon_audio_pll is used as a clock divider since the WM8731 CODEC does not support the standard 50 MHz clock; it uses the 50 MHz clock as a base to generate a 12.288 MHz clock.

alterra_up_avalon_audio_and_video_config configures our peripheral audio device.

altera_up_avalon_audio implements audio data transfer between the WM8731 CODEC and FPGA via FIFOs for the left and right audio channels.

These IP blocks together establish a data and clock path suitable for output via the Line Out Jack of the WM8731 CODEC. Figure 7 below shows the I/Os of the Wolfson WM8731 CODEC.

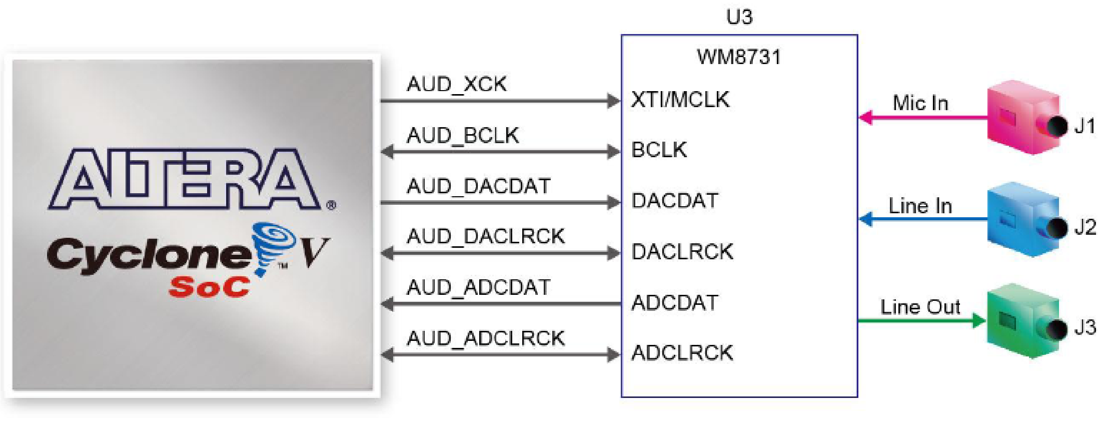


Figure 7: The I/Os of the Wolfson WM8731 CODEC

The architecture of the Audio is shown below in Figure 8.

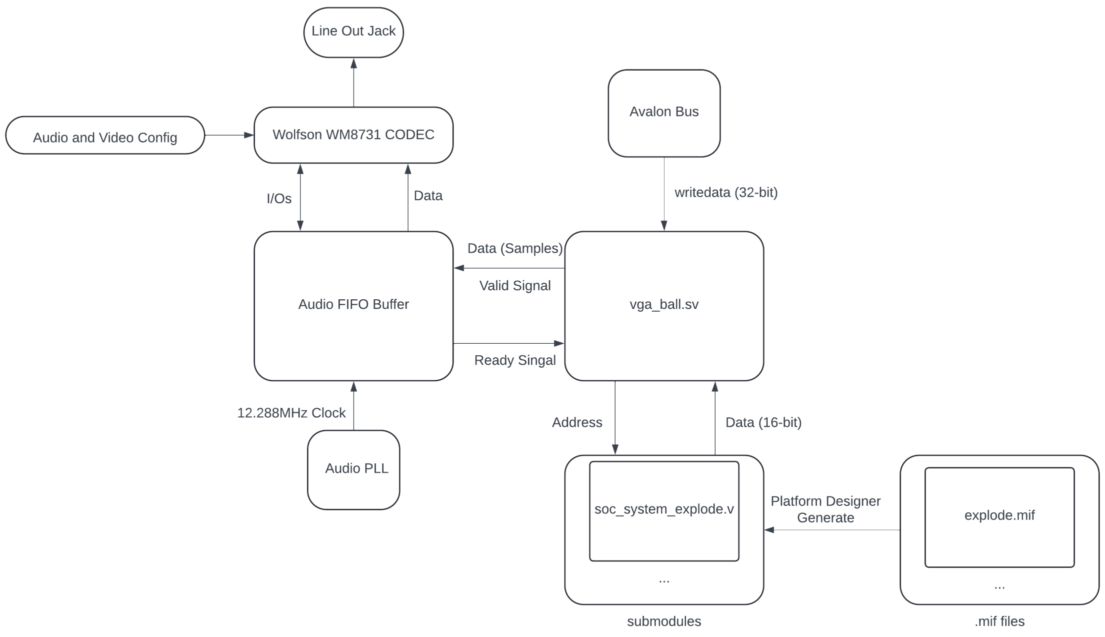
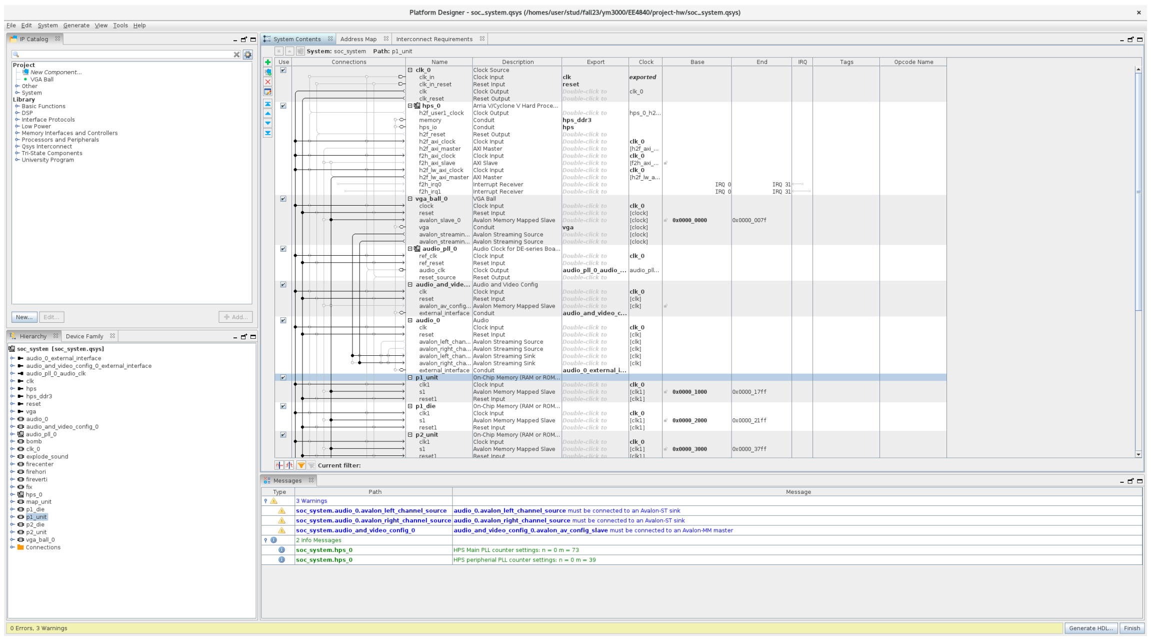


Figure 8: Audio Architecture

3.4 Connection (The .qsys File)

The final connection (the .qsys file) is shown below in Figure 9.



System: soc_system Path: p1_unit

Name	Description	Export	Clock	Base	End	IRQ
clk_0	Clock Source					
clk_in	Clock Input	clk	exported			
clk_in_reset	Reset Input	reset				
clk	Clock Output	Double-click to	clk_0			
clk_reset	Reset Output	Double-click to				
hps_0	Arria V/Cyclone V Hard Proce...					
h2f_user1_clock	Clock Output	Double-click to	hps_0_h2...			
memory	Conduit	hps_ddr3				
hps_io	Conduit	hps				
h2f_reset	Reset Output	Double-click to				
h2f_axi_clock	Clock Input	Double-click to	clk_0			
h2f_axi_master	AXI Master	Double-click to	[h2f_axi_...			
f2h_axi_clock	Clock Input	Double-click to	clk_0			
f2h_axi_slave	AXI Slave	Double-click to	[f2h_axi_...			
h2f_lw_axi_clock	Clock Input	Double-click to	clk_0			
h2f_lw_axi_master	AXI Master	Double-click to	[h2f_lw_a...			
f2h_irq0	Interrupt Receiver	Double-click to		IRQ 0	IRQ 31	
f2h_irq1	Interrupt Receiver	Double-click to		IRQ 0	IRQ 31	
vga_ball_0	VGA Ball					
clock	Clock Input	Double-click to	clk_0			
reset	Reset Input	Double-click to	[clock]			
avalon_slave_0	Avalon Memory Mapped Slave	Double-click to	[clock]	0x0000_0000	0x0000_007f	
vga	Conduit	vga	[clock]			
avalon_streamin...	Avalon Streaming Source	Double-click to	[clock]			
avalon_streamin...	Avalon Streaming Source	Double-click to	[clock]			
audio_pll_0	Audio Clock for DE-series Boa...					
ref_clk	Clock Input	Double-click to	clk_0			
ref_reset	Reset Input	Double-click to				
audio_clk	Clock Output	audio_pll_0_audio_...	audio_pll...			
reset_source	Reset Output	Double-click to				
audio_and_vide...	Audio and Video Config					
clk	Clock Input	Double-click to	clk_0			
reset	Reset Input	Double-click to	[clk]			
avalon_av_config...	Avalon Memory Mapped Slave	Double-click to	[clk]			
external_interface	Conduit	audio_and_video_c...				
audio_0	Audio					
clk	Clock Input	Double-click to	clk_0			
reset	Reset Input	Double-click to	[clk]			
avalon_left_chan...	Avalon Streaming Source	Double-click to	[clk]			
avalon_right_cha...	Avalon Streaming Source	Double-click to	[clk]			
avalon_left_chan...	Avalon Streaming Sink	Double-click to	[clk]			
avalon_right_cha...	Avalon Streaming Sink	Double-click to	[clk]			
external_interface	Conduit	audio_0_external_i...				
p1_unit	On-Chip Memory (RAM or ROM...					
clk1	Clock Input	Double-click to	clk_0			
s1	Avalon Memory Mapped Slave	Double-click to	[clk1]	0x0000_1000	0x0000_17ff	
reset1	Reset Input	Double-click to	[clk1]			
p1_die	On-Chip Memory (RAM or ROM...					
clk1	Clock Input	Double-click to	clk_0			
s1	Avalon Memory Mapped Slave	Double-click to	[clk1]	0x0000_2000	0x0000_21ff	
reset1	Reset Input	Double-click to	[clk1]			
p2_unit	On-Chip Memory (RAM or ROM...					
clk1	Clock Input	Double-click to	clk_0			
s1	Avalon Memory Mapped Slave	Double-click to	[clk1]	0x0000_3000	0x0000_37ff	
reset1	Reset Input	Double-click to	[clk1]			

Hierarchy

soc_system [soc_system.qsys]
- audio_0_external_interface
- audio_and_video_config_0_external_interface
- audio_pll_0_audio_clk
- clk
- hps
- hps_ddr3
- reset
- vga
- audio_0
- audio_and_video_config_0
- audio_pll_0
- bomb
- clk_0
- explode_sound
- firecenter
- firehori
- fireverti
- fix
- hps_0
- map_unit
- p1_die
- p1_unit
- p2_die
- p2_unit
- vga_ball_0
- Connections

Messages

Type	Path	Message
⚠	3 Warnings	
⚠	soc_system.audio_0.avalon_left_channel_source	audio_0.avalon_left_channel_source must be connected to an Avalon-ST sink
⚠	soc_system.audio_0.avalon_right_channel_source	audio_0.avalon_right_channel_source must be connected to an Avalon-ST sink
⚠	soc_system.audio_and_video_config_0	audio_and_video_config_0.avalon_av_config_slave must be connected to an Avalon-MM master
ⓘ	2 Info Messages	
ⓘ	soc_system.hps_0	HPS Main PLL counter settings: n = 0 m = 73
ⓘ	soc_system.hps_0	HPS peripherial PLL counter settings: n = 0 m = 39

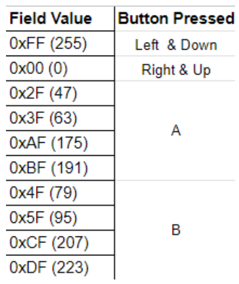
0 Errors, 3 Warnings

Figure 9: The Final Connection (the .qsys file)

4 Software

4.1 Controller

For this two-player game, the players will interface with the game using the game controller shown in Figure 10. The corresponding field value is 0xFF when the left and down buttons are pressed, and 0x00 when the right and up buttons are pressed. There are four different field values for when both A and B are pressed.

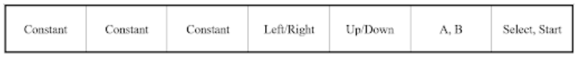


Field Value	Button Pressed
0xFF (255)	Left & Down
0x00 (0)	Right & Up
0x2F (47)	A
0x3F (63)	
0xAF (175)	
0xBF (191)	
0x4F (79)	B
0x5F (95)	
0xCF (207)	
0xDF (223)	

Figure 10: Values to control direction

The Game controller will be identified by the idProduct of 11 in hexadecimal. The game controller can be controlled using a single interface via USB. In our game, the players can be smoothly controlled using the arrow to indicate the direction of left/right/up/down. The button B is used for the game ready and plant wall after the game starts. The button A is used for the plant bombs.

We implemented our controller by changing the keyboard function in lab 2. We use the keyboard function to receive the 7-field package from the controller through the protocol, shown in Figure 11. The biggest problem in this function is that we need to set up a separate device for our controller so that they would not mess up with each other. After we have the controller-opening function, we will use the `libusb_interrupt_transfer` function to transfer their own 7-field code for each attribute in each field. The package will be received by our game and processed to determine the player's movement, bomb planting, or wall planting. The players can break the breakable wall and whoever kills the other first will win the game.



Constant	Constant	Constant	Left/Right	Up/Down	A, B	Select, Start

Figure 11: Controller protocol

4.2 Game Logic

The main game logic on the software side is presented in Figure 12. Our game logic will initialize the game by randomly generating the map, setting up the players, and setting up grids. The map is totally random generated without worrying about whether there will be a deadlock (it is explicitly explained in the section 4.2.4) The map is generated and written to the ram through the Avalon bus. The players need to press button B to start the game. Our game is based on the classic game called

Bomberman with some new game mechanics. We want this game to have more engaging gameplay or Higher playability.

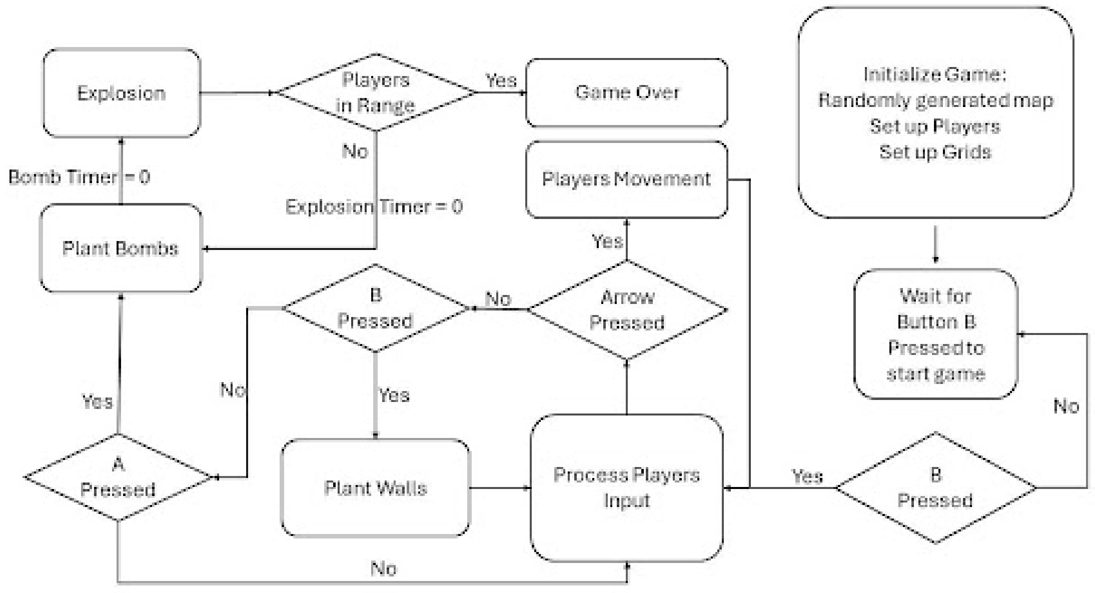


Figure 12: The block diagram of the game logic

4.2.1 Player Movement

Our two players have their own figure which can be recognized easily. We transfer the screen position of our player from the software to the hardware to ensure the smooth movement of our players on the screen. Also, we added the mechanism that the players will be in the tile position at the last, which makes it more fluent without the middle position. When we pressed the controller's left/right/up/down button, the players changed their position and moved at the speed we set up. We set up the direction mif for each player, which means that our players have different sprites for different directions. Specifically, the movement of players in the game is handled through direct input from game controllers. Also, it checks the current position of the player and updates their position based on the input received. In our game, we use a grid-based approach where each tile on the map has properties defining whether it is walkable. Thus, the players' movement is only possible if no obstacles (i.e., both hard and breakable walls, bombs, and the map boundary) are on the walking route.

4.2.2 Plant Bomb

The plant bomb mechanism is the traditional mechanism of the Bomberman. The planting of bombs is initiated by player input and is reflected immediately on the game map. When players press button A, one bomb will be planted, and the countdown will start. When the bomb timer reaches 0, it will explode in the four directions in the range of power with each direction having its own sprite. In this game, we set the bomb number to 1 and send the bomb number through the screen position to make sure the game can be played smoothly. When the explosion timer equals 0, the explosion will disappear. We have two kinds of walls in the game: one is a breakable wall, and the other is unbreakable; the bomb is able to destroy the breakable wall in the game.

4.2.3 Plant Wall

Planting wall is the new game mechanism, which is different from the traditional Bomberman game. We got this idea from the Tower Defense game. When players press button B after the game starts, they can plant infinite breakable walls. This mechanism increases the game's playability, and the players can apply new strategies to trap the other players in the wall to kill the other player.

4.2.4 Start of Game

First, there will be a start screen display before playing the game, shown in Figure 13.

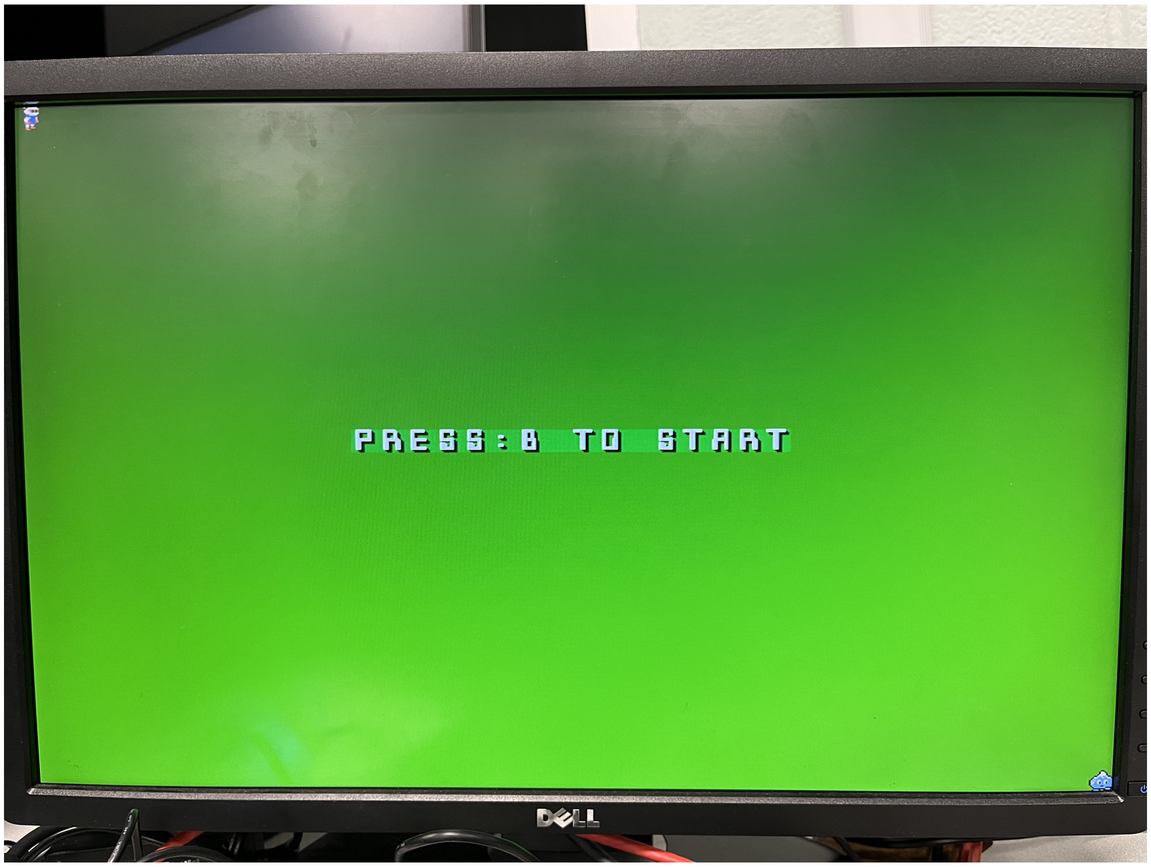


Figure 13: The start screen

Both players are required to press the button 'B' on the controller to start the game. The logic ensures that the game only starts once all players are ready.

The game map is randomly generated with four initial elements displayed: both players, hard walls, breakable walls, and the ground; Player 1 spawns in the top-left corner and Player 2 in the bottom-right corner, shown in Figure 14.

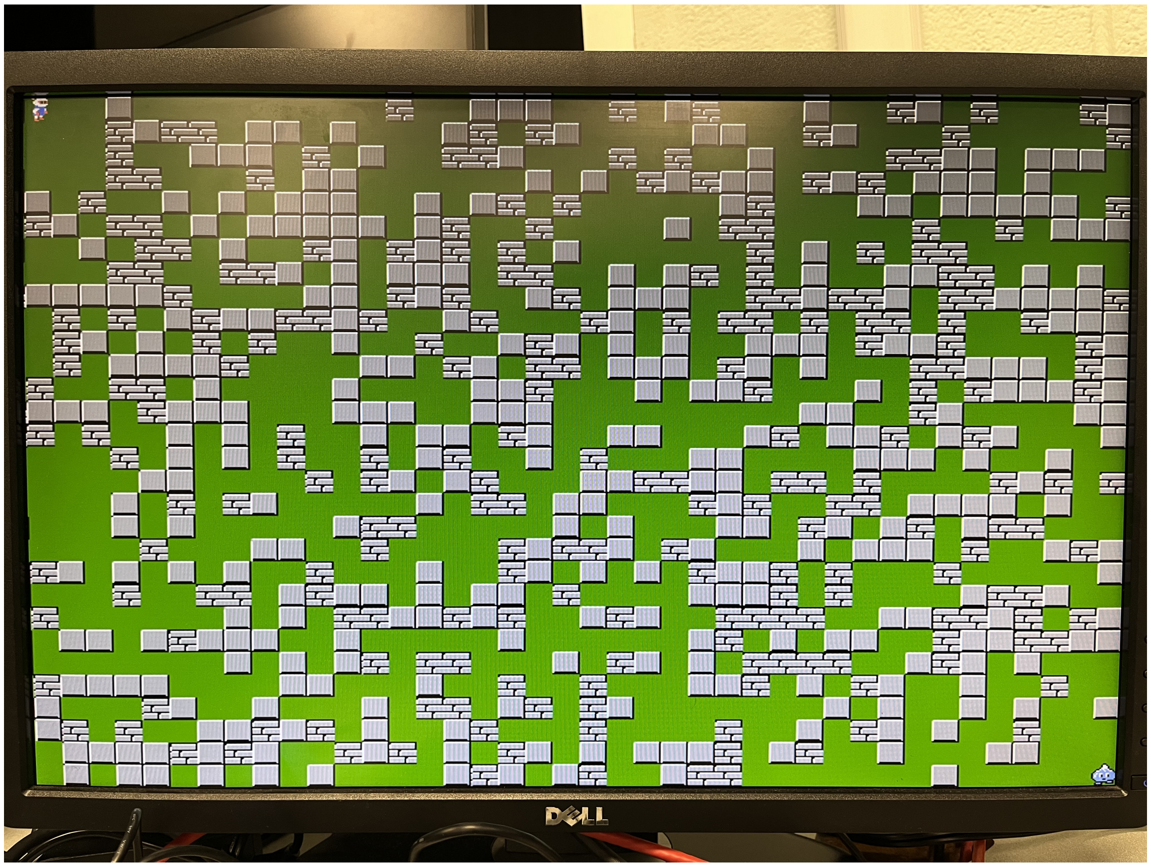


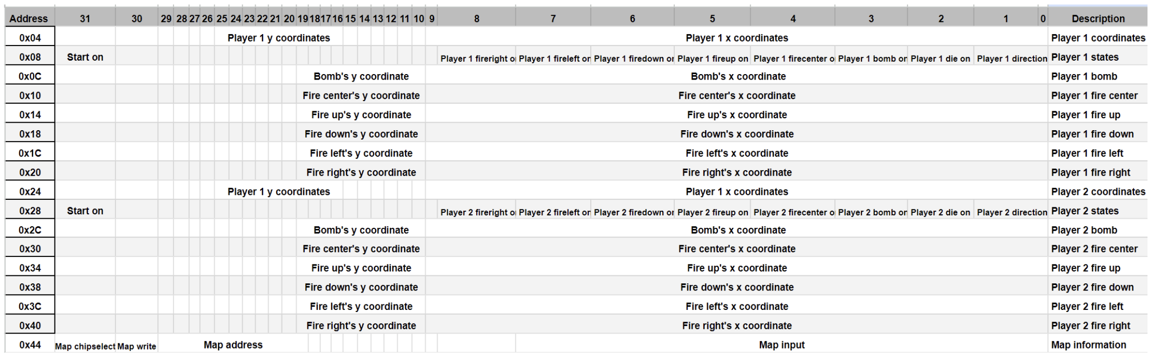
Figure 14: The the randomly generated map

To ensure at least one permanent walkable route in the randomly arranged elements, we predefined one such route and extended the generated elements based on this. Because the tile size of the map is 30×40, resulting in a total of 1200 elements, we have set the distribution to 50% ground, 30% hard walls, and 20% breakable walls to potentially increase the number of walkable routes on the map.

4.2.5 End of Game

When players touch the explosion area, it will die. We also set up the death sprites for the 2 players. When the number of alive players is lower than 2, the game is over and will show which player wins in the game.

4.3 Avalon Bus Interface



Address	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Description
0x04							Player 1 y coordinates																Player 1 x coordinates										Player 1 coordinates
0x08	Start on																							Player 1 fireright o	Player 1 fireleft on	Player 1 firedown on	Player 1 fireup on	Player 1 firecenter o	Player 1 bomb on	Player 1 die on	Player 1 direction		Player 1 states
0x0C													Bomb's y coordinate										Bomb's x coordinate										Player 1 bomb
0x10													Fire center's y coordinate										Fire center's x coordinate										Player 1 fire center
0x14													Fire up's y coordinate										Fire up's x coordinate										Player 1 fire up
0x18													Fire down's y coordinate										Fire down's x coordinate										Player 1 fire down
0x1C													Fire left's y coordinate										Fire left's x coordinate										Player 1 fire left
0x20													Fire right's y coordinate										Fire right's x coordinate										Player 1 fire right
0x24							Player 1 y coordinates																Player 1 x coordinates										Player 2 coordinates
0x28	Start on																							Player 2 fireright o	Player 2 fireleft on	Player 2 firedown on	Player 2 fireup on	Player 2 firecenter o	Player 2 bomb on	Player 2 die on	Player 2 direction		Player 2 states
0x2C													Bomb's y coordinate										Bomb's x coordinate										Player 2 bomb
0x30													Fire center's y coordinate										Fire center's x coordinate										Player 2 fire center
0x34													Fire up's y coordinate										Fire up's x coordinate										Player 2 fire up
0x38													Fire down's y coordinate										Fire down's x coordinate										Player 2 fire down
0x3C													Fire left's y coordinate										Fire left's x coordinate										Player 2 fire left
0x40													Fire right's y coordinate										Fire right's x coordinate										Player 2 fire right
0x44	Map chipselect	Map write	Map address																						Map input								Map information

Figure 15: The Avalon Bus Interface

As shown in Figure 15, we use 17 4-byte registers, each containing 32 bits, to write data from software to hardware. Specifically, we create registers for both players, including their coordinates, states, bombs, and the resulting explosion fires. We allocate the lower 10 bits for the x coordinates and the upper 10 bits for the y coordinates. The state registers contain enable signals for coordinates, bombs, and fires, each represented by 1 bit. The last register contains the map information, which enables us to randomly generate maps from the software side.

5 Conclusion and Improvement

5.1 Lesson Learned

Our first plan was simple in our design document since we are all new to the FPGA, SystemVerilog, and C. We wrote many time-permitted advanced features like audio, the randomly generated map, destroyable walls, and so on. With huge efforts, we managed to implement those new features in the end. We are all proud of ourselves and want to conclude what we learned here.

- We have learned how to implement the FPGA from this lesson. Since we three are all new to the FPGA, this lesson provided us with an opportunity to work with the FPGA. We have learned the design of hardware and software and the design of Avalon Bus Interfaces.

- How to display graphics is an important lesson for us. When you want to display something on the window, you can do it on the tile position and pixel position, both of them will allow you to show incredible play. However, you need to design the game logic carefully and remember how each other logic part will influence each other.

- We have learned how to make the audio work correctly per instance. For example, to make the explosion sound on time and on target, you need to have two ready singles to trigger it.

- Debug is an important part. Sometimes, it is hard for you to debug when you fail to display anything on the screen. The hardware could not give you any feedback. At this time, you need to print out the variable and ensure that you have successfully changed the data and the right data is transferred between hardware and software.

- RAM and ROM. We first use the ROM to display our map, which limits us to changing the map. After we learned how to use the RAM, we were able to generate the map randomly and display bombs and walls in tiles much more easily.

5.2 Advice

- Take care of the Design document. Setting your Avalon Bus Interface carefully can save you a lot of time.

- Don't make the design too complicated initially; you can add features later after finishing the basic function.

- Design the hardware architecture as simply as possible; it will take a lot of time to debug the SystemVerilog.

6 Contribution

- Hongkuan Yu configured the .qsys file, wrote the SystemVerilog of the hardware, and found all .mif files.

- Yuqi and Yelin implemented the game logic and the Avalon Bus Interface.

Appendix

A Hardware

A.1 vga_ball.sv

```
module vga_ball(input logic        clk,
                input logic            reset,
                input logic [31:0] writedata,
                input logic            write,
                input                    chipselect,
                input logic [4:0]  address,

                //For Audio
                input L_READY,
                input R_READY,
                output logic [15:0] L_DATA,
                output logic [15:0] R_DATA,
                output logic L_VALID,
                output logic R_VALID,

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

   logic [10:0]          hcount;
   logic [9:0]     vcount;

   logic [7:0]           background_r, background_g, background_b;

   logic [31:0]    p1_coordinate, p1_state, p1_bomb, p1_firecenter,
    ↪   p1_fireup, p1_firedown, p1_fireleft, p1_fireright,
    ↪   p2_coordinate, p2_state, p2_bomb, p2_firecenter, p2_fireup,
    ↪   p2_firedown, p2_fireleft, p2_fireright, map_info;

   vga_counters counters(.clk50(clk), .*);
```

```
// Audio
↪ ///////////////////////////////////////////////////////////////////////////

        logic [13:0] explode_address;
        logic [15:0] explode_data;
        soc_system_explode_sound(.address(explode_address),.clk(clk),.clken(1),
 .reset_req(0),.readdata(explode_data));

        logic [13:0] jingle_address;
        logic [15:0] jingle_data;
        soc_system_jingle_sound(.address(jingle_address),.clk(clk),.clken(1),
 .reset_req(0),.readdata(jingle_data));

        reg [11:0] counter;

        logic playing_explode1;
        logic playing_explode2;
        logic playing_jingle1;
        logic playing_jingle2;

        logic explode_ready1;
        logic explode_ready2;
        logic jingle_ready1;
        logic jingle_ready2;

        always_ff @(posedge clk) begin

                if (reset) begin

                        counter <= 0;
                        L_VALID <= 0;
                        R_VALID <= 0;

                        playing_explode1 <= 0;
                        playing_explode2 <= 0;
                        playing_jingle1 <= 0;
                        playing_jingle2 <= 0;
                        explode_address <= 0;
                        jingle_address <= 0;

                        explode_ready1 <= 1;
                        explode_ready2 <= 1;
                        jingle_ready1 <= 1;
                        jingle_ready2 <= 1;

                end
                else if (L_READY == 1 && R_READY == 1 && counter <
                 ↪  3125) begin

                        counter <= counter + 1;
                        L_VALID <= 0;
                        R_VALID <= 0;

                end
                else if (L_READY == 1 && R_READY == 1 && counter >=
                 ↪  3125) begin

                        counter <= 0;
                        L_VALID <= 1;
                        R_VALID <= 1;


                        //Flags
                        if (p1_firecenter_on == 1 && explode_ready1
                         ↪  == 1) begin

                                if (playing_explode2 == 0 &&
                                    ↪  playing_jingle1 == 0 &&
                                    ↪  playing_jingle2 == 0) begin


                                        playing_explode1 <= 1;
                                        playing_explode2 <= 0;
                                        playing_jingle1 <= 0;
                                        playing_jingle2 <= 0;
                                        explode_ready1 <= 0;

                                end

                        end
                        else if (p2_firecenter_on == 1 &&
                         ↪   explode_ready2 == 1) begin

                                if (playing_explode1 == 0 &&
                                    ↪   playing_jingle1 == 0 &&
                                    ↪   playing_jingle2 == 0)
                                    ↪   begin

                                        playing_explode1 <= 0;
                                        playing_explode2 <= 1;
                                        playing_jingle1 <= 0;
                                        playing_jingle2 <= 0;

                                        explode_ready2 <= 0;

                                end

                        end
                        else if (p1_bomb_on == 1 && jingle_ready1 ==
                         ↪   1) begin

                                if (playing_explode1 == 0 &&
                                    ↪   playing_explode2 == 0 &&
                                    ↪   playing_jingle2 == 0)
                                    ↪   begin

                                        playing_explode1 <= 0;
                                        playing_explode2 <= 0;
                                        playing_jingle1 <= 1;
                                        playing_jingle2 <= 0;

                                        jingle_ready1 <= 0;
                                end

                        end
                        else if (p2_bomb_on == 1 && jingle_ready2 ==
                         ↪  1) begin

                                if (playing_explode1 == 0 &&
                                    ↪  playing_explode2 == 0 &&
                                    ↪  playing_jingle1 == 0)
                                    ↪  begin

                                        playing_explode1 <= 0;
                                        playing_explode2 <= 0;
                                        playing_jingle1 <= 0;
                                        playing_jingle2 <= 1;

                                        jingle_ready2 <= 0;

                                end

                        end

                        //Play audio once per instant
                        if (p1_firecenter_on == 0) begin

                                explode_ready1 <= 1;

                        end

                        if (p2_firecenter_on == 0) begin

                                explode_ready2 <= 1;

                        end

                        if (p1_bomb_on == 0) begin
```

```
        jingle_ready1 <= 1;

end

if (p2_bomb_on == 0) begin

        jingle_ready2 <= 1;

end


// Playing Audio
if (playing_explode1 == 1) begin
        if (explode_address > 15000) begin
                explode_address <= 0;
                playing_explode1 <= 0;
        end
        else begin
                explode_address <=
                 ↪   explode_address + 1;
        end
        L_DATA <= explode_data;
        R_DATA <= explode_data;
end
else if (playing_explode2 == 1) begin
        if (explode_address > 15000) begin
                explode_address <= 0;
                playing_explode2 <= 0;
        end
        else begin
                explode_address <=
                 ↪   explode_address + 1;
        end
        L_DATA <= explode_data;
        R_DATA <= explode_data;
end
else if (playing_jingle1 == 1) begin
        if (jingle_address > 15000) begin
```

```
                                        jingle_address <= 0;
                                        playing_jingle1 <= 0;
                                end
                                else begin
                                        jingle_address <=
                                         ↪  jingle_address + 1;
                                end
                                L_DATA <= jingle_data;
                                R_DATA <= jingle_data;
                        end
                        else if (playing_jingle2 == 1) begin
                                if (jingle_address > 15000) begin
                                        jingle_address <= 0;
                                        playing_jingle2 <= 0;
                                end
                                else begin
                                        jingle_address <=
                                         ↪  jingle_address + 1;
                                end
                                L_DATA <= jingle_data;
                                R_DATA <= jingle_data;
                        end


                end
                else begin

                        L_VALID <= 0;
                        R_VALID <= 0;

                end
        end

// Display
 ↪  ////////////////////////////////////////////////////////////////////////
   always_ff @(posedge clk) begin
     if (reset) begin
      // set the background to green
```

```
    background_r <= 8'h30;
    background_g <= 8'h80;
    background_b <= 8'h00;

    //initialization
    p1_coordinate <= 32'd0;
    p1_state <= 32'd0;
    p1_bomb <= 32'd0;
    p1_firecenter <= 32'd0;
    p1_fireup <= 32'd0;
    p1_firedown <= 32'd0;
    p1_fireleft <= 32'd0;
    p1_fireright <= 32'd0;

    p2_coordinate <= 32'd0;
    p2_state <= 32'd0;
    p2_bomb <= 32'd0;
    p2_firecenter <= 32'd0;
    p2_fireup <= 32'd0;
    p2_firedown <= 32'd0;
    p2_fireleft <= 32'd0;
    p2_fireright <= 32'd0;
   end
   else if (chipselect && write) begin
     case (address)
      5'b00001 : p1_coordinate <= writedata;
      5'b00010 : p1_state <= writedata;
      5'b00011 : p1_bomb <= writedata;
      5'b00100 : p1_firecenter <= writedata;
      5'b00101 : p1_fireup <= writedata;
      5'b00110 : p1_firedown <= writedata;
      5'b00111 : p1_fireleft <= writedata;
      5'b01000 : p1_fireright <= writedata;

      5'b01001 : p2_coordinate <= writedata;
      5'b01010 : p2_state <= writedata;
      5'b01011 : p2_bomb <= writedata;
      5'b01100 : p2_firecenter <= writedata;
    5'b01101 : p2_fireup <= writedata;
    5'b01110 : p2_firedown <= writedata;
    5'b01111 : p2_fireleft <= writedata;
    5'b10000 : p2_fireright <= writedata;

    5'b10001 : map_info <= writedata;
   endcase
  end
end

 //p1
 logic [9:0] p1_address;
 logic [15:0] p1_output;
 soc_system_p1_unit
 ↪  p1_unit(.address(p1_address),.clk(clk),.clken(1),.reset_req(0),
 .readdata(p1_output));
 logic p1_en;

 logic [9:0] p1_x;
 logic [9:0] p1_y;
 logic [1:0] p1_dir;

 logic [7:0] p1_die_address;
 logic [15:0] p1_die_output;
 soc_system_p1_die
 ↪  die1(.address(p1_die_address),.clk(clk),.clken(1),.reset_req(0),
 .readdata(p1_die_output));
 logic p1_die_en;

 logic p1_die_on;


 //p2
 logic [9:0] p2_address;
 logic [15:0] p2_output;
 soc_system_p2_unit
 ↪  p2_unit(.address(p2_address),.clk(clk),.clken(1),.reset_req(0),
 .readdata(p2_output));
```

```
logic p2_en;

logic [9:0] p2_x;
logic [9:0] p2_y;
logic [1:0] p2_dir;

logic [7:0] p2_die_address;
logic [15:0] p2_die_output;
soc_system_p2_die
 ↪  die2(.address(p2_die_address),.clk(clk),.clken(1),.reset_req(0),
.readdata(p2_die_output));
logic p2_die_en;

logic p2_die_on;

//fixed wall
logic [7:0] fix_address;
logic [15:0] fix_output;
soc_system_fix
 ↪  fix(.address(fix_address),.clk(clk),.clken(1),.reset_req(0),
.readdata(fix_output));
logic fix_en;

//Destroyable wall
logic [7:0] wall_address;
logic [15:0] wall_output;
soc_system_wall
 ↪  wall(.address(wall_address),.clk(clk),.clken(1),.reset_req(0),
.readdata(wall_output));
logic wall_en;


//p1_bomb
logic [7:0] p1_bomb_address;
logic [15:0] p1_bomb_output;
soc_system_bomb
 ↪  bomb1(.address(p1_bomb_address),.clk(clk),.clken(1),.reset_req(0),
.readdata(p1_bomb_output));
```

```
logic p1_bomb_en;

logic p1_bomb_on;

logic [9:0] p1bomb_x;
logic [9:0] p1bomb_y;

//p2_bomb
logic [7:0] p2_bomb_address;
logic [15:0] p2_bomb_output;
soc_system_bomb
↪  bomb2(.address(p2_bomb_address),.clk(clk),.clken(1),.reset_req(0),
.readdata(p2_bomb_output));
logic p2_bomb_en;

logic p2_bomb_on;

logic [9:0] p2bomb_x;
logic [9:0] p2bomb_y;

//p1_fire
logic [7:0] p1_firecenter_address;write
logic [15:0] p1_firecenter_output;
soc_system_firecenter
↪  firecenter1(.address(p1_firecenter_address),.clk(clk),.clken(1),
.reset_req(0),.readdata(p1_firecenter_output));
logic p1_firecenter_en;

logic p1_firecenter_on;

logic [7:0] p1_fireup_address;
logic [15:0] p1_fireup_output;
soc_system_fireverti
↪  fireup1(.address(p1_fireup_address),.clk(clk),.clken(1),.reset_req(0),
.readdata(p1_fireup_output));
logic p1_fireup_en;

logic p1_fireup_on;
logic [7:0] p1_firedown_address;
logic [15:0] p1_firedown_output;
soc_system_fireverti
 ↪  firedown1(.address(p1_firedown_address),.clk(clk),.clken(1),.reset_req(0),
.readdata(p1_firedown_output));
logic p1_firedown_en;

logic p1_firedown_on;

logic [7:0] p1_fireleft_address;
logic [15:0] p1_fireleft_output;
soc_system_firehori
 ↪  fireleft1(.address(p1_fireleft_address),.clk(clk),.clken(1),.reset_req(0),
.readdata(p1_fireleft_output));
logic p1_fireleft_en;

logic p1_fireleft_on;

logic [7:0] p1_fireright_address;
logic [15:0] p1_fireright_output;
soc_system_firehori
 ↪  fireright1(.address(p1_fireright_address),.clk(clk),.clken(1),.reset_req(0),
.readdata(p1_fireright_output));
logic p1_fireright_en;

logic p1_fireright_on;

logic [9:0] p1firecenter_x;
logic [9:0] p1firecenter_y;
logic [9:0] p1fireup_x;
logic [9:0] p1fireup_y;
logic [9:0] p1firedown_x;
logic [9:0] p1firedown_y;write
logic [9:0] p1fireleft_x;
logic [9:0] p1fireleft_y;
logic [9:0] p1fireright_x;
logic [9:0] p1fireright_y;
```

```
//p2_fire
logic [7:0] p2_firecenter_address;
logic [15:0] p2_firecenter_output;
soc_system_firecenter
↪  firecenter2(.address(p2_firecenter_address),.clk(clk),.clken(1),
.reset_req(0),.readdata(p2_firecenter_output));
logic p2_firecenter_en;

logic p2_firecenter_on;

logic [7:0] p2_fireup_address;
logic [15:0] p2_fireup_output;
soc_system_fireverti
↪  fireup2(.address(p2_fireup_address),.clk(clk),.clken(1),.reset_req(0),
.readdata(p2_fireup_output));
logic p2_fireup_en;

logic p2_fireup_on;

logic [7:0] p2_firedown_address;
logic [15:0] p2_firedown_output;
soc_system_fireverti
↪  firedown2(.address(p2_firedown_address),.clk(clk),.clken(1),.reset_req(0),
.readdata(p2_firedown_output));
logic p2_firedown_en;

logic p2_firedown_on;

logic [7:0] p2_fireleft_address;
logic [15:0] p2_fireleft_output;
soc_system_firehori
↪  fireleft2(.address(p2_fireleft_address),.clk(clk),.clken(1),.reset_req(0),
.readdata(p2_fireleft_output));
logic p2_fireleft_en;

logic p2_fireleft_on;
logic [7:0] p2_fireright_address;
logic [15:0] p2_fireright_output;
soc_system_firehori
 ↪  fireright2(.address(p2_fireright_address),.clk(clk),.clken(1),.reset_req(0),
.readdata(p2_fireright_output));
logic p2_fireright_en;

logic p2_fireright_on;

logic [9:0] p2firecenter_x;
logic [9:0] p2firecenter_y;
logic [9:0] p2fireup_x;
logic [9:0] p2fireup_y;
logic [9:0] p2firedown_x;
logic [9:0] p2firedown_y;
logic [9:0] p2fireleft_x;
logic [9:0] p2fireleft_y;
logic [9:0] p2fireright_x;
logic [9:0] p2fireright_y;

//map
logic [10:0] map_address;
logic map_chipselect;
logic map_write;
logic [7:0] map_input;
logic [7:0] map_output;
soc_system_map_unit
 ↪  map_unit(.address(map_address),.clk(clk),.clken(1),.reset_req(0),

 ↪  .chipselect(map_chipselect),.write(map_write),.writedata(map_input),
.readdata(map_output));
logic map_en;

//p1_win
logic [10:0] p1_win_address;
logic [15:0] p1_win_output;
soc_system_p1_win
 ↪  p1_win(.address(p1_win_address),.clk(clk),.clken(1),.reset_req(0),
.readdata(p1_win_output));
logic p1_win_en;

//p2_win
logic [10:0] p2_win_address;
logic [15:0] p2_win_output;
soc_system_p2_win
 ↪  p2_win(.address(p2_win_address),.clk(clk),.clken(1),.reset_req(0),
.readdata(p2_win_output));
logic p2_win_en;

//start
logic [11:0] start_address;
logic [15:0] start_output;
soc_system_start
 ↪  start(.address(start_address),.clk(clk),.clken(1),.reset_req(0),
.readdata(start_output));
logic start_en;

logic start_on;

//tile
logic [5:0] tile16_x;
logic [5:0] tile16_y;


always_comb begin
  //about p1
  p1_x = p1_coordinate[9:0];
  p1_y = p1_coordinate[25:16];

  //state of p1
  p1_dir = p1_state[1:0];
  p1_die_on = ~p1_state[2];
  p1_bomb_on = p1_state[3];
  p1_firecenter_on = p1_state[4];
  p1_fireup_on = p1_state[5];
  p1_firedown_on = p1_state[6];
```

```
p1_fireleft_on = p1_state[7];
p1_fireright_on = p1_state[8];


//about p2
p2_x = p2_coordinate[9:0];
p2_y = p2_coordinate[25:16];


//state of p2
p2_dir = p2_state[1:0];
p2_die_on = ~p2_state[2];
p2_bomb_on = p2_state[3];
p2_firecenter_on = p2_state[4];
p2_fireup_on = p2_state[5];
p2_firedown_on = p2_state[6];
p2_fireleft_on = p2_state[7];
p2_fireright_on = p2_state[8];


//p1_bomb
p1bomb_x = p1_bomb[9:0];
p1bomb_y = p1_bomb[19:10];

//p1_fire
p1firecenter_x = p1_firecenter[9:0];
p1firecenter_y = p1_firecenter[19:10];
p1fireup_x = p1_fireup[9:0];
p1fireup_y = p1_fireup[19:10];
p1firedown_x = p1_firedown[9:0];
p1firedown_y = p1_firedown[19:10];
p1fireleft_x = p1_fireleft[9:0];
p1fireleft_y = p1_fireleft[19:10];
p1fireright_x = p1_fireright[9:0];
p1fireright_y = p1_fireright[19:10];

//p2_bomb
p2bomb_x = p2_bomb[9:0];
```

```
  p2bomb_y = p2_bomb[19:10];

  //p2_fire
  p2firecenter_x = p2_firecenter[9:0];
  p2firecenter_y = p2_firecenter[19:10];
  p2fireup_x = p2_fireup[9:0];
  p2fireup_y = p2_fireup[19:10];
  p2firedown_x = p2_firedown[9:0];
  p2firedown_y = p2_firedown[19:10];
  p2fireleft_x = p2_fireleft[9:0];
  p2fireleft_y = p2_fireleft[19:10];
  p2fireright_x = p2_fireright[9:0];
  p2fireright_y = p2_fireright[19:10];

  //tile
  tile16_x = hcount[10:1] >> 4; //divide by 16 to get the x
   ↪  coordinate. 640/16 = 40
  tile16_y = vcount[9:0] >> 4; //divide by 16 to get the y
   ↪  coordinate. 480/16 = 30

  //map
  map_chipselect = map_info[31];
  map_write = map_info[30];
  if (map_chipselect == 1 & map_write == 1) begin
    map_en = 0;
    map_address = map_info[29:19];
    map_input = map_info[7:0];
  end
  else begin
          map_en = 1;
    map_address = tile16_x + tile16_y * 40;
    map_input = 0;
  end

  //start
  start_on = p1_state[31] & p2_state[31];

// dir:
```

```
// 2'b00 --> down
// 2'b01 --> up
// 2'b10 --> left
// 2'b11 --> right
end

//show p1
always_ff @(posedge clk) begin
  if (hcount[10:1] >= p1_x && hcount[10:1] <= (p1_x + 10'd15) &&
   ↪  vcount[9:0] >= p1_y && vcount[9:0] <= (p1_y + 10'd15) &&
   ↪  p1_die_on == 0) begin

    p1_en <= 1;

    case(p1_dir)
     2'b00 : p1_address <= hcount[10:1] - p1_x + (vcount[9:0] -
      ↪  p1_y) * 16;
     2'b01 : p1_address <= hcount[10:1] - p1_x + (vcount[9:0] -
      ↪  p1_y) * 16 + 256;
     2'b10 : p1_address <= hcount[10:1] - p1_x + (vcount[9:0] -
      ↪  p1_y) * 16 + 512;
     2'b11 : p1_address <= hcount[10:1] - p1_x + (vcount[9:0] -
      ↪  p1_y) * 16 + 768;
    endcase

  end

  else begin

    p1_en <= 0;

  end

end

//p1 die
always_ff @(posedge clk) begin
```

```
  if (hcount[10:1] >= p1_x && hcount[10:1] <= (p1_x + 10'd15) &&
   ↪  vcount[9:0] >= p1_y && vcount[9:0] <= (p1_y + 10'd15) &&
   ↪  p1_die_on == 1) begin

    p1_die_en <= 1;

    p1_die_address <= hcount[10:1] - p1_x + (vcount[9:0] - p1_y)
     ↪  * 16;

  end

  else begin

    p1_die_en <= 0;

  end

end


//show p2
always_ff @(posedge clk) begin

  if (hcount[10:1] >= p2_x && hcount[10:1] <= (p2_x + 10'd15) &&
   ↪  vcount[9:0] >= p2_y && vcount[9:0] <= (p2_y + 10'd15) &&
   ↪  p2_die_on == 0) begin

    p2_en <= 1;

    case(p2_dir)
     2'b00 : p2_address <= hcount[10:1] - p2_x + (vcount[9:0] -
      ↪  p2_y) * 16;
     2'b01 : p2_address <= hcount[10:1] - p2_x + (vcount[9:0] -
      ↪  p2_y) * 16 + 256;
     2'b10 : p2_address <= hcount[10:1] - p2_x + (vcount[9:0] -
      ↪  p2_y) * 16 + 512;
     2'b11 : p2_address <= hcount[10:1] - p2_x + (vcount[9:0] -
      ↪  p2_y) * 16 + 768;
    endcase

  end

  else begin

    p2_en <= 0;

  end

end

//p2 die
always_ff @(posedge clk) begin

  if (hcount[10:1] >= p2_x && hcount[10:1] <= (p2_x + 10'd15) &&
  ↪  vcount[9:0] >= p2_y && vcount[9:0] <= (p2_y + 10'd15) &&
  ↪  p2_die_on == 1) begin

    p2_die_en <= 1;

    p2_die_address <= hcount[10:1] - p2_x + (vcount[9:0] - p2_y)
    ↪  * 16;

  end

  else begin

    p2_die_en <= 0;

  end

end

//show p1 bomb
always_ff @(posedge clk) begin
```

```
  if (hcount[10:1] >= p1bomb_x && hcount[10:1] <= (p1bomb_x +
  ↪  10'd15) && vcount[9:0] >= p1bomb_y && vcount[9:0] <=
  ↪  (p1bomb_y + 10'd15) && p1_bomb_on == 1) begin

    p1_bomb_en <= 1;

    p1_bomb_address <= hcount[10:1] - p1bomb_x + (vcount[9:0] -
    ↪  p1bomb_y) * 16;

  end

  else begin

    p1_bomb_en <= 0;

  end

end

//show p2 bomb
always_ff @(posedge clk) begin

  if (hcount[10:1] >= p2bomb_x && hcount[10:1] <= (p2bomb_x +
  ↪  10'd15) && vcount[9:0] >= p2bomb_y && vcount[9:0] <=
  ↪  (p2bomb_y + 10'd15) && p2_bomb_on == 1) begin

    p2_bomb_en <= 1;

    p2_bomb_address <= hcount[10:1] - p2bomb_x + (vcount[9:0] -
    ↪  p2bomb_y) * 16;

  end

  else begin

    p2_bomb_en <= 0;

  end
```

```
end

//show p1 bomb fire after explosion
//p1 fire center
always_ff @(posedge clk) begin

  if (hcount[10:1] >= p1firecenter_x && hcount[10:1] <=
   ↪  (p1firecenter_x + 10'd15) && vcount[9:0] >= p1firecenter_y
   ↪  && vcount[9:0] <= (p1firecenter_y + 10'd15) &&
   ↪  p1_firecenter_on == 1) begin

    p1_firecenter_en <= 1;

    p1_firecenter_address <= hcount[10:1] - p1firecenter_x +
     ↪  (vcount[9:0] - p1firecenter_y) * 16;

  end

  else begin

    p1_firecenter_en <= 0;

  end

end


//p1 fire up
always_ff @(posedge clk) begin

  if (hcount[10:1] >= p1fireup_x && hcount[10:1] <= (p1fireup_x +
   ↪  10'd15) && vcount[9:0] >= p1fireup_y && vcount[9:0] <=
   ↪  (p1fireup_y + 10'd15) && p1_fireup_on == 1) begin

    p1_fireup_en <= 1;
    p1_fireup_address <= hcount[10:1] - p1fireup_x + (vcount[9:0]
     ↪  - p1fireup_y) * 16;

  end

  else begin

    p1_fireup_en <= 0;

  end

end

//p1 fire down
always_ff @(posedge clk) begin

  if (hcount[10:1] >= p1firedown_x && hcount[10:1] <=
   ↪  (p1firedown_x + 10'd15) && vcount[9:0] >= p1firedown_y &&
   ↪  vcount[9:0] <= (p1firedown_y + 10'd15) && p1_firedown_on ==
   ↪  1) begin

    p1_firedown_en <= 1;

    p1_firedown_address <= hcount[10:1] - p1firedown_x +
     ↪  (vcount[9:0] - p1firedown_y) * 16;

  end

  else begin

    p1_firedown_en <= 0;

  end

end

//p1 fire left
always_ff @(posedge clk) begin
```

```
  if (hcount[10:1] >= p1fireleft_x && hcount[10:1] <=
   ↪  (p1fireleft_x + 10'd15) && vcount[9:0] >= p1fireleft_y &&
   ↪  vcount[9:0] <= (p1fireleft_y + 10'd15) && p1_fireleft_on ==
   ↪  1) begin

    p1_fireleft_en <= 1;

    p1_fireleft_address <= hcount[10:1] - p1fireleft_x +
     ↪  (vcount[9:0] - p1fireleft_y) * 16;

  end

  else begin

    p1_fireleft_en <= 0;

  end

end

//p1 fire right
always_ff @(posedge clk) begin

  if (hcount[10:1] >= p1fireright_x && hcount[10:1] <=
   ↪  (p1fireright_x + 10'd15) && vcount[9:0] >= p1fireright_y &&
   ↪  vcount[9:0] <= (p1fireright_y + 10'd15) && p1_fireright_on
   ↪  == 1) begin

    p1_fireright_en <= 1;

    p1_fireright_address <= hcount[10:1] - p1fireright_x +
     ↪  (vcount[9:0] - p1fireright_y) * 16;

  end

  else begin
```

```
    p1_fireright_en <= 0;

  end

end

//show p2 bomb fire after explosion
//p2 fire center
always_ff @(posedge clk) begin

  if (hcount[10:1] >= p2firecenter_x && hcount[10:1] <=
   ↪  (p2firecenter_x + 10'd15) && vcount[9:0] >= p2firecenter_y
   ↪  && vcount[9:0] <= (p2firecenter_y + 10'd15) &&
   ↪  p2_firecenter_on == 1) begin

    p2_firecenter_en <= 1;

    p2_firecenter_address <= hcount[10:1] - p2firecenter_x +
     ↪  (vcount[9:0] - p2firecenter_y) * 16;

  end

  else begin

    p2_firecenter_en <= 0;

  end

end


//p2 fire up
always_ff @(posedge clk) begin

  if (hcount[10:1] >= p2fireup_x && hcount[10:1] <= (p2fireup_x +
   ↪  10'd15) && vcount[9:0] >= p2fireup_y && vcount[9:0] <=
   ↪  (p2fireup_y + 10'd15) && p2_fireup_on == 1) begin
```

```
    p2_fireup_en <= 1;

    p2_fireup_address <= hcount[10:1] - p2fireup_x + (vcount[9:0]
     ↪  - p2fireup_y) * 16;

  end

  else begin

    p2_fireup_en <= 0;

  end

end

//p2 fire down
always_ff @(posedge clk) begin

  if (hcount[10:1] >= p2firedown_x && hcount[10:1] <=
   ↪  (p2firedown_x + 10'd15) && vcount[9:0] >= p2firedown_y &&
   ↪  vcount[9:0] <= (p2firedown_y + 10'd15) && p2_firedown_on ==
   ↪  1) begin

    p2_firedown_en <= 1;

    p2_firedown_address <= hcount[10:1] - p2firedown_x +
     ↪  (vcount[9:0] - p2firedown_y) * 16;

  end

  else begin

    p2_firedown_en <= 0;

  end
end

//p2 fire left
```

```
always_ff @(posedge clk) begin

  if (hcount[10:1] >= p2fireleft_x && hcount[10:1] <=
  ↪  (p2fireleft_x + 10'd15) && vcount[9:0] >= p2fireleft_y &&
  ↪  vcount[9:0] <= (p2fireleft_y + 10'd15) && p2_fireleft_on ==
  ↪  1) begin

    p2_fireleft_en <= 1;

    p2_fireleft_address <= hcount[10:1] - p2fireleft_x +
    ↪  (vcount[9:0] - p2fireleft_y) * 16;

  end

  else begin

    p2_fireleft_en <= 0;

  end

end

//p2 fire right
always_ff @(posedge clk) begin

  if (hcount[10:1] >= p2fireright_x && hcount[10:1] <=
  ↪  (p2fireright_x + 10'd15) && vcount[9:0] >= p2fireright_y &&
  ↪  vcount[9:0] <= (p2fireright_y + 10'd15) && p2_fireright_on
  ↪  == 1) begin

    p2_fireright_en <= 1;

    p2_fireright_address <= hcount[10:1] - p2fireright_x +
    ↪  (vcount[9:0] - p2fireright_y) * 16;

  end

  else begin
```

```
    p2_fireright_en <= 0;

  end

end

//map
always_ff @(posedge clk) begin

  if (map_en == 1 && map_output == 1) begin

    fix_en <= 1;
    fix_address <= hcount[4:1] + vcount[3:0] * 16;

  end

  else begin

    fix_en <= 0;

  end

end

always_ff @(posedge clk) begin

  if (map_en == 1 && map_output == 2) begin

    wall_en <= 1;
    wall_address <= hcount[4:1] + vcount[3:0] * 16;

  end

  else begin

    wall_en <= 0;
  end

end

//start
always_ff @(posedge clk) begin

  if (tile16_x >= 6'd12 && tile16_x <= 6'd27 && tile16_y == 6'd14
   ↪  && start_on == 0) begin

    start_en <= 1;
    start_address <= hcount[4:1] + vcount[3:0] * 16 + (tile16_x -
     ↪  6'd12) * 256;

  end

  else begin

    start_en <= 0;

  end

end

//p1_win
always_ff @(posedge clk) begin

  if (tile16_x >= 6'd17 && tile16_x <= 6'd22 && tile16_y == 6'd14
   ↪  && p2_die_on == 1) begin

    p1_win_en <= 1;
    p1_win_address <= hcount[4:1] + vcount[3:0] * 16 + (tile16_x
     ↪  - 6'd17) * 256;

  end

  else begin
```

```
      p1_win_en <= 0;

    end

  end

 //p2_win
  always_ff @(posedge clk) begin

    if (tile16_x >= 6'd17 && tile16_x <= 6'd22 && tile16_y == 6'd14
     ↪  && p1_die_on == 1) begin

      p2_win_en <= 1;
      p2_win_address <= hcount[4:1] + vcount[3:0] * 16 + (tile16_x
       ↪  - 6'd17) * 256;

    end

    else begin

      p2_win_en <= 0;

    end

  end

 //Color Mapping
 always_comb begin
    {VGA_R, VGA_G, VGA_B} = {background_r, background_g,
     ↪  background_b};
    if (VGA_BLANK_n) begin
      if (p1_die_en) begin
        {VGA_R, VGA_G, VGA_B} = {p1_die_output[15:12], 4'b0000,
         ↪  p1_die_output[11:8], 4'b0000, p1_die_output[7:4],
         ↪  4'b0000};
      end
      else if (p2_win_en) begin
        {VGA_R, VGA_G, VGA_B} = {p2_win_output[15:12], 4'b0000,
         ↪  p2_win_output[11:8], 4'b0000, p2_win_output[7:4],
         ↪  4'b0000};
      end
      else if (p2_die_en) begin
        {VGA_R, VGA_G, VGA_B} = {p2_die_output[15:12], 4'b0000,
         ↪  p2_die_output[11:8], 4'b0000, p2_die_output[7:4],
         ↪  4'b0000};
      end
      else if (p1_win_en) begin
        {VGA_R, VGA_G, VGA_B} = {p1_win_output[15:12], 4'b0000,
         ↪  p1_win_output[11:8], 4'b0000, p1_win_output[7:4],
         ↪  4'b0000};
      end
      else if (start_en) begin
        {VGA_R, VGA_G, VGA_B} = {start_output[15:12], 4'b0000,
         ↪  start_output[11:8], 4'b0000, start_output[7:4],
         ↪  4'b0000};
      end
      else if (fix_en) begin
        {VGA_R, VGA_G, VGA_B} = {fix_output[15:12], 4'b0000,
         ↪  fix_output[11:8], 4'b0000, fix_output[7:4], 4'b0000};
      end
      else if (p1_en) begin
        {VGA_R, VGA_G, VGA_B} = {p1_output[15:12], 4'b0000,
         ↪  p1_output[11:8], 4'b0000, p1_output[7:4], 4'b0000};
      end
      else if (p2_en) begin
        {VGA_R, VGA_G, VGA_B} = {p2_output[15:12], 4'b0000,
         ↪  p2_output[11:8], 4'b0000, p2_output[7:4], 4'b0000};
      end
      else if (p1_firecenter_en) begin
        {VGA_R, VGA_G, VGA_B} = {p1_firecenter_output[15:12],
         ↪  4'b0000, p1_firecenter_output[11:8], 4'b0000,
         ↪  p1_firecenter_output[7:4], 4'b0000};
      end
      else if (p1_fireup_en) begin
```

```
  {VGA_R, VGA_G, VGA_B} = {p1_fireup_output[15:12], 4'b0000,
   ↪  p1_fireup_output[11:8], 4'b0000, p1_fireup_output[7:4],
   ↪  4'b0000};
end
else if (p1_firedown_en) begin
       {VGA_R, VGA_G, VGA_B} = {p1_firedown_output[15:12],
        ↪  4'b0000, p1_firedown_output[11:8], 4'b0000,
        ↪  p1_firedown_output[7:4], 4'b0000};
end
else if (p1_fireleft_en) begin
  {VGA_R, VGA_G, VGA_B} = {p1_fireleft_output[15:12],
   ↪  4'b0000, p1_fireleft_output[11:8], 4'b0000,
   ↪  p1_fireleft_output[7:4], 4'b0000};
end
else if (p1_fireright_en) begin
  {VGA_R, VGA_G, VGA_B} = {p1_fireright_output[15:12],
   ↪  4'b0000, p1_fireright_output[11:8], 4'b0000,
   ↪  p1_fireright_output[7:4], 4'b0000};
end
else if (p2_firecenter_en) begin
  {VGA_R, VGA_G, VGA_B} = {p2_firecenter_output[15:12],
   ↪  4'b0000, p2_firecenter_output[11:8], 4'b0000,
   ↪  p2_firecenter_output[7:4], 4'b0000};
end
else if (p2_fireup_en) begin
  {VGA_R, VGA_G, VGA_B} = {p2_fireup_output[15:12], 4'b0000,
   ↪  p2_fireup_output[11:8], 4'b0000, p2_fireup_output[7:4],
   ↪  4'b0000};
end
else if (p2_firedown_en) begin
       {VGA_R, VGA_G, VGA_B} = {p2_firedown_output[15:12],
        ↪  4'b0000, p2_firedown_output[11:8], 4'b0000,
        ↪  p2_firedown_output[7:4], 4'b0000};
end
else if (p2_fireleft_en) begin
  {VGA_R, VGA_G, VGA_B} = {p2_fireleft_output[15:12],
   ↪  4'b0000, p2_fireleft_output[11:8], 4'b0000,
   ↪  p2_fireleft_output[7:4], 4'b0000};
```

```
        end
        else if (p2_fireright_en) begin
          {VGA_R, VGA_G, VGA_B} = {p2_fireright_output[15:12],
           ↪  4'b0000, p2_fireright_output[11:8], 4'b0000,
           ↪  p2_fireright_output[7:4], 4'b0000};
        end
        else if (wall_en) begin
          {VGA_R, VGA_G, VGA_B} = {wall_output[15:12], 4'b0000,
           ↪  wall_output[11:8], 4'b0000, wall_output[7:4], 4'b0000};
        end
        else if (p1_bomb_en) begin
          {VGA_R, VGA_G, VGA_B} = {p1_bomb_output[15:12], 4'b0000,
           ↪  p1_bomb_output[11:8], 4'b0000, p1_bomb_output[7:4],
           ↪  4'b0000};
        end
        else if (p2_bomb_en) begin
            {VGA_R, VGA_G, VGA_B} = {p2_bomb_output[15:12], 4'b0000,
             ↪  p2_bomb_output[11:8], 4'b0000, p2_bomb_output[7:4],
             ↪  4'b0000};
        end

      end
   end

endmodule

module vga_counters(
 input logic              clk50, reset,
 output logic [10:0] hcount,  // hcount[10:1] is pixel column
 output logic [9:0]  vcount,  // vcount[9:0] is pixel row
 output logic              VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n,
  ↪  VGA_SYNC_n);

/*
 * 640 X 480 VGA timing for a 50 MHz clock: one pixel every other
  ↪  cycle
 *
 * HCOUNT 1599 0             1279       1599 0
 *              _______________              ________
 * ___________|    Video      |____________|  Video
 *
 *
 * |SYNC| BP |<-- HACTIVE -->|FP|SYNC| BP |<-- HACTIVE
 *       _______________________      _____________
 * |____|       VGA_HS          |____|
 */
   // Parameters for hcount
   parameter HACTIVE      = 11'd 1280,
             HFRONT_PORCH = 11'd 32,
             HSYNC        = 11'd 192,
             HBACK_PORCH  = 11'd 96,
             HTOTAL       = HACTIVE + HFRONT_PORCH + HSYNC +
                            HBACK_PORCH; // 1600

   // Parameters for vcount
   parameter VACTIVE      = 10'd 480,
             VFRONT_PORCH = 10'd 10,
             VSYNC        = 10'd 2,
             VBACK_PORCH  = 10'd 33,
             VTOTAL       = VACTIVE + VFRONT_PORCH + VSYNC +
                            VBACK_PORCH; // 525

   logic endOfLine;

   always_ff @(posedge clk50 or posedge reset)
     if (reset)          hcount <= 0;
     else if (endOfLine) hcount <= 0;
     else                   hcount <= hcount + 11'd 1;

   assign endOfLine = hcount == HTOTAL - 1;

   logic endOfField;

   always_ff @(posedge clk50 or posedge reset)
     if (reset)          vcount <= 0;
     else if (endOfLine)
```

```
        if (endOfField)    vcount <= 0;
        else               vcount <= vcount + 10'd 1;

    assign endOfField = vcount == VTOTAL - 1;

    // Horizontal sync: from 0x520 to 0x5DF (0x57F)
    // 101 0010 0000 to 101 1101 1111
    assign VGA_HS = !( (hcount[10:8] == 3'b101) &
                       !(hcount[7:5] == 3'b111));
    assign VGA_VS = !( vcount[9:1] == (VACTIVE + VFRONT_PORCH) / 2);

    assign VGA_SYNC_n = 1'b0; // For putting sync on the green
     ↪  signal; unused

    // Horizontal active: 0 to 1279     Vertical active: 0 to 479
    // 101 0000 0000  1280               01 1110 0000  480
    // 110 0011 1111  1599               10 0000 1100  524
    assign VGA_BLANK_n = !( hcount[10] & (hcount[9] | hcount[8]) ) &
                         !( vcount[9] | (vcount[8:5] == 4'b1111) );

    /* VGA_CLK is 25 MHz
     *             __    __    __
     * clk50    __|  |__|  |__|
     *
     *             _____       __
     * hcount[0]__|     |_____|
     */
    assign VGA_CLK = hcount[0]; // 25 MHz clock: rising edge
     ↪  sensitive

endmodule
```

B Hardware Software Interface

B.1 vga_ball.h

```
#ifndef _VGA_BALL_H
#define _VGA_BALL_H

#include <linux/ioctl.h>


typedef struct {
    unsigned long p1_coordinate, p1_state, p1_bomb, p1_firecenter,
     ↪  p1_fireup, p1_firedown, p1_fireleft, p1_fireright,
     ↪  p2_coordinate, p2_state, p2_bomb, p2_firecenter, p2_fireup,
     ↪  p2_firedown, p2_fireleft, p2_fireright, map_info;
} vga_ball_color_t;

typedef struct {
    vga_ball_color_t game_state;
} vga_ball_arg_t;


#define VGA_BALL_MAGIC 'q'

/* ioctls and their arguments */
#define VGA_BALL_WRITE_STATE _IOW(VGA_BALL_MAGIC, 1, vga_ball_arg_t
↪  *)
#define VGA_BALL_READ_STATE  _IOR(VGA_BALL_MAGIC, 2, vga_ball_arg_t
↪  *)



#endif
```

B.2 vga_ball.c

```
/* * Device driver for the VGA video generator
 *
 * A Platform device implemented using the misc subsystem
 *
 * Stephen A. Edwards
 * Columbia University
 *
 * References:
 * Linux source: Documentation/driver-model/platform.txt
 *               drivers/misc/arm-charlcd.c
 * http://www.linuxforu.com/tag/linux-device-drivers/
 * http://free-electrons.com/docs/
 *
 * "make" to build
 * insmod vga_ball.ko
 *
 * Check code style with
 * checkpatch.pl --file --no-tree vga_ball.c
 */

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

#define DRIVER_NAME "vga_ball"
```

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

/*
 * Write segments of a single digit
 * Assumes digit is in range and the device information has been set
 ↪  up
 */

static void write_game_state(vga_ball_color_t *game_state)
{
        iowrite32(game_state->p1_coordinate, dev.virtbase + 0x04);
        iowrite32(game_state->p1_state, dev.virtbase + 0x08);
        iowrite32(game_state->p1_bomb, dev.virtbase + 0x0C);
        iowrite32(game_state->p1_firecenter, dev.virtbase + 0x10);
        iowrite32(game_state->p1_fireup, dev.virtbase + 0x14);
        iowrite32(game_state->p1_firedown, dev.virtbase + 0x18);
        iowrite32(game_state->p1_fireleft, dev.virtbase + 0x1C);
        iowrite32(game_state->p1_fireright, dev.virtbase + 0x20);
        iowrite32(game_state->p2_coordinate, dev.virtbase + 0x24);
        iowrite32(game_state->p2_state, dev.virtbase + 0x28);
        iowrite32(game_state->p2_bomb, dev.virtbase + 0x2C);
        iowrite32(game_state->p2_firecenter, dev.virtbase + 0x30);
        iowrite32(game_state->p2_fireup, dev.virtbase + 0x34);
        iowrite32(game_state->p2_firedown, dev.virtbase + 0x38);
        iowrite32(game_state->p2_fireleft, dev.virtbase + 0x3C);
        iowrite32(game_state->p2_fireright, dev.virtbase + 0x40);
        iowrite32(game_state->map_info, dev.virtbase + 0x44);


        dev.game_state = *game_state;
}



/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
 */
static long vga_ball_ioctl(struct file *f, unsigned int cmd, unsigned
↪   long arg)
{
        vga_ball_arg_t vla;
        //vga_ball_position_t position;

        switch (cmd) {
        case VGA_BALL_WRITE_STATE:
                if (copy_from_user(&vla, (vga_ball_arg_t *) arg,
                                   sizeof(vga_ball_arg_t)))
                        return -EACCES;
                write_game_state(&vla.game_state);
                break;

        case VGA_BALL_READ_STATE:
                  vla.game_state = dev.game_state;
                if (copy_to_user((vga_ball_arg_t *) arg, &vla,
                                 sizeof(vga_ball_arg_t)))
                        return -EACCES;
                break;

        default:
                return -EINVAL;
        }

        return 0;
}
```

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

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

/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init vga_ball_probe(struct platform_device *pdev)
{
        int ret;

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

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

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

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

        return 0;

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

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

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id vga_ball_of_match[] = {
        { .compatible = "csee4840,vga_ball-1.0" }, /**/
        {},
};
MODULE_DEVICE_TABLE(of, vga_ball_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
```

```
static struct platform_driver vga_ball_driver = {
        .driver        = {
                .name          = DRIVER_NAME,
                .owner          = THIS_MODULE,
                .of_match_table = of_match_ptr(vga_ball_of_match),
        },
        .remove        = __exit_p(vga_ball_remove),
};

/* Called when the module is loaded: set things up */
static int __init vga_ball_init(void)
{
        pr_info(DRIVER_NAME ": init\n");
        return platform_driver_probe(&vga_ball_driver,
         ↪  vga_ball_probe);
}

/* Calball when the module is unloaded: release resources */
static void __exit vga_ball_exit(void)
{
        platform_driver_unregister(&vga_ball_driver);
        pr_info(DRIVER_NAME ": exit\n");
}

module_init(vga_ball_init);
module_exit(vga_ball_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Stephen A. Edwards, Columbia University");
MODULE_DESCRIPTION("VGA ball driver");
```

C Game Logic

C.1 main.h

```
#ifndef _MAIN_H
#define _MAIN_H

#include <stdio.h>
#include "vga_ball.h"
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include "controller.h"
#include <libusb-1.0/libusb.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdbool.h>
#include <stdint.h>
#include <time.h>




/*game_state.h*/
#define MAP_SIZE_H 40
#define MAP_SIZE_V 30
#define TILE_SIZE 16
#define PLAYER_NUM 2


//initialise the map
void generateMap(int matrix[MAP_SIZE_H][MAP_SIZE_V], float ratio0,
↪  float ratio1, float ratio2);
void initialisemap(void);
```

```c
typedef enum {
    TERRAIN_GROUND,
    TERRAIN_WALL_BREAKABLE,
    TERRAIN_WALL_UNBREAKABLE,
    TERRAIN_WALL_BREAKABLE_B
} Terrain;



#define DEFAULT_MOVE_SPEED 2
#define DEFAULT_DIRECTION DIRECTION_RIGHT
#define DEFAULT_MAX_BOMBS 1

typedef enum {
    PLAYER_ONE   = 0,
    PLAYER_TWO   = 1,
    PLAYER_THREE = 2,
    PLAYER_FOUR  = 3,
    PLAYER_NONE
} player_id;

player_id playRound(void);

/*position.h*/


typedef struct {
    int16_t x;
    int16_t y;
} Position;


/*direction.h*/
typedef enum {
    DIRECTION_DOWN,
    DIRECTION_UP,
    DIRECTION_LEFT,
    DIRECTION_RIGHT,
    DIRECTION_IDLE
} direction;


/*Bomb.h*/
#define DEFAULT_BOMB_TIMER 20
#define DEFAULT_BOMB_RANGE 1
#define DEFAULT_BOMB_TYPE BOMB_TYPE_NORMAL

typedef enum {
    BOMB_EMPTY,
    BOMB_TYPE_NORMAL
} bomb_type;

/*explosition.h*/
#define DEFAULT_EXPLOSION_TIMER 20
typedef enum {
    EXPLOSION_EMPTY,
    EXPLOSION_TYPE_NORMAL,
    EXPLOSION_TYPE_CENTER,
    EXPLOSION_TYPE_LEFT,
    EXPLOSION_TYPE_RIGHT,
    EXPLOSION_TYPE_UP,
    EXPLOSION_TYPE_DOWN,
    EXPLOSION_TYPE_PERMANENT,
} explosion_type;


typedef struct {
    explosion_type type;
    int32_t timer;
    int owner;       // todo make a player_id
    bool up;
    bool down;
    bool right;
    bool left;
} Explosion;
```

```c
typedef struct {
    int owner;       // todo make a player_id
    int32_t timer;   // 0 when out of time
    int8_t range;    // number of grid units for the explosion
    bomb_type type;
    Position position;
    Explosion explosion;
    bool used;

    bool current_frame;
} Bomb;


typedef struct {
    player_id id;
    bool alive;
    Position tile_position;
    Position screen_position;
    Bomb bomb;

    bool moving;
    int8_t move_speed;
    direction move_direction;

    bool plant_bomb;
    int8_t max_bomb_number;
    int8_t current_bomb_number;

    uint8_t current_frame;

    uint8_t number_of_wins;
} Player;


extern Terrain terrain_grid[MAP_SIZE_H][MAP_SIZE_V];
extern Bomb bomb_grid[MAP_SIZE_H][MAP_SIZE_V];
extern Explosion explosion_grid[MAP_SIZE_H][MAP_SIZE_V];
extern bool changed_tiles[MAP_SIZE_H][MAP_SIZE_V];
extern Player players[PLAYER_NUM];

/*movelogic.h*/
void move(Player *player, direction dir);
bool checkWalkable(int32_t x, int32_t y);
bool readyScreen(void);

void getOccupiedTiles(Player* player, Position* tile1, Position*
↪  tile2);
void screenToTile(Position *pos);
bool isAligned(int32_t x, int32_t y);
void setPlayerTilePos(Player* player);
void setPlayerPosition(Player* player, int32_t x, int32_t y);
bool snapToGrid(Player* player, direction dir);
void clamp_x(int32_t* i);
void clamp_y(int32_t* i);
bool walkingOutOfBounds(Player* player, direction dir);

void getPixelOffset(Position* result, direction dir);
void getNextStep(Position* pos, direction dir);
void getOffsets(Position* next_pos, Position* pixel_offset, Player*
↪  player, direction dir);


/*bomb_logic.h*/
#define BOMB_ANIMATION_CYCLES 5

void countdownExplosions(void);

bool plantBombs(void);

bool countdownBombs(void);

void plantBomb(Player *player);

void explodeBomb(Bomb *bomb);
void explodeTile(int8_t x, int8_t y, Explosion *explosion);

void killPlayersInExplosion(void);


player_id playRound(void);

#endif
```

C.2 main.c

```
#include "main.h"


struct controller_list open_controllers();

int vga_ball_fd;

unsigned short coords;
unsigned short coords2;

vga_ball_color_t color = {0};

pthread_t p1b, p2b;

bool player1_ready = false;
bool player2_ready = false;


Terrain terrain_grid[MAP_SIZE_H][MAP_SIZE_V];
Bomb bomb_grid[MAP_SIZE_H][MAP_SIZE_V];
Explosion explosion_grid[MAP_SIZE_H][MAP_SIZE_V];
bool changed_tiles[MAP_SIZE_H][MAP_SIZE_V];
Player players[PLAYER_NUM];

int map[MAP_SIZE_H][MAP_SIZE_V];
```

```
struct controller_list devices;
struct controller_pkt controller1, controller2;
int fields1, fields2;
int size1 = sizeof(controller1);
int size2 = sizeof(controller2);

player_id getWinner(void) {
    for (int i = 0; i < PLAYER_NUM; i++) {
        if (players[i].alive) {
            return players[i].id;
        }
    }
    return PLAYER_NONE;
}



int main(){


    vga_ball_arg_t vla;
    static const char filename[] = "/dev/vga_ball";

  static const vga_ball_color_t colors[] = {
    { 0xff, 0x00, 0x00 }, /* Red */
    { 0x00, 0xff, 0x00 }, /* Green */
    { 0x00, 0x00, 0xff }, /* Blue */
    { 0xff, 0xff, 0x00 }, /* Yellow */
    { 0x00, 0xff, 0xff }, /* Cyan */
    { 0xff, 0x00, 0xff }, /* Magenta */
    { 0x80, 0x80, 0x80 }, /* Gray */
    { 0x00, 0x00, 0x00 }, /* Black */
    { 0xff, 0xff, 0xff }  /* White */
  };
  devices = open_controllers();
  size1 = sizeof(controller1);
```

```
size2 = sizeof(controller2);


# define COLORS 9

printf("Tanks Game Userspace program started\n");

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



  float ratio0 = 0.5, ratio1 = 0.3, ratio2 = 0.2; // 0 for ground,
   ↪  1 for unbreakable wall, 2 for breakable wall

  generateMap(map, ratio0, ratio1, ratio2);

  for (int i = 0; i < 3; i++)
  {
      for (int j = 0; j < 3; j++)
      {
          map[i][j] = 0;
      }

  }
  for (int i = MAP_SIZE_H-3; i < MAP_SIZE_H; i++)
  {
      for (int j = MAP_SIZE_V-3; j < MAP_SIZE_V; j++)
      {
          map[i][j] = 0;
      }

  }


  initialisemap();
```

```
    initialisePlayers();
    setupRound();



    for(;;){
        runGame();
        if (players[0].alive == false || players[1].alive == false)
            {
                exit(1);
            }
    }


}


void generateMap(int matrix[MAP_SIZE_H][MAP_SIZE_V], float ratio0,
 ↪  float ratio1, float ratio2) {
    int totalElements = MAP_SIZE_H * MAP_SIZE_V;
    int count0 = totalElements * ratio0;
    int count1 = totalElements * ratio1;
    int count2 = totalElements * ratio2;

    int *array = (int *)malloc(totalElements * sizeof(int));
    int index = 0;

    for (int i = 0; i < count0; i++) array[index++] = 0;
    for (int i = 0; i < count1; i++) array[index++] = 1;
    for (int i = 0; i < count2; i++) array[index++] = 2;

    srand(time(NULL));
    for (int i = totalElements - 1; i > 0; i--) {
        int j = rand() % (i + 1);
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
    index = 0;
    for (int i = 0; i < MAP_SIZE_H; i++) {
        for (int j = 0; j < MAP_SIZE_V; j++) {
            matrix[i][j] = array[index++];
        }
    }
    int currentRow = 0, currentCol = 0;
    matrix[currentRow][currentCol] = 0;

    while (currentRow < MAP_SIZE_H - 1 || currentCol < MAP_SIZE_V -
     ↪  1) {
        if (currentRow < MAP_SIZE_H - 1 && (currentCol == MAP_SIZE_V
         ↪  - 1 || rand() % 2)) {
            currentRow++;
        } else {
            currentCol++;
        }
        matrix[currentRow][currentCol] = (rand() % 2) * 2;
    }

    free(array);
}



void runGame(void){


    while (readyScreen()) {
        player_id winner = playRound();
        if (PLAYER_NUM<1)
        {
            /* code */
        }

   }
```

```
}


void initialisemap(void){
    uint32_t map_info;
    uint32_t map_address;
    for (int i = 0; i < MAP_SIZE_H; i++) {
        for (int j = 0; j < MAP_SIZE_V; j++) {
            color.map_info = 0x0;
            map_info = 0x0;
            map_info |= 0x80000000;
            map_info |= 0x40000000;
            map_info |= map[i][j];
            map_address = i + j*40;
            map_info |= (map_address << 19);
            color.map_info |= map_info;
            set_background_color(&color);
        }
    }
    map_info &= 0x80000000;

}


void initialisePlayers(void) {
    for (int i = 0; i < PLAYER_NUM; i++) {
        players[i].id = i;
        players[i].number_of_wins = 0;
            players[i].alive = true;
            players[i].plant_bomb = false;

    }
}


bool readyScreen(void) {
    while(!(player1_ready && player2_ready)) {
        updateControls();
        uint8_t p1_ab = controller1.ab;
        uint8_t p2_ab = controller2.ab;
        uint32_t mask = 1 << 31;

        if(!player1_ready && (p1_ab == 79 || p1_ab == 95 || p1_ab ==
         ↪  207 || p1_ab == 223)) {
            color.p1_state |= mask; //mask
            player1_ready = true;
            set_background_color(&color);
        }
        if(!player2_ready && (p2_ab == 79 || p1_ab == 95 || p1_ab ==
         ↪  207 || p1_ab == 223)) {
            color.p2_state |= mask; //mask
            player2_ready = true;
            set_background_color(&color);
        }
    }


    return true;

}

void updateControls(void) {

    libusb_interrupt_transfer(devices.device1, devices.device1_addr,
     ↪  (unsigned char *) &controller1, size1, &fields1, 0);
    libusb_interrupt_transfer(devices.device2, devices.device2_addr,
     ↪  (unsigned char *) &controller2, size2, &fields2, 0);
    applyPlayerInput(&players[PLAYER_ONE], &controller1);
    applyPlayerInput(&players[PLAYER_TWO], &controller2);
}

void setupRound(void) {
    setupGrids();
    setupPlayers();
    drawPlayers();
}

void setupGrids(void) {

    /* Set up default values */
    for (int i = 0; i < MAP_SIZE_H; i++) {
        for (int j = 0; j < MAP_SIZE_V; j++) {
            switch (map[i][j])
            {
            case 0:
                terrain_grid[i][j] = TERRAIN_GROUND;
                break;
            case 1:
                terrain_grid[i][j] = TERRAIN_WALL_UNBREAKABLE;
                break;
            case 2:
                terrain_grid[i][j] = TERRAIN_WALL_BREAKABLE;
            }


            bomb_grid[i][j].type = BOMB_EMPTY;

            bomb_grid[i][j].timer = 0;
            bomb_grid[i][i].used = 0;
            bomb_grid[i][j].range = 0;

            explosion_grid[i][j].type = EXPLOSION_EMPTY;
            explosion_grid[i][j].timer = 0;
            explosion_grid[i][j].up = 0;
            explosion_grid[i][j].down = 0;
            explosion_grid[i][j].left = 0;
            explosion_grid[i][j].right = 0;

            changed_tiles[i][j] = false;
        }
    }
}

void setupPlayers(void) {

    for (int i = 0; i < PLAYER_NUM; i++) {
        if (i == 0)
        {
            players[i].screen_position.x = 0;
            players[i].screen_position.y = 0;
            players[i].id = 0;
        }
        else
        {
            players[i].screen_position.x = (MAP_SIZE_H - 1) *
             ↪  TILE_SIZE + 1;
            players[i].screen_position.y = (MAP_SIZE_V - 1) *
             ↪  TILE_SIZE + 1;
            players[i].id = 1;
        }


        players[i].alive = true;

        players[i].bomb.owner = players[i].id;
        players[i].bomb.timer = DEFAULT_BOMB_TIMER;
        players[i].bomb.range = DEFAULT_BOMB_RANGE;
        players[i].bomb.type  = DEFAULT_BOMB_TYPE;
            players[i].bomb.owner = players[i].id;
        players[i].bomb.current_frame  = 0;
        players[i].bomb.explosion.type = EXPLOSION_TYPE_NORMAL;
        players[i].bomb.explosion.timer = 0;
        players[i].bomb.explosion.up = 0;
        players[i].bomb.explosion.down = 0;
        players[i].bomb.explosion.right = 0;
        players[i].bomb.explosion.left = 0;
        players[i].bomb.used = 0;

        players[i].moving          = false;
        players[i].move_speed      = DEFAULT_MOVE_SPEED;
        players[i].move_direction  = DEFAULT_DIRECTION;

        players[i].max_bomb_number     = DEFAULT_MAX_BOMBS;
        players[i].current_bomb_number = 0;

    }


}

void set_background_color(const vga_ball_color_t *c)
{
  vga_ball_arg_t vla;
  vla.game_state = *c;
  if (ioctl(vga_ball_fd, VGA_BALL_WRITE_STATE, &vla)) {
      perror("ioctl(VGA_BALL_SET_BACKGROUND) failed");
      return;
  }
}


void drawPlayers(void){
    for (int i = 0; i < PLAYER_NUM; i++) {

        if (!(players[i].alive)) {
            continue;
        }
        uint32_t coordinate = 0;
        uint32_t state = 0;

        switch (players[i].move_direction) {
            case DIRECTION_UP:
                coordinate = ((uint32_t) players[i].screen_position.y
                 ↪  << 16) | players[i].screen_position.x;
                state |= ((players[i].move_direction & 0x3) |
                 ↪  (players[i].alive << 2));
                break;
            case DIRECTION_DOWN:
                coordinate = ((uint32_t) players[i].screen_position.y
                 ↪  << 16) | players[i].screen_position.x;
                state |= ((players[i].move_direction & 0x3) |
                 ↪  (players[i].alive << 2));
                break;
            case DIRECTION_LEFT:
                coordinate = ((uint32_t) players[i].screen_position.y
                 ↪  << 16) | players[i].screen_position.x;
                state |= ((players[i].move_direction & 0x3) |
                 ↪  (players[i].alive << 2));
                break;
            case DIRECTION_RIGHT:
                coordinate = ((uint32_t) players[i].screen_position.y
                 ↪  << 16) | players[i].screen_position.x;
                state |= ((players[i].move_direction & 0x3) |
                 ↪  (players[i].alive << 2));
                break;
            case DIRECTION_IDLE:
                coordinate = ((uint32_t) players[i].screen_position.y
                 ↪  << 16) | players[i].screen_position.x;
                state |= ((0x0 & 0x3) | (players[i].alive << 2));
                break;
        }

        if (i==0){
            color.p1_state &= ~0x7;
            color.p1_coordinate = coordinate;
            color.p1_state |= state;
        }
        else{
            color.p2_state &= ~0x7;
            color.p2_coordinate = coordinate;
            color.p2_state |= state;
        }
```

```
        set_background_color(&color);
        usleep(30000);

        //printf("player %d, x position = %d, y position = %d, dir =
         ↪  %u\n", i, players[i].screen_position.x,
         ↪  players[i].screen_position.y,
         ↪  players[i].move_direction);

    }
}


player_id playRound(void) {

    int frames = 0;
    int players_alive = PLAYER_NUM;
    int final_countdown_count = 0;




    while(players_alive > 1) {

        updateControls();

        set_background_color(&color);
        applyPlayerInput();




        if (plantBombs()) {
        }


        if (countdownBombs()) {
        }
        countdownExplosions();


        render();

    }


}




void applyPlayerInput(void) {
   uint32_t p1_wall;
   uint32_t p2_wall;

    if (controller1.ab == 47 | controller1.ab == 63 | controller1.ab
     ↪  == 175 | controller1.ab == 191) {
        players[PLAYER_ONE].plant_bomb = true;
    } else if ((player1_ready == true) && (player2_ready == true) &&
     ↪  (controller1.ab == 79 | controller1.ab == 95 | controller1.ab
     ↪  == 207 | controller1.ab == 223)) {
        int x = players[PLAYER_ONE].tile_position.x;
        int y = players[PLAYER_ONE].tile_position.y;
        if (terrain_grid[x][y] == TERRAIN_GROUND) {

            changed_tiles[x][y] = true;
            p1_wall = (x + y * MAP_SIZE_H);
            uint32_t map_info = 0x80000000 | 0x40000000 | 0x2;
            map_info |= (p1_wall << 19);
            if(p1_wall != 0){
            terrain_grid[x][y] = TERRAIN_WALL_BREAKABLE;
            color.map_info = map_info;
            set_background_color(&color);
        }
        }
    }
```

```
if (controller1.dir_x == 255) {
    move(&players[PLAYER_ONE], DIRECTION_RIGHT);
} else if (controller1.dir_y == 0) {
    move(&players[PLAYER_ONE], DIRECTION_UP);
} else if (controller1.dir_x == 0) {
    move(&players[PLAYER_ONE], DIRECTION_LEFT);
} else if (controller1.dir_y == 255) {
    move(&players[PLAYER_ONE], DIRECTION_DOWN);
} else {
    move(&players[PLAYER_ONE], DIRECTION_IDLE);
}

if (controller2.ab == 47 | controller2.ab == 63 | controller2.ab
↪   == 175 | controller2.ab == 191) {
    players[PLAYER_TWO].plant_bomb = true;
} else if ((player1_ready == true) && (player2_ready == true) &&
↪   (controller2.ab == 79 | controller2.ab == 95 | controller2.ab
↪   == 207 | controller2.ab == 223)) {
    int x = players[PLAYER_TWO].tile_position.x;
    int y = players[PLAYER_TWO].tile_position.y;
    if (terrain_grid[x][y] == TERRAIN_GROUND) {

        changed_tiles[x][y] = true;
        p2_wall = (x + y * MAP_SIZE_H);
        uint32_t map_info = 0x80000000 | 0x40000000 | 0x2;
        map_info |= (p2_wall << 19);
       if(p2_wall != 0x4AF){
        terrain_grid[x][y] = TERRAIN_WALL_BREAKABLE;
        color.map_info = map_info;
        set_background_color(&color);
    }
    }
}

if (controller2.dir_x == 255) {
    move(&players[PLAYER_TWO], DIRECTION_RIGHT);
} else if (controller2.dir_y == 0) {
        move(&players[PLAYER_TWO], DIRECTION_UP);
    } else if (controller2.dir_x == 0) {
        move(&players[PLAYER_TWO], DIRECTION_LEFT);
    } else if (controller2.dir_y == 255) {
        move(&players[PLAYER_TWO], DIRECTION_DOWN);
    } else {
        move(&players[PLAYER_TWO], DIRECTION_IDLE);
    }

}

void render(void) {

    /* Render the tiles that have changed */
    for (int i = 0; i < MAP_SIZE_H; i++) {
        for (int j = 0; j < MAP_SIZE_V; j++) {
                redrawTile(i, j);
        }
    }

    drawPlayers();

}

void redrawTile(uint32_t x, uint32_t y) {
    /* Draw terrain */
    if (x >= MAP_SIZE_H || y >= MAP_SIZE_V) { return; }

    uint32_t bomb_coordinate = 0;
    uint32_t map_info = 0;
    uint32_t map_address = 0;


    switch (terrain_grid[x][y]) {
        case TERRAIN_GROUND:
            color.map_info &= 0x80000000;
            break;
        case TERRAIN_WALL_BREAKABLE:
            break;
        case TERRAIN_WALL_UNBREAKABLE:
            break;
        case TERRAIN_WALL_BREAKABLE_B:
            map_info |= 0x80000000;
            map_info |= 0x40000000;
            map_info |= 0x0;
            map_address = x + y*40;
            map_info |= (map_address << 19);
            color.map_info |= map_info;
            set_background_color(&color);

            terrain_grid[x][y] = TERRAIN_GROUND;
            printf("breakable: map_info = %d, x= %d, y = %d\n",
             ↪  color.map_info, x, y);
            usleep(20);

            break;
    }

    switch (bomb_grid[x][y].type) {
        case BOMB_TYPE_NORMAL:
                //bomb_coordinate = (((y << 4 - y) << 10) | (x << 4
                 ↪  - x));
                bomb_coordinate = ((y * TILE_SIZE) << 10 | x *
                 ↪  TILE_SIZE );
        if((!bomb_grid[x][y].used) && (bomb_grid[x][y].owner == 0) &&
         ↪  ((color.p1_state & 0x8) != 0x8 )){
            color.p1_bomb = bomb_coordinate;
            color.p1_state |= 0x8;
            bomb_grid[x][y].used = 1;
            set_background_color(&color);
        }
        else if((!bomb_grid[x][y].used) && (bomb_grid[x][y].owner ==
         ↪  1) && ((color.p2_state & 0x8) != 0x8 ))
            {
            color.p2_bomb = bomb_coordinate;
            color.p2_state |= 0x8;
        bomb_grid[x][y].used = 1;
        set_background_color(&color);
    }

            break;
    case BOMB_EMPTY:
            break;
    }



/* Draw bombs */
int32_t timer = bomb_grid[x][y].timer;

uint32_t explosion_coordinate = 0;

switch (explosion_grid[x][y].type) {
        //printf("explode position = %d\n",
         ↪  explosion_coordinate);
        //explosion_coordinate = ((y * TILE_SIZE) << 10 | x *
         ↪  TILE_SIZE );
    case EXPLOSION_EMPTY:
        break;
    case EXPLOSION_TYPE_NORMAL:
        if (explosion_grid[x][y].owner == 0) {
              explosion_coordinate = ((y * TILE_SIZE) << 10 | x *
               ↪  TILE_SIZE );
                color.p1_firecenter = explosion_coordinate;
                color.p1_state |= 0x10;
                if (explosion_grid[x][y].up == 1)
                {

                    explosion_coordinate = (((y-1) * TILE_SIZE)
                     ↪  << 10 | x * TILE_SIZE );
                    if (explosion_grid[x][y-1].type !=
                     ↪  EXPLOSION_TYPE_NORMAL)
                    {
```

```
                                explosion_grid[x][y-1].type =
                                 ↪  EXPLOSION_TYPE_UP;
        }

        //printf("down cor = %d\n",
         ↪  explosion_coordinate);
        //printf("%d, %d\n", x, y);
        color.p1_fireup = explosion_coordinate;
        color.p1_state |= 0x20;
    }
    if (explosion_grid[x][y].down == 1)
    {
        explosion_coordinate = (((y+1) * TILE_SIZE)
         ↪  << 10 | x * TILE_SIZE );

        if (explosion_grid[x][y+1].type !=
         ↪  EXPLOSION_TYPE_NORMAL)
        {
            explosion_grid[x][y+1].type =
             ↪  EXPLOSION_TYPE_DOWN;
        }
        //printf("down cor = %d\n",
         ↪  explosion_coordinate);
        //printf("%d, %d\n", x, y);
        color.p1_firedown = explosion_coordinate;
        color.p1_state |= 0x40;
    }
    if (explosion_grid[x][y].left == 1)
    {
        explosion_coordinate = ((y * TILE_SIZE) << 10
         ↪  | (x-1) * TILE_SIZE );
        if (explosion_grid[x-1][y].type !=
         ↪  EXPLOSION_TYPE_NORMAL)
        {
                                explosion_grid[x-1][y].type =
                                 ↪  EXPLOSION_TYPE_LEFT;
        }
        color.p1_fireleft = explosion_coordinate;
```

```
            color.p1_state |= 0x80;
        }
        if (explosion_grid[x][y].right == 1)
        {
            explosion_coordinate = ((y * TILE_SIZE) << 10
             ↪  | (x+1) * TILE_SIZE );
            if (explosion_grid[x+1][y].type !=
             ↪  EXPLOSION_TYPE_NORMAL)
            {
                            explosion_grid[x+1][y].type =
                             ↪  EXPLOSION_TYPE_RIGHT;
            }
            color.p1_fireright = explosion_coordinate;
            color.p1_state |= 0x100;
        }
            set_background_color(&color);
    } else if (explosion_grid[x][y].owner == 1)
    {
        explosion_coordinate = ((y * TILE_SIZE) << 10 | x
         ↪  * TILE_SIZE );
        color.p2_firecenter = explosion_coordinate;
        color.p2_state |= 0x10;
        if (explosion_grid[x][y].up == 1)
        {
            explosion_coordinate = (((y-1) * TILE_SIZE)
             ↪  << 10 | x * TILE_SIZE );
            explosion_grid[x][y-1].type =
             ↪  EXPLOSION_TYPE_UP;
            color.p2_fireup = explosion_coordinate;
            color.p2_state |= 0x20;
        }
        if (explosion_grid[x][y].down == 1)
        {
            explosion_coordinate = (((y+1) * TILE_SIZE)
             ↪  << 10 | x * TILE_SIZE );
            explosion_grid[x][y+1].type =
             ↪  EXPLOSION_TYPE_DOWN;
            color.p2_firedown = explosion_coordinate;
```

```
                    //printf("player 2\n");
                    //printf("%d\n", explosion_grid[x][y].left);
                    color.p2_state |= 0x40;
                }
                if (explosion_grid[x][y].left == 1)
                {
                    //printf(" in the switch");
                    explosion_coordinate = ((y * TILE_SIZE) << 10
                     ↪   | (x-1) * TILE_SIZE );
                    explosion_grid[x-1][y].type =
                     ↪   EXPLOSION_TYPE_LEFT;
                    color.p2_fireleft = explosion_coordinate;
                    color.p2_state |= 0x80;
                    //printf("%d",color.p2_state);
                }
                if (explosion_grid[x][y].right == 1)
                {
                    explosion_coordinate = ((y * TILE_SIZE) << 10
                     ↪   | (x+1) * TILE_SIZE );
                    explosion_grid[x+1][y].type =
                     ↪   EXPLOSION_TYPE_RIGHT;
                    color.p2_fireright = explosion_coordinate;
                    color.p2_state |= 0x100;
                }
                    set_background_color(&color);
            }
        break;
    }

}

void move(Player *player, direction dir) {

    /* Check for out of bounds movement */

    if (walkingOutOfBounds(player, dir)) {
        dir = DIRECTION_IDLE;
    }
```

```
//player.move_direction = dir;
Position pixel_offset = {0, 0};
Position next_pos = {0, 0};
getOffsets(&next_pos, &pixel_offset, player, dir);

bool aligned = isAligned(player->screen_position.x,
 ↪  player->screen_position.y);

Position tile1 = {0, 0};
Position tile2 = {0, 0};
getOccupiedTiles(player, &tile1, &tile2);

/* Mark tiles for rendering */
changed_tiles[tile1.x][tile1.y] = true;
changed_tiles[tile2.x][tile2.y] = true;

/* When joystick is not moving but player is still mid movement
 ↪  OR when trying to change directions */
if (dir == DIRECTION_IDLE || dir != player->move_direction) {
    if (!aligned) {
        bool snappedToGrid = snapToGrid(player, dir);
        if (snappedToGrid) {
            return;
        }

        /* Move normally if not */
        int32_t temp_x = player->screen_position.x + next_pos.x *
         ↪  player->move_speed;
        int32_t temp_y = player->screen_position.y + next_pos.y *
         ↪  player->move_speed;
        clamp_x(&temp_x);
        clamp_y(&temp_y);

        setPlayerPosition(player, temp_x, temp_y);
    } else {
        /* We are grid-aligned -> can change directions now (or
         ↪  stop moving) */
        if (dir != player->move_direction && dir !=
         ↪  DIRECTION_IDLE) {
            player->move_direction = dir;
        } else {
            player->moving = false;
        }
    }
    setPlayerTilePos(player);
    return;
}

/* Check if we're stepping onto another tile */
for (int i = 0; i <= player->move_speed; i++) {
    int32_t temp_x = player->screen_position.x + next_pos.x * i +
     ↪  pixel_offset.x;
    int32_t temp_y = player->screen_position.y + next_pos.y * i +
     ↪  pixel_offset.y;
    clamp_x(&temp_x);
    clamp_y(&temp_y);

    if (isAligned(temp_x, temp_y)) {

        /* We found another tile */
        Position temp;
        temp.x = temp_x;
        temp.y = temp_y;
        screenToTile(&temp);

        /* Stop if we can't walk on it */
        if (!checkWalkable(temp.x, temp.y)) {
            setPlayerPosition(player, player->screen_position.x +
             ↪  next_pos.x * i, player->screen_position.y +
             ↪  next_pos.y * i);
            player->moving = false;
            player->move_direction = dir;
            setPlayerTilePos(player);
```

```
                return;
            }

            break;
        }
    }

    /* No other case, so just keep moving */
    int32_t temp_x = player->screen_position.x + next_pos.x *
    ↪  player->move_speed;
    int32_t temp_y = player->screen_position.y + next_pos.y *
    ↪  player->move_speed;
    clamp_x(&temp_x);
    clamp_y(&temp_y);

    setPlayerPosition(player, temp_x, temp_y);

    player->moving = true;
    player->move_direction = dir;
    setPlayerTilePos(player);
}


void getNextStep(Position* pos, direction dir) {
    switch (dir) {
        case DIRECTION_UP:
            pos->y = -1;
            break;
        case DIRECTION_DOWN:
            pos->y = 1;
            break;
        case DIRECTION_LEFT:
            pos->x = -1;
            break;
        case DIRECTION_RIGHT:
            pos->x = 1;
            break;
        default:
            break;
    }
}

void getOffsets(Position* next_pos, Position* pixel_offset, Player*
 ↪  player, direction dir) {
    getPixelOffset(pixel_offset, dir);

    getNextStep(next_pos, player->move_direction);

}


bool snapToGrid(Player* player, direction dir) {

    Position pixel_offset = {0, 0};
    Position next_pos = {0, 0};
    getOffsets(&next_pos, &pixel_offset, player, dir);

    for (int i = 0; i <= player->move_speed; i++) {
        int32_t temp_x = player->screen_position.x + next_pos.x * i +
        ↪  pixel_offset.x;
        int32_t temp_y = player->screen_position.y + next_pos.y * i +
        ↪  pixel_offset.y;
        clamp_x(&temp_x);
        clamp_y(&temp_y);

        if (isAligned(temp_x, temp_y)) {
            setPlayerPosition(player, player->screen_position.x +
            ↪  next_pos.x * i, player->screen_position.y +
            ↪  next_pos.y * i);

            /* If the direction is changing, change the player
            ↪  attribute */
            if (dir != player->move_direction && dir !=
            ↪  DIRECTION_IDLE) {
                player->move_direction = dir;
            } else {
                /* Otherwise we're stopping */
                player->moving = false;
            }
            setPlayerTilePos(player);
            return true;
        }
    }
    return false;
}

/* Returns whether tile at given *tile* position can be walked on
 ↪  */
bool checkWalkable(int32_t x, int32_t y) {

    if (x < 0 || y < 0 || x >= MAP_SIZE_H || y >= MAP_SIZE_V) {
     ↪  return false; }
    if (bomb_grid[x][y].type != BOMB_EMPTY) { return false; }
    if (terrain_grid[x][y] != TERRAIN_GROUND) { return false; }

    return true;
}

/* Returns whether given *screen* position is aligned with the tile
 ↪  grid */
bool isAligned(int32_t x, int32_t y) {
    return (x % TILE_SIZE == 0) && (y % TILE_SIZE == 0);
}

/* Gets tile coordinates of tiles that are occupied by a player. If
 ↪  the player is
 * grid-aligned, tile1 will have the same coordinates as tile2 */
void getOccupiedTiles(Player* player, Position* tile1, Position*
 ↪  tile2) {

    if (isAligned(player->screen_position.x,
     ↪  player->screen_position.y)) {
        /* Aligned with grid */
        tile1->x = player->screen_position.x;
        tile1->y = player->screen_position.y;
        screenToTile(tile1);

        tile2->x = tile1->x;
        tile2->y = tile1->y;
    } else {
        int32_t offset_x = player->screen_position.x % TILE_SIZE;
        int32_t offset_y = player->screen_position.y % TILE_SIZE;

        /* First move backwards */
        tile1->x = player->screen_position.x - offset_x;
        tile1->y = player->screen_position.y - offset_y;

        screenToTile(tile1);

        /* Then forwards */
        if (offset_x == 0) {
            tile2->x = player->screen_position.x;
            tile2->y = player->screen_position.y + TILE_SIZE -
             ↪  offset_y;
        } else {
            tile2->y = player->screen_position.y;
            tile2->x = player->screen_position.x + TILE_SIZE -
             ↪  offset_x;
        }

        screenToTile(tile2);
    }
}

/* Gets offset in screen position when trying to check 'in front'
 ↪  */
void getPixelOffset(Position* result, direction dir) {
    switch (dir) {
        case DIRECTION_UP:
            result->x = 0;
            result->y = -TILE_SIZE;
            break;
```

```c
        case DIRECTION_LEFT:
            result->x = -TILE_SIZE;
            result->y = 0;
            break;
        case DIRECTION_IDLE:
            result->x = 0;
            result->y = 0;
            break;
        case DIRECTION_RIGHT:
            result->x = TILE_SIZE;
            result->y = 0;
            break;
        case DIRECTION_DOWN:
            result->x = 0;
            result->y = +TILE_SIZE;
            break;
    }
}


void setPlayerTilePos(Player* player) {
    int32_t offset_x = player->screen_position.x % TILE_SIZE;
    int32_t offset_y = player->screen_position.y % TILE_SIZE;

    Position result;

    if (offset_x == 0) {
        result.x = player->screen_position.x;

        /* Get the closer tile */
        if (offset_y < TILE_SIZE / 2) {
            result.y = player->screen_position.y - offset_y;
        } else {
            result.y = player->screen_position.y + TILE_SIZE -
             ↪  offset_y;
        }
    } else {
        result.y = player->screen_position.y;
        /* Get the closer tile */
        if (offset_x < TILE_SIZE / 2) {
            result.x = player->screen_position.x - offset_x;
        } else {
            result.x = player->screen_position.x + TILE_SIZE -
             ↪  offset_x;
        }
    }

    screenToTile(&result);

    player->tile_position.x = result.x;
    player->tile_position.y = result.y;
}

/* Helper function to convert from screen position to tile position
 ↪  */
void screenToTile(Position *pos) {
    pos->x /= TILE_SIZE;
    pos->y /= TILE_SIZE;

    if (pos->x < 0) {
        pos->x = 0;
    } else if (pos->x >= MAP_SIZE_H) {
        pos->x = MAP_SIZE_H - 1;
    }

    if (pos->y < 0) {
        pos->y = 0;
    } else if (pos->y >= MAP_SIZE_V) {
        pos->y = MAP_SIZE_V - 1;
    }
}

void setPlayerPosition(Player* player, int32_t x, int32_t y) {

    clamp_x(&x);
    clamp_y(&y);

    player->screen_position.x = x;
    player->screen_position.y = y;
}

void clamp_x(int32_t* i) {
    if (*i < 0) {
        *i = 0;
    } else if (*i > (MAP_SIZE_H - 1) * TILE_SIZE) {
        *i = (MAP_SIZE_H - 1) * TILE_SIZE;
    }
}

void clamp_y(int32_t* i) {
    if (*i < 0) {
        *i = 0;
    } else if (*i > (MAP_SIZE_V - 1) * TILE_SIZE) {
        *i = (MAP_SIZE_V - 1) * TILE_SIZE;
    }
}

bool walkingOutOfBounds(Player* player, direction dir) {
    if (player->tile_position.x == 0 && dir == DIRECTION_LEFT) {
        return true;
    }
    if (player->tile_position.y == 0 && dir == DIRECTION_UP) {
        return true;
    }
    if (player->screen_position.x >= (MAP_SIZE_H - 1) * TILE_SIZE -
     ↪  player->move_speed && dir == DIRECTION_RIGHT) {
        player->screen_position.x = (MAP_SIZE_H - 1) * TILE_SIZE;
        return true;
    }
    if (player->screen_position.y >= (MAP_SIZE_V - 1) * TILE_SIZE -
     ↪  player->move_speed && dir == DIRECTION_DOWN) {
        player->screen_position.y = (MAP_SIZE_V - 1) * TILE_SIZE;
        return true;
    }
    return false;
}

void countdownExplosions(void) {

    for (int x = 0; x < MAP_SIZE_H; x++) {
        for (int y = 0; y < MAP_SIZE_V; y++) {

            Explosion *explosion = &explosion_grid[x][y];

            if (explosion->type == EXPLOSION_TYPE_NORMAL) {
                explosion->timer--;
            //printf("explosion owner = %d\n", explosion->owner);
            //printf("explosion timer = %d\n", explosion->timer);


        if (explosion->timer == 0) {
            explosion_grid[x][y].type = EXPLOSION_EMPTY;
                    //printf("%d\n", explosion_grid[x][y].type);
                    if(explosion_grid[x][y].up == 1){
                        explosion_grid[x][y-1].type =
                        ↪  EXPLOSION_EMPTY;
            terrain_grid[x][y-1] = TERRAIN_GROUND;
                        //printf("%d\n",
                        ↪  explosion_grid[x][y-1].type);
                        explosion_grid[x][y].up = 0;
                    }
                    if(explosion_grid[x][y].down == 1){
                        explosion_grid[x][y+1].type =
                        ↪  EXPLOSION_EMPTY;
            terrain_grid[x][y+1] = TERRAIN_GROUND;

                        //printf("%d\n",
                        ↪  explosion_grid[x][y+1].type);
                        explosion_grid[x][y].down = 0;
                    }
                    if(explosion_grid[x][y].left == 1){
```

```
            explosion_grid[x-1][y].type =
            ↪  EXPLOSION_EMPTY;
terrain_grid[x-1][y] = TERRAIN_GROUND;
            //printf("%d\n",
            ↪  explosion_grid[x-1][y].type);
            explosion_grid[x][y].left = 0;
        }
        if(explosion_grid[x][y].right == 1){
            explosion_grid[x+1][y].type =
            ↪  EXPLOSION_EMPTY;
terrain_grid[x+1][y] = TERRAIN_GROUND;
            //printf("%d\n",
            ↪  explosion_grid[x+1][y].type);
            explosion_grid[x][y].right = 0;
        }

      //  printf("explosion owner = %d\n",
      ↪  explosion->owner);
       int mask = 0xFFFFFE0F;
       if(explosion->owner == 0){
            color.p1_state &= mask;
        }
       else if (explosion->owner == 1){
            color.p2_state &= mask;
            //printf("explosion disapper\n");
        }


        /* Set the terrain on this tile to be ground -
        ↪  destroy breakable wall */
        terrain_grid[x][y] = TERRAIN_GROUND;


        /* Update the changed tiles so explosion is no
        ↪  longer displayed */
        changed_tiles[x][y] = true;
    }
}
```

```
        }
    }

    killPlayersInExplosion();

}

bool plantBombs(void) {

    bool bomb_planted = false;
    for (int i = 0; i < PLAYER_NUM; i++) {
        if (players[i].plant_bomb) {
            players[i].plant_bomb = false;
            plantBomb(&(players[i]));
            bomb_planted = true;
        }
    }

    return bomb_planted;
}


bool countdownBombs(void) {
    bool bomb_exploded = false;

    for (int x = 0; x < MAP_SIZE_H; x++) {
        for (int y = 0; y < MAP_SIZE_V; y++) {

            Bomb *bomb = &bomb_grid[x][y];

            if (bomb->type != BOMB_EMPTY) {
                bomb->timer--;

                if ((bomb->timer) == 0) {
                    explodeBomb(bomb);
                    bomb_exploded = true;
                } else if ((bomb->timer) % BOMB_ANIMATION_CYCLES ==
                 ↪  0) {
                    /* Update the changed tiles so bomb is rendered
                     * to indicate imminient explostion */
                    bomb->current_frame = !(bomb->current_frame);
                    changed_tiles[x][y] = true;
                }
            }
        }
    }

    killPlayersInExplosion();
    return bomb_exploded;
}

void plantBomb(Player *player) {

    /* Get the coordinates for convenience */
    int8_t x = player->tile_position.x;
    int8_t y = player->tile_position.y;

    /* Check player is alive, has enough bombs and is not standing
     ↪  on a bomb */
    if ((!(player->alive)) ||
        (player->current_bomb_number >= player->max_bomb_number) ||
        (bomb_grid[x][y].type != BOMB_EMPTY)) { return; }

    /* Copy the player's bomb template into the bomb grid and set
     ↪  its position */
    Bomb bomb = player->bomb;
    bomb.position = player->tile_position;
    bomb_grid[x][y] = bomb;
    //printf("bomb x position = %d, bomb y position = %d, owner =
     ↪  %d\n", x, y, bomb.owner);
    changed_tiles[x][y] = true;

    player->current_bomb_number++;
}

void explodeBomb(Bomb *bomb) {
```

```
/* Check bomb is not empty */
if (bomb->type == BOMB_EMPTY) { return; }

/* Get the coordinates, range and explosion for convenience */
int8_t x = bomb->position.x;
int8_t y = bomb->position.y;
int8_t range = bomb->range;
//printf("bomb position x = %d, y = %d", x, y);


Explosion *explosion = &(bomb->explosion);
explosion->owner = bomb->owner;
explosion->timer = DEFAULT_EXPLOSION_TIMER;

//printf("owner = %d", explosion->owner);
/*
Explosion *explosion_L = &(bomb->explosion);
Explosion *explosion_R = &(bomb->explosion);
Explosion *explosion_D = &(bomb->explosion);
Explosion *explosion_U = &(bomb->explosion);

explosion_L->type = EXPLOSION_TYPE_LEFT;
explosion_R->type = EXPLOSION_TYPE_RIGHT;
explosion_U->type = EXPLOSION_TYPE_UP;
explosion_D->type = EXPLOSION_TYPE_DOWN;
*/



/* Decrement corresponding player bomb count */
players[bomb->owner].current_bomb_number--;

/* Remove bomb and explode tile */
bomb_grid[x][y].type = BOMB_EMPTY;
unsigned int mask = ~(1 << 3);
if (bomb_grid[x][y].owner == 0)
{
    color.p1_bomb = 0x0;
    color.p1_state &= mask;
    bomb_grid[x][y].used = 0;
    set_background_color(&color);
}
else if (bomb_grid[x][y].owner == 1)
{
    color.p2_bomb = 0x0;
    color.p2_state &= mask;
    bomb_grid[x][y].used = 0;
    set_background_color(&color);
}


explosion_grid[x][y] = *explosion;
//printf("bomb owner = %d, explosion owner = %d",
 ↪  bomb_grid[x][y].owner, explosion_grid[x][y].owner);
//explodeTile(x, y, explosion);

//printf("explode position x = %d, y = %d\n", x, y);

/* Explode tiles on the horizontal and vertical that are in
 ↪  range but
 * don't explode past an unbreakable wall */
switch (terrain_grid[x][y])
{
case TERRAIN_WALL_BREAKABLE:
    terrain_grid[x][y] = TERRAIN_WALL_BREAKABLE_B;
    break;

default:
    break;
}

bool left_blocked  = false;
bool right_blocked = false;
bool up_blocked    = false;
bool down_blocked  = false;
```

```
for (int i = 1; i <= range; i++) {

    if (!(up_blocked)) {
        switch (terrain_grid[x][y - i]) {
            case TERRAIN_WALL_UNBREAKABLE:
                        explosion_grid[x][y].up = 0;
                up_blocked = true;
                break;
            case TERRAIN_WALL_BREAKABLE:
                explosion_grid[x][y].up = 1;
                terrain_grid[x][y-i] = TERRAIN_WALL_BREAKABLE_B;
                //printf("up");

                //explodeTile(x, y + i, explosion);
                up_blocked = true;
                break;
            case TERRAIN_GROUND:
                //explodeTile(x, y + i, explosion);
                if (bomb_grid[x][y-i].type == BOMB_TYPE_NORMAL ||
                ↪  explosion_grid[x][y-i].type ==
                ↪  EXPLOSION_TYPE_NORMAL)
                {
                    explosion_grid[x][y].up = 0;
                    up_blocked = true;
                }
                else{
                    explosion_grid[x][y].up = 1;

                }

                break;
        }
    }

    if (!(down_blocked)) {
        switch (terrain_grid[x][y + i]) {
            case TERRAIN_WALL_UNBREAKABLE:
                explosion_grid[x][y].down = 0;
                down_blocked = true;
                break;
            case TERRAIN_WALL_BREAKABLE:
                explosion_grid[x][y].down = 1;
                terrain_grid[x][y+i] = TERRAIN_WALL_BREAKABLE_B;
                //printf("down");
                //explodeTile(x, y - i, explosion);
                            //explosion->down = 0;
                down_blocked = true;
                break;
            case TERRAIN_GROUND:
                //explodeTile(x, y - i, explosion);
                //printf("explosition type = %d\n",
                ↪  explosion_grid[x][y-i].type);
                if (bomb_grid[x][y+i].type == BOMB_TYPE_NORMAL ||
                ↪  explosion_grid[x][y+i].type ==
                ↪  EXPLOSION_TYPE_NORMAL)
                {
                     explosion_grid[x][y].down = 0;
                    down_blocked = true;
                }
                else{
                    explosion_grid[x][y].down = 1;
                }

                break;
        }
    }

    if (!(left_blocked)) {
        switch (terrain_grid[x - i][y]) {
            case TERRAIN_WALL_UNBREAKABLE:
                explosion_grid[x][y].left = 0;
                left_blocked = true;
                break;
            case TERRAIN_WALL_BREAKABLE:
                explosion_grid[x][y].left = 1;
                terrain_grid[x-i][y] = TERRAIN_WALL_BREAKABLE_B;
                //explodeTile(x, y + i, explosion);
                left_blocked = true;
                break;
            case TERRAIN_GROUND:
                //explodeTile(x - i, y, explosion);
                if (bomb_grid[x - i][y].type == BOMB_TYPE_NORMAL
                 ↪  || explosion_grid[x - i][y].type ==
                 ↪  EXPLOSION_TYPE_NORMAL)
                {
                     explosion_grid[x][y].left = 0;
                    left_blocked = true;
                }
                else{
                    explosion_grid[x][y].left = 1;
                }

                //printf("left is not block, %d\n",
                 ↪  explosion_grid[x][y].left);
                break;
        }
    }

    if (!(right_blocked)) {
        switch (terrain_grid[x + i][y]) {
            case TERRAIN_WALL_UNBREAKABLE:
                    explosion_grid[x][y].right = 0;
                right_blocked = true;
                break;
            case TERRAIN_WALL_BREAKABLE:
                explosion_grid[x][y].right = 1;
                terrain_grid[x+i][y] = TERRAIN_WALL_BREAKABLE_B;
                //explodeTile(x, y + i, explosion);
                up_blocked = true;
                right_blocked = true;
                break;
            case TERRAIN_GROUND:
                //explodeTile(x + i, y, explosion);
```

```
                    //printf("explosition type = %d\n",
                     ↪  explosion_grid[x+i][y].type);
                        if (bomb_grid[x + i][y].type ==
                         ↪  BOMB_TYPE_NORMAL || explosion_grid[x +
                         ↪  i][y].type == EXPLOSION_TYPE_NORMAL)
                    {
                        explosion_grid[x][y].right = 0;
                        right_blocked = true;
                    }
                    else{
                        explosion_grid[x][y].right = 1;
                    }
                    break;
            }
        }


    }
    changed_tiles[x][y] = true;
}

void explodeTile(int8_t x, int8_t y, Explosion *explosion) {
    bool isCoordinateInRange(int8_t x, int8_t y) {
            return ((x >= 0) && (x < MAP_SIZE_H)  &&  (y >= 0) && (y
             ↪  < MAP_SIZE_V));
    }

    /* Check position is valid, the explosion type is not empty and
     ↪  there is not
     * an unbreakable wall */
    if ((!isCoordinateInRange(x, y)) ||
        (explosion->type == EXPLOSION_EMPTY) ||
        (terrain_grid[x][y] == TERRAIN_WALL_UNBREAKABLE)) { return; }

    /* Set the explosion, overwriting any existing explosions */
    if (explosion_grid[x][y].type != EXPLOSION_TYPE_PERMANENT) {
        explosion_grid[x][y] = *explosion;
    }
    /* Set any bombs on this tile to explode on the next round */
    bomb_grid[x][y].timer = 1;




}

void killPlayersInExplosion(void) {

    Position pos_1 = {0, 0};
    Position pos_2 = {0, 0};

    for (int i = 0; i < PLAYER_NUM; i++) {
        if
        ↪  (explosion_grid[players[i].tile_position.x][players[i].tile_position.y].t
        ↪  != EXPLOSION_EMPTY) {
            players[i].alive = false;
            uint32_t mask = (1 << 31);     // Set the 32nd bit
            mask |= (0x1F << 4);           // Set the 5th to 9th bits
            ↪  (0x1F = 00011111 in binary)
            if (i==0)
            {
                color.p1_state &= mask;
                set_background_color(&color);
            }
            else {
                color.p2_state &= mask;
                set_background_color(&color);
            }

            //printf("players die\n");

            getOccupiedTiles(&(players[i]), &pos_1, &pos_2);

            /* Render both tiles the player occupied */
            changed_tiles[pos_1.x][pos_1.y] = true;
            changed_tiles[pos_2.x][pos_2.y] = true;

        }
    }
}
```

D Controller

D.1 controller.h

```
#ifndef _CONTROLLER_H
#define _CONTROLLER_H

#include <stdio.h>
#include <stdlib.h>
#include <libusb-1.0/libusb.h>

struct controller_list {

        struct libusb_device_handle *device1;
        struct libusb_device_handle *device2;
        uint8_t device1_addr;
        uint8_t device2_addr;

};

struct controller_pkt {

        uint8_t const1;
        uint8_t const2;
        uint8_t const3;
        uint8_t dir_x;
        uint8_t dir_y;
        uint8_t ab;
        uint8_t rl;

};

struct args_list {

        struct controller_list devices;
        char * buttons;
        int mode;
        int print;

};

extern struct controller_list open_controller(uint8_t *);

#endif
```

D.2 controller.c

```
#include "controller.h"
#include <libusb-1.0/libusb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct controller_list open_controllers() {

        printf("Searching for USB connections...\n");

        uint8_t endpoint_address = 0;
        struct controller_list devices;
        libusb_device **devs;
        struct libusb_device_descriptor desc;
        struct libusb_device_handle *controller = NULL;
        ssize_t num_devs;
```

```
// Boot libusb library
if (libusb_init(NULL) != 0) {
        printf("\nERROR: libusb failed to boot");
        exit(1);
}

if  ((num_devs = libusb_get_device_list(NULL, &devs)) < 0) {
        printf("\nERROR: no controllers found");
        exit(1);
}

int connection_count = 0;
for (int i = 0; i < num_devs; i++) {

        libusb_device *dev = devs[i];

        if (libusb_get_device_descriptor(dev, &desc) < 0) {
                printf("\nERROR: bad device descriptor.");
                exit(1);
        }

        // Our controllers have idProduct of 0x11
        if (desc.idProduct == 0x11) {

                struct libusb_config_descriptor *config;
                if ((libusb_get_config_descriptor(dev, 0,
                ↪   &config)) < 0) {
                        printf("\nERROR: bad config
                        ↪   descriptor.");
                        exit(1);
                }


                int r;
                const struct libusb_interface_descriptor
                ↪   *inter = config->interface[0].altsetting;
```

```
                    if ((r = libusb_open(dev, &controller)) != 0)
                     ↪  {
                            printf("\nERROR: couldn't open
                             ↪  controller");
                            exit(1);
                    }
                    if (libusb_kernel_driver_active(controller,
                     ↪  0)) {

                             ↪  libusb_detach_kernel_driver(controller,
                             ↪  0);
                    }

                     ↪  libusb_set_auto_detach_kernel_driver(controller,
                     ↪  0);
                    if ((r = libusb_claim_interface(controller,
                     ↪  0)) != 0) {
                            printf("\nERROR: couldn't claim
                             ↪  controller.");
                            exit(1);
                    }

                    endpoint_address =
                     ↪  inter->endpoint[0].bEndpointAddress;
                    connection_count++;

                    if (connection_count == 1) {
                            devices.device1 = controller;
                            devices.device1_addr =
                             ↪  endpoint_address;
                    } else {
                            devices.device2 = controller;
                            devices.device2_addr =
                             ↪  endpoint_address;
                    }
            }
    }
```

```
        if (connection_count < 2) {
                printf("ERROR: couldn't find 2 controllers.");
                exit(1);
        }

        found:
                printf("Connected %d controllers!\n",
                 ↪  connection_count);
                libusb_free_device_list(devs, 1);

        return devices;
}

void detect_presses(struct controller_pkt pkt, char *buttons, int
 ↪  mode) {

        char vals[] = "LRUDA";
        if (mode == 1) {
                strcpy(buttons, "00000");
                strcpy(vals, "11111");
        } else {
                strcpy(buttons, "_____");
        }

        if (pkt.dir_x == 0) {
                buttons[0] = vals[0];
        } else if (pkt.dir_x == 0xff) {
                buttons[1] = vals[1];
        }

        if (pkt.dir_y == 0x00) {
                buttons[2] = vals[2];
        } else if (pkt.dir_y == 0xff) {
                buttons[3] = vals[3];
        }

        // Check if button (A) is pressed
        uint8_t a = pkt.ab;
        if (a == 47 || a == 63 || a == 111 || a == 127 || a == 175 ||
         ↪  a == 191 || a == 239 || a == 255) {
                buttons[4] = vals[4];
        }

}

void *listen_controllers(void *arg) {

        struct args_list *args_p = arg;
        struct args_list args = *args_p;
        struct controller_list devices = args.devices;

        struct controller_pkt pkt1, pkt2;
        int fields1, fields2;
        int size1 = sizeof(pkt1);
        int size2 = sizeof(pkt2);
        char buttons1[] = "_____";
        char buttons2[] = "_____";

        for (;;) {

                libusb_interrupt_transfer(devices.device1,
                 ↪  devices.device1_addr, (unsigned char *) &pkt1,
                 ↪  size1, &fields1, 0);
                libusb_interrupt_transfer(devices.device2,
                 ↪  devices.device2_addr, (unsigned char *) &pkt2,
                 ↪  size2, &fields2, 0);

                // 7 fields should be transferred for each packet
                if (fields1 == 7 && fields2 == 7) {
                        detect_presses(pkt1, buttons1, args.mode);
                        detect_presses(pkt2, buttons2, args.mode);
                        strcat(buttons1, buttons2);
                        strcpy(args.buttons, buttons1);
                        if (args.print == 1) {
                                printf("%s\n", args.buttons);
                        }
                }
            }
        }
```