Final Report

Natalie Hughes, Qian Zhao, Shiyan Wang

May 2024

Contents

1 Introduction

1.1 Game Overview

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

1.2 System Architecture

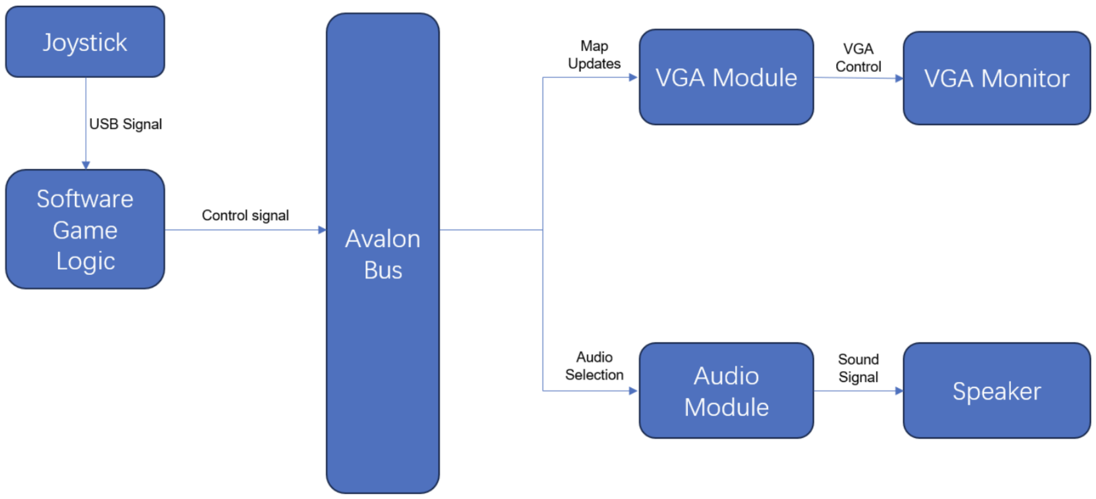
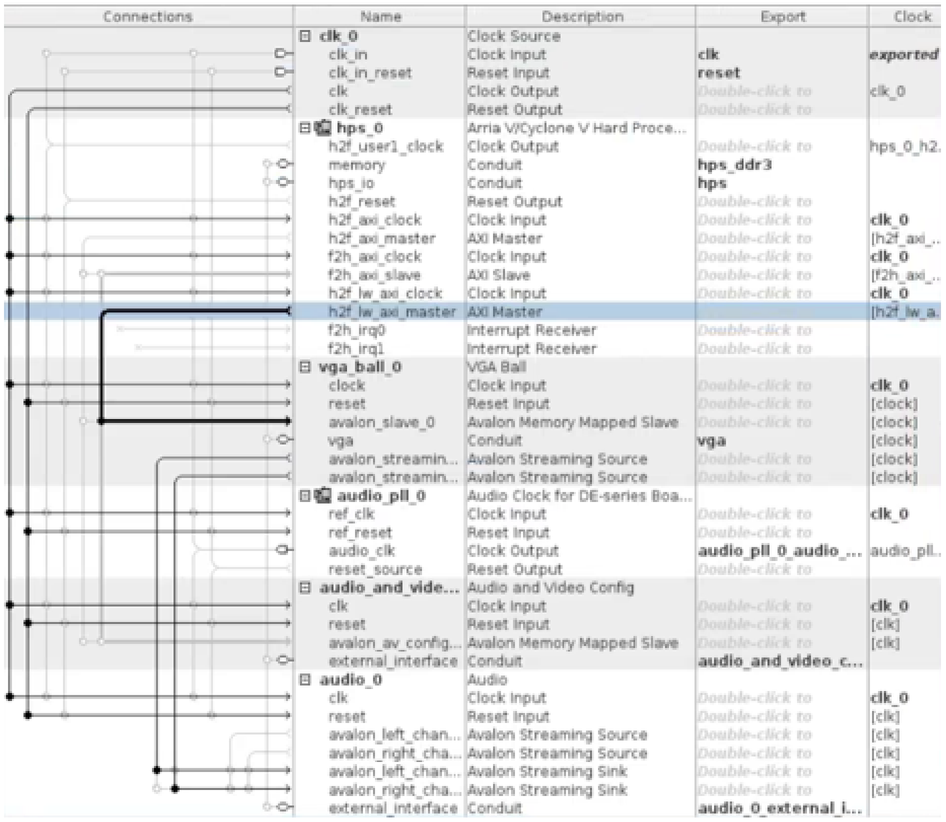


Figure 1: System Diagram

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

The software communicates with the FPGA hardware through the ‘vga.c‘ device driver. This driver uses the Avalon bus interface to relay control signals to the FPGA, which then updates the graphics on the VGA monitor. The FPGA hardware setup includes on-chip memory ROMs for storing sprite data necessary for the game’s graphics. The ’vga display.sv’ retrieves this sprite data based on the game’s logic and outputs it to the VGA monitor, ensuring that the graphics displayed are accurate and timely. Additionally, the audio cues are managed through the WM8731 CODEC, which outputs sound to connected earbuds or speakers.

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



Connections	Name	Description	Export	Clock
	⊟ **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		[h2f_lw_a.
	f2h_irq0	Interrupt Receiver	*Double-click to*	
	f2h_irq1	Interrupt Receiver	*Double-click to*	
	⊟ **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]
	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...**	

Figure 2: qsys-connection

In detail, the qsys connection is shown in Fig.2. The audio_0 is a default Intel FPGA IP core to feed data to codec and the task of the vga_ball is to give the correct data to audio_0 through the streaming source signal. The streaming source signal outputs a valid bit and a 16-bit data signal and its input is a ready signal. The audio_pll part is an Intel IP Core which outputs a 12.288MHz clock signal with 50MHz clock as reference clock. This clock is used to drive the codec. The audio_and_video_config is an Intel IP core which includes several IP signal to configure the codec, including telling the codec the data width, frequency and encoding. The vga_ball module will also be in charge of the video output.

2 Hardware

2.1 Graphics

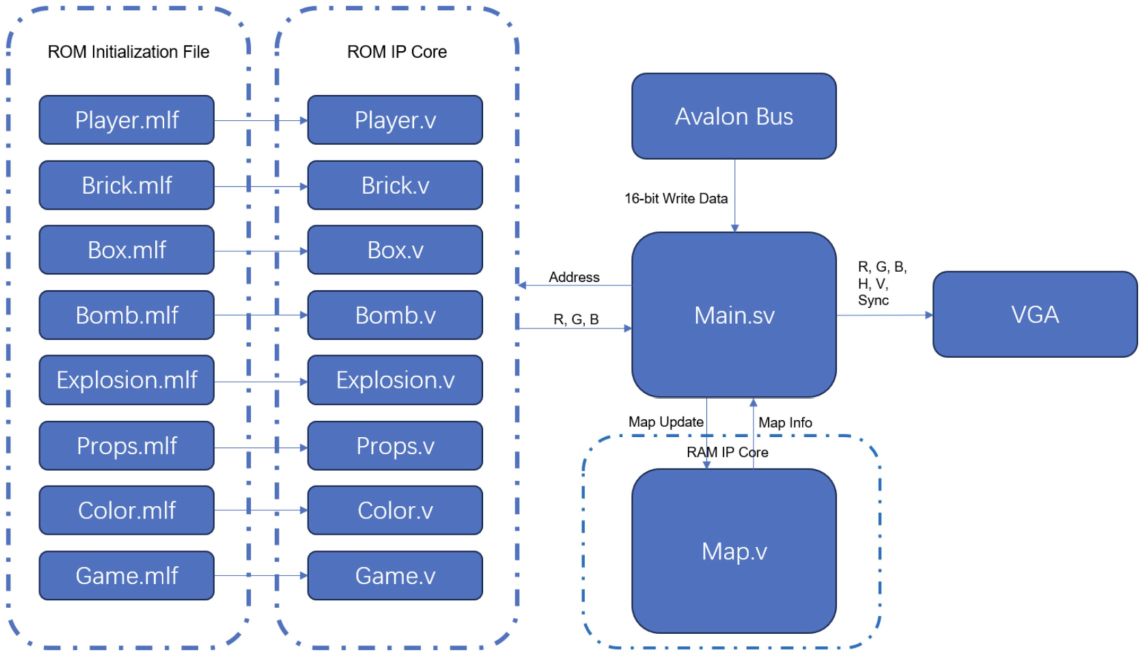


Figure 3: VGA Module Diagram

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

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

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

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

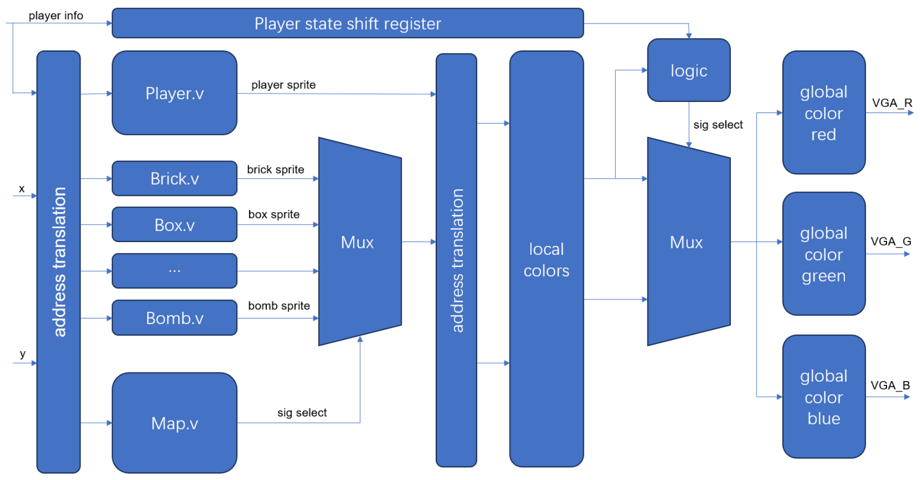


Figure 4: Detailed Connection

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

2.2 Memory

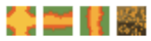
Category	Graphic	Size(bits)	Num Images	Total(bits)
Box		16*16	1	256
Stone		16*16	1	256
Map	N/A	40*30	1	1200
Player (x2)		16*16	7	3584
Bomb		16*16	1	256
Explosion		16*16	4	1024
Props		16*16	3	768
Audio	N/A	12109*16	1	193744
Total				201,088

Figure 5: Memory Table

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

2.3 Audio

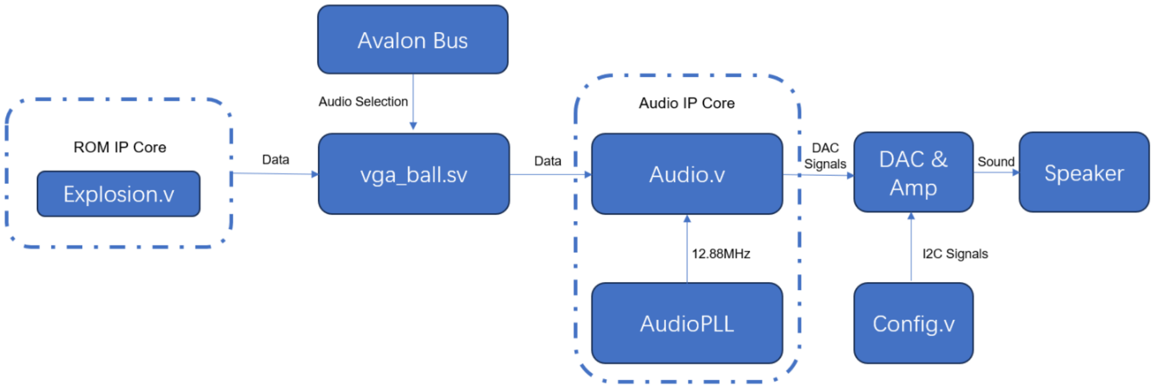


Figure 6: Audio

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

3 Software

3.1 User Input

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

3.2 Game Logic

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

Figure 7: Controller

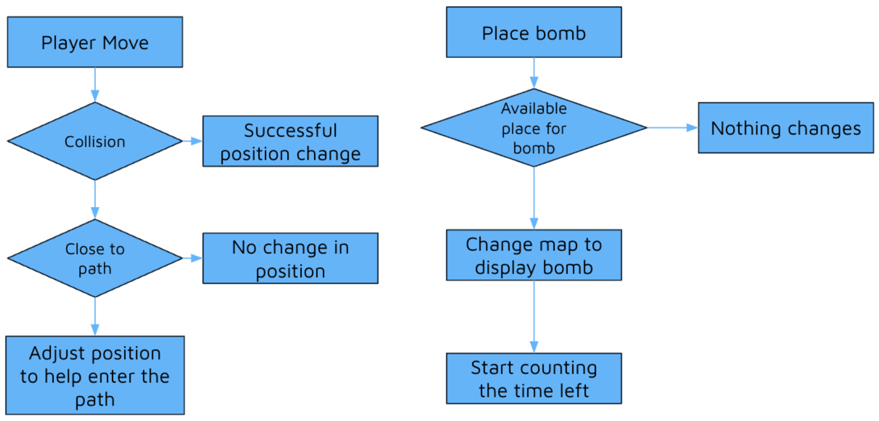

Player Move
Collision
Successful position change
Close to path
No change in position
Adjust position to help enter the path
Place bomb
Available place for bomb
Nothing changes
Change map to display bomb
Start counting the time left

Figure 8: Game Logic

3.2.1 Player Movement

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

3.2.2 Bomb Placement and Explosion

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

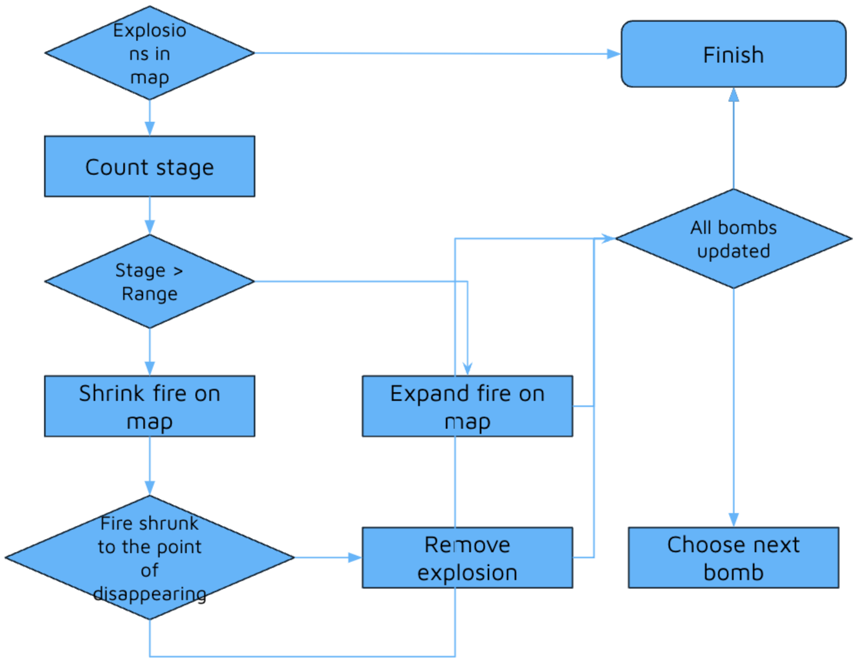


Figure 9: Explosion Game Logic

3.2.3 Power-Ups

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

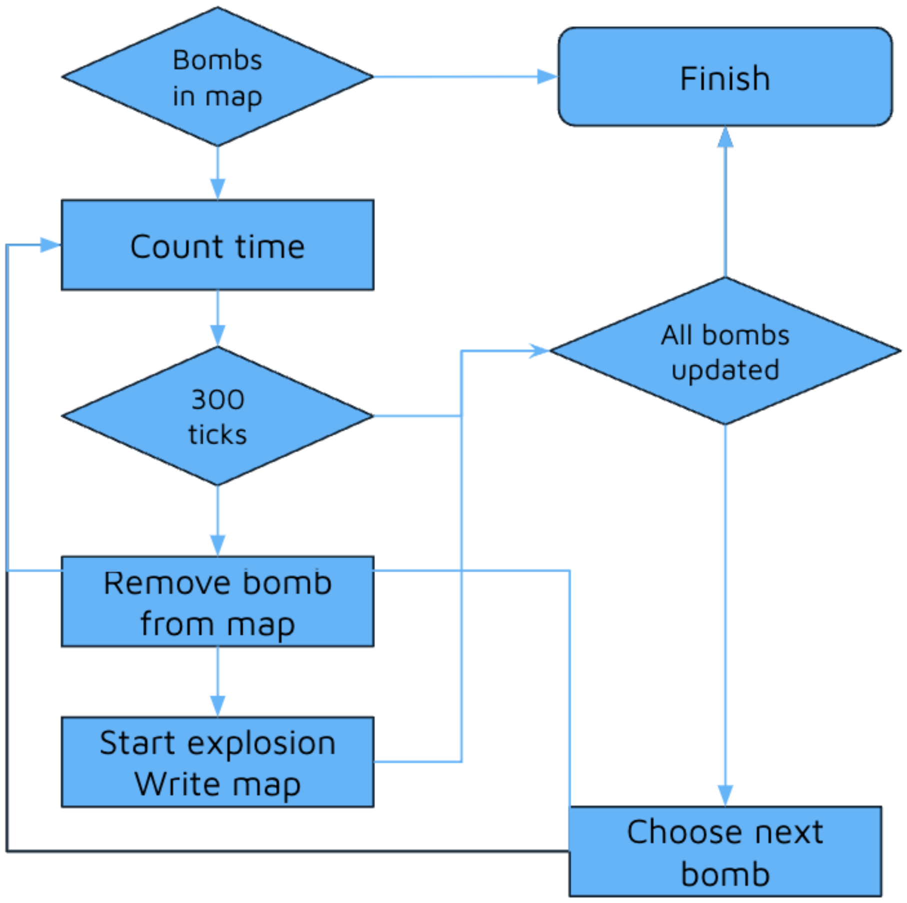


Figure 10: Bomb Game Logic

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

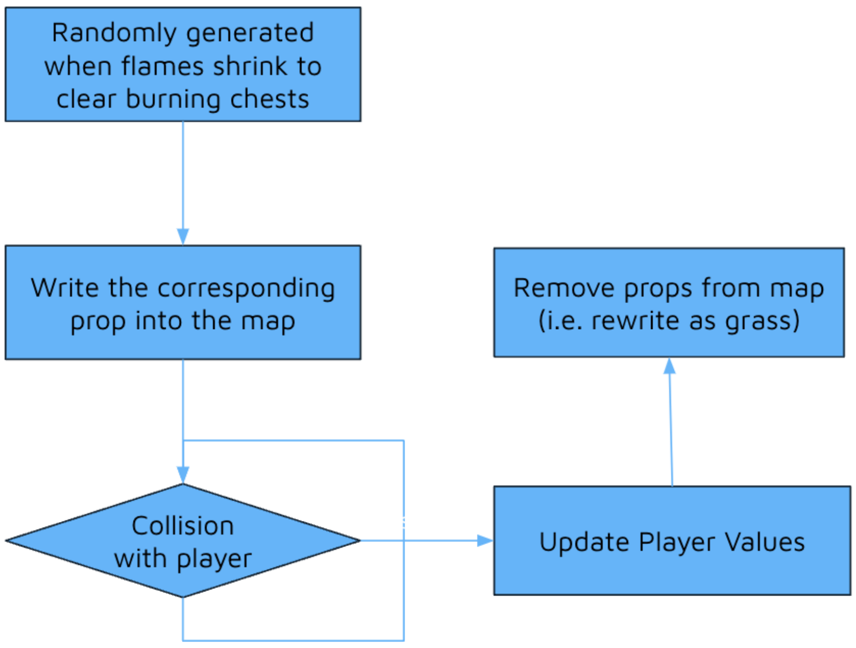

Randomly generated when flames shrink to clear burning chests
Write the corresponding prop into the map
Remove props from map (i.e. rewrite as grass)
Collision with player
Update Player Values

Figure 11: Power-Up Logic

3.2.4 Collision Detection

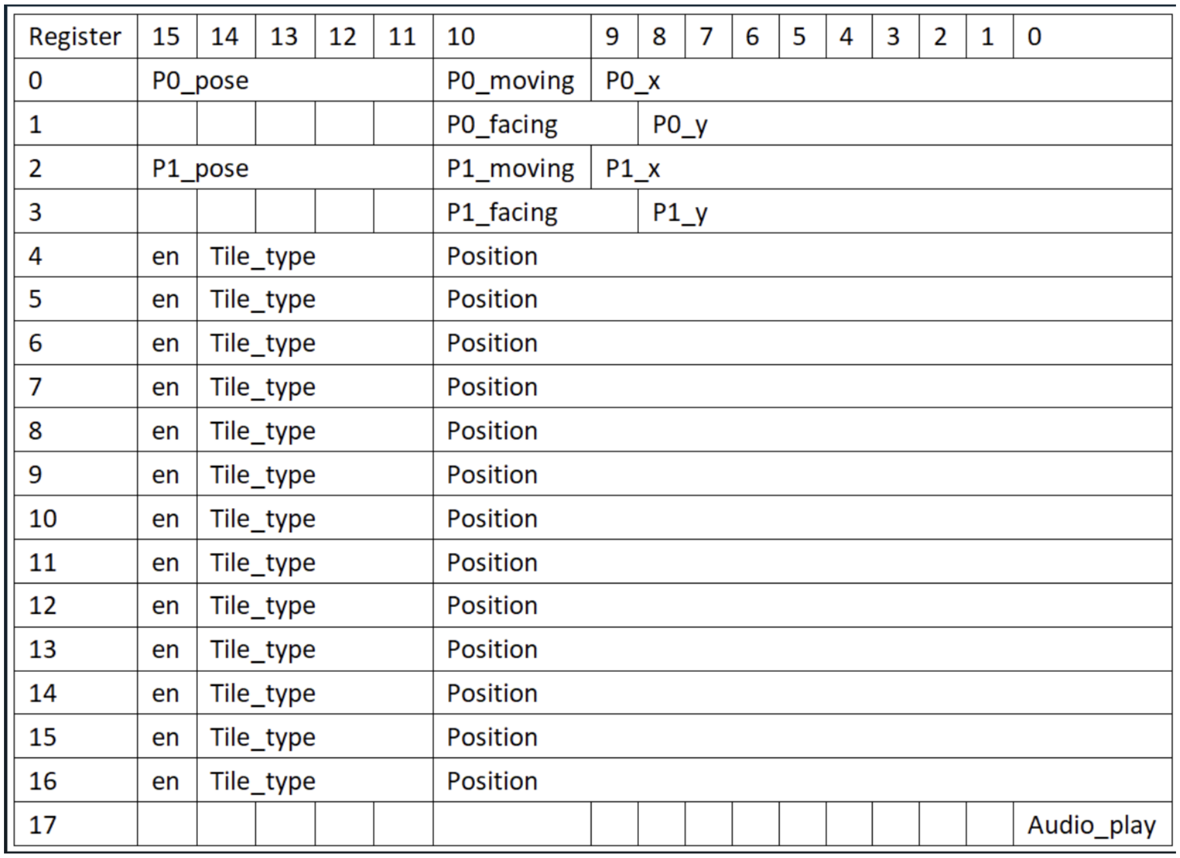
To determine collisions between a player and a bomb explosion, it calculates the grid coordinates of the corners of the explosion based on the position of the player, then checks the tiles at those coordinates on the map. If those coordinates match up, then there has been a player collision with an explosion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

3.2.5 End Game

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

3.3 Hardware-Software Interfacing

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



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

Figure 12: Register Interface

Player Registers (0-3): These registers manage the state, position, and movement details of two players. For each player, there are fields to track their pose, movement status, X and Y coordinates, and facing direction.
Tile Registers (4-16): These registers are used to handle environmental tiles within the game. Each tile has an enable flag, a type, and a position which collectively describe its characteristics and location in the game world.
Audio Register (17): This register includes a control bit to manage audio playback within the system.

4 Discussion

4.1 Challenges

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

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

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

4.2 Lessons Learned

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

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

4.3 Who did what section?

Qian Zhao: Hardware, part of the control and game logic.
Shiyan Wang: Collision detection and path fitting, debugging.
Natalie Hughes: Controller logic, player movement logic and documentation.

5 References

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

A vga_ball.sv

```
   /*
 * Avalon memory-mapped peripheral that generates VGA
 *
 * Stephen A. Edwards
 * Columbia University
 */

module vga_ball(input logic    clk,
           input logic      reset,
       input logic [15:0] writedata,
       input logic      write,
       input         chipselect,
       input logic [5:0] address,

       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,
       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
       );
   logic [7:0] local_pixel;
   logic [7:0] idle_down_sprite_raw;
   logic [7:0] background_sprite_raw;
   logic [7:0] stone_sprite_raw;
   logic [7:0] box_sprite_raw;
   logic [7:0] bomb_sprite_raw;
   logic [7:0] flamed_box_sprite_raw;
   logic [7:0] flame_middle_sprite_raw;
   logic [7:0] flame_h_sprite_raw;
   logic [7:0] flame_v_sprite_raw;
   logic [7:0] pickup_bi_sprite_raw;
   logic [7:0] pickup_fi_sprite_raw;
   logic [7:0] pickup_su_sprite_raw;



   logic [7:0] map_sig_select_raw;
   logic [3:0] map_sig_select;

   logic [1:0] background_sprite;
   logic [1:0] stone_sprite;
   logic [1:0] box_sprite;
   logic [1:0] bomb_sprite;
   logic [1:0] flamed_box_sprite;
   logic [1:0] flame_middle_sprite;
   logic [1:0] flame_h_sprite;
   logic [1:0] flame_v_sprite;
   logic [1:0] pickup_bi_sprite;
   logic [1:0] pickup_fi_sprite;
```

```
    logic [1:0] pickup_su_sprite;


    logic [1:0] map_sprite;
    logic [10:0] hcount;
    logic [9:0] vcount;
    logic [1:0] sprite_offset_temp; //Decides which 2 bits in the 1 byte sprite_info
    logic [1:0] sprite_offset;
    logic [7:0] map_local_color_addr;
    logic [7:0] global_color_addr;
    logic [10:0] map_read_addr;

    /*Registers for player0*/
    logic [15:0] player_info_00; // moving[10] + xinfo[9:0]
    logic [15:0] player_info_01; // facing[10:9] + yinfo[8:0]
    /*Registers for player1*/
    logic [15:0] player_info_10; // same format
    logic [15:0] player_info_11;
    logic [15:0] map_change_0;
    logic [15:0] map_change_1;
    logic [15:0] map_change_2;
    logic [15:0] map_change_3;
    logic [15:0] map_change_4;
    logic [15:0] map_change_5;
    logic [15:0] map_change_6;
    logic [15:0] map_change_7;
    logic [15:0] map_change_8;
    logic [15:0] map_change_9;
    logic [15:0] map_change_10;
    logic [15:0] map_change_11;
    logic [15:0] audio_play_0;


    /*The variable to determine whether the pixel belongs to player 0 and
        corresponding shift register*/
    logic is_player_0;
    logic is_player_1;
    logic is_player;
    logic [3:0] is_player_sr;

    /*Whether the player is moving*/
    logic moving;

    /*shift register for facing*/
    logic [1:0] facing;

    logic [10:0] player_addr_player;
    logic [10:0] player_addr_moving;
    logic [10:0] player_addr_facing;
    logic [10:0] player_addr_sprite;
    logic [10:0] player_addr_local;
    logic [10:0] player_addr;

    logic [7:0] player_sprite_raw;
    logic [1:0] player_sprite;
    logic [3:0] player_offset_sr;
    logic [7:0] player_color_addr;
    logic [15:0] player_color_addr_sr;
    logic [7:0] player_local_color_addr;

    logic [7:0] upper_layer_local_color_addr;
    logic [7:0] lower_layer_local_color_addr;
    logic [7:0] upper_layer_global_color_addr;
    logic [7:0] lower_layer_global_color_addr;

    logic map_write;
    logic [10:0] map_write_addr;
    logic [7:0] map_write_data;

    logic [13:0] audio_explosion_addr;
    logic [13:0] audio_divider;
    logic [15:0] audio_out;
    logic [1:0] audio_valid_sr;
    logic audio_start;




    parameter PLAYER_OFFSET = 11'd640,
              MOVING_OFFSET = 11'd192,
              IDLE_SIDE_OFFSET = 11'd64,
              IDLE_UP_OFFSET = 11'd128,
              MOVING_SIDE_OFFSET = 11'd128,
              MOVING_UP_OFFSET = 11'd320,
              PLAYER_COLOR_OFFSET = 8'd44;

    assign R_DATA = audio_out;
    assign L_DATA = audio_out;
    assign L_VALID = audio_valid_sr[1];
    assign R_VALID = audio_valid_sr[1];


    assign player_addr = player_addr_player + player_addr_moving +
        player_addr_facing + player_addr_sprite + player_addr_local[7:2];


    assign is_player_0 = (player_info_00[9:0] != 10'd0) && (hcount[10:1] >=
        (player_info_00[9:0] - 10'd7)) && (hcount[10:1] <= (player_info_00[9:0] +
        10'd8))
                        && (vcount[8:0] >= (player_info_01[8:0] - 9'd7)) &&
                            (vcount[8:0] <= (player_info_01[8:0] + 9'd8));

    assign is_player_1 = (player_info_10[9:0] != 10'd0) && (hcount[10:1] >=
        (player_info_10[9:0] - 10'd7)) && (hcount[10:1] <= (player_info_10[9:0] +
        10'd8))
                        && (vcount[8:0] >= (player_info_11[8:0] - 9'd7)) &&
                            (vcount[8:0] <= (player_info_11[8:0] + 9'd8));

    assign is_player = is_player_0 || is_player_1;
    assign upper_layer_local_color_addr[7:0] = player_local_color_addr[7:0];
    assign lower_layer_local_color_addr[7:0] = map_local_color_addr[7:0];


    assign map_write_data[7:4] = 4'd0;


    vga_counters count(.clk50(clk),
                       .hcount(hcount),
                       .vcount(vcount),
                       .VGA_CLK(VGA_CLK),
                       .VGA_HS(VGA_HS),
                       .VGA_VS(VGA_VS),
                       .VGA_BLANK_n(VGA_BLANK_n),
                       .VGA_SYNC_n(VGA_SYNC_n));

    //Decide which pixel to look at
    assign local_pixel[7:4] = vcount[3:0];
    assign local_pixel[3:0] = hcount[4:1];

    ram_empty_map empty_map_ram(.clock(clk),
                    .data(map_write_data),
                    .rdaddress(map_read_addr),
                    .wraddress(map_write_addr),
                    .wren(map_write),
                    .q(map_sig_select_raw));



    rom_players players_rom(.address(player_addr),
                              .clock(clk),
                              .q(player_sprite_raw));

    rom_background background_rom(.address(local_pixel[7:2]),
                                  .clock(clk),
                                  .q(background_sprite_raw));

    rom_stone stone_rom(.address(local_pixel[7:2]),
                        .clock(clk),
                        .q(stone_sprite_raw));

    rom_box box_rom(.address(local_pixel[7:2]),
                    .clock(clk),
                    .q(box_sprite_raw));

    rom_bomb bomb_rom(.address(local_pixel[7:2]),
                      .clock(clk),
                      .q(bomb_sprite_raw));

    rom_flamed_box flamed_box_rom(.address(local_pixel[7:2]),
                                    .clock(clk),
                                    .q(flamed_box_sprite_raw));

    rom_flame_middle flame_middle_rom(.address(local_pixel[7:2]),
                                        .clock(clk),
                                        .q(flame_middle_sprite_raw));

    rom_flame_h flame_h_rom(.address(local_pixel[7:2]),
                              .clock(clk),
                              .q(flame_h_sprite_raw));

    rom_flame_v flame_v_rom(.address(local_pixel[7:2]),
                              .clock(clk),
                              .q(flame_v_sprite_raw));

    rom_pickup_bi pickup_bi_rom(.address(local_pixel[7:2]),
                                  .clock(clk),
                                  .q(pickup_bi_sprite_raw));


    rom_pickup_fi pickup_fi_rom(.address(local_pixel[7:2]),
                                  .clock(clk),
                                  .q(pickup_fi_sprite_raw));

    rom_pickup_su pickup_su_rom(.address(local_pixel[7:2]),
                                  .clock(clk),
                                  .q(pickup_su_sprite_raw));

    audio_tank_explosion
        audio_explosion_rom(.address(audio_explosion_addr),.clock(clk),.q(audio_out));

    assign map_sig_select[3:0] = map_sig_select_raw[3:0];

    always_comb begin
        case (sprite_offset)
            2'd0 : begin
                background_sprite[1:0] = background_sprite_raw[1:0];
                stone_sprite[1:0] = stone_sprite_raw[1:0];
                box_sprite[1:0] = box_sprite_raw[1:0];
                bomb_sprite[1:0] = bomb_sprite_raw[1:0];
                flamed_box_sprite[1:0] = flamed_box_sprite_raw[1:0];
                flame_middle_sprite[1:0] = flame_middle_sprite_raw[1:0];
                flame_h_sprite[1:0] = flame_h_sprite_raw[1:0];
                flame_v_sprite[1:0] = flame_v_sprite_raw[1:0];
                pickup_bi_sprite[1:0] = pickup_bi_sprite_raw[1:0];
                pickup_fi_sprite[1:0] = pickup_fi_sprite_raw[1:0];
                pickup_su_sprite[1:0] = pickup_su_sprite_raw[1:0];
            end
            2'd1 : begin
                background_sprite[1:0] = background_sprite_raw[3:2];
                stone_sprite[1:0] = stone_sprite_raw[3:2];
                box_sprite[1:0] = box_sprite_raw[3:2];
                bomb_sprite[1:0] = bomb_sprite_raw[3:2];
                flamed_box_sprite[1:0] = flamed_box_sprite_raw[3:2];
                flame_middle_sprite[1:0] = flame_middle_sprite_raw[3:2];
                flame_h_sprite[1:0] = flame_h_sprite_raw[3:2];
                flame_v_sprite[1:0] = flame_v_sprite_raw[3:2];
                pickup_bi_sprite[1:0] = pickup_bi_sprite_raw[3:2];
                pickup_fi_sprite[1:0] = pickup_fi_sprite_raw[3:2];
                pickup_su_sprite[1:0] = pickup_su_sprite_raw[3:2];
            end
            2'd2 : begin
                background_sprite[1:0] = background_sprite_raw[5:4];
                stone_sprite[1:0] = stone_sprite_raw[5:4];
                box_sprite[1:0] = box_sprite_raw[5:4];
                bomb_sprite[1:0] = bomb_sprite_raw[5:4];
                flamed_box_sprite[1:0] = flamed_box_sprite_raw[5:4];
                flame_middle_sprite[1:0] = flame_middle_sprite_raw[5:4];
                flame_h_sprite[1:0] = flame_h_sprite_raw[5:4];
                flame_v_sprite[1:0] = flame_v_sprite_raw[5:4];
                pickup_bi_sprite[1:0] = pickup_bi_sprite_raw[5:4];
                pickup_fi_sprite[1:0] = pickup_fi_sprite_raw[5:4];
                pickup_su_sprite[1:0] = pickup_su_sprite_raw[5:4];
            end
            2'd3 : begin
                background_sprite[1:0] = background_sprite_raw[7:6];
                stone_sprite[1:0] = stone_sprite_raw[7:6];
                box_sprite[1:0] = box_sprite_raw[7:6];
                bomb_sprite[1:0] = bomb_sprite_raw[7:6];
                flamed_box_sprite[1:0] = flamed_box_sprite_raw[7:6];
                flame_middle_sprite[1:0] = flame_middle_sprite_raw[7:6];
                flame_h_sprite[1:0] = flame_h_sprite_raw[7:6];
                flame_v_sprite[1:0] = flame_v_sprite_raw[7:6];
                pickup_bi_sprite[1:0] = pickup_bi_sprite_raw[7:6];
                pickup_fi_sprite[1:0] = pickup_fi_sprite_raw[7:6];
                pickup_su_sprite[1:0] = pickup_su_sprite_raw[7:6];
            end
        endcase
        map_read_addr = (vcount[9:4] * 40 + hcount[10:5]);
        player_sprite = (player_sprite_raw >> (2 * player_offset_sr[3:2])) % 4;
        player_color_addr = player_addr[10:6] * 4 + PLAYER_COLOR_OFFSET;
        player_local_color_addr[7:0] = player_color_addr_sr[15:8] + player_sprite;

        case (map_sig_select)
            4'd0 : map_sprite = background_sprite;
            4'd1 : map_sprite = stone_sprite;
            4'd2 : map_sprite = box_sprite;
            4'd3 : map_sprite = bomb_sprite;
            4'd4 : map_sprite = flamed_box_sprite;
            4'd5 : map_sprite = flame_middle_sprite;
            4'd6 : map_sprite = flame_h_sprite;
            4'd7 : map_sprite = flame_v_sprite;
            4'd8 : map_sprite = pickup_bi_sprite;
            4'd9 : map_sprite = pickup_fi_sprite;
            4'd10 : map_sprite = pickup_su_sprite;
            default : map_sprite = background_sprite;
        endcase
        map_local_color_addr = map_sig_select * 4 + map_sprite;
        if (is_player_1) begin
            player_addr_player = PLAYER_OFFSET;
            facing = player_info_11[10:9];
            moving = player_info_10[10];
            player_addr_sprite = player_info_10 [12:11] * 64;
            if (facing == 2'd1)
                player_addr_local = (vcount - player_info_11[8:0] + 7) * 16 + 15 -
                    (hcount[10:1] - player_info_10[9:0] + 7);
            else
                player_addr_local = (vcount - player_info_11[8:0] + 7) * 16 +
                    (hcount[10:1] - player_info_10[9:0] + 7);
        end else if (is_player_0) begin
            player_addr_player = 11'd0;
            facing = player_info_01[10:9];
            moving = player_info_00[10];
            player_addr_sprite = player_info_00 [12:11] * 64;
            if (facing == 2'd1)
                player_addr_local = (vcount - player_info_01[8:0] + 7) * 16 + 15 - (hcount[10:1] - player_info_00[9:0] + 7);
            else
                player_addr_local = (vcount - player_info_01[8:0] + 7) * 16 + (hcount[10:1] - player_info_00[9:0] + 7);
        end else begin
            player_addr_player = 11'd0;
            facing = 2'd0;
            moving = 1'd0;
            player_addr_sprite = 11'd0;
            player_addr_local = 11'd0;
        end
        if (moving) begin
            player_addr_moving = MOVING_OFFSET;
            case (facing)
                2'd0 : player_addr_facing = 11'd0;
                2'd1 : player_addr_facing = MOVING_SIDE_OFFSET;
                2'd2 : player_addr_facing = MOVING_UP_OFFSET;
                2'd3 : player_addr_facing = MOVING_SIDE_OFFSET;
            endcase
        end else begin
            player_addr_moving = 11'd0;
            case (facing)
                2'd0 : player_addr_facing = 11'd0;
                2'd1 : player_addr_facing = IDLE_SIDE_OFFSET;
                2'd2 : player_addr_facing = IDLE_UP_OFFSET;
                2'd3 : player_addr_facing = IDLE_SIDE_OFFSET;
            endcase
        end
        if (is_player_sr[3]) begin
            if (upper_layer_global_color_addr == 8'd0)
                global_color_addr = lower_layer_global_color_addr;
            else
                global_color_addr = upper_layer_global_color_addr;
        end else begin
            global_color_addr = lower_layer_global_color_addr;
        end
        case(address)
            5'd4 : begin
                map_write_addr[10:0] = map_change_0[10:0];
                map_write_data[3:0] = map_change_0[14:11];
                map_write = map_change_0[15];
            end
            5'd5 : begin
                map_write_addr[10:0] = map_change_1[10:0];
                map_write_data[3:0] = map_change_1[14:11];
                map_write = map_change_1[15];
            end
            5'd6 : begin
                map_write_addr[10:0] = map_change_2[10:0];
                map_write_data[3:0] = map_change_2[14:11];
                map_write = map_change_2[15];
            end
            5'd7 : begin
                map_write_addr[10:0] = map_change_3[10:0];
                map_write_data[3:0] = map_change_3[14:11];
                map_write = map_change_3[15];
            end
            5'd8 : begin
                map_write_addr[10:0] = map_change_4[10:0];
                map_write_data[3:0] = map_change_4[14:11];
                map_write = map_change_4[15];
            end
            5'd9 : begin
                map_write_addr[10:0] = map_change_5[10:0];
                map_write_data[3:0] = map_change_5[14:11];
                map_write = map_change_5[15];
            end
            5'd10 : begin
                map_write_addr[10:0] = map_change_6[10:0];
                map_write_data[3:0] = map_change_6[14:11];
                map_write = map_change_6[15];
            end
            5'd11 : begin
                map_write_addr[10:0] = map_change_7[10:0];
                map_write_data[3:0] = map_change_7[14:11];
                map_write = map_change_7[15];
            end
            5'd12 : begin
                map_write_addr[10:0] = map_change_8[10:0];
                map_write_data[3:0] = map_change_8[14:11];
                map_write = map_change_8[15];
            end
            5'd13 : begin
                map_write_addr[10:0] = map_change_9[10:0];
                map_write_data[3:0] = map_change_9[14:11];
                map_write = map_change_9[15];
            end
            5'd14 : begin
                map_write_addr[10:0] = map_change_10[10:0];
                map_write_data[3:0] = map_change_10[14:11];
                map_write = map_change_10[15];
            end
            5'd15 : begin
                map_write_addr[10:0] = map_change_11[10:0];
                map_write_data[3:0] = map_change_11[14:11];
                map_write = map_change_11[15];
            end
            default : begin
                map_write_addr[10:0] = 11'd0;
                map_write_data[3:0] = 4'd0;
                map_write = 0;
            end
        endcase

    end

    rom_local_color local_colors_rom(.address_a(upper_layer_local_color_addr[6:0]),
                                     .address_b(lower_layer_local_color_addr[6:0]),
                                     .clock(clk),
                                     .q_a(upper_layer_global_color_addr),
                                     .q_b(lower_layer_global_color_addr));

    rom_global_color_r global_color_r(.address(global_color_addr[5:0]),
                                      .clock(clk),
                                      .q(VGA_R));

    rom_global_color_g global_color_g(.address(global_color_addr[5:0]),
                                      .clock(clk),
                                      .q(VGA_G));


    rom_global_color_b global_color_b(.address(global_color_addr[5:0]),
                                      .clock(clk),
                                      .q(VGA_B));



    always_ff @(posedge clk) begin
       sprite_offset[1:0] <= sprite_offset_temp[1:0];
       sprite_offset_temp[1:0] <= local_pixel[1:0];
       player_offset_sr[1:0] <= player_addr_local[1:0];
       player_offset_sr[3:2] <= player_offset_sr[1:0];
       player_color_addr_sr[7:0] <= player_color_addr[7:0];
       player_color_addr_sr[15:8] <= player_color_addr_sr[7:0];
       is_player_sr[0] <= is_player;
       is_player_sr[1] <= is_player_sr[0];
       is_player_sr[2] <= is_player_sr[1];
       is_player_sr[3] <= is_player_sr[2];
       audio_valid_sr[1] <= audio_valid_sr[0];

       if (chipselect && write) begin
          case(address)
             5'd0 : player_info_00[15:0] <= writedata[15:0];
             5'd1 : player_info_01[15:0] <= writedata[15:0];
             5'd2 : player_info_10[15:0] <= writedata[15:0];
             5'd3 : player_info_11[15:0] <= writedata[15:0];
             5'd4 : map_change_0 [15:0] <= writedata[15:0];
             5'd5 : map_change_1 [15:0] <= writedata[15:0];
             5'd6 : map_change_2 [15:0] <= writedata[15:0];
             5'd7 : map_change_3 [15:0] <= writedata[15:0];
             5'd8 : map_change_4 [15:0] <= writedata[15:0];
             5'd9 : map_change_5 [15:0] <= writedata[15:0];
             5'd10 : map_change_6 [15:0] <= writedata[15:0];
             5'd11 : map_change_7 [15:0] <= writedata[15:0];
             5'd12 : map_change_8 [15:0] <= writedata[15:0];
             5'd13 : map_change_9 [15:0] <= writedata[15:0];
             5'd14 : map_change_10 [15:0] <= writedata[15:0];
             5'd15 : map_change_11 [15:0] <= writedata[15:0];
             5'd16 : audio_play_0 [15:0] <= writedata[15:0];
          endcase
       end
```

```

      if (L_READY && R_READY) begin
         if (audio_play_0 == 16'd1) begin
            audio_start <= 1'd1;
            audio_explosion_addr <= 14'd0;
            audio_valid_sr[0] <= 1'd0;
         end
         else if (audio_divider == 14'b0 && audio_start == 1'd1) begin
            if (audio_explosion_addr == 14'd12109) begin
               audio_explosion_addr <= 14'd0;
               audio_start <= 1'd0;
            end
            else begin
               audio_explosion_addr <= audio_explosion_addr + 14'd1;
               audio_start <= 1'd1;
            end
            audio_valid_sr[0] <= 1;
         end
         else
            audio_valid_sr[0] <= 0;

         if (audio_divider == 14'd3125)
            audio_divider <= 14'd0;
         else
            audio_divider <= audio_divider + 14'd1;
      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;
    logic [10:0] effective_hcount;
    logic [9:0] effective_vcount;

    count_sr sr_count_ram(.clock(clk50),
                 .shiftin({hcount[10:0], vcount[9:0]}),
                 .shiftout({effective_hcount[10:0], effective_vcount[9:0]}));

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

   assign endOfLine = hcount == HTOTAL - 1;

   logic endOfField;

   always_ff @(posedge clk50)
    if (endOfLine)
        if (endOfField)
            vcount <= 0;
        else
            vcount <= vcount + 10'd 1;

    assign endOfField = vcount == VTOTAL - 1;
    assign VGA_CLK = hcount[0];
    assign VGA_HS = !( (effective_hcount[10:8] == 3'b101) &
              !(effective_hcount[7:5] == 3'b111));
   assign VGA_VS = !( effective_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 = !( effective_hcount[10] & (effective_hcount[9] |
       effective_hcount[8]) ) &
           !( effective_vcount[9] | (effective_vcount[8:5] == 4'b1111) );

endmodule
```

B vga.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 "hello.h"

#define DRIVER_NAME "vga_ball"

/* Device registers */
#define PLAYER1X(x) (x)
#define PLAYER1Y(x) ((x)+2)
#define TANK1_P(x) ((x)+4)
#define TANK1_D(x) ((x)+6)
#define ENEMY1_P(x) ((x)+8)
#define ENEMY1_D(x) ((x)+10)
#define ENEMY2_P(x) ((x)+12)
#define ENEMY2_D(x) ((x)+14)
#define ENEMY3_P(x) ((x)+16)
#define ENEMY3_D(x) ((x)+18)
#define ENEMY4_P(x) ((x)+20)
#define ENEMY4_D(x) ((x)+22)
#define BULLET1(x) ((x)+24)
#define BULLET2(x) ((x)+26)
#define BULLET3(x) ((x)+28)
#define BULLET4(x) ((x)+30)
#define BULLET5(x) ((x)+32)
```

```c
#define RESERVE4(x) ((x)+34)
/*
 * Information about our device
 */
struct vga_ball_dev {
    struct resource res; /* Resource: our registers */
    void __iomem *virtbase; /* Where registers can be accessed in memory */
        game_info_t background;
} dev;

/*
 * Write segments of a single digit
 * Assumes digit is in range and the device information has been set up
 */
static void write_background(game_info_t *background)
{
    iowrite16(background->playerinfo00, PLAYER1X(dev.virtbase) ); //display player
    iowrite16(background->playerinfo01, PLAYER1Y(dev.virtbase) );
    iowrite16(background->playerinfo10, TANK1_P(dev.virtbase) );
    iowrite16(background->playerinfo11, TANK1_D(dev.virtbase) );
    iowrite16(background->map_change_0, ENEMY1_P(dev.virtbase) );
    iowrite16(background->map_change_1, ENEMY1_D(dev.virtbase) );
    iowrite16(background->map_change_2, ENEMY2_P(dev.virtbase) );
    iowrite16(background->map_change_3, ENEMY2_D(dev.virtbase) );
    iowrite16(background->map_change_4, ENEMY3_P(dev.virtbase) );
    iowrite16(background->map_change_5, ENEMY3_D(dev.virtbase) );
    iowrite16(background->map_change_6, ENEMY4_P(dev.virtbase) );
    iowrite16(background->map_change_7, ENEMY4_D(dev.virtbase) );
    iowrite16(background->map_change_8, BULLET1(dev.virtbase) );
    iowrite16(background->map_change_9, BULLET2(dev.virtbase) );
    iowrite16(background->map_change_10, BULLET3(dev.virtbase) );
    iowrite16(background->map_change_11, BULLET4(dev.virtbase) );
    iowrite16(background->reserved_0, BULLET5(dev.virtbase) );
    iowrite16(background->reserved_1, RESERVE4(dev.virtbase) );


    //iowrite16(0x7320, TANK1_D(dev.virtbase) );
    dev.background = *background;
}

/*
 * 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;

    switch (cmd) {
    case VGA_BALL_WRITE_BACKGROUND:
        if (copy_from_user(&vla, (vga_ball_arg_t *) arg,
                  sizeof(vga_ball_arg_t)))
            return -EACCES;
        write_background(&vla.background);
        break;

    case VGA_BALL_READ_BACKGROUND:
        vla.background = dev.background;
        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)
{
        game_info_t beige = {0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
                0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
                0x0000,0x0000};
    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;
    }

    /* Set an initial color */
        write_background(&beige);


    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 */
```

```c
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 hello.h

```c
    #ifndef _VGA_BALL_H
#define _VGA_BALL_H

#include <linux/ioctl.h>

typedef struct {
    unsigned short playerinfo00, playerinfo01, playerinfo10, playerinfo11,
    map_change_0, map_change_1, map_change_2, map_change_3, map_change_4,
        map_change_5, map_change_6,map_change_7, map_change_8, map_change_9,
        map_change_10, map_change_11, reserved_0, reserved_1;
} game_info_t;


typedef struct {
    game_info_t background;
} vga_ball_arg_t;

#define VGA_BALL_MAGIC 'q'

/* ioctls and their arguments */
#define VGA_BALL_WRITE_BACKGROUND _IOW(VGA_BALL_MAGIC, 1, vga_ball_arg_t *)
#define VGA_BALL_READ_BACKGROUND _IOR(VGA_BALL_MAGIC, 2, vga_ball_arg_t *)

#endif
```

D controller.h

```c
    #ifndef _CONTROLLER_H
#define _CONTROLLER_H

#include <stdio.h>
#include <stdlib.h>
#include <libusb-1.0/libusb.h>
#include "bomberman.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 codes[7];

};

struct args_list {

       struct controller_list devices;
       char buttons[11];
       int mode;
       int print;
       control_info_t *control_info;
};

struct controller_list open_controllers();
void *listen_controllers(void *arg);

#endif
```

E bomberman.h

```
    #ifndef _BOMBERMAN_H
#define _BOMBERMAN_H
#include <stdint.h>
#include "hello.h"
#define PLAYER_X_UPPER_LIM (uint16_t) 631
#define PLAYER_X_LOWER_LIM (uint16_t) 8
#define PLAYER_Y_UPPER_LIM (uint16_t) 471
#define PLAYER_Y_LOWER_LIM (uint16_t) 8
#define SET_BIT(x,y) x |= ((uint16_t) 1 << y)
#define SET_BITS(x,y,z) x |= ((uint16_t) z << y)
#define CLEAR_BIT(x,y) x &= ~((uint16_t) 1 << y)
#define MAP_SIZE 1200
#define PLAYER0_INIT_X (uint16_t) 168
#define PLAYER0_INIT_Y (uint16_t) 167
#define PLAYER1_INIT_X (uint16_t) 488
#define PLAYER1_INIT_Y (uint16_t) 327
#define abs_diff(x, y) ((x < y) ? (y - x) : (x - y))


enum FACING {DOWN, LEFT, UP, RIGHT};
enum STAT {STATIC, MOVING};
enum POSE {IDLE, SIDE0, SIDE1, SIDE2, DOWN0, DOWN1, UP0, UP1};
enum PLAYER {PLAYER0, PLAYER1};

```

```c
typedef struct {
    uint16_t vxpos;
    uint16_t vypos;
    enum FACING facing;
    enum STAT status;
    enum POSE pose;
    uint16_t pos_tick;
    uint16_t vspeed;
    uint16_t bomb_range;
    uint16_t max_bombs;
    uint16_t bombs_left;
    uint16_t bomb_colddown;
    uint16_t dead;
} player_info_t;

typedef struct {
    enum FACING direction0;
    enum FACING direction1;
    unsigned long long press_tick0;
    unsigned long long press_tick1;
    int attempt_place_bomb_0;
    int attempt_place_bomb_1;
    int idle0;
    int idle1;
} control_info_t;

struct Bomb {
    int time_left;
    enum PLAYER player;
    uint16_t pos;
    struct Bomb *next;
    uint16_t range;
};

struct Explosion {
    int stage;
    uint16_t range;
    int ends[4]; /*Down, Left, Up, Right*/
    int stage_time_left;
    struct Explosion *next;
};
void init_explosion_sound(void);
void set_player_pos(uint16_t *pos, player_info_t *info);
void display_game_over(void);
void set_player_status (uint16_t vxpos, uint16_t vypos, enum FACING facing, enum
    STAT status, enum POSE pose, uint16_t pos_tick, uint16_t vspeed, uint16_t
    bomb_range, uint16_t max_bombs, uint16_t bombs_left, uint16_t bomb_colddown,
    uint16_t dead, enum PLAYER player);
void write_player_info(void);
void pass_game_info(void);
void generate_software_map(void);
void init_players(void);
int is_player_moving(player_info_t *info);
enum FACING get_player_facing(player_info_t *info);
void get_player_vpos(uint16_t *pos, player_info_t *info);
uint16_t get_player_vspeed(player_info_t *info);
```

```
void handle_player_movement(void);
void reset_map_change_list(void);
void sync_hw_map_change(void);
void handle_player_place_bomb(void);
void handle_bomb_explode(void);
int map_change_list_append_tile(uint16_t pos, uint16_t tile);
void handle_explosion(void);
void handle_player_prop_get(void);
void free_explosion(void);
void free_bombs(void);
#endif
```

F controller.c

```
    #include "controller.h"
#include <libusb-1.0/libusb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/time.h>
#include "controller.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);
        }

        //printf("Detected %d devices...\n", num_devs);
        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) {
```

```c
                        printf("\nERROR: bad device descriptor.");
                        exit(1);
                }

                // Our controllers have idProduct of 17
                if (desc.idProduct == 0xe401) {

                        //printf("FOUND: idProduct-%d ", desc.idProduct);
                        struct libusb_config_descriptor *config;
                        if ((libusb_get_config_descriptor(dev, 0, &config)) < 0) {
                                printf("\nERROR: bad config descriptor.");
                                exit(1);
                        }
                        //printf("interfaces-%d\n", config->bNumInterfaces);

                        // Our controllers only have a single interface, no need for looping
                        // This interface also only has one .num_altsetting, no need for looping

                        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;
                                //printf("%d:%d,%d:%d\n",devices.device1,devices.device1_addr,devices.device2,d
                                                      goto found;
                        }
                }
        }

        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 pkt1, struct controller_pkt pkt2,
    control_info_t *control_info, int print) {

	// Choose whether you want human-readable or binary output
	if (pkt1.codes[0] == 0x7f && pkt1.codes[1] == 0xff && pkt1.codes[2] == 0x00)
	    { //Player0 press Down direction
		control_info -> idle0 = 0;
		if (control_info -> direction0 == DOWN)
			control_info -> press_tick0 ++;

		else {
			control_info -> direction0 = DOWN;
			control_info -> press_tick0 = 0;
		}
		if (print)
			printf("Player 0: Down\n");
	} else if (pkt1.codes[0] == 0x00 && pkt1.codes[1] == 0x7f && pkt1.codes[2] ==
	    0x00) {
		control_info -> idle0 = 0;
		if (control_info -> direction0 == LEFT)
			control_info -> press_tick0 ++;
		else {
			control_info -> direction0 = LEFT;
			control_info -> press_tick0 = 0;
		}
		if (print)
			printf("Player 0: Left\n");
	} else if (pkt1.codes[0] == 0x7f && pkt1.codes[1] == 0x00 && pkt1.codes[2] ==
	    0x00) {
		control_info -> idle0 = 0;
		if (control_info -> direction0 == UP)
			control_info -> press_tick0 ++;
		else {
			control_info -> direction0 = UP;
			control_info -> press_tick0 = 0;
		}
		if (print)
			printf("Player 0: Up\n");
	} else if (pkt1.codes[0] == 0xff && pkt1.codes[1] == 0x7f && pkt1.codes[2] ==
	    0x00) {
		control_info -> idle0 = 0;
		if (control_info -> direction0 == RIGHT)
			control_info -> press_tick0 ++;
		else {
			control_info -> direction0 = RIGHT;
			control_info -> press_tick0 = 0;
		}
		if (print)
			printf("Player 0: Right\n");
```

```
        } else {
                control_info -> press_tick0 = 0;
                control_info -> idle0 = 1;
        }
        if (pkt2.codes[0] == 0x7f && pkt2.codes[1] == 0xff && pkt2.codes[2] == 0x00)
            { //Player1 press Down direction
                control_info -> idle1 = 0;
                if (control_info -> direction1 == DOWN)
                        control_info -> press_tick1 ++;

                else {
                        control_info -> direction1 = DOWN;
                        control_info -> press_tick1 = 0;
                }
                if (print)
                        printf("Player 1: Down\n");
        } else if (pkt2.codes[0] == 0x00 && pkt2.codes[1] == 0x7f && pkt2.codes[2] ==
            0x00) {
                control_info -> idle1 = 0;
                if (control_info -> direction1 == LEFT)
                        control_info -> press_tick1 ++;
                else {
                        control_info -> direction1 = LEFT;
                        control_info -> press_tick1 = 0;
                }
                if (print)
                        printf("Player 1: Left\n");
        } else if (pkt2.codes[0] == 0x7f && pkt2.codes[1] == 0x00 && pkt2.codes[2] ==
            0x00) {
                control_info -> idle1 = 0;
                if (control_info -> direction1 == UP)
                        control_info -> press_tick1 ++;
                else {
                        control_info -> direction1 = UP;
                        control_info -> press_tick1 = 0;
                }
                if (print)
                        printf("Player 1: Up\n");
        } else if (pkt2.codes[0] == 0xff && pkt2.codes[1] == 0x7f && pkt2.codes[2] ==
            0x00) {
                control_info -> idle1 = 0;
                if (control_info -> direction1 == RIGHT)
                        control_info -> press_tick1 ++;
                else {
                        control_info -> direction1 = RIGHT;
                        control_info -> press_tick1 = 0;
                }
                if (print)
                        printf("Player 1: Right\n");
        } else {
                control_info -> press_tick1 = 0;
                control_info -> idle1 = 1;
        }
        if (pkt1.codes[5] == 0x2f) {
                control_info -> attempt_place_bomb_0 = 1;
                if (print)
```

```c
                        printf("Player 0: Place bomb!\n");
        } else
                control_info -> attempt_place_bomb_0 = 0;
        if (pkt2.codes[5] == 0x2f) {
                control_info -> attempt_place_bomb_1 = 1;
                if (print)
                        printf("Player 1: Place bomb!\n");
        } else
                control_info -> attempt_place_bomb_1 = 0;


}

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[] = "_____";
        struct timeval stop, start;
        unsigned long inteval[10];
        int count = 0;
        gettimeofday(&start, NULL);
        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, pkt2, args.control_info, args.print);

                }

        }
}
```

G bomberman.c

```c
    #include <unistd.h>
#include <stdio.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
```

```c
#include <unistd.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>
#include "controller.h"
#include "bomberman.h"


int vga_fd;
game_info_t global_info;
player_info_t player0_info;
player_info_t player1_info;
control_info_t control_info;
uint16_t *map;
struct controller_list controllers;
struct args_list c_args_list;
short map_change_list[12];
int map_change_list_next;

struct Bomb *bombs_head;
struct Explosion *explosion_head;
void write_tile(uint16_t tile_pos, uint16_t tile_type, unsigned short *pos);
uint16_t p1_win_arr[] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0,
    1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0,
    0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0,
    1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1,
    0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0,
    1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1,
    0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0,
    0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1,
    0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0,
    0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0,
    0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0,
    0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0,
    0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0,
    1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
    0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0,
    0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1,
    1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
uint16_t p2_win_arr[] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0,
    1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0,
    0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0,
    1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1,
    0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0,
    0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1,
    0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0,
    0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1,
    1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0,
    0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0,
    0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0,
    0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0,
    0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0,
    1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
    0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0,
    0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1,
    1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
```

```c
uint16_t tie_arr[] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
    1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
    0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
    0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1,
    1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
    0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
    0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
    0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}; 

void start_explosion_sound(void)
{
    global_info.reserved_0 = (unsigned short) 1;
    return;
}
void init_explosion_sound(void)
{
    global_info.reserved_0 = (unsigned short) 0;
    return;
}
```

```c

int start_explosion(struct Bomb *bomb)
{
    if (map_change_list_next >= 12)
        return -1;
    start_explosion_sound();
    printf("Start explosion at %d\n", bomb->pos);
    map_change_list_append_tile(bomb->pos, 5);
    map[(bomb->pos)] = 5;
    struct Explosion *new = (struct Explosion *) malloc(sizeof(struct Explosion));
    new->stage = 0;
    new->range = bomb->range;
    int pos = bomb -> pos;
    printf("map[%d]=%d\n", pos, map[pos]);
    new->stage_time_left = 50 / ((bomb->range) + 2);
    for (int i = 0; i < 4; i ++)
        new->ends[i] = pos;
    if (explosion_head == NULL){
        new->next = NULL;
        explosion_head = new;
    }
    else{
        new->next = explosion_head;
        explosion_head = new;
    }
    return 0;
}

void free_explosion(void)
{
    struct Explosion *curr;
    while (curr) {
        struct Explosion *temp = curr->next;
        free(temp);
        curr = temp;
    }
}

int is_flame(uint16_t tile) {
    return ((tile >= 5) && (tile <= 7));
}

int is_flame_obstacle(uint16_t tile) {
    return ((tile > 0) && (tile < 4) && (tile != 2));
}


int can_expand_flame(int from, int to) {
    uint16_t from_tile = map[from];
    if (is_flame(from_tile))
        if (!is_flame_obstacle(map[to]))
            return 1;
    return 0;
}
/*Modify ends, hw, map*/
int expand_flame(int ends[])
```

```c
{
    /*Get number of registers needed for the map change*/
    int needs_modify[4]; /*A structure to denote which end needs to be expanded*/
    printf("Start expanding flame, ends = [%d %d %d %d]\n", ends[0], ends[1],
        ends[2], ends[3]);
    for (int i = 0; i < 4; i++) {
        int pos = ends[i];
        printf("map[%d] = %d\n", pos, map[pos]);
        int pos_r = pos / 40;
        int pos_c = pos % 40;
        int to_r;
        int to_c;
        switch (i) {
            case 0:
                to_r = pos_r + 1;
                to_c = pos_c;
                break;
            case 1:
                to_r = pos_r;
                to_c = pos_c - 1;
                break;
            case 2:
                to_r = pos_r - 1;
                to_c = pos_c;
                break;
            case 3:
                to_r = pos_r;
                to_c = pos_c + 1;
                break;
        }
        if ((to_r >= 0 && to_r < 30) && (to_c >= 0 && to_c < 40)) {
            int to_pos = to_r * 40 + to_c;
            if (can_expand_flame(pos, to_pos))
                needs_modify[i] = 1;
            else
                needs_modify[i] = 0;
        } else {
            needs_modify[i] = 0;
        }
    }
    int num_reg_needed = 0;
    for (int i = 0; i < 4; i++)
        num_reg_needed += needs_modify[i];
    if (map_change_list_next + num_reg_needed > 12)
        return -1;
    printf("Needs modify [%d, %d, %d, %d]\n", needs_modify[0], needs_modify[1],
        needs_modify[2], needs_modify[3]);
    for (int i = 0; i < 4; i++) {
        if(needs_modify[i]) {
            int pos = ends[i];
            int pos_r = pos / 40;
            int pos_c = pos % 40;
            int to_r;
            int to_c;
            switch (i) {
                case 0:
                    to_r = pos_r + 1;
                    to_c = pos_c;
                    break;
                case 1:
                    to_r = pos_r;
                    to_c = pos_c - 1;
                    break;
                case 2:
                    to_r = pos_r - 1;
                    to_c = pos_c;
                    break;
                case 3:
                    to_r = pos_r;
                    to_c = pos_c + 1;
                    break;
            }
            uint16_t target_flame;
            if (i % 2 == 0)
                target_flame = 6;
            else
                target_flame = 7;
            int to_pos = to_r * 40 + to_c;
            uint16_t original_tile = map[to_pos];
            uint16_t target_tile;
            if (is_flame(original_tile)) {
                if (target_flame != original_tile)
                    target_tile = 5;
                else {
                    target_tile = target_flame;
                }

            }
            else if (original_tile == 2)
                target_tile = 4;
            else {
                target_tile = target_flame;
            }
            /*Change the hw*/
            map_change_list_append_tile(to_pos, target_tile);
            map[to_pos] = target_tile;
            ends[i] = to_pos;
            printf("Write flame to row %d col %d\n", to_r, to_c);
        }
    }

    return 0;
}

int shrink_flame(int ends[], int center) {
    /*get number of registers to change*/
    printf("Start shrinking flame, ends = [%d, %d, %d, %d]\n", ends[0], ends[1],
         ends[2], ends[3]);
    int needs_modify[4];
    for (int i = 0; i < 4; i ++) {
        if ((is_flame(map[ends[i]]) || map[ends[i]] == 4) && ends[i] != center)
            needs_modify[i] = 1;
        else if (ends[i] != center) {
            int pos = ends[i];
            int pos_r = pos / 40;
            int pos_c = pos % 40;
            int to_r;
            int to_c;
            switch (i) {
                case 0:
                    to_r = pos_r - 1;
                    to_c = pos_c;
                    break;
                case 1:
                    to_r = pos_r;
                    to_c = pos_c + 1;
                    break;
                case 2:
                    to_r = pos_r + 1;
                    to_c = pos_c;
                    break;
                case 3:
                    to_r = pos_r;
                    to_c = pos_c - 1;
                    break;
            }
            int to_pos = to_r * 40 + to_c;
            ends[i] = to_pos;
            needs_modify[i] = 0;
        }
        else {
            needs_modify[i] = 0;
        }
    }
    int num_reg_needed = 0;
    for (int i = 0; i < 4; i ++)
        num_reg_needed += needs_modify[i];
    if (map_change_list_next + num_reg_needed > 12)
        return -1;
    for (int i = 0; i < 4; i ++) {
        if (needs_modify[i]) {
            int pos = ends[i];
            int pos_r = pos / 40;
            int pos_c = pos % 40;
            int to_r;
            int to_c;
            switch (i) {
                case 0:
                    to_r = pos_r - 1;
                    to_c = pos_c;
                    break;
                case 1:
                    to_r = pos_r;
                    to_c = pos_c + 1;
                    break;
                case 2:
                    to_r = pos_r + 1;
                    to_c = pos_c;
                    break;
                case 3:
                    to_r = pos_r;
                    to_c = pos_c - 1;
                    break;
            }
            if (map[ends[i]] != 5) {
                if (map[ends[i]] != 4) {
                    map_change_list_append_tile(ends[i], (uint16_t) 0);
                    map[ends[i]] = 0;
                    printf("delete flame at %d\n", ends[i]);
                }
                else {
                    uint16_t prop_id = rand() % 10;
                    if (prop_id < 3) {
                        map_change_list_append_tile(ends[i], (uint16_t) prop_id + 8);
                        map[ends[i]] = prop_id + 8;
                        printf("place prop %d at %d\n", prop_id, ends[i]);
                    }
                    else {
                        map_change_list_append_tile(ends[i], (uint16_t) 0);
                        map[ends[i]] = 0;
                        printf("delete flame at %d\n", ends[i]);
                    }
                }
            }
            else {
                if((i % 2) == 0) {
                    map_change_list_append_tile(ends[i], (uint16_t) 7);
                    map[ends[i]] = 7;
                }
                else {
                    map_change_list_append_tile(ends[i], (uint16_t) 6);
                    map[ends[i]] = 6;
                }
            }
            int to_pos = to_r * 40 + to_c;
            ends[i] = to_pos;
        }
    }
    return 0;
}

void handle_explosion(void)
{
    struct Explosion *curr = explosion_head;
    struct Explosion *prev = explosion_head;
    while(curr){
        if(curr->stage_time_left <= 0){
            int stage = curr->stage;
            int range = (int) curr->range;
            printf("stage = %d, range = %d\n", stage, range);
            if (stage < range) {
                if (expand_flame(curr->ends)) {
                    printf("Warning: Need enough registers to do the flame
                         expansion!\n");
                    continue;
                }
            }
            else if ((stage > range) && (stage <= range * 2)) {
                int center_c = curr->ends[0] % 40;
                int center_r = curr->ends[1] / 40;
                int center = center_r * 40 + center_c;
                if (shrink_flame(curr->ends, center)) {
                    printf("Warning: Need enough registers to do the flame shrink!\n");
                    continue;
                }
            }
            else if (stage > range * 2){
                int center_c = curr->ends[0] % 40;
                int center_r = curr->ends[1] / 40;
                int center = center_r * 40 + center_c;
                for (int i = 0; i < 4; i++) {
                    if (curr->ends[i] != center)
                        printf("Warning: explosion has not been shrinked to 1\n");
                }
                if (map_change_list_append_tile(center, (uint16_t) 0)) {
                    printf("Warning: Need enough registers to do the flame
                         deletion!\n");
                    continue;
                }
                map[center] = 0;
                if (curr == explosion_head) {
                    explosion_head = curr->next;
                    free(curr);
                    curr = explosion_head;
                    continue;
                }
                else {
                    prev->next = curr->next;
                    free(curr);
                    curr = prev->next;
                    continue;
                }
            }

            curr->stage ++;
            curr->stage_time_left = 50 / (curr->range + 2);
            prev = curr;
            curr = curr->next;
        }
        else {
            curr->stage_time_left --;
            prev = curr;
            curr = curr->next;
        }
    }
}

void insert_bomb(enum PLAYER player, uint16_t pos) {
    struct Bomb *new = (struct Bomb *) malloc(sizeof(struct Bomb));
    new->time_left = 300;
    new->player = player;
    new->pos = pos;
    switch (player) {
        case PLAYER0:
            new->range = player0_info.bomb_range;
            break;
        case PLAYER1:
            new->range = player1_info.bomb_range;
            break;
    }
    if (bombs_head == NULL) {
        bombs_head = new;
        bombs_head->next = NULL;
    }
    else {
        new->next = bombs_head;
        bombs_head = new;
    }
}

void player_get_prop(uint16_t pos, player_info_t *info) {
    if (map[pos] == 8) {
        if (!map_change_list_append_tile(pos, (uint16_t) 0)) {
            map[pos] = 0;
            if (info->max_bombs < 6) {
                info->max_bombs ++;
                info->bombs_left ++;
                info->bomb_colddown = 50;
            }
        }
    }
    else if (map[pos] == 9) {
        if (!map_change_list_append_tile(pos, (uint16_t) 0)) {
            map[pos] = 0;
            if (info->bomb_range < 5) {
                info->bomb_range ++;
            }
        }
    }
    else if (map[pos] == 10) {
        if (!map_change_list_append_tile(pos, (uint16_t) 0)) {
            map[pos] = 0;
            if (info->vspeed < 4) {
                info->vspeed ++;
            }
        }
    }
}
void handle_player_prop_get()
{
    uint16_t vpos[2];
    get_player_vpos(vpos, &player0_info);
    uint16_t pos[2];
    pos[0] = vpos[0] / 3;
    pos[1] = vpos[1] / 3;
    int xpos = (int) pos[0];
```

```c
    int ypos = (int) pos[1];
    int blocktlx = (xpos - 6) / 16;
    int blocktly = (ypos - 6) / 16;
    player_get_prop(blocktly * 40 + blocktlx, &player0_info);
    int blocktrx = (xpos + 7) / 16;
    int blocktry = (ypos - 6) / 16;
    player_get_prop(blocktry * 40 + blocktrx, &player0_info);
    int blockblx = (xpos - 6) / 16;
    int blockbly = (ypos + 7) / 16;
    player_get_prop(blockbly * 40 + blockblx, &player0_info);
    int blockbrx = (xpos + 7) / 16;
    int blockbry = (ypos + 7) / 16;
    player_get_prop(blockbry * 40 + blockbrx, &player0_info);
    get_player_vpos(vpos, &player1_info);
    pos[0] = vpos[0] / 3;
    pos[1] = vpos[1] / 3;
    xpos = (int) pos[0];
    ypos = (int) pos[1];
    blocktlx = (xpos - 6) / 16;
    blocktly = (ypos - 6) / 16;
    player_get_prop(blocktly * 40 + blocktlx, &player1_info);
    blocktrx = (xpos + 7) / 16;
    blocktry = (ypos - 6) / 16;
    player_get_prop(blocktry * 40 + blocktrx, &player1_info);
    blockblx = (xpos - 6) / 16;
    blockbly = (ypos + 7) / 16;
    player_get_prop(blockbly * 40 + blockblx, &player1_info);
    blockbrx = (xpos + 7) / 16;
    blockbry = (ypos + 7) / 16;
    player_get_prop(blockbry * 40 + blockbrx, &player1_info);
}
void handle_bomb_explode(void) {
    struct Bomb *curr = bombs_head;
    struct Bomb *prev = bombs_head;
    while(curr) {
        curr->time_left --;
        if (curr->time_left <= 0) {
            if (curr == bombs_head) {
                if (start_explosion(curr)) {
                    prev = curr;
                    curr = curr->next;
                    continue;
                }
                bombs_head = curr->next;
                switch(curr->player) {
                    case PLAYER0:
                        player0_info.bombs_left ++;
                        break;
                    case PLAYER1:
                        player1_info.bombs_left ++;
                        break;
                }
                struct Bomb* temp = curr->next;
                free(curr);
                curr = temp;
            }
            else {
                if (start_explosion(curr)) {
                    prev = curr;
                    curr = curr->next;
                    continue;
                }
                prev->next = curr->next;
                switch(curr->player) {
                    case PLAYER0:
                        player0_info.bombs_left ++;
                        break;
                    case PLAYER1:
                        player1_info.bombs_left ++;
                        break;
                }
                struct Bomb *temp = curr->next;
                free(curr);
                curr = temp;
            }
        }
        else {
            prev = curr;
            curr = curr->next;
        }

    }
}

void free_bombs(void) {
    struct Bomb* curr = bombs_head;
    while(curr) {
        struct Bomb *temp = curr->next;
        free(curr);
        curr = temp;
    }
}

void reset_map_change_list(void) {
    memset((void *) map_change_list, 0, 12 * sizeof(short));
    map_change_list_next = 0;
}

void sync_hw_map_change() {
    memcpy(&global_info.map_change_0, &map_change_list[0], sizeof(short));
    memcpy(&global_info.map_change_1, &map_change_list[1], sizeof(short));
    memcpy(&global_info.map_change_2, &map_change_list[2], sizeof(short));
    memcpy(&global_info.map_change_3, &map_change_list[3], sizeof(short));
    memcpy(&global_info.map_change_4, &map_change_list[4], sizeof(short));
    memcpy(&global_info.map_change_5, &map_change_list[5], sizeof(short));
    memcpy(&global_info.map_change_6, &map_change_list[6], sizeof(short));
    memcpy(&global_info.map_change_7, &map_change_list[7], sizeof(short));
    memcpy(&global_info.map_change_8, &map_change_list[8], sizeof(short));
    memcpy(&global_info.map_change_9, &map_change_list[9], sizeof(short));
    memcpy(&global_info.map_change_10, &map_change_list[10], sizeof(short));
    memcpy(&global_info.map_change_11, &map_change_list[11], sizeof(short));
}
```

```
void display_game_over()
{
    if (player0_info.dead && !player1_info.dead) {
        uint16_t pos[2] = {21, 21};
        uint16_t pos2[2] = {21, 42};
        set_player_pos(pos, &player0_info);
        set_player_pos(pos2, &player1_info);
        for(int i = 0; i < 100; i++) {
            write_tile(i * 12, p2_win_arr[i * 12], &global_info.map_change_0);
            write_tile(i * 12 + 1, p2_win_arr[i * 12 + 1], &global_info.map_change_1);
            write_tile(i * 12 + 2, p2_win_arr[i * 12 + 2], &global_info.map_change_2);
            write_tile(i * 12 + 3, p2_win_arr[i * 12 + 3], &global_info.map_change_3);
            write_tile(i * 12 + 4, p2_win_arr[i * 12 + 4], &global_info.map_change_4);
            write_tile(i * 12 + 5, p2_win_arr[i * 12 + 5], &global_info.map_change_5);
            write_tile(i * 12 + 6, p2_win_arr[i * 12 + 6], &global_info.map_change_6);
            write_tile(i * 12 + 7, p2_win_arr[i * 12 + 7], &global_info.map_change_7);
            write_tile(i * 12 + 8, p2_win_arr[i * 12 + 8], &global_info.map_change_8);
            write_tile(i * 12 + 9, p2_win_arr[i * 12 + 9], &global_info.map_change_9);
            write_tile(i * 12 + 10, p2_win_arr[i * 12 + 10],
                &global_info.map_change_10);
            write_tile(i * 12 + 11, p2_win_arr[i * 12 + 11],
                &global_info.map_change_11);
            write_player_info();
            pass_game_info();
            usleep(2000);
        }
        printf("player1 wins!\n");
    }
    else if (player1_info.dead && !player0_info.dead){
        uint16_t pos[2] = {21, 21};
        uint16_t pos2[2] = {21, 42};
        set_player_pos(pos, &player0_info);
        set_player_pos(pos2, &player1_info);
        for(int i = 0; i < 100; i++) {
            write_tile(i * 12, p1_win_arr[i * 12], &global_info.map_change_0);
            write_tile(i * 12 + 1, p1_win_arr[i * 12 + 1], &global_info.map_change_1);
            write_tile(i * 12 + 2, p1_win_arr[i * 12 + 2], &global_info.map_change_2);
            write_tile(i * 12 + 3, p1_win_arr[i * 12 + 3], &global_info.map_change_3);
            write_tile(i * 12 + 4, p1_win_arr[i * 12 + 4], &global_info.map_change_4);
            write_tile(i * 12 + 5, p1_win_arr[i * 12 + 5], &global_info.map_change_5);
            write_tile(i * 12 + 6, p1_win_arr[i * 12 + 6], &global_info.map_change_6);
            write_tile(i * 12 + 7, p1_win_arr[i * 12 + 7], &global_info.map_change_7);
            write_tile(i * 12 + 8, p1_win_arr[i * 12 + 8], &global_info.map_change_8);
            write_tile(i * 12 + 9, p1_win_arr[i * 12 + 9], &global_info.map_change_9);
            write_tile(i * 12 + 10, p1_win_arr[i * 12 + 10],
                &global_info.map_change_10);
            write_tile(i * 12 + 11, p1_win_arr[i * 12 + 11],
                &global_info.map_change_11);
            write_player_info();
            pass_game_info();
            usleep(2000);
        }
        printf("player0_wins!\n");
    }
    else {
        uint16_t pos[2] = {21, 21};
```

```c
        uint16_t pos2[2] = {21, 42};
        set_player_pos(pos, &player0_info);
        set_player_pos(pos2, &player1_info);
        for(int i = 0; i < 100; i++) {
            write_tile(i * 12, tie_arr[i * 12], &global_info.map_change_0);
            write_tile(i * 12 + 1, tie_arr[i * 12 + 1], &global_info.map_change_1);
            write_tile(i * 12 + 2, tie_arr[i * 12 + 2], &global_info.map_change_2);
            write_tile(i * 12 + 3, tie_arr[i * 12 + 3], &global_info.map_change_3);
            write_tile(i * 12 + 4, tie_arr[i * 12 + 4], &global_info.map_change_4);
            write_tile(i * 12 + 5, tie_arr[i * 12 + 5], &global_info.map_change_5);
            write_tile(i * 12 + 6, tie_arr[i * 12 + 6], &global_info.map_change_6);
            write_tile(i * 12 + 7, tie_arr[i * 12 + 7], &global_info.map_change_7);
            write_tile(i * 12 + 8, tie_arr[i * 12 + 8], &global_info.map_change_8);
            write_tile(i * 12 + 9, tie_arr[i * 12 + 9], &global_info.map_change_9);
            write_tile(i * 12 + 10, tie_arr[i * 12 + 10], &global_info.map_change_10);
            write_tile(i * 12 + 11, tie_arr[i * 12 + 11], &global_info.map_change_11);
            write_player_info();
            pass_game_info();
            usleep(2000);
        }
        printf("It's a tie!\n");
    }
}
void set_player_status (uint16_t vxpos, uint16_t vypos, enum FACING facing, enum
    STAT status, enum POSE pose, uint16_t pos_tick, uint16_t vspeed, uint16_t
    bomb_range, uint16_t max_bombs, uint16_t bombs_left, uint16_t bomb_colddown,
    uint16_t dead, enum PLAYER player)
{
    player_info_t *target_info;
    if (player == PLAYER0)
        target_info = &player0_info;
    else
        target_info = &player1_info;
    target_info->vxpos = vxpos;
    target_info->vypos = vypos;
    target_info->facing = facing;
    target_info->status = status;
    target_info->pose = pose;
    target_info->pos_tick = pos_tick;
    target_info->vspeed = vspeed;
    target_info->bomb_range = bomb_range;
    target_info->max_bombs = max_bombs;
    target_info->bombs_left = bombs_left;
    target_info->bomb_colddown = bomb_colddown;
    target_info->dead = dead;
}

void init_players()
{
    set_player_status (PLAYER0_INIT_X * 3, PLAYER0_INIT_Y * 3, DOWN, STATIC, IDLE,
        0, 1, 1, 1, 1, 0, 0, PLAYER0);
    set_player_status (PLAYER1_INIT_X * 3, PLAYER1_INIT_Y * 3, DOWN, STATIC, IDLE,
        0, 1, 1, 1, 1, 0, 0, PLAYER1);
}


```

```c
void write_player_info()
{
    uint16_t pos = 0;
    global_info.playerinfo00 = player0_info.vxpos / 3;
    if (player0_info.status == MOVING)
        SET_BIT(global_info.playerinfo00, 10);
    if (player0_info.pose == SIDE1 || player0_info.pose == DOWN1 ||
         player0_info.pose == UP1)
            pos = 1;
    else if (player0_info.pose == SIDE2)
            pos = 2;
    SET_BITS(global_info.playerinfo00, 11, pos);
    global_info.playerinfo01 = player0_info.vypos / 3;
    SET_BITS(global_info.playerinfo01, 9, player0_info.facing);


    global_info.playerinfo10 = player1_info.vxpos / 3;
    if (player1_info.status == MOVING)
        SET_BIT(global_info.playerinfo10, 10);
    pos = 0;
    if (player1_info.pose == SIDE1 || player1_info.pose == DOWN1 ||
         player1_info.pose == UP1)
            pos = 1;
    else if (player1_info.pose == SIDE2)
            pos = 2;
    SET_BITS(global_info.playerinfo10, 11, pos);
    global_info.playerinfo11 = player1_info.vypos / 3;
    SET_BITS(global_info.playerinfo11, 9, player1_info.facing);
}

void write_tile(uint16_t tile_pos, uint16_t tile_type, unsigned short *pos)
{
    *pos = tile_pos;
    SET_BITS(*pos, 11, tile_type);
    SET_BIT(*pos, 15);
}

void pass_game_info()
{
    vga_ball_arg_t vla;
    vla.background = global_info;
    if (ioctl(vga_fd, VGA_BALL_WRITE_BACKGROUND, &vla)) {
        perror("ioctl(VGA_BALL_SET_BACKGROUND) failed");
        return;
    }
}

uint16_t manhattan_distance(uint16_t x_0, uint16_t y_0, uint16_t x_1, uint16_t y_1)
{
    uint16_t x_diff;
    if (x_0 < x_1)
        x_diff = x_1 - x_0;
    else
        x_diff = x_0 - x_1;
    uint16_t y_diff;
    if (y_0 < y_1)
```

```c
        y_diff = y_1 - y_0;
    else
        y_diff = y_0 - y_1;
    return x_diff + y_diff;

}

uint16_t is_obstacle(uint16_t tile)
{
    return ((tile > 0) && (tile < 8));
}

void generate_software_map()
{
    time_t t;
    srand((unsigned) time(&t));
    write_player_info();
    pass_game_info();
    map = (uint16_t *) malloc (MAP_SIZE * sizeof(uint16_t));
    uint16_t row_p0 = 10;
    uint16_t col_p0 = 10;
    uint16_t row_p1 = 20;
    uint16_t col_p1 = 30;
    for (uint16_t i = 0; i < MAP_SIZE; i ++) {
        uint16_t row = i / 40;
        uint16_t col = i % 40;
        if (row % 2 == 1 && col % 2 == 1) {
            map[i] = 1;
            continue;
        }
        if (rand() % 10 < 2 && manhattan_distance(row, col, row_p0, col_p0) > 10 &&
             manhattan_distance(row, col, row_p1, col_p1) > 10) {
            map[i] = 2;
            continue;
        }
        map[i] = 0;
    }
    for (int i = 0; i < 100; i++) {
        write_tile(i * 12, map[i * 12], &global_info.map_change_0);
        write_tile(i * 12 + 1, map[i * 12 + 1], &global_info.map_change_1);
        write_tile(i * 12 + 2, map[i * 12 + 2], &global_info.map_change_2);
        write_tile(i * 12 + 3, map[i * 12 + 3], &global_info.map_change_3);
        write_tile(i * 12 + 4, map[i * 12 + 4], &global_info.map_change_4);
        write_tile(i * 12 + 5, map[i * 12 + 5], &global_info.map_change_5);
        write_tile(i * 12 + 6, map[i * 12 + 6], &global_info.map_change_6);
        write_tile(i * 12 + 7, map[i * 12 + 7], &global_info.map_change_7);
        write_tile(i * 12 + 8, map[i * 12 + 8], &global_info.map_change_8);
        write_tile(i * 12 + 9, map[i * 12 + 9], &global_info.map_change_9);
        write_tile(i * 12 + 10, map[i * 12 + 10], &global_info.map_change_10);
        write_tile(i * 12 + 11, map[i * 12 + 11], &global_info.map_change_11);
        pass_game_info();
        usleep(2000);
    }
}

int is_player_moving(player_info_t *info)
```

```
{
    if (info -> status == STATIC)
        return 0;
    else
        return 1;
}

enum FACING get_player_facing(player_info_t *info)
{
    return info -> facing;
}

void get_player_vpos(uint16_t *pos, player_info_t *info)
{
    pos[0] = info -> vxpos;
    pos[1] = info -> vypos;
}

uint16_t get_player_vspeed(player_info_t *info)
{
    return info -> vspeed;
}

void set_player_moving (int moving, player_info_t *info)
{
    if (moving)
        info -> status = MOVING;
    else
        info -> status = STATIC;
}

void set_player_pos (uint16_t *pos, player_info_t *info)
{
    info -> vxpos = pos[0];
    info -> vypos = pos[1];
}

void set_player_sprite(enum POSE pos, player_info_t *info)
{
    info -> pose = pos;
}

void set_player_facing(enum FACING facing, player_info_t *info)
{
    info -> facing = facing;
}

int detect_static_flame_collision(uint16_t *pos) {
    int xpos = (int) pos[0];
    int ypos = (int) pos[1];
    int blocktlx = (xpos - 6) / 16;
    int blocktly = (ypos - 6) / 16;
    int blocktrx = (xpos + 7) / 16;
    int blocktry = (ypos - 6) / 16;
    int blockblx = (xpos - 6) / 16;
    int blockbly = (ypos + 7) / 16;
    int blockbrx = (xpos + 7) / 16;
    int blockbry = (ypos + 7) / 16;
    uint16_t tile_tl = map[blocktly * 40 + blocktlx];
    uint16_t tile_tr = map[blocktry * 40 + blocktrx];
    uint16_t tile_bl = map[blockbly * 40 + blockblx];
    uint16_t tile_br = map[blockbry * 40 + blockbrx];
    return (is_flame(tile_tl) || is_flame(tile_tr) || is_flame(tile_br) ||
        is_flame(tile_bl));
}
void handle_player_movement()
{
    uint16_t player0_curr_vpos[2];
    uint16_t player1_curr_vpos[2];
    uint16_t player0_attempt_vpos[2];
    uint16_t player1_attempt_vpos[2];
    get_player_vpos(player0_curr_vpos, &player0_info);
    get_player_vpos(player1_curr_vpos, &player1_info);
    memcpy((void *) player0_attempt_vpos, (void *) player0_curr_vpos, 2 *
        sizeof(uint16_t));
    memcpy((void *) player1_attempt_vpos, (void *) player1_curr_vpos, 2 *
        sizeof(uint16_t));
    if (control_info.direction0 == get_player_facing(&player0_info) &&
        control_info.press_tick0 > 10 && control_info.idle0 == 0) {
        uint16_t player0_vs = get_player_vspeed(&player0_info);
        switch (control_info.direction0) {
            case DOWN:
                player0_attempt_vpos[1] += player0_vs;
                break;
            case LEFT:
                player0_attempt_vpos[0] -= player0_vs;
                break;
            case UP:
                player0_attempt_vpos[1] -= player0_vs;
                break;
            case RIGHT:
                player0_attempt_vpos[0] += player0_vs;
                break;
        }
        uint16_t player0_attempt_xpos = player0_attempt_vpos[0] / 3;
        uint16_t player0_attempt_ypos = player0_attempt_vpos[1] / 3;
        int num_corner = 0;
        int fit = 0;
        int blocktlx = (player0_attempt_xpos - 6) / 16;
        int blocktly = (player0_attempt_ypos - 6) / 16;
        if (is_obstacle(map[blocktly * 40 + blocktlx])){
            if (is_flame(map[blocktly * 40 + blocktlx]))
                player0_info.dead = 1;
            num_corner++;
            if (get_player_facing(&player0_info) == LEFT){
                fit = 0; //move down
            }
            else{
                fit = 1; //move right
            }
        }

        int blocktrx = (player0_attempt_xpos + 7) / 16;
        int blocktry = (player0_attempt_ypos - 6) / 16;
        if (is_obstacle(map[blocktry * 40 + blocktrx])){
            if (is_flame(map[blocktry * 40 + blocktrx]))
                player0_info.dead = 1;
            num_corner++;
            if (get_player_facing(&player0_info) == RIGHT){
                fit = 0; //move down
            }
            else{
                fit = 2; //move left
            }
        }

        int blockblx = (player0_attempt_xpos - 6) / 16;
        int blockbly = (player0_attempt_ypos + 7) / 16;
        if (is_obstacle(map[blockbly * 40 + blockblx])){
            if (is_flame(map[blockbly * 40 + blockblx]))
                player0_info.dead = 1;
            num_corner++;
            if (get_player_facing(&player0_info) == LEFT){
                fit = 3; //move up
            }
            else{
                fit = 1; //move right
            }
        }

        int blockbrx = (player0_attempt_xpos + 7) / 16;
        int blockbry = (player0_attempt_ypos + 7) / 16;
        if (is_obstacle(map[blockbry * 40 + blockbrx])){
            if (is_flame(map[blockbry * 40 + blockbrx]))
                player0_info.dead = 1;
            num_corner++;
            if (get_player_facing(&player0_info) == RIGHT){
                fit = 3; //move up
            }
            else{
                fit = 2; //move left
            }
        }
        if (num_corner == 1){
            memcpy((void *) player0_attempt_vpos, (void *) player0_curr_vpos, 2 *
                 sizeof(uint16_t));
            switch(fit){
                case 0:
                    player0_attempt_vpos[1] += 1;
                    break;
                case 1:
                    player0_attempt_vpos[0] += 1;
                    break;
                case 2:
                    player0_attempt_vpos[0] -= 1;
                    break;
                case 3:
                    player0_attempt_vpos[1] -= 1;
                break;
            }
        }
        if (num_corner > 1){
            memcpy((void *) player0_attempt_vpos, (void *) player0_curr_vpos, 2 *
                 sizeof(uint16_t));
        }
        if (player0_attempt_xpos > PLAYER_X_UPPER_LIM || player0_attempt_xpos <
             PLAYER_X_LOWER_LIM || player0_attempt_ypos > PLAYER_Y_UPPER_LIM ||
             player0_attempt_ypos < PLAYER_Y_LOWER_LIM)
            memcpy((void *) player0_attempt_vpos, (void *) player0_curr_vpos, 2 *
                 sizeof(uint16_t));
        if (abs_diff(player0_attempt_xpos, player1_curr_vpos[0] / 3) < 16 &&
             abs_diff(player0_attempt_ypos, player1_curr_vpos[1] / 3) < 16)
            memcpy((void *) player0_attempt_vpos, (void *) player0_curr_vpos, 2 *
                 sizeof(uint16_t));
        set_player_pos(player0_attempt_vpos, &player0_info);
        printf("player0_attempt_vpos = (%d, %d)\n", player0_attempt_xpos,
             player0_attempt_ypos);
        set_player_moving(1, &player0_info);
        /*Determine the pose*/
        uint16_t pos_tick = ++ player0_info.pos_tick;
        if (pos_tick > 1000)
            pos_tick = 0;
        player0_info.pos_tick = pos_tick;
        enum POSE pos;
        switch (control_info.direction0) {
            case DOWN:
                if ((pos_tick / 20) % 2 == 1) pos = DOWN1;
                else pos = DOWN0;
                break;
            case LEFT:
                if ((pos_tick / 20) % 3 == 1) pos = SIDE1;
                else if (((pos_tick / 20) % 3 == 2)) pos = SIDE2;
                else pos = SIDE0;
                break;
            case UP:
                if ((pos_tick / 20) % 2 == 1) pos = UP1;
                else pos = UP0;
                break;
            case RIGHT:
                if ((pos_tick / 20) % 3 == 1) pos = SIDE1;
                else if (((pos_tick / 20) % 3 == 2)) pos = SIDE2;
                else pos = SIDE0;
                break;
        }
        set_player_sprite(pos, &player0_info);


    }
    else {
        /*Turn*/
        uint16_t player0_vpos[2];
        uint16_t player0_pos[2];
        get_player_vpos(player0_vpos, &player0_info);
        player0_pos[0] = player0_vpos[0] / 3;
        player0_pos[1] = player0_vpos[1] / 3;
        if (detect_static_flame_collision(player0_pos))
            player0_info.dead = 1;
        set_player_facing(control_info.direction0, &player0_info);
        set_player_moving(0, &player0_info);
        player0_info.pos_tick = 0;
        set_player_sprite(IDLE, &player0_info);
    }
    /*Update p0 pos*/
    get_player_vpos(player0_curr_vpos, &player0_info);

    /*Player1*/
    if (control_info.direction1 == get_player_facing(&player1_info) &&
         control_info.press_tick1 > 10 && control_info.idle1 == 0) {
        uint16_t player1_vs = get_player_vspeed(&player1_info);
        switch (control_info.direction1) {
            case DOWN:
                player1_attempt_vpos[1] += player1_vs;
                break;
            case LEFT:
                player1_attempt_vpos[0] -= player1_vs;
                break;
            case UP:
                player1_attempt_vpos[1] -= player1_vs;
                break;
            case RIGHT:
                player1_attempt_vpos[0] += player1_vs;
                break;
        }

        uint16_t player1_attempt_xpos = player1_attempt_vpos[0] / 3;
        uint16_t player1_attempt_ypos = player1_attempt_vpos[1] / 3;

        int num_corner = 0;
        int fit = 0;
        int blocktlx = (player1_attempt_xpos - 6) / 16;
        int blocktly = (player1_attempt_ypos - 6) / 16;
        if (is_obstacle(map[blocktly * 40 + blocktlx])){
            if (is_flame(map[blocktly * 40 + blocktlx]))
                player1_info.dead = 1;
            num_corner++;
            if (get_player_facing(&player1_info) == LEFT){
                fit = 0; //move down
            }
            else{
                fit = 1; //move right
            }
        }

        int blocktrx = (player1_attempt_xpos + 7) / 16;
        int blocktry = (player1_attempt_ypos - 6) / 16;
        if (is_obstacle(map[blocktry * 40 + blocktrx])){
            if (is_flame(map[blocktry * 40 + blocktrx]))
                player1_info.dead = 1;
            num_corner++;
            if (get_player_facing(&player1_info) == RIGHT){
```

```c
            fit = 0; //move down
        }
        else{
            fit = 2; //move left
        }
    }

    int blockblx = (player1_attempt_xpos - 6) / 16;
    int blockbly = (player1_attempt_ypos + 7) / 16;
    if (is_obstacle(map[blockbly * 40 + blockblx])){
        if (is_flame(map[blockbly * 40 + blockblx]))
            player1_info.dead = 1;
        num_corner++;
        if (get_player_facing(&player1_info) == LEFT){
            fit = 3; //move up
        }
        else{
            fit = 1; //move right
        }
    }

    int blockbrx = (player1_attempt_xpos + 7) / 16;
    int blockbry = (player1_attempt_ypos + 7) / 16;
    if (is_obstacle(map[blockbry * 40 + blockbrx])){
        if (is_flame(map[blockbry * 40 + blockbrx]))
            player1_info.dead = 1;
        num_corner++;
        if (get_player_facing(&player1_info) == RIGHT){
            fit = 3; //move up
        }
        else{
            fit = 2; //move left
        }
    }
    if (num_corner == 1){
        memcpy((void *) player1_attempt_vpos, (void *) player1_curr_vpos, 2 *
             sizeof(uint16_t));
        switch(fit){
            case 0:
                player1_attempt_vpos[1] += 1;
                break;
            case 1:
                player1_attempt_vpos[0] += 1;
                break;
            case 2:
                player1_attempt_vpos[0] -= 1;
                break;
            case 3:
                player1_attempt_vpos[1] -= 1;
                break;
        }
    }
    if (num_corner > 1){
        memcpy((void *) player1_attempt_vpos, (void *) player1_curr_vpos, 2 *
             sizeof(uint16_t));
    }
```

```c
        if (player1_attempt_xpos > PLAYER_X_UPPER_LIM || player1_attempt_xpos <
             PLAYER_X_LOWER_LIM || player1_attempt_ypos > PLAYER_Y_UPPER_LIM ||
             player1_attempt_ypos < PLAYER_Y_LOWER_LIM)
            memcpy((void *) player1_attempt_vpos, (void *) player1_curr_vpos, 2 *
                 sizeof(uint16_t));
        if ((abs_diff(player1_attempt_xpos, (player0_curr_vpos[0] / (uint16_t) 3)) <
             (uint16_t) 16) && (abs_diff(player1_attempt_ypos, (player0_curr_vpos[1]
             / (uint16_t) 3)) < (uint16_t) 16))
            memcpy((void *) player1_attempt_vpos, (void *) player1_curr_vpos, 2 *
                 sizeof(uint16_t));
        set_player_pos(player1_attempt_vpos, &player1_info);
        printf("player1_attempt_vpos = (%d, %d)\n", player1_attempt_xpos,
             player1_attempt_ypos);
        set_player_moving(1, &player1_info);
        /*Determine the pose*/
        uint16_t pos_tick = ++ player1_info.pos_tick;
        if (pos_tick > 1000)
            pos_tick = 0;
        player1_info.pos_tick = pos_tick;
        enum POSE pos;
        switch (control_info.direction1) {
            case DOWN:
                if ((pos_tick / 20) % 2 == 1) pos = DOWN1;
                else pos = DOWN0;
                break;
            case LEFT:
                if ((pos_tick / 20) % 3 == 1) pos = SIDE1;
                else if (((pos_tick / 20) % 3 == 2)) pos = SIDE2;
                else pos = SIDE0;
                break;
            case UP:
                if ((pos_tick / 20) % 2 == 1) pos = UP1;
                else pos = UP0;
                break;
            case RIGHT:
                if ((pos_tick / 20) % 3 == 1) pos = SIDE1;
                else if (((pos_tick / 20) % 3 == 2)) pos = SIDE2;
                else pos = SIDE0;
                break;
        }
        set_player_sprite(pos, &player1_info);


    }
    else {
        uint16_t player1_vpos[2];
        uint16_t player1_pos[2];
        get_player_vpos(player1_vpos, &player1_info);
        player1_pos[0] = player1_vpos[0] / 3;
        player1_pos[1] = player1_vpos[1] / 3;
        if (detect_static_flame_collision(player1_pos))
            player1_info.dead = 1;
        /*Turn*/
        set_player_facing(control_info.direction1, &player1_info);
        set_player_moving(0, &player1_info);
        player1_info.pos_tick = 0;
        set_player_sprite(IDLE, &player1_info);
    }

}

int can_player_place_bomb(player_info_t *info) {
    return (info->bomb_colddown == 0) && (info->bombs_left > 0);
}
/*Change the memory*/
int map_change_list_append_tile(uint16_t pos, uint16_t tile)
{
    if (map_change_list_next >= 12)
        return -1;
    else {
        write_tile(pos, tile, &map_change_list[map_change_list_next]);
        map_change_list_next ++;
        return 0;
    }
}


void handle_player_place_bomb ()
{
    int can_player0_place_bomb = can_player_place_bomb(&player0_info);
    if (control_info.attempt_place_bomb_0 || control_info.attempt_place_bomb_1) {
        printf("Player 0: bombs left %d, bomb colddown %d\n",
             player0_info.bombs_left, player0_info.bomb_colddown);
        printf("Player 1: bombs left %d, bomb colddown %d\n",
             player1_info.bombs_left, player1_info.bomb_colddown);
    }
    if (can_player0_place_bomb && control_info.attempt_place_bomb_0) {
        enum FACING facing = get_player_facing(&player0_info);
        uint16_t curr_vpos[2];
        get_player_vpos(curr_vpos, &player0_info);
        int player_x = (int) curr_vpos[0] / 3;
        int player_y = (int) curr_vpos[1] / 3;
        int attempt_pos_r;
        int attempt_pos_c;
        int player_r = player_y / 16;
        int player_c = player_x / 16;
        switch (facing) {
            case DOWN:
                attempt_pos_r = player_r + 1;
                attempt_pos_c = player_c;
                break;
            case LEFT:
                attempt_pos_r = player_r;
                attempt_pos_c = player_c - 1;
                break;
            case UP:
                attempt_pos_r = player_r - 1;
                attempt_pos_c = player_c;
                break;
            case RIGHT:
                attempt_pos_r = player_r;
                attempt_pos_c = player_c + 1;
                break;
        }
        if (attempt_pos_r >= 0 && attempt_pos_r < 30 && attempt_pos_c >= 0 &&
            attempt_pos_c < 40) {

            int attempt_center_x = attempt_pos_c * 16 + 7;
            int attempt_center_y = attempt_pos_r * 16 + 7;
            int md = abs_diff(player_x, attempt_center_x) + abs_diff(player_y,
                attempt_center_y);
            int attempt_pos = attempt_pos_c + attempt_pos_r * 40;
            if (map[attempt_pos] == 0 && md > 12) {
                if (map[attempt_pos] == 0) {
                    if (map_change_list_append_tile((uint16_t) attempt_pos, (uint16_t)
                        3) == 0) {
                        map[attempt_pos] = 3;
                        if (md <= 16) {
                            uint16_t player_next_vpos[2];
                            player_next_vpos[0] = (uint16_t) (player_c * 16 + 8)* 3;
                            player_next_vpos[1] = (uint16_t) (player_r * 16 + 8)* 3;
                            set_player_pos(player_next_vpos, &player0_info);
                        }
                        insert_bomb(PLAYER0, attempt_pos);
                        player0_info.bombs_left --;
                        player0_info.bomb_colddown = 50;
                    }
                }
            }
        }
    }
    else {
        if (player0_info.bomb_colddown > 0)
            player0_info.bomb_colddown --;
    }
    int can_player1_place_bomb = can_player_place_bomb(&player1_info);
    if (can_player1_place_bomb && control_info.attempt_place_bomb_1) {
        enum FACING facing = get_player_facing(&player1_info);
        uint16_t curr_vpos[2];
        get_player_vpos(curr_vpos, &player1_info);
        int player_x = (int) curr_vpos[0] / 3;
        int player_y = (int) curr_vpos[1] / 3;
        int attempt_pos_r;
        int attempt_pos_c;
        int player_r = player_y / 16;
        int player_c = player_x / 16;
        switch (facing) {
            case DOWN:
                attempt_pos_r = player_r + 1;
                attempt_pos_c = player_c;
                break;
            case LEFT:
                attempt_pos_r = player_r;
                attempt_pos_c = player_c - 1;
                break;
            case UP:
                attempt_pos_r = player_r - 1;
                attempt_pos_c = player_c;
```

```c
                break;
            case RIGHT:
                attempt_pos_r = player_r;
                attempt_pos_c = player_c + 1;
                break;
        }
        if (attempt_pos_r >= 0 && attempt_pos_r < 30 && attempt_pos_c >= 0 &&
            attempt_pos_c < 40) {

            int attempt_center_x = attempt_pos_c * 16 + 7;
            int attempt_center_y = attempt_pos_r * 16 + 7;
            int md = abs_diff(player_x, attempt_center_x) + abs_diff(player_y,
                attempt_center_y);
            int attempt_pos = attempt_pos_c + attempt_pos_r * 40;
            if (map[attempt_pos] == 0 && md > 12) {
                if (map[attempt_pos] == 0) {
                    if (map_change_list_append_tile((uint16_t) attempt_pos, (uint16_t)
                        3) == 0) {
                        map[attempt_pos] = 3;
                        if (md <= 16) {
                            uint16_t player_next_vpos[2];
                            player_next_vpos[0] = (uint16_t) (player_c * 16 + 8)* 3;
                            player_next_vpos[1] = (uint16_t) (player_r * 16 + 8)* 3;
                            set_player_pos(player_next_vpos, &player1_info);

                        }
                        insert_bomb(PLAYER1, attempt_pos);
                        player1_info.bombs_left --;
                        player1_info.bomb_colddown = 50;
                    }
                }
            }
        }
    }
    else {
        if (player1_info.bomb_colddown > 0)
            player1_info.bomb_colddown --;
    }
}
```

H main.c

```c
    #include <stdio.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>
#include "controller.h"
#include "bomberman.h"
```

```c
extern int vga_fd;
extern game_info_t global_info;
extern player_info_t player0_info;
extern player_info_t player1_info;
extern control_info_t control_info;
extern uint16_t *map;
extern struct controller_list controllers;
extern struct args_list c_args_list;
extern struct Bomb* bombs_head;
extern struct Explosion *explosion_head;

int main()
{
    controllers = open_controllers();
    pthread_t control_thread;
    memset((void *) &control_info, 0, sizeof(control_info_t));
    c_args_list.mode = 0;
    c_args_list.print = 1;
    c_args_list.devices = controllers;
    c_args_list.control_info = &control_info;


    if (pthread_create(&control_thread, NULL, &listen_controllers, (void *)
        &c_args_list)) {
        fprintf(stderr, "Could not create controller thread\n");
        return -1;
    }
    static const char filename[] = "/dev/vga_ball";
    printf("VGA ball Userspace program started\n");

    if ( (vga_fd = open(filename, O_RDWR)) == -1) {
        fprintf(stderr, "could not open %s\n", filename);
        return -1;
    }
    memset((void *) &global_info, 0, sizeof(game_info_t));
    generate_software_map();
    init_players();
    /*for (int i = 0; i < 10000; i ++){
        xpos0 += v0;
        xpos1 += v1;
        pos_counter ++;
        pos_counter = pos_counter % 18;
        if (xpos0 > PLAYER_X_UPPER_LIM) {
            xpos0 = PLAYER_X_UPPER_LIM;
            facing0 = LEFT;
            v0 = -1;
        }
        if (xpos0 < PLAYER_X_LOWER_LIM) {
            xpos0 = PLAYER_X_LOWER_LIM;
            facing0 = RIGHT;
            v0 = 1;
        }
        if (xpos1 > PLAYER_X_UPPER_LIM) {
            xpos1 = PLAYER_X_UPPER_LIM;
            facing1 = LEFT;
            v1 = -1;
        }
        if (xpos1 < PLAYER_X_LOWER_LIM) {
            xpos1 = PLAYER_X_LOWER_LIM;
            facing1 = RIGHT;
            v1 = 1;
        }
        if (pos_counter / 6 == 0)
            pos = SIDE0;
        else if (pos_counter / 6 == 1)
            pos = SIDE1;
        else pos = SIDE2;

        set_player_status(xpos0, 136, facing0, MOVING, pos, PLAYER0);
        set_player_status(xpos1, 400, facing1, MOVING, pos, PLAYER1);
        write_player_info();
        pass_game_info();
        usleep(20000);
    }*/
    bombs_head = NULL;
    explosion_head = NULL;
    for(;;) {
        init_explosion_sound();
        reset_map_change_list();
        handle_explosion();
        handle_bomb_explode();
        handle_player_movement();
        if (player0_info.dead || player1_info.dead)
            break;
        handle_player_place_bomb();
        handle_player_prop_get();
        write_player_info();
        sync_hw_map_change();
        pass_game_info();
        usleep(10000);
    }
    display_game_over();
    pthread_cancel(control_thread);
    pthread_join(control_thread, NULL);
    free(map);
    free_bombs();
    free_explosion();
    return 0;
}
```

I Makefile

```
    ifneq (${KERNELRELEASE},)

# KERNELRELEASE defined: we are being compiled as part of the Kernel
        obj-m := vga.o

else

# We are being compiled as a module: use the Kernel build system
```

```

   KERNEL_SOURCE := /usr/src/linux-headers-$(shell uname -r)
       PWD := $(shell pwd)

default: module main

module:
   ${MAKE} -C ${KERNEL_SOURCE} SUBDIRS=${PWD} modules
main: controller.o bomberman.o main.o
   gcc -o software_moving controller.o bomberman.o main.o -lusb-1.0 -lpthread
main.o: main.c
   gcc -o main.o -c main.c -lusb-1.0 -lpthread
bomberman.o: bomberman.c
   gcc -o bomberman.o -c bomberman.c -lusb-1.0 -lpthread
controller.o: controller.c
   gcc -o controller.o -c controller.c -lusb-1.0 -lpthread
clean:
   ${MAKE} -C ${KERNEL_SOURCE} SUBDIRS=${PWD} clean
   rm *.o
TARFILES = Makefile README hello.h vga.c software_moving.c
TARFILE = lab3-sw.tar.gz
.PHONY : tar
tar : $(TARFILE)

$(TARFILE) : $(TARFILES)
   tar zcfC $(TARFILE) .. $(TARFILES:%=lab3-sw/%)
endif
```