CSEE4840 Embedded Systems Final Report: Pac-Man

Caiwu Chen, Emma Li, Haoming Ma, Tz-Jie Yu

May 14, 2025

Contents

1 Introduction

1.1 Game Overview

In our project, we re-create the classic arcade game Pac-Man on an FPGA platform, DE1-SoC Board, and using the VGA monitor. Pac-Man is a game in which the player navigates a maze, aiming to consume all the pellets while avoiding being caught by roaming ghosts. Our implementation separates responsibilities between hardware and software: the FPGA is responsible for rendering the game's graphics to a VGA display, while the software controls the game logic and communicates with the hardware via a device driver. User input is provided through a USB game controller (keyboard with arrow keys), enabling real-time control of Pac-Man's movement.

1.2 System Architecture

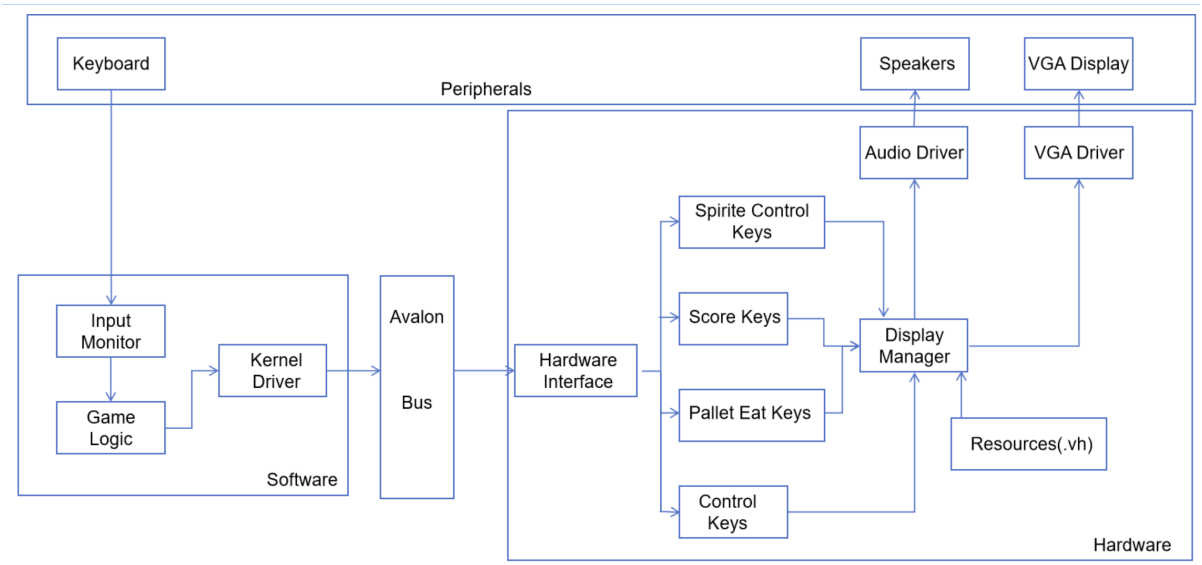

Figure 1: System Diagram

Our Pac-Man game is implemented on an FPGA platform, where the player uses a USB controller to navigate Pac-Man through a maze. The controller communicates with the game logic software via the USB protocol using the `libusb` library. The software handles player movement, pellet consumption, ghost behavior, collision detection, and score tracking.

The `controller.c` file manages USB controller initialization and input recognition. Player inputs are translated into game actions and processed by the main game loop.

Interaction with the FPGA hardware is handled through the `vga_ball.c` device driver, which communicates with the `vga_ball.sv` hardware module using a memory-mapped interface. The FPGA stores sprite data in on-chip ROMs, which the `vga_ball.sv` module retrieves to display Pac-Man, ghosts, pellets, and the maze on the VGA monitor.

Registers on the FPGA keep track of game state, including Pac-Man and ghost positions, pellet consumption, and the current score. These registers are updated by the software in real time to ensure consistent game behavior.

Audio feedback is provided using the WM8731 audio CODEC, which plays background music and sound effects such as pellet chomps and death sounds through connected headphones or speakers.

This system achieves seamless integration between software and hardware. Controller inputs are processed in software, used to update the game state, and passed to the FPGA, which renders graphics and audio accordingly. This architecture allows for a responsive and interactive gameplay experience on an FPGA-based platform.

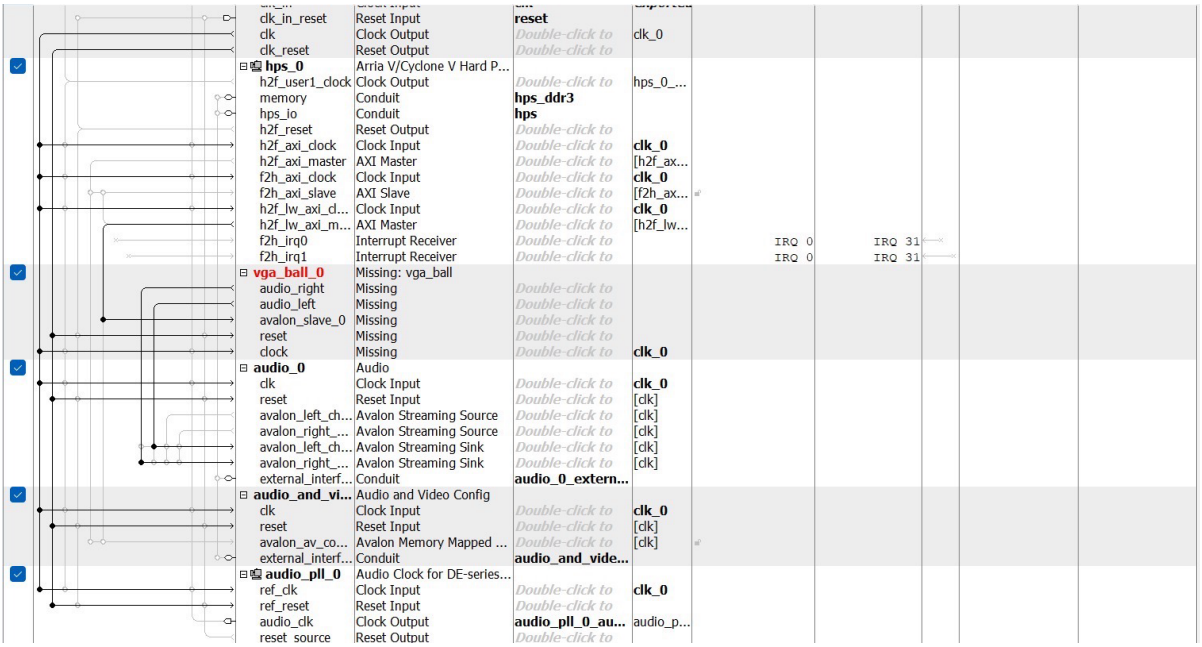

Figure 2: Qsys Connections

2 Hardware

2.1 Graphics

Our current design uses SystemVerilog to implement and render all visual components of the game, including the maze, player sprite, ghosts, pellets, characters and score digits.

Static elements such as the maze walls, pellets, and text (e.g., "SCORE") are rendered using a tile-based rendering system. The VGA display is divided into a fixed 8×8 pixel grid, and each grid cell displays a tile defined in a memory file. Considering that our VGA monitor has a resolution of 640 x 480, the total number of tiles of the VGA monitor is 80 x 60 = 4800. A tilemap stored in block RAM defines which tile appears at each position on the screen. Each tile's appearance is determined by an indexed pattern. Each tile bitmap will be rendered after deciding its position.

Maze and pellet presence is tracked using a dedicated tiles RAM, where each row of the maze corresponds to its tile ID. Since we have a total of 74 different tile types, we use an 8-bit tile ID to map the tile bitmaps and the corresponding VGA monitor position. Two kinds of pellets are also two types of tiles we predefined, and are rendered together with the static elements during the maze rendering part. While software updates the position of the Pac-Man and signals if pellet tiles need to be updated, enabling real-time dynamic updates to the pellet layout during gameplay.

Sprites, including Pac-Man, ghosts, and score digits, are stored in on-chip Block RAM or ROM. These sprites are rendered using a custom sprite engine, which reads sprite descriptor data (e.g., horizontal and vertical position, states or directions) and draws active sprites during VGA scanline generation. This approach enables layering of moving sprites over a static tilemap background with minimal flicker and latency.

2.2 Memory

For Sprite declaration, we have 21 different kinds of images. The Pac-Man has 5 different states, including facing up, down, left, right, and eating. For ghosts, we have 4 different directions, including facing up, down, left, and right. While the traditional Pac-Man has four ghosts, Blinky, Pinky, Inky, and Clyde, we need to store each ghost and its directions. All sprites are 16 x 16 bits. Border and pellet tiles are 8 x 8 bits. Letters and numbers are 8 x 16 bits, however, they actually take up two tiles, so we divide them into top and bottom parts to store in

the pattern generation table. For simplicity, we still count the combination of the top and bottom parts as one tile. We take advantage of two different sounds, one is for the Pac-Man eating a pellet, and one is for when Pac-Man is caught by the ghost. They do not exceed 2s per sound, and each one is less than 3×10^6 bits.

The Table of the memory is shown below:

Category	Name	Graphics	Size (bits) Width x Height x Channel x Bit-depth	# of images	Total size (bits)
Sprite	Pac-Man		16 x 16 x 3 x 8	16	98304
Sprite	Ghost		16 x 16 x 3 x 8	5	30720
Tile	Borders		8 x 8 x 3 x 8	36	53760
Tile	Pellets		8 x 8 x 3 x 8	2	3072
Tile	Letters		8 x 16 x 3 x 8	26	79872
Tile	Numbers		8 x 16 x 3 x 8	10	30720
Sound	Music				545408

Figure 3: Resource Budget

2.3 Audio

To support audio output in our Pac-Man game, we make use of the DE1-SoC FPGA board's built-in audio hardware. The board features three audio jacks—line out, line in, and microphone—connected to a Wolfson WM8731 audio CODEC chip. This chip handles analog-to-digital (ADC) and digital-to-analog (DAC) conversions, enabling the FPGA to interface with analog audio signals. Communication between the Nios II processor and the CODEC is managed by a dedicated audio core. This core simplifies sound playback by providing two 128-entry FIFOs: one for the left channel and one for the right. These FIFOs allow stereo audio output by buffering audio samples before sending them to the LINEOUT jack. This setup is used to play game sounds such as the intro theme, pellet chomp, and Pac-Man's death sound.

The audio circuit diagram is shown below:

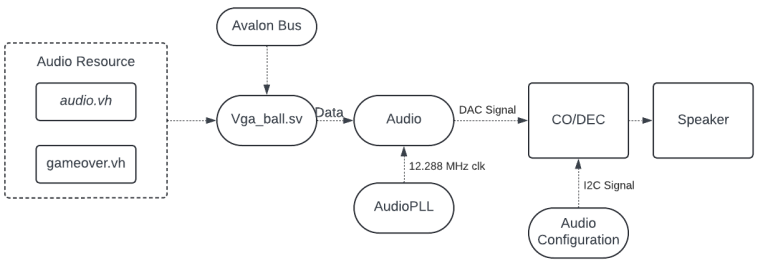

Figure 4: Audio Diagram

We started by sourcing audio files from the Internet, specifically the Pac-Man theme and death sound, extracting them from YouTube and the Pac-Man Database in.mp3 format. These files were then converted to a format compatible with the DE1-SoC FPGA board and its WM8731 audio CODEC. Using an audio conversion tool,

FFmpeg, a free software project that processes to convert audio, we transformed the .mp3 files into 16-bit .wav files with a mono channel and an 8000 Hz sample rate. This format was selected to match the left-justified audio mode supported by the CODEC and to simplify playback. After confirming the audio contained valid, non-silent samples, we used a Python script to convert the .wav files into .vh format, representing each 16-bit audio sample in hexadecimal. This .vh file was then loaded into on-chip memory, allowing the Verilog module to stream audio samples to the audio core's FIFO buffers for playback through the board's line-out jack.

3 Software

3.1 User Input

The Pac-Man game receives player input through a **KIWITATA SNES USB Gamepad**, interfaced via the `libusb-1.0` library. This controller provides directional inputs (D-Pad) and action buttons that allow the player to control Pac-Man and manage game states.

Controller Initialization is performed using the `open_controller()` function, which scans connected USB devices and opens the first detected SNES gamepad (with `idProduct == 0x11`). Once a compatible device is found, it claims the interface, detaches the kernel driver if necessary, and stores the endpoint address for polling input data.

Input Interruption is continuously handled by the `listen_controller()` function. It performs an `interrupt_transfer` to receive 7-byte packets from the gamepad. The received data structure (`controller_pkt`) encodes button states and directional inputs.

Button Mapping is decoded in the `detect_presses()` function:

- D-Pad: Maps to movement directions — Left, Right, Up, Down
- A Button: Start or reset the game.
- X Button: Pause the game.

The decoded inputs update a shared `buttons` array, which is then used by the game logic to update Pac-Man's movement direction and manage control flags (Start, Pause, Reset).

This modular design separates USB handling from the main game loop, enabling non-blocking input processing and smooth gameplay.

Figure 5: Controller

3.2 Game Logic

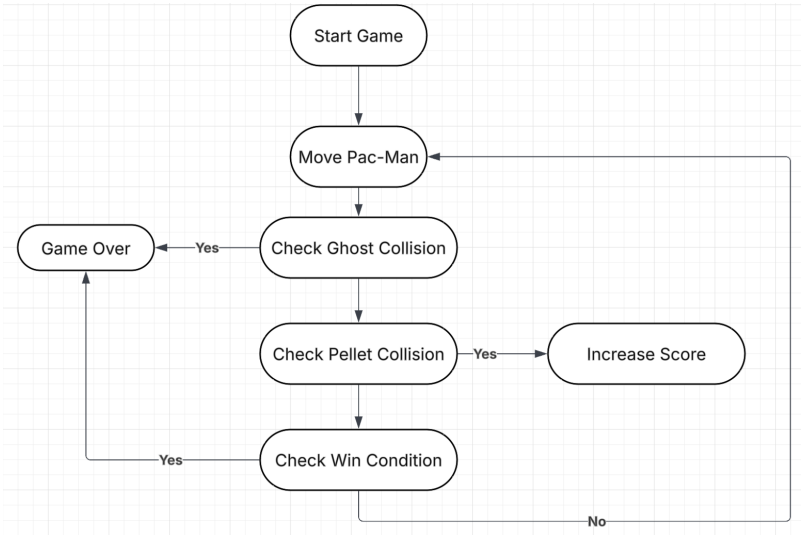

Figure 6: Game Logic

The game begins when the player presses the `A` button. The player controls Pac-Man, a character that continuously moves in the direction of its mouth. The movement direction can be changed at any time using the directional pad, which serves as the primary input during gameplay.

As Pac-Man moves across the board and encounters pellets, they are consumed, removed from the game state, and the player's score is incremented. The score is tracked in software and communicated to the hardware using a dedicated memory-mapped register. The current score is rendered on the VGA display using a tile-based sprite system with an 8×8 pixel font per digit.

The main objective is to eat all the pellets in the maze.

Enemy characters, referred to as ghosts, are non-playable sprites that roam the maze following predefined or randomized movement patterns. All four ghosts in the game use a *Chase mode*, where they attempt to pursue Pac-Man based on their individual logic. A collision between any ghost and Pac-Man results in a game-over state.

End Conditions

The game concludes when one of the following conditions is met:

- The player eats all pellets (win condition)
- The player collides with a ghost (loss condition)

To restart the game after a win or loss, the player simply presses the `A` button again.

3.2.1 PacMan Movement

Most of the Pac-Man movement is implemented in the `update_pacman` function.

```
// PacMan Movement
void update_pacman() {
    int new_x = pacman_x;
    int new_y = pacman_y;
    switch (pacman_dir) {
        case 0: new_y -= step_size ; break;
        case 1: new_x -= step_size ; break;
        case 2: new_y += step_size ; break;
        case 3: new_x += step_size ; break;
```

```
    }

    if (can_move_to(new_x, new_y)) {
        pacman_x = new_x;
        pacman_y = new_y;
    }

    int tile_x = pacman_x / TILE_WIDTH;
    int tile_y = pacman_y / TILE_HEIGHT;
    if (get_pellet_bit(tile_x, tile_y)) {
        clear_pellet_bit(tile_x, tile_y);
        set_tile(tile_x, tile_y, 0);
        pacman_dir = 5; // Eat pellet animation direction
        score += 10;
        *SCORE_REG = generate_packed_score(score);
        last_pellet_index = tile_y * SCREEN_WIDTH_TILES + tile_x;
        printf("[Pac-Man] Ate pellet at (%d, %d). Score = %d\n", tile_x, tile_y, score);
    } else {
        last_pellet_index = PELLET_NONE;
    }
    sprite_t* pac = &sprites[SPRITE_PACMAN];
    pac->x = pacman_x;
    pac->y = pacman_y;
    pac->visible = 1;
    pac->direction = pacman_dir;
    pac->frame = 0;
}
```

This function first reads the current position and direction of Pac-Man. Based on the direction (up, down, left, or right), it calculates a potential new position using a fixed step size. Before committing to the move, the function checks whether the new tile is walkable using the `can_move_to` function, which detects walls or obstacles in the tilemap.

If the move is valid, Pac-Man's position is updated. The function also checks whether there is a pellet in the new tile. If a pellet is detected, it is cleared from the pellet memory, the score is increased, and the corresponding tile is visually updated. Pac-Man's sprite data—including position, frame, and visibility—is updated accordingly for display on the VGA monitor.

Movement direction is controlled via arrow key input from a USB keyboard. Keys are translated into directional codes, which update the global `pacman_dir` variable used during movement. If Pac-Man collides with a ghost, a game-over flag is set, and the game loop halts until the user restarts the game.

3.2.2 Ghost Movement

Traditional Pac-Man ghosts follow distinct movement patterns and AI logic to create unique behaviors for each ghost. The figure below illustrates the behavioral logic and movement patterns of the ghosts in the game.

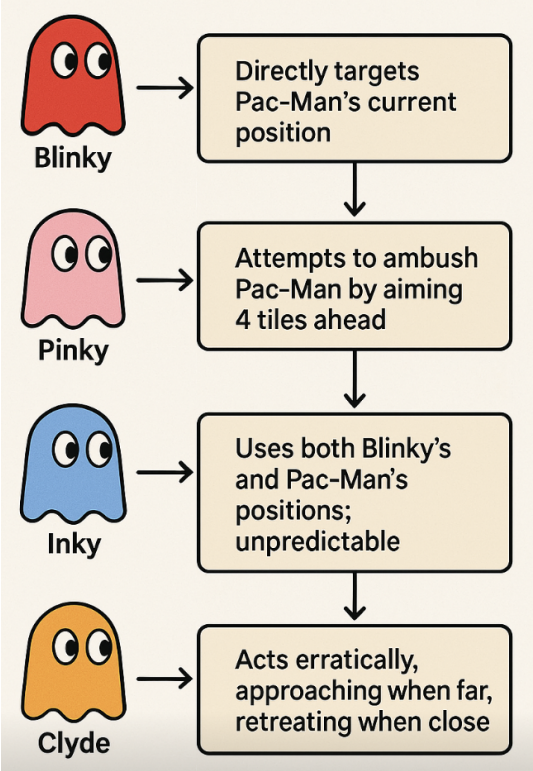

Figure 7: Ghost Behavior

Ghost movement is primarily handled in the `update_ghosts` function.

```
void update_ghosts() {
    static int ghost_tick = 0;
    ghost_tick++;
    if (ghost_tick % 10 != 0) return;

    for (int i = 0; i < NUM_GHOSTS; i++) {
        ghost_t* g = &ghosts[i];
        int best_dir = g->dir;
        int best_dist = 1 << 30;

        int curr_tx = g->x / TILE_WIDTH;
        int curr_ty = g->y / TILE_HEIGHT;

        int target_x = pacman_x, target_y = pacman_y;

        if (i == 0) {
            target_x = pacman_x;
            target_y = pacman_y;
        } else if (i == 1) {
            target_x = pacman_x;
            target_y = pacman_y;
            switch (pacman_dir) {
                case 0: target_y -= TILE_HEIGHT * 4; break;
                case 1: target_x -= TILE_WIDTH * 4; break;
                case 2: target_y += TILE_HEIGHT * 4; break;
                case 3: target_x += TILE_WIDTH * 4; break;
            }
        } else if (i == 2) {
```

```
            int bx = ghosts[0].x;
            int by = ghosts[0].y;
            int ref_x = pacman_x, ref_y = pacman_y;
            switch (pacman_dir) {
                case 0: ref_y -= TILE_HEIGHT * 2; break;
                case 1: ref_x -= TILE_WIDTH * 2; break;
                case 2: ref_y += TILE_HEIGHT * 2; break;
                case 3: ref_x += TILE_WIDTH * 2; break;
            }
            target_x = ref_x + (ref_x - bx);
            target_y = ref_y + (ref_y - by);
        } else if (i == 3) {
            int d = abs(g->x - pacman_x) + abs(g->y - pacman_y);
            if (d < TILE_WIDTH * 8) {
                target_x = 0;
                target_y = 0;
            }
        }

        for (int d = 0; d < 4; d++) {
            int dx = 0, dy = 0;
            switch (d) {
                case 0: dy = -step_size; break;
                case 1: dx = -step_size; break;
                case 2: dy = +step_size; break;
                case 3: dx = +step_size; break;
            }

            int new_x = g->x + dx;
            int new_y = g->y + dy;
            int tx = new_x / TILE_WIDTH;
            int ty = new_y / TILE_HEIGHT;

            if (!can_move_to(new_x, new_y) || is_tile_occupied_by_other_ghost(tx, ty, i)) continue;

            int dist = abs(new_x - target_x) + abs(new_y - target_y);

            if (rand() % 100 < 20) {
                best_dir = d;
                break;
            }

            if (dist < best_dist) {
                best_dist = dist;
                best_dir = d;
            }
        }

        g->dir = best_dir;
        switch (best_dir) {
            case 0: g->y -= step_size; break;
            case 1: g->x -= step_size; break;
            case 2: g->y += step_size; break;
            case 3: g->x += step_size; break;
        }

        g->sprite->x = g->x;
        g->sprite->y = g->y;
    }
}
```

Each ghost has a direction and a position stored in its own ghost_t struct, along with a pointer to its sprite for display updates. The movement is updated periodically—every 10 ticks—to control ghost speed relative to Pac-Man.

For each ghost, the function calculates a potential new position based on its current direction. Before moving, it checks whether the target tile is walkable using `can_move_to`, and ensures it is not already occupied by another ghost through `is_tile_occupied_by_other_ghost`. If both checks pass, the ghost updates its position.

If movement is blocked by a wall or another ghost, the ghost rotates to a new direction by incrementing its direction value modulo 4, cycling through up, left, down, and right. This basic avoidance behavior ensures the ghosts don't get stuck but does not implement pathfinding.

After updating positions, each ghost's sprite data is refreshed with its new coordinates, allowing the VGA display to render the updated state. Collisions between ghosts and Pac-Man are checked in the `check_gameover` function, and if a ghost overlaps with Pac-Man's tile, the game ends.

3.2.3 Pellet Mechanics

During initialization in `game_init_playfield()`, each character in the maze layout string defines the visual structure of the maze. When a pellet character ("`.`" or "`P`") is encountered, the corresponding tile is set in the tilemap and the associated bit is set in the `PELLET_RAM_BASE` array:

```
if (tiles[i] == '.' || tiles[i] == 'P') {
    PELLET_RAM_BASE[y] |= (1 << (31 - x));
}
```

Each row of `PELLET_RAM_BASE` represents a horizontal line of pellets in the maze, with each bit indicating whether a pellet is present at that column. This structure allows for efficient bitwise access and update of pellet presence.

During gameplay, the `update_pacman()` function checks whether Pac-Man's current tile contains a pellet. If a pellet is found, it is cleared both visually and logically:

```
if (get_pellet_bit(tile_x, tile_y)) {
    clear_pellet_bit(tile_x, tile_y);
    set_tile(tile_x, tile_y, 0); // Replace with blank tile
    score += 10;
    *SCORE_REG = generate_packed_score(score);
    last_pellet_index = tile_y * SCREEN_WIDTH_TILES + tile_x;
}
```

The tile is replaced with a blank tile, the score is updated, and the consumed pellet's location is stored in `last_pellet_index`.

This index is converted to a VGA tile address and passed to the FPGA through `update_all_to_driver()`:

```
state.pellet_to_eat = pellet_y * 80 + pellet_x + 6 + 1220;
```

This allows the FPGA hardware to blank out the consumed pellet on the VGA display in real-time. If no pellet is consumed in a given frame, a sentinel value `PELLET_NONE` is passed to indicate no update is needed.

This architecture supports efficient, real-time pellet rendering and updating with minimal overhead, ensuring visual consistency and responsiveness across software and hardware.

3.2.4 Collision Detection

Collision detection occurs in multiple parts of the game logic. The `can_move_to` function prevents Pac-Man and ghosts from walking into walls or unwalkable tiles, based on values in the tilemap. For enemy collisions, the `check_gameover` function determines whether Pac-Man's current position overlaps with any of the ghosts. If such a collision occurs, the game-over flag is triggered and the game loop halts until a reset. Pellet collisions are handled in `update_pacman`, where tile-based pellet detection also updates the game score and visual state.

```
bool can_move_to(int px, int py) {
    int tx = px / TILE_WIDTH;
    int ty = py / TILE_HEIGHT;
    uint8_t tile = TILEMAP_BASE[ty * SCREEN_WIDTH_TILES + tx];
    if (tile == 0x40 || tile == 1 || tile == 0) {

    }
    else{
        printf("Tile at (%d, %d) is not walkable: %d\n", tx, ty, tile);
    }
    return (tile == 0x40 || tile == 1 || tile == 0 || tile == 0xFD);
}

bool check_gameover() {
    for (int i = 0; i < NUM_GHOSTS; i++) {
        int dx = abs(pacman_x - ghosts[i].x);
        int dy = abs(pacman_y - ghosts[i].y);
        if (dx < TILE_WIDTH && dy < TILE_HEIGHT) {
            return true;
        }
    }
    return false;
}
```

3.2.5 Game State Management

The game uses a set of control flags stored in a memory-mapped control register to manage different states such as start, pause, reset, and game over. The `wait_for_start_signal` function blocks until the Enter key is pressed, signaling the beginning of the game by setting the `CTRL_START` flag. During gameplay, pressing the space bar toggles the pause state by setting or clearing `CTRL_PAUSE`, while pressing 'r' sets the `CTRL_RESET` flag and restarts the game. The game loop checks these flags to determine whether to pause updates, restart, or terminate.

```
void wait_for_start_signal() {
    printf("Waiting for START signal...\n");
    int transferred;
    while (1) {
        int r = libusb_interrupt_transfer(keyboard, endpoint_address,
                                          (unsigned char *)&packet, sizeof(packet),
                                          &transferred, 1);
        if (r == 0 && transferred == sizeof(packet)) {
            for (int i = 0; i < MAX_KEYS; i++) {
                uint8_t key = packet.keycode[i];
                if (key != 0) {
                    char c = usb_to_ascii(key, packet.modifiers);
                    if (c == '\n') {
                        set_control_flag(CTRL_START);
                        printf("START signal received. Game starting...\n");
                        return;
                    }
                }
            }
        }
        usleep(10000);
    }

    printf("Game started!\n");
}
```

3.2.6 Sprite and Display Updates

All visual elements in the game—including Pac-Man, ghosts, and pellets—are rendered via a VGA display system. The `update_all_to_driver` function collects the current state of all sprites, the score, control flags, and the last pellet eaten into a `vga_all_state_t` structure. This data is sent to the VGA kernel driver using the `ioctl` system call, allowing the hardware module to render the updated game frame. This separation of game logic and rendering ensures that the CPU can control gameplay while the VGA driver handles efficient display updates.

```
void update_all_to_driver() {
    vga_all_state_t state;

    for (int i = 0; i < 5; i++) {
        state.sprites[i].x = sprites[i].x ;
        state.sprites[i].y = sprites[i].y ;
        state.sprites[i].frame = sprites[i].frame;
        state.sprites[i].visible = sprites[i].visible;
        state.sprites[i].direction = direction_to_control(sprites[i].direction);
        state.sprites[i].type_id = sprites[i].type_id;
        state.sprites[i].reserved1 = sprites[i].rsv1;
        state.sprites[i].reserved2 = sprites[i].rsv2;
    }

    state.score = * SCORE_REG;
    state.control = *CONTROL_REG;

    if (last_pellet_index != PELLET_NONE) {
        int pellet_x = last_pellet_index % SCREEN_WIDTH_TILES;
        int pellet_y = last_pellet_index / SCREEN_WIDTH_TILES;
        state.pellet_to_eat = pellet_y * 80 + pellet_x + 6 + 1220;
    } else {
        state.pellet_to_eat = 0xFFFF;
    }

    last_pellet_index = PELLET_NONE;

    if (ioctl(vga_ball_fd, VGA_BALL_WRITE_ALL, &state)) {
        perror("ioctl(VGA_BALL_WRITE_ALL) failed");
    }

    // printf all content for debugging
    printf("=== State Debug ===\n");
    printf("Score: %d\n", state.score);
    printf("Control: 0x%02X\n", state.control);
    for (int i = 0; i < 5; i++) {
        printf("Sprite %d: x=%d, y=%d, frame=%d, visible=%d, direction=%d, type_id=%d\n",
               i,
               state.sprites[i].x,
               state.sprites[i].y,
               state.sprites[i].frame,
               state.sprites[i].visible,
               state.sprites[i].direction,
               state.sprites[i].type_id);
    }
    printf("Pellet to eat: %d\n", state.pellet_to_eat);
    printf("====================\n");
}
```

3.2.7 Tilemap and Maze Representation

The maze layout is stored as a 2D tilemap, initialized in game_init_playfield using a character array. Each character corresponds to a specific tile type—walls, open paths, pellets, and decorations—mapped to tile IDs via a lookup table. For example, a space character (' ') maps to an empty walkable tile, while characters like 'U', 'L', or '.' represent walls and pellet tiles. The tilemap serves as the basis for movement validation and rendering, and also defines pellet placement and collision zones.

```
void game_init_playfield(void) {
    static const char* tiles =
       //0123456789012345678901234567
        "0UUUUUUUUUUUU45UUUUUUUUUUUU1" // 3
        "L............rl............R" // 4
        "L.ebbf.ebbbf.rl.ebbbf.ebbf.R" // 5
        "L.r  l.r   l.rl.r   l.r  l.R" // 6
        "L.guuh.guuuh.gh.guuuh.guuh.R" // 7
        "L..........................R" // 8
        "L.ebbf.ef.ebbbbbbf.ef.ebbf.R" // 9
        "L.guuh.rl.guuyxuuh.rl.guuh.R" // 10
        "L......rl....rl....rl......R" // 11
        "2BBBBf.rzbbf rl ebbwl.eBBBB3" // 12
        "     L.rxuuh gh guuyl.R    " // 13
        "     L.rl        rl.R      " // 14
        "     L.rl mjs tjn rl.R     " // 15
        "UUUUUh.gh i     q gh.gUUUUU" // 16
        "       .   i     q   .      " // 17
        "BBBBBf.ef i     q ef.eBBBBB" // 18
        "     L.rl okkkkkkp rl.R    " // 19
        "     L.rl         rl.R     " // 20
        "     L.rl ebbbbbbf rl.R    " // 21
        "0UUUUh.gh guuyxuuh gh.gUUUU1" // 22
        "L............rl............R" // 23
        "L.ebbf.ebbbf.rl.ebbbf.ebbf.R" // 24
        "L.guyl.guuuh.gh.guuuh.rxuh.R" // 25
        "L...rl.......  .......rl...R" // 26
        "6bf.rl.ef.ebbbbbbf.ef.rl.eb8" // 27
        "7uh.gh.rl.guuyxuuh.rl.gh.gu9" // 28
        "L......rl....rl....rl......R" // 29
        "L.ebbbbwzbbf.rl.ebbwzbbbbf.R" // 30
        "L.guuuuuuuuh.gh.guuuuuuuuh.R" // 31
        "L..........................R" // 32
        "2BBBBBBBBBBBBBBBBBBBBBBBBB3"; // 33
       //0123456789012345678901234567

    uint8_t t[128];
    for (int i = 0; i < 128; i++) t[i] = 1;
    t[' '] = 0x40; t['0'] = 0xD1; t['1'] = 0xD0; t['2'] = 0xD5; t['3'] = 0xD4;
    t['4'] = 0xFB; t['5'] = 0xFA; t['6'] = 0xD7; t['7'] = 0xD9;
    t['8'] = 0xD6; t['9'] = 0xD8; t['U'] = 0xDB; t['L'] = 0xD3;
    t['R'] = 0xD2; t['B'] = 0xDC; t['b'] = 0xDF; t['e'] = 0xE7;
    t['f'] = 0xE6; t['g'] = 0xEB; t['h'] = 0xEA; t['l'] = 0xE8;
    t['r'] = 0xE9; t['u'] = 0xE5; t['w'] = 0xF5; t['x'] = 0xF2;
    t['y'] = 0xF3; t['z'] = 0xF4; t['m'] = 0xED; t['n'] = 0xEC;
    t['o'] = 0xEF; t['p'] = 0xEE; t['j'] = 0xDD; t['i'] = 0xD2;
    t['k'] = 0xDB; t['q'] = 0xD3; t['s'] = 0xF1; t['t'] = 0xF0;
    t['-'] = 0xFE; t['P'] = 0xFD;

    for (int y = 0, i = 0; y <= 30; y++) {
        for (int x = 0; x < 28; x++, i++) {
            set_tile(x, y, t[tiles[i] & 127]);
            if (tiles[i] == '.' || tiles[i] == 'P') {
```

```
                PELLET_RAM_BASE[y] |= (1 << (31 - x));
            }
        }
    }
}
```

3.3 End Game

The game ends when Pac-Man collides with any ghost, which is determined in the `check_gameover` function. This function compares Pac-Man's position to each ghost's position and checks for overlap within a single tile. If a collision is detected, the game sets the `CTRL_GAME_OVER` flag in the control register and prints a "Game Over" message to the terminal.

Once the game-over condition is triggered, the main loop pauses all movement and waits for the player to restart the game. The `game_loop` function enters a blocking loop where it listens for keyboard input. If the player presses the 'r' key, the `CTRL_RESET` flag is set, and the game exits the current loop to begin a fresh round through game_init.

This approach cleanly separates active gameplay from endgame and restart logic. While the game is in a game-over state, visual elements (Pac-Man, ghosts, tilemap) remain frozen on-screen, allowing the player to see the final frame before choosing to reset. By using control flags to manage state transitions, the design keeps logic modular and easily extensible for future features like win conditions, restart counters, or score displays.

```
bool check_gameover() {
    for (int i = 0; i < NUM_GHOSTS; i++) {
        int dx = abs(pacman_x - ghosts[i].x);
        int dy = abs(pacman_y - ghosts[i].y);
        if (dx < TILE_WIDTH && dy < TILE_HEIGHT) {
            return true;
        }
    }
    // If get all pellets
    if (score >= 2680 + 60) {
        printf("Game Over! You win!\n");
        return true;
    }
    return false;
}
```

3.4 Hardware/Software Interface

Base Address	Name	Description
0x0000 - 0x0027	Sprite Descriptors	Descriptors for sprites, including position, orientation. Max 5 sprites (1Pac-man, 4 ghosts), 8B for each.
0x0028 - 0x0029	Score Register	Current score (16 bits)
0x002A	Control Register	Signals for start/reset/pause/display sync etc.
0x002C-0x003F	Reserved Register	Reserved for future use

Figure 8: Address Map

Each sprite is described by an 8-byte structure that controls its on-screen position, animation state, and visual properties.

Byte	Description
0	X position (in pixels or tiles)
1	Y position
2	Animation frame index
3	Visibility flag (0 = invisible, 1 = visible)
4	Direction (0 = up, 1 = right, 2 = down, 3 = left)
5	Type ID (0 = Pac-Man, 1–5 = Ghosts)
6	Reserved (future use)
7	Reserved (future use)

Table 1: Sprite descriptor layout (8 bytes per sprite)

Score Register

Stores the player's current score. Supports up to 4 decimal digits (0–9999).

- **Format:** 16-bit integer
- **Access:** Written by software to update the on-screen score

Control Register

Used by software to control the hardware module state and game flow.

Bit	Name	Function
0	`Start`	Start or resume the game
1	`Reset`	Reset all game state
2	`Pause`	Pause game logic and rendering
3	`VBLANK_ACK`	Acknowledge VBlank (frame sync)
4	`Game_Over`	Set game over state (triggers audio/UI)

Table 2: Control register bit definitions

4 Discussion

4.1 Challenges

Throughout the development of our Pac-Man game, we faced several challenges spanning both software and hardware domains.

- **Ghost Movement Logic:** Designing basic ghost behavior without full pathfinding was another challenge. The ghosts needed to avoid walls and each other while still posing a threat to the player. Our simple rotation-based logic worked but required tuning to feel responsive and fair during gameplay.
- **Tilemap Accuracy and Debugging:** Accurately rendering the maze based on ASCII tilemaps required a reliable mapping system. Any misalignment in sprite positions, tile IDs, or walkability flags resulted in gameplay bugs that were sometimes difficult to trace due to the separation of logic and graphics.
- **Multimember Coordination:** Coordinating hardware, software, and documentation tasks across a distributed team also presented logistical challenges, particularly when changes in one domain (e.g., hardware memory layout) affected others (e.g., game logic structure or rendering).
- **Audio Integration:** One of the most significant challenges was getting audio to work on the FPGA. Although the board supports stereo output via the WM8731 audio CODEC, understanding the audio pipeline and its left-justified data format took time. Converting .mp3 files into the correct .wav format, writing Python scripts to translate samples into hexadecimal, and debugging the streaming logic to FIFO buffers required careful attention to audio timing and memory constraints.

4.2 Lessons Learned

This project gave us hands-on experience building a complete game system on an FPGA platform. We deepened our understanding of embedded system design by integrating real-time game logic with VGA-rendered graphics and memory-mapped I/O.

- **Designing Robust Game Logic:** Implementing Pac-Man's movement, pellet collection, ghost AI, and collision detection required careful handling of timing, game states, and memory interactions. We learned how to build reliable, frame-based logic that remained responsive to real-time input and game events.
- **Efficient Hardware Representation:** Translating a tile-based game into hardware-compatible memory structures taught us how to manage limited FPGA memory and optimize data storage. We used custom RAM layouts for pellets and tilemaps, enabling fast lookups and updates from the game logic.
- **Sprite and Tile Rendering on VGA:** Displaying dynamic elements on the VGA screen pushed us to think in terms of pixel-level graphics. We gained practical experience with timing signals, double-buffering, and how hardware modules can handle rendering efficiently without burdening the CPU.
- **Hardware/Software Synchronization:** Coordinating movement, visual updates, and control signals across the software and hardware boundary showed us the importance of consistent data formatting and well-defined interfaces. Careful use of memory-mapped registers allowed smooth integration.
- **Debugging at the System Level:** Bugs in logic or graphics often had visual symptoms on the VGA screen, helping us learn how to debug complex interactions across modules. We developed a more methodical approach to diagnosing hardware/software mismatches.
- **Team-Based Embedded Development:** Working as a team on tightly coupled hardware and software components reinforced the value of modular design, clear ownership, and version control. We learned to build around each other's progress and communicate constraints early.

4.3 Who Did What

Haoming and Tz-Jie led the development of the game logic and input controls, ensuring smooth player movement and reliable collision detection. Caiwu focused on the hardware design as well as the implementation of sprite and tile-based graphics, bringing the visual elements of the game to life on the VGA display and handling audio implementation. Emma was responsible for the documentation and converting, and integrating sound effects into the system.

5 References

1. Pixabay. "Pac-Man Sound Effects." *Pixabay*, https://pixabay.com/sound-effects/search/pac-man/. Accessed May 14, 2025.
2. Loopy. "NES Documentation." *NesDev Wiki*, https://www.nesdev.org/NESDoc.pdf. Accessed May 14, 2025.
3. Wolfson Microelectronics. "WM8731 Audio DAC Datasheet." *SparkFun Electronics*, https://cdn.sparkfun.com/datasheets/ Accessed May 14, 2025.
4. "Play Free Pac-Man Game." *FreePacman.org*, https://freepacman.org/. Accessed May 14, 2025.

Appendix

Hardware

A vga_ball.sv

```
include "ghost_up.vh"
include "ghost_down.vh"
include "ghost_left.vh"
include "ghost_right.vh"
module vga_ball (
    input clk,
    input reset,
    input [15:0] writedata,
    input write,
    input chipselect,
    input [4:0] address,

    output reg [7:0] VGA_R, VGA_G, VGA_B,
    output VGA_CLK, VGA_HS, VGA_VS,
    output VGA_BLANK_n, VGA_SYNC_n,

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

);

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

    vga_counters counters_inst (
        .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)
    );

    localparam DIR_UP = 3'd0, DIR_RIGHT = 3'd1, DIR_DOWN = 3'd2, DIR_LEFT = 3'd3, DIR_EAT = 3'd4;
    localparam SCREEN_X_OFFSET = 25 * 8; // 8 pixels per tile
    localparam SCREEN_Y_OFFSET = 14 * 8;

    reg [9:0] pacman_x;
    reg [9:0] pacman_y;
    reg [2:0] pacman_dir;
    reg [12:0] trigger_tile_index;

    reg [9:0] ghost_x[0:3];
    reg [9:0] ghost_y[0:3];
    reg [1:0] ghost_dir[0:3];

    reg gameover_latched;
```

```
    reg [25:0] gameover_wait;

    wire [6:0] pac_tile_x = pacman_x[9:3];
    wire [6:0] pac_tile_y = pacman_y[9:3];
    wire [12:0] pacman_tile_index = pac_tile_y * 80 + pac_tile_x;//

    reg [12:0] tile[0:4799];
    reg [7:0] tile_bitmaps[0:879];

    reg [16:0] score;
    reg [31:0] pacman_up[0:15], pacman_right[0:15], pacman_down[0:15], pacman_left[0:15],
        pacman_eat[0:15];

    wire [6:0] tile_x = hcount[10:4];
    wire [6:0] tile_y = vcount[9:3];
    wire [2:0] tx = hcount[3:1];
    wire [2:0] ty = vcount[2:0];

    wire [12:0] tile_index = tile_y * 80 + tile_x;
    wire [7:0] tile_id = tile[tile_index];
    wire [7:0] bitmap_row = tile_bitmaps[tile_id * 8 + ty];
    wire pixel_on = bitmap_row[7 - tx];

    wire on_pacman = (hcount[10:1] >= pacman_x && hcount[10:1] < pacman_x + 16 &&
                    vcount >= pacman_y && vcount < pacman_y + 16);

    reg [31:0] pacman_row;
    integer i, gi, gx, gy;
    integer d0, d1, d2, d3;
    integer base_tile;
    integer base_score_tile;
    reg [1:0] ghost_pixel;

    initial begin
        $readmemh("map.vh", tile);
        $readmemh("tiles.vh", tile_bitmaps);
        base_tile = 752;
    end

// ROMs for sound effects
   reg [15:0] audio_data[0:17554];   // audio.vh = pellet SFX
   reg [15:0] gameover_data[0:16532]; // gameover.vh = game over SFX

// Playback control
   reg [13:0] audio_index;
   reg [15:0] audio_sample;
   reg [15:0] sample_clock;
   reg        playing_audio;
   reg        playing_gameover;

   localparam FAST_SAMPLE_PERIOD = 285; // 50MHz / 175,550Hz


// Target playback: finish in 0.1s
localparam FAST_SAMPLE_PERIOD = 285;
initial begin
    $readmemh("audio.vh", bgm_data);
    $readmemh("gameover.vh", gameover_data);
end
```

```
    always @(posedge clk or posedge reset) begin
    if (reset) begin
        // Game reset
        gameover_latched <= 0;
        gameover_wait <= 0;
        pacman_x <= 340;
        pacman_y <= 240;
        pacman_dir <= DIR_RIGHT;
        score <= 0;
        ghost_x[0] <= 0; ghost_y[0] <= 0; ghost_dir[0] <= DIR_LEFT;
        ghost_x[1] <= 80; ghost_y[1] <= 0; ghost_dir[1] <= DIR_RIGHT;
        ghost_x[2] <= 160; ghost_y[2] <= 0; ghost_dir[2] <= DIR_UP;
        ghost_x[3] <= 250; ghost_y[3] <= 0; ghost_dir[3] <= DIR_DOWN;

        // Audio reset
        sample_clock   <= 0;
        audio_index    <= 0;
        audio_sample   <= 0;
        playing_audio  <= 0;
        playing_gameover <= 0;
        L_DATA <= 0; R_DATA <= 0;
        L_VALID <= 0; R_VALID <= 0;
    end else begin
        // === Audio playback ===
        if (sample_clock >= FAST_SAMPLE_PERIOD) begin
            sample_clock <= 0;

            // Game over sound has priority
            if (playing_gameover) begin
                audio_sample <= gameover_data[audio_index];
                if (audio_index < 9999)
                    audio_index <= audio_index + 1;
                else
                    playing_gameover <= 0;
            end
            // Pellet sound
            // Pellet sound
else if (playing_audio) begin
    if (audio_index < 17555) begin
        audio_sample <= audio_data[audio_index];
        audio_index <= audio_index + 1;
    end else begin
        audio_sample <= 16'd0;
        playing_audio <= 0;
        audio_index <= 0; // reset index for next use
    end
end else begin
                audio_sample <= 16'd0;
            end
        end else begin
            sample_clock <= sample_clock + 1;
        end

        // Avalon-ST streaming
        if (L_READY) begin
            L_DATA <= audio_sample;
            L_VALID <= (sample_clock == 0);
        end else begin
            L_VALID <= 0;
```

```
end

if (R_READY) begin
    R_DATA <= audio_sample;
    R_VALID <= (sample_clock == 0);
end else begin
    R_VALID <= 0;
end

// === Register write logic ===
if (chipselect && write) begin
    case (address[4:0])
        6'h00: begin pacman_x <= writedata[7:0]; pacman_y <= writedata[15:8]; end
        6'h02: pacman_dir <= writedata[2:0];
        6'h04: begin ghost_x[0] <= writedata[7:0]; ghost_y[0] <= writedata[15:8]; end
        6'h06: ghost_dir[0] <= writedata[1:0];
        6'h08: begin ghost_x[1] <= writedata[7:0]; ghost_y[1] <= writedata[15:8]; end
        6'h0A: ghost_dir[1] <= writedata[1:0];
        6'h0C: begin ghost_x[2] <= writedata[7:0]; ghost_y[2] <= writedata[15:8]; end
        6'h0E: ghost_dir[2] <= writedata[1:0];
        6'h10: begin ghost_x[3] <= writedata[7:0]; ghost_y[3] <= writedata[15:8]; end
        6'h12: ghost_dir[3] <= writedata[1:0];
        6'h14: score <= writedata;
        6'h15: begin
            if (writedata[7:0] == 8'b0) begin
                $readmemh("map.vh", tile);
                score <= 0;
                pacman_x <= 340;
                pacman_y <= 240;
                pacman_dir <= DIR_RIGHT;
                ghost_dir[0] <= DIR_LEFT;
                ghost_dir[1] <= DIR_RIGHT;
                ghost_dir[2] <= DIR_UP;
                ghost_dir[3] <= DIR_DOWN;
                ghost_x[0] <= 100; ghost_y[0] <= 100;
                ghost_x[1] <= 200; ghost_y[1] <= 100;
                ghost_x[2] <= 300; ghost_y[2] <= 100;
                ghost_x[3] <= 400; ghost_y[3] <= 100;
            end else if (writedata[4]) begin
                gameover_latched <= 1;
            end
        end
        6'h16: trigger_tile_index <= writedata;
    endcase
end

// === Game Over display and sound trigger ===
if (gameover_latched) begin
    // Draw "GAME OVER"
    tile[3795 + 0] <= 38 + (6 * 2); // G
    tile[3795 + 1] <= 38 + (0 * 2); // A
    tile[3795 + 2] <= 38 + (12 * 2); // M
    tile[3795 + 3] <= 38 + (4 * 2); // E
    tile[3795 + 4] <= 12'h25;
    tile[3795 + 5] <= 38 + (14 * 2); // O
    tile[3795 + 6] <= 38 + (21 * 2); // V
    tile[3795 + 7] <= 38 + (4 * 2); // E
    tile[3795 + 8] <= 38 + (17 * 2); // R

    tile[3875 + 0] <= 38 + (6 * 2) + 1;
```

```
            tile[3875 + 1] <= 38 + (0 * 2) + 1;
            tile[3875 + 2] <= 38 + (12 * 2) + 1;
            tile[3875 + 3] <= 38 + (4 * 2) + 1;
            tile[3875 + 4] <= 12'h25;
            tile[3875 + 5] <= 38 + (14 * 2) + 1;
            tile[3875 + 6] <= 38 + (21 * 2) + 1;
            tile[3875 + 7] <= 38 + (4 * 2) + 1;
            tile[3875 + 8] <= 38 + (17 * 2) + 1;

            if (!playing_gameover) begin
                playing_gameover <= 1;
                audio_index <= 0;
            end

            gameover_wait <= gameover_wait + 1;
            if (gameover_wait == 50_000_000) begin
                $readmemh("map.vh", tile);
                score <= 0;
                pacman_x <= 340;
                pacman_y <= 240;
                pacman_dir <= DIR_RIGHT;
                ghost_dir[0] <= DIR_LEFT;
                ghost_dir[1] <= DIR_RIGHT;
                ghost_dir[2] <= DIR_UP;
                ghost_dir[3] <= DIR_DOWN;
                ghost_x[0] <= 100; ghost_y[0] <= 100;
                ghost_x[1] <= 200; ghost_y[1] <= 100;
                ghost_x[2] <= 300; ghost_y[2] <= 100;
                ghost_x[3] <= 400; ghost_y[3] <= 100;
                gameover_latched <= 0;
                gameover_wait <= 0;
            end
        end
    tile[base_tile + 0] = 38 + (18 * 2);
        tile[base_tile + 1] = 38 + (2 * 2);
        tile[base_tile + 2] = 38 + (14 * 2);
        tile[base_tile + 3] = 38 + (17 * 2);
        tile[base_tile + 4] = 38 + (4 * 2);
        tile[base_tile + 80] = 38 + (18 * 2) + 1;
        tile[base_tile + 81] = 38 + (2 * 2) + 1;
        tile[base_tile + 82] = 38 + (14 * 2) + 1;
        tile[base_tile + 83] = 38 + (17 * 2) + 1;
        tile[base_tile + 84] = 38 + (4 * 2) + 1;

        d3 = score[15:12];
        d2 = score[11:8];
        d1 = score[7:4];
        d0 = score[3:0];

        base_score_tile = 761;
        tile[base_score_tile + 0] = 38 + (26 * 2) + d3 * 2;
        tile[base_score_tile + 1] = 38 + (26 * 2) + d2 * 2;
        tile[base_score_tile + 2] = 38 + (26 * 2) + d1 * 2;
        tile[base_score_tile + 3] = 38 + (26 * 2) + d0 * 2;
        tile[base_score_tile + 80] = 38 + (26 * 2) + d3 * 2 + 1;
        tile[base_score_tile + 81] = 38 + (26 * 2) + d2 * 2 + 1;
        tile[base_score_tile + 82] = 38 + (26 * 2) + d1 * 2 + 1;
        tile[base_score_tile + 83] = 38 + (26 * 2) + d0 * 2 + 1;

        // === Pellet eaten   clear tile + play sound ===
```

```
        if (trigger_tile_index != 13'd65535) begin
            tile[trigger_tile_index] <= 8'h25;

            if (!playing_audio && !playing_gameover) begin
                playing_audio <= 1;
                audio_index <= 0;
            end

            trigger_tile_index <= 13'd65535; // reset trigger
        end
    end
end

    initial begin
        $readmemh("pacman_up.vh", pacman_up);
        $readmemh("pacman_right.vh", pacman_right);
        $readmemh("pacman_down.vh", pacman_down);
        $readmemh("pacman_left.vh", pacman_left);
   $readmemh("pacman_eat.vh", pacman_eat);
    end

always @(*) begin
    VGA_R = 0; VGA_G = 0; VGA_B = 0;

    if (pixel_on) begin
        if (tile_id == 8'h0A || tile_id >= 8'h26 || tile_id == 8'h14)
            {VGA_R, VGA_G, VGA_B} = 24'hFFFFFF;
        else
            {VGA_R, VGA_G, VGA_B} = 24'h0000FF;
    end

    pacman_row = 32'b0;
    if (vcount >= pacman_y + SCREEN_Y_OFFSET &&
        vcount < pacman_y + SCREEN_Y_OFFSET + 16) begin
        case (pacman_dir)
            DIR_UP:   pacman_row = pacman_up[vcount - pacman_y - SCREEN_Y_OFFSET];
            DIR_RIGHT: pacman_row = pacman_right[vcount - pacman_y - SCREEN_Y_OFFSET];
            DIR_DOWN: pacman_row = pacman_down[vcount - pacman_y - SCREEN_Y_OFFSET];
            DIR_LEFT: pacman_row = pacman_left[vcount - pacman_y - SCREEN_Y_OFFSET];
            DIR_EAT:  pacman_row = pacman_eat[vcount - pacman_y - SCREEN_Y_OFFSET];
        endcase
    end

    if (hcount[10:1] >= pacman_x + SCREEN_X_OFFSET &&
        hcount[10:1] < pacman_x + SCREEN_X_OFFSET + 16 &&
        vcount >= pacman_y + SCREEN_Y_OFFSET &&
        vcount < pacman_y + SCREEN_Y_OFFSET + 16 &&
        pacman_row[15 - (hcount[10:1] - pacman_x - SCREEN_X_OFFSET)]) begin
        VGA_R = 8'hFF;
        VGA_G = 8'hFF;
        VGA_B = 8'h00;
    end

    for (gi = 0; gi < 4; gi = gi + 1) begin
        if (hcount[10:1] >= ghost_x[gi] + SCREEN_X_OFFSET &&
            hcount[10:1] < ghost_x[gi] + SCREEN_X_OFFSET + 16 &&
            vcount >= ghost_y[gi] + SCREEN_Y_OFFSET &&
            vcount < ghost_y[gi] + SCREEN_Y_OFFSET + 16) begin

            gx = hcount[10:1] - ghost_x[gi] - SCREEN_X_OFFSET;
```

```
            gy = vcount - ghost_y[gi] - SCREEN_Y_OFFSET;

            case (ghost_dir[gi])
                DIR_UP:   ghost_pixel = GHOST_UP[gy][gx];
                DIR_DOWN: ghost_pixel = GHOST_DOWN[gy][gx];
                DIR_LEFT: ghost_pixel = GHOST_LEFT[gy][gx];
                DIR_RIGHT: ghost_pixel = GHOST_RIGHT[gy][gx];
                default:  ghost_pixel = 2'b00;
            endcase

            case (ghost_pixel)
                2'b01: case (gi)
                    0: begin VGA_R = 8'hFF; VGA_G = 0; VGA_B = 0;    end
                    1: begin VGA_R = 8'hFF; VGA_G = 8'hAA; VGA_B = 8'hFF; end
                    2: begin VGA_R = 8'hFF; VGA_G = 8'hAA; VGA_B = 0; end
                    3: begin VGA_R = 0;   VGA_G = 8'hFF; VGA_B = 8'hFF; end
                endcase
                2'b10: begin VGA_R = 8'hFF; VGA_G = 8'hFF; VGA_B = 8'hFF; end
                2'b11: begin VGA_R = 0;   VGA_G = 0;    VGA_B = 8'h88; end
            endcase
        end
    end
end
endmodule



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

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

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

```
  logic endOfLine;

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

  assign endOfLine = hcount == HTOTAL - 1;

  logic endOfField;

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

  assign endOfField = vcount == VTOTAL - 1;

  // Horizontal sync: from 0x520 to 0x5DF (0x57F)
  // 101 0010 0000 to 101 1101 1111 (active LOW during 1312-1503) (192 cycles)
  assign VGA_HS = !( (hcount[10:8] == 3'b101) & !(hcount[7:5] == 3'b111));
  assign VGA_VS = !( vcount[9:1] == (VACTIVE + VFRONT_PORCH) / 2);

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

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

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

endmodule
```

Software

B main.c

```
#include <stdio.h>
#include "vga_ball.h"
#include "usbkeyboard.h"
#include <sys/ioctl.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <libusb-1.0/libusb.h>

#define WIDTH 640
#define HEIGHT 480
```

```c
#include <stdint.h>
#include <stdbool.h>

#include <string.h>
#include <libusb-1.0/libusb.h>

#include "controller.h"


#define INPUT_BUFFER_SIZE 256
#define CHAT_COLS 64
#define INPUT_ROWS 2
#define MAX_KEYS 6

#define CTRL_START     (1 << 0)
#define CTRL_RESET     (1 << 1)
#define CTRL_PAUSE     (1 << 2)
#define CTRL_VBLANK_ACK (1 << 3)
#define CTRL_GAME_OVER (1 << 4)

#define PELLET_NONE 0xFFFF
uint16_t last_pellet_index = PELLET_NONE;

typedef struct {
    uint8_t x, y;
    uint8_t frame;
    uint8_t visible;
    uint8_t direction;
    uint8_t type_id;
    uint8_t rsv1, rsv2;
} sprite_t;



// === USB   Keyboard   ===
struct usb_keyboard_packet packet;
struct libusb_device_handle *keyboard;
uint8_t endpoint_address;

//      ASCII
char usb_to_ascii(uint8_t keycode, uint8_t modifiers) {
    static char ascii_map[256] = {0};
    ascii_map[0x04] = 'a'; ascii_map[0x05] = 'b'; ascii_map[0x06] = 'c';
    ascii_map[0x07] = 'd'; ascii_map[0x08] = 'e'; ascii_map[0x09] = 'f';
    ascii_map[0x0A] = 'g'; ascii_map[0x0B] = 'h'; ascii_map[0x0C] = 'i';
    ascii_map[0x0D] = 'j'; ascii_map[0x0E] = 'k'; ascii_map[0x0F] = 'l';
    ascii_map[0x10] = 'm'; ascii_map[0x11] = 'n'; ascii_map[0x12] = 'o';
    ascii_map[0x13] = 'p'; ascii_map[0x14] = 'q'; ascii_map[0x15] = 'r';
    ascii_map[0x16] = 's'; ascii_map[0x17] = 't'; ascii_map[0x18] = 'u';
    ascii_map[0x19] = 'v'; ascii_map[0x1A] = 'w'; ascii_map[0x1B] = 'x';
    ascii_map[0x1C] = 'y'; ascii_map[0x1D] = 'z';
    ascii_map[0x1E] = '1'; ascii_map[0x1F] = '2'; ascii_map[0x20] = '3';
    ascii_map[0x21] = '4'; ascii_map[0x22] = '5'; ascii_map[0x23] = '6';
    ascii_map[0x24] = '7'; ascii_map[0x25] = '8'; ascii_map[0x26] = '9';
    ascii_map[0x27] = '0';
    ascii_map[0x2C] = ' '; //
    ascii_map[0x28] = '\n'; // Enter
    ascii_map[0x2A] = '\b'; // Backspace
    ascii_map[0x50] = 2;  // Left arrow
    ascii_map[0x4F] = 3;  // Right arrow
```

```
    ascii_map[0x52] = 4;  // Up arrow
    ascii_map[0x51] = 5;  // Down arrow

    if (modifiers & USB_LSHIFT || modifiers & USB_RSHIFT) {
        if (keycode >= 0x04 && keycode <= 0x1D) {
            return ascii_map[keycode] - 32; //
        }
    }
    return ascii_map[keycode];
}

//
void handle_input(char c) {
    extern uint8_t pacman_dir;
    if (c == 2) { // left
        pacman_dir = 1;
    } else if (c == 3) { // right
        pacman_dir = 3;
    } else if (c == 4) { // up
        pacman_dir = 0;
    } else if (c == 5) { // down
        pacman_dir = 2;
    }
}



typedef struct {
    int x, y;
    uint8_t dir;
    sprite_t* sprite;
} ghost_t;

#define TILE_WIDTH 8
#define TILE_HEIGHT 8
#define SCREEN_WIDTH_TILES 28
#define SCREEN_HEIGHT_TILES 33
#define NUM_GHOSTS 4
#define SPRITE_PACMAN 0
#define SPRITE_GHOST_0 1
#define SPRITE_GHOST_1 2
#define SPRITE_GHOST_2 3
#define SPRITE_GHOST_3 4


//   RAM
uint8_t fake_tilemap[40 * 30];
uint32_t fake_pellet_ram[30];
sprite_t fake_sprites[5];
uint16_t fake_score;
uint8_t fake_control = 1;

#define TILEMAP_BASE   (fake_tilemap)
#define PELLET_RAM_BASE (fake_pellet_ram)
#define SPRITE_BASE    ((uint8_t*) fake_sprites)
#define SCORE_REG      (&fake_score)
#define CONTROL_REG    (&fake_control)

ghost_t ghosts[NUM_GHOSTS];
sprite_t* sprites = (sprite_t*) SPRITE_BASE;
```

```
#define PACMAN_INIT_X 1 * TILE_WIDTH + TILE_WIDTH / 2
#define PACMAN_INIT_Y 1 * TILE_HEIGHT + TILE_HEIGHT / 2
int pacman_x = PACMAN_INIT_X;
int pacman_y = PACMAN_INIT_Y;
uint8_t pacman_dir = 1; //
uint16_t score = 0;


void set_control_flag(uint8_t flag) {
    *CONTROL_REG |= flag;
}

void clear_control_flag(uint8_t flag) {
    *CONTROL_REG &= ~flag;
}
void clear_all_control_flags() {
    *CONTROL_REG = 0;
}

bool is_control_flag_set(uint8_t flag) {
    return (*CONTROL_REG & flag) != 0;
}

uint8_t direction_to_control(uint8_t dir) {
    switch (dir) {
        case 0: return 0; // up
        case 1: return 3; // left
        case 2: return 2; // down
        case 3: return 1; // right
        case 5: return 4; // eat pellet
        default: return 0;
    }
}
int vga_ball_fd;

void update_all_to_driver() {
    vga_all_state_t state;

    //            sprite
    for (int i = 0; i < 5; i++) {
        // state.sprites[i] = fake_sprites[i];
        state.sprites[i].x = sprites[i].x ;
        state.sprites[i].y = sprites[i].y ;
        state.sprites[i].frame = sprites[i].frame;
        state.sprites[i].visible = sprites[i].visible;
        state.sprites[i].direction = direction_to_control(sprites[i].direction);
        state.sprites[i].type_id = sprites[i].type_id;
        state.sprites[i].reserved1 = sprites[i].rsv1;
        state.sprites[i].reserved2 = sprites[i].rsv2;
    }

    state.score = * SCORE_REG;
    state.control = *CONTROL_REG;

    // state.pellet_to_eat = last_pellet_index;

    if (last_pellet_index != PELLET_NONE) {
        int pellet_x = last_pellet_index % SCREEN_WIDTH_TILES;
        int pellet_y = last_pellet_index / SCREEN_WIDTH_TILES;
        state.pellet_to_eat = pellet_y * 80 + pellet_x + 6 + 1220;
```

```
    } else {
        state.pellet_to_eat = 0xFFFF; //
    }

    last_pellet_index = PELLET_NONE;

    if (ioctl(vga_ball_fd, VGA_BALL_WRITE_ALL, &state)) {
        perror("ioctl(VGA_BALL_WRITE_ALL) failed");
    }



    // printf all content for debugging
    printf("=== State Debug ===\n");
    printf("Score: %d\n", state.score);
    printf("Control: 0x%02X\n", state.control);
    for (int i = 0; i < 5; i++) {
        printf("Sprite %d: x=%d, y=%d, frame=%d, visible=%d, direction=%d, type_id=%d\n",
               i,
               state.sprites[i].x,
               state.sprites[i].y,
               state.sprites[i].frame,
               state.sprites[i].visible,
               state.sprites[i].direction,
               state.sprites[i].type_id);
    }
    printf("Pellet to eat: %d\n", state.pellet_to_eat);
    printf("====================\n");
}

//
void set_tile(int x, int y, uint8_t tile_id) {
    TILEMAP_BASE[y * SCREEN_WIDTH_TILES + x] = tile_id;
}

uint8_t get_pellet_bit(int x, int y) {
    return (PELLET_RAM_BASE[y] >> (31 - x)) & 1;
}

void clear_pellet_bit(int x, int y) {
    PELLET_RAM_BASE[y] &= ~(1 << (31 - x));
}

bool can_move_to(int px, int py) {
    int tx = px / TILE_WIDTH;
    int ty = py / TILE_HEIGHT;
    // if (tx < 0 || tx >= 40 || ty < 0 || ty >= 30) return false;
    uint8_t tile = TILEMAP_BASE[ty * SCREEN_WIDTH_TILES + tx];
    if (tile == 0x40 || tile == 1 || tile == 0) {

    }
    else{
        printf("Tile at (%d, %d) is not walkable: %d\n", tx, ty, tile);
    }
    return (tile == 0x40 || tile == 1 || tile == 0 || tile == 0xFD);
}
const int step_size = TILE_HEIGHT ;

uint16_t generate_packed_score(uint16_t score) {
    int s = score % 10000; //       4
```

```
    return ((s / 1000) << 12) |
           (((s / 100) % 10) << 8) |
           (((s / 10) % 10) << 4) |
           (s % 10);
}

void update_pacman() {
    int new_x = pacman_x;
    int new_y = pacman_y;
    switch (pacman_dir) {
        case 0: new_y -= step_size ; break;
        case 1: new_x -= step_size ; break;
        case 2: new_y += step_size ; break;
        case 3: new_x += step_size ; break;
    }

    if (can_move_to(new_x, new_y)) {
        pacman_x = new_x;
        pacman_y = new_y;
    }

    int tile_x = pacman_x / TILE_WIDTH;
    int tile_y = pacman_y / TILE_HEIGHT;

    if (get_pellet_bit(tile_x, tile_y)) {
        clear_pellet_bit(tile_x, tile_y);
        set_tile(tile_x, tile_y, 0);
        pacman_dir = 5; // Eat pellet animation direction
        score += 10;
        *SCORE_REG = generate_packed_score(score);
        last_pellet_index = tile_y * SCREEN_WIDTH_TILES + tile_x;
        printf("[Pac-Man] Ate pellet at (%d, %d). Score = %d\n", tile_x, tile_y, score);
    } else {
        last_pellet_index = PELLET_NONE;
    }
    sprite_t* pac = &sprites[SPRITE_PACMAN];


    // TP
    if (pacman_x <= 2 * TILE_WIDTH && pacman_y == 14 * TILE_HEIGHT + TILE_HEIGHT / 2){
        pacman_x = 25 * TILE_WIDTH + TILE_WIDTH / 2;
        printf("Teleport to (%d, %d)\n", pacman_x, pacman_y);
    }
    else if (pacman_x >= 26 * TILE_WIDTH && pacman_y == 14 * TILE_HEIGHT + TILE_HEIGHT / 2){
        pacman_x = 2 * TILE_WIDTH + TILE_WIDTH / 2;
        printf("Teleport to (%d, %d)\n", pacman_x, pacman_y);
    }

    pac->x = pacman_x;
    pac->y = pacman_y;
    pac->visible = 1;
    pac->direction = pacman_dir;
    pac->frame = 0;
}

bool is_ghost_tile(int x, int y) {
    for (int i = 0; i < NUM_GHOSTS; i++) {
        if (ghosts[i].x / TILE_WIDTH == x && ghosts[i].y / TILE_HEIGHT == y) return true;
    }
    return false;
```

```
}

void print_tilemap() {
    printf("=== Tilemap View ===\n");
    for (int y = 0; y <= 30; y++) {
        for (int x = 0; x < 28; x++) {
            uint8_t tile = TILEMAP_BASE[y * SCREEN_WIDTH_TILES + x];
            if (pacman_x / TILE_WIDTH == x && pacman_y / TILE_HEIGHT == y) {
                putchar('P');
            } else if (is_ghost_tile(x, y)) {
                putchar('G');
            } else if (tile == 0 || tile == 0x40) {
                putchar(' ');
            } else if (tile == 1) {
                putchar('.');
            } else {
                putchar('#');
            }
        }
        putchar('\n');
    }
    printf("====================\n");
}



bool is_tile_occupied_by_other_ghost(int gx, int gy, int self_index) {
    for (int i = 0; i < NUM_GHOSTS; i++) {
        if (i == self_index) continue; //
        int other_tx = ghosts[i].x / TILE_WIDTH;
        int other_ty = ghosts[i].y / TILE_HEIGHT;
        if (other_tx == gx && other_ty == gy) {
            return true;
        }
    }
    return false;
}
#include <stdlib.h> // for rand()

#define EPSILON 0.2f // 20%

//       Manhattan
int manhattan_distance(int x1, int y1, int x2, int y2) {
    return abs(x1 - x2) + abs(y1 - y2);
}

//                       [0, 1, 2, 3]
int random_direction() {
    return rand() % 4;
}

void update_ghosts() {
    static int ghost_tick = 0;
    ghost_tick++;
    if (ghost_tick % 10 != 0) return;

    for (int i = 0; i < NUM_GHOSTS; i++) {
        ghost_t* g = &ghosts[i];
        int best_dir = g->dir;
        int best_dist = 1 << 30; //
```

```
//       tile
int curr_tx = g->x / TILE_WIDTH;
int curr_ty = g->y / TILE_HEIGHT;

//                    tile
int target_x = pacman_x, target_y = pacman_y;

if (i == 0) {
    // Blinky:      Pac-Man
    target_x = pacman_x;
    target_y = pacman_y;
} else if (i == 1) {
    // Pinky:       Pac-Man       4
    target_x = pacman_x;
    target_y = pacman_y;
    switch (pacman_dir) {
        case 0: target_y -= TILE_HEIGHT * 4; break;
        case 1: target_x -= TILE_WIDTH * 4; break;
        case 2: target_y += TILE_HEIGHT * 4; break;
        case 3: target_x += TILE_WIDTH * 4; break;
    }
} else if (i == 2) {
    // Inky:       Blinky    Pac-Man
    int bx = ghosts[0].x;
    int by = ghosts[0].y;
    int ref_x = pacman_x, ref_y = pacman_y;
    switch (pacman_dir) {
        case 0: ref_y -= TILE_HEIGHT * 2; break;
        case 1: ref_x -= TILE_WIDTH * 2; break;
        case 2: ref_y += TILE_HEIGHT * 2; break;
        case 3: ref_x += TILE_WIDTH * 2; break;
    }
    target_x = ref_x + (ref_x - bx);
    target_y = ref_y + (ref_y - by);
} else if (i == 3) {
    // Clyde:         Pac-Man
    int d = abs(g->x - pacman_x) + abs(g->y - pacman_y);
    if (d < TILE_WIDTH * 8) {
        target_x = 0;
        target_y = 0; //
    }
}

//
for (int d = 0; d < 4; d++) {
    int dx = 0, dy = 0;
    switch (d) {
        case 0: dy = -step_size; break;
        case 1: dx = -step_size; break;
        case 2: dy = +step_size; break;
        case 3: dx = +step_size; break;
    }

    int new_x = g->x + dx;
    int new_y = g->y + dy;
    int tx = new_x / TILE_WIDTH;
    int ty = new_y / TILE_HEIGHT;

    if (!can_move_to(new_x, new_y) || is_tile_occupied_by_other_ghost(tx, ty, i)) continue;
```

```
            int dist = abs(new_x - target_x) + abs(new_y - target_y);

            //  -greedy     :
            if (rand() % 100 < 20) { // 20%
                best_dir = d;
                break;
            }

            if (dist < best_dist) {
                best_dist = dist;
                best_dir = d;
            }
        }

        //
        g->dir = best_dir;
        switch (best_dir) {
            case 0: g->y -= step_size; break;
            case 1: g->x -= step_size; break;
            case 2: g->y += step_size; break;
            case 3: g->x += step_size; break;
        }

        g->sprite->x = g->x;
        g->sprite->y = g->y;
    }
}


void game_init_playfield(void) {
    static const char* tiles =
      //01234567890123456789012345678
       "0UUUUUUUUUUUU45UUUUUUUUUUUU1" // 3
       "L............rl............R" // 4
       "L.ebbf.ebbbf.rl.ebbbf.ebbf.R" // 5
       "L.r  l.r   l.rl.r   l.r  l.R" // 6
       "L.guuh.guuuh.gh.guuuh.guuh.R" // 7
       "L.........................R" // 8
       "L.ebbf.ef.ebbbbbbf.ef.ebbf.R" // 9
       "L.guuh.rl.guuyxuuh.rl.guuh.R" // 10
       "L......rl....rl....rl......R" // 11
       "2BBBBf.rzbbf.rl.ebbwl.eBBBB3" // 12
       "     L.rxuuh.gh.guuyl.R    " // 13
       "     L.rl..........rl.R    " // 14
       "     L.rl.mjs tjn.rl.R     " // 15
       "UUUUUh.gh.i     q.gh.gUUUUU" // 16
       "       .   i      q   .     " // 17
       "BBBBBf.ef.i     q.ef.eBBBBB" // 18
       "     L.rl.okkkkkkp.rl.R    " // 19
       "     L.rl.        .rl.R    " // 20
       "     L.rl.ebbbbbbf.rl.R    " // 21
       "0UUUUh.gh.guuyxuuh.gh.gUUUU1" // 22
       "L............rl............R" // 23
       "L.ebbf.ebbbf.rl.ebbbf.ebbf.R" // 24
       "L.guyl.guuuh.gh.guuuh.rxuh.R" // 25
       "L...rl................rl...R" // 26
       "6bf.rl.ef.ebbbbbbf.ef.rl.eb8" // 27
       "7uh.gh.rl.guuyxuuh.rl.gh.gu9" // 28
       "L......rl....rl....rl......R" // 29
```

```
        "L.ebbbbwzbbf.rl.ebbwzbbbbf.R" // 30
        "L.guuuuuuuuh.gh.guuuuuuuuh.R" // 31
        "L.........................R" // 32
        "2BBBBBBBBBBBBBBBBBBBBBBBBB3"; // 33
       //01234567890123456789012345678

    uint8_t t[128];
    for (int i = 0; i < 128; i++) t[i] = 1;
    t[' '] = 0x40; t['0'] = 0xD1; t['1'] = 0xD0; t['2'] = 0xD5; t['3'] = 0xD4;
    t['4'] = 0xFB; t['5'] = 0xFA; t['6'] = 0xD7; t['7'] = 0xD9;
    t['8'] = 0xD6; t['9'] = 0xD8; t['U'] = 0xDB; t['L'] = 0xD3;
    t['R'] = 0xD2; t['B'] = 0xDC; t['b'] = 0xDF; t['e'] = 0xE7;
    t['f'] = 0xE6; t['g'] = 0xEB; t['h'] = 0xEA; t['l'] = 0xE8;
    t['r'] = 0xE9; t['u'] = 0xE5; t['w'] = 0xF5; t['x'] = 0xF2;
    t['y'] = 0xF3; t['z'] = 0xF4; t['m'] = 0xED; t['n'] = 0xEC;
    t['o'] = 0xEF; t['p'] = 0xEE; t['j'] = 0xDD; t['i'] = 0xD2;
    t['k'] = 0xDB; t['q'] = 0xD3; t['s'] = 0xF1; t['t'] = 0xF0;
    t['-'] = 0xFE; t['P'] = 0xFD;

    for (int y = 0, i = 0; y <= 30; y++) {
        for (int x = 0; x < 28; x++, i++) {
            set_tile(x, y, t[tiles[i] & 127]);
            if (tiles[i] == '.' || tiles[i] == 'P') {
                PELLET_RAM_BASE[y] |= (1 << (31 - x));
            }
        }
    }
}
void init_ghosts() {
    const int start_positions[4][2] = {
        {13, 14}, {14, 14}, {13, 15}, {14, 15}

    };
    for (int i = 0; i < NUM_GHOSTS; i++) {
        ghosts[i].x = start_positions[i][0] * TILE_WIDTH + TILE_WIDTH / 2;
        ghosts[i].y = start_positions[i][1] * TILE_HEIGHT + TILE_HEIGHT / 2;
        ghosts[i].dir = i % 4;
        ghosts[i].sprite = &sprites[SPRITE_GHOST_0 + i];
        ghost_t* g = &ghosts[i];
        g->sprite->x = g->x;
        g->sprite->y = g->y;
        g->sprite->visible = 1;
        g->sprite->frame = 0;
    }
}


bool paused = false;

void process_control_keys(char c) {
    if (c == ' ') { //
        paused = !paused;
        if (paused) {
            set_control_flag(CTRL_PAUSE);
            printf("Game paused.\n");
        } else {
            clear_control_flag(CTRL_PAUSE);
            printf("Game resumed.\n");
        }
    } else if (c == 'r' || c == 'R') {
```

```
        set_control_flag(CTRL_RESET);
        printf("Game reset signal sent.\n");
        //                      game_loop()        main
        exit(0);
    }
}
bool check_collision() {
    for (int i = 0; i < NUM_GHOSTS; i++) {
        if ((ghosts[i].x / TILE_WIDTH == pacman_x / TILE_WIDTH) &&
            (ghosts[i].y / TILE_HEIGHT == pacman_y / TILE_HEIGHT)) {
            return true;
        }
    }
    return false;
}

void game_init() {
    *SCORE_REG = 0;
    *CONTROL_REG = 0;
    // fake_control = CTRL_START;
    game_init_playfield();
    init_ghosts();
    sprite_t* pac = &sprites[SPRITE_PACMAN];
    pacman_x = PACMAN_INIT_X;
    pacman_y = PACMAN_INIT_Y;
    pac->x = pacman_x;
    pac->y = pacman_y;
    pac->visible = 1;
    pac->frame = 0;
    clear_all_control_flags();
    update_all_to_driver();

}


int abs(int value) {
    return value < 0 ? -value : value;
}
bool check_gameover() {
    for (int i = 0; i < NUM_GHOSTS; i++) {
        int dx = abs(pacman_x - ghosts[i].x);
        int dy = abs(pacman_y - ghosts[i].y);
        if (dx < TILE_WIDTH && dy < TILE_HEIGHT) {
            return true;
        }
    }
    //  score
    if (score >= 2680 + 60) {
        printf("Game Over! You win!\n");
        return true;
    }
    return false;
}


int main() {
    struct controller_list controller = open_controller();
    uint8_t endpoint_address = controller.device1_addr;
    struct libusb_device_handle* device = controller.device1;
    struct controller_pkt packet;
```

```
vga_ball_fd = open("/dev/vga_ball", O_RDWR);
if (vga_ball_fd == -1) {
    perror("Failed to open /dev/vga_ball");
    return 1;
}

printf("Controller connected. Starting Pac-Man...\n");

while (1) {
    game_init();
    printf("Waiting for START (A button)...\n");

    int transferred;
    while (!(is_control_flag_set(CTRL_START))) {
        if (libusb_interrupt_transfer(device, endpoint_address,
                                (unsigned char*)&packet, sizeof(packet),
                                &transferred, 0) == 0 && transferred == 7) {
            if (packet.ab & 0x20) { // A button
                set_control_flag(CTRL_START);
                break;
            }
        }
        usleep(10000);
    }

    printf("Game started. Use controller to play.\n");

    while (1) {
        if (libusb_interrupt_transfer(device, endpoint_address,
                                (unsigned char*)&packet, sizeof(packet),
                                &transferred, 0) == 0 && transferred == 7) {

            if (packet.dir_x == 0x00) pacman_dir = 1; // left
            if (packet.dir_x == 0xff) pacman_dir = 3; // right
            if (packet.dir_y == 0x00) pacman_dir = 0; // up
            if (packet.dir_y == 0xff) pacman_dir = 2; // down

            if (packet.ab & 0x10) {
                *CONTROL_REG ^= CTRL_PAUSE;
                usleep(200000); // debounce
            }
        }

        if (*CONTROL_REG & CTRL_PAUSE) {
            usleep(100000);
            continue;
        }

        update_pacman();
        update_ghosts();

        if (check_gameover()) {
            *CONTROL_REG |= CTRL_GAME_OVER;
            update_all_to_driver();
            printf("[Game] Game Over! Press A to restart.\n");
            break;
        }

        update_all_to_driver();
```

```
            usleep(100000);
        }

        while (!(*CONTROL_REG & CTRL_RESET)) {
            if (libusb_interrupt_transfer(device, endpoint_address,
                                          (unsigned char*)&packet, sizeof(packet),
                                          &transferred, 0) == 0 && transferred == 7) {
                if (packet.ab & 0x20) { // A again = restart
                    *CONTROL_REG |= CTRL_RESET;
                }
            }
            usleep(100000);
        }
    }

    libusb_close(device);
    libusb_exit(NULL);
    close(vga_ball_fd);
    return 0;
}
```

C controller.c

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

struct controller_list open_controller() {
    printf("Searching for USB controller...\n");

    struct controller_list device;
    libusb_device **devs;
    struct libusb_device_descriptor desc;
    struct libusb_device_handle *controller = NULL;
    ssize_t num_devs;

    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 USB devices found");
        exit(1);
    }

    for (int i = 0; i < num_devs; i++) {
        libusb_device *dev = devs[i];

        if (libusb_get_device_descriptor(dev, &desc) < 0)
            continue;

        if (desc.idProduct == 0x11) {
            struct libusb_config_descriptor *config;
            if ((libusb_get_config_descriptor(dev, 0, &config)) < 0)
                continue;
```

```
            const struct libusb_interface_descriptor *inter = config->interface[0].altsetting;
            if (libusb_open(dev, &controller) != 0)
                continue;

            if (libusb_kernel_driver_active(controller, 0))
                libusb_detach_kernel_driver(controller, 0);
            libusb_set_auto_detach_kernel_driver(controller, 0);

            if (libusb_claim_interface(controller, 0) != 0)
                continue;

            device.device1 = controller;
            device.device1_addr = inter->endpoint[0].bEndpointAddress;
            libusb_free_device_list(devs, 1);

            printf("Controller connected.\n");
            return device;
        }
    }

    printf("ERROR: couldn't find a controller.\n");
    exit(1);
}

void detect_presses(struct controller_pkt pkt, char *buttons, int mode) {
    char vals[] = "LRUDAXY";
    if (mode == 1) {
        strcpy(buttons, "11111");
    } else {
        strcpy(buttons, "_____");
    }

    if (pkt.dir_x == 0x00) buttons[0] = vals[0]; // Left
    if (pkt.dir_x == 0xff) buttons[1] = vals[1]; // Right
    if (pkt.dir_y == 0x00) buttons[2] = vals[2]; // Up
    if (pkt.dir_y == 0xff) buttons[3] = vals[3]; // Down

    if ((pkt.ab & 0x20)) {
        buttons[4] = vals[4]; // A pressed
    }
    if ((pkt.ab & 0x10)) {
        buttons[5] = vals[5]; // X pressed
    }


}

void *listen_controller(void *arg) {
    struct args_list *args_p = arg;
    struct args_list args = *args_p;
    struct controller_pkt pkt;
    int transferred;
    int size = sizeof(pkt);
    char buttons[6] = "_____";

    while (1) {
        libusb_interrupt_transfer(
            args.devices.device1,
            args.devices.device1_addr,
```

```
            (unsigned char *)&pkt,
            size,
            &transferred,
            0
        );

        if (transferred == 7) {
            detect_presses(pkt, buttons, args.mode);
            strcpy(args.buttons, buttons);
            if (args.print) {
                printf("%s\n", args.buttons);
            }
        }
    }

    return NULL;
}
```

D controller.h

```
#ifndef _CONTROLLER_H
#define _CONTROLLER_H

#include <libusb-1.0/libusb.h>
#include <stdint.h>

//
struct controller_list {
    struct libusb_device_handle *device1;
    uint8_t device1_addr;
};

//               7
struct controller_pkt {
    uint8_t const1;
    uint8_t const2;
    uint8_t const3;
    uint8_t dir_x;
    uint8_t dir_y;
    uint8_t ab;
    uint8_t rl;
};

//
struct args_list {
    struct controller_list devices;
    char *buttons; //         6
    int mode;      //         1
    int print;     //         1         buttons
};

//               libusb
struct controller_list open_controller();

//              controllerpthread
void *listen_controller(void *arg);

//             controller_pktA            "LRUDA"
```

```
void detect_presses(struct controller_pkt pkt, char *buttons, int mode);

#endif
```

E vga_ball.c

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

#define DRIVER_NAME "vga_ball"

#define SPRITE_DESC_OFFSET(i) (dev.virtbase + (i) * 8)
#define SCORE_REG_OFFSET    (dev.virtbase + 0x28)
#define CONTROL_REG_OFFSET  (dev.virtbase + 0x2A)
#define PELLETE_EAT_REG_OFFSET (dev.virtbase + 0x2C)

#define NUM_SPRITES 5

struct vga_ball_dev {
    struct resource res;
    void __iomem *virtbase;
} dev;

static void write_all_state(vga_all_state_t *state) {

    int i = 0;
    for (i = 0; i < NUM_SPRITES; i++) {
        void __iomem *base = SPRITE_DESC_OFFSET(i);
        iowrite8(state->sprites[i].x,     base);
        iowrite8(state->sprites[i].y,     base + 1);
        iowrite8(state->sprites[i].frame, base + 2);
        iowrite8(state->sprites[i].visible, base + 3);
        iowrite8(state->sprites[i].direction,base + 4);
        iowrite8(state->sprites[i].type_id, base + 5);
        iowrite8(state->sprites[i].reserved1, base + 6);
        iowrite8(state->sprites[i].reserved2, base + 7);
    }
    iowrite16(state->score, SCORE_REG_OFFSET);
    iowrite8(state->control, CONTROL_REG_OFFSET);
    iowrite8(0, CONTROL_REG_OFFSET + 1); // Clear the control flag
    iowrite16(state->pellet_to_eat, PELLETE_EAT_REG_OFFSET);
}

static void read_all_state(vga_all_state_t *state) {
    int i = 0;
    for (i = 0; i < NUM_SPRITES; i++) {
        void __iomem *base = SPRITE_DESC_OFFSET(i);
        state->sprites[i].x       = ioread8(base);
        state->sprites[i].y       = ioread8(base + 1);
```

```
        state->sprites[i].frame  = ioread8(base + 2);
        state->sprites[i].visible = ioread8(base + 3);
        state->sprites[i].direction = ioread8(base + 4);
        state->sprites[i].type_id = ioread8(base + 5);
        state->sprites[i].reserved1   = ioread8(base + 6);
        state->sprites[i].reserved2   = ioread8(base + 7);
    }
    state->score = ioread16(SCORE_REG_OFFSET);
    state->control = ioread8(CONTROL_REG_OFFSET);
    state->pellet_to_eat = ioread16(PELLETE_EAT_REG_OFFSET);
}

static long vga_ball_ioctl(struct file *f, unsigned int cmd, unsigned long arg) {
    vga_all_state_t user_state;

    switch (cmd) {
        case VGA_BALL_WRITE_ALL:
            if (copy_from_user(&user_state, (void __user *)arg, sizeof(user_state)))
                return -EFAULT;
            write_all_state(&user_state);
            break;

        case VGA_BALL_READ_ALL:
            read_all_state(&user_state);
            if (copy_to_user((void __user *)arg, &user_state, sizeof(user_state)))
                return -EFAULT;
            break;

        default:
            return -EINVAL;
    }

    return 0;
}

static const struct file_operations vga_ball_fops = {
    .owner = THIS_MODULE,
    .unlocked_ioctl = vga_ball_ioctl,
};

static struct miscdevice vga_ball_misc_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = DRIVER_NAME,
    .fops = &vga_ball_fops,
};

static int __init vga_ball_probe(struct platform_device *pdev) {
    int ret;

    ret = misc_register(&vga_ball_misc_device);
    if (ret)
        return ret;

    ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
    if (ret)
        goto out_deregister;

    if (request_mem_region(dev.res.start, resource_size(&dev.res), DRIVER_NAME) == NULL) {
        ret = -EBUSY;
        goto out_deregister;
```

```
    }

    dev.virtbase = of_iomap(pdev->dev.of_node, 0);
    if (dev.virtbase == NULL) {
        ret = -ENOMEM;
        goto out_release_mem;
    }

    return 0;

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

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;
}

#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

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),
};

static int __init vga_ball_init(void) {
    pr_info(DRIVER_NAME ": init\n");
    return platform_driver_probe(&vga_ball_driver, vga_ball_probe);
}

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("Your Name");
MODULE_DESCRIPTION("Pac-Man VGA Controller - Sprite and Score Management");
```

F vga_ball.h

```
#ifndef _VGA_BALL_H_
#define _VGA_BALL_H_

#include <linux/ioctl.h>
// #include <stdint.h>
#include <linux/types.h>
#define VGA_BALL_MAGIC 'q'

// Sprite descriptor structure (matches your memory map)
struct sprite_desc {
    __u8 x;
    __u8 y;
    __u8 frame;
    __u8 visible;
    __u8 direction;
    __u8 type_id;
    __u8 reserved1;
    __u8 reserved2;
};

// IOCTL interface
#define VGA_BALL_WRITE_ALL _IOW(VGA_BALL_MAGIC, 1, struct vga_all_state)
#define VGA_BALL_READ_ALL _IOR(VGA_BALL_MAGIC, 2, struct vga_all_state)

// Aggregate structure
struct vga_all_state {
    struct sprite_desc sprites[5]; // Pac-Man + 4 ghosts
    __u16 score;
    __u8 control;
    __u16 pellet_to_eat;
};

typedef struct sprite_desc sprite_desc_t;
typedef struct vga_all_state vga_all_state_t;

#endif
```