Pac-Man on the DE1-SoC FPGA

CSEE 4840 Embedded System Design Final Project Report

Austin Gnecco, Anastasiia Merudanova, Connor Marvin, Shishir Sharma

May 14, 2026

Abstract

This project implements a playable Pac-Man-style game on the DE1-SoC platform using a mixed hardware and software architecture. The FPGA fabric renders the VGA tilemap, Pac-Man sprite, four ghost sprites, palette-indexed colors, and sprite-state visuals. The HPS side runs the game loop in Linux userspace, communicates with a custom kernel driver, reads keyboard or USB controller input, updates the tilemap, plays audio, and manages the Pac-Man game rules. The final system supports pellets, super pellets, score and high-score display, lives, ghost AI, vulnerable ghosts, eaten ghost eyes returning to the ghost box, and a game-over condition that stops the program when the player runs out of lives.

Contents

1 Introduction

he goal of the project was to build an interactive Pac-Man game that uses both sides of the DE1-SoC. The FPGA is responsible for deterministic video generation, while the ARM HPS is responsible for higher-level gameplay decisions. This division keeps pixel timing, tile rendering, sprite overlay, transparency, and palette lookup in hardware, where the design can meet VGA timing reliably, while keeping pathfinding, collision handling, controller input, audio control, scoring, and game state in software, where iteration is faster.

The project was developed around a custom Platform Designer peripheral named `/vga_pacman` The hardware exposes an Avalon-MM register interface to Linux through the HPS lightweight bridge. Software communicates with the peripheral through a Linux kernel driver that creates `/dev/vga_pacman`. Rather than sending raw pixels, the software sends compact scene state: Pac-Man and ghost positions, direction and animation fields, ghost state, and tilemap updates. The driver translates these ioctl requests into memory-mapped register writes, and the FPGA continuously uses those registers and tilemap entries to render the current scene to VGA.

This architecture was chosen because Pac-Man is naturally a tile-and-sprite game. A tilemap and sprite-register interface requires far less memory bandwidth than a full framebuffer, while still giving software enough control to update gameplay every frame.

2 System Overview

The fundamental blocks of project can be broken down into four distinct groups:

1. Software layer: Handles input, movement, collision detection, ghost logic, scoring, lives, and audio events.

2. Linux driver layer: Allows for communication between the HPS and hardware by translating ioctl calls into accesses memory-mapped hardware accesses.

3. Avalon-MM hardware interface layer: Stores sprite positions, control bits, READY status, and tilemap entries in FPGA-accessible registers and RAM.

4. FPGA Hardware: Generates VGA timing, tile graphics, sprite overlays, RGB output, and audio output timing.

These four blocks all interface together to create the complete system. The system architecture is elaborated in Figure 1 which shows the dataflow paths throughout the system. Each one of these blocks will be described in the following sections.

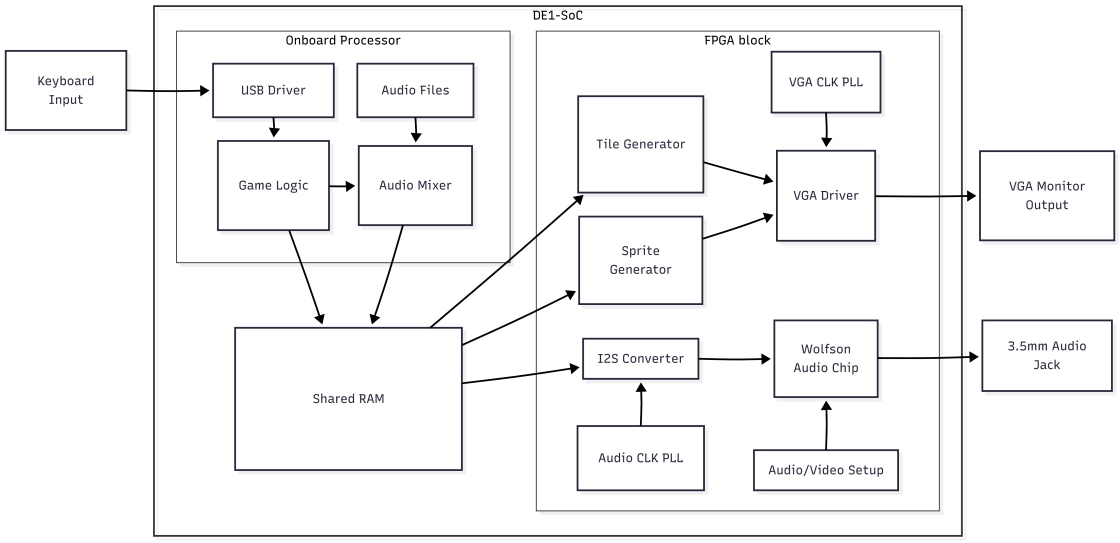


Figure 1: Pac-Man System Block Diagram

2.1 Register Map

The register map below shows how the Linux HPS communicates with the hardware across the Avalon Memory Mapped bridge.

Table 1: Sprite, Tile, and Audio Register Map (DE1-SoC System)

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VGA Graphics Block (Base 0xff200000)																
0x0000							Px9	Px8	Px7	Px6	Px5	Px4	Px3	Px2	Px1	Px0
0x0001								Py8	Py7	Py6	Py5	Py4	Py3	Py2	Py1	Py0
0x0002											Pi5	Pi4	Pi3	Pi2	Pi1	Pi0
0x0003																ready
...	*Unused / Reserved Address Space*															
0x0010							G0x9	G0x8	G0x7	G0x6	G0x5	G0x4	G0x3	G0x2	G0x1	G0x0
0x0011								G0y8	G0y7	G0y6	G0y5	G0y4	G0y3	G0y2	G0y1	G0y0
0x0012									G0s1	G0s0	G0i5	G0i4	G0i3	G0i2	G0i1	G0i0
...	*Unused Sprite Registers*															
0x0014							G1x9	G1x8	G1x7	G1x6	G1x5	G1x4	G1x3	G1x2	G1x1	G1x0
0x0015								G1y8	G1y7	G1y6	G1y5	G1y4	G1y3	G1y2	G1y1	G1y0
0x0016									G1s1	G1s0	G1i5	G1i4	G1i3	G1i2	G1i1	G1i0
...	*Unused Sprite Registers*															
0x0018							G2x9	G2x8	G2x7	G2x6	G2x5	G2x4	G2x3	G2x2	G2x1	G2x0
0x0019								G2y8	G2y7	G2y6	G2y5	G2y4	G2y3	G2y2	G2y1	G2y0
0x001A									G2s1	G2s0	G2i5	G2i4	G2i3	G2i2	G2i1	G2i0
...	*Unused Sprite Registers*															
0x001C							G3x9	G3x8	G3x7	G3x6	G3x5	G3x4	G3x3	G3x2	G3x1	G3x0
0x001D								G3y8	G3y7	G3y6	G3y5	G3y4	G3y3	G3y2	G3y1	G3y0
0x001E									G3s1	G3s0	G3i5	G3i4	G3i3	G3i2	G3i1	G3i0
...	*Unused / Reserved Address Space*															
0x0100									r0c0i7	r0c0i6	r0c0i5	r0c0i4	r0c0i3	r0c0i2	r0c0i1	r0c0i0
0x0101									r0c1i7	r0c1i6	r0c1i5	r0c1i4	r0c1i3	r0c1i2	r0c1i1	r0c1i0
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
0x0826									r29c38i7	r29c38i6	r29c38i5	r29c38i4	r29c38i3	r29c38i2	r29c38i1	r29c38i0
0x0827									r29c39i7	r29c39i6	r29c39i5	r29c39i4	r29c39i3	r29c39i2	r29c39i1	r29c39i0
Altera UP Audio Core Block (Base 0xff240000)																
Audio 0x00													CW	CR	WE	RE
Audio 0x01	RR7	RR6	RR5	RR4	RR3	RR2	RR1	RR0	RL7	RL6	RL5	RL4	RL3	RL2	RL1	RL0
Audio 0x02	WL7	WL6	WL5	WL4	WL3	WL2	WL1	WL0	WR7	WR6	WR5	WR4	WR3	WR2	WR1	WR0
Audio 0x03	L15	L14	L13	L12	L11	L10	L9	L8	L7	L6	L5	L4	L3	L2	L1	L0
Audio 0x04	R15	R14	R13	R12	R11	R10	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0

The hardware block `vga_pacman_mm.sv` exposes a word-addressed Avalon-MM slave. Software updates sprite coordinates, sprite control fields, and tilemap entries through this interface.

Table 2: Main Avalon-MM register map.

Address	Register	Description
0x00000	Pac-Man X	Pixel X position
0x00001	Pac-Man Y	Pixel Y position
0x00002	Pac-Man control	Visible, direction, animation frame
0x00003	READY	VSync synchronization flag
0x00010	Ghost 0 X	Pixel X position for ghost 0
0x00011	Ghost 0 Y	Pixel Y position for ghost 0
0x00012	Ghost 0 control	Visible, direction, frame, state
0x00014	Ghost 1 X	Pixel X position for ghost 1
0x00015	Ghost 1 Y	Pixel Y position for ghost 1
0x00016	Ghost 1 control	Visible, direction, frame, state
0x00018	Ghost 2 X	Pixel X position for ghost 2
0x00019	Ghost 2 Y	Pixel Y position for ghost 2
0x0001A	Ghost 2 control	Visible, direction, frame, state
0x0001C	Ghost 3 X	Pixel X position for ghost 3
0x0001D	Ghost 3 Y	Pixel Y position for ghost 3
0x0001E	Ghost 3 control	Visible, direction, frame, state
0x00100--0x0087F	Tilemap RAM	Background tile IDs

The ghost control word packs its fields as:

Listing 1: Ghost control register packing.

```
bit 0     = visible
bits 2:1  = direction
bits 5:3  = frame
bits 7:6  = state: 0 normal, 1 vulnerable, 2 eyes
```

2.2 Setup in Quartus

The Pac-Man hardware system was integrated in Intel Quartus using Platform Designer. Figure **??** shows the main system-level connections between the HPS, custom VGA peripheral, clocking blocks, and audio components. The design uses the DE1-SoC's ARM HPS as the software controller and the FPGA fabric as the hardware rendering and audio-output engine.

The `hps_0` block provides the processor-side interface and connects to the FPGA logic through the HPS-to-FPGA Avalon-MM bridge. The custom `vga_pacman_0` peripheral is connected as an Avalon-MM slave so that software running on Linux can update Pac-Man position, ghost position, control bits, READY status, and tilemap entries through memory-mapped register writes. In the address map, the Pac-Man peripheral is assigned a base address range of `0x0000_0000` to `0x0003_ffff`, giving the software side a fixed region for communicating with the FPGA game logic.

The system also includes clock-generation blocks. The main system clock drives the HPS and Avalon-MM interface, while a separate VGA clock generated by the PLL is used by the VGA rendering logic. This separation is important because VGA output requires deterministic pixel timing, while the processor-side register interface runs on the system bus clock. The custom Pac-Man peripheral exports a VGA conduit, which connects the generated RGB, horizontal sync, vertical sync, and related video signals to the external VGA pins.

Audio support is also included in the Platform Designer system. The `audio_0`, `audio_and_video_config_0`, and `audio_pll_0` blocks provide the interface needed to communicate with the DE1-SoC audio codec. The audio peripheral is assigned its own Avalon-MM address range, separate from the Pac-Man VGA peripheral. This allows software to control or configure audio output independently while the FPGA handles timing-sensitive video generation.

Overall, the Quartus setup connects the software-controlled HPS side with the hardware-rendered FPGA side. The HPS writes game-state updates into memory-mapped registers, and the FPGA uses those values every frame to draw the tilemap, sprites, and output signals. This structure keeps gameplay logic flexible in software while keeping VGA and audio timing deterministic in hardware.

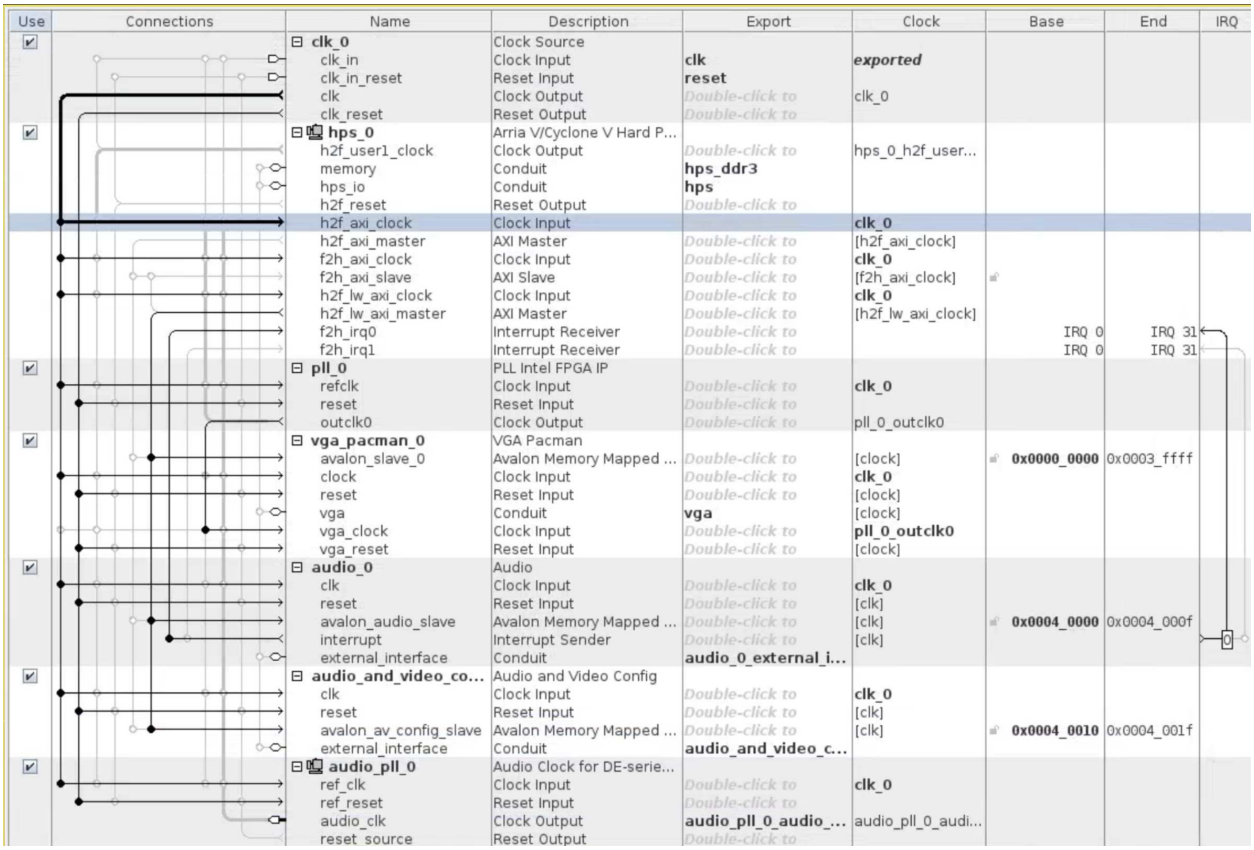


Figure 2: System Setup in Quartus

2.3 Project Folder Organization

The implementation is split across two main folders. The `PacMan` folder is the FPGA hardware project. It contains the Quartus project, Platform Designer system, SystemVerilog rendering modules, memory initialization files, palette data, and generated bitstream outputs. The `pacman_driver` folder is the software project that runs on the Linux side of the DE1-SoC. It contains the kernel module, userspace game program, audio helpers, controller input code, and utility programs for drawing/debugging tilemaps.

Folder	Responsibility
`hardware/`	Hardware/FPGA project. Generates the `soc_system` Platform Designer hardware, compiles the VGA peripheral, stores the `.mem` sprite/tile ROM files, and produces FPGA programming outputs such as `.sof` and `.rbf`.
`pacman_driver/`	Software/Linux project. Builds `vga_pacman.ko`, exposes `/dev/vga_pacman`, runs the Pac-Man game loop, reads input, plays audio, updates score/lives, and sends sprite/tile updates to the FPGA.

Table 3: High-level purpose of the two main project directories.

This folder split also matches the hardware/software boundary. The FPGA does not decide game rules. It only draws the current tilemap and sprites based on register values. The software decides what should happen next in the game, then sends that state to the FPGA. For example, when Pac-Man eats a super pellet, `pacman_hw_game.c` changes the ghost mode to vulnerable, `vga_pacman.c` packs that mode into the ghost control register, and `ghost_sprite.sv` selects the vulnerable ghost ROM during rendering.

3 Hardware Design

3.1 Hardware Files

The `hardware/` folder contains the FPGA design for the Pac-Man system. This contains both the hardware .sv and .mem files written by us, and the support files generated by Quartus in `socsystem/`. The main hardware files in the project are below:

- `vga_pacman_mm.sv`: Avalon-MM wrapper that exposes Pac-Man, ghost, READY, and tilemap registers.
- `vga_graphics.sv`: Main VGA compositor that combines background tiles, Pac-Man, and ghosts.
- `ghost_sprite.sv`: Ghost sprite renderer, including normal, vulnerable, and eyes sprite selection.
- `pacman_sprite.sv`: Pac-Man sprite renderer.
- `tileset_rom.sv` and `palette_rom.sv`: ROM modules for tile graphics and color lookup.
- `*.mem`: Palette-indexed memory initialization data for sprites, tiles, and the initial tilemap.

3.2 VGA Driver

The display pipeline is built around `vga_counters.sv`, which produces the horizontal count, vertical count, blanking signal, and sync pulses. The active display region is treated as a 640 by 480 pixel screen. Since each background tile is 16 by 16 pixels, the visible screen contains 40 columns and 30 rows of tiles.

3.3 Tilemap and Tileset

The background maze is rendered using an 8-bit tile number from tilemap RAM. The visible tilemap uses 40 by 30 entries, while the hardware row stride is 64 entries. The tilemap is implemented with `twoportbram.sv`: one port is used by the VGA clock domain for rendering, and the other port is used by the HPS/Avalon clock domain for software updates.

The tile image data is stored in `tileset.mem`. The renderer addresses it with:

Listing 2: Tileset ROM address construction.

```
addr = {tilenumber, vcount[3:0], hcount[3:0]};
```

This gives each tile a 16 by 16 pixel bitmap, with one 4-bit palette index per pixel. The palette ROM then converts the final 4-bit color index into 24-bit RGB.

Table 4: On-chip Memory Usage for FPGA Graphics

Name	Type	Image	Size (bits)	# of Frames	Total size
Pac-Man	Sprite		32×32	4	16,384
Ghosts	Sprite		32×32	5	20,480
Map Pieces	Tile		16×16	6	12,288
Dots	Tile		16×16	2	4,096
Characters	Tile		16×16	36	65,536

3.4 Palette-Indexed Rendering

All major graphics use palette indexes instead of direct RGB. This reduces memory width and keeps all art consistent. The `palette.mem` file contains 16 RGB entries, while the sprite and tile memory files contain only 4-bit indexes. Important palette roles include:

Index	Use	Color meaning
`0`	Transparent/background	Black or transparent sprite pixel
`3`	Pac-Man	Yellow
`4`	Red ghost	Blinky body
`5`	Pink ghost / face detail	Pink body and vulnerable face
`6`	Cyan ghost	Inky body
`7`	Orange ghost	Clyde body
`8`	Eye detail	Dark blue pupil / eyes state
`C`	Vulnerable body	Blue vulnerable ghost body
`E`	Eye white	White eye region

Table 5: Palette indexes used by the sprite and tile rendering pipeline.

3.5 Pac-Man Sprite Renderer

`pacman_sprite.sv` renders Pac-Man from `pacman_sprite.mem`. The Pac-Man sprite is 32 by 32 pixels and contains 12 frames. The frame selection depends on direction and animation frame. The sprite outputs:

- `pac_on`: high when the current VGA pixel is an opaque Pac-Man pixel.
- `pac_colorindex`: 4-bit palette index for that pixel.

The final renderer gives Pac-Man priority over ghosts and the background, so Pac-Man remains visible when sprites overlap.

3.6 Ghost Sprite Renderer

Each ghost is also rendered as a 32 by 32 sprite. The normal colored ghosts use separate ROM files:

- `ghost_red.mem`
- `ghost_blue.mem`
- `ghost_pink.mem`
- `ghost_orange.mem`

Two additional ROMs were added for special states:

- `ghost_vulnerable.mem`: blue vulnerable ghost sprite.
- `ghost_eyes.mem`: eyes-only sprite after Pac-Man eats a vulnerable ghost.

The ghost sprite module now accepts a 2-bit state input. State `0` renders the normal color, state `1` renders the vulnerable ghost, and state `2` renders the eyes sprite.

Listing 3: Ghost state selection in hardware.

```
unique case (ghost_state)
    2'd1:    sprite_pixel = vulnerable_pixel;
    2'd2:    sprite_pixel = eyes_pixel;
    default: sprite_pixel = normal_pixel;
endcase
```

3.7 Final Pixel Priority

The final pixel color is selected in `vga_graphics.sv`. The priority order is:

1. Pac-Man sprite
2. Ghost 0
3. Ghost 1
4. Ghost 2

5. Ghost 3

6. Background tile

Once the final 4-bit palette index is selected, `palette_rom.sv` outputs the 24-bit VGA RGB value.

4 Linux Kernel Driver

The `pacman_driver` folder contains the software side of the project. It has two major software layers. The first layer is the Linux kernel driver, which is the only code that directly accesses the FPGA's memory-mapped registers. The second layer is the userspace game program, which implements Pac-Man gameplay and calls the driver through ioctls.

File or directory	Role in the software project
`vga_pacman.c`	Kernel module that maps the FPGA Avalon-MM peripheral and creates `/dev/vga_pacman`.
`vga_pacman_ioctl.h`	Shared ioctl definitions used by both the kernel module and userspace programs.
`pacman_hw_game.c`	Main playable game loop: movement, collisions, pellets, score, lives, vulnerable mode, eyes mode, and game over.
`audio_driver.c/.h`	Userspace helper for loading and playing WAV sound effects.
`pacman_audio.c/.h`	Kernel-side audio support used by the audio device path.
`Sounds/`	WAV files for start music, siren, pellet eating, ghost eating, and death.
`draw_*.c`, `fill_screen_tiles.c`	Utility programs for testing tile IDs and drawing/debugging maze layouts.
`Makefile`	Builds the kernel modules and userspace executables on the DE1-SoC Linux image.

Table 6: Important files in the `pacman_driver` software folder.

The kernel module `vga_pacman.c` is a direct-mapped misc driver. When loaded, it creates `/dev/vga_pacman`. The driver maps the physical Avalon-MM address space with `ioremap`, then reads and writes 16-bit hardware words using `ioread16` and `iowrite16`.

The userspace program does not directly access physical memory. Instead, it uses ioctl calls defined in `vga_pacman_ioctl.h`. This creates a cleaner boundary between game logic and low-level register access.

4.1 Sprite ioctl ABI

Each sprite update contains pixel position, direction, animation frame, visibility, and now a ghost state field. Pac-Man leaves `state` as zero. Ghosts use it to select between normal, vulnerable, and eyes.

Listing 4: Sprite structure used by userspace and the kernel driver.

```
1 struct vga_pacman_sprite {
```

```
    __u16 x;
    __u16 y;
    __u8  dir;
    __u8  frame;
    __u8  visible;
    __u8  state;
};
```

The kernel driver packs that state into bits 7:6 of the hardware control register:

Listing 5: Driver control-word packing.

```
ctrl = 0;
ctrl |= (s->visible ? 1 : 0) & 0x1;
ctrl |= (s->dir & 0x3) << 1;
ctrl |= (s->frame & 0x7) << 3;
ctrl |= (s->state & 0x3) << 6;
```

5 Userspace Game Logic

The main game program is `pacman_hw_game.c`. This program opens `/dev/vga_pacman`, initializes audio and input, draws the score and lives display, and then runs the game loop.

5.1 Game Loop

The game loop advances in fixed delay steps. Each iteration:

1. Reads controller or keyboard input.
2. Advances Pac-Man and ghost animations.
3. Moves Pac-Man at the Pac-Man movement interval.
4. Eats pellets under Pac-Man.
5. Activates power mode when a super pellet is eaten.
6. Moves ghosts at the ghost movement interval.
7. Handles ghost collisions.
8. Updates score, lives, win, and game-over display.
9. Sends current sprite states to the hardware.

5.1.1 Pixel Interpolation

The pixel interpolation that is performed by the system in order to allow for a smooth style of movement consists of calculating the amount of distance moved per tick and interpolating that distance in pixels. Since Pac-Man and the Ghosts do not change their speeds in this version, each sprite moves with a specific and constant speed. Because of this, we are able to very easily interpolate where the pixel should reside after the next frame refresh. Effectively, we are able to

tell the pixel to go in between tiles because of the fact that we are using it as a Sprite instead of a tile, which allows us to change pixel locations very quickly.

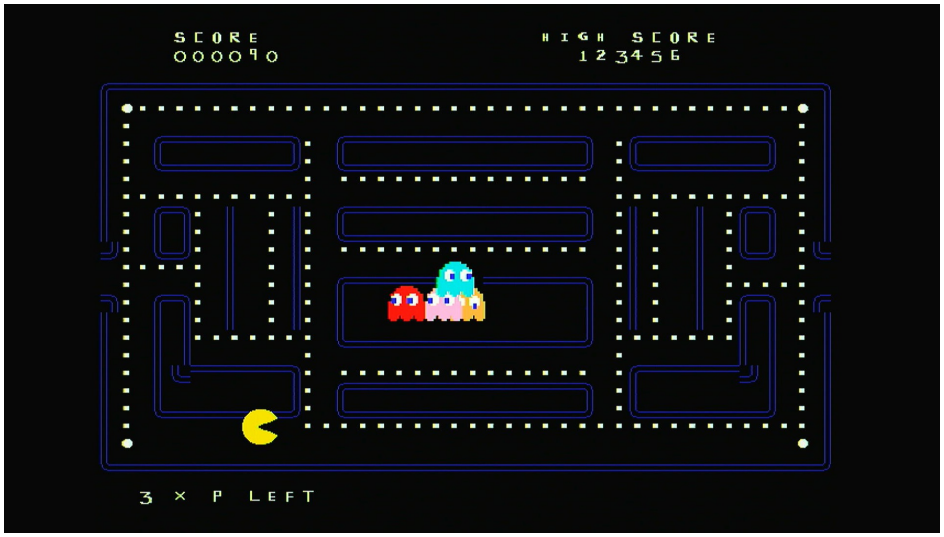


Figure 3: Sprite pixel interpolation between tile positions.

As seen above, both Pac-Man and the ghosts are intersecting multiple tiles, yet there is no tearing due to the fact that we have that custom pixel interpolation that effectively provides a level of buffer logic for the code to understand where the pixels will need to be placed next.

5.2 Maze and Pellets

The maze is represented as a 40 by 30 character map in software. Characters identify walls, open paths, normal pellets, and power pellets. The software keeps a live pellet array so that eaten pellets are removed from both the logical game state and the hardware tilemap.

Tile meaning	Behavior
Open path	Pac-Man and ghosts may move through it
Wall	Blocks movement
Normal pellet	Adds 10 points and clears the tile
Power pellet	Adds 50 points, clears the tile, and starts vulnerable mode

Table 7: Software interpretation of the maze and pellet map.

5.2.1 2x2 Hitbox Clarification

Due to the fact that the tiles are 16-by-16-pixel graphics and the sprites are 32-by-32-pixel graphics, the program utilizes a two-by-two hitbox for collision. This 2X2 hitbox is exactly why we chose to utilize matrices for the game logic instead of other forms of data. By having this 2-by-2 hitbox able to be contained within an easy-to-use format, it made prototyping much easier.

5.2.2 Tunnel Wrapping

As previously mentioned in the interpolation section, we are able to directly control where the pixels are drawn for the moving sprites, such as Pac-Man and the Ghosts. Because of this, we are able to integrate the tunnel mechanics into our version of Pac-Man. Since in that specific location, you can only move one direction the program requires a special case. To do this the program recognizes if it is moving in that special location on either side, and then it will need to write the pixel locations to their alternate locations on the other side of the maze.

5.3 Ghost AI

The four ghosts use different target rules inspired by classic Pac-Man behavior:

- Blinky targets Pac-Man's current position.
- Pinky targets several tiles ahead of Pac-Man.
- Inky uses Pac-Man's forward position and Blinky's position to compute a projected target.
- Clyde targets Pac-Man when far away and a corner when close.

When a ghost is vulnerable, it targets a corner far from Pac-Man. When a ghost has been eaten, it switches into the eyes state and targets its home position in the ghost box.

5.4 Vulnerable and Eyes State Logic

When Pac-Man eats a super pellet, the software sets a power timer:

Listing 6: Power timer behavior.

```
power_until_tick = current_tick + POWER_DURATION_TICKS;
```

While the current tick is less than `power_until_tick`, ghosts become vulnerable unless they are already in the eyes state. This is important because an eaten ghost should return to the box instead of becoming vulnerable again before it finishes returning.

Listing 7: Ghost state mapping before sending to hardware.

```
if (ghost->mode == GHOST_FRIGHTENED) {
    state = VGA_GHOST_VULNERABLE;
}
else if (ghost->mode == GHOST_EYES) {
    state = VGA_GHOST_EYES;
}
else {
    state = VGA_GHOST_NORMAL;
}
```

When Pac-Man collides with a vulnerable ghost, the score increases, the eaten-ghost sound plays, and the ghost switches to eyes:

Listing 8: Eaten ghost transition.

```
if (ghost->mode == GHOST_FRIGHTENED) {
    score += SCORE_GHOST;
    ghosts_eaten++;
```

```
    send_eaten_ghost_home(ghost);
}
```

The eyes state ignores collisions with Pac-Man and moves toward the ghost's home position. Once the ghost overlaps its home or the spawn box, it returns to normal mode and leaves the box again.

5.5 Lives and Game Over

The player starts with three lives. If Pac-Man collides with a normal ghost, a life is subtracted and the sprites reset to their starting positions. If lives reaches zero, the game draws "GAME OVER", clears the running flag, exits the loop, and terminates the program. To play again, the user must restart the executable.

Listing 9: Game-over behavior after the last life is lost.

```
if (lives <= 0) {
    game_over = 1;
    running = 0;
    draw_game_over_hw(fd);
    return;
}
```

6 Input and Audio

6.1 Input

The game supports keyboard input and a USB controller. Keyboard input uses nonblocking terminal reads for WASD or arrow keys. The USB controller path uses `libusb` to poll a DragonRise/KIWITATA-style controller. The latest valid direction is stored as Pac-Man's requested next direction.

6.1.1 Direction Buffering

As stated within the various sections, the polling rate for the USB controller and keyboard is much higher than the refresh rate. To avoid misinputs and incorrect data, we decided to implement the movement format of the direction of movement, being unable to change until that Sprite enters a new tile. In other words, if Pac-Man is moving between two tiles and you change direction with the controller or keyboard, He will not change direction until he fully enters that new tile.

6.2 Audio Subsystem Architecture

In order to get audio working, we implmented a hardware-interrupt-driven audio pipeline. The architecture is built across three distinct layers: the DE1-SoC FPGA Audio IP block, a custom Linux kernel driver with wait queues, and a multi-threaded userspace mixer.

6.2.1 Hardware Layer: Altera UP Audio IP & Wolfson WM8731 Codec

At the hardware foundation, the project leverages the physical Wolfson WM8731 Audio Codec on the DE1-SoC development board, paired with the Altera Audio IP Core mapped within the Platform Designer system.

1. **Codec Configuration via I2C:** The ARM HPS coordinates with the Wolfson WM8731 chip over a dedicated physical I2C control bus. At game startup, the system configures the codec register registers to establish an 8 kHz sampling rate, enable the stereo audio path, activate the line-out stage, and mute unused line-in paths.

2. **Avalon-MM Interface & Write FIFOs:** The audio core exposes a 16-bit word-addressed registers set via the Avalon-MM interface. The core incorporates two 128-word hardware FIFOs (one Left channel, one Right channel) feeding the codec's DAC:

   - `REG_CONTROL` (Offset 0): Holds FIFO clear bits (Bits 3:2) and the write interrupt enable bit (`WE` - Bit 1). When `WE` is high, the hardware is configured to assert a physical interrupt line whenever the write FIFOs fall below a 75% fill capacity (meaning they have at least 32 free slots).
   - `REG_FIFOSPACE` (Offset 1): Read-only register revealing the current available slots in the Left (Bits 31:24) and Right (Bits 23:16) FIFOs.
   - `REG_LEFTDATA` (Offset 2) and `REG_RIGHTDATA` (Offset 3): Writing 16-bit signed PCM samples here directly queues them into the DAC FIFOs.

6.2.2 Kernel Layer: Interrupt Synchronization via `pacman_audio.ko`

To prevent CPU-hogging polling loops in userspace, a custom kernel module `pacman_audio.ko` is registered against the physical audio device node mapped via the device tree structure as compatible with `"csee4840,pacman-audio-1.0"`.

1. **Interrupt Registration:** The driver maps the physical interrupt line (GIC SPI 40, which registers as Linux IRQ 72) and binds a highly responsive Interrupt Service Routine (ISR) to it.

2. **Kernel Wait Queues:** The driver exposes a custom character device file node at `/dev/pacman_audio` supporting a blocking `ioctl(..., PACMAN_AUDIO_WAIT_INTERRUPT)` command.

3. **ISR Synchronization Loop:** When the FPGA circular write FIFO level drops below the write threshold, the physical interrupt line is pulled high. The ARM GIC intercepts this signal and transfers execution to our kernel ISR:

Listing 10: Kernel driver Interrupt Service Routine outline.

```
1    static irqreturn_t pacman_audio_isr(int irq, void *dev_id) {
2        // Clear interrupt signal in FPGA hardware control registers
3        iowrite32(0x2, audio_base_reg);
4
5        // Wake up sleeping userspace mixing threads
6        wake_up_interruptible(&audio_wait_queue);
7        return IRQ_HANDLED;
8    }
```

   This mechanism suspends the userspace thread in kernel-space, ensuring zero CPU execution consumption while waiting for the hardware buffers to empty.

6.2.3 Userspace Layer: Multi-Threaded Real-Time Mixer

The userspace audio library `audio_driver.c` spawns a dedicated high-priority background thread `audio_processing_thread` immediately upon initialization.

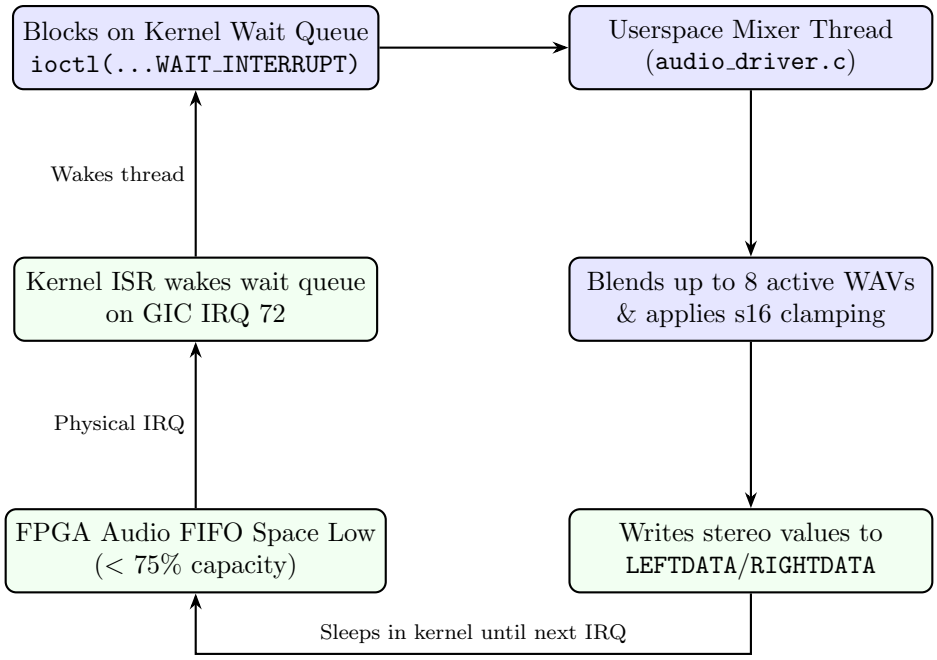


Figure 4: Optimized interrupt-driven audio loop with side-entry routing.

The background loop implements a multi-channel linear mixing pipeline:

1. **Interrupt Sync Wait:** The thread blocks on `ioctl(sync_audio_fd, PACMAN_AUDIO_WAIT_INTERRUPT)`.
2. **FIFO Space Query:** Upon waking, the thread reads `REG_FIFOSPACE` and determines the available slots:

Listing 11: FIFO space queries.

```
    uint32_t space = audio_regs[REG_FIFOSPACE];
    int write_space_l = (space >> 24) & 0xFF;
    int write_space_r = (space >> 16) & 0xFF;
    int fill_count = (write_space_l < write_space_r) ? write_space_l :
       write_space_r;
```

3. **Software Mixer Loop:** For each active audio input, the inputs are summed together to get an optput stream. To prevent clipping distortion when multiple high-amplitude sounds (e.g., background siren and pellet eating) play simultaneously, the mixed 32-bit values are passed through a bounding function:

Listing 12: PCM clamping to prevent signal wrapping.

```
    static inline int16_t clamp_to_s16(int32_t val) {
        if (val > 32767)  return 32767;
        if (val < -32768) return -32768;
        return (int16_t)val;
    }
```

The resulting 16-bit clamped stereo values are written directly to the left and right channels.

4. **Stereo/Mono Adaptation:** The mixer supports both mono and stereo uncompressed 16-bit PCM WAV assets. If an asset is mono, the single sample is duplicated across both Left

and Right registers. If stereo, alternating Left/Right array positions are fed to the respective hardware channels.

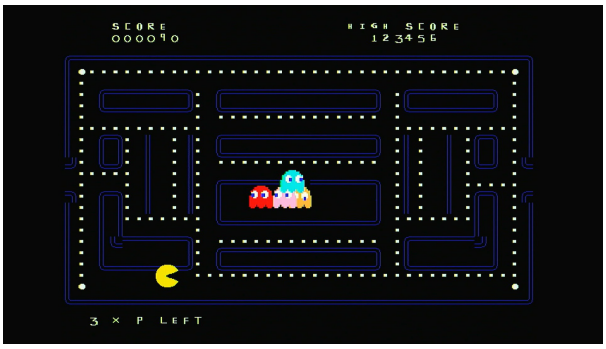
Using this hybrid pipeline, sound effects such as the alternating dual-frequency `waka` sounds trigger instantly upon pellet consumption, completely eliminating lagging audio.

7 Testing and Verification

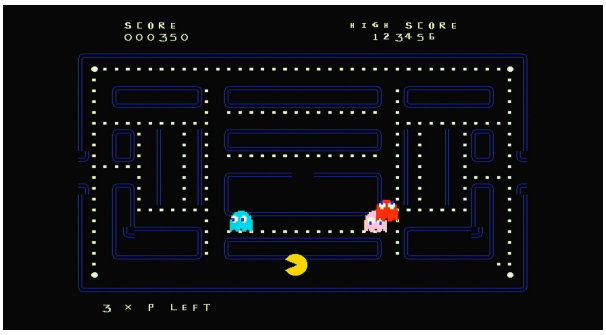
Testing of modules in this project was done in two stages. First modules were tested individually for their intended functionality, often with basic testbenches to drive inputs and compare expected outputs. Once the modules had been independantly verified, they were integrated together and debugged if there were deviations from the intended functionality. Text output from the HPS to the Linux window was often used to debug software, while hardware was debugged through Verilator simulation.

7.1 Gameplay State Screenshots

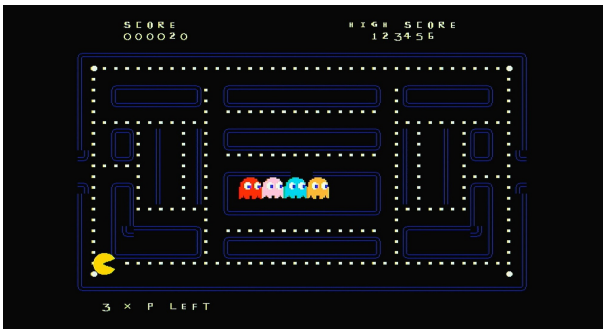
These screenshots depict different states of the game ranging from the beginning to the end of the round.



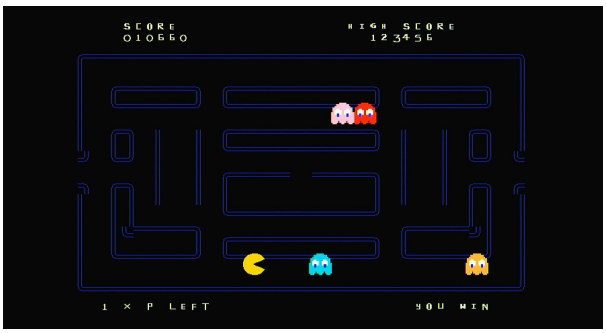
(a) Ghosts leaving spawn



(b) Ghosts fleeing from Pacman



(c) Beginning Screen



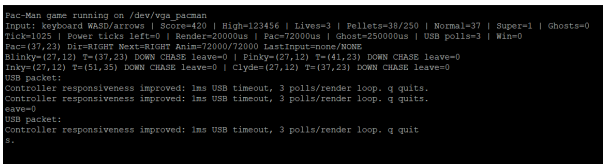
(d) Winning Screen

Figure 5: Four gameplay-state screenshots.

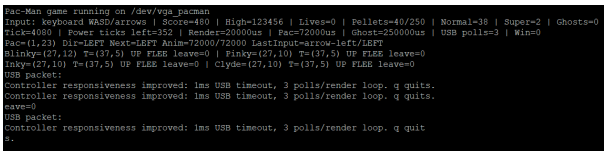
7.2 Terminal Output Screenshots

These terminal outputs were exceptionally useful for debugging code during times when drivers were not functioning correctly. There was originally a textual form of maze, but that was removed

in later versions. These include data such as movement and state for the ghosts and Pac-Man, and also game statistics such as the number of pellets left.



(a) Pacman debug terminal snapshot during "Chase" state



(b) Pacman debug terminal snapshot during "Flee" state

Figure 6: Terminal output screenshots.

8 Challenges

The main challenge was keeping the hardware and software state encodings synchronized. The FPGA renderer originally knew only the normal colored ghost sprites. Adding vulnerable and eyes states required changes in three places: the sprite ROM data, the SystemVerilog ghost renderer, and the driver/userspace control word packing.

Another challenge was deciding where game rules should live. The FPGA can render different ghost states, but it should not decide when a ghost is vulnerable or eaten. That logic belongs in software because it depends on pellet timers, collision detection, lives, score, and ghost AI. The final design keeps the FPGA focused on rendering and keeps the game rules in C.

9 Future Work

While we were able to achive the core functionality promised in our proposal, there are a number of additionas and refinements that would make this game a better experience. Some of these include:

- Add fruit bonus items and level progression.
- Add flashing vulnerable ghosts near the end of the power timer.
- Improve ghost pathfinding to more closely match the arcade rules.
- Display remaining lives as Pac-Man icons instead of a text counter.
- Add a start screen and restart button instead of requiring program restart after game over.
- Add screenshots and oscilloscope or SignalTap captures to the final report.

10 Conclusion

The project successfully combines hardware rendering and software game logic to create an interactive Pac-Man game on the DE1-SoC. The FPGA renders a tile-based maze, palette-indexed sprites, and multiple ghost states at VGA timing. The HPS software handles movement, pellet consumption, score, lives, audio, input, ghost AI, vulnerable mode, eaten ghost eyes, and game over. In the end, we were able to deliver a working game with nearly all the functions of the original Pac-Man.

A Important Source Files

Folder	File	Purpose
PacMan/	vga_pacman_mm.sv	Avalon-MM wrapper and register map
PacMan/	vga_graphics.sv	Top-level VGA graphics compositor
PacMan/	ghost_sprite.sv	Ghost sprite ROM renderer and ghost-state selector
PacMan/	pacman_sprite.sv	Pac-Man sprite renderer
PacMan/	palette_rom.sv	4-bit palette index to 24-bit RGB
PacMan/	twoportbram.sv	Dual-port tilemap RAM
PacMan/	tileset.mem	Background tile bitmap data
PacMan/	ghost_vulnerable.mem	Vulnerable ghost sprite ROM
PacMan/	ghost_eyes.mem	Eaten ghost eyes sprite ROM
PacMan/	soc_system.qsys	Platform Designer system containing the HPS and VGA peripheral
PacMan/	Makefile	Hardware build targets for Platform Designer, Quartus, RBF, device tree, and boot files
pacman_driver/	vga_pacman.c	Linux kernel driver for VGA peripheral
pacman_driver/	vga_pacman_ioctl.h	Shared userspace/kernel ioctl ABI
pacman_driver/	pacman_hw_game.c	Main userspace gameplay loop
pacman_driver/	audio_driver.c	WAV loading and playback helpers
pacman_driver/	Sounds/	Pac-Man sound effects and background loops
pacman_driver/	Makefile	Builds driver modules and userspace programs

B Build and Run Notes

On the DE1-SoC Linux image, the driver project can be rebuilt from the `pacman_driver` directory:

```
make clean
make
```

The kernel module must be loaded before running the game:

```
sudo insmod ./vga_pacman.ko
./pacman_hw_game
```

After game over, the program exits. To play again, run `./pacman_hw_game` again.

C Sound Credit

The sounds for this project were public use and aquired from the website below:
https://sounds.spriters-resource.com/arcade/pacman/asset/404131/

D Project Code

###ALL CODE, STARTING WITH HARDWARE###

```
module ghost_sprite #(
    parameter int SPR_W       = 32,
    parameter int SPR_H       = 32,
    parameter int NUM_FRAMES  = 8,
    parameter int GHOST_COLOR = 0   // 0=red, 1=blue/cyan, 2=pink, 3=orange
)(
    input  logic       VGA_CLK,
    input  logic [9:0] pix_x,
    input  logic [8:0] pix_y,
    input  logic [9:0] ghost_x,
    input  logic [8:0] ghost_y,
    input  logic [1:0] ghost_dir,
    input  logic [2:0] ghost_frame,
    input  logic       ghost_visible,
    input  logic [1:0] ghost_state,     // 00=normal, 01=vulnerable, 10=eyes
    output logic       ghost_on,
    output logic [3:0] ghost_colorindex
);
    localparam logic [1:0] DIR_UP    = 2'd0;
    localparam logic [1:0] DIR_RIGHT = 2'd1;
    localparam logic [1:0] DIR_DOWN  = 2'd2;
    localparam logic [1:0] DIR_LEFT  = 2'd3;
    localparam int SPR_PIXELS = SPR_W * SPR_H;
    localparam int MEM_DEPTH  = NUM_FRAMES * SPR_PIXELS;
    localparam int ADDR_W     = $clog2(MEM_DEPTH);
    logic [4:0] lx;
    logic [4:0] ly;
    logic       in_box;
    logic       in_box_d;
    logic [2:0] sprite_name;
    logic [3:0] sprite_pixel;
    logic [ADDR_W-1:0] addr;
    /*
     * One 4-bit palette index per pixel.
     * Index 0 is transparent.
     */
    logic [3:0] sprite_mem_normal [0:MEM_DEPTH-1];
    logic [3:0] sprite_mem_vulnerable [0:MEM_DEPTH-1];
    logic [3:0] sprite_mem_eyes [0:MEM_DEPTH-1];
    generate
        if (GHOST_COLOR == 0) begin : GEN_RED_GHOST
            initial begin
                $readmemh("ghost_red.mem", sprite_mem_normal);
            end
        end else if (GHOST_COLOR == 1) begin : GEN_BLUE_GHOST
            initial begin
                $readmemh("ghost_blue.mem", sprite_mem_normal);
            end
        end else if (GHOST_COLOR == 2) begin : GEN_PINK_GHOST
            initial begin
                $readmemh("ghost_pink.mem", sprite_mem_normal);
            end
        end else begin : GEN_ORANGE_GHOST
            initial begin
                $readmemh("ghost_orange.mem", sprite_mem_normal);
            end
        end
    endgenerate
    initial begin
        $readmemh("ghost_vulnerable.mem", sprite_mem_vulnerable);
        $readmemh("ghost_eyes.mem", sprite_mem_eyes);
    end
    /*
     * 8 frames total:
     *   0 = up_0
```

```
     *   1 = up_1
     *   2 = right_0
     *   3 = right_1
     *   4 = down_0
     *   5 = down_1
     *   6 = left_0
     *   7 = left_1
     */
    always_comb begin
        unique case (ghost_dir)
            DIR_UP:    sprite_name =
ghost_frame[0] ? 3'd1 : 3'd0;
            DIR_RIGHT: sprite_name =
ghost_frame[0] ? 3'd3 : 3'd2;
            DIR_DOWN:  sprite_name =
ghost_frame[0] ? 3'd5 : 3'd4;
            default:   sprite_name =
ghost_frame[0] ? 3'd7 : 3'd6; // DIR_LEFT
        endcase
    end
    /*
     * Address generation.
     */
    always_comb begin
        in_box = 1'b0;
        lx     = 5'd0;
        ly     = 5'd0;
        addr   = '0;
        in_box =
            ghost_visible &&
            (pix_x >= ghost_x) &&
            (pix_x <  ghost_x + SPR_W) &&
            (pix_y >= ghost_y) &&
            (pix_y <  ghost_y + SPR_H);
        if (in_box) begin
            lx = pix_x - ghost_x;
            ly = pix_y - ghost_y;
            addr = sprite_name * SPR_PIXELS + ly
* SPR_W + lx;
        end
    end
    /*
     * Synchronous ROM read, matching
pacman_sprite style.
     */
    always_ff @(posedge VGA_CLK) begin
        if (ghost_state == 2'd1) begin
            sprite_pixel <=
sprite_mem_vulnerable[addr];
        end else if (ghost_state == 2'd2) begin
            sprite_pixel <=
sprite_mem_eyes[addr];
        end else begin
            sprite_pixel <=
sprite_mem_normal[addr];
        end
        in_box_d     <= in_box;
    end
    /*
     * Output stage.
     * Palette index 0 is transparent.
     */
    always_comb begin
        ghost_on         = 1'b0;
        ghost_colorindex = 4'h0;
        if (in_box_d && sprite_pixel != 4'h0)
begin
            ghost_on         = 1'b1;
            ghost_colorindex = sprite_pixel;
        end
    end
endmodule
module pacman_sprite #(
    parameter int SPR_W      = 32,
    parameter int SPR_H      = 32,
    parameter int NUM_FRAMES = 12
)(
    input  logic       VGA_CLK,
    input  logic [9:0] pix_x,
    input  logic [8:0] pix_y,
```

```
    input  logic [9:0] pac_x,
    input  logic [8:0] pac_y,
    input  logic [1:0] pac_dir,
    input  logic [2:0] pac_frame,
    input  logic       pac_visible,
    output logic       pac_on,
    output logic [3:0] pac_colorindex
);
    localparam logic [1:0] DIR_UP    = 2'd0;
    localparam logic [1:0] DIR_RIGHT = 2'd1;
    localparam logic [1:0] DIR_DOWN  = 2'd2;
    localparam logic [1:0] DIR_LEFT  = 2'd3;
    localparam int SPR_PIXELS = SPR_W * SPR_H;
    localparam int MEM_DEPTH  = NUM_FRAMES *
SPR_PIXELS;
    localparam int ADDR_W     =
$clog2(MEM_DEPTH);
    logic [4:0] lx;
    logic [4:0] ly;
    logic       in_box;
    logic       in_box_d;
    logic [3:0] sprite_name;
    logic [3:0] sprite_pixel;
    logic [ADDR_W-1:0] addr;
    /*
     * One 4-bit palette index per pixel.
     */
    logic [3:0] sprite_mem [0:MEM_DEPTH-1];
    initial begin
        $readmemh("pacman_sprite.mem",
sprite_mem);
    end
    /*
     * Choose animation frame from direction +
pac_frame.
     */
    always_comb begin
        unique case (pac_dir)
            DIR_RIGHT: begin
                unique case (pac_frame)
                    3'd0:    sprite_name = 4'd0;
                    3'd1:    sprite_name = 4'd1;
                    3'd2:    sprite_name = 4'd2;
                    3'd3:    sprite_name = 4'd1;
                    default: sprite_name = 4'd0;
                endcase
            end
            DIR_LEFT: begin
                unique case (pac_frame)
                    3'd0:    sprite_name = 4'd3;
                    3'd1:    sprite_name = 4'd4;
                    3'd2:    sprite_name = 4'd5;
                    3'd3:    sprite_name = 4'd4;
                    default: sprite_name = 4'd3;
                endcase
            end
            DIR_UP: begin
                unique case (pac_frame)
                    3'd0:    sprite_name = 4'd6;
                    3'd1:    sprite_name = 4'd7;
                    3'd2:    sprite_name = 4'd8;
                    3'd3:    sprite_name = 4'd7;
                    default: sprite_name = 4'd6;
                endcase
            end
            default: begin // DIR_DOWN
                unique case (pac_frame)
                    3'd0:    sprite_name = 4'd9;
                    3'd1:    sprite_name = 4'd10;
                    3'd2:    sprite_name = 4'd11;
                    3'd3:    sprite_name = 4'd10;
                    default: sprite_name = 4'd9;
                endcase
            end
        endcase
    end
```

```
    /*
     * Address generation.
     *
     * This stage checks whether the current VGA pixel is inside
     * Pac-Man's 32x32 sprite box and computes the ROM address.
     */
    always_comb begin
        in_box = 1'b0;
        lx = 5'd0;
        ly = 5'd0;
        addr = '0;
        in_box =
            pac_visible &&
            (pix_x >= pac_x) &&
            (pix_x <  pac_x + SPR_W) &&
            (pix_y >= pac_y) &&
            (pix_y <  pac_y + SPR_H);
        if (in_box) begin
            lx = pix_x - pac_x;
            ly = pix_y - pac_y;
            addr = sprite_name * SPR_PIXELS + ly * SPR_W + lx;
        end
    end
    /*
     * Synchronous ROM read.
     *
     * This is better FPGA style than a fully combinational memory read.
     * sprite_pixel becomes valid one clock after addr/in_box.
     */
    always_ff @(posedge VGA_CLK) begin
        sprite_pixel <= sprite_mem[addr];
        in_box_d <= in_box;
    end
    /*
     * Output stage.
     */
    always_comb begin
        pac_on = 1'b0;
        pac_colorindex = 4'h0;
        if (in_box_d && sprite_pixel != 4'h0) begin
            pac_on = 1'b1;
            pac_colorindex = sprite_pixel;
        end
    end
endmodule


module palette_rom (
    input  logic        clk,
    input  logic [3:0]  addr,
    output logic [23:0] dout
);
    logic [23:0] mem [0:15];
    initial begin
        $readmemh("palette.mem", mem);
    end
    always_ff @(posedge clk) begin
        dout <= mem[addr];
    end
endmodule
module tileset_rom (
    input  logic        clk,
    input  logic [15:0] addr,
    output logic [3:0]  dout
);
    logic [3:0] mem [0:65535];
    initial begin
        $readmemh("tileset.mem", mem);
    end
    always_ff @(posedge clk) begin
        dout <= mem[addr];
    end
endmodule
```

```
module twoportbram
  #(parameter int DATA_BITS = 8,
    parameter int ADDRESS_BITS = 10,
    parameter int INIT_TILEMAP = 0)
   (input logic                  clk1, clk2,
    input logic [ADDRESS_BITS-1:0] addr1, addr2,
    input logic [DATA_BITS-1:0]    din1, din2,
    input logic                   we1, we2,
    output logic [DATA_BITS-1:0]   dout1, dout2);
   localparam WORDS = 1 << ADDRESS_BITS;
   /* verilator lint_off MULTIDRIVEN */
   logic [DATA_BITS-1:0] mem [WORDS-1:0];
   /* verilator lint_on MULTIDRIVEN */
   integer i;
   initial begin
      for (i = 0; i < WORDS; i = i + 1) begin
         mem[i] = {DATA_BITS{1'b0}};
      end
      if (INIT_TILEMAP != 0) begin
         $readmemh("tilemap.mem", mem);
      end
   end
   always_ff @(posedge clk1) begin
      if (we1) begin
         mem[addr1] <= din1;
         dout1 <= din1;
      end else begin
         dout1 <= mem[addr1];
      end
   end
   always_ff @(posedge clk2) begin
      if (we2) begin
         mem[addr2] <= din2;
         dout2 <= din2;
      end else begin
         dout2 <= mem[addr2];
      end
   end
endmodule

module vga_counters(
   input  logic       VGA_CLK, VGA_RESET,
   output logic [9:0] hcount, // 0-639 active,
640-799 blank/sync
   output logic [9:0] vcount, // 0-479 active,
480-524 blank/sync
   output logic       VGA_HS, VGA_VS,
VGA_BLANK_n);


   logic endOfLine;
   assign endOfLine = hcount == 10'd 799;


   always_ff @(posedge VGA_CLK or posedge
VGA_RESET)
     if (VGA_RESET)      hcount <= 10'd 797;
     else if (endOfLine) hcount <= 0;
     else                hcount <= hcount + 10'd
1;


   logic endOfFrame;
   assign endOfFrame = vcount == 10'd 524;


   always_ff @(posedge VGA_CLK or posedge
VGA_RESET)
     if (VGA_RESET)      vcount <= 10'd 524;
     else if (endOfLine)
       if (endOfFrame)   vcount <= 10'd 0;
       else              vcount <= vcount + 10'd
1;
   // 656 <= hcount <= 751
   assign VGA_HS = !( hcount[9:7] == 3'b101 &
                      hcount[6:4] != 3'b000 &
hcount[6:4] != 3'b111 );
   assign VGA_VS = !( vcount[9:1] == 9'd 245 );
// Lines 490 and 491
   // hcount < 640 && vcount < 480
   assign VGA_BLANK_n = !( hcount[9] & (hcount[8]
| hcount[7]) ) &
                        !( vcount[9] |
(vcount[8:5] == 4'b1111) );
```

```
endmodule


module vga_graphics
  (input logic         VGA_CLK,
   input logic         VGA_RESET,
   output logic [7:0]  VGA_R,
   output logic [7:0]  VGA_G,
   output logic [7:0]  VGA_B,
   output logic        VGA_HS,
   output logic        VGA_VS,
   output logic        VGA_BLANK_n,
   input logic         mem_clk,
   input logic [10:0]  tm_address,
   input logic         tm_we,
   input logic [7:0]   tm_din,
   output logic [7:0]  tm_dout,
   input logic [9:0]   pac_x,
   input logic [8:0]   pac_y,
   input logic [1:0]   pac_dir,
   input logic [2:0]   pac_frame,
   input logic         pac_visible,
   input logic [9:0]   ghost0_x,
   input logic [8:0]   ghost0_y,
   input logic [1:0]   ghost0_dir,
   input logic [2:0]   ghost0_frame,
   input logic         ghost0_visible,
   input logic [1:0]   ghost0_state,
   input logic [9:0]   ghost1_x,
   input logic [8:0]   ghost1_y,
   input logic [1:0]   ghost1_dir,
   input logic [2:0]   ghost1_frame,
   input logic         ghost1_visible,
   input logic [1:0]   ghost1_state,
   input logic [9:0]   ghost2_x,
   input logic [8:0]   ghost2_y,
   input logic [1:0]   ghost2_dir,
   input logic [2:0]   ghost2_frame,
   input logic         ghost2_visible,
   input logic [1:0]   ghost2_state,
   input logic [9:0]   ghost3_x,
   input logic [8:0]   ghost3_y,
   input logic [1:0]   ghost3_dir,
   input logic [2:0]   ghost3_frame,
   input logic         ghost3_visible,
   input logic [1:0]   ghost3_state);
   logic [9:0] hcount;
   logic [8:0] vcount;
   logic [3:0] hcount1_4;
   logic [3:0] vcount1_4;
   logic VGA_HS0, VGA_HS1, VGA_HS2;
   logic VGA_BLANK_n0, VGA_BLANK_n1,
VGA_BLANK_n2;
   logic [10:0] tmap_addr;
   logic [7:0]  tilenumber;
   logic [3:0]  tile_colorindex;
   logic       pac_on;
   logic [3:0] pac_colorindex;
   logic       pac_on_delay;
   logic [3:0] pac_color_delay;
   logic       ghost0_on, ghost1_on, ghost2_on,
ghost3_on;
   logic [3:0] ghost0_colorindex,
ghost1_colorindex, ghost2_colorindex,
ghost3_colorindex;
   logic       ghost0_on_delay, ghost1_on_delay,
ghost2_on_delay, ghost3_on_delay;
   logic [3:0] ghost0_color_delay,
ghost1_color_delay, ghost2_color_delay,
ghost3_color_delay;
   logic [3:0]  final_colorindex;
   logic [23:0] palette_rgb;
   /* verilator lint_off UNUSED */
   logic unconnected;
   /* verilator lint_on UNUSED */
   vga_counters cntrs(
      .vcount({unconnected, vcount}),
      .VGA_BLANK_n(VGA_BLANK_n0),
```

```
      .VGA_HS(VGA_HS0),
      .*
   );
   /* 16x16 tiles, 40 visible columns, 30 visible
rows, 64-entry row stride. */
   assign tmap_addr = {vcount[8:4], hcount[9:4]};
   twoportbram #(
      .DATA_BITS(8),
      .ADDRESS_BITS(11),
      .INIT_TILEMAP(1)
   ) tilemap (
      .clk1(VGA_CLK),
      .clk2(mem_clk),
      .addr1(tmap_addr),
      .we1(1'b0),
      .din1(8'h00),
      .dout1(tilenumber),
      .addr2(tm_address),
      .we2(tm_we),
      .din2(tm_din),
      .dout2(tm_dout)
   );
   pacman_sprite pacman_inst(
      .VGA_CLK(VGA_CLK),
      .pix_x(hcount),
      .pix_y(vcount),
      .pac_x(pac_x),
      .pac_y(pac_y),
      .pac_dir(pac_dir),
      .pac_frame(pac_frame),
      .pac_visible(pac_visible),
      .pac_on(pac_on),
      .pac_colorindex(pac_colorindex)
   );
   ghost_sprite #(.GHOST_COLOR(0)) ghost_red_inst
(
      .VGA_CLK(VGA_CLK),
      .pix_x(hcount),
      .pix_y(vcount),
      .ghost_x(ghost0_x),
      .ghost_y(ghost0_y),
      .ghost_dir(ghost0_dir),
      .ghost_frame(ghost0_frame),
      .ghost_visible(ghost0_visible),
      .ghost_state(ghost0_state),
      .ghost_on(ghost0_on),
      .ghost_colorindex(ghost0_colorindex)
   );
   ghost_sprite #(.GHOST_COLOR(1))
ghost_blue_inst (
      .VGA_CLK(VGA_CLK),
      .pix_x(hcount),
      .pix_y(vcount),
      .ghost_x(ghost1_x),
      .ghost_y(ghost1_y),
      .ghost_dir(ghost1_dir),
      .ghost_frame(ghost1_frame),
      .ghost_visible(ghost1_visible),
      .ghost_state(ghost1_state),
      .ghost_on(ghost1_on),
      .ghost_colorindex(ghost1_colorindex)
   );
   ghost_sprite #(.GHOST_COLOR(2))
ghost_pink_inst (
      .VGA_CLK(VGA_CLK),
      .pix_x(hcount),
      .pix_y(vcount),
      .ghost_x(ghost2_x),
      .ghost_y(ghost2_y),
      .ghost_dir(ghost2_dir),
      .ghost_frame(ghost2_frame),
      .ghost_visible(ghost2_visible),
      .ghost_state(ghost2_state),
      .ghost_on(ghost2_on),
      .ghost_colorindex(ghost2_colorindex)
   );
```

```
   ghost_sprite #(.GHOST_COLOR(3))
ghost_orange_inst (
      .VGA_CLK(VGA_CLK),
      .pix_x(hcount),
      .pix_y(vcount),
      .ghost_x(ghost3_x),
      .ghost_y(ghost3_y),
      .ghost_dir(ghost3_dir),
      .ghost_frame(ghost3_frame),
      .ghost_visible(ghost3_visible),
      .ghost_state(ghost3_state),
      .ghost_on(ghost3_on),
      .ghost_colorindex(ghost3_colorindex)
   );
   always_ff @(posedge VGA_CLK) begin
      hcount1_4 <= hcount[3:0];
      vcount1_4 <= vcount[3:0];
      VGA_BLANK_n1 <= VGA_BLANK_n0;
      VGA_HS1 <= VGA_HS0;
   end
   tileset_rom tileset_rom_inst (
      .clk(VGA_CLK),
      .addr({tilenumber, vcount1_4, hcount1_4}),
      .dout(tile_colorindex)
   );
   always_ff @(posedge VGA_CLK) begin
      pac_on_delay    <= pac_on;
      pac_color_delay <= pac_colorindex;
      ghost0_on_delay    <= ghost0_on;
      ghost0_color_delay <= ghost0_colorindex;
      ghost1_on_delay    <= ghost1_on;
      ghost1_color_delay <= ghost1_colorindex;
      ghost2_on_delay    <= ghost2_on;
      ghost2_color_delay <= ghost2_colorindex;
      ghost3_on_delay    <= ghost3_on;
      ghost3_color_delay <= ghost3_colorindex;
      VGA_BLANK_n2 <= VGA_BLANK_n1;
      VGA_HS2 <= VGA_HS1;
   end
   always_comb begin
      if (pac_on_delay) begin
         final_colorindex = pac_color_delay;
      end else if (ghost0_on_delay) begin
         final_colorindex = ghost0_color_delay;
      end else if (ghost1_on_delay) begin
         final_colorindex = ghost1_color_delay;
      end else if (ghost2_on_delay) begin
         final_colorindex = ghost2_color_delay;
      end else if (ghost3_on_delay) begin
         final_colorindex = ghost3_color_delay;
      end else begin
         final_colorindex = tile_colorindex;
      end
   end
   palette_rom palette_rom_inst (
      .clk(VGA_CLK),
      .addr(final_colorindex),
      .dout(palette_rgb)
   );
   assign {VGA_R, VGA_G, VGA_B} = palette_rgb;
   always_ff @(posedge VGA_CLK) begin
      VGA_BLANK_n <= VGA_BLANK_n2;
      VGA_HS <= VGA_HS2;
   end
endmodule
module vga_pacman_mm
  (input  logic        clk,
   input  logic        reset,
   input  logic        chipselect,
   input  logic        write,
   input  logic [16:0] address,
   input  logic [15:0] writedata,
   output logic [15:0] readdata,
   input  logic        vga_clk_in,
   input  logic        VGA_RESET,
```

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

    localparam logic [1:0] DIR_UP    = 2'd0;
    localparam logic [1:0] DIR_RIGHT = 2'd1;
    localparam logic [1:0] DIR_DOWN  = 2'd2;
    localparam logic [1:0] DIR_LEFT  = 2'd3;

    localparam logic [16:0] REG_PAC_X    =
17'h00000;
    localparam logic [16:0] REG_PAC_Y    =
17'h00001;
    localparam logic [16:0] REG_PAC_CTRL =
17'h00002;
    localparam logic [16:0] REG_READY    =
17'h00003;
    localparam logic [16:0] REG_G0_X     =
17'h00010;
    localparam logic [16:0] REG_G0_Y     =
17'h00011;
    localparam logic [16:0] REG_G0_CTRL  =
17'h00012;
    localparam logic [16:0] REG_G1_X     =
17'h00014;
    localparam logic [16:0] REG_G1_Y     =
17'h00015;
    localparam logic [16:0] REG_G1_CTRL  =
17'h00016;
    localparam logic [16:0] REG_G2_X     =
17'h00018;
    localparam logic [16:0] REG_G2_Y     =
17'h00019;
    localparam logic [16:0] REG_G2_CTRL  =
17'h0001A;
    localparam logic [16:0] REG_G3_X     =
17'h0001C;
    localparam logic [16:0] REG_G3_Y     =
17'h0001D;
    localparam logic [16:0] REG_G3_CTRL  =
17'h0001E;
    localparam logic [16:0] REG_TILE_FIRST =
17'h00100;
    localparam logic [16:0] REG_TILE_LAST  =
17'h0087F;

    /* Pac-Man Avalon-visible registers. */
    logic [9:0] pac_x_reg;
    logic [8:0] pac_y_reg;
    logic [1:0] pac_dir_reg;
    logic [2:0] pac_frame_reg;
    logic       pac_visible_reg;

     /* Four ghost Avalon-visible register sets.
*/
    logic [9:0] ghost0_x_reg, ghost1_x_reg,
ghost2_x_reg, ghost3_x_reg;
    logic [8:0] ghost0_y_reg, ghost1_y_reg,
ghost2_y_reg, ghost3_y_reg;
    logic [1:0] ghost0_dir_reg, ghost1_dir_reg,
ghost2_dir_reg, ghost3_dir_reg;
    logic [2:0] ghost0_frame_reg,
ghost1_frame_reg, ghost2_frame_reg,
ghost3_frame_reg;
    logic       ghost0_visible_reg,
ghost1_visible_reg, ghost2_visible_reg,
ghost3_visible_reg;
    logic [1:0] ghost0_state_reg,
ghost1_state_reg, ghost2_state_reg,
ghost3_state_reg;

    /* READY / VSync synchronization into Avalon
clock domain. */
    logic ready;
    logic vsync_sync1;
    logic vsync_sync2;
    logic vsync_prev;
    logic vsync_falling_edge;
    assign vsync_falling_edge = (vsync_prev ==
1'b1) && (vsync_sync2 == 1'b0);

    /* Tilemap CPU port. */
    logic [10:0] tm_address;
    logic        tm_we;
    logic [7:0]  tm_din;
    logic [7:0]  tm_dout;
    logic [16:0] tm_offset;
    assign tm_offset  = address - REG_TILE_FIRST;
    assign tm_address = tm_offset[10:0];
    assign tm_din     = writedata[7:0];
```

```
assign VGA_CLK = vga_clk_in;
vga_graphics graphics_core(
   .VGA_CLK(vga_clk_in),
   .VGA_RESET(VGA_RESET),
   .VGA_R(VGA_R),
   .VGA_G(VGA_G),
   .VGA_B(VGA_B),
   .VGA_HS(VGA_HS),
   .VGA_VS(VGA_VS),
   .VGA_BLANK_n(VGA_BLANK_n),
   .mem_clk(clk),
   .tm_address(tm_address),
   .tm_we(tm_we),
   .tm_din(tm_din),
   .tm_dout(tm_dout),
   .pac_x(pac_x_reg),
   .pac_y(pac_y_reg),
   .pac_dir(pac_dir_reg),
   .pac_frame(pac_frame_reg),
   .pac_visible(pac_visible_reg),
   .ghost0_x(ghost0_x_reg),
   .ghost0_y(ghost0_y_reg),
   .ghost0_dir(ghost0_dir_reg),
   .ghost0_frame(ghost0_frame_reg),
   .ghost0_visible(ghost0_visible_reg),
   .ghost0_state(ghost0_state_reg),
   .ghost1_x(ghost1_x_reg),
   .ghost1_y(ghost1_y_reg),
   .ghost1_dir(ghost1_dir_reg),
   .ghost1_frame(ghost1_frame_reg),
   .ghost1_visible(ghost1_visible_reg),
   .ghost1_state(ghost1_state_reg),
   .ghost2_x(ghost2_x_reg),
   .ghost2_y(ghost2_y_reg),
   .ghost2_dir(ghost2_dir_reg),
   .ghost2_frame(ghost2_frame_reg),
   .ghost2_visible(ghost2_visible_reg),
   .ghost2_state(ghost2_state_reg),
   .ghost3_x(ghost3_x_reg),
   .ghost3_y(ghost3_y_reg),
   .ghost3_dir(ghost3_dir_reg),
   .ghost3_frame(ghost3_frame_reg),
   .ghost3_visible(ghost3_visible_reg),
   .ghost3_state(ghost3_state_reg)
);
function automatic logic [15:0] pack_ctrl(
   input logic       visible,
   input logic [1:0] dir,
   input logic [2:0] frame,
   input logic [1:0] state
);
   pack_ctrl = {8'd0, state, frame, dir,
visible};
endfunction
always_ff @(posedge clk or posedge reset)
begin
   if (reset) begin
      pac_x_reg       <= 10'd304;
      pac_y_reg       <= 9'd224;
      pac_dir_reg     <= DIR_RIGHT;
      pac_frame_reg   <= 3'd0;
      pac_visible_reg <= 1'b1;
      ghost0_x_reg       <= 10'd336;
      ghost0_y_reg       <= 9'd224;
      ghost0_dir_reg     <= DIR_LEFT;
      ghost0_frame_reg   <= 3'd0;
      ghost0_visible_reg <= 1'b1;
      ghost0_state_reg   <= 2'd0;
      ghost1_x_reg       <= 10'd368;
      ghost1_y_reg       <= 9'd224;
      ghost1_dir_reg     <= DIR_RIGHT;
      ghost1_frame_reg   <= 3'd0;
      ghost1_visible_reg <= 1'b1;
      ghost1_state_reg   <= 2'd0;
      ghost2_x_reg       <= 10'd336;
```

```
         ghost2_y_reg       <= 9'd256;
         ghost2_dir_reg     <= DIR_UP;
         ghost2_frame_reg   <= 3'd0;
         ghost2_visible_reg <= 1'b1;
         ghost2_state_reg   <= 2'd0;
         ghost3_x_reg       <= 10'd368;
         ghost3_y_reg       <= 9'd256;
         ghost3_dir_reg     <= DIR_DOWN;
         ghost3_frame_reg   <= 3'd0;
         ghost3_visible_reg <= 1'b1;
         ghost3_state_reg   <= 2'd0;
         ready       <= 1'b0;
         vsync_sync1 <= 1'b1;
         vsync_sync2 <= 1'b1;
         vsync_prev  <= 1'b1;
         tm_we    <= 1'b0;
         readdata <= 16'h0000;
      end else begin
         /* Default: tilemap write-enable pulses
for one Avalon clock only. */
         tm_we    <= 1'b0;
         readdata <= 16'h0000;
         /* Cross VGA_VS into Avalon clock domain
and detect active-low VSync start. */
         vsync_sync1 <= VGA_VS;
         vsync_sync2 <= vsync_sync1;
         vsync_prev  <= vsync_sync2;
         if (vsync_falling_edge) begin
            ready <= 1'b1;
         end
         if (chipselect && write) begin
            unique case (address)
               REG_PAC_X: begin
                  pac_x_reg <= writedata[9:0];
               end
               REG_PAC_Y: begin
                  pac_y_reg <= writedata[8:0];
               end
               REG_PAC_CTRL: begin
                  pac_visible_reg <=
writedata[0];
                  pac_dir_reg     <=
writedata[2:1];
                  pac_frame_reg   <=
writedata[5:3];
               end
               REG_READY: begin
                  /*
                   * Normal software use writes 0
after finishing a frame update.
                   * Writing 1 is allowed as a
software bring-up/debug force-ready.
                   */
                  ready <= writedata[0];
               end
               REG_G0_X: begin
                  ghost0_x_reg <= writedata[9:0];
               end
               REG_G0_Y: begin
                  ghost0_y_reg <= writedata[8:0];
               end
               REG_G0_CTRL: begin
                  ghost0_visible_reg <=
writedata[0];
                  ghost0_dir_reg     <=
writedata[2:1];
                  ghost0_frame_reg   <=
writedata[5:3];
                  ghost0_state_reg   <=
writedata[7:6];
               end
               REG_G1_X: begin
                  ghost1_x_reg <= writedata[9:0];
               end
               REG_G1_Y: begin
                  ghost1_y_reg <= writedata[8:0];
               end
               REG_G1_CTRL: begin
                  ghost1_visible_reg <=
writedata[0];
```

```
                ghost1_dir_reg     <=
writedata[2:1];
                ghost1_frame_reg   <=
writedata[5:3];
                ghost1_state_reg   <=
writedata[7:6];
            end
            REG_G2_X: begin
                ghost2_x_reg <= writedata[9:0];
            end
            REG_G2_Y: begin
                ghost2_y_reg <= writedata[8:0];
            end
            REG_G2_CTRL: begin
                ghost2_visible_reg <=
writedata[0];
                ghost2_dir_reg     <=
writedata[2:1];
                ghost2_frame_reg   <=
writedata[5:3];
                ghost2_state_reg   <=
writedata[7:6];
            end
            REG_G3_X: begin
                ghost3_x_reg <= writedata[9:0];
            end
            REG_G3_Y: begin
                ghost3_y_reg <= writedata[8:0];
            end
            REG_G3_CTRL: begin
                ghost3_visible_reg <=
writedata[0];
                ghost3_dir_reg     <=
writedata[2:1];
                ghost3_frame_reg   <=
writedata[5:3];
                ghost3_state_reg   <=
writedata[7:6];
            end
            default: begin
                if (address >= REG_TILE_FIRST
&& address <= REG_TILE_LAST) begin
                    tm_we <= 1'b1;
                end
            end
        endcase
    end else if (chipselect && !write) begin
        unique case (address)
            REG_PAC_X: begin
                readdata <= {6'd0, pac_x_reg};
            end
            REG_PAC_Y: begin
                readdata <= {7'd0, pac_y_reg};
            end
            REG_PAC_CTRL: begin
                readdata <=
pack_ctrl(pac_visible_reg, pac_dir_reg,
pac_frame_reg, 2'd0);
            end
            REG_READY: begin
                readdata <= {15'd0, ready};
            end
            REG_G0_X: begin
                readdata <= {6'd0,
ghost0_x_reg};
            end
            REG_G0_Y: begin
                readdata <= {7'd0,
ghost0_y_reg};
            end
            REG_G0_CTRL: begin
                readdata <=
pack_ctrl(ghost0_visible_reg, ghost0_dir_reg,
ghost0_frame_reg, ghost0_state_reg);
            end
            REG_G1_X: begin
                readdata <= {6'd0,
ghost1_x_reg};
            end
            REG_G1_Y: begin
                readdata <= {7'd0,
ghost1_y_reg};
            end
            REG_G1_CTRL: begin
```

```
                readdata <=
pack_ctrl(ghost1_visible_reg, ghost1_dir_reg,
ghost1_frame_reg, ghost1_state_reg);

            end

            REG_G2_X: begin

                readdata <= {6'd0,
ghost2_x_reg};

            end

            REG_G2_Y: begin

                readdata <= {7'd0,
ghost2_y_reg};

            end

            REG_G2_CTRL: begin

                readdata <=
pack_ctrl(ghost2_visible_reg, ghost2_dir_reg,
ghost2_frame_reg, ghost2_state_reg);

            end

            REG_G3_X: begin

                readdata <= {6'd0,
ghost3_x_reg};

            end

            REG_G3_Y: begin

                readdata <= {7'd0,
ghost3_y_reg};

            end

            REG_G3_CTRL: begin

                readdata <=
pack_ctrl(ghost3_visible_reg, ghost3_dir_reg,
ghost3_frame_reg, ghost3_state_reg);

            end

            default: begin

                if (address >= REG_TILE_FIRST
&& address <= REG_TILE_LAST) begin

                    readdata <= {8'd0, tm_dout};

                end

            end

        endcase

    end

  end

endmodule
```

```
#include "audio_driver.h"
```

```
#include "pacman_audio.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <fcntl.h>

#include <unistd.h>

#include <sys/mman.h>

#include <sys/ioctl.h>

#include <pthread.h>

#include <stdint.h>

// Altera UP Audio IP Register offsets (Word
offsets)

#define REG_CONTROL    0

#define REG_FIFOSPACE  1

#define REG_LEFTDATA   2

#define REG_RIGHTDATA  3

static int uio_fd = -1;

static int sync_audio_fd = -1;

static volatile uint32_t *audio_regs =
MAP_FAILED;

static pthread_t audio_thread;

static volatile int run_thread = 0;

static active_sound_t active_sounds[MAX_SOUNDS];

static pthread_mutex_t mixer_mutex =
PTHREAD_MUTEX_INITIALIZER;

// Help clamping values to 16-bit signed range

static inline int16_t clamp_to_s16(int32_t val) {

    if (val > 32767) return 32767;

    if (val < -32768) return -32768;

    return (int16_t)val;

}

// Simple WAV parser (handles standard 16-bit
PCM)

sound_t* audio_load_wav(const char *filepath) {

    FILE *f = fopen(filepath, "rb");

    if (!f) {

        printf("Audio: Failed to open %s\n",
filepath);

        return NULL;

    }
```

```
    char id[4];
    uint32_t size;

    // Read RIFF header
    if (fread(id, 1, 4, f) != 4 || memcmp(id,
"RIFF", 4) != 0) {
        printf("Audio: %s is not a valid RIFF
file\n", filepath);
        fclose(f);
        return NULL;
    }

    fseek(f, 8, SEEK_SET);
    if (fread(id, 1, 4, f) != 4 || memcmp(id,
"WAVE", 4) != 0) {
        printf("Audio: %s is not a valid WAVE
file\n", filepath);
        fclose(f);
        return NULL;
    }
    // Search for fmt and data chunks
    uint16_t format_tag = 0;
    uint16_t channels = 0;
    uint32_t sample_rate = 0;
    uint16_t bits_per_sample = 0;
    uint32_t data_size = 0;
    while (fread(id, 1, 4, f) == 4) {
        if (fread(&size, 4, 1, f) != 1) break;

        long next_chunk_pos = ftell(f) + size;
        if (memcmp(id, "fmt ", 4) == 0) {
            fread(&format_tag, 2, 1, f);
            fread(&channels, 2, 1, f);
            fread(&sample_rate, 4, 1, f);
            fseek(f, 6, SEEK_CUR); // Skip bytes
per second and block align
            fread(&bits_per_sample, 2, 1, f);
            if (format_tag != 1 ||
bits_per_sample != 16) {
                printf("Audio: %s must be 16-bit
uncompressed PCM (format=%d, bits=%d)\n",
                       filepath, format_tag,
bits_per_sample);
                fclose(f);
                return NULL;
            }
        }
        else if (memcmp(id, "data", 4) == 0) {
            data_size = size;
            sound_t *sound =
malloc(sizeof(sound_t));
            sound->channels = channels;
            sound->sample_rate = sample_rate;
            sound->num_samples = data_size / 2;
// 2 bytes per sample
            sound->samples = malloc(data_size);

            if (fread(sound->samples, 1,
data_size, f) != data_size) {
                printf("Audio: Warning, read
unexpected byte count from data chunk in %s\n",
filepath);
            }

            fclose(f);
            return sound;
        }
        fseek(f, next_chunk_pos, SEEK_SET);
    }
    printf("Audio: Failed to find data chunk in
%s\n", filepath);
    fclose(f);
    return NULL;
}
void audio_free_sound(sound_t *sound) {
    if (sound) {
        free(sound->samples);
        free(sound);
    }
}
```

```
// Background thread that blocks on hardware interrupts or polls /dev/mem to mix audio samples in real time

static void* audio_processing_thread(void *arg) {

    uint32_t interrupt_info = 0;

    printf("Audio thread started (hardware interrupt mode enabled with polling fallback).\n");

    while (run_thread) {

        if (sync_audio_fd >= 0) {

            // Priority 1: Use custom pacman_audio driver's wait_interrupt ioctl (Hardware Interrupt Mode)

            if (ioctl(sync_audio_fd, PACMAN_AUDIO_WAIT_INTERRUPT) < 0) {

                if (!run_thread) break; // Exit cleanly if shutting down

                perror("Audio synchronization ioctl failed");

                usleep(500); // Sleep briefly to prevent an infinite tight loop if ioctl fails

            }

        } else if (uio_fd >= 0) {

            // Priority 2: Use standard UIO driver (Standard Linux interrupt mode)

            ssize_t bytes_read = read(uio_fd, &interrupt_info, sizeof(interrupt_info));

            if (bytes_read < 0) {

                if (!run_thread) break; // Exit cleanly if shutting down

                perror("Audio UIO read failed");

                break;

            }

        } else {

            // Priority 3: No drivers available (Direct physical polling mode)

            // 500us is a perfect interval (gives 2000Hz polling rate, extremely light on CPU)

            usleep(500);

        }

        pthread_mutex_lock(&mixer_mutex);

        // Fetch how many empty spaces we have in Left and Right FIFOs

        uint32_t fifo_space = audio_regs[REG_FIFOSPACE];

        int write_space_l = (fifo_space >> 24) & 0xFF;

        int write_space_r = (fifo_space >> 16) & 0xFF;

        int fill_count = (write_space_l < write_space_r) ? write_space_l : write_space_r;

        // Feed the FIFOs with mixed audio samples

        while (fill_count > 0) {

            int32_t mixed_l = 0;

            int32_t mixed_r = 0;

            for (int i = 0; i < MAX_SOUNDS; i++) {

                active_sound_t *active = &active_sounds[i];

                if (!active->active || !active->sound) continue;

                sound_t *snd = active->sound;

                uint32_t idx = active->current_index;

                int16_t sample_l = 0;

                int16_t sample_r = 0;

                if (snd->channels == 2) {

                    // Stereo

                    if (idx < snd->num_samples - 1) {

                        sample_l = snd->samples[idx];

                        sample_r = snd->samples[idx + 1];

                    }

                    active->current_index += 2;

                } else {

                    // Mono (duplicate to both channels)

                    if (idx < snd->num_samples) {

                        sample_l = snd->samples[idx];

                        sample_r = sample_l;

                    }

                    active->current_index += 1;

                }

                mixed_l += sample_l;
```

```
                mixed_r += sample_r;

                // Handle completion / looping

                if (active->current_index >= snd->num_samples) {

                    if (active->loop) {

                        active->current_index = 0;

                    } else {

                        active->active = 0;

                    }

                }

            }

            // Scale digital volume down by 4x (-12dB)

            mixed_l = mixed_l / 4;

            mixed_r = mixed_r / 4;

            // Write the clamped stereo sample to FIFOs

            audio_regs[REG_LEFTDATA] = clamp_to_s16(mixed_l);

            audio_regs[REG_RIGHTDATA] = clamp_to_s16(mixed_r);

            fill_count--;

        }

        pthread_mutex_unlock(&mixer_mutex);

        if (uio_fd >= 0) {

            // Tell the generic-uio driver to unmask the interrupt line for the next cycle

            uint32_t reenable = 1;

            if (write(uio_fd, &reenable, sizeof(reenable)) < 0) {

                if (!run_thread) break; // Exit cleanly if shutting down

                perror("Audio UIO write failed");

                break;

            }

        }

    }

    printf("Audio thread exiting.\n");

    return NULL;

}
```

```
int audio_init(void) {

    memset(active_sounds, 0, sizeof(active_sounds));

    // Try to open pacman_audio for hardware interrupt synchronization first

    sync_audio_fd = open("/dev/pacman_audio", O_RDWR);

    if (sync_audio_fd >= 0) {

        printf("Audio: Opened /dev/pacman_audio successfully. Testing interrupt sync...\n");

        // Map physical audio registers directly via /dev/mem

        int mem_fd = open("/dev/mem", O_RDWR | O_SYNC);

        if (mem_fd >= 0) {

            audio_regs = (volatile uint32_t *)mmap(NULL, 0x1000, PROT_READ | PROT_WRITE, MAP_SHARED, mem_fd, 0xff240000);

            close(mem_fd);

            if (audio_regs != MAP_FAILED) {

                printf("Audio: Initialized via custom /dev/pacman_audio hardware interrupt sync!\n");

                // Clear FIFOs, set write threshold interrupt trigger (the driver manages GIC mask directly)

                audio_regs[REG_CONTROL] = 0xC;  // Clear FIFOs

                audio_regs[REG_CONTROL] = 0x2;  // Set WE bit to trigger interrupts when low

            } else {

                perror("Audio: Failed to mmap audio registers for pacman_audio mode");

                close(sync_audio_fd);

                sync_audio_fd = -1;

            }

        } else {

            perror("Audio: Failed to open /dev/mem for pacman_audio mode");

            close(sync_audio_fd);

            sync_audio_fd = -1;

        }

    }

    if (sync_audio_fd < 0) {
```

```
        // Fallback to standard UIO
        uio_fd = open("/dev/uio0", O_RDWR);
        if (uio_fd >= 0) {
            // Map UIO registers (offset is 0 for
UIO device mapping)
            audio_regs = (volatile uint32_t
*)mmap(NULL, 0x1000, PROT_READ | PROT_WRITE,
MAP_SHARED, uio_fd, 0);
            if (audio_regs == MAP_FAILED) {
                perror("Audio: Failed to mmap UIO
registers");
                close(uio_fd);
                uio_fd = -1;
            } else {
                printf("Audio: Initialized via
UIO interrupt mode (/dev/uio0)\n");
                audio_regs[REG_CONTROL] = 0xC;
                audio_regs[REG_CONTROL] = 0x2;
            }
        }
    }
    if (sync_audio_fd < 0 && uio_fd < 0) {
        printf("Audio: No interrupt drivers
available. Falling back to direct /dev/mem
polling mode...\n");
        int mem_fd = open("/dev/mem", O_RDWR |
O_SYNC);
        if (mem_fd < 0) {
            perror("Audio: Failed to open
/dev/mem");
            return -1;
        }
        audio_regs = (volatile uint32_t
*)mmap(NULL, 0x1000, PROT_READ | PROT_WRITE,
MAP_SHARED, mem_fd, 0xff240000);
        close(mem_fd); // File descriptor can be
closed immediately after mapping
        if (audio_regs == MAP_FAILED) {
            perror("Audio: Failed to mmap
physical registers via /dev/mem");
            return -1;
        }
        printf("Audio: Direct /dev/mem mapping
successful!\n");
        audio_regs[REG_CONTROL] = 0xC;  // Clear
FIFOs
        audio_regs[REG_CONTROL] = 0x0;  //
Disable interrupts since we are in polling mode
    }
    // Start background thread
    run_thread = 1;
    if (pthread_create(&audio_thread, NULL,
audio_processing_thread, NULL) != 0) {
        perror("Audio: Failed to create thread");
        if (audio_regs != MAP_FAILED) {
            munmap((void *)audio_regs, 0x1000);
        }
        if (uio_fd >= 0) {
            close(uio_fd);
        }
        if (sync_audio_fd >= 0) {
            close(sync_audio_fd);
        }
        return -1;
    }
    printf("Audio subsystem initialized
successfully!\n");
    return 0;
}
void audio_shutdown(void) {
    if (run_thread) {
        run_thread = 0;

        if (sync_audio_fd >= 0) {
            // Closing the fd automatically wakes
up any blocking ioctls inside the kernel
            close(sync_audio_fd);
            sync_audio_fd = -1;
        }

        if (uio_fd >= 0) {
            // Write to UIO fd to break block if
thread is waiting
            uint32_t reenable = 1;
```

```
            write(uio_fd, &reenable,
sizeof(reenable));
            close(uio_fd);
            uio_fd = -1;
        }

        pthread_join(audio_thread, NULL);
    }
    if (audio_regs != MAP_FAILED) {
        audio_regs[REG_CONTROL] = 0x0; // Disable
interrupts
        munmap((void *)audio_regs, 0x1000);
    }

    printf("Audio subsystem shut down.\n");
}
int audio_play(sound_t *sound, int loop) {
    if (!sound) return -1;
    pthread_mutex_lock(&mixer_mutex);

    int channel_idx = -1;
    for (int i = 0; i < MAX_SOUNDS; i++) {
        if (!active_sounds[i].active) {
            channel_idx = i;
            break;
        }
    }
    if (channel_idx != -1) {
        active_sounds[channel_idx].sound = sound;
        active_sounds[channel_idx].current_index
= 0;
        active_sounds[channel_idx].loop = loop;
        active_sounds[channel_idx].active = 1;
    } else {
        printf("Audio: Maximum sound channels
reached!\n");
    }
    pthread_mutex_unlock(&mixer_mutex);
    return channel_idx;
}
void audio_stop(int handle) {
    if (handle < 0 || handle >= MAX_SOUNDS)
return;

    pthread_mutex_lock(&mixer_mutex);
    active_sounds[handle].active = 0;
    pthread_mutex_unlock(&mixer_mutex);
}
#ifndef AUDIO_DRIVER_H
#define AUDIO_DRIVER_H
#include <stdint.h>
#define MAX_SOUNDS 8
// Structure representing a loaded sound effect
in memory
typedef struct {
    int16_t *samples;
    uint32_t num_samples;
    int channels;          // 1 (mono) or 2
(stereo)
    int sample_rate;       // e.g., 44100
} sound_t;
// Structure representing an active playing
instance of a sound
typedef struct {
    sound_t *sound;
    uint32_t current_index;
    int loop;
    int active;
} active_sound_t;
// Initialize the audio subsystem (opens UIO
device, maps registers, starts thread)
int audio_init(void);
// Shut down the audio subsystem and free
resources
void audio_shutdown(void);
// Load a standard WAV file into memory
sound_t* audio_load_wav(const char *filepath);
// Free a loaded sound
void audio_free_sound(sound_t *sound);
```

```
// Play a loaded sound (returns handle index, or
-1 if full)

int audio_play(sound_t *sound, int loop);

// Stop a playing sound instance

void audio_stop(int handle);

#endif // AUDIO_DRIVER_H

/* SPDX-License-Identifier: MIT

 * draw_numbered_wall_maze.c

 *

 * Draws the current Pac-Man 40x30 tile layout.

 *

 * Spreadsheet key:

 *   .  = unused outside area / blank

 *   0  = empty open tile

 *   9  = normal pellet

 *   A  = super pellet, from spreadsheet value 10

 *   1  = vertical left wall

 *   2  = vertical right wall

 *   3  = horizontal top wall

 *   4  = horizontal bottom wall

 *   5  = curve right top

 *   6  = curve left top

 *   7  = curve right bottom

 *   8  = curve left bottom

 */

#include <errno.h>

#include <fcntl.h>

#include <stdint.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/ioctl.h>

#include <unistd.h>

#include "vga_pacman_ioctl.h"

#define TILE_EMPTY              0x00

#define TILE_DOT                0x05

#define TILE_POWER_DOT          0x3A

#define TILE_VERT_LEFT          0x30

#define TILE_VERT_RIGHT         0x36

#define TILE_HORIZ_TOP          0x49

#define TILE_HORIZ_BOTTOM       0x24

#define TILE_CURVE_RIGHT_TOP    0x4A

#define TILE_CURVE_LEFT_TOP     0x4B

#define TILE_CURVE_RIGHT_BOTTOM 0x27

#define TILE_CURVE_LEFT_BOTTOM  0x29

#define MAP_H VGA_PACMAN_VISIBLE_TILE_ROWS

#define MAP_W VGA_PACMAN_VISIBLE_TILE_COLS

static const char *tile_map[MAP_H] = {

    "....................................",

    "....................................",

    "....................................",

    "....................................",

    "533333333333333333333333333333333336",

    "1A99999999999999999999999999999999A2",

    "190000000000000000000000000000000092",

    "190533333369053333333333360953333336092",

    "190744444489074444444444480974444448092",

    "190000000009099999999999990900000000092",

    "199999999990000000000000000999999999992",

    "190569010929053333333333360919020956092",

    "190129010929074444444444480919020912092",

    "890789010929099999999999990919020978097",

    "099999010929000000000000000919020900090",

    "090009010929053333300333360919020999990",

    "690569010929010000000000020919020956095",

    "190129010929010000000000020919020912092",

    "190129999999074444444444480999999912092",

    "190120000009000000000000000900000012092",

    "190173333369099999999999990953333382092",

    "190100000029053333333333360910000002092",

    "190744444489074444444444480974444448092",

    "199999999999999999999999999999999999992",

    "1A000000000000000000000000000000000A2",

    "744444444444444444444444444444444444448",

    "....................................",
```

```
    "....................................",
    "....................................",
    "...................................."
};
static int validate_tile_map(void)
{
    int row;
    int len;
    for (row = 0; row < MAP_H; row++) {
        len = strlen(tile_map[row]);
        if (len != MAP_W) {
            printf("ERROR: tile_map row %d has
length %d, expected %d\n",
                   row,
                   len,
                   MAP_W);
            return -1;
        }
    }
    return 0;
}
static uint8_t tile_from_char(char c)
{
    if (c == '.') {
        return TILE_EMPTY;
    }
    if (c == '0') {
        return TILE_EMPTY;
    }
    if (c == '9') {
        return TILE_DOT;
    }
    if (c == 'A') {
        return TILE_POWER_DOT;
    }
    if (c == '1') {
        return TILE_VERT_LEFT;
    }
    if (c == '2') {
        return TILE_VERT_RIGHT;
    }
    if (c == '3') {
        return TILE_HORIZ_TOP;
    }
    if (c == '4') {
        return TILE_HORIZ_BOTTOM;
    }
    if (c == '5') {
        return TILE_CURVE_RIGHT_TOP;
    }
    if (c == '6') {
        return TILE_CURVE_LEFT_TOP;
    }
    if (c == '7') {
        return TILE_CURVE_RIGHT_BOTTOM;
    }
    if (c == '8') {
        return TILE_CURVE_LEFT_BOTTOM;
    }
    return TILE_EMPTY;
}
static int write_tile_hw(int fd, uint16_t row,
uint16_t col, uint8_t tile_id)
{
    struct vga_pacman_tile t;
    memset(&t, 0, sizeof(t));
    t.row = row;
    t.col = col;
    t.tile_id = tile_id;
    return ioctl(fd, VGA_PACMAN_SET_TILE, &t);
}
int main(void)
{
    int fd;
    int row;
    int col;
```

```
    int count_empty;
    int count_dot;
    int count_power;
    int count_wall;
    uint8_t tile_id;
    char c;
    if (validate_tile_map() != 0) {
        return 1;
    }
    fd = open(VGA_PACMAN_DEVICE, O_RDWR);
    if (fd < 0) {
        perror("open " VGA_PACMAN_DEVICE);
        printf("Make sure the driver is loaded first:\n");
        printf("  insmod ./vga_pacman.ko\n");
        return 1;
    }
    count_empty = 0;
    count_dot = 0;
    count_power = 0;
    count_wall = 0;
    printf("Drawing numbered wall maze with pellets and power pellets...\n");
    for (row = 0; row < MAP_H; row++) {
        for (col = 0; col < MAP_W; col++) {
            c = tile_map[row][col];
            tile_id = tile_from_char(c);
            if (c == '.' || c == '0') {
                count_empty++;
            }
            else if (c == '9') {
                count_dot++;
            }
            else if (c == 'A') {
                count_power++;
            }
            else {
                count_wall++;
            }
            if (write_tile_hw(fd, (uint16_t)row, (uint16_t)col, tile_id) < 0) {
                perror("VGA_PACMAN_SET_TILE");
                close(fd);
                return 1;
            }
        }
    }
    printf("Done.\n");
    printf("Wall tiles:       %d\n", count_wall);
    printf("Normal pellets:   %d\n", count_dot);
    printf("Power pellets:    %d\n", count_power);
    printf("Empty tiles:      %d\n", count_empty);
    printf("Total tiles:      %d\n", count_wall + count_dot + count_power + count_empty);
    close(fd);
    return 0;
}
/* SPDX-License-Identifier: MIT
 * draw_pellet_wall_maze.c
 *
 * Debug utility for DE1-SoC Pac-Man VGA tilemap.
 *
 * This draws a 40x30 debug maze where:
 *   '#' = draw pellet marker tile
 *   '.' = draw blank tile
 *
 * The purpose is NOT to make the final game look correct.
 * The purpose is to create a clean visible graph-paper reference.
 */
#include <errno.h>
#include <fcntl.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include "vga_pacman_ioctl.h"
#define TILE_EMPTY 0x00
#define TILE_MARKER 0x05
#define MAP_H VGA_PACMAN_VISIBLE_TILE_ROWS
#define MAP_W VGA_PACMAN_VISIBLE_TILE_COLS
/* 40 columns x 30 rows.
 *
 * '#' = marker pellet wall
 * '.' = blank/open
 *
 * This is the updated 2x2-sprite-aware version
from your table.
 */
static const char *debug_maze[MAP_H] = {
    "....................................",
    "....................................",
    "....................................",
    "....................................",
    "########################################",
    "#..................................#",
    "#..................................#",
    "#..########..##############..########..#",
    "#..########..##############..########..#",
    "#..................................#",
    "#..................................#",
    "#..##..#..#..##############..#..#..##..#",
    "#..##..#..#..##############..#..#..##..#",
    "#..##..#..#................#..#..##..#",
    ".......#..#................#..#.......",
    ".......#..#..######..######..#..#.......",
    "#..##..#..#..##.........##..#..#..##..#",
    "#..##..#..#..##.........##..#..#..##..#",
    "#..##.......##############.......##..#",
    "#..##..........................##..#",
    "#..########................########..#",
    "#..########..##############..########..#",
    "#..########..##############..########..#",
    "#..................................#",
    "#..................................#",
    "########################################",
    "....................................",
    "....................................",
    "...................................."
};
static int validate_debug_maze(void)
{
    int row;
    int len;
    for (row = 0; row < MAP_H; row++) {
        len = strlen(debug_maze[row]);
        if (len != MAP_W) {
            printf("ERROR: debug_maze row %d has
length %d, expected %d\n",
                   row,
                   len,
                   MAP_W);
            return -1;
        }
    }
    return 0;
}
static int write_tile_hw(int fd, uint16_t row,
uint16_t col, uint8_t tile_id)
{
    struct vga_pacman_tile t;
    memset(&t, 0, sizeof(t));
    t.row = row;
    t.col = col;
    t.tile_id = tile_id;
    return ioctl(fd, VGA_PACMAN_SET_TILE, &t);
}
int main(void)
```

```
{
    int fd;
    int row;
    int col;
    uint8_t tile_id;
    int marker_count;
    int blank_count;
    if (validate_debug_maze() != 0) {
        return 1;
    }
    fd = open(VGA_PACMAN_DEVICE, O_RDWR);
    if (fd < 0) {
        perror("open " VGA_PACMAN_DEVICE);
        printf("Make sure the driver is loaded first:\n");
        printf("  insmod ./vga_pacman.ko\n");
        return 1;
    }
    marker_count = 0;
    blank_count = 0;
    printf("Drawing pellet-wall debug maze...\n");
    for (row = 0; row < MAP_H; row++) {
        for (col = 0; col < MAP_W; col++) {
            if (debug_maze[row][col] == '#') {
                tile_id = TILE_MARKER;
                marker_count++;
            }
            else {
                tile_id = TILE_EMPTY;
                blank_count++;
            }
            if (write_tile_hw(fd, (uint16_t)row, (uint16_t)col, tile_id) < 0) {
                perror("VGA_PACMAN_SET_TILE");
                close(fd);
                return 1;
            }
        }
    }
    printf("Done.\n");
    printf("Marker tiles: %d\n", marker_count);
    printf("Blank tiles:  %d\n", blank_count);
    printf("Total tiles:  %d\n", marker_count + blank_count);
    printf("To restore the original map, reboot or reload the FPGA bitstream.\n");
    close(fd);
    return 0;
}
/* SPDX-License-Identifier: MIT
 * draw_tile_index_rows.c
 *
 * DE1-SoC Pac-Man VGA tile index viewer.
 *
 * This draws one tile ID per row across the whole 40-column screen.
 *
 * Default:
 *   row 0  = tile 0
 *   row 1  = tile 1
 *   row 2  = tile 2
 *   ...
 *   row 29 = tile 29
 *
 * Optional base argument:
 *   ./draw_tile_index_rows 0x20
 *
 * Then:
 *   row 0  = tile 0x20
 *   row 1  = tile 0x21
 *   ...
 *   row 29 = tile 0x3d
 *
 * This lets you page through the tileset and identify which tile ID
 * corresponds to each wall/sprite/tile graphic.
 */
```

```
#include <errno.h>
#include <fcntl.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include "vga_pacman_ioctl.h"
#define MAP_H VGA_PACMAN_VISIBLE_TILE_ROWS
#define MAP_W VGA_PACMAN_VISIBLE_TILE_COLS
static int parse_u8_value(const char *text,
uint8_t *out)
{
    char *endptr;
    unsigned long value;
    if (text == NULL) {
        return -1;
    }
    errno = 0;
    value = strtoul(text, &endptr, 0);
    if (errno != 0) {
        return -1;
    }
    if (endptr == text || *endptr != '\0') {
        return -1;
    }
    if (value > 255) {
        return -1;
    }
    *out = (uint8_t)value;
    return 0;
}
static int write_tile_hw(int fd, uint16_t row,
uint16_t col, uint8_t tile_id)
{
    struct vga_pacman_tile t;
    memset(&t, 0, sizeof(t));
    t.row = row;
    t.col = col;
    t.tile_id = tile_id;
    return ioctl(fd, VGA_PACMAN_SET_TILE, &t);
}
int main(int argc, char **argv)
{
    int fd;
    int row;
    int col;
    uint8_t base_tile;
    uint8_t tile_id;
    int write_count;
    base_tile = 0x00;
    if (argc > 2) {
        printf("Usage:\n");
        printf("  %s\n", argv[0]);
        printf("  %s <base_tile_id>\n", argv[0]);
        printf("\nExamples:\n");
        printf("  %s 0x00\n", argv[0]);
        printf("  %s 0x20\n", argv[0]);
        printf("  %s 0x40\n", argv[0]);
        return 1;
    }
    if (argc == 2) {
        if (parse_u8_value(argv[1], &base_tile)
!= 0) {
            printf("Bad base tile ID: %s\n",
argv[1]);
            return 1;
        }
    }
    fd = open(VGA_PACMAN_DEVICE, O_RDWR);
    if (fd < 0) {
        perror("open " VGA_PACMAN_DEVICE);
        printf("Make sure the driver is loaded
first:\n");
        printf("  insmod ./vga_pacman.ko\n");
        return 1;
```

```
    }

    printf("Drawing tile index rows.\n");

    printf("Base tile ID: 0x%02x\n", base_tile);

    printf("Each row is filled with one tile
ID:\n");

    write_count = 0;

    for (row = 0; row < MAP_H; row++) {

        tile_id = (uint8_t)(base_tile + row);

        printf("  row %2d = tile 0x%02x (%3u)\n",
row, tile_id, tile_id);

        for (col = 0; col < MAP_W; col++) {

            if (write_tile_hw(fd, (uint16_t)row,
(uint16_t)col, tile_id) < 0) {

                perror("VGA_PACMAN_SET_TILE");

                close(fd);

                return 1;

            }

            write_count++;

        }

    }

    printf("Done. Wrote %d tiles.\n",
write_count);

    printf("To see the next page of tile IDs, run
something like:\n");

    printf("  %s 0x20\n", argv[0]);

    printf("  %s 0x40\n", argv[0]);

    printf("  %s 0x60\n", argv[0]);

    close(fd);

    return 0;

}

/* SPDX-License-Identifier: MIT

 * fill_screen_tiles.c

 *

 * Utility program for DE1-SoC Pac-Man VGA
debugging.

 *

 * This fills the entire visible 40x30 tilemap
with one tile ID.

 * Use it to make every visible tile show a
pellet or another marker tile

 * so that the physical VGA screen can be mapped
onto graph paper.

 *

 * Usage:

 *     ./fill_screen_tiles

 *     ./fill_screen_tiles 0x05

 *     ./fill_screen_tiles 0x3A

 *     ./fill_screen_tiles 0x00

 *

 * Default:

 *     0x05 = normal pellet tile, based on the
current project code.

 */

#include <errno.h>

#include <fcntl.h>

#include <stdint.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/ioctl.h>

#include <unistd.h>

#include "vga_pacman_ioctl.h"

#define DEFAULT_TILE_ID 0x05

static int parse_tile_id(const char *text,
uint8_t *tile_id)

{

    char *endptr;

    unsigned long value;

    if (text == NULL) {

        return 0;

    }

    errno = 0;

    value = strtoul(text, &endptr, 0);

    if (errno != 0) {

        return -1;

    }

    if (endptr == text || *endptr != '\0') {

        return -1;

    }
```

```
    if (value > 255) {
        return -1;
    }
    *tile_id = (uint8_t)value;
    return 0;
}

static int write_tile_hw(int fd, uint16_t row,
uint16_t col, uint8_t tile_id)
{
    struct vga_pacman_tile t;
    memset(&t, 0, sizeof(t));
    t.row = row;
    t.col = col;
    t.tile_id = tile_id;
    return ioctl(fd, VGA_PACMAN_SET_TILE, &t);
}

int main(int argc, char **argv)
{
    int fd;
    int row;
    int col;
    uint8_t tile_id;
    int write_count;
    tile_id = DEFAULT_TILE_ID;
    if (argc > 2) {
        printf("Usage:\n");
        printf("  %s\n", argv[0]);
        printf("  %s <tile_id>\n", argv[0]);
        printf("\nExamples:\n");
        printf("  %s 0x05\n", argv[0]);
        printf("  %s 0x3A\n", argv[0]);
        printf("  %s 0x00\n", argv[0]);
        return 1;
    }
    if (argc == 2) {
        if (parse_tile_id(argv[1], &tile_id) !=
0) {
            printf("Bad tile ID: %s\n", argv[1]);
            return 1;
        }
    }
    fd = open(VGA_PACMAN_DEVICE, O_RDWR);
    if (fd < 0) {
        perror("open " VGA_PACMAN_DEVICE);
        printf("Make sure the driver is loaded
first:\n");
        printf("  insmod ./vga_pacman.ko\n");
        return 1;
    }
    printf("Filling visible tilemap with tile ID
0x%02x\n", tile_id);
    printf("Rows: 0..%d, Cols: 0..%d\n",
           VGA_PACMAN_VISIBLE_TILE_ROWS - 1,
           VGA_PACMAN_VISIBLE_TILE_COLS - 1);
    write_count = 0;
    for (row = 0; row <
VGA_PACMAN_VISIBLE_TILE_ROWS; row++) {
        for (col = 0; col <
VGA_PACMAN_VISIBLE_TILE_COLS; col++) {
            if (write_tile_hw(fd, (uint16_t)row,
(uint16_t)col, tile_id) < 0) {
                perror("VGA_PACMAN_SET_TILE");
                close(fd);
                return 1;
            }
            write_count++;
        }
    }
    printf("Done. Wrote %d tiles.\n",
write_count);
    printf("To restore the original FPGA tilemap,
reboot or reload the FPGA bitstream.\n");
    close(fd);
    return 0;
}

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
```

```c
#include <signal.h>

#include <unistd.h>

#include <limits.h>

#include <fcntl.h>

#include <string.h>

#include <errno.h>

#include <sys/ioctl.h>

#include <libusb-1.0/libusb.h>

#include "vga_pacman.h"

/* ================= USB: KIWITATA / SNES-style
controller ================= */

#define VENDOR_ID  0x0079

#define PRODUCT_ID 0x0011

#define INTERFACE_NUM 0

#define ENDPOINT_IN   0x81

#define DPAD_X_BYTE 3

#define DPAD_Y_BYTE 4

#define DPAD_CENTER 0x7f

#define DPAD_LOW    0x00

#define DPAD_HIGH   0xff

/* ================= GAME / HARDWARE TIMING
================= */

#define GHOST_SPEED_DIV 5

#define FRAME_SLEEP_US 70000

#define TILE_SIZE 16

/* Offsets for drawing the small demo maze on the
VGA tile field.

   Change these if you want to shift the entire
game board on screen. */

#define SCREEN_TILE_X0 0

#define SCREEN_TILE_Y0 0

/* Sprite position adjustment.

   Your sprites are 32x32 while the logical grid
is tile-based.

   This places the sprite's top-left at
tile_x*16, tile_y*16. */

#define SPRITE_X_OFFSET 0

#define SPRITE_Y_OFFSET 0

/* ================= TYPES ================= */

typedef enum {

    DIR_UP = 0,

    DIR_LEFT = 1,

    DIR_DOWN = 2,

    DIR_RIGHT = 3,

    DIR_NONE = 4

} Direction;

typedef enum {

    MODE_CHASE,

    MODE_SCATTER,

    MODE_FRIGHTENED

} GhostMode;

typedef enum {

    GHOST_BLINKY,

    GHOST_PINKY,

    GHOST_INKY,

    GHOST_CLYDE

} GhostType;

typedef struct {

    int x;

    int y;

} Vec2;

typedef struct {

    Vec2 pos;

    Direction dir;

} PacmanState;

typedef struct {

    GhostType type;

    Vec2 pos;

    Direction dir;

    GhostMode mode;

    Vec2 scatter_target;

    const char *name;

    char symbol;

    uint32_t hw_base;

} Ghost;

/* ================= TERMINAL MAZE MODEL
================= */

#define MAZE_H 11
```

```
#define MAZE_W 19
static const char *maze[MAZE_H] = {
    "###################",
    "#........#........#",
    "#.###.##.#.##.###.#",
    "#.#.............#.#",
    "#.#.###.###.###.#.#",
    "#.....#.....#.....#",
    "###.#.#.###.#.#.###",
    "#...#...#.#...#...#",
    "#.#####.#.#.#####.#",
    "#.................#",
    "###################"
};
static const Vec2 dir_vec[4] = {
    { 0, -1 },
    { -1, 0 },
    { 0, 1 },
    { 1, 0 }
};
/* ================= GLOBALS ================= */
static int running = 1;
static Direction last_dpad_dir = DIR_NONE;
static int last_x_raw = -1;
static int last_y_raw = -1;
static int last_packet_len = 0;
static unsigned char last_packet[64];
/* ================= SIGNAL ================= */
void handle_sigint(int sig) {
    running = 0;
}
/* ================= BASIC UTILS
================= */
const char *dir_name(Direction d) {
    switch (d) {
        case DIR_UP: return "UP";
        case DIR_LEFT: return "LEFT";
        case DIR_DOWN: return "DOWN";
        case DIR_RIGHT: return "RIGHT";
        default: return "NONE";
    }
}
Direction opposite_direction(Direction d) {
    if (d == DIR_UP) return DIR_DOWN;
    if (d == DIR_DOWN) return DIR_UP;
    if (d == DIR_LEFT) return DIR_RIGHT;
    if (d == DIR_RIGHT) return DIR_LEFT;
    return DIR_NONE;
}
Vec2 vec_add(Vec2 a, Vec2 b) {
    Vec2 out;
    out.x = a.x + b.x;
    out.y = a.y + b.y;
    return out;
}
int squared_distance(Vec2 a, Vec2 b) {
    int dx;
    int dy;
    dx = a.x - b.x;
    dy = a.y - b.y;
    return dx * dx + dy * dy;
}
int in_bounds(Vec2 p) {
    if (p.x < 0 || p.x >= MAZE_W) return 0;
    if (p.y < 0 || p.y >= MAZE_H) return 0;
    return 1;
}
int is_wall(Vec2 p) {
    if (!in_bounds(p)) return 1;
    return maze[p.y][p.x] == '#';
}
int is_legal_move(Vec2 pos, Direction dir) {
    Vec2 next;
    if (dir == DIR_NONE) return 0;
    next = vec_add(pos, dir_vec[(int)dir]);
```

```
    return !is_wall(next);
}

/* ================= HARDWARE IOCTL HELPERS
================= */

int dir_to_hw(Direction d) {
    if (d == DIR_UP) return HW_DIR_UP;
    if (d == DIR_RIGHT) return HW_DIR_RIGHT;
    if (d == DIR_DOWN) return HW_DIR_DOWN;
    if (d == DIR_LEFT) return HW_DIR_LEFT;
    return HW_DIR_RIGHT;
}

int tile_to_pixel_x(int tile_x) {
    return (SCREEN_TILE_X0 + tile_x) * TILE_SIZE
+ SPRITE_X_OFFSET;
}

int tile_to_pixel_y(int tile_y) {
    return (SCREEN_TILE_Y0 + tile_y) * TILE_SIZE
+ SPRITE_Y_OFFSET;
}

void hw_write_reg(int fd, uint32_t reg, uint32_t
value) {
    struct vga_pacman_reg r;
    r.reg = reg;
    r.value = value;
    if (ioctl(fd, VGA_PACMAN_WRITE_REG, &r) < 0)
{
        printf("WRITE_REG failed: %s\n",
strerror(errno));
    }
}

uint32_t hw_read_reg(int fd, uint32_t reg) {
    struct vga_pacman_reg r;
    r.reg = reg;
    r.value = 0;
    if (ioctl(fd, VGA_PACMAN_READ_REG, &r) < 0) {
        printf("READ_REG failed: %s\n",
strerror(errno));
        return 0;
    }
    return r.value;
}

void hw_write_entity(
    int fd,
    uint32_t base_reg,
    int x_px,
    int y_px,
    Direction dir,
    int frame,
    int visible
) {
    struct vga_pacman_entity e;
    e.base_reg = base_reg;
    e.x = (uint32_t)x_px;
    e.y = (uint32_t)y_px;
    e.dir = (uint32_t)dir_to_hw(dir);
    e.frame = (uint32_t)frame;
    e.visible = (uint32_t)visible;
    if (ioctl(fd, VGA_PACMAN_WRITE_ENTITY, &e) <
0) {
        printf("WRITE_ENTITY failed: %s\n",
strerror(errno));
    }
}

void hw_write_tile(int fd, uint32_t x, uint32_t
y, uint32_t tile_id) {
    struct vga_pacman_tile t;
    t.x = x;
    t.y = y;
    t.tile_id = tile_id;
    if (ioctl(fd, VGA_PACMAN_WRITE_TILE, &t) < 0)
{
        printf("WRITE_TILE failed: %s\n",
strerror(errno));
    }
}

void hw_wait_ready(int fd) {
    int timeout;
    uint32_t ready;
    timeout = 100000;
    while (timeout > 0) {
```

```
        ready = hw_read_reg(fd, REG_READY);
        if ((ready & 1) != 0) {
            return;
        }
        timeout--;
    }
}
void hw_clear_ready(int fd) {
    hw_write_reg(fd, REG_READY, 0);
}
void draw_to_vga(
    int fd,
    PacmanState pac,
    Ghost blinky,
    Ghost pinky,
    Ghost inky,
    Ghost clyde,
    int tick_count
) {
    int frame;
    int pac_x;
    int pac_y;
    int bx;
    int by;
    int px;
    int py;
    int ix;
    int iy;
    int cx;
    int cy;
    frame = (tick_count / 4) & 3;
    pac_x = tile_to_pixel_x(pac.pos.x);
    pac_y = tile_to_pixel_y(pac.pos.y);
    bx = tile_to_pixel_x(blinky.pos.x);
    by = tile_to_pixel_y(blinky.pos.y);
    px = tile_to_pixel_x(pinky.pos.x);
    py = tile_to_pixel_y(pinky.pos.y);
    ix = tile_to_pixel_x(inky.pos.x);
    iy = tile_to_pixel_y(inky.pos.y);
    cx = tile_to_pixel_x(clyde.pos.x);
    cy = tile_to_pixel_y(clyde.pos.y);
    hw_wait_ready(fd);
    hw_write_entity(fd, REG_PAC_X, pac_x, pac_y,
pac.dir, frame, 1);
    hw_write_entity(fd, REG_G0_X, bx, by,
blinky.dir, frame, 1);
    hw_write_entity(fd, REG_G1_X, px, py,
pinky.dir, frame, 1);
    hw_write_entity(fd, REG_G2_X, ix, iy,
inky.dir, frame, 1);
    hw_write_entity(fd, REG_G3_X, cx, cy,
clyde.dir, frame, 1);
    hw_clear_ready(fd);
}
/* ================= GAME LOGIC =================
*/
void step_pacman(PacmanState *pac) {
    if (pac->dir == DIR_NONE) {
        return;
    }
    if (is_legal_move(pac->pos, pac->dir)) {
        pac->pos = vec_add(pac->pos,
dir_vec[(int)pac->dir]);
    }
}
Vec2 tiles_ahead_of_pacman(PacmanState pac, int
tiles, int emulate_bug) {
    Vec2 out;
    out = pac.pos;
    if (pac.dir == DIR_NONE) {
        return out;
    }
    if (pac.dir == DIR_UP && emulate_bug) {
        out.x -= tiles;
        out.y -= tiles;
        return out;
    }
    out.x += dir_vec[(int)pac.dir].x * tiles;
```

```
    out.y += dir_vec[(int)pac.dir].y * tiles;

    return out;

}

Vec2 compute_chase_target(Ghost ghost,
PacmanState pac, Ghost blinky) {

    Vec2 target;

    Vec2 two_ahead;

    Vec2 v;

    int dist2;

    switch (ghost.type) {

        case GHOST_BLINKY:

            return pac.pos;

        case GHOST_PINKY:

            return tiles_ahead_of_pacman(pac, 4,
1);

        case GHOST_INKY:

            two_ahead =
tiles_ahead_of_pacman(pac, 2, 1);

            v.x = two_ahead.x - blinky.pos.x;

            v.y = two_ahead.y - blinky.pos.y;

            target.x = two_ahead.x + v.x;

            target.y = two_ahead.y + v.y;

            return target;

        case GHOST_CLYDE:

            dist2 = squared_distance(ghost.pos,
pac.pos);

            if (dist2 <= 64) {

                return ghost.scatter_target;

            }

            return pac.pos;

    }

    return pac.pos;

}

Vec2 compute_target_tile(Ghost ghost, PacmanState
pac, Ghost blinky) {

    if (ghost.mode == MODE_SCATTER) {

        return ghost.scatter_target;

    }

    if (ghost.mode == MODE_FRIGHTENED) {

        return ghost.pos;

    }

    return compute_chase_target(ghost, pac,
blinky);

}

Direction choose_next_direction(Ghost ghost,
PacmanState pac, Ghost blinky) {

    Vec2 target;

    Vec2 next;

    Direction best_dir;

    Direction d;

    Direction rev;

    int best_dist;

    int dist;

    int i;

    if (ghost.mode == MODE_FRIGHTENED) {

        for (i = 0; i < 4; i++) {

            d = (Direction)i;

            if (is_legal_move(ghost.pos, d)) {

                return d;

            }

        }

        return DIR_NONE;

    }

    target = compute_target_tile(ghost, pac,
blinky);

    best_dir = DIR_NONE;

    best_dist = INT_MAX;

    for (i = 0; i < 4; i++) {

        d = (Direction)i;

        if (d == opposite_direction(ghost.dir)) {

            continue;

        }

        if (!is_legal_move(ghost.pos, d)) {

            continue;

        }

        next = vec_add(ghost.pos,
dir_vec[(int)d]);

        dist = squared_distance(next, target);

        if (dist < best_dist) {
```

```
            best_dist = dist;
            best_dir = d;
        }
    }
    if (best_dir == DIR_NONE) {
        rev = opposite_direction(ghost.dir);
        if (is_legal_move(ghost.pos, rev)) {
            return rev;
        }
    }
    return best_dir;
}
void step_ghost(Ghost *ghost, PacmanState pac,
Ghost blinky) {
    Direction next_dir;
    next_dir = choose_next_direction(*ghost, pac,
blinky);
    if (next_dir != DIR_NONE) {
        ghost->dir = next_dir;
        ghost->pos = vec_add(ghost->pos,
dir_vec[(int)next_dir]);
    }
}
/* ================= USB CONTROLLER
================= */
Direction decode_dpad_direction(unsigned char
*data, int len) {
    int x;
    int y;
    if (len <= DPAD_X_BYTE || len <= DPAD_Y_BYTE)
{
        return DIR_NONE;
    }
    x = data[DPAD_X_BYTE];
    y = data[DPAD_Y_BYTE];
    last_x_raw = x;
    last_y_raw = y;
    if (x == DPAD_LOW) {
        return DIR_LEFT;
    }
    if (x == DPAD_HIGH) {
        return DIR_RIGHT;
    }
    if (y == DPAD_LOW) {
        return DIR_UP;
    }
    if (y == DPAD_HIGH) {
        return DIR_DOWN;
    }
    return DIR_NONE;
}
Direction
read_joystick_direction(libusb_device_handle
*handle, Direction old_dir) {
    unsigned char data[64];
    int transferred;
    int ret;
    int i;
    Direction d;
    ret = libusb_interrupt_transfer(handle,
ENDPOINT_IN, data, 8, &transferred, 10);
    if (ret == 0 && transferred > 0) {
        last_packet_len = transferred;
        for (i = 0; i < transferred; i++) {
            last_packet[i] = data[i];
        }
        d = decode_dpad_direction(data,
transferred);
        last_dpad_dir = d;
        if (d != DIR_NONE) {
            return d;
        }
    }
    return old_dir;
}
int joystick_init(libusb_context **ctx,
libusb_device_handle **handle) {
    int ret;
    ret = libusb_init(ctx);
    if (ret < 0) {
```

```
        printf("libusb_init failed\n");

        return 0;

    }

    *handle =
libusb_open_device_with_vid_pid(*ctx, VENDOR_ID,
PRODUCT_ID);

    if (*handle == NULL) {

        printf("Could not open KIWITATA
controller\n");

        libusb_exit(*ctx);

        return 0;

    }

    if (libusb_kernel_driver_active(*handle,
INTERFACE_NUM) == 1) {

        libusb_detach_kernel_driver(*handle,
INTERFACE_NUM);

    }

    ret = libusb_claim_interface(*handle,
INTERFACE_NUM);

    if (ret != 0) {

        printf("Could not claim controller
interface: %s\n", libusb_error_name(ret));

        libusb_close(*handle);

        libusb_exit(*ctx);

        return 0;

    }

    return 1;

}

void joystick_close(libusb_context *ctx,
libusb_device_handle *handle) {

    libusb_release_interface(handle,
INTERFACE_NUM);

    libusb_attach_kernel_driver(handle,
INTERFACE_NUM);

    libusb_close(handle);

    libusb_exit(ctx);

}

/* ================= TERMINAL DEBUG
================= */

void terminal_start(void) {

    printf("\033[2J");

    printf("\033[H");

    printf("\033[?25l");

    fflush(stdout);

}

void terminal_stop(void) {

    printf("\033[?25h");

    printf("\033[0m");

    printf("\n");

    fflush(stdout);

}

char ghost_symbol_at(int x, int y, Ghost b, Ghost
p, Ghost i, Ghost c) {

    if (x == b.pos.x && y == b.pos.y) return
b.symbol;

    if (x == p.pos.x && y == p.pos.y) return
p.symbol;

    if (x == i.pos.x && y == i.pos.y) return
i.symbol;

    if (x == c.pos.x && y == c.pos.y) return
c.symbol;

    return '\0';

}

void print_grid(PacmanState pac, Ghost b, Ghost
p, Ghost i, Ghost c, int tick_count) {

    int x;

    int y;

    int k;

    char gs;

    printf("\033[H");

    for (y = 0; y < MAZE_H; y++) {

        for (x = 0; x < MAZE_W; x++) {

            gs = ghost_symbol_at(x, y, b, p, i,
c);

            if (x == pac.pos.x && y == pac.pos.y)
{

                printf("P");

            }

            else if (gs != '\0') {

                printf("%c", gs);

            }

            else {

                printf("%c", maze[y][x]);
```

```
            }
        }
        printf("\033[K\n");
    }
    printf("\033[K\n");
    printf("Tick: %d | Ghost speed: 1/%d Pac-Man
speed\033[K\n",
           tick_count,
           GHOST_SPEED_DIV);
    printf("Pac: (%d,%d) %s | Dpad: %s |
XRaw=0x%02x YRaw=0x%02x\033[K\n",
           pac.pos.x,
           pac.pos.y,
           dir_name(pac.dir),
           dir_name(last_dpad_dir),
           last_x_raw & 0xff,
           last_y_raw & 0xff);
    printf("Blinky: (%d,%d) %s | Pinky: (%d,%d)
%s\033[K\n",
           b.pos.x, b.pos.y, dir_name(b.dir),
           p.pos.x, p.pos.y, dir_name(p.dir));
    printf("Inky:   (%d,%d) %s | Clyde: (%d,%d)
%s\033[K\n",
           i.pos.x, i.pos.y, dir_name(i.dir),
           c.pos.x, c.pos.y, dir_name(c.dir));
    printf("Packet:");
    for (k = 0; k < last_packet_len; k++) {
        printf(" %02x", last_packet[k]);
    }
    printf("\033[K\n");
    printf("VGA output active through
/dev/vga_pacman ioctl. Ctrl+C to stop.\033[K\n");
    fflush(stdout);
}
/* ================ MAIN ================ */
int main(void) {
    libusb_context *usb_ctx = NULL;
    libusb_device_handle *joy_handle = NULL;
    int vga_fd;
    PacmanState pac;
    Ghost blinky;
    Ghost pinky;
    Ghost inky;
    Ghost clyde;
    int tick_count;
    signal(SIGINT, handle_sigint);
    vga_fd = open("/dev/vga_pacman", O_RDWR);
    if (vga_fd < 0) {
        printf("Could not open /dev/vga_pacman:
%s\n", strerror(errno));
        printf("Make sure you ran: insmod
vga_pacman.ko\n");
        return 1;
    }
    if (!joystick_init(&usb_ctx, &joy_handle)) {
        close(vga_fd);
        return 1;
    }
    printf("Initial VGA register readback:\n");
    printf("PAC_X    = 0x%04x\n",
hw_read_reg(vga_fd, REG_PAC_X));
    printf("PAC_Y    = 0x%04x\n",
hw_read_reg(vga_fd, REG_PAC_Y));
    printf("PAC_CTRL = 0x%04x\n",
hw_read_reg(vga_fd, REG_PAC_CTRL));
    printf("READY    = 0x%04x\n",
hw_read_reg(vga_fd, REG_READY));
    usleep(1000000);
    pac.pos.x = 9;
    pac.pos.y = 5;
    pac.dir = DIR_RIGHT;
    blinky.type = GHOST_BLINKY;
    blinky.pos.x = 9;
    blinky.pos.y = 1;
    blinky.dir = DIR_LEFT;
    blinky.mode = MODE_CHASE;
    blinky.scatter_target.x = 17;
    blinky.scatter_target.y = 0;
    blinky.name = "Blinky";
    blinky.symbol = 'B';
```

```
    blinky.hw_base = REG_G0_X;
    pinky.type = GHOST_PINKY;
    pinky.pos.x = 1;
    pinky.pos.y = 1;
    pinky.dir = DIR_RIGHT;
    pinky.mode = MODE_CHASE;
    pinky.scatter_target.x = 1;
    pinky.scatter_target.y = 0;
    pinky.name = "Pinky";
    pinky.symbol = 'K';
    pinky.hw_base = REG_G1_X;
    inky.type = GHOST_INKY;
    inky.pos.x = 17;
    inky.pos.y = 9;
    inky.dir = DIR_LEFT;
    inky.mode = MODE_CHASE;
    inky.scatter_target.x = 17;
    inky.scatter_target.y = 10;
    inky.name = "Inky";
    inky.symbol = 'I';
    inky.hw_base = REG_G2_X;
    clyde.type = GHOST_CLYDE;
    clyde.pos.x = 1;
    clyde.pos.y = 9;
    clyde.dir = DIR_RIGHT;
    clyde.mode = MODE_CHASE;
    clyde.scatter_target.x = 1;
    clyde.scatter_target.y = 10;
    clyde.name = "Clyde";
    clyde.symbol = 'C';
    clyde.hw_base = REG_G3_X;
    tick_count = 0;
    terminal_start();
    while (running) {
        tick_count++;
        pac.dir =
read_joystick_direction(joy_handle, pac.dir);
        step_pacman(&pac);
        if (tick_count % GHOST_SPEED_DIV == 0) {
            step_ghost(&blinky, pac, blinky);
            step_ghost(&pinky, pac, blinky);
            step_ghost(&inky, pac, blinky);
            step_ghost(&clyde, pac, blinky);
        }
        draw_to_vga(vga_fd, pac, blinky, pinky,
inky, clyde, tick_count);
        print_grid(pac, blinky, pinky, inky,
clyde, tick_count);
        usleep(FRAME_SLEEP_US);
    }
    terminal_stop();
    joystick_close(usb_ctx, joy_handle);
    close(vga_fd);
    return 0;
}
```

MAKEFILE

```
obj-m += vga_pacman.o pacman_audio.o
KDIR := /usr/src/linux-headers-4.19.0
PWD := $(shell pwd)
CC := cc
all:
    make -C $(KDIR) SUBDIRS=$(PWD) modules
    $(CC) pacman_hw_test.c -o pacman_hw_test
    $(CC) pacman_hw_game.c audio_driver.c -o
pacman_hw_game -lusb-1.0 -lpthread
    $(CC) Ghosts_and_Joystick.c -o
Ghosts_and_Joystick -lusb-1.0
    $(CC) fill_screen_tiles.c -o
fill_screen_tiles
    $(CC) draw_pellet_wall_maze.c -o
draw_pellet_wall_maze
    $(CC) draw_numbered_wall_maze.c -o
draw_numbered_wall_maze
    $(CC) draw_tile_index_rows.c -o
draw_tile_index_rows
clean:
    make -C $(KDIR) SUBDIRS=$(PWD) clean
    rm -f pacman_hw_test
    rm -f pacman_hw_game
```

```
    rm -f Ghosts_and_Joystick
    rm -f fill_screen_tiles
    rm -f draw_pellet_wall_maze
    rm -f draw_numbered_wall_maze
    rm -f draw_tile_index_rows
```

PACMAN_AUDIO.C

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/miscdevice.h>
#include <linux/uaccess.h>
#include <linux/init.h>
#include <linux/interrupt.h>
#include <linux/wait.h>
#include <linux/of.h>
#include <linux/of_irq.h>
#include "pacman_audio.h"
#define DRIVER_NAME "pacman_audio"
static int audio_irq = -1;
static DECLARE_WAIT_QUEUE_HEAD(audio_wq);
static int audio_interrupt_triggered = 0;
static irqreturn_t audio_interrupt_handler(int
irq, void *dev_id)
{
    // Disable interrupt dynamically to prevent
storming the CPU
    disable_irq_nosync(irq);

    audio_interrupt_triggered = 1;
    wake_up_interruptible(&audio_wq);

    return IRQ_HANDLED;
}
static long pacman_audio_ioctl(struct file *file,
unsigned int cmd, unsigned long arg)
{
    switch (cmd) {
        case PACMAN_AUDIO_WAIT_INTERRUPT:
            // Sleep until interrupt fires
            wait_event_interruptible(audio_wq,
audio_interrupt_triggered != 0);

            // Clear flag
            audio_interrupt_triggered = 0;

            // Re-enable interrupt
            if (audio_irq >= 0) {
                enable_irq(audio_irq);
            }
            return 0;
        default:
            return -ENOTTY;
    }
}
static int pacman_audio_open(struct inode *inode,
struct file *file)
{
    return 0;
}
static int pacman_audio_release(struct inode
*inode, struct file *file)
{
    return 0;
}
static const struct file_operations
pacman_audio_fops = {
    .owner = THIS_MODULE,
    .open = pacman_audio_open,
    .release = pacman_audio_release,
    .unlocked_ioctl = pacman_audio_ioctl,
};
static struct miscdevice pacman_audio_miscdev = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = "pacman_audio",
    .fops = &pacman_audio_fops,
    .mode = 0666,
};
static int __init pacman_audio_init(void)
```

```
{

    int ret;

    struct device_node *dn;

    printk(KERN_INFO DRIVER_NAME ": loading\n");

    ret = misc_register(&pacman_audio_miscdev);

    if (ret != 0) {

        printk(KERN_ERR DRIVER_NAME ":
misc_register failed\n");

        return ret;

    }

    // Dynamic device tree lookup for the audio
peripheral node compatible with "csee4840,pacman-
audio-1.0"

    dn = of_find_compatible_node(NULL, NULL,
"csee4840,pacman-audio-1.0");

    if (dn) {

        audio_irq = irq_of_parse_and_map(dn, 0);

        if (audio_irq >= 0) {

            ret = request_irq(audio_irq,
audio_interrupt_handler, 0, DRIVER_NAME, NULL);

            if (ret == 0) {

                printk(KERN_INFO DRIVER_NAME ":
registered audio interrupt %d successfully\n",
audio_irq);

            } else {

                printk(KERN_ERR DRIVER_NAME ":
failed to request audio interrupt %d (error
%d)\n", audio_irq, ret);

                audio_irq = -1;

            }

        } else {

            printk(KERN_ERR DRIVER_NAME ": failed
to map audio interrupt from device tree\n");

        }

        of_node_put(dn);

    } else {

        printk(KERN_WARNING DRIVER_NAME ": could
not find compatible 'csee4840,pacman-audio-1.0'
audio node in device tree\n");

    }

    printk(KERN_INFO DRIVER_NAME ": loaded as
/dev/pacman_audio\n");

    return 0;

}

static void __exit pacman_audio_exit(void)

{

    printk(KERN_INFO DRIVER_NAME ":
unloading\n");

    if (audio_irq >= 0) {

        free_irq(audio_irq, NULL);

        printk(KERN_INFO DRIVER_NAME ": freed
audio interrupt %d\n", audio_irq);

    }

    misc_deregister(&pacman_audio_miscdev);

}

module_init(pacman_audio_init);

module_exit(pacman_audio_exit);

MODULE_LICENSE("GPL");

MODULE_AUTHOR("Pac-Man Lab");

MODULE_DESCRIPTION("Pac-Man Audio Interrupt
Driver");
```

PACMAN_AUDIO.H

```
#ifndef PACMAN_AUDIO_H

#define PACMAN_AUDIO_H

#include <linux/ioctl.h>

#define PACMAN_AUDIO_MAGIC 'a'

/* IOCTL command to wait for an audio FIFO
interrupt */

#define
PACMAN_AUDIO_WAIT_INTERRUPT  _IO(PACMAN_AUDIO_MAG
IC, 1)

#endif /* PACMAN_AUDIO_H */
```

PACMAN_HW_GAME.C

```
/* SPDX-License-Identifier: MIT

 * pacman_hw_game.c

 *

 * Smooth-render Pac-Man movement/game loop for
DE1-SoC.

 */

#include <errno.h>

#include <fcntl.h>

#include <limits.h>

#include <signal.h>
```

```
#include <stdint.h>

#include <stdio.h>

#include <string.h>

#include <sys/ioctl.h>

#include <termios.h>

#include <unistd.h>

#include <libusb-1.0/libusb.h>

#include "vga_pacman_ioctl.h"

#include "audio_driver.h"

/* ================= AUDIO GLOBALS
================= */

static int audio_enabled = 0;

static int siren_handle = -1;

static int waka_toggle = 0;

static sound_t *snd_start = NULL;

static sound_t *snd_siren = NULL;

static sound_t *snd_waka0 = NULL;

static sound_t *snd_waka1 = NULL;

static sound_t *snd_death = NULL;

static sound_t *snd_eat_ghost = NULL;

/* ================= USB CONTROLLER
================= */

#define USB_VENDOR_ID       0x0079

#define USB_PRODUCT_ID      0x0011

#define USB_INTERFACE_NUM   0

#define USB_ENDPOINT_IN     0x81

#define DPAD_X_BYTE         3

#define DPAD_Y_BYTE         4

#define DPAD_LOW            0x00

#define DPAD_HIGH           0xff

#define USB_POLL_TIMEOUT_MS 1

#define USB_POLLS_PER_LOOP  3

/* ================= GAME TIMING
================= */

#define LOOP_DELAY_US               20000

#define PACMAN_MOVE_INTERVAL_US     100000

#define GHOST_MOVE_INTERVAL_US      200000

#define STATUS_PRINT_INTERVAL_US    100000

#define POWER_DURATION_US           10000000

#define
POWER_DURATION_TICKS        ((POWER_DURATION_US +
LOOP_DELAY_US - 1) / LOOP_DELAY_US)

/* ================= TILE / SPRITE GEOMETRY
================= */

#define TILE_PIX            16

#define SPRITE_PIX          32

#define SPRITE_TILE_OFFSET  ((SPRITE_PIX -
TILE_PIX) / 2)

/* ================= TUNNEL ================= */

#define TUNNEL_Y            14

#define TUNNEL_LEFT_X       0

#define TUNNEL_RIGHT_X      38

/* ================= GHOST SPAWN
================= */

#define SPAWN_X_MIN         15

#define SPAWN_X_MAX         24

#define SPAWN_Y_MIN         16

#define SPAWN_Y_MAX         17

#define SPAWN_EXIT_X        19

#define SPAWN_EXIT_Y        15

/* ================= START POSITIONS
================= */

#define PAC_START_X         1

#define PAC_START_Y         23

#define BLINKY_HOME_X       15

#define BLINKY_HOME_Y       16

#define PINKY_HOME_X        17

#define PINKY_HOME_Y        16

#define INKY_HOME_X         19

#define INKY_HOME_Y         16

#define CLYDE_HOME_X        21

#define CLYDE_HOME_Y        16

/* ================= GHOST TARGET CORNERS
================= */

#define TARGET_TOP_LEFT_X       1

#define TARGET_TOP_LEFT_Y       5

#define TARGET_TOP_RIGHT_X      37

#define TARGET_TOP_RIGHT_Y      5

#define TARGET_BOTTOM_LEFT_X    1
```

```
#define TARGET_BOTTOM_LEFT_Y    23
#define TARGET_BOTTOM_RIGHT_X   37
#define TARGET_BOTTOM_RIGHT_Y   23
#define CLYDE_CLOSE_DIST_SQ     64

/* ================= TILE IDS =================
*/
#define TILE_EMPTY          0x00
#define TILE_DOT            0x05
#define TILE_POWER_DOT      0x3A
#define TILE_VERT_LEFT          0x30
#define TILE_VERT_RIGHT         0x36
#define TILE_HORIZ_TOP          0x49
#define TILE_HORIZ_BOTTOM       0x24
#define TILE_CURVE_RIGHT_TOP    0x4A
#define TILE_CURVE_LEFT_TOP     0x4B
#define TILE_CURVE_RIGHT_BOTTOM 0x27
#define TILE_CURVE_LEFT_BOTTOM  0x29

/* ================= ALPHANUMERIC TILE IDS
================= */
#define TILE_A              0x09
#define TILE_B              0x0A
#define TILE_C              0x0B
#define TILE_D              0x0C
#define TILE_E              0x0D
#define TILE_F              0x0E
#define TILE_G              0x0F
#define TILE_H              0x10
#define TILE_I              0x11
#define TILE_J              0x12
#define TILE_K              0x13
#define TILE_L              0x14
#define TILE_M              0x15
#define TILE_N              0x16
#define TILE_O              0x18
#define TILE_P              0x19
#define TILE_Q              0x1A
#define TILE_R              0x1B
#define TILE_S              0x1C
#define TILE_T              0x1D
#define TILE_U              0x1E
#define TILE_V              0x1F
#define TILE_W              0x20
#define TILE_X              0x21
#define TILE_Y              0x22
#define TILE_Z              0x23
#define TILE_NUM_0          0x4E
#define TILE_NUM_1          0x32
#define TILE_NUM_2          0x48
#define TILE_NUM_3          0x40
#define TILE_NUM_4          0x2F
#define TILE_NUM_5          0x2E
#define TILE_NUM_6          0x38
#define TILE_NUM_7          0x37
#define TILE_NUM_8          0x2D
#define TILE_NUM_9          0x17

/* ================= SCORE / HIGH SCORE DISPLAY
================= */
#define SCORE_LABEL_ROW         1
#define SCORE_LABEL_COL         4
#define SCORE_DIGIT_ROW         2
#define SCORE_DIGIT_COL         4
#define SCORE_DIGIT_COUNT       6
#define HIGH_LABEL_ROW          1
#define HIGH_LABEL_COL          24
#define HIGH_DIGIT_ROW          2
#define HIGH_DIGIT_COL          26
#define HIGH_SCORE_FIXED        123456

/* ================= LIFE / WIN DISPLAY
================= */
#define LIFE_ROW                27
#define LIFE_COL                2
#define WIN_ROW                 27
#define WIN_COL                 30

/* ================= SCORING ================= */
#define SCORE_DOT           10
#define SCORE_POWER         50
```

```
#define SCORE_GHOST         500
#define PELLET_NONE         0
#define PELLET_DOT          1
#define PELLET_POWER        2
/* ================= TYPES ================= */
typedef enum {
    DIR_UP = 0,
    DIR_LEFT = 1,
    DIR_DOWN = 2,
    DIR_RIGHT = 3,
    DIR_NONE = 4
} Direction;
typedef enum {
    GHOST_BLINKY,
    GHOST_PINKY,
    GHOST_INKY,
    GHOST_CLYDE
} GhostType;
typedef enum {
    GHOST_NORMAL,
    GHOST_FRIGHTENED,
    GHOST_EYES
} GhostMode;
typedef struct {
    int x;
    int y;
} Vec2;
typedef struct {
    Vec2 pos;
    Vec2 prev_pos;
    int move_timer_us;
    int move_duration_us;
    Direction dir;
    Direction next_dir;
} PacmanState;
typedef struct {
    GhostType type;
    Vec2 pos;
    Vec2 prev_pos;
    Vec2 home_pos;
    Vec2 target;
    int move_timer_us;
    int move_duration_us;
    Direction dir;
    GhostMode mode;
    int eaten_count;
    int leaving_spawn;
    const char *name;
} Ghost;
/* ================= 40x30 TILE MAP
================= */
#define LOGIC_H VGA_PACMAN_VISIBLE_TILE_ROWS
#define LOGIC_W VGA_PACMAN_VISIBLE_TILE_COLS
static const char *tile_map[LOGIC_H] = {
    "....................................",
    "....................................",
    "....................................",
    "....................................",
    "53333333333333333333333333333333333336",
    "1A99999999999999999999999999999999A2",
    "19000000000000000000000000000000000092",
    "19053333336905333333333336095333336092",
    "19074444448907444444444448097444448092",
    "19000000000909999999999999090000000092",
    "19999999999900000000000000099999999992",
    "19056901092905333333333336091902095 6092",
    "19012901092907444444444444809190209 12092",
    "89078901092909999999999999909190209 78097",
    "09999901092900000000000000009190209 00090",
    "09000901092905333330033333609190209 99990",
    "69056901092901000000000000209190209 56095",
    "19012901092901000000000000209190209 12092",
    "19012999999074444444444448099999991 2092",
    "19012000000900000000000000090000001 2092",
    "19017333336909999999999999095333338 2092",
```

```
    "19010000002905333333333333609100000020 92",
    "19074444448907444444444444809744444480 92",
    "19999999999999999999999999999999999999 992",
    "1A000000000000000000000000000000000000A2",
    "74444444444444444444444444444444444448",
    ".....................................",
    ".....................................",
    ".....................................",
    "....................................."
};
static const Vec2 dir_vec[4] = {
    { 0, -1 },
    { -1, 0 },
    { 0, 1 },
    { 1, 0 }
};
/* ================ GLOBALS ================ */
static int running = 1;
static int score = 0;
static int last_drawn_score = -1;
static int lives = 3;
static int last_drawn_lives = -1;
static int pellets_eaten = 0;
static int normal_pellets_eaten = 0;
static int power_pellets_eaten = 0;
static int total_pellets = 0;
static int ghosts_eaten = 0;
static int power_until_tick = 0;
static int win_drawn = 0;
static int game_is_done = 0;
static unsigned char
pellet_live[LOGIC_H][LOGIC_W];
static struct termios saved_termios;
static int terminal_configured = 0;
static int controller_mode = 0;
static libusb_context *usb_ctx = NULL;
static libusb_device_handle *joy_handle = NULL;
static Direction last_input_dir = DIR_NONE;
static char last_input_name[32] = "none";
static int last_packet_len = 0;
static unsigned char last_packet[8];
/* ================ BASIC HELPERS
================ */
static void handle_sigint(int sig)
{
    (void)sig;
    running = 0;
}
static const char *dir_name(Direction d)
{
    switch (d) {
    case DIR_UP:
        return "UP";
    case DIR_LEFT:
        return "LEFT";
    case DIR_DOWN:
        return "DOWN";
    case DIR_RIGHT:
        return "RIGHT";
    default:
        return "NONE";
    }
}
static const char *ghost_name(GhostType t)
{
    switch (t) {
    case GHOST_BLINKY:
        return "Blinky";
    case GHOST_PINKY:
        return "Pinky";
    case GHOST_INKY:
        return "Inky";
    case GHOST_CLYDE:
        return "Clyde";
    default:
        return "Ghost";
```

```
    }
}
static const char *ghost_mode_name(GhostMode
mode)
{
    if (mode == GHOST_FRIGHTENED) {
        return "FLEE";
    }
    if (mode == GHOST_EYES) {
        return "EYES";
    }
    return "CHASE";
}
static void remember_input(Direction d, const
char *name)
{
    last_input_dir = d;
    snprintf(last_input_name,
sizeof(last_input_name), "%s", name);
}
static Direction opposite_direction(Direction d)
{
    if (d == DIR_UP) {
        return DIR_DOWN;
    }
    if (d == DIR_DOWN) {
        return DIR_UP;
    }
    if (d == DIR_LEFT) {
        return DIR_RIGHT;
    }
    if (d == DIR_RIGHT) {
        return DIR_LEFT;
    }
    return DIR_NONE;
}
static Vec2 vec_add(Vec2 a, Vec2 b)
{
    Vec2 out;
    out.x = a.x + b.x;
    out.y = a.y + b.y;
    return out;
}
static Vec2 vec_make(int x, int y)
{
    Vec2 out;
    out.x = x;
    out.y = y;
    return out;
}
static int squared_distance(Vec2 a, Vec2 b)
{
    int dx;
    int dy;
    dx = a.x - b.x;
    dy = a.y - b.y;
    return dx * dx + dy * dy;
}
static int sprites_overlap(Vec2 a, Vec2 b)
{
    int dx;
    int dy;
    dx = a.x - b.x;
    if (dx < 0) {
        dx = -dx;
    }
    dy = a.y - b.y;
    if (dy < 0) {
        dy = -dy;
    }
    if (dx < 2 && dy < 2) {
        return 1;
    }
    return 0;
}
/* ================= POWER TIMER
================= */
```

```
static void reset_power_timer_from_now(int
current_tick)

{

    power_until_tick = current_tick +
POWER_DURATION_TICKS;

}

/* ================= MAP VALIDATION
================= */

static int validate_tile_map(void)

{

    int row;

    int len;

    for (row = 0; row < LOGIC_H; row++) {

        len = strlen(tile_map[row]);

        if (len != LOGIC_W) {

            printf("ERROR: tile_map row %d has
length %d, expected %d\n", row, len, LOGIC_W);

            return -1;

        }

    }

    return 0;

}

/* ================= SMOOTH RENDER HELPERS
================= */

static int abs_int(int value)

{

    if (value < 0) {

        return -value;

    }

    return value;

}

static int logical_x_to_pixel(int x)

{

    int pixel;

    pixel = x * TILE_PIX - SPRITE_TILE_OFFSET;

    if (pixel < 0) {

        return 0;

    }

    if (pixel > VGA_PACMAN_SCREEN_W - SPRITE_PIX)
{

        return VGA_PACMAN_SCREEN_W - SPRITE_PIX;

    }

    return pixel;

}

static int logical_y_to_pixel(int y)

{

    int pixel;

    pixel = y * TILE_PIX - SPRITE_TILE_OFFSET;

    if (pixel < 0) {

        return 0;

    }

    if (pixel > VGA_PACMAN_SCREEN_H - SPRITE_PIX)
{

        return VGA_PACMAN_SCREEN_H - SPRITE_PIX;

    }

    return pixel;

}

static int interpolate_pixel(int start_pixel, int
end_pixel, int timer_us, int duration_us)

{

    int delta;

    int pixel;

    if (duration_us <= 0) {

        return end_pixel;

    }

    if (timer_us >= duration_us) {

        return end_pixel;

    }

    if (timer_us < 0) {

        return start_pixel;

    }

    delta = end_pixel - start_pixel;

    pixel = start_pixel + (delta * timer_us) /
duration_us;

    return pixel;

}

static int render_x_pixel(Vec2 prev_pos, Vec2
pos, int timer_us, int duration_us)

{
```

```
    int start_pixel;

    int end_pixel;

    if (abs_int(pos.x - prev_pos.x) > 4) {

        return logical_x_to_pixel(pos.x);

    }

    start_pixel = logical_x_to_pixel(prev_pos.x);

    end_pixel = logical_x_to_pixel(pos.x);

    return interpolate_pixel(start_pixel,
end_pixel, timer_us, duration_us);

}

static int render_y_pixel(Vec2 prev_pos, Vec2
pos, int timer_us, int duration_us)

{

    int start_pixel;

    int end_pixel;

    if (abs_int(pos.y - prev_pos.y) > 4) {

        return logical_y_to_pixel(pos.y);

    }

    start_pixel = logical_y_to_pixel(prev_pos.y);

    end_pixel = logical_y_to_pixel(pos.y);

    return interpolate_pixel(start_pixel,
end_pixel, timer_us, duration_us);

}

static void advance_pacman_animation(PacmanState
*pac)

{

    pac->move_timer_us += LOOP_DELAY_US;

    if (pac->move_timer_us > pac-
>move_duration_us) {

        pac->move_timer_us = pac-
>move_duration_us;

    }

}

static void advance_ghost_animation(Ghost *ghost)

{

    ghost->move_timer_us += LOOP_DELAY_US;

    if (ghost->move_timer_us > ghost-
>move_duration_us) {

        ghost->move_timer_us = ghost-
>move_duration_us;

    }

}

/* ================= TUNNEL HELPERS
================= */

static int sprite_is_on_tunnel_row(Vec2 p)

{

    if (p.y == TUNNEL_Y) {

        return 1;

    }

    return 0;

}

static Vec2 next_pos_with_tunnel_wrap(Vec2 pos,
Direction dir)

{

    Vec2 next;

    next = pos;

    if (dir == DIR_NONE) {

        return next;

    }

    if (sprite_is_on_tunnel_row(pos)) {

        if (dir == DIR_LEFT && pos.x <=
TUNNEL_LEFT_X) {

            next.x = TUNNEL_RIGHT_X;

            next.y = pos.y;

            return next;

        }

        if (dir == DIR_RIGHT && pos.x >=
TUNNEL_RIGHT_X) {

            next.x = TUNNEL_LEFT_X;

            next.y = pos.y;

            return next;

        }

    }

    next = vec_add(pos, dir_vec[(int)dir]);

    return next;

}

/* ================= COLLISION / MOVEMENT
LEGALITY ================= */

static int tile_in_bounds(Vec2 p)

{

    if (p.x < 0 || p.x >= LOGIC_W) {
```

```
        return 0;
    }
    if (p.y < 0 || p.y >= LOGIC_H) {
        return 0;
    }
    return 1;
}
static int tile_is_open(Vec2 p)
{
    char c;
    if (!tile_in_bounds(p)) {
        return 0;
    }
    c = tile_map[p.y][p.x];
    if (c == '0' || c == '9' || c == 'A') {
        return 1;
    }
    return 0;
}
static int tile_is_inside_spawn(Vec2 p)
{
    if (p.x < SPAWN_X_MIN || p.x > SPAWN_X_MAX) {
        return 0;
    }
    if (p.y < SPAWN_Y_MIN || p.y > SPAWN_Y_MAX) {
        return 0;
    }
    return 1;
}
static int sprite_overlaps_spawn(Vec2 p)
{
    Vec2 p00;
    Vec2 p10;
    Vec2 p01;
    Vec2 p11;
    p00.x = p.x;
    p00.y = p.y;
    p10.x = p.x + 1;
    p10.y = p.y;
    p01.x = p.x;
    p01.y = p.y + 1;
    p11.x = p.x + 1;
    p11.y = p.y + 1;
    if (tile_is_inside_spawn(p00)) {
        return 1;
    }
    if (tile_is_inside_spawn(p10)) {
        return 1;
    }
    if (tile_is_inside_spawn(p01)) {
        return 1;
    }
    if (tile_is_inside_spawn(p11)) {
        return 1;
    }
    return 0;
}
static int sprite_pos_is_open(Vec2 p)
{
    Vec2 p00;
    Vec2 p10;
    Vec2 p01;
    Vec2 p11;
    p00.x = p.x;
    p00.y = p.y;
    p10.x = p.x + 1;
    p10.y = p.y;
    p01.x = p.x;
    p01.y = p.y + 1;
    p11.x = p.x + 1;
    p11.y = p.y + 1;
    if (!tile_is_open(p00)) {
        return 0;
    }
```

```
    if (!tile_is_open(p10)) {
        return 0;
    }
    if (!tile_is_open(p01)) {
        return 0;
    }
    if (!tile_is_open(p11)) {
        return 0;
    }
    return 1;
}
static int is_legal_pacman_move(Vec2 pos,
Direction dir)
{
    Vec2 next;
    if (dir == DIR_NONE) {
        return 0;
    }
    next = next_pos_with_tunnel_wrap(pos, dir);
    if (!sprite_pos_is_open(next)) {
        return 0;
    }
    if (sprite_overlaps_spawn(next)) {
        return 0;
    }
    return 1;
}
static int is_legal_ghost_move(Ghost ghost,
Direction dir)
{
    Vec2 next;
    if (dir == DIR_NONE) {
        return 0;
    }
    next = next_pos_with_tunnel_wrap(ghost.pos,
dir);
    if (!sprite_pos_is_open(next)) {
        return 0;
    }
    if (ghost.leaving_spawn || ghost.mode ==
GHOST_EYES) {
        return 1;
    }
    if (sprite_overlaps_spawn(next)) {
        return 0;
    }
    return 1;
}
static Vec2 choose_open_sprite_pos(Vec2
preferred)
{
    Vec2 best;
    Vec2 candidate;
    int best_dist;
    int dist;
    int row;
    int col;
    if (sprite_pos_is_open(preferred)) {
        return preferred;
    }
    best = preferred;
    best_dist = INT_MAX;
    for (row = 0; row < LOGIC_H - 1; row++) {
        for (col = 0; col < LOGIC_W - 1; col++) {
            candidate.x = col;
            candidate.y = row;
            if (!sprite_pos_is_open(candidate)) {
                continue;
            }
            dist = squared_distance(candidate,
preferred);
            if (dist < best_dist) {
                best_dist = dist;
                best = candidate;
            }
        }
    }
    return best;
```

```
}

/* ================= INPUT ================= */

static void terminal_start(void)

{

    struct termios raw;

    int flags;

    if (tcgetattr(STDIN_FILENO, &saved_termios)
== 0) {

        raw = saved_termios;

        raw.c_lflag &= ~(ICANON | ECHO);

        raw.c_cc[VMIN] = 0;

        raw.c_cc[VTIME] = 0;

        if (tcsetattr(STDIN_FILENO, TCSANOW,
&raw) == 0) {

            terminal_configured = 1;

        }

    }

    flags = fcntl(STDIN_FILENO, F_GETFL, 0);

    if (flags >= 0) {

        fcntl(STDIN_FILENO, F_SETFL, flags |
O_NONBLOCK);

    }

    printf("\033[2J\033[H\033[?25l");

    fflush(stdout);

}

static void terminal_stop(void)

{

    if (terminal_configured) {

        tcsetattr(STDIN_FILENO, TCSANOW,
&saved_termios);

    }

    printf("\033[?25h\033[0m\n");

    fflush(stdout);

}

static Direction
read_keyboard_direction(Direction old_dir)

{

    unsigned char data[32];

    ssize_t n;

    int i;

    Direction newest;

    newest = old_dir;

    n = read(STDIN_FILENO, data, sizeof(data));

    if (n < 0) {

        if (errno != EAGAIN && errno != EINTR) {

            perror("read keyboard");

        }

        return old_dir;

    }

    for (i = 0; i < n; i++) {

        if (data[i] == '\033' && i + 2 < n &&
data[i + 1] == '[') {

            if (data[i + 2] == 'A') {

                newest = DIR_UP;

                remember_input(newest, "arrow-
up");

            }

            else if (data[i + 2] == 'B') {

                newest = DIR_DOWN;

                remember_input(newest, "arrow-
down");

            }

            else if (data[i + 2] == 'C') {

                newest = DIR_RIGHT;

                remember_input(newest, "arrow-
right");

            }

            else if (data[i + 2] == 'D') {

                newest = DIR_LEFT;

                remember_input(newest, "arrow-
left");

            }

            i += 2;

            continue;

        }

        if (data[i] == 'w' || data[i] == 'W') {

            newest = DIR_UP;

            remember_input(newest, "w");

        }
```

```
        else if (data[i] == 'a' || data[i] ==
'A') {
            newest = DIR_LEFT;
            remember_input(newest, "a");
        }
        else if (data[i] == 's' || data[i] ==
'S') {
            newest = DIR_DOWN;
            remember_input(newest, "s");
        }
        else if (data[i] == 'd' || data[i] ==
'D') {
            newest = DIR_RIGHT;
            remember_input(newest, "d");
        }
        else if (data[i] == 'q' || data[i] ==
'Q') {
            remember_input(DIR_NONE, "q");
            running = 0;
        }
    }
    return newest;
}
static Direction
decode_joystick_direction(unsigned char *data,
int len)
{
    int x;
    int y;
    if (len <= DPAD_X_BYTE || len <= DPAD_Y_BYTE)
{
        return DIR_NONE;
    }
    x = data[DPAD_X_BYTE];
    y = data[DPAD_Y_BYTE];
    if (x == DPAD_LOW) {
        remember_input(DIR_LEFT, "dpad-left");
        return DIR_LEFT;
    }
    if (x == DPAD_HIGH) {
        remember_input(DIR_RIGHT, "dpad-right");
        return DIR_RIGHT;
    }
    if (y == DPAD_LOW) {
        remember_input(DIR_UP, "dpad-up");
        return DIR_UP;
    }
    if (y == DPAD_HIGH) {
        remember_input(DIR_DOWN, "dpad-down");
        return DIR_DOWN;
    }
    return DIR_NONE;
}
static Direction
read_joystick_direction(libusb_device_handle
*handle, Direction old_dir)
{
    unsigned char data[8];
    int transferred;
    int ret;
    int i;
    int poll;
    Direction newest_dir;
    Direction decoded_dir;
    newest_dir = old_dir;
    for (poll = 0; poll < USB_POLLS_PER_LOOP;
poll++) {
        transferred = 0;
        ret = libusb_interrupt_transfer(
            handle,
            USB_ENDPOINT_IN,
            data,
            8,
            &transferred,
            USB_POLL_TIMEOUT_MS
        );
        if (ret == 0 && transferred > 0) {
            last_packet_len = transferred;
```

```
            for (i = 0; i < transferred && i < 8;
i++) {
                last_packet[i] = data[i];
            }
            decoded_dir =
decode_joystick_direction(data, transferred);
            if (decoded_dir != DIR_NONE) {
                newest_dir = decoded_dir;
            }
        }
        else if (ret == LIBUSB_ERROR_TIMEOUT) {
            continue;
        }
        else {
            continue;
        }
    }
    return newest_dir;
}
static int joystick_init(void)
{
    int ret;
    ret = libusb_init(&usb_ctx);
    if (ret < 0) {
        return 0;
    }
    joy_handle =
libusb_open_device_with_vid_pid(usb_ctx,
USB_VENDOR_ID, USB_PRODUCT_ID);
    if (joy_handle == NULL) {
        libusb_exit(usb_ctx);
        usb_ctx = NULL;
        return 0;
    }
    if (libusb_kernel_driver_active(joy_handle,
USB_INTERFACE_NUM) == 1) {
        libusb_detach_kernel_driver(joy_handle,
USB_INTERFACE_NUM);
    }
    ret = libusb_claim_interface(joy_handle,
USB_INTERFACE_NUM);
    if (ret != 0) {
        libusb_close(joy_handle);
        libusb_exit(usb_ctx);
        joy_handle = NULL;
        usb_ctx = NULL;
        return 0;
    }
    return 1;
}
static void joystick_close(void)
{
    if (joy_handle != NULL) {
        libusb_release_interface(joy_handle,
USB_INTERFACE_NUM);
        libusb_attach_kernel_driver(joy_handle,
USB_INTERFACE_NUM);
        libusb_close(joy_handle);
        joy_handle = NULL;
    }
    if (usb_ctx != NULL) {
        libusb_exit(usb_ctx);
        usb_ctx = NULL;
    }
}
static Direction read_game_input(Direction
old_dir)
{
    Direction d;
    d = old_dir;
    if (controller_mode && joy_handle != NULL) {
        d = read_joystick_direction(joy_handle,
d);
    }
    d = read_keyboard_direction(d);
    return d;
}
/* ================= HARDWARE =================
*/
```

```
static uint8_t hw_dir_from_sw(Direction d)

{

    if (d == DIR_UP) {

        return VGA_DIR_UP;

    }

    if (d == DIR_RIGHT) {

        return VGA_DIR_RIGHT;

    }

    if (d == DIR_DOWN) {

        return VGA_DIR_DOWN;

    }

    if (d == DIR_LEFT) {

        return VGA_DIR_LEFT;

    }

    return VGA_DIR_RIGHT;

}

static int set_pacman_hw(int fd, const
PacmanState *pac, int frame)

{

    struct vga_pacman_sprite s;

    memset(&s, 0, sizeof(s));

    s.x = render_x_pixel(pac->prev_pos, pac->pos,
pac->move_timer_us, pac->move_duration_us);

    s.y = render_y_pixel(pac->prev_pos, pac->pos,
pac->move_timer_us, pac->move_duration_us);

    s.dir = hw_dir_from_sw(pac->dir);

    s.frame = frame & 0x3;

    s.visible = 1;

    return ioctl(fd, VGA_PACMAN_SET_PACMAN, &s);

}

static int set_ghost_hw(int fd, int index, const
Ghost *ghost, int frame)

{

    struct vga_pacman_ghost g;

    memset(&g, 0, sizeof(g));

    g.index = index;

    g.sprite.x = render_x_pixel(ghost->prev_pos,
ghost->pos, ghost->move_timer_us, ghost-
>move_duration_us);

    g.sprite.y = render_y_pixel(ghost->prev_pos,
ghost->pos, ghost->move_timer_us, ghost-
>move_duration_us);

    g.sprite.dir = hw_dir_from_sw(ghost->dir);

    g.sprite.frame = frame & 0x1;

    g.sprite.visible = 1;

    if (ghost->mode == GHOST_FRIGHTENED) {

        g.sprite.state = 1;

    } else if (ghost->mode == GHOST_EYES) {

        g.sprite.state = 2;

    } else {

        g.sprite.state = 0;

    }

    return ioctl(fd, VGA_PACMAN_SET_GHOST, &g);

}

static int write_tile_hw(int fd, int row, int
col, uint8_t tile_id)

{

    struct vga_pacman_tile t;

    if (row < 0 || row >=
VGA_PACMAN_VISIBLE_TILE_ROWS) {

        return -1;

    }

    if (col < 0 || col >=
VGA_PACMAN_VISIBLE_TILE_COLS) {

        return -1;

    }

    memset(&t, 0, sizeof(t));

    t.row = row;

    t.col = col;

    t.tile_id = tile_id;

    return ioctl(fd, VGA_PACMAN_SET_TILE, &t);

}

/* ================= DISPLAY HELPERS
================= */

static uint8_t digit_tile_from_int(int digit)

{

    if (digit == 0) {

        return TILE_NUM_0;

    }
```

```
    if (digit == 1) {
        return TILE_NUM_1;
    }
    if (digit == 2) {
        return TILE_NUM_2;
    }
    if (digit == 3) {
        return TILE_NUM_3;
    }
    if (digit == 4) {
        return TILE_NUM_4;
    }
    if (digit == 5) {
        return TILE_NUM_5;
    }
    if (digit == 6) {
        return TILE_NUM_6;
    }
    if (digit == 7) {
        return TILE_NUM_7;
    }
    if (digit == 8) {
        return TILE_NUM_8;
    }
    if (digit == 9) {
        return TILE_NUM_9;
    }
    return TILE_EMPTY;
}

static int draw_six_digit_value_hw(int fd, int
row, int col, int value)

{
    int clamped_value;
    int divisor;
    int digit;
    int i;
    uint8_t tile_id;
    clamped_value = value;
    if (clamped_value < 0) {
        clamped_value = 0;
    }
    if (clamped_value > 999999) {
        clamped_value = 999999;
    }
    divisor = 100000;
    for (i = 0; i < SCORE_DIGIT_COUNT; i++) {
        digit = clamped_value / divisor;
        clamped_value = clamped_value % divisor;
        tile_id = digit_tile_from_int(digit);
        if (write_tile_hw(fd, row, col + i,
tile_id) < 0) {
            return -1;
        }
        divisor = divisor / 10;
    }
    return 0;
}

/* ================= SCORE / HIGH SCORE DISPLAY
================= */

static int draw_score_label_hw(int fd)

{
    if (write_tile_hw(fd, SCORE_LABEL_ROW,
SCORE_LABEL_COL + 0, TILE_S) < 0) {
        return -1;
    }
    if (write_tile_hw(fd, SCORE_LABEL_ROW,
SCORE_LABEL_COL + 1, TILE_C) < 0) {
        return -1;
    }
    if (write_tile_hw(fd, SCORE_LABEL_ROW,
SCORE_LABEL_COL + 2, TILE_O) < 0) {
        return -1;
    }
    if (write_tile_hw(fd, SCORE_LABEL_ROW,
SCORE_LABEL_COL + 3, TILE_R) < 0) {
        return -1;
    }
```

```
    if (write_tile_hw(fd, SCORE_LABEL_ROW,
SCORE_LABEL_COL + 4, TILE_E) < 0) {
        return -1;
    }
    return 0;
}
static int draw_high_score_label_hw(int fd)
{
    if (write_tile_hw(fd, HIGH_LABEL_ROW,
HIGH_LABEL_COL + 0, TILE_H) < 0) {
        return -1;
    }
    if (write_tile_hw(fd, HIGH_LABEL_ROW,
HIGH_LABEL_COL + 1, TILE_I) < 0) {
        return -1;
    }
    if (write_tile_hw(fd, HIGH_LABEL_ROW,
HIGH_LABEL_COL + 2, TILE_G) < 0) {
        return -1;
    }
    if (write_tile_hw(fd, HIGH_LABEL_ROW,
HIGH_LABEL_COL + 3, TILE_H) < 0) {
        return -1;
    }
    if (write_tile_hw(fd, HIGH_LABEL_ROW,
HIGH_LABEL_COL + 4, TILE_EMPTY) < 0) {
        return -1;
    }
    if (write_tile_hw(fd, HIGH_LABEL_ROW,
HIGH_LABEL_COL + 5, TILE_S) < 0) {
        return -1;
    }
    if (write_tile_hw(fd, HIGH_LABEL_ROW,
HIGH_LABEL_COL + 6, TILE_C) < 0) {
        return -1;
    }
    if (write_tile_hw(fd, HIGH_LABEL_ROW,
HIGH_LABEL_COL + 7, TILE_O) < 0) {
        return -1;
    }
    if (write_tile_hw(fd, HIGH_LABEL_ROW,
HIGH_LABEL_COL + 8, TILE_R) < 0) {
        return -1;
    }
    if (write_tile_hw(fd, HIGH_LABEL_ROW,
HIGH_LABEL_COL + 9, TILE_E) < 0) {
        return -1;
    }
    return 0;
}
static int draw_score_value_hw(int fd, int value)
{
    return draw_six_digit_value_hw(fd,
SCORE_DIGIT_ROW, SCORE_DIGIT_COL, value);
}
static int draw_score_hw(int fd)
{
    if (draw_score_label_hw(fd) < 0) {
        return -1;
    }
    if (draw_score_value_hw(fd, score) < 0) {
        return -1;
    }
    return 0;
}
static int draw_high_score_hw(int fd)
{
    if (draw_high_score_label_hw(fd) < 0) {
        return -1;
    }
    if (draw_six_digit_value_hw(fd,
HIGH_DIGIT_ROW, HIGH_DIGIT_COL, HIGH_SCORE_FIXED)
< 0) {
        return -1;
    }
    return 0;
}
static void update_score_display_if_needed(int
fd)
{
    if (score != last_drawn_score) {
        if (draw_score_value_hw(fd, score) < 0) {
```

```
            perror("draw score value");
        }
        last_drawn_score = score;
    }
}
/* ================= LIFE DISPLAY
================= */
static int draw_lives_hw(int fd)
{
    int display_lives;
    uint8_t lives_tile;
    display_lives = lives;
    if (display_lives < 0) {
        display_lives = 0;
    }
    if (display_lives > 9) {
        display_lives = 9;
    }
    lives_tile =
digit_tile_from_int(display_lives);
    if (write_tile_hw(fd, LIFE_ROW, LIFE_COL + 0,
lives_tile) < 0) {
        return -1;
    }
    if (write_tile_hw(fd, LIFE_ROW, LIFE_COL + 1,
TILE_EMPTY) < 0) {
        return -1;
    }
    if (write_tile_hw(fd, LIFE_ROW, LIFE_COL + 2,
TILE_X) < 0) {
        return -1;
    }
    if (write_tile_hw(fd, LIFE_ROW, LIFE_COL + 3,
TILE_EMPTY) < 0) {
        return -1;
    }
    if (write_tile_hw(fd, LIFE_ROW, LIFE_COL + 4,
TILE_P) < 0) {
        return -1;
    }
    if (write_tile_hw(fd, LIFE_ROW, LIFE_COL + 5,
TILE_EMPTY) < 0) {
        return -1;
    }
    if (write_tile_hw(fd, LIFE_ROW, LIFE_COL + 6,
TILE_L) < 0) {
        return -1;
    }
    if (write_tile_hw(fd, LIFE_ROW, LIFE_COL + 7,
TILE_E) < 0) {
        return -1;
    }
    if (write_tile_hw(fd, LIFE_ROW, LIFE_COL + 8,
TILE_F) < 0) {
        return -1;
    }
    if (write_tile_hw(fd, LIFE_ROW, LIFE_COL + 9,
TILE_T) < 0) {
        return -1;
    }
    return 0;
}
static void update_lives_display_if_needed(int
fd)
{
    if (lives != last_drawn_lives) {
        if (draw_lives_hw(fd) < 0) {
            perror("draw lives");
        }
        last_drawn_lives = lives;
    }
}
/* ================= WIN DISPLAY
================= */
static int draw_you_win_hw(int fd)
{
    if (write_tile_hw(fd, WIN_ROW, WIN_COL + 0,
TILE_Y) < 0) {
        return -1;
    }
```

```
    if (write_tile_hw(fd, WIN_ROW, WIN_COL + 1,
TILE_O) < 0) {

        return -1;

    }

    if (write_tile_hw(fd, WIN_ROW, WIN_COL + 2,
TILE_U) < 0) {

        return -1;

    }

    if (write_tile_hw(fd, WIN_ROW, WIN_COL + 3,
TILE_EMPTY) < 0) {

        return -1;

    }

    if (write_tile_hw(fd, WIN_ROW, WIN_COL + 4,
TILE_W) < 0) {

        return -1;

    }

    if (write_tile_hw(fd, WIN_ROW, WIN_COL + 5,
TILE_I) < 0) {

        return -1;

    }

    if (write_tile_hw(fd, WIN_ROW, WIN_COL + 6,
TILE_N) < 0) {

        return -1;

    }

    return 0;

}

static int draw_game_over_hw(int fd)

{

    if (write_tile_hw(fd, WIN_ROW, WIN_COL + 0,
TILE_G) < 0) return -1;

    if (write_tile_hw(fd, WIN_ROW, WIN_COL + 1,
TILE_A) < 0) return -1;

    if (write_tile_hw(fd, WIN_ROW, WIN_COL + 2,
TILE_M) < 0) return -1;

    if (write_tile_hw(fd, WIN_ROW, WIN_COL + 3,
TILE_E) < 0) return -1;

    if (write_tile_hw(fd, WIN_ROW, WIN_COL + 4,
TILE_EMPTY) < 0) return -1;

    if (write_tile_hw(fd, WIN_ROW, WIN_COL + 5,
TILE_O) < 0) return -1;

    if (write_tile_hw(fd, WIN_ROW, WIN_COL + 6,
TILE_V) < 0) return -1;

    if (write_tile_hw(fd, WIN_ROW, WIN_COL + 7,
TILE_E) < 0) return -1;

    if (write_tile_hw(fd, WIN_ROW, WIN_COL + 8,
TILE_R) < 0) return -1;

    return 0;

}

static void update_win_display_if_needed(int fd)

{

    if (win_drawn) {

        return;

    }

    if (total_pellets <= 0) {

        return;

    }

    if (pellets_eaten < total_pellets) {

        return;

    }

    if (draw_you_win_hw(fd) < 0) {

        perror("draw YOU WIN");

    }

    win_drawn = 1;

    game_is_done = 1;

    if (audio_enabled && siren_handle != -1) {

        audio_stop(siren_handle);

        siren_handle = -1;

    }

}

static uint8_t tile_from_char(char c)

{

    if (c == '.') return TILE_EMPTY;

    if (c == '0') return TILE_EMPTY;

    if (c == '9') return TILE_DOT;

    if (c == 'A') return TILE_POWER_DOT;

    if (c == '1') return TILE_VERT_LEFT;

    if (c == '2') return TILE_VERT_RIGHT;

    if (c == '3') return TILE_HORIZ_TOP;

    if (c == '4') return TILE_HORIZ_BOTTOM;

    if (c == '5') return TILE_CURVE_RIGHT_TOP;

    if (c == '6') return TILE_CURVE_LEFT_TOP;

    if (c == '7') return TILE_CURVE_RIGHT_BOTTOM;
```

```
    if (c == '8') return TILE_CURVE_LEFT_BOTTOM;
    return TILE_EMPTY;
}
static void redraw_entire_maze_hw(int fd)
{
    int row, col;
    for (row = 0; row < LOGIC_H; row++) {
        for (col = 0; col < LOGIC_W; col++) {
            write_tile_hw(fd, row, col,
tile_from_char(tile_map[row][col]));
        }
    }
}
/* ================= PELLETS ================= */
static void init_pellets_from_tile_map(void)
{
    int row;
    int col;
    char c;
    score = 0;
    last_drawn_score = -1;
    lives = 3;
    last_drawn_lives = -1;
    pellets_eaten = 0;
    normal_pellets_eaten = 0;
    power_pellets_eaten = 0;
    total_pellets = 0;
    ghosts_eaten = 0;
    power_until_tick = 0;
    win_drawn = 0;
    for (row = 0; row < LOGIC_H; row++) {
        for (col = 0; col < LOGIC_W; col++) {
            c = tile_map[row][col];
            if (c == '9') {
                pellet_live[row][col] =
PELLET_DOT;
                total_pellets++;
            }
            else if (c == 'A') {
                pellet_live[row][col] =
PELLET_POWER;
                total_pellets++;
            }
            else {
                pellet_live[row][col] =
PELLET_NONE;
            }
        }
    }
}
static int eat_pellet_at_tile(int fd, int row,
int col)
{
    int pellet_kind;
    if (row < 0 || row >= LOGIC_H) {
        return PELLET_NONE;
    }
    if (col < 0 || col >= LOGIC_W) {
        return PELLET_NONE;
    }
    pellet_kind = pellet_live[row][col];
    if (pellet_kind == PELLET_NONE) {
        return PELLET_NONE;
    }
    pellet_live[row][col] = PELLET_NONE;
    pellets_eaten++;
    if (pellet_kind == PELLET_DOT) {
        normal_pellets_eaten++;
        score += SCORE_DOT;
    }
    else if (pellet_kind == PELLET_POWER) {
        power_pellets_eaten++;
        score += SCORE_POWER;
    }
    if (write_tile_hw(fd, row, col, TILE_EMPTY) <
0) {
        perror("clear eaten pellet");
```

```
    }

    return pellet_kind;

}

static int eat_pellets_under_pacman(int fd, const
PacmanState *pac)

{

    int ate_dot;

    int ate_power;

    int kind;

    /*

     * Pac-Man uses a 2x2 tile hitbox. If a power
pellet and a normal

     * pellet are eaten on the same movement
step, power must win.

     */

    ate_dot = 0;

    ate_power = 0;

    kind = eat_pellet_at_tile(fd, pac->pos.y,
pac->pos.x);

    if (kind == PELLET_POWER) ate_power = 1;

    else if (kind == PELLET_DOT) ate_dot = 1;

    kind = eat_pellet_at_tile(fd, pac->pos.y,
pac->pos.x + 1);

    if (kind == PELLET_POWER) ate_power = 1;

    else if (kind == PELLET_DOT) ate_dot = 1;

    kind = eat_pellet_at_tile(fd, pac->pos.y + 1,
pac->pos.x);

    if (kind == PELLET_POWER) ate_power = 1;

    else if (kind == PELLET_DOT) ate_dot = 1;

    kind = eat_pellet_at_tile(fd, pac->pos.y + 1,
pac->pos.x + 1);

    if (kind == PELLET_POWER) ate_power = 1;

    else if (kind == PELLET_DOT) ate_dot = 1;

    if (ate_power) {

        return PELLET_POWER;

    }

    if (ate_dot) {

        return PELLET_DOT;

    }

    return PELLET_NONE;

}

/* ================= GAME LOGIC =================
*/

static int ghosts_are_frightened(int tick)

{

    if (tick < power_until_tick) {

        return 1;

    }

    return 0;

}

static void set_all_ghost_modes(Ghost *b, Ghost
*p, Ghost *i, Ghost *c, GhostMode mode)

{

    if (b->mode != GHOST_EYES) b->mode = mode;

    if (p->mode != GHOST_EYES) p->mode = mode;

    if (i->mode != GHOST_EYES) i->mode = mode;

    if (c->mode != GHOST_EYES) c->mode = mode;

}

static void reset_ghost_to_home(Ghost *ghost)

{

    ghost->pos = ghost->home_pos;

    ghost->prev_pos = ghost->home_pos;

    ghost->move_timer_us = ghost-
>move_duration_us;

    ghost->mode = GHOST_NORMAL;

    ghost->dir = DIR_UP;

    ghost->eaten_count++;

    ghost->leaving_spawn = 1;

}

static void reset_all_positions(PacmanState *pac,
Ghost *b, Ghost *p, Ghost *i, Ghost *c)

{

    pac->pos =
choose_open_sprite_pos(vec_make(PAC_START_X,
PAC_START_Y));

    pac->prev_pos = pac->pos;

    pac->move_timer_us = pac->move_duration_us;

    pac->dir = DIR_RIGHT;

    pac->next_dir = DIR_RIGHT;

    reset_ghost_to_home(b);
```

```
    reset_ghost_to_home(p);
    reset_ghost_to_home(i);
    reset_ghost_to_home(c);
    power_until_tick = 0;
}
static void lose_life_and_reset(int fd,
PacmanState *pac, Ghost *b, Ghost *p, Ghost *i,
Ghost *c)
{
    if (audio_enabled) {
        if (siren_handle != -1) {
            audio_stop(siren_handle);
            siren_handle = -1;
        }
        if (snd_death) {
            audio_play(snd_death, 0);
        }
    }
    usleep(2500000);
    if (lives > 0) {
        lives--;
        reset_all_positions(pac, b, p, i, c);
        update_lives_display_if_needed(fd);
        if (audio_enabled && snd_siren) {
            siren_handle = audio_play(snd_siren,
1);
        }
    } else {
        game_is_done = 1;
        if (draw_game_over_hw(fd) < 0) {
            perror("draw GAME OVER");
        }
        update_lives_display_if_needed(fd);
    }
}
static void update_pacman_direction(PacmanState
*pac)
{
    if (is_legal_pacman_move(pac->pos, pac-
>next_dir)) {
        pac->dir = pac->next_dir;
    }
}
static void step_pacman(PacmanState *pac)
{
    Vec2 old_pos;
    Vec2 next_pos;
    update_pacman_direction(pac);
    if (pac->dir != DIR_NONE &&
is_legal_pacman_move(pac->pos, pac->dir)) {
        old_pos = pac->pos;
        next_pos = next_pos_with_tunnel_wrap(pac-
>pos, pac->dir);
        pac->prev_pos = old_pos;
        pac->pos = next_pos;
        pac->move_timer_us = 0;
        pac->move_duration_us =
PACMAN_MOVE_INTERVAL_US;
    }
}
static Vec2 get_tile_ahead_of_pacman(PacmanState
pac, int amount)
{
    Vec2 target;
    Vec2 step;
    target = pac.pos;
    if (pac.dir == DIR_NONE) {
        return target;
    }
    step = dir_vec[(int)pac.dir];
    target.x = target.x + step.x * amount;
    target.y = target.y + step.y * amount;
    return target;
}
static Vec2 get_blinky_target(PacmanState pac)
{
    return pac.pos;
}
static Vec2 get_pinky_target(PacmanState pac)
```

```
{
    return get_tile_ahead_of_pacman(pac, 4);
}
static Vec2 get_inky_target(PacmanState pac, Vec2
blinky_pos)
{
    Vec2 ahead;
    Vec2 target;
    int vx;
    int vy;
    ahead = get_tile_ahead_of_pacman(pac, 2);
    vx = ahead.x - blinky_pos.x;
    vy = ahead.y - blinky_pos.y;
    target.x = ahead.x + vx;
    target.y = ahead.y + vy;
    return target;
}
static Vec2 get_clyde_target(PacmanState pac,
Ghost clyde)
{
    int dist;
    dist = squared_distance(clyde.pos, pac.pos);
    if (dist <= CLYDE_CLOSE_DIST_SQ) {
        return vec_make(TARGET_BOTTOM_LEFT_X,
TARGET_BOTTOM_LEFT_Y);
    }
    return pac.pos;
}
static Vec2 get_ghost_target(Ghost ghost,
PacmanState pac, Vec2 blinky_pos)
{
    if (ghost.type == GHOST_BLINKY) {
        return get_blinky_target(pac);
    }
    if (ghost.type == GHOST_PINKY) {
        return get_pinky_target(pac);
    }
    if (ghost.type == GHOST_INKY) {
        return get_inky_target(pac, blinky_pos);
    }
    if (ghost.type == GHOST_CLYDE) {
        return get_clyde_target(pac, ghost);
    }
    return pac.pos;
}
static Vec2
get_farthest_corner_from_pacman(PacmanState pac)
{
    Vec2 best;
    Vec2 corner;
    int best_dist;
    int dist;
    best = vec_make(TARGET_TOP_LEFT_X,
TARGET_TOP_LEFT_Y);
    best_dist = squared_distance(best, pac.pos);
    corner = vec_make(TARGET_TOP_RIGHT_X,
TARGET_TOP_RIGHT_Y);
    dist = squared_distance(corner, pac.pos);
    if (dist > best_dist) {
        best_dist = dist;
        best = corner;
    }
    corner = vec_make(TARGET_BOTTOM_LEFT_X,
TARGET_BOTTOM_LEFT_Y);
    dist = squared_distance(corner, pac.pos);
    if (dist > best_dist) {
        best_dist = dist;
        best = corner;
    }
    corner = vec_make(TARGET_BOTTOM_RIGHT_X,
TARGET_BOTTOM_RIGHT_Y);
    dist = squared_distance(corner, pac.pos);
    if (dist > best_dist) {
        best_dist = dist;
        best = corner;
    }
    return best;
}
```

```
static Direction
choose_dir_toward_target_for_leaving(Ghost ghost,
Vec2 target)

{

    Direction best_dir;

    Vec2 next;

    int best_dist;

    int dist;

    int i;

    best_dir = DIR_NONE;

    best_dist = INT_MAX;

    for (i = 0; i < 4; i++) {

        Direction d;

        d = (Direction)i;

        if (!is_legal_ghost_move(ghost, d)) {

            continue;

        }

        next =
next_pos_with_tunnel_wrap(ghost.pos, d);

        dist = squared_distance(next, target);

        if (dist < best_dist) {

            best_dist = dist;

            best_dir = d;

        }

    }

    return best_dir;

}

static int ghost_can_path_to_pos(Ghost ghost,
Vec2 p)

{

    if (!sprite_pos_is_open(p)) {

        return 0;

    }

    /*

     * Eyes and spawn-leaving ghosts must be
allowed inside the spawn box.

     * Normal/frightened ghosts are blocked from
re-entering the spawn.

     */

    if (ghost.mode == GHOST_EYES ||
ghost.leaving_spawn) {

        return 1;

    }

    if (sprite_overlaps_spawn(p)) {

        return 0;

    }

    return 1;

}

static Direction choose_dir_bfs(Ghost ghost, Vec2
target, int allow_reverse)

{

    Vec2 queue[LOGIC_H * LOGIC_W];

    int visited[LOGIC_H][LOGIC_W];

    Direction first_dir[LOGIC_H][LOGIC_W];

    int head;

    int tail;

    int row;

    int col;

    int i;

    Vec2 best;

    int best_dist;

    for (row = 0; row < LOGIC_H; row++) {

        for (col = 0; col < LOGIC_W; col++) {

            visited[row][col] = 0;

            first_dir[row][col] = DIR_NONE;

        }

    }

    if (!tile_in_bounds(ghost.pos)) {

        return DIR_NONE;

    }

    if (!tile_in_bounds(target) ||
!sprite_pos_is_open(target)) {

        target = choose_open_sprite_pos(target);

    }

    head = 0;

    tail = 0;

    queue[tail] = ghost.pos;

    tail++;
```

```
    visited[ghost.pos.y][ghost.pos.x] = 1;

    first_dir[ghost.pos.y][ghost.pos.x] =
DIR_NONE;

    best = ghost.pos;

    best_dist = squared_distance(ghost.pos,
target);

    while (head < tail) {

        Vec2 cur;

        cur = queue[head];

        head++;

        if (squared_distance(cur, target) <
best_dist) {

            best = cur;

            best_dist = squared_distance(cur,
target);

        }

        if (cur.x == target.x && cur.y ==
target.y) {

            best = cur;

            break;

        }

        for (i = 0; i < 4; i++) {

            Direction d;

            Vec2 next;

            d = (Direction)i;

            /*

             * The no-reverse rule only applies
to the first step and only

             * when the caller explicitly asks
for it.

             */

            if (!allow_reverse && cur.x ==
ghost.pos.x && cur.y == ghost.pos.y) {

                if (d ==
opposite_direction(ghost.dir)) {

                    continue;

                }

            }

            next = next_pos_with_tunnel_wrap(cur,
d);

            if (!tile_in_bounds(next)) {

                continue;

            }

            if (visited[next.y][next.x]) {

                continue;

            }

            if (!ghost_can_path_to_pos(ghost,
next)) {

                continue;

            }

            visited[next.y][next.x] = 1;

            if (cur.x == ghost.pos.x && cur.y ==
ghost.pos.y) {

                first_dir[next.y][next.x] = d;

            }

            else {

                first_dir[next.y][next.x] =
first_dir[cur.y][cur.x];

            }

            queue[tail] = next;

            tail++;

        }

    }

    if (best.x == ghost.pos.x && best.y ==
ghost.pos.y) {

        return DIR_NONE;

    }

    return first_dir[best.y][best.x];

}

static Direction choose_ghost_dir(Ghost ghost,
PacmanState pac, Vec2 blinky_pos)

{

    Direction best_dir;

    Direction rev;

    Vec2 next;

    Vec2 target;

    int best_dist;

    int dist;

    int i;

    /*

     * Eyes mode must use real pathfinding
instead of greedy distance.
```

```
     * Greedy distance gets trapped and causes
the eyes to bounce in place.
     */
    if (ghost.mode == GHOST_EYES) {
        return choose_dir_bfs(ghost,
ghost.home_pos, 1);
    }
    /*
     * Leaving spawn also gets pathfinding so
ghosts cleanly exit the box.
     * Use SPAWN_EXIT_Y - 1 because the sprites
are 2 tiles tall; at y=15
     * the lower half still overlaps the spawn
rows.
     */
    if (ghost.leaving_spawn) {
        Vec2 exit_pos;
        exit_pos.x = SPAWN_EXIT_X;
        exit_pos.y = SPAWN_EXIT_Y - 1;
        return choose_dir_bfs(ghost, exit_pos,
1);
    }
    best_dir = DIR_NONE;
    best_dist = INT_MAX;
    if (ghost.mode == GHOST_FRIGHTENED) {
        target =
get_farthest_corner_from_pacman(pac);
    }
    else {
        target = get_ghost_target(ghost, pac,
blinky_pos);
    }
    for (i = 0; i < 4; i++) {
        Direction d;
        d = (Direction)i;
        if (ghost.mode == GHOST_NORMAL) {
            if (d ==
opposite_direction(ghost.dir)) {
                continue;
            }
        }
        if (!is_legal_ghost_move(ghost, d)) {
            continue;
        }
        next =
next_pos_with_tunnel_wrap(ghost.pos, d);
        dist = squared_distance(next, target);
        if (dist < best_dist) {
            best_dist = dist;
            best_dir = d;
        }
    }
    if (best_dir == DIR_NONE) {
        rev = opposite_direction(ghost.dir);
        if (is_legal_ghost_move(ghost, rev)) {
            return rev;
        }
    }
    return best_dir;
}
static void step_ghost(Ghost *ghost, PacmanState
pac, Vec2 blinky_pos, int current_tick)
{
    Direction next_dir;
    Vec2 old_pos;
    Vec2 next_pos;
    /*
     * Eyes mode has absolute priority.
     * The ghost stays as eyes until it actually
reaches its home tile.
     */
    if (ghost->mode == GHOST_EYES) {
        ghost->leaving_spawn = 0;
        if (ghost->pos.x == ghost->home_pos.x &&
ghost->pos.y == ghost->home_pos.y) {
            ghost->prev_pos = ghost->pos;
            ghost->move_timer_us = ghost-
>move_duration_us;
            ghost->leaving_spawn = 1;
            if
(ghosts_are_frightened(current_tick)) {
```

```
                ghost->mode = GHOST_FRIGHTENED;
            }
            else {
                ghost->mode = GHOST_NORMAL;
            }
        }
    }
    next_dir = choose_ghost_dir(*ghost, pac,
blinky_pos);
    if (next_dir != DIR_NONE) {
        old_pos = ghost->pos;
        next_pos =
next_pos_with_tunnel_wrap(ghost->pos, next_dir);
        ghost->dir = next_dir;
        ghost->prev_pos = old_pos;
        ghost->pos = next_pos;
        ghost->move_timer_us = 0;
        if (ghost->mode == GHOST_EYES) {
            ghost->move_duration_us =
GHOST_MOVE_INTERVAL_US / 2;
        }
        else {
            ghost->move_duration_us =
GHOST_MOVE_INTERVAL_US;
        }
    }
    if (ghost->mode == GHOST_FRIGHTENED) {
        ghost->target =
get_farthest_corner_from_pacman(pac);
    }
    else if (ghost->mode == GHOST_EYES) {
        ghost->target = ghost->home_pos;
    }
    else if (ghost->leaving_spawn) {
        ghost->target.x = SPAWN_EXIT_X;
        ghost->target.y = SPAWN_EXIT_Y - 1;
    }
    else {
        ghost->target = get_ghost_target(*ghost,
pac, blinky_pos);
    }
    if (ghost->leaving_spawn) {
        if (!sprite_overlaps_spawn(ghost->pos)) {
            ghost->leaving_spawn = 0;
        }
    }
}
static int handle_one_ghost_contact(int fd,
PacmanState *pac, Ghost *ghost, Ghost *b, Ghost
*p, Ghost *i, Ghost *c)
{
    if (ghost->mode == GHOST_EYES) {
        return 0;
    }
    if (!sprites_overlap(pac->pos, ghost->pos)) {
        return 0;
    }
    if (ghost->mode == GHOST_FRIGHTENED) {
        score += SCORE_GHOST;
        ghosts_eaten++;
        ghost->mode = GHOST_EYES;
        ghost->leaving_spawn = 0;
        ghost->target = ghost->home_pos;
        ghost->move_duration_us =
GHOST_MOVE_INTERVAL_US / 2;
        ghost->move_timer_us = ghost-
>move_duration_us;
        ghost->prev_pos = ghost->pos;
        update_score_display_if_needed(fd);
        if (audio_enabled && snd_eat_ghost) {
            audio_play(snd_eat_ghost, 0);
        }
        return 0;
    }
    lose_life_and_reset(fd, pac, b, p, i, c);
    return 1;
}
static void handle_ghost_contacts(int fd,
PacmanState *pac, Ghost *b, Ghost *p, Ghost *i,
Ghost *c)
```

```
{
    if (handle_one_ghost_contact(fd, pac, b, b,
p, i, c)) {
        return;
    }
    if (handle_one_ghost_contact(fd, pac, p, b,
p, i, c)) {
        return;
    }
    if (handle_one_ghost_contact(fd, pac, i, b,
p, i, c)) {
        return;
    }
    if (handle_one_ghost_contact(fd, pac, c, b,
p, i, c)) {
        return;
    }
}
/* ================= STATUS ================= */
static void print_status(
    PacmanState pac,
    Ghost b,
    Ghost p,
    Ghost i,
    Ghost c,
    int tick
) {
    int k;
    int power_left;
    power_left = power_until_tick - tick;
    if (power_left < 0) {
        power_left = 0;
    }
    printf("\033[H");
    printf("Pac-Man game running on %s\033[K\n",
VGA_PACMAN_DEVICE);
    if (controller_mode) {
        printf("Input: KIWITATA controller | ");
    }
    else {
        printf("Input: keyboard WASD/arrows | ");
    }
    printf("Score=%d | High=%d | Lives=%d |
Pellets=%d/%d | Normal=%d | Super=%d |
Ghosts=%d\033[K\n",
           score,
           HIGH_SCORE_FIXED,
           lives,
           pellets_eaten,
           total_pellets,
           normal_pellets_eaten,
           power_pellets_eaten,
           ghosts_eaten);
    printf("Tick=%d | Power ticks left=%d |
Render=%dus | Pac=%dus | Ghost=%dus | USB
polls=%d | Win=%d\033[K\n",
           tick,
           power_left,
           LOOP_DELAY_US,
           PACMAN_MOVE_INTERVAL_US,
           GHOST_MOVE_INTERVAL_US,
           USB_POLLS_PER_LOOP,
           win_drawn);
    printf("Pac=(%d,%d) Dir=%s Next=%s Anim=%d/%d
LastInput=%s/%s\033[K\n",
           pac.pos.x,
           pac.pos.y,
           dir_name(pac.dir),
           dir_name(pac.next_dir),
           pac.move_timer_us,
           pac.move_duration_us,
           last_input_name,
           dir_name(last_input_dir));
    printf("%s=(%d,%d) T=(%d,%d) %s %s leave=%d |
%s=(%d,%d) T=(%d,%d) %s %s leave=%d\033[K\n",
           b.name, b.pos.x, b.pos.y, b.target.x,
b.target.y, dir_name(b.dir),
ghost_mode_name(b.mode), b.leaving_spawn,
           p.name, p.pos.x, p.pos.y, p.target.x,
p.target.y, dir_name(p.dir),
ghost_mode_name(p.mode), p.leaving_spawn);
```

```
    printf("%s=(%d,%d) T=(%d,%d) %s %s leave=%d |
%s=(%d,%d) T=(%d,%d) %s %s leave=%d\033[K\n",

           i.name, i.pos.x, i.pos.y, i.target.x,
i.target.y, dir_name(i.dir),
ghost_mode_name(i.mode), i.leaving_spawn,

           c.name, c.pos.x, c.pos.y, c.target.x,
c.target.y, dir_name(c.dir),
ghost_mode_name(c.mode), c.leaving_spawn);

    printf("USB packet:");

    for (k = 0; k < last_packet_len; k++) {

        printf(" %02x", last_packet[k]);

    }

    printf("\033[K\n");

    printf("Controller responsiveness improved:
1ms USB timeout, 3 polls/render loop. q
quits.\033[K\n");

    fflush(stdout);

}

/* ================= MAIN ================= */

int main(int argc, char **argv)

{

    int fd;

    int tick_count;

    int pac_frame;

    int ghost_frame;

    int ate_power;

    int pacman_move_accum_us;

    int blinky_accum_us;

    int pinky_accum_us;

    int inky_accum_us;

    int clyde_accum_us;

    int status_accum_us;

    Vec2 blinky_pos_for_ai;

    PacmanState pac;

    Ghost blinky;

    Ghost pinky;

    Ghost inky;

    Ghost clyde;

    (void)argc;

    (void)argv;

    if (validate_tile_map() != 0) {

        return 1;

    }

    signal(SIGINT, handle_sigint);

    fd = open(VGA_PACMAN_DEVICE, O_RDWR);

    if (fd < 0) {

        perror("open " VGA_PACMAN_DEVICE);

        fprintf(stderr, "Run: insmod
./vga_pacman.ko\n");

        return 1;

    }

    controller_mode = joystick_init();

    if (controller_mode) {

        printf("KIWITATA / DragonRise controller
found.\n");

    }

    else {

        printf("Controller not found. Falling
back to keyboard input.\n");

    }

    init_pellets_from_tile_map();

    if (draw_score_hw(fd) < 0) {

        perror("draw initial score");

    }

    last_drawn_score = score;

    if (draw_high_score_hw(fd) < 0) {

        perror("draw high score");

    }

    if (draw_lives_hw(fd) < 0) {

        perror("draw initial lives");

    }

    last_drawn_lives = lives;

    pac.pos =
choose_open_sprite_pos(vec_make(PAC_START_X,
PAC_START_Y));

    pac.prev_pos = pac.pos;

    pac.move_timer_us = PACMAN_MOVE_INTERVAL_US;

    pac.move_duration_us =
PACMAN_MOVE_INTERVAL_US;

    pac.dir = DIR_RIGHT;
```

```
    pac.next_dir = DIR_RIGHT;

    blinky.type = GHOST_BLINKY;

    blinky.name = ghost_name(GHOST_BLINKY);

    blinky.home_pos =
choose_open_sprite_pos(vec_make(BLINKY_HOME_X,
BLINKY_HOME_Y));

    blinky.pos = blinky.home_pos;

    blinky.prev_pos = blinky.pos;

    blinky.move_timer_us =
GHOST_MOVE_INTERVAL_US;

    blinky.move_duration_us =
GHOST_MOVE_INTERVAL_US;

    blinky.dir = DIR_UP;

    blinky.target = pac.pos;

    blinky.mode = GHOST_NORMAL;

    blinky.eaten_count = 0;

    blinky.leaving_spawn = 1;

    pinky.type = GHOST_PINKY;

    pinky.name = ghost_name(GHOST_PINKY);

    pinky.home_pos =
choose_open_sprite_pos(vec_make(PINKY_HOME_X,
PINKY_HOME_Y));

    pinky.pos = pinky.home_pos;

    pinky.prev_pos = pinky.pos;

    pinky.move_timer_us = GHOST_MOVE_INTERVAL_US;

    pinky.move_duration_us =
GHOST_MOVE_INTERVAL_US;

    pinky.dir = DIR_UP;

    pinky.target = pac.pos;

    pinky.mode = GHOST_NORMAL;

    pinky.eaten_count = 0;

    pinky.leaving_spawn = 1;

    inky.type = GHOST_INKY;

    inky.name = ghost_name(GHOST_INKY);

    inky.home_pos =
choose_open_sprite_pos(vec_make(INKY_HOME_X,
INKY_HOME_Y));

    inky.pos = inky.home_pos;

    inky.prev_pos = inky.pos;

    inky.move_timer_us = GHOST_MOVE_INTERVAL_US;

    inky.move_duration_us =
GHOST_MOVE_INTERVAL_US;

    inky.dir = DIR_UP;

    inky.target = pac.pos;

    inky.mode = GHOST_NORMAL;

    inky.eaten_count = 0;

    inky.leaving_spawn = 1;

    clyde.type = GHOST_CLYDE;

    clyde.name = ghost_name(GHOST_CLYDE);

    clyde.home_pos =
choose_open_sprite_pos(vec_make(CLYDE_HOME_X,
CLYDE_HOME_Y));

    clyde.pos = clyde.home_pos;

    clyde.prev_pos = clyde.pos;

    clyde.move_timer_us = GHOST_MOVE_INTERVAL_US;

    clyde.move_duration_us =
GHOST_MOVE_INTERVAL_US;

    clyde.dir = DIR_UP;

    clyde.target = pac.pos;

    clyde.mode = GHOST_NORMAL;

    clyde.eaten_count = 0;

    clyde.leaving_spawn = 1;

    tick_count = 0;

    pac_frame = 0;

    ghost_frame = 0;

    pacman_move_accum_us = 0;

    blinky_accum_us = 0;

    pinky_accum_us = 0;

    inky_accum_us = 0;

    clyde_accum_us = 0;

    status_accum_us = 0;

    terminal_start();

    // Initialize Audio subsystem

    if (audio_init() == 0) {

        audio_enabled = 1;

        snd_start =
audio_load_wav("Sounds/start.wav");

        snd_siren =
audio_load_wav("Sounds/siren0.wav");

        snd_waka0 =
audio_load_wav("Sounds/eat_dot_0.wav");
```

```
        snd_waka1 =
audio_load_wav("Sounds/eat_dot_1.wav");

        snd_death =
audio_load_wav("Sounds/death_0.wav");

        snd_eat_ghost =
audio_load_wav("Sounds/eat_ghost.wav");

    } else {

        printf("Warning: Audio initialization
failed. Running in silent mode.\n");

    }

    if (audio_enabled && snd_start) {

        printf("Playing start music...\n");

        audio_play(snd_start, 0);

        usleep(4500000); // 4.5 second delay for
intro melody

    }

    if (audio_enabled && snd_siren) {

        siren_handle = audio_play(snd_siren, 1);
// Loop background siren

    }

    while (running) {

        Direction requested_dir;

        if (game_is_done > 0) {

            requested_dir =
read_game_input(DIR_NONE);

            if (requested_dir != DIR_NONE) {

                printf("Restarting game...\n");

                init_pellets_from_tile_map();

                reset_all_positions(&pac,
&blinky, &pinky, &inky, &clyde);

                update_score_display_if_needed(fd
);

                update_lives_display_if_needed(fd
);



                redraw_entire_maze_hw(fd);



                game_is_done = 0;

                pacman_move_accum_us = 0;

                blinky_accum_us = 0;

                pinky_accum_us = 0;

                inky_accum_us = 0;

                clyde_accum_us = 0;

                status_accum_us = 0;


                if (audio_enabled && snd_start) {

                    audio_play(snd_start, 0);

                    usleep(4500000);

                }

                if (audio_enabled && snd_siren) {

                    siren_handle =
audio_play(snd_siren, 1);

                }

            }

            usleep(LOOP_DELAY_US);

            continue;

        }

        tick_count++;

        pacman_move_accum_us += LOOP_DELAY_US;

        blinky_accum_us += LOOP_DELAY_US;

        pinky_accum_us += LOOP_DELAY_US;

        inky_accum_us += LOOP_DELAY_US;

        clyde_accum_us += LOOP_DELAY_US;

        status_accum_us += LOOP_DELAY_US;

        advance_pacman_animation(&pac);

        advance_ghost_animation(&blinky);

        advance_ghost_animation(&pinky);

        advance_ghost_animation(&inky);

        advance_ghost_animation(&clyde);

        pac_frame = (tick_count / 4) & 0x3;

        ghost_frame = (tick_count / 8) & 0x1;

        if (ghosts_are_frightened(tick_count)) {

            set_all_ghost_modes(&blinky, &pinky,
&inky, &clyde, GHOST_FRIGHTENED);

        }

        else {

            set_all_ghost_modes(&blinky, &pinky,
&inky, &clyde, GHOST_NORMAL);

        }

        requested_dir =
read_game_input(pac.next_dir);
```

```
        if (requested_dir != DIR_NONE) {

            pac.next_dir = requested_dir;

        }

        if (pacman_move_accum_us >=
PACMAN_MOVE_INTERVAL_US) {

            pacman_move_accum_us -=
PACMAN_MOVE_INTERVAL_US;

            step_pacman(&pac);

            int eaten_kind =
eat_pellets_under_pacman(fd, &pac);

            if (eaten_kind == PELLET_POWER) {

                reset_power_timer_from_now(tick_c
ount);

                set_all_ghost_modes(&blinky,
&pinky, &inky, &clyde, GHOST_FRIGHTENED);

            }

            if (eaten_kind != PELLET_NONE) {

                if (audio_enabled) {

                    audio_play(waka_toggle ?
snd_waka0 : snd_waka1, 0);

                    waka_toggle = !waka_toggle;

                }

            }

            handle_ghost_contacts(fd, &pac,
&blinky, &pinky, &inky, &clyde);

            update_score_display_if_needed(fd);

            update_lives_display_if_needed(fd);

            update_win_display_if_needed(fd);

        }

        blinky_pos_for_ai = blinky.pos;

        int blinky_interval = (blinky.mode ==
GHOST_EYES) ? (GHOST_MOVE_INTERVAL_US / 2) :
GHOST_MOVE_INTERVAL_US;

        if (blinky_accum_us >= blinky_interval) {

            blinky_accum_us -= blinky_interval;

            blinky_pos_for_ai = blinky.pos;

            step_ghost(&blinky, pac,
blinky_pos_for_ai, tick_count);

            handle_ghost_contacts(fd, &pac,
&blinky, &pinky, &inky, &clyde);

        }

        int pinky_interval = (pinky.mode ==
GHOST_EYES) ? (GHOST_MOVE_INTERVAL_US / 2) :
GHOST_MOVE_INTERVAL_US;

        if (pinky_accum_us >= pinky_interval) {

            pinky_accum_us -= pinky_interval;

            step_ghost(&pinky, pac,
blinky_pos_for_ai, tick_count);

            handle_ghost_contacts(fd, &pac,
&blinky, &pinky, &inky, &clyde);

        }

        int inky_interval = (inky.mode ==
GHOST_EYES) ? (GHOST_MOVE_INTERVAL_US / 2) :
GHOST_MOVE_INTERVAL_US;

        if (inky_accum_us >= inky_interval) {

            inky_accum_us -= inky_interval;

            step_ghost(&inky, pac,
blinky_pos_for_ai, tick_count);

            handle_ghost_contacts(fd, &pac,
&blinky, &pinky, &inky, &clyde);

        }

        int clyde_interval = (clyde.mode ==
GHOST_EYES) ? (GHOST_MOVE_INTERVAL_US / 2) :
GHOST_MOVE_INTERVAL_US;

        if (clyde_accum_us >= clyde_interval) {

            clyde_accum_us -= clyde_interval;

            step_ghost(&clyde, pac,
blinky_pos_for_ai, tick_count);

            handle_ghost_contacts(fd, &pac,
&blinky, &pinky, &inky, &clyde);

        }

        // Only update display elements if any
ghost took a step

        if (blinky_accum_us == 0 ||
pinky_accum_us == 0 || inky_accum_us == 0 ||
clyde_accum_us == 0) {

            update_score_display_if_needed(fd);

            update_lives_display_if_needed(fd);

            update_win_display_if_needed(fd);

        }

        if (set_pacman_hw(fd, &pac, pac_frame) <
0) {

            perror("VGA_PACMAN_SET_PACMAN");

        }

        if (set_ghost_hw(fd, 0, &blinky,
ghost_frame) < 0) {
```

```
            perror("VGA_PACMAN_SET_GHOST
blinky");

        }

        if (set_ghost_hw(fd, 1, &inky,
ghost_frame) < 0) {

            perror("VGA_PACMAN_SET_GHOST inky");

        }

        if (set_ghost_hw(fd, 2, &pinky,
ghost_frame) < 0) {

            perror("VGA_PACMAN_SET_GHOST pinky");

        }

        if (set_ghost_hw(fd, 3, &clyde,
ghost_frame) < 0) {

            perror("VGA_PACMAN_SET_GHOST clyde");

        }

        if (status_accum_us >=
STATUS_PRINT_INTERVAL_US) {

            status_accum_us = 0;

            print_status(pac, blinky, pinky,
inky, clyde, tick_count);

        }

        usleep(LOOP_DELAY_US);

    }

    terminal_stop();

    if (audio_enabled) {

        audio_shutdown();

        audio_free_sound(snd_start);

        audio_free_sound(snd_siren);

        audio_free_sound(snd_waka0);

        audio_free_sound(snd_waka1);

        audio_free_sound(snd_death);

        audio_free_sound(snd_eat_ghost);

    }

    if (controller_mode) {

        joystick_close();

    }

    close(fd);

    return 0;

}
```

PACMAN_HW_TEST.C

```
#include <stdio.h>

#include <stdlib.h>

#include <stdint.h>

#include <fcntl.h>

#include <unistd.h>

#include <sys/ioctl.h>

#include <string.h>

#include <errno.h>

#include "vga_pacman.h"

static int dir_to_hw(int dir)

{

    if (dir == 0) return HW_DIR_UP;

    if (dir == 1) return HW_DIR_LEFT;

    if (dir == 2) return HW_DIR_DOWN;

    if (dir == 3) return HW_DIR_RIGHT;

    return HW_DIR_RIGHT;

}

static void write_reg(int fd, uint32_t reg,
uint32_t value)

{

    struct vga_pacman_reg r;

    r.reg = reg;

    r.value = value;

    if (ioctl(fd, VGA_PACMAN_WRITE_REG, &r) < 0)
{

        printf("WRITE_REG failed: %s\n",
strerror(errno));

    }

}

static uint32_t read_reg(int fd, uint32_t reg)

{

    struct vga_pacman_reg r;

    r.reg = reg;

    r.value = 0;

    if (ioctl(fd, VGA_PACMAN_READ_REG, &r) < 0) {

        printf("READ_REG failed: %s\n",
strerror(errno));

        return 0;

    }
```

```
    return r.value;
}

static void write_entity(
    int fd,
    uint32_t base_reg,
    uint32_t x,
    uint32_t y,
    uint32_t hw_dir,
    uint32_t frame,
    uint32_t visible
) {
    struct vga_pacman_entity e;
    e.base_reg = base_reg;
    e.x = x;
    e.y = y;
    e.dir = hw_dir;
    e.frame = frame;
    e.visible = visible;
    if (ioctl(fd, VGA_PACMAN_WRITE_ENTITY, &e) < 0) {
        printf("WRITE_ENTITY failed: %s\n", strerror(errno));
    }
}

static void write_tile(int fd, uint32_t x, uint32_t y, uint32_t tile_id)
{
    struct vga_pacman_tile t;
    t.x = x;
    t.y = y;
    t.tile_id = tile_id;
    if (ioctl(fd, VGA_PACMAN_WRITE_TILE, &t) < 0) {
        printf("WRITE_TILE failed: %s\n", strerror(errno));
    }
}

static void animate_test(int fd)
{
    int i;
    int frame;
    int pac_x;
    int ghost_x;
    for (i = 0; i < 1000; i++) {
        frame = (i / 8) & 3;
        pac_x = 64 + (i % 480);
        ghost_x = 544 - (i % 480);
        write_entity(fd, REG_PAC_X, pac_x, 224, HW_DIR_RIGHT, frame, 1);
        write_entity(fd, REG_G0_X, ghost_x, 160, HW_DIR_LEFT, frame, 1);
        write_entity(fd, REG_G1_X, 160, 192, HW_DIR_RIGHT, frame, 1);
        write_entity(fd, REG_G2_X, 320, 256, HW_DIR_UP, frame, 1);
        write_entity(fd, REG_G3_X, 480, 288, HW_DIR_DOWN, frame, 1);
        usleep(16000);
    }
}

int main(void)
{
    int fd;
    int i;
    fd = open("/dev/vga_pacman", O_RDWR);
    if (fd < 0) {
        printf("Could not open /dev/vga_pacman: %s\n", strerror(errno));
        return 1;
    }
    printf("Pac-Man hardware ioctl test started\n");
    printf("Initial register readback:\n");
    printf("PAC_X    = 0x%04x\n", read_reg(fd, REG_PAC_X));
    printf("PAC_Y    = 0x%04x\n", read_reg(fd, REG_PAC_Y));
    printf("PAC_CTRL = 0x%04x\n", read_reg(fd, REG_PAC_CTRL));
    printf("READY    = 0x%04x\n", read_reg(fd, REG_READY));
    printf("Writing fixed sprite positions\n");
```

```
    write_entity(fd, REG_PAC_X, 304, 224,
HW_DIR_RIGHT, 0, 1);

    write_entity(fd, REG_G0_X, 336, 224,
HW_DIR_LEFT, 0, 1);

    write_entity(fd, REG_G1_X, 368, 224,
HW_DIR_RIGHT, 0, 1);

    write_entity(fd, REG_G2_X, 336, 256,
HW_DIR_UP, 0, 1);

    write_entity(fd, REG_G3_X, 368, 256,
HW_DIR_DOWN, 0, 1);

    printf("Writing test tile IDs along row
2\n");

    for (i = 0; i < 16; i++) {

        write_tile(fd, i + 2, 2, i);

    }

    printf("Animating sprites\n");

    animate_test(fd);

    printf("Pac-Man hardware ioctl test
finished\n");

    close(fd);

    return 0;

}
```

RUN_FILL_SCREEN.SH

```
#!/bin/sh

cd /root/pacman_driver || exit 1

echo "Building..."

make || exit 1

echo "Reloading vga_pacman driver..."

rmmod vga_pacman 2>/dev/null

insmod ./vga_pacman.ko || exit 1

echo "Filling screen with tile..."

./fill_screen_tiles "$@"
```

RUN_PELLET_WALL_MAZE.SH

```
#!/bin/sh

cd /root/pacman_driver || exit 1

echo "Building..."

make || exit 1

echo "Reloading vga_pacman driver..."

rmmod vga_pacman 2>/dev/null

insmod ./vga_pacman.ko || exit 1

echo "Drawing pellet-wall debug maze..."

./draw_pellet_wall_maze
```

Vga_pacman_ioctl.h

```
#ifndef VGA_PACMAN_IOCTL_H

#define VGA_PACMAN_IOCTL_H

#ifdef __linux__

#include <linux/ioctl.h>

#include <linux/types.h>

#else

#include <stdint.h>

#include <sys/ioctl.h>

typedef uint8_t  __u8;

typedef uint16_t __u16;

#endif

#define VGA_PACMAN_DEVICE "/dev/vga_pacman"

#define VGA_PACMAN_SCREEN_W 640

#define VGA_PACMAN_SCREEN_H 480

#define VGA_PACMAN_VISIBLE_TILE_ROWS 30

#define VGA_PACMAN_VISIBLE_TILE_COLS 40

#define VGA_PACMAN_HW_TILE_STRIDE 64

#define VGA_PACMAN_NUM_GHOSTS 4

/* Hardware direction encoding used by
pacman_sprite.sv / ghost_sprite.sv. */

#define VGA_DIR_UP    0

#define VGA_DIR_RIGHT 1

#define VGA_DIR_DOWN  2

#define VGA_DIR_LEFT  3

struct vga_pacman_sprite {

    __u16 x;       /* pixel X, normally 0..639 */

    __u16 y;       /* pixel Y, normally 0..479 */

    __u8  dir;     /* VGA_DIR_* */

    __u8  frame;   /* Pac-Man: 0..3 normally;
Ghost: bit 0 toggles animation */

    __u8  visible; /* 0 hidden, nonzero visible
*/

    __u8  state;   /* 0 normal, 1 vulnerable, 2
eyes */

};
```

```
struct vga_pacman_ghost {

    __u8 index;    /* 0=red, 1=blue/cyan, 2=pink,
3=orange */

    __u8 reserved[3];

    struct vga_pacman_sprite sprite;

};

struct vga_pacman_tile {

    __u16 row;     /* visible row 0..29 */

    __u16 col;     /* visible col 0..39 */

    __u8  tile_id; /* tile number stored in
tilemap RAM */

    __u8  reserved[3];

};

struct vga_pacman_tilemap {

    __u8 tile[VGA_PACMAN_VISIBLE_TILE_ROWS *
VGA_PACMAN_VISIBLE_TILE_COLS];

};

#define VGA_PACMAN_IOC_MAGIC 'p'

#define
VGA_PACMAN_SET_PACMAN  _IOW(VGA_PACMAN_IOC_MAGIC,
0x01, struct vga_pacman_sprite)

#define
VGA_PACMAN_GET_PACMAN  _IOR(VGA_PACMAN_IOC_MAGIC,
0x02, struct vga_pacman_sprite)

#define
VGA_PACMAN_SET_GHOST   _IOW(VGA_PACMAN_IOC_MAGIC,
0x03, struct vga_pacman_ghost)

#define
VGA_PACMAN_GET_GHOST   _IOWR(VGA_PACMAN_IOC_MAGIC
, 0x04, struct vga_pacman_ghost)

#define
VGA_PACMAN_SET_TILE    _IOW(VGA_PACMAN_IOC_MAGIC,
0x05, struct vga_pacman_tile)

#define
VGA_PACMAN_GET_TILE    _IOWR(VGA_PACMAN_IOC_MAGIC
, 0x06, struct vga_pacman_tile)

#define VGA_PACMAN_WRITE_TILEMAP
_IOW(VGA_PACMAN_IOC_MAGIC, 0x07, struct
vga_pacman_tilemap)

#define
VGA_PACMAN_GET_READY   _IOR(VGA_PACMAN_IOC_MAGIC,
0x08, __u8)

#define VGA_PACMAN_CLEAR_READY
_IO(VGA_PACMAN_IOC_MAGIC, 0x09)

#endif /* VGA_PACMAN_IOCTL_H */
```

VGA_PACMAN.C

```
// SPDX-License-Identifier: GPL-2.0

/*

 * vga_pacman.c

 *

 * Direct-mapping misc driver for the Avalon-MM
Pac-Man VGA graphics block.

 *

 * This version does NOT depend on device-tree
matching.

 * insmod vga_pacman.ko should directly create:

 *

 *     /dev/vga_pacman

 *

 * Default physical base:

 *

 *     0xff200000

 *

 * If needed, override with:

 *

 *     insmod vga_pacman.ko phys_base=0xff201000

 *

 * Register map uses 16-bit word addresses:

 *

 *     0x0000 : Pac-Man X

 *     0x0001 : Pac-Man Y

 *     0x0002 : Pac-Man CTRL {frame[2:0],
dir[1:0], visible}

 *     0x0003 : READY

 *

 *     0x0010 + ghost*4 + 0 : Ghost X

 *     0x0010 + ghost*4 + 1 : Ghost Y

 *     0x0010 + ghost*4 + 2 : Ghost CTRL
{frame[2:0], dir[1:0], visible}

 *

 *     0x0100 - 0x087f : Tilemap RAM, 30 visible
rows, 64-word stride.

 */

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/fs.h>
```

```
#include <linux/miscdevice.h>

#include <linux/uaccess.h>

#include <linux/io.h>

#include <linux/mutex.h>

#include <linux/slab.h>

#include <linux/init.h>

#include "vga_pacman_ioctl.h"

#define DRV_NAME "vga_pacman"

#define REG_PAC_X        0x0000

#define REG_PAC_Y        0x0001

#define REG_PAC_CTRL     0x0002

#define REG_READY        0x0003

#define REG_GHOST_BASE   0x0010

#define REG_GHOST_STRIDE 0x0004

#define REG_GHOST_X(g)    (REG_GHOST_BASE + ((g)
* REG_GHOST_STRIDE) + 0)

#define REG_GHOST_Y(g)    (REG_GHOST_BASE + ((g)
* REG_GHOST_STRIDE) + 1)

#define REG_GHOST_CTRL(g) (REG_GHOST_BASE + ((g)
* REG_GHOST_STRIDE) + 2)

#define REG_TILE_BASE    0x0100

#define REG_TILE(row, col) (REG_TILE_BASE +
((row) * VGA_PACMAN_HW_TILE_STRIDE) + (col))

#define DEFAULT_PHYS_BASE 0xff200000

#define DEFAULT_MAP_SPAN  0x40000

/*

 * Most DE1-SoC Platform Designer 16-bit Avalon-
MM slaves appear as halfword

 * spaced registers from HPS Linux, so the
default byte offset is word_addr << 1.

 *

 * If the hardware ever uses 32-bit spacing, load
with:

 *

 *     insmod vga_pacman.ko word_shift=2

 */

static unsigned long phys_base =
DEFAULT_PHYS_BASE;

module_param(phys_base, ulong, 0644);

MODULE_PARM_DESC(phys_base, "physical base
address of Pac-Man Avalon-MM slave");

static unsigned long map_span = DEFAULT_MAP_SPAN;

module_param(map_span, ulong, 0644);

MODULE_PARM_DESC(map_span, "ioremap span in
bytes");

static unsigned int word_shift = 1;

module_param(word_shift, uint, 0644);

MODULE_PARM_DESC(word_shift, "byte offset shift
for Avalon word addresses; default 1 for 16-bit
slave");

struct vga_pacman_dev {

    void __iomem *regs;

    struct miscdevice miscdev;

    struct mutex lock;

};

static struct vga_pacman_dev vga_dev;

static inline void vga_write_word(struct
vga_pacman_dev *vdev, u32 word_addr, u16 value)

{

    iowrite16(value, vdev->regs + (word_addr <<
word_shift));

}

static inline u16 vga_read_word(struct
vga_pacman_dev *vdev, u32 word_addr)

{

    return ioread16(vdev->regs + (word_addr <<
word_shift));

}

static inline u16 pack_ctrl(const struct
vga_pacman_sprite *s)

{

    u16 ctrl;

    ctrl = 0;

    ctrl |= (s->visible ? 1 : 0) & 0x1;

    ctrl |= (s->dir & 0x3) << 1;

    ctrl |= (s->frame & 0x7) << 3;

    ctrl |= (s->state & 0x3) << 6;

    return ctrl;

}

static inline void unpack_ctrl(u16 ctrl, struct
vga_pacman_sprite *s)

{
```

```
    s->visible = ctrl & 0x1;

    s->dir = (ctrl >> 1) & 0x3;

    s->frame = (ctrl >> 3) & 0x7;

    s->state = (ctrl >> 6) & 0x3;

}

static int validate_sprite(const struct
vga_pacman_sprite *s)

{

    if (s->x >= VGA_PACMAN_SCREEN_W) {

        return -EINVAL;

    }

    if (s->y >= VGA_PACMAN_SCREEN_H) {

        return -EINVAL;

    }

    if (s->dir > VGA_DIR_LEFT) {

        return -EINVAL;

    }

    if (s->frame > 7) {

        return -EINVAL;

    }

    return 0;

}

static void write_pacman(struct vga_pacman_dev
*vdev, const struct vga_pacman_sprite *s)

{

    vga_write_word(vdev, REG_PAC_X, s->x);

    vga_write_word(vdev, REG_PAC_Y, s->y);

    vga_write_word(vdev, REG_PAC_CTRL,
pack_ctrl(s));

}

static void read_pacman(struct vga_pacman_dev
*vdev, struct vga_pacman_sprite *s)

{

    s->x = vga_read_word(vdev, REG_PAC_X) &
0x03ff;

    s->y = vga_read_word(vdev, REG_PAC_Y) &
0x01ff;

    unpack_ctrl(vga_read_word(vdev,
REG_PAC_CTRL), s);

}

static void write_ghost(struct vga_pacman_dev
*vdev, u8 index, const struct vga_pacman_sprite
*s)

{

    vga_write_word(vdev, REG_GHOST_X(index), s-
>x);

    vga_write_word(vdev, REG_GHOST_Y(index), s-
>y);

    vga_write_word(vdev, REG_GHOST_CTRL(index),
pack_ctrl(s));

}

static void read_ghost(struct vga_pacman_dev
*vdev, u8 index, struct vga_pacman_sprite *s)

{

    s->x = vga_read_word(vdev,
REG_GHOST_X(index)) & 0x03ff;

    s->y = vga_read_word(vdev,
REG_GHOST_Y(index)) & 0x01ff;

    unpack_ctrl(vga_read_word(vdev,
REG_GHOST_CTRL(index)), s);

}

static long vga_pacman_ioctl(struct file *file,
unsigned int cmd, unsigned long arg)

{

    struct vga_pacman_dev *vdev;

    void __user *argp;

    struct vga_pacman_sprite sprite;

    struct vga_pacman_ghost ghost;

    struct vga_pacman_tile tile;

    struct vga_pacman_tilemap *tilemap;

    __u8 ready;

    int ret;

    int row;

    int col;

    vdev = &vga_dev;

    argp = (void __user *)arg;

    tilemap = NULL;

    ret = 0;

    if (_IOC_TYPE(cmd) != VGA_PACMAN_IOC_MAGIC) {

        return -ENOTTY;

    }

    mutex_lock(&vdev->lock);
```

```
    switch (cmd) {
        case VGA_PACMAN_SET_PACMAN:
            if (copy_from_user(&sprite, argp,
sizeof(sprite))) {
                ret = -EFAULT;
                break;
            }
            ret = validate_sprite(&sprite);
            if (ret != 0) {
                break;
            }
            write_pacman(vdev, &sprite);
            break;
        case VGA_PACMAN_GET_PACMAN:
            memset(&sprite, 0, sizeof(sprite));
            read_pacman(vdev, &sprite);
            if (copy_to_user(argp, &sprite,
sizeof(sprite))) {
                ret = -EFAULT;
            }
            break;
        case VGA_PACMAN_SET_GHOST:
            if (copy_from_user(&ghost, argp,
sizeof(ghost))) {
                ret = -EFAULT;
                break;
            }
            if (ghost.index >=
VGA_PACMAN_NUM_GHOSTS) {
                ret = -EINVAL;
                break;
            }
            ret = validate_sprite(&ghost.sprite);
            if (ret != 0) {
                break;
            }
            write_ghost(vdev, ghost.index,
&ghost.sprite);
            break;
        case VGA_PACMAN_GET_GHOST:
            if (copy_from_user(&ghost, argp,
sizeof(ghost))) {
                ret = -EFAULT;
                break;
            }
            if (ghost.index >=
VGA_PACMAN_NUM_GHOSTS) {
                ret = -EINVAL;
                break;
            }
            read_ghost(vdev, ghost.index,
&ghost.sprite);
            if (copy_to_user(argp, &ghost,
sizeof(ghost))) {
                ret = -EFAULT;
            }
            break;
        case VGA_PACMAN_SET_TILE:
            if (copy_from_user(&tile, argp,
sizeof(tile))) {
                ret = -EFAULT;
                break;
            }
            if (tile.row >=
VGA_PACMAN_VISIBLE_TILE_ROWS) {
                ret = -EINVAL;
                break;
            }
            if (tile.col >=
VGA_PACMAN_VISIBLE_TILE_COLS) {
                ret = -EINVAL;
                break;
            }
            vga_write_word(vdev,
REG_TILE(tile.row, tile.col), tile.tile_id);
            break;
        case VGA_PACMAN_GET_TILE:
            if (copy_from_user(&tile, argp,
sizeof(tile))) {
                ret = -EFAULT;
                break;
```

```
            }

            if (tile.row >=
VGA_PACMAN_VISIBLE_TILE_ROWS) {

                ret = -EINVAL;

                break;

            }

            if (tile.col >=
VGA_PACMAN_VISIBLE_TILE_COLS) {

                ret = -EINVAL;

                break;

            }

            tile.tile_id = vga_read_word(vdev,
REG_TILE(tile.row, tile.col)) & 0xff;

            if (copy_to_user(argp, &tile,
sizeof(tile))) {

                ret = -EFAULT;

            }

            break;

        case VGA_PACMAN_WRITE_TILEMAP:

            tilemap = memdup_user(argp,
sizeof(*tilemap));

            if (IS_ERR(tilemap)) {

                ret = PTR_ERR(tilemap);

                tilemap = NULL;

                break;

            }

            for (row = 0; row <
VGA_PACMAN_VISIBLE_TILE_ROWS; row++) {

                for (col = 0; col <
VGA_PACMAN_VISIBLE_TILE_COLS; col++) {

                    vga_write_word(

                        vdev,

                        REG_TILE(row, col),

                        tilemap->tile[row *
VGA_PACMAN_VISIBLE_TILE_COLS + col]

                    );

                }

            }

            break;

        case VGA_PACMAN_GET_READY:

            ready = vga_read_word(vdev,
REG_READY) & 0x1;

            if (copy_to_user(argp, &ready,
sizeof(ready))) {

                ret = -EFAULT;

            }

            break;

        case VGA_PACMAN_CLEAR_READY:

            vga_write_word(vdev, REG_READY, 0);

            break;

        default:

            ret = -ENOTTY;

            break;

    }

    kfree(tilemap);

    mutex_unlock(&vdev->lock);

    return ret;

}

static int vga_pacman_open(struct inode *inode,
struct file *file)

{

    return 0;

}

static int vga_pacman_release(struct inode
*inode, struct file *file)

{

    return 0;

}

static const struct file_operations
vga_pacman_fops = {

    .owner = THIS_MODULE,

    .open = vga_pacman_open,

    .release = vga_pacman_release,

    .unlocked_ioctl = vga_pacman_ioctl,

#ifdef CONFIG_COMPAT

    .compat_ioctl = vga_pacman_ioctl,

#endif

};

static int __init vga_pacman_init(void)

{
```

```
    int ret;

    printk(KERN_INFO DRV_NAME ": loading direct-
mapped driver\n");

    printk(KERN_INFO DRV_NAME ":
phys_base=0x%08lx map_span=0x%08lx
word_shift=%u\n",

           phys_base,

           map_span,

           word_shift);

    memset(&vga_dev, 0, sizeof(vga_dev));

    mutex_init(&vga_dev.lock);

    vga_dev.regs = ioremap(phys_base, map_span);

    if (vga_dev.regs == NULL) {

        printk(KERN_ERR DRV_NAME ": ioremap
failed\n");

        return -ENOMEM;

    }

    vga_dev.miscdev.minor = MISC_DYNAMIC_MINOR;

    vga_dev.miscdev.name = "vga_pacman";

    vga_dev.miscdev.fops = &vga_pacman_fops;

    vga_dev.miscdev.mode = 0666;

    ret = misc_register(&vga_dev.miscdev);

    if (ret != 0) {

        printk(KERN_ERR DRV_NAME ": misc_register
failed: %d\n", ret);

        iounmap(vga_dev.regs);

        vga_dev.regs = NULL;

        return ret;

    }

    printk(KERN_INFO DRV_NAME ": loaded as
/dev/vga_pacman\n");

    return 0;

}

static void __exit vga_pacman_exit(void)

{

    printk(KERN_INFO DRV_NAME ": unloading\n");

    misc_deregister(&vga_dev.miscdev);

    if (vga_dev.regs != NULL) {

        iounmap(vga_dev.regs);

        vga_dev.regs = NULL;

    }

}

module_init(vga_pacman_init);

module_exit(vga_pacman_exit);

MODULE_AUTHOR("Pac-Man DE1-SoC project");

MODULE_DESCRIPTION("Direct-mapped Avalon-MM
driver for Pac-Man VGA tile/sprite graphics");

MODULE_LICENSE("GPL");
```

VGA_PACMAN.H

```
#ifndef VGA_PACMAN_H

#define VGA_PACMAN_H

#include <linux/ioctl.h>

#ifdef __KERNEL__

#include <linux/types.h>

#else

#include <stdint.h>

#endif

#define VGA_PACMAN_MAGIC 'p'

#define REG_PAC_X       0x00000

#define REG_PAC_Y       0x00001

#define REG_PAC_CTRL    0x00002

#define REG_READY       0x00003

#define REG_G0_X        0x00010

#define REG_G0_Y        0x00011

#define REG_G0_CTRL     0x00012

#define REG_G1_X        0x00014

#define REG_G1_Y        0x00015

#define REG_G1_CTRL     0x00016

#define REG_G2_X        0x00018

#define REG_G2_Y        0x00019

#define REG_G2_CTRL     0x0001A

#define REG_G3_X        0x0001C

#define REG_G3_Y        0x0001D

#define REG_G3_CTRL     0x0001E

#define REG_TILE_FIRST  0x00100

#define TILEMAP_STRIDE  64

#define TILEMAP_COLS    40

#define TILEMAP_ROWS    30
```

```
#define HW_DIR_UP       0

#define HW_DIR_RIGHT    1

#define HW_DIR_DOWN     2

#define HW_DIR_LEFT     3

struct vga_pacman_reg {

    uint32_t reg;

    uint32_t value;

};

struct vga_pacman_entity {

    uint32_t base_reg;

    uint32_t x;

    uint32_t y;

    uint32_t dir;

    uint32_t frame;

    uint32_t visible;

};

struct vga_pacman_tile {

    uint32_t x;

    uint32_t y;

    uint32_t tile_id;

};

#define
VGA_PACMAN_WRITE_REG    _IOW(VGA_PACMAN_MAGIC, 1,
struct vga_pacman_reg)

#define
VGA_PACMAN_READ_REG     _IOWR(VGA_PACMAN_MAGIC,
2, struct vga_pacman_reg)

#define VGA_PACMAN_WRITE_ENTITY
_IOW(VGA_PACMAN_MAGIC, 3, struct
vga_pacman_entity)

#define
VGA_PACMAN_WRITE_TILE   _IOW(VGA_PACMAN_MAGIC, 4,
struct vga_pacman_tile)

#endif
```

VGA_PACMAN.MOD.C

```
#include <linux/build-salt.h>

#include <linux/module.h>

#include <linux/vermagic.h>

#include <linux/compiler.h>

BUILD_SALT;

MODULE_INFO(vermagic, VERMAGIC_STRING);

MODULE_INFO(name, KBUILD_MODNAME);

__visible struct module __this_module

__attribute__((section(".gnu.linkonce.this_module
"))) = {

    .name = KBUILD_MODNAME,

    .init = init_module,

#ifdef CONFIG_MODULE_UNLOAD

    .exit = cleanup_module,

#endif

    .arch = MODULE_ARCH_INIT,

};

#ifdef RETPOLINE

MODULE_INFO(retpoline, "Y");

#endif

static const char __module_depends[]

__used

__attribute__((section(".modinfo"))) =

"depends=";
```