FPGA-Driven 'Forest Fire and Ice' Game: Design and Implementation

Yonghao Lin
yl5763@columbia.edu

Yang Cao
yc4535@columbia.edu

Zhenqi Li
zl3508@columbia.edu

Yifan Mao
ym3064@columbia.edu

Weijie wang
ww2739@columbia.edu

Contents

1 Introduction

This project aims to implement a simplified version of the popular cooperative puzzle-platformer game *Fireboy and Watergirl* (We call it **ForestFireIce**) on the Terasic DE1-SoC FPGA platform. The game features two characters—Fire Boy and Water Girl—who must navigate through a series of levels filled with obstacles, traps, and interactive elements.

The system leverages both the FPGA fabric and the Hard Processor System (HPS) of the Cyclone V SoC. The FPGA is responsible for generating 640×480 @ 60Hz VGA video output, performing real-time tile-and-sprite composition and audio control part. Meanwhile, the dual-core ARM processor executes high-level game logic, player control, and system coordination.

The game is controlled via a USB Joy Pad input, with the video output rendered to a VGA monitor. The game is designed to run at a steady 60 frames per second to ensure smooth and responsive interaction.

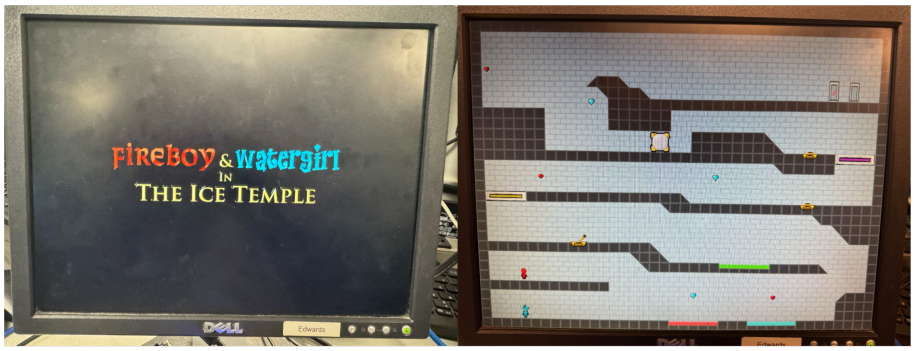

Figure 1: Game of "Forest Fire and Ice"

2 System Block Diagram

This system clearly partitioned into two domains:

- **Hardware Domain (FPGA / SystemVerilog)** — handles all pixel-rate activities (VGA timing, tile+sprite mixing, audio play).
- **Software Domain (ARM Core / C)**: Focuses on high-level game logic such as collision detection, rule enforcement, and player state transitions.

Communication between the software and hardware domains is conducted via the **Avalon Bus**, a memory-mapped interface that allows the ARM processor to write/read registers/BRAM in the FPGA.

2.1 Data Communication Design

To maximize performance and maintain modularity, all rendering is handled by the FPGA hardware. The software domain is responsible only for transmitting high-level game state data to the hardware. Each sprite (e.g., Fire Boy, Water Girl, game objects) is described by a fixed-size sprite attribute entry in the memory-mapped `SPRITE_ATTR_TABLE`, updated once per frame.

Each sprite attribute entry (Inside `SPRITE_ATTR_TABLE`) includes the following fields:

- **Enable**: A flag indicating sprite visibility (1 = visible, 0 = hidden).
- **Flip**: A horizontal flip control for mirroring the sprite.
- **X, Y Coordinates**: The sprite's position on screen in pixel units (sprite_x: 0–639, sprite_y: 0–479).
- **Frame ID**: An 8-bit index used to select a specific frame from the sprite pattern ROM.

These sprite attributes are written by the software to the MMIO `SPRITE_ATTR_TABLE`, which is stored in FPGA on-chip RAM and used directly by the sprite-and-tile displaying.

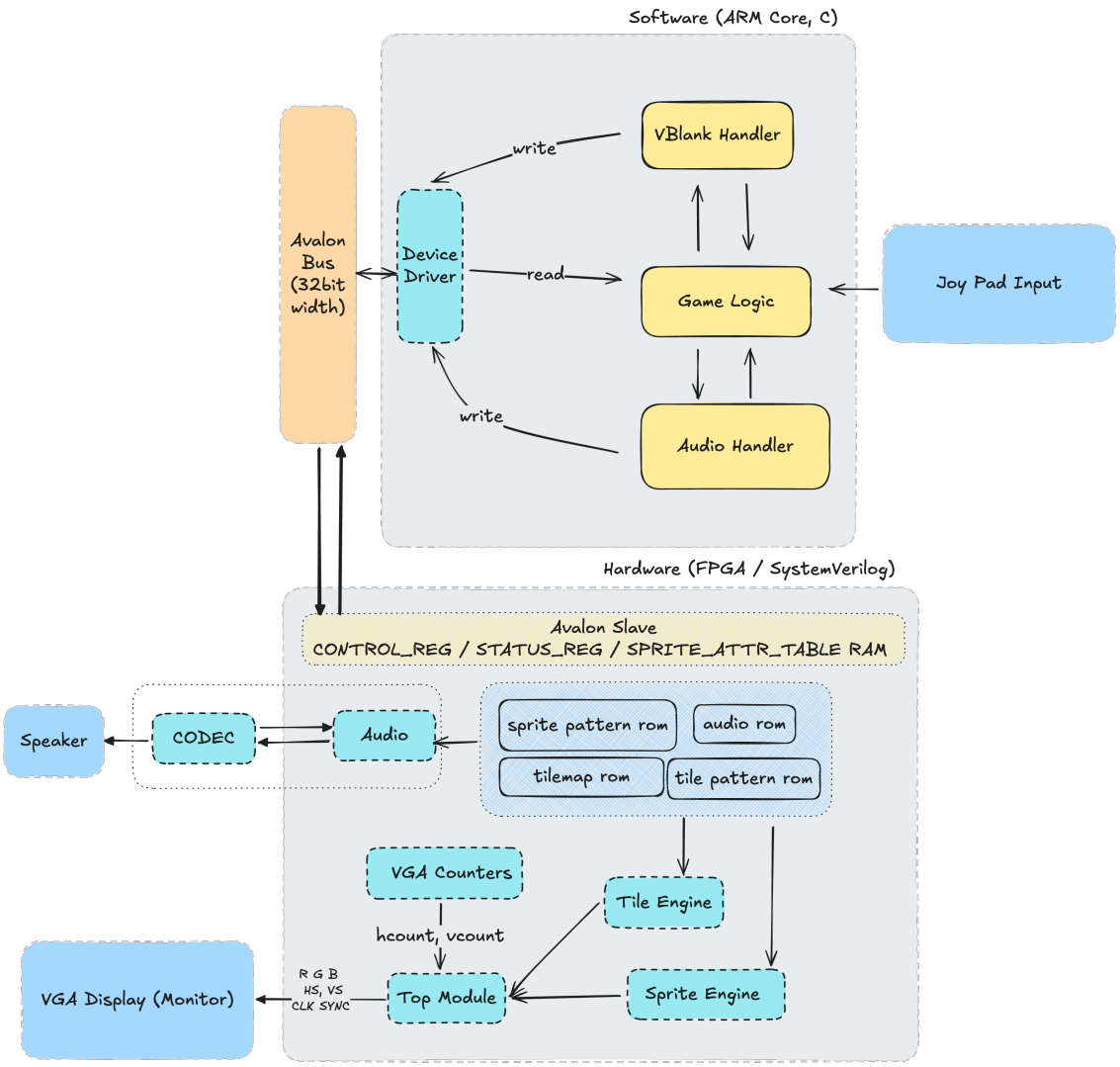

Figure 2: Block Diagram of "Forest Fire and Ice"

2.2 Hardware Responsibilities

- Generates synchronized VGA signals (HSYNC, VSYNC, RGB) for real-time video output.
- Stores tile maps, tile/sprite patterns, and audio samples in on-chip BRAM.
- Provides MMIO-based register interfaces (CONTROL_REG, STATUS_REG) accessible from HPS.
- Selects and renders tilemap tiles using a Tile Engine based on CONTROL_REG[1:0] selection.
- Manages sprite rendering and composition over tiles via the Sprite Engine.
- Outputs audio data stored in BRAM to WM8731 codec.

2.3 Software Responsibilities

The software periodically polls input devices and executes all high-level game logic. Its responsibilities include:

- Parsing user input (e.g., from USB Joy Pad) to update player movement and game state.
- Handling interactions with environment elements such as doors, lever, and switches.
- Detecting collisions in software.
- Polling the `STATUS_REG` to read VGA scanline (`vcount`) and safely write updates during the VBlank window (i.e., when `vcount` ≥ 480).

- Updating the `CONTROL_REG` to select the current tilemap and trigger audio playback.

By separating game logic from display logic, and synchronizing all MMIO writes during the vertical blanking interval, the software ensures smooth, tear-free frame rendering at 60Hz.

3 Hardware

3.1 VGA Display Overview

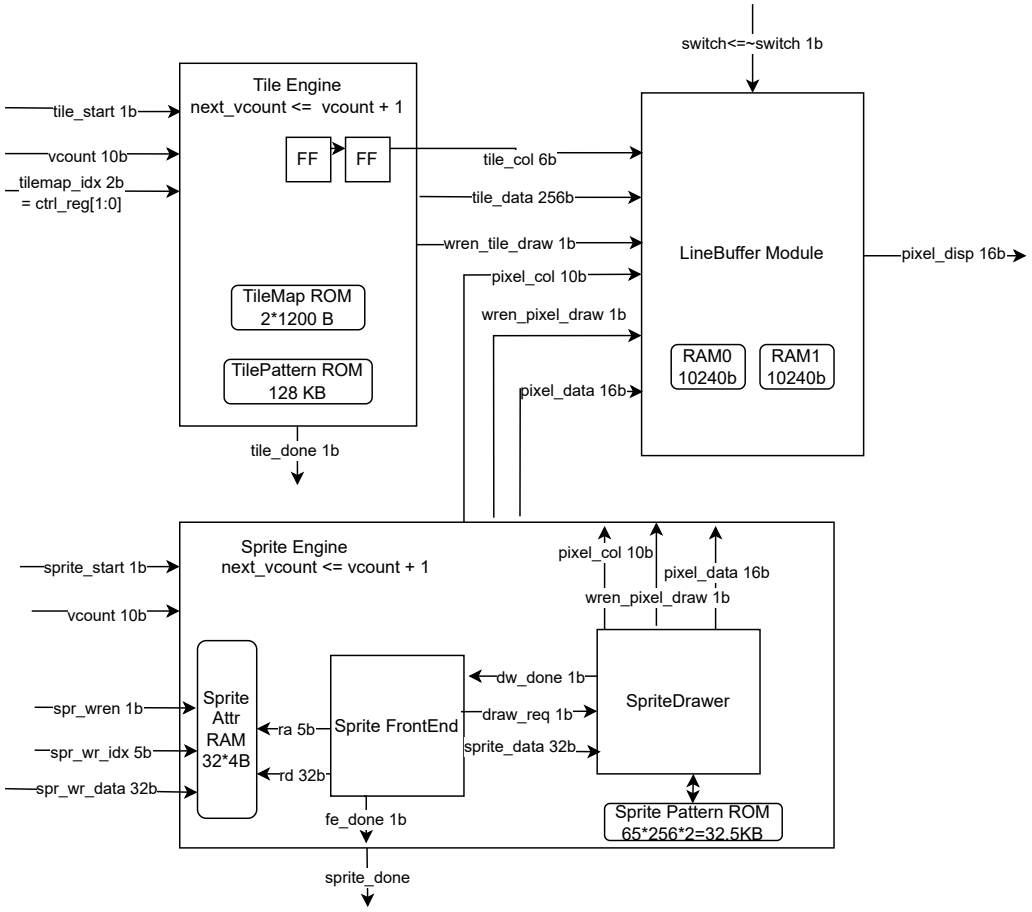

Figure 3: VGA Display Overview

`vga_top` is the single entry point of the graphics system. It accepts a 50 MHz core clock, reset, and a Avalon-slave interface, and it emits all VGA signals for a 640×480@60 Hz display (25 MHz pixel clock, 8-bit `R`/`G`/`B`, `HS`, `VS`, `BLANK_n`, `SYNC_n`).

Internally, two generators feed a dual-bank line buffer: **(i)** `tile_engine` streams 40 background tiles per line, **(ii)** `sprite_engine` overwrites individual pixels wherever an non-transparent pixel is present. Both write 16-bit RGB555^{T} words in which MSB (i.e. bit 15) is a transparency bit; writes are simply suppressed when that bit is '1', so layering is decided entirely by write-enable logic and the later sprite pass. A single flip-flop swaps the draw / display line buffers at the end of each line (during HBlank), allowing one line buffer to be scanned out while the next is rendered.

During scan-out the 5-bit color fields are left-shifted by three to produce 8-bit VGA levels, duplicating each logical pixel horizontally so that the 25 MHz clock meets the VGA 640×480 timing. All finer details of timing, buffering and engine protocols are deferred to the following sections.

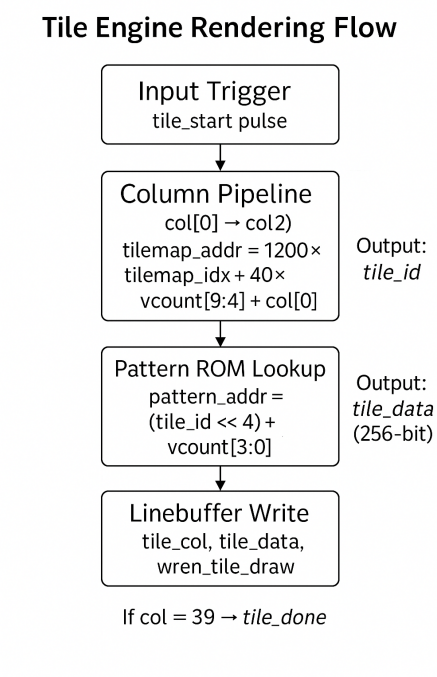

Figure 4: Tile Engine

3.2 Tile Engine

The `tile_engine` generates the *background layer*, delivering one 16-pixel chunk (a row of a 16×16 tile) every clock while traversing per line. Its external interface is minimal yet sufficient for pipelined streaming:

Port	Dir.	Width	Function
`clk`, `reset`	in	1	global timing & reset
`tile_start`	in	1	pulse at $hcount = 0$ of each visible line
`tilemap_idx`	in	2	selects one of our 40×30 tile maps
`vcount`	in	10	current scanning line (0–479)
`tile_col`	out	6	tile column (0–39) written this cycle
`tile_data`	out	256	16 pixels, RGB555^{T}
`wren_tile_draw`	out	1	write enable for the line-buffer draw bank
`tile_done`	out	1	goes high after column 39 retires

Operation. When `tile_start` asserts, a 2-deep column pipeline (`col[1]`...`col[2]`) is initialised to zero. Each cycle the engine performs two table look-ups:

$$tileID\_addr = 1200 \times tilemap\_idx + 40 \times \big(next\_vcount[9:4]\big) + col[0] \qquad \Longrightarrow \qquad tile\_id \in [0, 255]$$

$$pattern\_addr = (tile\_id \ll 4) + next\_vcount[3:0] \quad \Longrightarrow \quad tile\_data \in \text{RGB555}^{\text{T}} \times 16$$

The extra pipeline stages hide the dual-port ROM latency so that `wren_tile_draw` is asserted exactly when the 256-bit `tile_data` arrives, and the corresponding column index is presented on `tile_col`. After the 40^{th} column the engine raises `tile_done`, signalling to `vga_top` that the background for the current drawing line is complete and the sprite pass may begin.

Transparency and layering. Background tiles are always written, because each tile pixel has its MSB *always* be 0, indicating non-transparent. During the sprite pass, a pixel with MSB = 1 is considered transparent and its write is skipped; a pixel with MSB = 0 is non-transparent and overwrites the background pixel at the same location, implementing the desired layering.

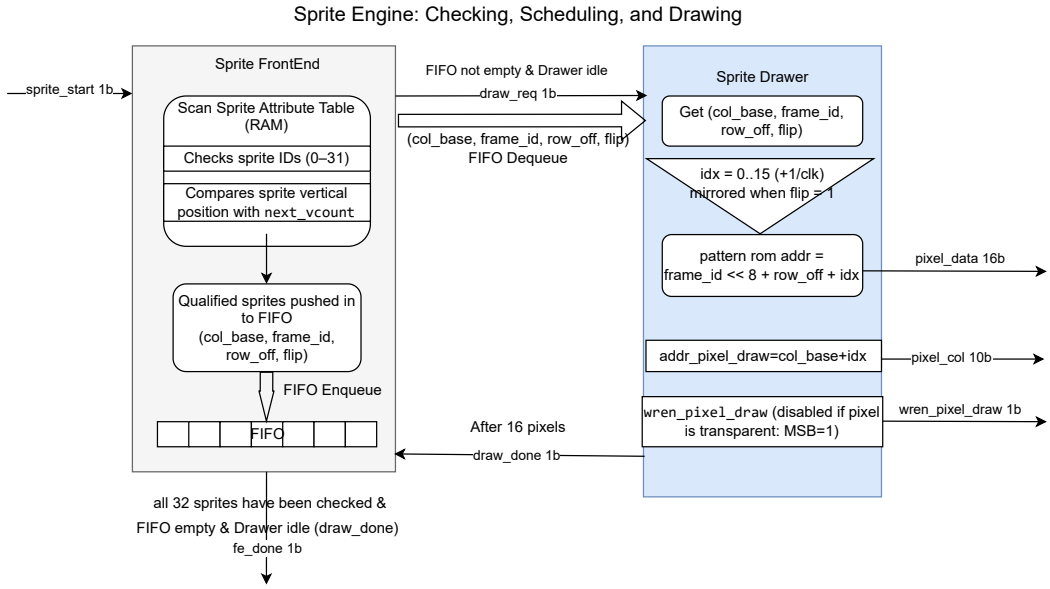

Figure 5: Sprite Engine

3.3 Sprite Engine

The sprite layer is rendered by the composite module `sprite_engine`, which internally instantiates two tightly-coupled sub-modules: `sprite_frontend` (checking & scheduling) and `sprite_drawer` (pixel drawing). Together they transform a RAM-based attribute table into an ordered stream of RGB555$^{\mathrm{T}}$ pixels that overwrite the background tiles already stored in the line buffer.

Maintains a *sprite attribute table RAM* that software can modify at any time (one 32-bit word per sprite attribute entry: *enable*, *frame_id*, *x*, *y*, *flip*).

External interface (as seen by `vga_top`).

Port	Dir.	Width	Meaning
`clk`, `reset`	in	1	system timing
`sprite_start`	in	1	pulse after `tile_done`
`vcount`	in	10	current scanning line (0–479)
`addr_pixel_draw`	out	10	horizontal pixel address (0–639)
`data_pixel_draw`	out	16	pixel color, RGB555$^{\mathrm{T}}$
`wren_pixel_draw`	out	1	write enable for the line-buffer draw bank
`sprite_done`	out	1	asserted when all sprites for the line are drawn

Only non-transparent pixels ($T = 0$) generate a write; transparent ones will be skipped, accomplishing layering without extra logic.

3.3.1 `sprite_frontend` – checking and scheduling

- At each `sprite_start`, it scans the table in order from **sprite_id** 0 to 31, covering all 32 sprite attribute entries, comparing the drawing `next_vcount` with every sprite's vertical position to decide whether it should be presented in the current drawing line (i.e. `next_vcount`) .
- Attributes of qualifying (active) sprites (column base, frame ID, row offset within the sprite, and flip flag) — are pushed into a small FIFO buffer with 7 usable slots (8 total, with one reserved to distinguish between 'full' and 'empty' states). The FIFO preserves the on-screen priority: sprites pushed later appear on top of earlier ones.
- When the drawer is idle and the FIFO is not empty the frontend asserts `draw_req` together with the attributes of the sprite (FIFO dequeue).

- When all sprite attributes (32 sprites) have been checked, the FIFO is empty, and `draw_done` is high (indicating that the `sprite_drawer` is not actively drawing pixels), the `sprite_frontend` triggers `fe_done`, upon which the `sprite_engine` triggers `sprite_done`.

3.3.2 `sprite_drawer` – pixel drawing

- On `draw_req` it raises `start`, latches the attributes, and begins streaming 16 pixels from a sprite pattern ROM (speed: 1 pixel/clk):

$$rom\_addr = (frame\_id \ll 8) + row\_offset + idx$$

  where $idx = 0 \ldots 15$, +1/clk (mirrored when *flip*=1).

- Each clock, `data_pixel_draw` presents one RGB555$^{\mathrm{T}}$ 16-bit word and `addr_pixel_draw` emits $Col_{\mathrm{base}} + idx$. If T/data[15] = 1 the pixel is transparent and `wren_pixel_draw` is disabled.

- After 16 pixels, it pulses `draw_done`, allowing the next sprite attribute in the FIFO to dequeue and be sent to the `sprite_drawer` to start drawing.

3.3.3 Unified appearance to `vga_top`

All handshaking between `sprite_frontend` and `sprite_drawer` is internal; the composite `sprite_engine` exposes just some clean interfaces to `vga_top`. The top-level therefore treats the sprite subsystem as a stateless producer that, once triggered by `sprite_start`, finishes autonomously and announces completion with `sprite_done`. This abstraction keeps the higher-level design simple while still allowing dozens of independently moving, flipping and animating sprites to be drawn every line with deterministic timing.

3.4 Line Buffer

The double-buffered **line buffer** decouples pixel drawing from pixel displaying, allowing tile and sprite rendering to complete within the full 1600 clock cycles (800 pixels × 2 clocks per pixel). It consists of two identical dual–port RAM that are alternately designated ***draw*** or ***display***; a single flip-flop in `vga_top` swaps their roles at the end of every scan-line (during HBlank).

Tile write port (*A-port*) Addressed by `tile_col` (0–39) and enabled with `wren_tile_draw`, it stores a 256 bit word containing 16 RGB555$^{\mathrm{T}}$ pixels—the current drawing line of a 16 × 16 tile.

Sprite write port (*B-port*) Addressed by `addr_pixel_draw` (0–639) and enabled with `wren_pixel_draw`, it overwrites individual 16-bit pixels. The write enable is forced low when the incoming pixel has T = 1; thus transparent pixel data leave the tile color intact, while non-transparent sprite pixels automatically stick "on top".

Display read port While one line buffer is being written by `Tile Engine` and `Sprite Engine`, the other is read by the VGA Top module using `addr_pixel_disp`. Each 16-bit pixel color is widened to 24-bit RGB on the fly ($R, G, B \ll 3$) and transmitted to the monitor, with every logical pixel duplicated horizontally so that a 25 MHz clock meets 640×480 VGA timing.

Because all compositing decisions are resolved at write time through the transparency bit, the read side merely streams pixels at video rate; no run-time arbitration or blending logic is required. The ping-pong (i.e. two line buffers) organization guarantees zero contention: while one line buffer is scanned out, both `Tile Engine` and `Sprite Engine` can freely write the next line into the other line buffer, completing long before the scanner reaches the right border (HBlank) of the display.

3.5 Audio Play

Figure 6 illustrates the Qsys connection in detail. The `audio_0` module is an Intel FPGA IP core responsible for sending data to the audio codec. The `audio_play` module feeds the correct data to `audio_0` using a streaming source interface, which includes a valid bit, an 16-bit data signal, and a ready signal input. The `audio_pll` module—another Intel IP core—generates a 12.288 MHz clock signal from a 50 MHz reference, which is used to drive the codec. Additionally,

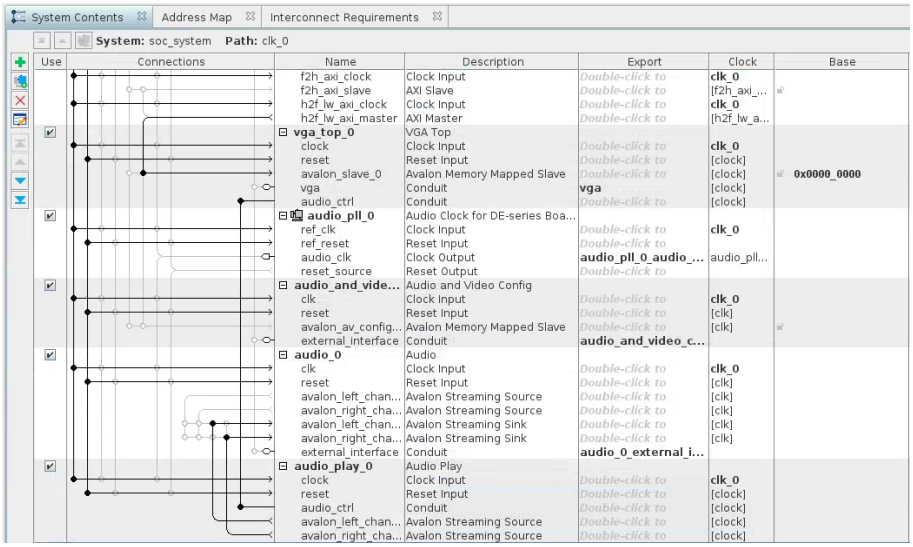

Figure 6: Qsys connections

the `audio_and_video_config` IP core configures the codec by specifying parameters such as data width, sampling frequency, and encoding format.

In *ForestFireIce*, four types of audio enhance the game's vividness, including background music and three sound effects when either character is dead, the fire boy jumps, or the water girl jumps. The software selects audio clip playback by writing the audio select bits in the `CONTROL_REG[31:29]`. The four pieces of sound are pre-loaded into BRAM as 8 kHz, 8-bit mono samples before being passed to a 4 kHz low-pass filter, and they are stored in a 2-port Audio Rom using single Memory Initialization File (MIF).

Under normal game play, both the left and right channels play the background music by setting the start and end of the `sound_address` to the corresponding addresses of the background music. When specific events occur, the left channel continues to play the background music, while the right channel plays the appropriate sound effects as selected via `CONTROL_REG[30:29]`. Once the sound effect finishes, the right channel resumes background music playback.

4 The Hardware/Software Interface

This section details the MMIO-based interface between the HPS software and the custom FPGA hardware. All communication between the two sides occurs through memory-mapped registers and shared RAM regions, exposed via an Avalon-MM slave interface mapped into the HPS address space. The interface allows software to control rendering, sprite positioning, and audio playback.

4.1 Register Map (Byte Addressed, 32-bit Wide)

The register map is shown in Table 1.

Offset	Register	Description	Bits	Value Range	R/W
0x00	`CTRL_REG`	Control register (tilemap index, audio ctrl)	[31:0]	See bit field below	W
0x04	`STATUS_REG`	Current pixel column and row	[19:0]	[19:10]: col (0–639) [9:0]: row (0–479)	R
0x08–0x7F	Reserved	Reserved for future use	–	–	–
0x80–0xFF	`SPRITE_ATTR_TABLE[n]`	Sprite attribute table (32 entries, 4 bytes each)	[31:0]	See format below	W

Table 1: Register Map

CTRL_REG Bit Field Description

The CTRL_REG Bit field description is shown in Table 2.

Bits	Name	Description
[1:0]	`tilemap_idx`	Tilemap index (2 bits): selects one of 4 tilemaps (0–3)
[28:2]	Reserved	Unused, reserved for future use
[30:29]	`sfx_sel`	Sound effect selector: 00 = None, 01/10/11 = 3 sound effects
[31]	`bgm_en`	Background music enable: 1 = On, 0 = Off

Table 2: CTRL_REG Bit Field Description

SPRITE_ATTR_TABLE Format (Each Entry = 4 Bytes)

The SPRITE_attr_table format is shown in Table 3. Each entry at offset: `0x80 + (n × 4)`, where $n \in [0, 31]$:

Bits	Field	Description
[31]	`enable`	1 = visible, 0 = hidden
[30]	`flip`	1 = horizontally flipped
[29:27]	Reserved	Unused
[26:18]	`sprite_y`	Vertical position (0–479)
[17:8]	`sprite_x`	Horizontal position (0–639)
[7:0]	`frame_id`	Sprite frame index (0–255)

Table 3: SPRITE_ATTR_TABLE Entry Format

4.2 Communication Protocol

- The HPS software updates the `SPRITE_ATTR_TABLE` at each VBlank period, typically once per frame.
- The software selects which tilemap and audio play to use by writing to `CONTROL_REG`.
- Hardware updates the `STATUS_REG` every clock cycle with real-time scan position.

4.3 Access Mechanism

- All registers and MMIO RAMs are exposed via an Avalon-MM slave peripheral. A corresponding Linux device driver will be implemented to provide structured access from user-space programs via standard interfaces such as `ioctl()` or `mmap()`, with the driver using `of_iomap()` internally to map device registers. Alternatively, `/dev/mem` may be used for low-level access during development.
- Software should align all 32-bit writes and reads to word boundaries and avoid partial byte writes.

4.4 Polling-Based Synchronization

- This system does not currently use interrupts for synchronization. Instead, the HPS software polls the `STATUS_REG` each frame to track VGA timing.
- Specifically, the software reads the `vcount` value from `STATUS_REG[19:10]`.
- While `0` $\leq$ `vcount` $<$ `480`, rendering is in progress; software prepares updated data (sprite attributes, control register).

- When `vcount` $\geq$ `480` (during VBlank), software performs writes to hardware (e.g., sprite attribute table, CONTROL_REG). This avoids visual tearing by ensuring all updates take effect at the start of the next frame.
- This polling-based synchronization is simple and effective for the game's frame-locked rendering loop.

5 Resource Budgets

The DE1-SoC platform provides limited on-chip FPGA resources, particularly in terms of BRAM (Block RAM), logic elements (LEs), and DSP blocks. This section summarizes the estimated consumption of these limited resources, based on the current system architecture.

5.1 On-Chip Memory (BRAM) Usage

Table 4 below provides a detailed breakdown of the estimated BRAM usage across major system components, including tilemaps, tile/sprite patterns, sprite attribute table, and audio data. This analysis ensures that our overall design remains well within the Cyclone V FPGA's 495 KB on-chip memory constraint, with sufficient margin for future extension.

Component	**Memory Type**	**Size Estimate**	**Description**
Tile Maps (x2)	BRAM (ROM)	2.4 KB	2 static tilemaps, each 40×30 tiles (1 byte per tile)
Tile Pattern	BRAM (ROM)	69 KB	138 tiles, each 16×16 pixels, 16-bit color
Sprite Pattern	BRAM (ROM)	33 KB	Multiple sprite animation frames, 16-bit color per pixel (Details see Table 5)
Sprite Attribute Table	BRAM (RAM)	128 B	32 sprite attribute entries, each 4 bytes
Audio Sample ROM	BRAM (ROM)	54 KB	$\leq$7 seconds of mono 8kHz 8-bit audio
MMIO Control Registers	Register	negligible	CONTROL_REG, STATUS_REG, etc.
Total		**~158 KB**	Well within ~495 KB BRAM budget of the Cyclone V FPGA

Table 4: Estimated On-Chip BRAM Usage

We adopted two methods to construct the tilemap, both relying on tile IDs and a tilemap grid to render the 40×30 screen. Each position in the grid stores a tile ID that specifies which tile to display. For the start and end screens, we used a Python script to automatically extract each tile from the full image and generate the corresponding tilemap. In contrast, for the main game screen, we manually designed each tile and assembled the tilemap using a CSV file. This manual process helped us avoid unnecessary increases in tile ID count caused by minor pixel differences, allowing us to use only 13 unique tile IDs in the main game screen. This significantly reduced memory consumption and made the implementation more suitable for the limited resources of the DE1-SoC platform. The two backgrounds are shown in Figure. 7 and Figure. 8.

Figure 7: Start Screen Tilemap

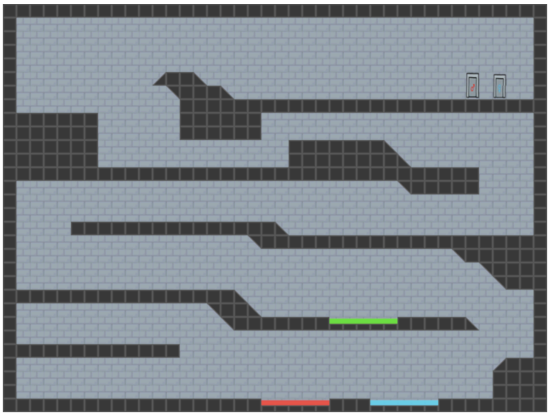

Figure 8: Main Game Tilemap

Table 5 below summarizes the estimated ROM usage for all sprites in the game, including characters, items, and interactive objects. Each frame pixel is stored using 8-bit color-ID to retrieve RGB888 from Color Palette.

Sprite Pattern	Frame Size(Bytes)	Frame Adress	Frames	Total Memory (Bytes)	Description
Fireboy head	16x16x2	0x0000–0x10FF	17	8704	stand(2), walk(5), drop(5), down(5)
Fireboy feet	16x16x2	0x1100–0x15FF	5	2560	stand(1), walk(3), drop & down(1)
Watergirl head	16x16x2	0x1600–0x26FF	17	8704	stand(2), walk(5), drop(5), down(5)
Watergirl feet	16x16x2	0x2700–0x2BFF	5	2560	stand(1), walk(3), drop & down(1)
Red Gem (x3)	16x16x2	0x2C00–0x2CFF	1	512	Collection, disappears after collision
Blue Gem (x3)	16x16x2	0x2D00–0x2DFF	1	512	Collection, disappears after collision
Yellow Button (x6)	16x16x2	0x2E00–0x2FFF	2	1024	Pressed/unpressed animation states
Yellow Lever (x2)	16x16x2	0x3000–0x32FF	3	1536	Toggleable lever: left/mid/right state
Yellow Elevator (x4)	16x16x2	0x3300–0x36FF	4	2048	Moving platform (up/down) animation
Purple Button (x2)	16x16x2	0x3700–0x38FF	2	1024	Pressed/unpressed animation states
Purple Elevator (x4)	16x16x2	0x3900–0x3CFF	4	2048	Moving platform (up/down) animation
Box (x4)	16x16x2	0x3D00–0x40FF	4	2048	Movable block used in puzzles
Total: 32	**16x16x2**	**0x0000–0x40FF**	**65**	**33280**	Total number

Table 5: Detailed Sprite Pattern Table with Memory Breakdown

The total ROM usage for all sprite patterns is approximately 32.5 KB, which is well within the 32 KB allocated in BRAM for the Sprite Pattern Table. This leaves ample room for future expansion, including additional animation frames or new sprite types. The details of the tiles and sprites are shown in Figure 9 and Figure 10.

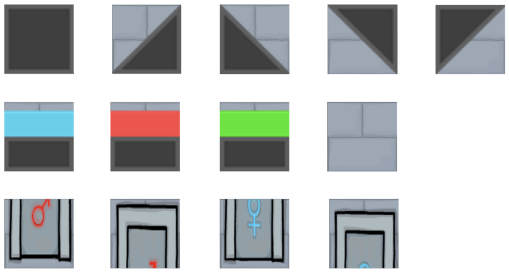

Figure 9: 13 tiles

Figure 10: 65 sprite patterns

5.2 DSP and Multiplier Resources

- The design avoids use of hardware multipliers or DSP blocks.
- All tile and sprite positioning is handled via simple integer arithmetic (division by powers of two and modulo).
- Audio playback is stream-based, with no filtering or mixing done in hardware.

5.3 External Memory and Bandwidth Considerations

- All visual and audio assets are fully preloaded into BRAM at boot time — there is **no runtime streaming from HPS DDR3**.
- Tilemaps and tile/sprite patterns are static during runtime; only sprite attribute table updates (128B per frame) are written via MMIO.
- Audio playback does not require data fetching from SD or DDR3 after boot.

6 Software

6.1 System Overview

The software system adopts a modular architecture with clear separation of responsibilities across four major components: input handling, game logic, rendering, and collision detection. Each module operates independently while communicating through well-defined interfaces, as shown in Figure 11. At the core of the system lies a frame-synchronized main loop implemented in `main.c`,

which governs the game's runtime behavior. The execution begins with a one-time initialization phase, during which hardware interfaces are configured, all game entities (players, boxes, gems) are created, and subsystems like input and sprite animation are initialized.

During each frame, the game performs the following sequence:

- **Frame sync:** Waits for the vertical scanline to reset (`row == 0`) to align logic with display timing;
- **Logic update:** Polls controller input, updates player physics and interactions, handles box movement, and checks gem collection;
- **Vertical blank wait:** Waits for the screen to enter the blanking interval before modifying VGA memory;
- **Rendering:** Writes active sprites (players, boxes, items) to hardware registers including their position, frame ID, and flip status.

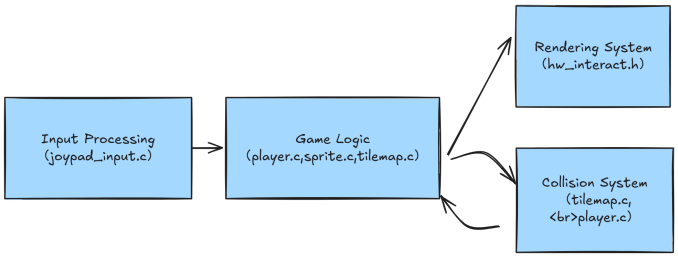

Figure 11: Software Architecture and Module Interaction

Input Handling The input system captures low-level controller events from Linux's `/dev/input/eventX` interface using non-blocking polling. It supports standard 8-button joypads with a directional pad and buttons A, B, X, Y, Start, and Select. Each device maintains its state using the `joypad_state_t` structure, which records button and axis values.

Events are categorized as either button presses (`EV_KEY`) or directional inputs (`EV_ABS`), and normalized to handle noise. The function `get_player_action()` converts raw input into symbolic actions like jump or move, prioritizing vertical actions over horizontal movement. The system supports hot-plugging and exposes both high-level and low-level APIs for integration with game logic.

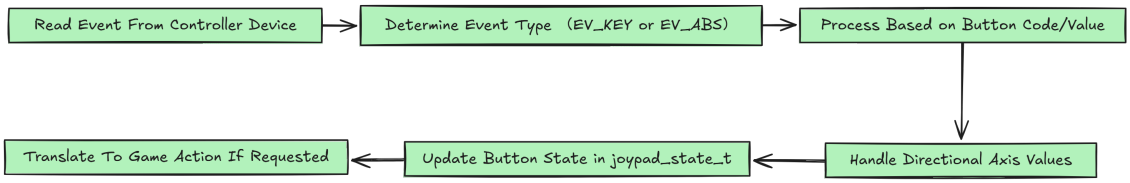

Figure 12: Controller input is processed and mapped into symbolic actions.

Game Coordination and State Management Besides input and rendering, the main loop centrally manages game-wide state. It coordinates updates across all subsystems, including physics integration, animation updates, object interactions, and eventual win condition detection. This tightly timed execution ensures smooth gameplay and flicker-free visual output at a stable frame rate.

6.2 Tilemap and Algorithms

6.2.1 Tilemap

The game map is stored in a 30×40 two-dimensional array named `tilemap[30][40]`, where each tile represents a 16×16 pixel region. Each tile code corresponds to a specific environmental element, defined as follows:

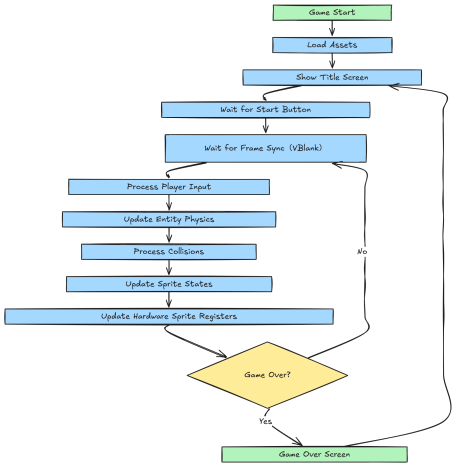

Figure 13: Main Loop Flow Diagram

Tile Code	Description
0	`TILE_EMPTY`: Empty space, walkable
1	`TILE_WALL`: Solid wall, blocks movement
2	`TILE_FIRE`: Fire pool, only Fireboy survives
3	`TILE_WATER`: Water pool, only Watergirl survives
4	`TILE_Green`: Green toxic pool, fatal to both characters
5	`TILE_SLOPE_L_UP`: Left-high to right-low slope (\)
6	`TILE_SLOPE_R_UP`: Right-high to left-low slope (/)
7	`TILE_CEIL_R`: Ceiling slope, left-high to right-low (\)
8	`TILE_CEIL_L`: Ceiling slope, left-low to right-high (/)
9	`TILE_GOAL`: Level goal tile, for WaterGirl
10	`TILE_GOAL2`: Level goal tile, for FireBpyz

The function `get_tile_at_pixel(x, y)` retrieves the tile code at a specific pixel location, while `is_tile_blocked(x, y, w, h)` determines whether an entity's rectangular region is obstructed by the tile environment. Special tiles such as slopes and slanted ceilings are further processed using internal tile-relative geometry, allowing for precise collision detection and surface alignment. This tilemap layer not only provides the visible terrain layout, but also acts as the backbone of the game's physics and interaction logic.

6.2.2 Tilemap System and Environmental Triggers

Beyond simple terrain rendering, the tilemap subsystem supports interactive logic and environmental awareness for gameplay objects. It forms the foundational layer for encoding game mechanics directly into level layouts.

Optimized Tile Sampling To ensure performance, the engine avoids exhaustive tile checks. Instead, for each moving object (e.g., player, box), a bounding rectangle is computed and only the tiles within that area are sampled. This reduces collision tile lookups from potentially thousands per frame to just 4–6, depending on the object size and speed.

Entities are updated in priority order—typically: **player** → **box** → **item**—to ensure logical consistency and minimal perceived input delay.

Tile-Based Interaction Logic The tilemap encodes interactive semantics beyond geometry:

- **Spawn Alignment:** Objects and players are aligned to the tile grid upon initialization.
- **Hazard Detection:** Tiles can trigger damage or instant death based on character type (e.g., fire, water).
- **Goal Tiles:** Detect when both characters reach target tiles to advance levels.

- **Trigger Zones:** Pressure plates and switches are implemented as tile states that can alter the environment (e.g., activate elevators).

These features allow level designers to express game logic declaratively by placing special tiles, creating rich environmental mechanics without additional scripting.

6.3 Collision Detection and Physical Interaction

A core function of the tilemap is to support precise and robust collision detection across complex terrain. Our engine implements a multi-stage system that combines predictive testing, positional correction, and asymmetric hitbox logic to ensure responsive and believable character-environment interaction.

6.3.1 Predictive AABB Collision Checking

The foundation of our collision system is axis-aligned bounding box (AABB) testing. Instead of waiting for an object to penetrate a wall or floor tile, we compute its next position (`next_x`, `next_y`) in advance and test for blockage. This predictive approach prevents characters from embedding into terrain or solid objects.

If a collision is detected along the updated axis, we cancel that component of the position update and set the corresponding velocity to zero. For example, if `new_y` would intersect the floor, the character's vertical velocity `vy` is zeroed, and their `y` coordinate is left unchanged. This ensures consistent physics behavior and guards against tunneling.

6.3.2 Positional Correction via Platform and Slope Alignment

To address the instability caused by floating-point inaccuracies during landing, the engine employs positional correction techniques that forcibly align characters to ground surfaces when near contact is detected.

Platform Snapping (Basic Version) When a character is falling and is about to land on a flat platform, we use:

```
adjust_to_platform_y(player_t* p)
```

This function scans a small vertical region below the character's feet and, if a platform tile is detected, it takes over control of the `y` coordinate, snapping the character precisely to the top of the tile. If the offset is small (within a defined threshold), vertical velocity is preserved; otherwise, it is zeroed. This prevents the character from being repeatedly marked as airborne by gravity and re-grounded by collision, which would otherwise result in animation jitter between falling and idle states.

Slope Locking and Y-Alignment (Advanced Version) A more advanced version of this correction mechanism is used for navigating sloped tiles. The function:

```
adjust_to_slope_y(player_t* p)
```

computes a pixel-accurate vertical position based on the horizontal position within the slope tile. The character's foot position is adjusted accordingly, allowing for seamless ascent and descent over sloped terrain.

Slope alignment presents two additional challenges:

- **Uphill Snapping:** When climbing a slope, simply adding horizontal velocity may cause the character to collide with an adjacent wall tile, especially if their `y` position lags behind. By updating `y` based on `x` (rather than waiting for downward collision), we ensure the character is lifted along the slope gradient smoothly and avoids premature wall collision.
- **Stable Slope State:** On small inclines, constant transitions between slightly falling and slightly grounded can cause the animation state machine to oscillate between `STATE_FALLING` and `STATE_RUNNING`. By snapping tightly to the slope surface and suppressing minor `vy` perturbations, the character maintains a continuous motion state. The slope essentially "grabs" the player and prevents unnecessary toggling or freezing at edge cases.

6.3.3 Asymmetric and Contextual Hitbox Design

Each dynamic entity—such as players, boxes, and platforms—employs separate, independently defined collision boxes. We refer to this as *asymmetric contextual hitboxing*. Each entity maintains one or more bounding regions that are selectively used depending on the interaction type and direction.

For example, character-vs-character collisions use one set of boxes, while box-vs-wall or player-vs-box interactions use another. This decoupling serves two critical purposes:

- **Animation-Conscious Collision:** Hitboxes can be tightly fitted to match the current animation frame, resulting in more faithful visual and logical overlap.
- **Deadlock Recovery:** Certain puzzles may result in objects (e.g., a box) being pushed into tight corners. By using smaller hitboxes for the player, such as Fireboy's narrow bounding profile, the player can squeeze through tile gaps and apply a pushing force from the opposite side. Notably, the collision box used between player and box is smaller than that between box and wall, creating a usable escape route.

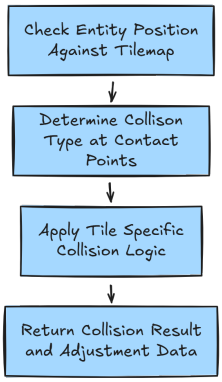

Figure 14: Collision Detection Flow Diagram

Collision Resolution Functions To handle terrain, dynamic objects, and special platforms, the engine uses three key predicate functions:

```
bool is_tile_blocked(int x, int y);
bool is_box_blocked(int x, int y);
bool is_elevator_blocked(int x, int y);
```

- `is_tile_blocked()` detects walls and solid tiles and is used in AABB-style collision checks.
- `is_box_blocked()` detects collisions with movable objects such as boxes.
- `is_elevator_blocked()` checks whether the character intersects with a moving platform and retrieves platform velocity if needed.

Each frame, these functions are used to predict collisions before updating the position. Combined with context-specific hitboxes (described previously), they ensure smooth interaction with static and dynamic world elements.

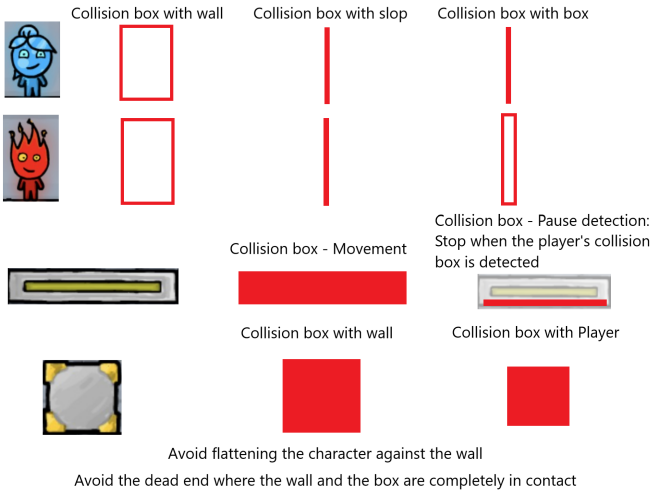

Figure 15: Hitbox Visualization Examples

6.4 Player and Sprite Logic

This module governs the behavior, movement, and visual representation of the main characters—*Fireboy* and *Watergirl*. It integrates physical simulation, collision processing, state transitions, and synchronized animation.

Character Data Model Each player is encapsulated in a `player_t` structure that stores floating-point positions, velocity vectors, character type, current state, and animation control variables. The use of floating-point arithmetic enables smooth sub-pixel movement, while the state variables govern how physical and visual behavior evolve over time.

Input and Movement Processing Player input is handled through the `player_handle_input()` function, which interprets button signals into movement and jumping commands. Horizontal movement is applied at a fixed velocity, while jumping triggers a vertical impulse if the character is grounded. In mid-air, players retain limited horizontal control. This input stage decouples control intent from physical simulation, ensuring consistent responses even under varying frame rates.

Physics and Collision Logic Movement is governed by the `player_update_physics()` function as mentioned early, which applies gravity, predicts next-frame positions, and tests for potential collisions before committing updates. The system performs axis-aligned bounding box (AABB) collision checks against tiles, boxes, and moving platforms. If a collision is predicted, the corresponding velocity component is zeroed and the position update is suppressed.

To avoid jitter near the ground, the system uses correction functions:

- `adjust_to_platform_y()` is used to snap players cleanly onto flat platforms when falling.
- `adjust_to_slope_y()` provides sub-tile vertical correction based on horizontal offset, enabling smooth traversal of inclined tiles.

These ensure that the character does not rapidly toggle between "falling" and "grounded" states when hovering near terrain edges.

State Machine and Animation Control The character behavior is driven by a four-state finite state machine: `IDLE`, `RUNNING`, `JUMPING`, and `FALLING`. These states are inferred from input and physics outcomes—for example, leaving the ground triggers a transition to `FALLING`, while standing still reverts to `IDLE`. This model ensures coherent animation control and smooth interaction transitions.

Dual-Sprite Animation System Each character is composed of two layered sprites: `upper_sprite` for the head and `lower_sprite` for the legs. These are animated independently but remain synchronized via shared timers and physical state. Different frame sets are defined for Fireboy and Watergirl using symbolic constants (e.g., `FB_HEAD_WALK`, `WG_LEG_JUMP`), enabling distinct character styles.

Sprite flipping is handled based on movement direction, ensuring the character always faces the intended direction. The modular dual-sprite design improves expressiveness while reducing sprite resource requirements.

Other Animated Entities The same framework is used to animate other game objects:

- **Boxes:** Built from four connected sprites, they align to the tile grid and respond to player pushing mechanics.
- **Diamonds:** Red and blue collectible items that use sine-wave vertical animation and disappear upon character contact.
- **Hazards and Goals:** Pools trigger death events based on player type, while doors complete the level when both characters arrive.

All these entities are updated via the same sprite management pipeline, maintaining visual consistency across the game.

6.5 Sprite Rendering and VGA Interface

The `sprite.c` module manages all visual elements in the game, providing a unified abstraction layer over VGA hardware. It supports per-frame animation, visibility control, and optimized communication with the VGA controller.

Sprite Representation Each sprite contains position, frame index, visibility flags, and horizontal/vertical flip states. These attributes are updated per frame based on physical state and game logic. The sprite system uses symbolic indices to refer to sprite slots, which implicitly define rendering order.

Frame Synchronization and Update Batching To prevent visual artifacts such as tearing, sprite updates are deferred until the vertical blanking interval (VBlank), detected via the scanline position. Updated sprites are batched using the `write_sprite()` function, which issues hardware-specific commands with minimal overhead. Unchanged sprites are not re-issued, conserving bandwidth and CPU cycles.

Z-order and Layered Rendering Visual stacking is achieved by allocating sprite indices according to function:

- Low indices: background and environmental tiles
- Mid indices: boxes, buttons, collectibles
- High indices: player characters and foreground objects

This ensures correct occlusion and predictable rendering order without needing a dedicated z-buffer.

Entity-Specific Rendering Logic Several entity types employ customized rendering strategies:

- **Buttons:** Change sprite frame when pressed
- **Elevators:** Move smoothly by updating their `y` coordinate each frame
- **Boxes:** Rendered as 2x2 sprite blocks, synchronized with physics motion
- **Diamonds:** Use sine-based vertical offsets for floating effect

Hardware Abstraction and Extensibility The abstraction layer in `sprite.c` allows all game logic to manipulate sprites without knowledge of VGA timing and sprite memory layout. This enables future extension, such as animated backgrounds or HUD overlays, without reworking the rendering infrastructure.

7 Task Allocation

- Yonghao Lin: Built a hardware VGA display system (VGA top, sprite engine, tile engine, line buffer) and wrote the device driver for HW/SW communication. Coordinated with the audio part to reuse the existing driver.
- Yang Cao: Main software components, including logic and function of character and sprite, collision handling, slope and flat surface correction mechanisms, and hitbox design. Collaborated on input handling and hardware I/O.
- Zhenqi Li: In charge of the background tile map, sprite graphics, image processing, generated mif files, and efficient storage. Implemented game logic, including game-over, death conditions, and general control, using the tilemap system.
- Yifan Mao: Primarily responsible for input design and character behavior design, and collaborated on implementing the game logic.
- Weijie Wang: Implemented the audio part, including connections on QSYS, merging the background music and sound effects, and converting them to a Memorize Initialization File.

8 Appendix

A: Hardware

```
module vga_top(input logic         clk,
                    input logic      reset,
                    input logic [31:0]  writedata,
                    input logic      write,
                    input            chipselect,
                    input logic [5:0]  address, // 64

                    output logic [31:0] readdata,
                    output logic [7:0] VGA_R, VGA_G, VGA_B,
                    output logic         VGA_CLK, VGA_HS, VGA_VS,
                    VGA_BLANK_n,
                    output logic         VGA_SYNC_n,

                    output logic [2:0] audio_ctrl);

    // current VGA pixel coord
    logic [10:0]        hcount;
    logic [9:0]      vcount;

    logic [31:0] status_reg;
    logic [31:0] ctrl_reg;

    assign status_reg[19:0] = {hcount[10:1], vcount};
    // linebuffer
    // addr
    logic [5:0] addr_tile_disp;
    logic [9:0] addr_pixel_disp;
    logic [5:0] addr_tile_draw;
    logic [9:0] addr_pixel_draw;

    // indata
    logic [255:0] data_tile_disp;
    logic [15:0]  data_pixel_disp;
    logic [255:0] data_tile_draw;
    logic [15:0]  data_pixel_draw;

    // wren
    logic wren_tile_disp;
    logic wren_pixel_disp;
    logic wren_tile_draw;
    logic wren_pixel_draw;
```

```
42
43      // outdata
44      logic [255:0] q_tile_disp;
45      logic [15:0]  q_pixel_disp;
46      logic [255:0] q_tile_draw;
47      logic [15:0]  q_pixel_draw;
48
49      logic switch;
50
51      linebuffer u_linebuffer (.*);
52
53      // connection between tile_engine and linebuffer
54      assign addr_tile_draw = tile_col;
55      assign data_tile_draw = tile_data;
56
57      // tile engine
58      logic tile_start;
59      // output declaration of module tile_engine
60      logic[5:0] tile_col;
61      logic [255:0] tile_data;
62      logic tile_done;
63
64      tile_engine u_tile_engine(
65          .clk            (clk          ),
66          .reset          (reset        ),
67          .tile_start     (tile_start   ),
68          .tilemap_idx    (ctrl_reg[1:0]  ),
69          .vcount         (vcount       ),
70          .tile_col       (tile_col     ),
71          .tile_data      (tile_data    ),
72          .tile_done      (tile_done    ),
73          .wren_tile_draw (wren_tile_draw)
74      );
75
76
77      // sprite engine
78      logic sprite_start;
79      logic sprite_done;
80
81      logic sprite_write_reg;
82      logic [4:0] sprite_wr_idx;
83      logic [31:0] sprite_writedata;
84
85      always_ff @(posedge clk) begin
86          if (reset) begin
87              sprite_write_reg <= 0;
88              sprite_wr_idx    <= 0;
89              sprite_writedata <= 0;
90          end else begin
91          // latch data to keep stable
92          if (chipselect && write && address[5]) begin
93              sprite_write_reg <= 1;
94              sprite_wr_idx    <= address[4:0];
95              sprite_writedata <= writedata;
96          end else begin
97                  sprite_write_reg <= 0;
98              end
99          end
100     end
101     sprite_engine u_sprite_engine(
102         .clk            (clk          ),
103         .reset          (reset        ),
104         .sprite_start   (sprite_start   ),
105         .vcount         (vcount       ),
106         .spr_wr_en      (sprite_write_reg     ),
107         .spr_wr_idx     (sprite_wr_idx    ),
```

```
108         .spr_wr_data    (sprite_writedata    ),
109         .sprite_pixel_col (addr_pixel_draw),
110         .sprite_pixel_data (data_pixel_draw),
111         .wren_pixel_draw (wren_pixel_draw),
112         .done (sprite_done)
113     );
114
115
116
117     vga_counters counters(.clk50(clk), .*);
118
119     always_ff @(posedge clk) begin
120         if (reset) begin
121             // status_reg <= 0;
122             ctrl_reg <= 0;
123             tile_start <= 0;
124             sprite_start <= 0;
125             wren_tile_disp <= 0;
126             wren_pixel_disp <= 0;
127
128             switch <= 0;
129
130             audio_ctrl <= 0;
131         end
132         else begin
133             if (vcount < 479 || vcount == 524) begin
134                 if (hcount == 0) begin
135                     tile_start <= 1;
136                 end else begin
137                     tile_start <= 0;
138                 end
139                 // sprite start
140                 // 60 clk enough to draw tile
141                 if (hcount == 60 && tile_done && sprite_done) begin
142                     sprite_start <= 1;
143                 end
144
145                 if (sprite_start) begin // 1 cycle pulse
146                     sprite_start <= 0;
147                 end
148
149                 // 1 cycle flip "switch", 1 cycle read "switch" to "
                        disp_sel",1 cycle read memory
150                 // more cycles to insure robust
151                 if (hcount == 1590 && tile_done && sprite_done)
152                     switch <= ~switch;
153             end
154
155             if (chipselect) begin
156                 if (write) begin
157                     case (address)
158                         6'h0: begin
159                             ctrl_reg <= writedata;
160                             // audio part
161                             audio_ctrl <= writedata[31:29];
162                         end
163                     endcase
164                 end
165                 else begin // read
166                     case (address)
167                         6'h1: readdata <= status_reg;
168                     endcase
169                 end
170             end
171
172         end
```

```
173     end
174
175     // 1 cycle delay
176     assign addr_pixel_disp = hcount[10:1] < 639 ? hcount[10:1] + 1 : 0;
177     // MSB is transparency bit ( 1 = transparent, 0 = no)
178     assign VGA_R = q_pixel_disp[14:10] << 3;
179     assign VGA_G = q_pixel_disp[9:5] << 3;
180     assign VGA_B = q_pixel_disp[4:0] << 3;
181
182
183 endmodule
184
185 module vga_counters(
186         input logic          clk50, reset,
187         output logic [10:0] hcount,  // hcount[10:1] is pixel column
188         output logic [9:0]  vcount,  // vcount[9:0] is pixel row
189         output logic         VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n,
            VGA_SYNC_n);
190
191     /*
192      * 640 X 480 VGA timing for a 50 MHz clock: one pixel every other cycle
193      *
194      * HCOUNT 1599 0             1279       1599 0
195      *             ---------------              --------
196      * ____________|    Video      |____________|  Video
197      *
198      *
199      * |SYNC| BP |<-- HACTIVE -->|FP|SYNC| BP |<-- HACTIVE
200      *       ------------------------     -------------
201      * |____|       VGA_HS          |____|
202      */
203     // Parameters for hcount
204     parameter HACTIVE      = 11'd 1280,
205               HFRONT_PORCH = 11'd 32,
206               HSYNC        = 11'd 192,
207               HBACK_PORCH  = 11'd 96,
208               HTOTAL       = HACTIVE + HFRONT_PORCH + HSYNC +
209               HBACK_PORCH; // 1600
210
211     // Parameters for vcount
212     parameter VACTIVE      = 10'd 480,
213               VFRONT_PORCH = 10'd 10,
214               VSYNC        = 10'd 2,
215               VBACK_PORCH  = 10'd 33,
216               VTOTAL       = VACTIVE + VFRONT_PORCH + VSYNC +
217               VBACK_PORCH; // 525
218
219     logic endOfLine;
220
221     always_ff @(posedge clk50 or posedge reset)
222         if (reset)
223             hcount <= 0;
224         else if (endOfLine)
225             hcount <= 0;
226         else
227             hcount <= hcount + 11'd 1;
228
229     assign endOfLine = hcount == HTOTAL - 1;
230
231     logic endOfField;
232
233     always_ff @(posedge clk50 or posedge reset)
234         if (reset)
235             vcount <= 0;
236         else if (endOfLine)
237             if (endOfField)
```

```
                vcount <= 0;
            else
                vcount <= vcount + 10'd 1;

    assign endOfField = vcount == VTOTAL - 1;

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

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

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

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

endmodule
```

Listing 1: vga_top.sv

```
module linebuffer(
    input logic clk,
    input logic reset,
    input logic switch,
    input logic [5:0] addr_tile_disp,
    input logic [9:0] addr_pixel_disp,
    input logic [5:0] addr_tile_draw,
    input logic [9:0] addr_pixel_draw,

    // indata
    input logic [255:0] data_tile_disp,
    input logic [15:0]  data_pixel_disp,
    input logic [255:0] data_tile_draw,
    input logic [15:0]  data_pixel_draw,

    // wren
    input logic wren_tile_disp,
    input logic wren_pixel_disp,
    input logic wren_tile_draw,
    input logic wren_pixel_draw,

    // outdata
    output logic [255:0] q_tile_disp,
    output logic [15:0]  q_pixel_disp,
    output logic [255:0] q_tile_draw,
    output logic [15:0]  q_pixel_draw
);

    logic [5:0] addr_tile[1:0];
    logic [9:0] addr_pixel[1:0];

    logic [255:0] data_tile[1:0];
```

```
    logic [15:0]  data_pixel[1:0];

    logic wren_tile[1:0];
    logic wren_pixel[1:0];

    logic [255:0] q_tile[1:0];
    logic [15:0]  q_pixel[1:0];

    linebuffer_ram linebuffer_ram0(
        .address_a  (addr_tile[0] ),
        .address_b  (addr_pixel[0] ),
        .clock      (clk      ),
        .data_a     (data_tile[0] ),
        .data_b     (data_pixel[0] ),
        .wren_a     (wren_tile[0]  ),
        .wren_b     (wren_pixel[0] ),
        .q_a        (q_tile[0]     ),
        .q_b        (q_pixel[0]     )
    );

    linebuffer_ram linebuffer_ram1(
        .address_a  (addr_tile[1] ),
        .address_b  (addr_pixel[1] ),
        .clock      (clk      ),
        .data_a     (data_tile[1] ),
        .data_b     (data_pixel[1] ),
        .wren_a     (wren_tile[1]  ),
        .wren_b     (wren_pixel[1] ),
        .q_a        (q_tile[1]     ),
        .q_b        (q_pixel[1]     )
    );
    logic disp_sel;

    always_ff @(posedge clk) begin
        if (reset) begin
            disp_sel <= 0;
        end else begin
            disp_sel <= switch;
        end
    end

    always_comb begin
        if (disp_sel) begin
            // RAM0 disp, RAM1 draw
            addr_tile[0] = addr_tile_disp;
            addr_pixel[0] = addr_pixel_disp;
            data_tile[0] = data_tile_disp;
            data_pixel[0] = data_pixel_disp;
            wren_tile[0] = wren_tile_disp;
            wren_pixel[0] = wren_pixel_disp;
            q_tile_disp =  q_tile[0];
            q_pixel_disp = q_pixel[0];

            addr_tile[1] = addr_tile_draw;
            addr_pixel[1] = addr_pixel_draw;
            data_tile[1] = data_tile_draw;
            data_pixel[1] = data_pixel_draw;
            wren_tile[1] = wren_tile_draw;
            wren_pixel[1] = wren_pixel_draw;
            q_tile_draw =  q_tile[1];
            q_pixel_draw = q_pixel[1];
        end else begin
            // RAM0 draw, RAM1 disp
            addr_tile[1] = addr_tile_disp;
            addr_pixel[1] = addr_pixel_disp;
            data_tile[1] = data_tile_disp;
```

```
            data_pixel[1] = data_pixel_disp;
            wren_tile[1] = wren_tile_disp;
            wren_pixel[1] = wren_pixel_disp;
            q_tile_disp =  q_tile[1];
            q_pixel_disp = q_pixel[1];

            addr_tile[0] = addr_tile_draw;
            addr_pixel[0] = addr_pixel_draw;
            data_tile[0] = data_tile_draw;
            data_pixel[0] = data_pixel_draw;
            wren_tile[0] = wren_tile_draw;
            wren_pixel[0] = wren_pixel_draw;
            q_tile_draw =  q_tile[0];
            q_pixel_draw = q_pixel[0];
        end
    end

endmodule
```

Listing 2: linebuffer.sv

```
module tile_engine(
    input logic clk,
    input logic reset,
    input logic tile_start,
    input logic [1:0] tilemap_idx,
    input logic [9:0] vcount,
    output logic [5:0] tile_col,
    output logic [255:0] tile_data,
    output logic tile_done,
    output logic wren_tile_draw
);

// internal
logic [11:0] tilemap_addr;
logic [11:0] tile_pattern_addr;


logic [7:0] tile_id;
logic [9:0] next_vcount;
logic [5:0] col[2:0];
assign tile_col = col[2];

assign wren_tile_draw = (!tile_done) && (col[0] > 0);

assign tilemap_addr = tilemap_idx * 1200 +  (next_vcount >> 4) * 40 + col
    [0];
tilemap u_tilemap(
    .address    (tilemap_addr  ),
    .clock      (clk),
    .q          (tile_id)
);

assign tile_pattern_addr = (tile_id << 4) + next_vcount[3:0];
tile_pattern u_tile_pattern(
    .address    (tile_pattern_addr  ),
    .clock      (clk),
    .q          (tile_data)
);

always_ff @(posedge clk) begin
    if (reset) begin
        col[0] <= 0;
        col[1] <= 0;
        col[2] <= 0;
        tile_done <= 1;
```

```
    end else begin
        if (tile_start) begin
            col[0] <= 0;
            col[1] <= 0;
            col[2] <= 0;
            if (vcount < 479) begin
                next_vcount <= vcount + 1;
                tile_done <= 0;
            end else if (vcount >= 479 && vcount < 524) begin
                tile_done <= 1;
            end else if (vcount == 524) begin
                next_vcount <= 0;
                tile_done <= 0;
            end
        end else if (!tile_done) begin
            col[1] <= col[0];
            col[2] <= col[1];
            if (col[0] < 39) begin
                col[0] <= col[0] + 1;
            end
            if (col[2] == 39) begin
                tile_done <= 1;
            end
        end
    end
end

endmodule
```

Listing 3: tile_engine.sv

```
/* sprite_attr
[31] : Enable = 1, Disable = 0
[30] : Flip = 1, otherise = 0
[29: 27]: Reserved
[26:18] sprite_pos_row (0-479)
[17:8] : sprite_pos_col (0-639)
[7:0] : frame_id (0-255)
*/

// Test example: 1000 0100 0000 0001 0000 0000 0000 0001
//               8      4    0    1       0    0    0    1
module sprite_engine #(
    parameter NUM_SPRITE = 32,
    parameter MAX_SLOT   = 8
)(
    input  logic        clk,
    input  logic        reset,

    input  logic        sprite_start,

    input  logic [9:0]  vcount,

    input  logic                              spr_wr_en,
    input  logic [$clog2(NUM_SPRITE)-1:0]     spr_wr_idx,
    input  logic [31:0]                       spr_wr_data,

    output logic [9:0]  sprite_pixel_col,
    output logic [15:0] sprite_pixel_data,
    output logic        wren_pixel_draw,
    // debug
    // input logic [4:0] debug_addr,
    // output logic [31:0] debug_data,

    output logic        done
);
```

```
36      logic [9:0] next_vcount;
37      assign next_vcount = (vcount < 10'd479) ? vcount + 10'd1 :
38                           (vcount == 10'd524) ? 10'd0      : vcount + 1;
39
40      logic [31:0] attr_rd;
41      logic [$clog2(NUM_SPRITE)-1:0] attr_ra;
42
43      sprite_attr_ram u_ram(
44          .clock (clk),
45          .data (spr_wr_data),
46          .rdaddress (attr_ra),
47          .wraddress (spr_wr_idx),
48          .wren (spr_wr_en),
49          .q(attr_rd) );
50
51      // FE
52      logic fe_draw_req, fe_flip, fe_done;
53      logic dw_done;
54      logic [9:0] fe_col;
55      logic [7:0] fe_frame;
56      logic [3:0] fe_rowoff;
57
58      sprite_frontend #(
59          .NUM_SPRITE (NUM_SPRITE),
60          .MAX_SLOT   (MAX_SLOT)
61      ) u_fe (
62          .clk        (clk),
63          .reset      (reset),
64          .start_row  (sprite_start),
65          .next_vcount(next_vcount),
66          .ra         (attr_ra),
67          .rd_data    (attr_rd),
68          .draw_done (dw_done),
69          .draw_req   (fe_draw_req),
70          .col_base   (fe_col),
71          .flip       (fe_flip),
72          .frame_id   (fe_frame),
73          .row_off    (fe_rowoff),
74          .fe_done    (fe_done)
75      );
76
77      // ------------------- ROM ----------------------------------------
78      logic [15:0] rom_addr, rom_q;
79      sprite_pattern_rom u_rom (.clock(clk), .address(rom_addr), .q(rom_q));
80
81      // ------------------- Drawer -------------------------------------
82      sprite_drawer u_dw (
83          .clk       (clk),
84          .reset     (reset),
85          .start     (fe_draw_req),
86          .col_base  (fe_col),
87          .flip      (fe_flip),
88          .frame_id  (fe_frame),
89          .row_off   (fe_rowoff),
90          .rom_addr  (rom_addr),
91          .rom_q     (rom_q),
92          .pixel_col (sprite_pixel_col),
93          .pixel_data(sprite_pixel_data),
94          .wren      (wren_pixel_draw),
95          .done      (dw_done)
96      );
97
98      assign done = (fe_done) || (vcount >= 479 && vcount < 524);
99
100 endmodule
```

Listing 4: sprite_engine.sv

```
module sprite_frontend #(
    parameter NUM_SPRITE = 32,
    parameter MAX_SLOT   = 8
)(
    input  logic            clk,
    input  logic            reset,

    input  logic            start_row,
    input  logic [9:0]      next_vcount,

    output logic [$clog2(NUM_SPRITE)-1:0] ra,
    input  logic [31:0]     rd_data,

    input  logic            draw_done,          // drawer: 1  idle   0
        busy
    output logic            draw_req,
    output logic [9:0]      col_base,
    output logic            flip,
    output logic [7:0]      frame_id,
    output logic [3:0]      row_off,

    output logic            fe_done
);

    localparam int IDXW  = $clog2(NUM_SPRITE);
    localparam int QPW   = $clog2(MAX_SLOT);
    localparam int MASK  = MAX_SLOT - 1;

    logic [IDXW-1:0] scan_idx;
    logic [IDXW-1:0] scan_idx_d;
    logic [IDXW:0] scan_idx_checked; // NUM_SPRITE
    assign ra = scan_idx;

    logic hit;
    assign hit = rd_data[31] && (next_vcount >= {1'b0,rd_data[26:18]}) && (
        next_vcount < rd_data[26:18]+16);
    // FIFO
    typedef struct packed { logic [9:0] col; logic flip; logic [7:0] frame;
        logic [3:0] rowoff; } ent_t;
    ent_t           fifo [MAX_SLOT];
    logic [QPW-1:0] head, tail;
    logic [QPW-1:0]   cnt;

    logic drawing;
    logic enqueue;
    assign enqueue = hit && cnt < MAX_SLOT - 1 && ({1'b0,scan_idx_d} !=
        scan_idx_checked) && scan_idx > 0;

    logic dequeue;
    assign dequeue = !drawing && cnt != 0;

    always_ff @(posedge clk) begin
        //---------------------------------------------- reset / blank
        if (reset) begin
            scan_idx <= 0;
            scan_idx_d <= 0;
            scan_idx_checked <= NUM_SPRITE;
            head<=0;
            tail<=0;
            cnt <= 0;
            draw_req<=0;
            drawing <= 0;
            fe_done<=1;
```

```
60          end
61          else if (start_row) begin
62              scan_idx <= 0;
63              scan_idx_d <= 0;
64              scan_idx_checked <= NUM_SPRITE;
65              head<=0;
66              tail<=0;
67              cnt <= 0;
68              draw_req<=0;
69              drawing <= 0;
70              if (next_vcount < 10'd480) begin
71                  fe_done <= 0;       // visible line
72              end else begin
73                  fe_done <= 1; // blank
74              end
75          end
76          //----------------------------- normal run
77          else if (!fe_done) begin
78              draw_req <= 0;
79              cnt <= cnt + enqueue - dequeue;
80
81              scan_idx_d <= scan_idx;
82              /*
83              Why we need "-2": Important, because if we just -1, scan_idx
                    will become 9 and keep this value, scan_idx_d become 8 for
                    1 clk,
84              then after 1clk, scan_idx_d become 9, so we lost 8(scan_idx_d)'
                    s value, it becomes 9's value, waiting for enqueue.
85              */
86              if(scan_idx < NUM_SPRITE - 1 && cnt < MAX_SLOT - 2) begin
87                  scan_idx <= scan_idx + 1'b1;
88              end
89              // Enqueue
90              if (enqueue) begin
91                  scan_idx_checked <= {1'b0,scan_idx_d};
92                  fifo[tail].col <= rd_data[17:8];
93                  fifo[tail].flip <= rd_data[30];
94                  fifo[tail].frame <=rd_data[7:0];
95                  fifo[tail].rowoff <= (next_vcount-rd_data[26:18]) & 4'hf;
96                  tail <= (tail + 1'b1) & MASK;
97              end
98
99              // drawer idle and queue non empty     Dequeue
100             if (dequeue) begin
101                 col_base <= fifo[head].col;
102                 flip     <= fifo[head].flip;
103                 frame_id <= fifo[head].frame;
104                 row_off  <= fifo[head].rowoff;
105                 // 1 clk pulse
106                 draw_req <= 1;
107
108                 // drawing state long period
109                 drawing <= 1;
110                 head <= (head + 1'b1) & MASK;
111             end
112
113             if(drawing) begin
114                 // draw_req <= 0;
115                 // Warning!!: should be very careful, because "after"
                        draw_req 1 clk pulse, draw_done set to 0.
116                 // So we can not quickly check draw_done, since it is "1"
                        when draw_req = 1
117                 // After draw_req drop to 0(lasting 1 clk), draw_done set
                        to 0. Then we can check draw_done to detect "drawing"
                        state.
118                 if (draw_done && !draw_req)
```

```
                    drawing <= 0;
            end

            // This line is done.
            // Why we need "!drawing" instead of "draw_done", because after the last one send to drawer,
            // it will fe_done <=1 next cycle due to draw_done still 1(but it will drop to 0 next cycle)

            // If only one sprite need to send to drawer
            // And it is scan_idx_d = NUM_SPRITE-1, because drawing need 1 clk to pull up,
            // So when this sprite enqueue, at the same cycle, cnt is still 0, then it will trigger fe_done <=1
            // So we will miss this sprite.
            if (scan_idx_d == NUM_SPRITE-1 && cnt==0 && !drawing && !enqueue)
                fe_done <= 1;
        end
    end
endmodule
```

Listing 5: sprite_frontend.sv

```
module sprite_drawer (
    input  logic        clk,
    input  logic        reset,

    input  logic        start,
    input  logic [9:0]  col_base,
    input  logic        flip,
    input  logic [7:0]  frame_id,
    input  logic [3:0]  row_off,

    // ROM
    output logic [15:0] rom_addr,
    input  logic [15:0] rom_q,

    output logic [9:0]  pixel_col,
    output logic [15:0] pixel_data,
    output logic        wren,
    output logic        done
);

    logic [3:0] idx;          // 0  15
    logic [3:0] idx_d;
    logic       valid_d;

    always_ff @(posedge clk) begin
        if (reset) begin
            done  <= 1;
            idx   <= 0;
            idx_d <= 0;
            valid_d <= 0;
            rom_addr <= 0;
            wren <= 0;
        end else begin
            if (start) begin
                done  <= 0;
                valid_d <= 0;
                idx <= 0;
                idx_d <= 0; // FxxKKK, I forgot to reset it, it let me struggle for a long time.
                rom_addr <= {frame_id, row_off, 4'b0};
                wren <= 0;
            end
```

```
            else begin
                // Output
                pixel_col  <= flip ? (col_base + (10'd15 - {6'b0, idx_d}))
                    : (col_base + {6'b0, idx_d});
                pixel_data <= rom_q;
                wren <= (!done) && (valid_d) && (rom_q[15] == 1'b0) && (
                    pixel_col <= 639);
                if (!done) begin
                    idx_d   <= idx;
                    if (idx < 15) begin
                        rom_addr <= rom_addr + 16'd1;
                        idx      <= idx + 1;
                        valid_d  <= 1;
                    end

                    if (idx_d == 15) begin
                        done <= 1'b1;
                        valid_d <= 1'b0;
                    end
                end
            end
        end
    end
endmodule
```

Listing 6: sprite_drawer.sv

```
`define BGM_BEGIN 16'h0
`define BGM_END 16'hAF8A
/*
  bgm.mp3: 0x0 to 0xAF8A
  death.mp3: 0xAF8B to 0xC4F5
  jumpfb.mp3: 0xC4F6 to 0xCA7E
  jumpwg.mp3: 0xCA7F to 0xD5BE

audio_ctrl[2:0]
[2] is bgm-start, [1:0] is sound selection

Reference: spring 2024 Bubble Bobble
*/
module audio_play(input logic         clk,
                      input logic       reset,

                      input left_chan_ready,
                      input right_chan_ready,

                      input logic [2:0] audio_ctrl,

                      output logic [15:0] sample_data_l,
                      output logic sample_valid_l,
                      output logic [15:0] sample_data_r,
                      output logic sample_valid_r);

    logic [15:0] sound_begin_addresses [3:0] = '{16'h0 ,16'hAF8B, 16'hC4F6,
        16'hCA7F};
    logic [15:0] sound_end_addresses [3:0]   = '{16'h0 ,16'hC4F5, 16'hCA7E,
        16'hD5BE};

    logic [15:0] sound_address;
    logic [15:0] sound_end_address;
    logic [15:0] bgm_address; // loop
    logic [7:0] sound_data;
    logic [7:0] bgm_data;

    logic left_busy;
    logic right_busy;
```

```
38
39      logic bgm_playing;
40      logic sfx_playing;
41
42      logic [2:0] audio_ctrl_prev;
43
44      audio_rom u_audio_rom(
45                  .address_a    (bgm_address),
46                  .address_b  (sound_address),
47                  .clock    (clk    ),
48                  .q_a          (bgm_data),
49                  .q_b        (sound_data)
50              );
51
52
53
54      assign sample_data_l = bgm_data << 8;
55      assign sample_data_r = sound_data << 8;
56
57      always_ff @(posedge clk) begin
58          if (reset) begin
59              sample_valid_l <= 0;
60              sample_valid_r <= 0;
61              left_busy <= 0;
62              right_busy <= 0;
63              sound_address <= `BGM_BEGIN;
64              bgm_address <= `BGM_BEGIN;
65              sound_end_address <= `BGM_END;
66              bgm_playing <= 0;
67              sfx_playing <= 0;
68              audio_ctrl_prev <= 0;
69          end
70          else begin
71              if (audio_ctrl[2] != audio_ctrl_prev[2]) begin
72                  audio_ctrl_prev[2] <= audio_ctrl[2];
73                  if (audio_ctrl[2]) begin
74                      bgm_playing <= 1;
75                      bgm_address <= `BGM_BEGIN;
76                      if(!sfx_playing) begin
77                          sound_address <= `BGM_BEGIN;
78                          sound_end_address <= `BGM_END;
79                      end
80                  end
81                  else begin
82                      bgm_playing <= 0;
83                  end
84              end
85              if (audio_ctrl[1:0] != audio_ctrl_prev[1:0]) begin
86                  audio_ctrl_prev[1:0] <= audio_ctrl[1:0];
87                  if (audio_ctrl[1:0] !=0) begin
88                      sound_address <= sound_begin_addresses[audio_ctrl
                            [1:0]];
89                      sound_end_address <= sound_end_addresses[audio_ctrl
                            [1:0]];
90                      sfx_playing <= 1;
91                  end
92                  else begin
93                      sfx_playing <= 0;
94                  end
95              end
96              if (bgm_playing || sfx_playing) begin
97                  if (left_chan_ready == 1 && right_chan_ready == 1) begin
98                      if(left_busy == 0 && right_busy == 0) begin
99                          if(sound_address >= sound_end_address) begin
100                             if(sfx_playing) begin
101                                 sound_address <= bgm_address;
```

```
                                sound_end_address <= `BGM_END;
                                sfx_playing <= 0;
                            end
                            else begin
                                sound_address <= bgm_address;
                                sound_end_address <= `BGM_END;
                            end
                        end
                        else begin
                            sound_address <= sound_address + 1;
                        end

                        if (bgm_address >= `BGM_END) begin
                            bgm_address <= `BGM_BEGIN;
                        end
                        else begin
                            bgm_address <= bgm_address + 1;
                        end
                    end
                    left_busy <= 1;
                    right_busy <= 1;
                    sample_valid_l <= 1;
                    sample_valid_r <= 1;
                end
                else if (left_chan_ready == 0 && right_chan_ready == 0)
                    begin
                    left_busy <= 0;
                    right_busy <= 0;
                    sample_valid_l <= 0;
                    sample_valid_r <= 0;
                end
            end
            else begin
                sample_valid_l <= 0;
                sample_valid_r <= 0;
            end

        end
    end

endmodule
```

Listing 7: audio_play.sv

B: TestBench

```
`timescale 1ns/1ps

module tb_sprite_engine;

    parameter NUM_SPRITE = 32;
    parameter MAX_SLOT   = 8;

    logic clk;
    logic reset;
    logic sprite_start;
    logic [9:0] vcount;

    logic chipselect;
    logic write;
    logic [5:0] address;
    logic [31:0] writedata;

    logic spr_wr_en;
    logic [4:0] spr_wr_idx;
```

```
    logic [31:0] spr_wr_data;

    logic sprite_write_reg;
    logic [4:0] sprite_wr_idx;
    logic [31:0] sprite_writedata;

    always_ff @(posedge clk) begin
        if (reset) begin
            sprite_write_reg <= 0;
            sprite_wr_idx    <= 0;
            sprite_writedata <= 0;
        end else begin
            if (chipselect && write && address[5]) begin
                sprite_write_reg <= 1;
                sprite_wr_idx    <= address[4:0];
                sprite_writedata <= writedata;
            end else begin
                sprite_write_reg <= 0;
            end
        end
    end

    assign spr_wr_en   = sprite_write_reg;
    assign spr_wr_idx  = sprite_wr_idx;
    assign spr_wr_data = sprite_writedata;

    logic [9:0]  sprite_pixel_col;
    logic [15:0] sprite_pixel_data;
    logic        wren_pixel_draw;
    logic        done;

    always #5 clk = ~clk;

    // DUT
    sprite_engine #(
        .NUM_SPRITE(NUM_SPRITE),
        .MAX_SLOT(MAX_SLOT)
    ) u_eng (
        .clk(clk),
        .reset(reset),
        .sprite_start(sprite_start),
        .vcount(vcount),
        .spr_wr_en(spr_wr_en),
        .spr_wr_idx(spr_wr_idx),
        .spr_wr_data(spr_wr_data),
        .sprite_pixel_col(sprite_pixel_col),
        .sprite_pixel_data(sprite_pixel_data),
        .wren_pixel_draw(wren_pixel_draw),
        .done(done)
    );

    initial begin
        integer i;

        clk = 0;
        reset = 1;
        sprite_start = 0;
        vcount = 0;

        chipselect = 0;
        write = 0;
        address = 0;
        writedata = 0;

        #20 reset = 0;

```

```
86          $display("Writing sprites...");
87          write_sprite(0, 32'h83200000);
88          write_sprite(1, 32'h83201401);
89          write_sprite(2, 32'h83202802);
90          write_sprite(3, 32'h83203C03);
91          write_sprite(4, 32'h83205004);
92          write_sprite(5, 32'h83206405);
93          write_sprite(6, 32'h83207806);
94          write_sprite(7, 32'h83208C07);
95          write_sprite(8, 32'h8320A008);
96          write_sprite(9, 32'h8320B409);
97          write_sprite(10, 32'h8320C80A);
98          write_sprite(11, 32'h8320DC0B);
99          write_sprite(12, 32'h8320F00C);
100         write_sprite(13, 32'h8321040D);
101         write_sprite(14, 32'h8321180E);
102         write_sprite(15, 32'h83212C0F);
103         write_sprite(16, 32'h83214010);
104         write_sprite(17, 32'h83215411);
105         write_sprite(18, 32'h83216812);
106         write_sprite(19, 32'h83217C13);
107         write_sprite(20, 32'h83219014);
108         write_sprite(21, 32'h8321A415);
109         write_sprite(22, 32'h8321B816);
110         write_sprite(23, 32'h8321CC17);
111         write_sprite(24, 32'h8321E018);
112         write_sprite(25, 32'h8321F419);
113         write_sprite(26, 32'h8322081A);
114         write_sprite(27, 32'h83221C1B);
115         write_sprite(28, 32'h8322301C);
116         write_sprite(29, 32'h8322441D);
117         write_sprite(30, 32'h8322581E);
118         write_sprite(31, 32'h83226C1F);
119         $display("Write complete.");
120
121         vcount = 200;
122         sprite_start = 1;
123         @(posedge clk);
124         sprite_start = 0;
125         @(posedge clk);
126         @(posedge clk);
127         @(posedge clk);
128         @(posedge clk);
129         @(posedge clk);
130
131         wait(done);
132
133         @(posedge clk);
134         @(posedge clk);
135         @(posedge clk);
136         @(posedge clk);
137         @(posedge clk);
138
139         $display("Simulation complete: sprite_engine finished.");
140         $stop;
141     end
142
143     task write_sprite(input [4:0] idx, input [31:0] data);
144         begin
145             @(posedge clk);
146             chipselect = 1;
147             write = 1;
148             address = {1'b1, idx};
149             writedata = data;
150
151             @(posedge clk);
```

```
            chipselect = 0;
            write = 0;
            address = 0;
            writedata = 0;
        end
    endtask

endmodule
```

Listing 8: tb_sprite_engine.sv

```
`timescale 1ns/1ps

module tb_sprite_frontend;

    parameter NUM_SPRITE = 32;
    parameter MAX_SLOT   = 8;

    logic clk;
    logic reset;
    logic start_row;
    logic [9:0] next_vcount;

    // frontend <-> RAM
    logic [$clog2(NUM_SPRITE)-1:0] attr_ra;
    logic [31:0] attr_rd;

    // frontend <-> drawer
    logic draw_req;
    logic draw_done;
    logic [9:0] fe_col;
    logic fe_flip;
    logic [7:0] fe_frame;
    logic [3:0] fe_rowoff;
    logic fe_done;

    // drawer <-> ROM
    logic [15:0] rom_addr;
    logic [15:0] rom_q;
    logic [9:0]  pixel_col;
    logic [15:0] pixel_data;
    logic wren;

    logic [31:0] sprite_attr_ram [NUM_SPRITE];

    logic [15:0] sprite_rom [0:65535];

    // clock generation
    always #5 clk = ~clk;

    // DUT: frontend
    sprite_frontend #(
        .NUM_SPRITE(NUM_SPRITE),
        .MAX_SLOT(MAX_SLOT)
    ) u_fe (
        .clk(clk),
        .reset(reset),
        .start_row(start_row),
        .next_vcount(next_vcount),
        .ra(attr_ra),
        .rd_data(attr_rd),
        .draw_done(draw_done),
        .draw_req(draw_req),
        .col_base(fe_col),
        .flip(fe_flip),
        .frame_id(fe_frame),
```

```
        .row_off(fe_rowoff),
        .fe_done(fe_done)
    );

    // Read from sprite_attr_ram
    always_ff @(posedge clk) begin
        attr_rd <= sprite_attr_ram[attr_ra];
    end

    // DUT: drawer
    sprite_drawer u_drawer (
        .clk        (clk),
        .reset      (reset),
        .start      (draw_req),
        .col_base   (fe_col),
        .flip       (fe_flip),
        .frame_id   (fe_frame),
        .row_off    (fe_rowoff),
        .rom_addr   (rom_addr),
        .rom_q      (rom_q),
        .pixel_col  (pixel_col),
        .pixel_data (pixel_data),
        .wren       (wren),
        .done       (draw_done)
    );

    // ROM read
    always_ff @(posedge clk) begin
        rom_q <= sprite_rom[rom_addr];
    end
    integer i;
    initial begin
        clk = 0;
        reset = 1;
        start_row = 0;
        next_vcount = 10'd0;

        sprite_attr_ram[0]  = 32'h83200000;
        sprite_attr_ram[1]  = 32'h83201401;
        sprite_attr_ram[2]  = 32'h83202802;
        sprite_attr_ram[3]  = 32'h83203C03;
        sprite_attr_ram[4]  = 32'h83205004;
        sprite_attr_ram[5]  = 32'h83206405;
        sprite_attr_ram[6]  = 32'h83207806;
        sprite_attr_ram[7]  = 32'h83208C07;
        sprite_attr_ram[8]  = 32'h8320A008;
        sprite_attr_ram[9]  = 32'h8320B409;
        sprite_attr_ram[10] = 32'h8320C80A;
        sprite_attr_ram[11] = 32'h8320DC0B;
        sprite_attr_ram[12] = 32'h8320F00C;
        sprite_attr_ram[13] = 32'h8321040D;
        sprite_attr_ram[14] = 32'h8321180E;
        sprite_attr_ram[15] = 32'h83212C0F;
        sprite_attr_ram[16] = 32'h83214010;
        sprite_attr_ram[17] = 32'h83215411;
        sprite_attr_ram[18] = 32'h83216812;
        sprite_attr_ram[19] = 32'h83217C13;
        sprite_attr_ram[20] = 32'h83219014;
        sprite_attr_ram[21] = 32'h8321A415;
        sprite_attr_ram[22] = 32'h8321B816;
        sprite_attr_ram[23] = 32'h8321CC17;
        sprite_attr_ram[24] = 32'h8321E018;
        sprite_attr_ram[25] = 32'h8321F419;
        sprite_attr_ram[26] = 32'h8322081A;
        sprite_attr_ram[27] = 32'h83221C1B;
        sprite_attr_ram[28] = 32'h8322301C;
```

```
        sprite_attr_ram[29] = 32'h8322441D;
        sprite_attr_ram[30] = 32'h8322581E;
        sprite_attr_ram[31] = 32'h83226C1F;

        #20 reset = 0;

        #20;
        next_vcount = 10'd200;
        start_row = 1;
        #10 start_row = 0;

        wait(fe_done);
        #20;

        $display("Simulation finished.");
        $stop;
    end

endmodule
```

Listing 9: tb_sprite_frontend.sv

C: Device Driver

```
/*
 * vga_top      driver for FPGA VGA core
 *
 * Registers (byte offsets, 32-bit wide)
 *   0x00   CTRL_REG            W
 *   0x04   STATUS_REG          R
 *   0x80..0xFC   SPRITE[n]     R/W   (n = 0-31)
 *
 */

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

#define DRIVER_NAME "vga_top"

/* ---------- register helpers ---------- */
#define CTRL_REG(base)     ((base) + 0x00)
#define STATUS_REG(base)   ((base) + 0x04)
#define SPRITE_REG(base,n) ((base) + 0x80 + ((n) * 4))

/*
 * Information about our device
 */
struct vga_top_dev {
    struct resource res; /* Resource: our registers */
    void __iomem *virtbase; /* Where registers can be accessed in memory */
    u32 cached_ctrl;
} dev;

```

```
42 /*
43  * Handle ioctl() calls from userspace:
44  * Read or write the segments on single digits.
45  * Note extensive error checking of arguments
46  */
47 static long vga_top_ioctl(struct file *f, unsigned int cmd, unsigned long
        arg)
48 {
49     vga_top_ctrl_arg_t   c_arg;
50     vga_top_status_arg_t s_arg;
51     vga_top_sprite_arg_t sp_arg;
52 
53     switch (cmd) {
54     case VGA_TOP_WRITE_CTRL:
55         if (copy_from_user(&c_arg, (vga_top_ctrl_arg_t *) arg, sizeof(
                vga_top_ctrl_arg_t)))
56             return -EACCES;
57             iowrite32(c_arg.value, CTRL_REG(dev.virtbase));
58             dev.cached_ctrl = c_arg.value;
59         break;
60 
61     case VGA_TOP_READ_STATUS:
62         s_arg.value = ioread32(STATUS_REG(dev.virtbase));
63         if (copy_to_user((vga_top_status_arg_t *) arg, &s_arg, sizeof(
                vga_top_status_arg_t)))
64             return -EACCES;
65         break;
66 
67     case VGA_TOP_WRITE_SPRITE:
68         if (copy_from_user(&sp_arg, (vga_top_sprite_arg_t *) arg, sizeof(
                vga_top_sprite_arg_t)))
69             return -EACCES;
70         if (sp_arg.index > 31)
71             return -EINVAL;
72         iowrite32(sp_arg.attr_word, SPRITE_REG(dev.virtbase, sp_arg.index))
                ;
73         break;
74 
75     default:
76         return -EINVAL;
77     }
78 
79     return 0;
80 }
81 
82 /* The operations our device knows how to do */
83 static const struct file_operations vga_top_fops = {
84     .owner      = THIS_MODULE,
85     .unlocked_ioctl = vga_top_ioctl,
86 };
87 
88 /* Information about our device for the "misc" framework -- like a char dev
        */
89 static struct miscdevice vga_top_misc_device = {
90     .minor      = MISC_DYNAMIC_MINOR,
91     .name       = DRIVER_NAME,
92     .fops       = &vga_top_fops,
93 };
94 
95 /*
96  * Initialization code: get resources (registers) and display
97  * a welcome message
98  */
99 static int __init vga_top_probe(struct platform_device *pdev)
100 {
101     int ret;
```

```
102
103     /* Register ourselves as a misc device: creates /dev/vga_top */
104     ret = misc_register(&vga_top_misc_device);
105
106     /* Get the address of our registers from the device tree */
107     ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
108     if (ret) {
109         ret = -ENOENT;
110         goto out_deregister;
111     }
112
113     /* Make sure we can use these registers */
114     if (request_mem_region(dev.res.start, resource_size(&dev.res),
115                            DRIVER_NAME) == NULL) {
116         ret = -EBUSY;
117         goto out_deregister;
118     }
119
120     /* Arrange access to our registers */
121     dev.virtbase = of_iomap(pdev->dev.of_node, 0);
122     if (dev.virtbase == NULL) {
123         ret = -ENOMEM;
124         goto out_release_mem_region;
125     }
126
127     return 0;
128
129 out_release_mem_region:
130     release_mem_region(dev.res.start, resource_size(&dev.res));
131 out_deregister:
132     misc_deregister(&vga_top_misc_device);
133     return ret;
134 }
135
136 /* Clean-up code: release resources */
137 static int vga_top_remove(struct platform_device *pdev)
138 {
139     iounmap(dev.virtbase);
140     release_mem_region(dev.res.start, resource_size(&dev.res));
141     misc_deregister(&vga_top_misc_device);
142     return 0;
143 }
144
145 /* Which "compatible" string(s) to search for in the Device Tree */
146 #ifdef CONFIG_OF
147 static const struct of_device_id vga_top_of_match[] = {
148     { .compatible = "csee4840,vga_top-1.0" },
149     {},
150 };
151 MODULE_DEVICE_TABLE(of, vga_top_of_match);
152 #endif
153
154 /* Information for registering ourselves as a "platform" driver */
155 static struct platform_driver vga_top_driver = {
156     .driver = {
157         .name   = DRIVER_NAME,
158         .owner  = THIS_MODULE,
159         .of_match_table = of_match_ptr(vga_top_of_match),
160     },
161     .remove = __exit_p(vga_top_remove),
162 };
163
164 /* Called when the module is loaded: set things up */
165 static int __init vga_top_init(void)
166 {
167     pr_info(DRIVER_NAME ": init\n");
```

```
168     return platform_driver_probe(&vga_top_driver, vga_top_probe);
169 }
170
171 /* Calball when the module is unloaded: release resources */
172 static void __exit vga_top_exit(void)
173 {
174     platform_driver_unregister(&vga_top_driver);
175     pr_info(DRIVER_NAME ": exit\n");
176 }
177
178 module_init(vga_top_init);
179 module_exit(vga_top_exit);
180
181 MODULE_LICENSE("GPL");
182 MODULE_AUTHOR("ForestFireIce CSEE 4840");
183 MODULE_DESCRIPTION("VGA_TOP misc driver");
```

Listing 10: vga_top.c

D: Software

```
1  #include "vga_top.h"
2  #include <fcntl.h>
3  #include <unistd.h>
4  #include <stdio.h>
5  #include <stdint.h>
6  #include <sys/ioctl.h>
7
8  int vga_top_fd;
9
10 inline uint32_t make_attr_word(uint8_t enable, uint8_t flip,
11                                 uint16_t x, uint16_t y,
12                                 uint8_t frame)
13 {
14     return ((uint32_t)(enable & 1) << 31) |
15            ((uint32_t)(flip & 1) << 30) |
16            (0u << 27) |
17            ((uint32_t)(y & 0x1FF) << 18) |
18            ((uint32_t)(x & 0x3FF) << 8) |
19            (frame & 0xFF);
20 }
21
22 void write_ctrl(uint32_t value)
23 {
24     vga_top_ctrl_arg_t arg = {.value = value};
25     if (ioctl(vga_top_fd, VGA_TOP_WRITE_CTRL, &arg))
26     {
27         perror("ioctl(VGA_TOP_WRITE_CTRL) failed");
28         return;
29     }
30 }
31
32 /* build control word */
33 inline uint32_t make_ctrl_word(uint8_t tilemap_idx,
34                                 uint8_t bgm_on,
35                                 uint8_t sfx_sel)
36 {
37     uint32_t tmap = (uint32_t)(tilemap_idx & 0x3);   // [1:0]
38     uint32_t audio = ((uint32_t)(bgm_on & 0x1) << 2) // [31:29] bit2 = BGM
39                    | (sfx_sel & 0x3);                // [1:0] = SFX
                          selection
40     return (audio << 29) | tmap;
41 }
42
43 /* High-level wrapper: set map and audio simultaneously */
```

```c
void set_map_and_audio(uint8_t tilemap_idx,
                       uint8_t bgm_on,
                       uint8_t sfx_sel)
{
    uint32_t ctrl = make_ctrl_word(tilemap_idx, bgm_on, sfx_sel);
    write_ctrl(ctrl);
}

void write_sprite(uint8_t index,
                  uint8_t enable, uint8_t flip,
                  uint16_t x, uint16_t y,
                  uint8_t frame)
{
    vga_top_sprite_arg_t arg = {
        .index = index,
        .attr_word = make_attr_word(enable, flip, x, y, frame)};
    if (ioctl(vga_top_fd, VGA_TOP_WRITE_SPRITE, &arg))
    {
        perror("ioctl(VGA_TOP_WRITE_SPRITE) failed");
        return;
    }
}

void read_status(unsigned *col, unsigned *row)
{
    vga_top_status_arg_t arg;
    if (ioctl(vga_top_fd, VGA_TOP_READ_STATUS, &arg))
    {
        perror("ioctl(VGA_TOP_READ_STATUS) failed");
        return;
    }

    *col = (arg.value >> 10) & 0x3FF;
    *row = arg.value & 0x3FF;
}
```

Listing 11: hw_interact.c

```c
/**
 * @file joypad_input.c
 * @brief Implementation of the Joypad controller input device driver
 *
 * This file provides the implementation for using game controllers (
    Joypads)
 * as input devices, supporting the connection of two controllers
 * to control two game characters.
 * Specifically supports the classic 8-button joypad layout.
 */

#include "../include/joypad_input.h"
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <linux/joystick.h>
#include <linux/input.h>
#include <sys/time.h>

/* Constants */
#define JOYPAD_1_DEVICE "/dev/input/event0" // First joypad device
#define JOYPAD_2_DEVICE "/dev/input/event1" // Second joypad device

/* Classic joypad button mapping - Direction buttons on the left side (D-
    pad) */
```

```c
#define JOYPAD_BTN_UP 0    // Up direction button
#define JOYPAD_BTN_DOWN 1  // Down direction button
#define JOYPAD_BTN_LEFT 2  // Left direction button
#define JOYPAD_BTN_RIGHT 3 // Right direction button

/* Classic joypad button mapping - Function buttons on the right side */
#define JOYPAD_BTN_A 4 // A button
#define JOYPAD_BTN_B 5 // B button
#define JOYPAD_BTN_X 6 // X button
#define JOYPAD_BTN_Y 7 // Y button

/* Joypad state structure */
typedef struct
{
    int fd;         // Device file descriptor
    bool connected; // Connection status

    // Direction button states (left side four buttons)
    bool btn_up;
    bool btn_down;
    bool btn_left;
    bool btn_right;

    // Function button states (right side four buttons)
    bool btn_a;
    bool btn_b;
    bool btn_x;
    bool btn_y;

} joypad_state_t;

/* Global variables */
static joypad_state_t joypads[2]; // Support for up to two joypads

/**
 * @brief Initialize the Joypad input module
 * Attempts to connect to joypad devices and set them to non-blocking mode
 *
 * @return 0 on success, -1 on failure
 */
int input_handler_init()
{
    printf("Initializing Joypad Input Handler...\n");

    // Initialize joypad states
    for (int i = 0; i < 2; i++)
    {
        joypads[i].connected = false;
        joypads[i].btn_up = false;
        joypads[i].btn_down = false;
        joypads[i].btn_left = false;
        joypads[i].btn_right = false;
        joypads[i].btn_a = false;
        joypads[i].btn_b = false;
        joypads[i].btn_x = false;
        joypads[i].btn_y = false;
        joypads[i].fd = -1;
    }

    // Try to open the first joypad
    joypads[0].fd = open(JOYPAD_1_DEVICE, O_RDONLY | O_NONBLOCK);
    if (joypads[0].fd != -1)
    {
        joypads[0].connected = true;
        printf("Successfully connected first joypad (Player 1)\n");
    }
```

```
93      else
94      {
95          printf("Could not connect first joypad, keyboard will be used as
                fallback\n");
96      }
97
98      // Try to open the second joypad
99      joypads[1].fd = open(JOYPAD_2_DEVICE, O_RDONLY | O_NONBLOCK);
100     if (joypads[1].fd != -1)
101     {
102         joypads[1].connected = true;
103         printf("Successfully connected second joypad (Player 2)\n");
104     }
105     else
106     {
107         printf("Could not connect second joypad, keyboard will be used as
                fallback\n");
108     }
109
110     return 0;
111 }
112
113 /**
114  * @brief Clean up the Joypad input module
115  * Close joypad device files
116  */
117 void input_handler_cleanup()
118 {
119     printf("Cleaning up Joypad Input Handler...\n");
120
121     // Close joypad devices
122     for (int i = 0; i < 2; i++)
123     {
124         if (joypads[i].connected && joypads[i].fd != -1)
125         {
126             close(joypads[i].fd);
127             joypads[i].connected = false;
128         }
129     }
130 }
131
132 /**
133  * @brief Update Joypad state
134  * Read joypad events and update status
135  *
136  * @param player_index Player index (0 or 1)
137  */
138 static void update_joypad_state(int player_index)
139 {
140     if (player_index < 0 || player_index > 1 || !joypads[player_index].
            connected)
141     {
142         return;
143     }
144
145     // Using input_event structure to read events
146     struct input_event event;
147
148     // Read all pending events
149     while (read(joypads[player_index].fd, &event, sizeof(event)) > 0)
150     {
151         // Handle button events (type=1)
152         if (event.type == 1)
153         { // EV_KEY
154             switch (event.code)
155             {
```

```
            case 288: // X button
                joypads[player_index].btn_x = (event.value != 0);
                break;
            case 289: // A button
                joypads[player_index].btn_a = (event.value != 0);
                break;
            case 290: // B button
                joypads[player_index].btn_b = (event.value != 0);
                break;
            case 291: // Y button
                joypads[player_index].btn_y = (event.value != 0);
                break;
            case 296: // Select button
                // Can add Select button handling here
                break;
            case 297: // Start button
                // Can add Start button handling here
                break;
            }
        }
        // Handle directional events (type=3)
        else if (event.type == 3)
        { // EV_ABS
            if (event.code == 0)
            { // X axis
                if (event.value == 0)
                { // Left button pressed
                    joypads[player_index].btn_left = true;
                    joypads[player_index].btn_right = false;
                }
                else if (event.value == 255)
                { // Right button pressed
                    joypads[player_index].btn_left = false;
                    joypads[player_index].btn_right = true;
                }
                else if (event.value == 127)
                { // Left and right buttons released
                    joypads[player_index].btn_left = false;
                    joypads[player_index].btn_right = false;
                }
            }
            else if (event.code == 1)
            { // Y axis
                if (event.value == 0)
                { // Up button pressed
                    joypads[player_index].btn_up = true;
                    joypads[player_index].btn_down = false;
                }
                else if (event.value == 255)
                { // Down button pressed
                    joypads[player_index].btn_up = false;
                    joypads[player_index].btn_down = true;
                }
                else if (event.value == 127 || event.value == 126)
                { // Up and down buttons released
                    joypads[player_index].btn_up = false;
                    joypads[player_index].btn_down = false;
                }
            }
        }
    }
}

/**
 * @brief Get the current game action for a player
 * Determine the current game action based on joypad state
```

```
 *
 * @param player_index Player index (0 for Fireboy, 1 for Watergirl)
 * @return The game action corresponding to the current input
 */
game_action_t get_player_action(int player_index)
{
    // Ensure valid player index
    if (player_index < 0 || player_index > 1)
    {
        return ACTION_NONE;
    }

    // If joypad is connected, update joypad state
    if (joypads[player_index].connected)
    {
        update_joypad_state(player_index);

        // Determine action based on joypad state
        // Jump has highest priority (using X button and up direction)
        if (joypads[player_index].btn_x || joypads[player_index].btn_up)
        {
            return ACTION_JUMP;
        }
        // Left/right movement (using direction buttons)
        else if (joypads[player_index].btn_left)
        {
            return ACTION_MOVE_LEFT;
        }
        else if (joypads[player_index].btn_right)
        {
            return ACTION_MOVE_RIGHT;
        }
    }

    // Default no action
    return ACTION_NONE;
}

/**
 * @brief Connect a new player joypad device
 * Attempts to open and connect a joypad device at the specified path
 *
 * @param device_path The joypad device path
 * @param player_index The player index to bind to
 * @return 0 on success, -1 on failure
 */
int insert_joypad(const char *device_path, int player_index)
{
    if (player_index < 0 || player_index > 1)
    {
        printf("Error: Invalid player index\n");
        return -1;
    }

    // If there's already a connected joypad, close it first
    if (joypads[player_index].connected && joypads[player_index].fd != -1)
    {
        close(joypads[player_index].fd);
        joypads[player_index].connected = false;
    }

    // Try to open the new joypad device
    joypads[player_index].fd = open(device_path, O_RDONLY | O_NONBLOCK);
    if (joypads[player_index].fd != -1)
    {
        joypads[player_index].connected = true;
```

```
288         // Reset all button states
289         joypads[player_index].btn_up = false;
290         joypads[player_index].btn_down = false;
291         joypads[player_index].btn_left = false;
292         joypads[player_index].btn_right = false;
293         joypads[player_index].btn_a = false;
294         joypads[player_index].btn_b = false;
295         joypads[player_index].btn_x = false;
296         joypads[player_index].btn_y = false;
297 
298         printf("Successfully connected Player %d joypad\n", player_index +
                1);
299         return 0;
300     }
301     else
302     {
303         printf("Failed to connect Player %d joypad\n", player_index + 1);
304         return -1;
305     }
306 }
307 
308 /**
309  * @brief Get the default joypad device path for a player
310  *
311  * @param player_index Player index (0 for first player, 1 for second
        player)
312  * @return The default joypad device path
313  */
314 const char *get_default_joypad_path(int player_index)
315 {
316     if (player_index == 0)
317     {
318         return JOYPAD_1_DEVICE;
319     }
320     else if (player_index == 1)
321     {
322         return JOYPAD_2_DEVICE;
323     }
324     else
325     {
326         return NULL; // Invalid player index
327     }
328 }
329 
330 /**
331  * @brief Check if a player's joypad is connected
332  *
333  * @param player_index Player index (0 for first player, 1 for second
        player)
334  * @return 1 if connected, 0 if not connected
335  */
336 int is_joypad_connected(int player_index)
337 {
338     if (player_index < 0 || player_index > 1)
339     {
340         return 0; // Invalid player index
341     }
342 
343     return joypads[player_index].connected ? 1 : 0;
344 }
345 
346 /**
347  * @brief Get the state of a specific button for a player
348  *
349  * @param player_index Player index
350  * @param button_id Button ID (see JOYPAD_BTN_* constants)
```

```
 * @return 1 if pressed, 0 if not pressed
 */
int get_joypad_button_state(int player_index, int button_id)
{
    if (player_index < 0 || player_index > 1 || !joypads[player_index].
        connected)
    {
        return 0;
    }

    // Update joypad state
    update_joypad_state(player_index);

    // Return state based on button ID
    switch (button_id)
    {
    case JOYPAD_BTN_UP:
        return joypads[player_index].btn_up ? 1 : 0;
    case JOYPAD_BTN_DOWN:
        return joypads[player_index].btn_down ? 1 : 0;
    case JOYPAD_BTN_LEFT:
        return joypads[player_index].btn_left ? 1 : 0;
    case JOYPAD_BTN_RIGHT:
        return joypads[player_index].btn_right ? 1 : 0;
    case JOYPAD_BTN_A:
        return joypads[player_index].btn_a ? 1 : 0;
    case JOYPAD_BTN_B:
        return joypads[player_index].btn_b ? 1 : 0;
    case JOYPAD_BTN_X:
        return joypads[player_index].btn_x ? 1 : 0;
    case JOYPAD_BTN_Y:
        return joypads[player_index].btn_y ? 1 : 0;
    default:
        return 0;
    }
}
```

Listing 12: joypad_input.c

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include "hw_interact.h"
#include "player.h"
#include "joypad_input.h"
#include "sprite.h"
#include "type.h"
#include <time.h>

player_t players[NUM_PLAYERS];
item_t items[NUM_ITEMS];
box_t boxes[NUM_BOXES];
lever_t levers[NUM_LEVERS];
elevator_t elevators[NUM_ELEVATORS];
button_t buttons[NUM_BUTTONS];
unsigned frame_counter = 0;

int main()
{
    if ((vga_top_fd = open("/dev/vga_top", O_RDWR)) == -1)
    {
        fprintf(stderr, "Error: cannot open /dev/vga_top\n");
        return -1;
    }
Logo:
    input_handler_init();
```

```
    for (int i = 0; i < 32; i++)
    {
        write_sprite(i, 0, 0, 0, 0, 0); // disable=0, position 0, frame 0
    }
    while (1)
    {
        set_map_and_audio(0, 0, 0); // Start VGA controller
        for (int i = 0; i < NUM_PLAYERS; i++)
        {
            game_action_t action = get_player_action(i);
            if (action != ACTION_NONE)
            {
                goto Game;
            }
        }
    }
    // debug_draw_test_sprites();
Game:
    set_map_and_audio(1, 1, 0);
    input_handler_init();
    player_init(&players[0], 64, 360, 0, 1, PLAYER_FIREBOY);
    player_init(&players[1], 64, 420, 2, 3, PLAYER_WATERGIRL);

    // player_init(&players[0], 368, 224, 0, 1, PLAYER_FIREBOY);
    // player_init(&players[1], 320, 152, 2, 3, PLAYER_WATERGIRL);

    item_init(&items[0], 0, 0, 4, BLUE_GEM_FRAME);
    item_place_on_tile(&items[0], 21, 26);
    items[0].sprite.frame_count = 1;
    items[0].sprite.frame_start = BLUE_GEM_FRAME;
    items[0].owner_type = ITEM_WATERGIRL_ONLY;
    items[0].float_anim = true;
    items[0].width = 12;  // Collision box width
    items[0].height = 12; // Collision box height

    item_init(&items[1], 0, 0, 5, RED_GEM_FRAME);
    item_place_on_tile(&items[1], 29, 26);
    items[1].sprite.frame_count = 1;
    items[1].sprite.frame_start = RED_GEM_FRAME;
    items[1].owner_type = ITEM_FIREBOY_ONLY;
    items[1].float_anim = true;
    items[1].width = 12;  // Collision box width
    items[1].height = 12; // Collision box height

    item_init(&items[2], 0, 0, 6, RED_GEM_FRAME);
    item_place_on_tile(&items[2], 6, 14);
    items[2].sprite.frame_count = 1;
    items[2].sprite.frame_start = RED_GEM_FRAME;
    items[2].owner_type = ITEM_FIREBOY_ONLY;
    items[2].float_anim = true;
    items[2].width = 12;  // Collision box width
    items[2].height = 12; // Collision box height

    item_init(&items[3], 0, 0, 7, BLUE_GEM_FRAME);
    item_place_on_tile(&items[3], 23, 14);
    items[3].sprite.frame_count = 1;
    items[3].sprite.frame_start = BLUE_GEM_FRAME;
    items[3].owner_type = ITEM_WATERGIRL_ONLY;
    items[3].float_anim = true;
    items[3].width = 12;  // Collision box width
    items[3].height = 12; // Collision box height

    item_init(&items[4], 0, 0, 8, BLUE_GEM_FRAME);
    item_place_on_tile(&items[4], 11, 7);
    items[4].sprite.frame_count = 1;
    items[4].sprite.frame_start = BLUE_GEM_FRAME;
```

```
94      items[4].owner_type = ITEM_WATERGIRL_ONLY;
95      items[4].float_anim = true;
96      items[4].width = 12;  // Collision box width
97      items[4].height = 12; // Collision box height
98
99      item_init(&items[5], 0, 0, 9, RED_GEM_FRAME);
100     item_place_on_tile(&items[5], 1, 4);
101     items[5].sprite.frame_count = 1;
102     items[5].sprite.frame_start = RED_GEM_FRAME;
103     items[5].owner_type = ITEM_FIREBOY_ONLY;
104     items[5].float_anim = true;
105     items[5].width = 12;  // Collision box width
106     items[5].height = 12; // Collision box height
107
108     box_init(&boxes[0], 17, 10, 10, BOX_FRAME);
109
110     lever_init(&levers[0], 9, 21, 22);
111
112     elevator_init(&elevators[0], 1, 16, 16, 19, 14, 51);
113     elevator_init(&elevators[1], 35, 12, 12, 16, 18, 57);
114
115     button_init(&buttons[0], 32, 12, 26);
116     button_init(&buttons[1], 32, 17, 29);
117
118     unsigned col = 0, row = 0;
119     while (1)
120     {
121
122         // clock_t start = clock();
123         frame_counter++;
124         // === Frame synchronization: execute only once at the top of each
                frame (row==0) ===
125         do
126         {
127             read_status(&col, &row);
128         } while (row != 0);
129
130         // === 1. Logic update phase ===
131         for (int i = 0; i < NUM_PLAYERS; i++)
132         {
133             player_handle_input(&players[i], i);
134             int situation = player_update_physics(&players[i]);
135             if (situation == 1)
136             {
137                 set_map_and_audio(1, 1, 0);
138                 set_map_and_audio(1, 1, 2);
139                 sleep(1);
140                 goto Logo;
141             }
142             else if (situation == 2)
143             {
144                 goto Logo;
145             }
146
147             for (int j = 0; j < NUM_ITEMS; j++)
148             {
149                 if (!items[j].active)
150                     continue;
151
152                 // Determine if the character is allowed to collect
153                 if ((items[j].owner_type == ITEM_FIREBOY_ONLY && players[i
                        ].type != PLAYER_FIREBOY) ||
154                     (items[j].owner_type == ITEM_WATERGIRL_ONLY && players[
                            i].type != PLAYER_WATERGIRL))
155                 {
156                     continue;
```

```
157                 }
158
159                 float pw = SPRITE_W_PIXELS;      // Width stays at 16
160                 float ph = PLAYER_HITBOX_HEIGHT; // Actual height that
                        participates in collision
161                 float px = players[i].x;
162                 float py = players[i].y + PLAYER_HITBOX_OFFSET_Y; // Skip
                        transparent pixel area at the top
163
164                 if (check_overlap(px, py, pw, ph,
165                                   items[j].x, items[j].y, items[j].width,
                                      items[j].height))
166                 {
167                     items[j].active = false;
168                 }
169             }
170             for (int i = 0; i < NUM_BOXES; i++)
171             {
172                 for (int j = 0; j < NUM_PLAYERS; j++)
173                 {
174                     box_try_push(&boxes[i], &players[j]);
175                 }
176                 box_update_position(&boxes[i], players);
177             }
178             for (int i = 0; i < NUM_LEVERS; i++)
179             {
180                 lever_update(&levers[0], players);
181             }
182             for (int i = 0; i < NUM_ELEVATORS; i++)
183             {
184                 if (i == 0)
185                     elevator_update(&elevators[i], levers[0].activated,
                            players);
186                 if (i == 1)
187                 {
188                     elevator_update(&elevators[i], buttons[0].pressed ||
                            buttons[1].pressed, players);
189                 }
190             }
191             for (int i = 0; i < NUM_BUTTONS; i++)
192             {
193                 button_update(&buttons[i], players);
194             }
195         }
196         // === 2. Wait for blanking area ===
197         do
198         {
199             read_status(&col, &row);
200         } while (row < VACTIVE);
201
202         // === 3. Write sprites to VGA ===
203         for (int i = 0; i < NUM_PLAYERS; i++)
204         {
205             player_update_sprite(&players[i]);
206         }
207         for (int j = 0; j < NUM_ITEMS; j++)
208         {
209             item_update_sprite(&items[j]);
210         }
211         for (int i = 0; i < NUM_BOXES; i++)
212         {
213             box_update_sprite(&boxes[i]);
214         }
215         // clock_t end = clock();
216         // float duration = (float)(end - start) / CLOCKS_PER_SEC * 1000;
217         // printf("[FRAME] duration = %.2f ms\n", duration);
```

```
218     }
219
220     // Will not reach here. If there are exit conditions in the future,
            resources can be released:
221     input_handler_cleanup();
222     close(vga_top_fd);
223     return 0;
224 }
225 // void debug_draw_test_sprites()
226 // {
227 // int index = 14;
228 // int y = 200;
229
230 // // Lever base (2 frames)
231 // for (int i = 0; i < 2; ++i)
232 //     write_sprite(index++, 1, 0, 16 * i + 200, y, LEVER_BASE_FRAME + i);
233
234 // y += 20;
235 // // Lever animation (3 frames)
236 // for (int i = 0; i < 3; ++i)
237 //     write_sprite(index++, 1, 0, 16 * i + 200, y, LEVER_ANIM_FRAME + i);
238
239 // y += 20;
240 // // Yellow elevator (4 frames)
241 // for (int i = 0; i < 4; ++i)
242 //     write_sprite(index++, 1, 0, 16 * i + 200, y, LIFT_YELLOW_FRAME + i);
243
244 // y += 20;
245 // // Purple button (2 frames)
246 // for (int i = 0; i < 2; ++i)
247 //     write_sprite(index++, 1, 0, 16 * i + 200, y, BUTTON_PURPLE_FRAME + i
        );
248
249 // y += 20;
250 // // Purple elevator (4 frames)
251 // for (int i = 0; i < 4; ++i)
252 //     write_sprite(index++, 1, 0, 16 * i + 200, y, LIFT_PURPLE_FRAME + i);
253 // }
```

Listing 13: main.c

```
1  #include "player.h"
2  #include "joypad_input.h"
3  #include "tilemap.h"
4  #include "hw_interact.h"
5  #include <math.h> // For floor()
6  #include "type.h"
7  #include <stdio.h> // Adding this at the top
8  #include <stdlib.h>
9  #include <string.h>
10 #include <stdbool.h>
11
12 #define GRAVITY 0.2f
13 #define JUMP_VELOCITY -4.5f
14 #define MOVE_SPEED 2.5f
15
16 extern box_t boxes[NUM_BOXES];
17 void debug_print_player_state(player_t *p, const char *tag)
18 {
19     float center_x = p->x + SPRITE_W_PIXELS / 2.0f;
20     float foot_y = p->y + PLAYER_HEIGHT_PIXELS + 1;
21     int tile = get_tile_at_pixel(center_x, foot_y);
22     int tx = (int)(center_x / TILE_SIZE);
23     int ty = (int)(foot_y / TILE_SIZE);
24
25     printf("[%s] x=%.1f y=%.1f vx=%.2f vy=%.2f on_ground=%d foot_tile=%d (%
```

```
        d,%d)\n",
26              tag, p->x, p->y, p->vx, p->vy, p->on_ground, tile, tx, ty);
27  }
28  void player_init(player_t *p, int x, int y,
29                   uint8_t upper_index, uint8_t lower_index,
30                   player_type_t type)
31  {
32      p->x = x;
33      p->y = y;
34      p->vx = p->vy = 0;
35      p->on_ground = false;
36      p->state = STATE_IDLE;
37      p->type = type;
38      p->frame_timer = 0;
39      p->frame_index = 0;
40      p->was_on_slope_last_frame = false;
41
42      if (type == PLAYER_FIREBOY)
43      {
44          sprite_set(&p->upper_sprite, upper_index, 0);
45          sprite_set(&p->lower_sprite, lower_index, 0);
46      }
47      else
48      {
49          sprite_set(&p->upper_sprite, upper_index, 0);
50          sprite_set(&p->lower_sprite, lower_index, 0);
51      }
52  }
53
54  void player_handle_input(player_t *p, int player_index)
55  {
56      game_action_t action = get_player_action(player_index);
57
58      // Handle jumping, must be placed first
59      if (action == ACTION_JUMP && p->on_ground)
60      {
61          set_map_and_audio(1, 1, 0);
62          set_map_and_audio(1, 1, 1);
63          p->vy = JUMP_VELOCITY;
64          p->on_ground = false;
65          p->state = STATE_JUMPING;
66      }
67
68      // Handle horizontal movement
69      if (action == ACTION_MOVE_LEFT)
70      {
71          p->vx = -MOVE_SPEED;
72          p->lower_sprite.flip = 1;
73          p->upper_sprite.flip = 1;
74      }
75      else if (action == ACTION_MOVE_RIGHT)
76      {
77          p->vx = MOVE_SPEED;
78          p->lower_sprite.flip = 0;
79          p->upper_sprite.flip = 0;
80      }
81      else if (p->on_ground) // Don't immediately cancel horizontal velocity
            in air
82      {
83          p->vx = 0;
84      }
85  }
86
87  int player_update_physics(player_t *p)
88  {
89      p->vy += GRAVITY;
```

```
90      // Vertical movement
91      float tempVy = 0.0f;
92      float new_y = p->y + p->vy;
93      if (is_death(p->x, new_y + 1, SPRITE_W_PIXELS, PLAYER_HEIGHT_PIXELS, p
            ->type))
94      {
95          return 1;
96      }
97      if (check_both_players_goal())
98      {
99          return 2;
100     }
101     if (!is_tile_blocked(p->x, new_y + 1, SPRITE_W_PIXELS,
            PLAYER_HEIGHT_PIXELS) &&
102         !is_box_blocked(p->x + SPRITE_W_PIXELS / 2.0f, new_y +
                PLAYER_HITBOX_OFFSET_Y, 1.0f, PLAYER_HITBOX_HEIGHT) &&
103         !is_elevator_blocked(p->x + SPRITE_W_PIXELS / 2.0f - 2, new_y +
                PLAYER_HITBOX_OFFSET_Y, 4.0f, PLAYER_HITBOX_HEIGHT, &tempVy))
104     {
105         p->y = new_y;
106         p->on_ground = false;
107     }
108     else if (is_elevator_blocked(p->x + SPRITE_W_PIXELS / 2.0f - 2, new_y +
             PLAYER_HITBOX_OFFSET_Y, 4.0f, PLAYER_HITBOX_HEIGHT, &tempVy))
109     {
110         // Call your attachment function
111         adjust_to_platform_y(p);
112
113         if (p->vy > 0)
114             p->on_ground = true;
115
116         p->vy = tempVy;
117     }
118     else
119     {
120         // Call your attachment function
121         adjust_to_platform_y(p);
122
123         if (p->vy > 0)
124             p->on_ground = true;
125
126         p->vy = 0;
127     }
128     // Horizontal movement
129     float new_x = p->x + p->vx;
130
131     // Calculate current position and target position's foot center point
132     float cur_foot_x = p->x + SPRITE_W_PIXELS / 2.0f;
133     float new_foot_x = new_x + SPRITE_W_PIXELS / 2.0f;
134     float foot_y = p->y + PLAYER_HEIGHT_PIXELS;
135
136     // Determine whether current position's foot adjacent to left and right
             are in slope range
137     bool on_slope = false;
138     for (int dx = -1; dx <= 1; ++dx)
139     {
140         int tile = get_tile_at_pixel(new_foot_x + dx * 8, foot_y);
141         if (tile == TILE_SLOPE_L_UP || tile == TILE_SLOPE_R_UP)
142         {
143             on_slope = true;
144             break;
145         }
146     }
147     if (on_slope && p->vy >= 0)
148     {
149         p->x = new_x;
```

```
150         adjust_to_slope_y(p);
151     }
152     else if (!is_tile_blocked(new_x, p->y, SPRITE_W_PIXELS,
            PLAYER_HEIGHT_PIXELS) &&
153              !is_box_blocked(new_x + SPRITE_W_PIXELS / 2.0f, p->y +
                     PLAYER_HITBOX_OFFSET_Y, 1.0f, PLAYER_HITBOX_HEIGHT) &&
154              !is_elevator_blocked(new_x + SPRITE_W_PIXELS / 2.0f - 2, p->y
                     + PLAYER_HITBOX_OFFSET_Y, 4.0f, PLAYER_HITBOX_HEIGHT - 4,
                     &tempVy))
155     {
156         p->x = new_x;
157     }
158     else
159     {
160         if (is_tile_blocked(new_x, p->y, SPRITE_W_PIXELS,
                PLAYER_HEIGHT_PIXELS))
161         {
162             p->vx = 0;
163         }
164         else if (is_box_blocked(new_x + SPRITE_W_PIXELS / 2.0f, p->y +
                PLAYER_HITBOX_OFFSET_Y, 1.0f, PLAYER_HITBOX_HEIGHT))
165         {
166             if (boxes[0].vx != 0)
167                 p->vx = boxes[0].vx;
168             else
169             {
170                 int player_index = (p->type == PLAYER_FIREBOY) ? 0 : 1;
171                 game_action_t action = get_player_action(player_index);
172                 if (action == ACTION_MOVE_RIGHT)
173                     p->vx = 0.5f;
174                 else if (action == ACTION_MOVE_LEFT)
175                     p->vx = -0.5f;
176                 else
177                     p->vx = 0;
178             }
179         }
180         else
181         {
182             p->vx = 0;
183         }
184     }
185 
186     // State switching
187     if (!p->on_ground)
188     {
189         if (p->vy < -0.1f)
190             p->state = STATE_JUMPING;
191         else if (p->vy > 0.1f)
192             p->state = STATE_FALLING;
193         else
194             p->state = STATE_IDLE; // Rarely seen motionless in air
195     }
196     else if (p->vx != 0)
197     {
198         p->state = STATE_RUNNING;
199     }
200     else
201     {
202         p->state = STATE_IDLE;
203     }
204     // if (p->type == PLAYER_WATERGIRL)
205     //     debug_print_player_state(p, p->type == "WATERGIRL");
206     return 0;
207 }
208 void adjust_to_slope_y(player_t *p)
209 {
```

```
210     // Calculate the horizontal position of the character's foot center
211     float center_x = p->x + SPRITE_W_PIXELS / 2.0f;
212
213     // Calculate the current y-coordinate of the character's foot (28
            height)
214     float base_foot_y = p->y + PLAYER_HEIGHT_PIXELS;
215
216     // Begin a small vertical search around the foot area to find if feet
            are exactly on a slope
217     // Search dy from -4 to +2, can capture small vertical errors (like
            floating or pressed in)
218     for (int dy = -4; dy <= 2; ++dy) // Empirical offset
219     {
220         float foot_y = base_foot_y + dy; // Current search point's position
                in y direction
221
222         // Get the tile type at this position
223         int tile = get_tile_at_pixel(center_x, foot_y);
224
225         // Only perform alignment processing if a slope tile is detected
226         if (tile == TILE_SLOPE_L_UP || tile == TILE_SLOPE_R_UP)
227         {
228             // Calculate the x offset of the current point within the tile
                    (i.e., top left is (0,0), how many pixels is current x
                    within the tile)
229             float x_in_tile = fmod(center_x, TILE_SIZE);
230             int x_local = (int)x_in_tile;
231
232             // Calculate the y height that this x offset should correspond
                    to in the current slope tile
233             // Left slope rises from bottom-left to top-right: the more
                    right, the higher: y = x
234             // Right slope rises from bottom-right to top-left: the more
                    left, the higher: y = TILE_SIZE - 1 - x
235             int min_y = (tile == TILE_SLOPE_L_UP)
236                             ? x_local
237                             : TILE_SIZE - 1 - x_local;
238
239
240             float tile_top_y = ((int)(foot_y / TILE_SIZE)) * TILE_SIZE;
241
242             // // Set the character's y coordinate:
243             // // - tile_top_y + min_y: gets the y height of the slope
                    surface
244             // // - minus PLAYER_HEIGHT_PIXELS: let the character stand on
                    the slope surface
245             // // - minus 3: an empirical offset for fine adjustment (can
                    be debugged)
246             // p->y = tile_top_y + min_y - PLAYER_HEIGHT_PIXELS - 3;
247
248             // // Set character grounded state
249             // p->on_ground = true;
250             // // Stop vertical velocity (won't continue falling or rising)
251             // p->vy = 0;
252
253             // // Exit search after successful processing
254             // break;
255             // Record old y
256             float old_y = p->y;
257
258             // Calculate attachment target y
259             float new_y = tile_top_y + min_y - PLAYER_HEIGHT_PIXELS - 3; //
                    Empirical offset
260
261             // If the y difference before and after is very small, keep the
                    original vy to avoid animation judgment being disrupted
```

```
262             if (fabsf(new_y - old_y) < 0.2f)
263             {
264                 p->y = new_y;
265                 // Preserve vy, don't force set to 0
266             }
267             else
268             {
269                 p->y = new_y;
270                 p->vy = 0; // Only reset to zero when there's significant
                        attachment
271             }
272 
273             p->on_ground = true;
274             break;
275         }
276     }
277 }
278 
279 void adjust_to_platform_y(player_t *p)
280 {
281     // Calculate character's foot center horizontal position
282     float foot_center_x = p->x + SPRITE_W_PIXELS / 2.0f;
283 
284     // Calculate character's current foot y coordinate (28 height)
285     float base_foot_y = p->y + PLAYER_HITBOX_OFFSET_Y +
            PLAYER_HITBOX_HEIGHT;
286 
287     // Begin a small vertical search around the foot area to find if feet
            are exactly on a slope
288     // Search dy from -4 to +2, can capture small vertical errors (like
            floating or pressed in)
289     for (int dy = -4; dy <= 2; ++dy) // Empirical offset values
290     {
291         float foot_y = base_foot_y + dy; // Current search point's position
                in y direction
292 
293         // Get the tile type at this position
294         int tile = get_tile_at_pixel(foot_center_x, foot_y);
295 
296         if (tile == 1) // Platform tile
297         {
298             float tile_top_y = ((int)(foot_y / TILE_SIZE)) * TILE_SIZE;
299             float new_y = tile_top_y - PLAYER_HEIGHT_PIXELS - 1; //
                    Empirical offset
300 
301             float old_y = p->y;
302 
303             if (fabsf(new_y - old_y) < 0.2f)
304             {
305                 p->y = new_y;
306                 // Preserve vy
307             }
308             else
309             {
310                 p->y = new_y;
311                 p->vy = 0;
312             }
313 
314             p->on_ground = true;
315             break;
316         }
317     }
318 }
319 // Fireboy
320 #define FB_HEAD_IDLE ((uint8_t)0)       // 0x0000 >> 8 = 0
321 #define FB_HEAD_WALK ((uint8_t)2)       // 0x0200 >> 8 = 2
```

```
322 #define FB_HEAD_UPDOWN ((uint8_t)7)    // 0x0700 >> 8 = 7
323 #define FB_HEAD_DOWNWALK ((uint8_t)12) // 0x0C00 >> 8 = 12
324
325 #define FB_LEG_IDLE ((uint8_t)17)         // 0x1100 >> 8 = 17
326 #define FB_LEG_WALK ((uint8_t)18)         // 0x1200 >> 8 = 18
327 #define FB_LEG_UPorDOWNWALK ((uint8_t)21) // 0x1500 >> 8 = 21
328
329 // Watergirl
330 #define WG_HEAD_IDLE ((uint8_t)22)     // 0x1600 >> 8 = 22
331 #define WG_HEAD_WALK ((uint8_t)24)     // 0x1800 >> 8 = 24
332 #define WG_HEAD_UPWALK ((uint8_t)29)   // 0x2100 >> 8 = 29
333 #define WG_HEAD_DOWNWALK ((uint8_t)34) // 0x2200 >> 8 = 34
334
335 #define WG_LEG_IDLE ((uint8_t)39)         // 0x2700 >> 8 = 39
336 #define WG_LEG_WALK ((uint8_t)40)         // 0x2800 >> 8 = 40
337 #define WG_LEG_UPorDOWNWALK ((uint8_t)43) // 0x2B00 >> 8 = 43
338
339 // Called each frame: automatically cycles between walking frames
340 static int get_frame_id(player_t *p, bool is_upper)
341 {
342     int base = 0;
343
344     if (p->type == PLAYER_FIREBOY)
345     {
346         if (p->state == STATE_IDLE)
347             base = is_upper ? FB_HEAD_IDLE : FB_LEG_IDLE;
348         else if (p->state == STATE_RUNNING)
349             base = is_upper ? FB_HEAD_WALK : FB_LEG_WALK;
350         else if (p->state == STATE_JUMPING)
351             base = is_upper ? FB_HEAD_UPDOWN : FB_LEG_UPorDOWNWALK;
352         else if (p->state == STATE_FALLING)
353             base = is_upper ? FB_HEAD_DOWNWALK : FB_LEG_UPorDOWNWALK;
354     }
355     else // WATERGIRL
356     {
357         if (p->state == STATE_IDLE)
358             base = is_upper ? WG_HEAD_IDLE : WG_LEG_IDLE;
359         else if (p->state == STATE_RUNNING)
360             base = is_upper ? WG_HEAD_WALK : WG_LEG_WALK;
361         else if (p->state == STATE_JUMPING)
362             base = is_upper ? WG_HEAD_UPWALK : WG_LEG_UPorDOWNWALK;
363         else if (p->state == STATE_FALLING)
364             base = is_upper ? WG_HEAD_DOWNWALK : WG_LEG_UPorDOWNWALK;
365     }
366
367     int frame_count = get_frame_count(p, is_upper);
368     return base + (p->frame_index % frame_count);
369 }
370
371 int get_frame_count(player_t *p, bool is_upper)
372 {
373     if (p->state == STATE_RUNNING)
374         return is_upper ? 5 : 3;
375
376     if (p->type == PLAYER_FIREBOY)
377     {
378         if (p->state == STATE_IDLE)
379             return is_upper ? 2 : 1;
380         else if (p->state == STATE_JUMPING || p->state == STATE_FALLING)
381             return is_upper ? 5 : 3;
382     }
383     else
384     {
385         if (p->state == STATE_IDLE)
386             return is_upper ? 2 : 1;
387         else if (p->state == STATE_JUMPING || p->state == STATE_FALLING)
```

```
388             return is_upper ? 5 : 3;
389     }
390
391     return 1; // fallback
392 }
393
394 void player_update_sprite(player_t *p)
395 {
396     // Determine if animation is needed
397     bool animate = false;
398
399     switch (p->state)
400     {
401     case STATE_RUNNING:
402     case STATE_IDLE:
403         animate = true; // Both walking and standing can cycle through
                animations (like blinking)
404         break;
405     case STATE_JUMPING:
406     case STATE_FALLING:
407     default:
408         animate = false; // No animation in air or unknown states
409         break;
410     }
411
412     if (animate)
413     {
414         p->frame_timer++;
415         if (p->frame_timer >= MAX_FRAME_TIMER)
416         {
417             p->frame_timer = 0;
418             int frame_count = get_frame_count(p, true); // Upper body
                    determines frame length
419             p->frame_index = (p->frame_index + 1) % frame_count;
420         }
421     }
422     else
423     {
424         p->frame_index = 0;
425     }
426
427     if (p->type == PLAYER_FIREBOY)
428     {
429         // Set frame ID
430         p->lower_sprite.frame_id = get_frame_id(p, false);
431         p->upper_sprite.frame_id = get_frame_id(p, true);
432
433         // Set position and enable
434         // Body
435         p->lower_sprite.x = p->x;
436         p->lower_sprite.y = p->y + SPRITE_H_PIXELS - 1;
437         p->lower_sprite.enable = true;
438         // Head
439         p->upper_sprite.x = p->x;
440         p->upper_sprite.y = p->y + 5;
441         p->upper_sprite.enable = true;
442     }
443     if (p->type == PLAYER_WATERGIRL)
444     { // Set frame ID
445         p->lower_sprite.frame_id = get_frame_id(p, false);
446         p->upper_sprite.frame_id = get_frame_id(p, true);
447
448         // Set position and enable
449         // Body
450         p->lower_sprite.x = p->x;
451         p->lower_sprite.y = p->y + SPRITE_H_PIXELS - 2;
```

```
452         p->lower_sprite.enable = true;
453         // Head
454         p->upper_sprite.x = p->x;
455         p->upper_sprite.y = p->y + 4;
456         p->upper_sprite.enable = true;
457     }
458 
459     sprite_update(&p->lower_sprite);
460     sprite_update(&p->upper_sprite);
461 }
```

Listing 14: player.c

```
1  #include "sprite.h"
2  #include "hw_interact.h"
3  #include <math.h> // Math library for floating point operations
4  #include "type.h"
5  #include <stdio.h>
6  
7  #define BOX_PUSH_SPEED 0.5f
8  #define BOX_FRICTION 0.2f
9  
10 extern box_t boxes[NUM_BOXES];
11 
12 void sprite_set(sprite_t *s, uint8_t index, uint8_t frame_count)
13 {
14     s->index = index;
15     s->x = s->y = 0;
16     s->frame_id = 0;
17     s->flip = 0;
18     s->enable = false;
19     s->frame_count = frame_count;
20 }
21 
22 void sprite_animate(sprite_t *s)
23 {
24     if (s->frame_count > 0)
25     {
26         uint8_t rel = (s->frame_id - s->frame_start + 1) % s->frame_count;
27         s->frame_id = s->frame_start + rel;
28     }
29 }
30 
31 void sprite_update(sprite_t *s)
32 {
33     write_sprite(s->index, s->enable, s->flip, s->x, s->y, s->frame_id);
34 }
35 
36 void sprite_clear(sprite_t *s)
37 {
38     s->enable = false;
39     sprite_update(s);
40 }
41 
42 void item_init(item_t *item, float x, float y, uint8_t sprite_index,
    uint8_t frame_id)
43 {
44     item->x = x;
45     item->y = y;
46     item->width = 16;
47     item->height = 16;
48     item->active = true;
49     sprite_set(&item->sprite, sprite_index, 0);
50     item->sprite.x = (uint16_t)x;
51     item->sprite.y = (uint16_t)y;
52     item->sprite.frame_id = frame_id;
```

```
53      item->sprite.enable = true;
54      sprite_update(&item->sprite);
55  }
56
57  void item_update_sprite(item_t *item)
58  {
59      if (item->active)
60      {
61          float offset = 0.0f;
62          if (item->float_anim)
63          {
64              offset = 0.01f * sinf((float)frame_counter * 0.1f + item->
                    sprite.index); // Different amplitude and frequency for
                    floating animation
65          }
66
67          item->sprite.x = (uint16_t)item->x;
68          item->sprite.y = (uint16_t)(item->y + offset);
69          item->sprite.enable = true;
70          sprite_update(&item->sprite);
71      }
72      else
73      {
74          sprite_clear(&item->sprite);
75      }
76  }
77
78  void box_init(box_t *b, int tile_x, int tile_y, int sprite_base_index,
        uint8_t frame_id)
79
80  {
81      b->x = tile_x * 16;
82      b->y = tile_y * 16;
83      b->vx = 0;
84      b->active = true;
85
86      for (int i = 0; i < 4; i++)
87      {
88          sprite_set(&b->sprites[i], sprite_base_index + i, 1);
89          b->sprites[i].frame_id = frame_id + i;
90          b->sprites[i].enable = true;
91      }
92  }
93
94  void box_update_sprite(box_t *b)
95  {
96      float x = b->x;
97      float y = b->y;
98
99      b->sprites[0].x = x;
100     b->sprites[0].y = y + 1;
101
102     b->sprites[1].x = x + 15;
103     b->sprites[1].y = y + 1;
104
105     b->sprites[2].x = x;
106     b->sprites[2].y = y + 16;
107
108     b->sprites[3].x = x + 15;
109     b->sprites[3].y = y + 16;
110
111     for (int i = 0; i < 4; i++)
112     {
113         sprite_update(&b->sprites[i]);
114     }
115 }
```

```
116
117 void box_try_push(box_t *box, const player_t *p)
118 {
119     float pw = SPRITE_W_PIXELS;
120     float ph = PLAYER_HITBOX_HEIGHT;
121     float px = p->x;
122     float py = p->y + PLAYER_HITBOX_OFFSET_Y;
123
124     float bw = 32.0f;
125     float bh = 32.0f;
126     float bx = box->x;
127     float by = box->y;
128
129     bool vertical_overlap = (py + ph > by) && (py < by + bh);
130     if (!vertical_overlap)
131     {
132         return;
133     }
134
135     float p_center_x = px + pw / 2.0f;
136     float b_left = bx;
137     float b_right = bx + bw;
138     const float PUSH_TOLERANCE = 5.0f; // Expanded to 5 pixel range
139
140     if ((fabsf(p_center_x - b_left) <= PUSH_TOLERANCE) && p->vx > 0)
141     {
142         box->vx = BOX_PUSH_SPEED;
143     }
144     else if ((fabsf(p_center_x - b_right) <= PUSH_TOLERANCE) && p->vx < 0)
145     {
146         box->vx = -BOX_PUSH_SPEED;
147     }
148 }
149 void box_update_position(box_t *box, player_t *players)
150 {
151     float next_x = box->x + box->vx;
152
153     bool blocked = false;
154     if (box->vx > 0)
155         blocked |= is_tile_blocked(next_x + 31, box->y + 2, 1, 28);
156     else if (box->vx < 0)
157         blocked |= is_tile_blocked(next_x + 1, box->y + 2, 1, 28);
158
159     bool will_overlap_non_pusher = false;
160
161     for (int i = 0; i < NUM_PLAYERS; i++)
162     {
163         float px = players[i].x;
164         float py = players[i].y + PLAYER_HITBOX_OFFSET_Y;
165         float pw = SPRITE_W_PIXELS;
166         float ph = PLAYER_HITBOX_HEIGHT;
167
168         float p_center_x = px + pw / 2.0f;
169         bool vertical_overlap = (py + ph > box->y) && (py < box->y + 32);
170         if (!vertical_overlap)
171             continue;
172
173         // Check if player is at edge and pushing the box (overlap allowed)
174         bool is_pusher = false;
175         if (box->vx > 0 && fabsf(p_center_x - box->x) <= 10.0f && players[i
            ].vx > 0)
176             is_pusher = true;
177         else if (box->vx < 0 && fabsf(p_center_x - (box->x + 32)) <= 10.0f
            && players[i].vx < 0)
178             is_pusher = true;
179
```

```
180         // Non-pusher that will be overlapped, prevent movement
181         if (!is_pusher && check_overlap(next_x + 2, box->y + 2, 28, 28, px
                + SPRITE_W_PIXELS / 2.0f, py + PLAYER_HITBOX_OFFSET_Y, 1.0f,
                PLAYER_HITBOX_HEIGHT))
182         {
183             will_overlap_non_pusher = true;
184             break;
185         }
186     }
187     if (!blocked && !will_overlap_non_pusher)
188     {
189         box->x = next_x;
190     }
191     if (box->vx > 0)
192         box->vx -= BOX_FRICTION;
193     else if (box->vx < 0)
194         box->vx += BOX_FRICTION;
195     if (fabsf(box->vx) < BOX_FRICTION)
196         box->vx = 0;
197 }
198
199 bool is_box_blocked(float x, float y, float w, float h)
200 {
201     for (int i = 0; i < NUM_BOXES; i++)
202     {
203         if (!boxes[i].active)
204             continue;
205
206         float bx = boxes[i].x;
207         float by = boxes[i].y;
208
209         if (check_overlap(x, y, w, h, bx + 2, by + 2, 28, 28))
210         {
211             return true;
212         }
213     }
214
215     return false;
216 }
217
218 bool check_overlap(float x1, float y1, float w1, float h1,
219                    float x2, float y2, float w2, float h2)
220 {
221     return (x1 < x2 + w2) && (x1 + w1 > x2) &&
222            (y1 < y2 + h2) && (y1 + h1 > y2);
223 }
224
225 void lever_init(lever_t *lvr, float tile_x, float tile_y, uint8_t
    sprite_index_base)
226 {
227     lvr->x = tile_x * 16;
228     lvr->y = tile_y * 16;
229     lvr->activated = false;
230     lvr->sprite_base_index = sprite_index_base;
231
232     // Frame resource definitions
233     lvr->base_frame[0] = LEVER_BASE_FRAME + 0;
234     lvr->base_frame[1] = LEVER_BASE_FRAME + 1;
235     lvr->handle_frames[0] = LEVER_ANIM_FRAME + 0; // Left
236     lvr->handle_frames[1] = LEVER_ANIM_FRAME + 1; // Middle
237     lvr->handle_frames[2] = LEVER_ANIM_FRAME + 2; // Right
238
239     // Set up base sprites (2 tiles)
240     for (int i = 0; i < 2; ++i)
241     {
242         sprite_set(&lvr->base_sprites[i], sprite_index_base + i, 0);
```

```
243         lvr->base_sprites[i].x = (uint16_t)(lvr->x + i * 16);
244         lvr->base_sprites[i].y = (uint16_t)(lvr->y - 4);
245         lvr->base_sprites[i].frame_id = lvr->base_frame[i];
246         lvr->base_sprites[i].enable = true;
247         sprite_update(&lvr->base_sprites[i]);
248     }
249 
250     // Set up lever handle
251     sprite_set(&lvr->handle_sprite_left, sprite_index_base + 2, 0);
252     lvr->handle_sprite_left.x = (uint16_t)(lvr->x + 5);
253     lvr->handle_sprite_left.y = (uint16_t)(lvr->y - 16);
254     lvr->handle_sprite_left.frame_id = lvr->handle_frames[1]; // Middle
        frame
255     lvr->handle_sprite_left.enable = false;
256     sprite_update(&lvr->handle_sprite_left);
257     // Set up lever handle
258     sprite_set(&lvr->handle_sprite_right, sprite_index_base + 3, 0);
259     lvr->handle_sprite_right.x = (uint16_t)(lvr->x + 13);
260     lvr->handle_sprite_right.y = (uint16_t)(lvr->y - 16);
261     lvr->handle_sprite_right.frame_id = lvr->handle_frames[2]; // Middle
        frame
262     lvr->handle_sprite_right.enable = true;
263     sprite_update(&lvr->handle_sprite_right);
264 }
265 
266 void lever_update(lever_t *lvr, const player_t *players)
267 {
268     for (int i = 0; i < NUM_PLAYERS; ++i)
269     {
270         const player_t *p = &players[i];
271         float px = p->x + SPRITE_W_PIXELS / 2.0f;
272         float py = p->y + 32;
273 
274         if (fabsf(py - lvr->y) > 12.0f)
275             continue;
276 
277         // Current position is right (false), player moves from right to
            left to switch to left position
278         if (!lvr->activated && px >= lvr->x + 20 && px <= lvr->x + 28 && p
            ->vx < -0.3f)
279         {
280             lvr->activated = true;
281             lvr->handle_sprite_left.enable = true;
282             lvr->handle_sprite_right.enable = false;
283             sprite_update(&lvr->handle_sprite_left);
284             sprite_update(&lvr->handle_sprite_right);
285             break;
286         }
287 
288         // Current position is left (true), player moves from left to right
             to switch to right position
289         if (lvr->activated && px >= lvr->x + 4 && px <= lvr->x + 12 && p->
            vx > 0.3f)
290         {
291             lvr->activated = false;
292             lvr->handle_sprite_left.enable = false;
293             lvr->handle_sprite_right.enable = true;
294             sprite_update(&lvr->handle_sprite_left);
295             sprite_update(&lvr->handle_sprite_right);
296             break;
297         }
298     }
299 }
300 
301 void elevator_init(elevator_t *elv, float tile_x, float tile_y, float
    min_tile_y, float max_tile_y, uint8_t sprite_index_base, uint8_t
```

```
    frame_index)
{
    float x = tile_x * 16;
    float y = tile_y * 16;

    elv->x = x;
    elv->y = y;
    elv->min_y = min_tile_y * 16;
    elv->max_y = max_tile_y * 16;
    elv->vy = 0.0f;
    elv->moving_up = true;
    elv->active = false;
    elv->sprite_base_index = sprite_index_base;

    // Left block
    sprite_set(&elv->sprites[0], sprite_index_base + 0, 0);
    elv->sprites[0].x = (uint16_t)(x + 1);
    elv->sprites[0].y = (uint16_t)(y);
    elv->sprites[0].frame_id = frame_index + 0;
    elv->sprites[0].enable = true;
    sprite_update(&elv->sprites[0]);

    // Left middle block
    sprite_set(&elv->sprites[1], sprite_index_base + 1, 0);
    elv->sprites[1].x = (uint16_t)(x + 16);
    elv->sprites[1].y = (uint16_t)(y);
    elv->sprites[1].frame_id = frame_index + 1;
    elv->sprites[1].enable = true;
    sprite_update(&elv->sprites[1]);

    // Right middle block
    sprite_set(&elv->sprites[2], sprite_index_base + 2, 0);
    elv->sprites[2].x = (uint16_t)(x + 32);
    elv->sprites[2].y = (uint16_t)(y);
    elv->sprites[2].frame_id = frame_index + 2;
    elv->sprites[2].enable = true;
    sprite_update(&elv->sprites[2]);

    // Right block
    sprite_set(&elv->sprites[3], sprite_index_base + 3, 0);
    elv->sprites[3].x = (uint16_t)(x + 47);
    elv->sprites[3].y = (uint16_t)(y);
    elv->sprites[3].frame_id = frame_index + 3;
    elv->sprites[3].enable = true;
    sprite_update(&elv->sprites[3]);
}
bool is_elevator_blocked(float x, float y, float w, float h, float *vy_out)
{
    for (int i = 0; i < NUM_ELEVATORS; i++)
    {
        elevator_t *elv = &elevators[i];

        for (int j = 0; j < 4; j++)
        {
            sprite_t *s = &elv->sprites[j];

            float ex = s->x;
            float ey = s->y;

            if (check_overlap(x, y, w, h, ex, ey, 16, 16))
            {
                if (vy_out)
                    *vy_out = elv->vy; // Return elevator vertical speed (
                        for synchronization)
                return true;
            }
```

```
366         }
367     }
368     return false;
369 }
370 void elevator_update(elevator_t *elv, bool go_up, player_t *players)
371 {
372     // Determine target direction
373     if (!go_up)
374     {
375         if (elv->y > elv->min_y)
376         {
377             elv->vy = -0.2f;
378         }
379         else
380         {
381             elv->y = elv->min_y;
382             elv->vy = 0;
383         }
384     }
385     else
386     {
387         if (elv->y < elv->max_y)
388         {
389             elv->vy = 0.2f;
390         }
391         else
392         {
393             elv->y = elv->max_y;
394             elv->vy = 0;
395         }
396     }
397
398     // WARNING: Predict if next position will collide with player before
        moving
399     if (elv->vy > 0.0f)
400     {
401         float next_y = elv->y + elv->vy + 6;
402         bool will_collide_with_player = false;
403
404         for (int i = 0; i < NUM_PLAYERS; ++i)
405         {
406             const player_t *p = &players[i];
407             float px = p->x + SPRITE_W_PIXELS / 2.0f;
408             float py = p->y + PLAYER_HITBOX_OFFSET_Y;
409
410             if (px >= elv->x && px <= elv->x + 64 &&
411                 check_overlap(px, py, 1.0f, PLAYER_HITBOX_HEIGHT,
412                               elv->x + 1, next_y + 8.0, 62.0f, 1.0f))
413             {
414                 will_collide_with_player = true;
415                 break;
416             }
417         }
418
419         if (will_collide_with_player)
420         {
421             elv->vy = 0;
422             return;
423         }
424     }
425     // Apply movement
426     elv->y += elv->vy;
427
428     for (int i = 0; i < 4; ++i)
429     {
430         elv->sprites[i].y = (uint16_t)(elv->y);
```

```
431         sprite_update(&elv->sprites[i]);
432     }
433
434     // Player movement synchronization
435     for (int i = 0; i < NUM_PLAYERS; ++i)
436     {
437         player_t *p = &players[i];
438         float px = p->x + SPRITE_W_PIXELS / 2.0f;
439         float foot_y = p->y + PLAYER_HEIGHT_PIXELS;
440
441         if (px >= elv->x && px <= elv->x + 64 &&
442             fabsf(foot_y - elv->y) < 4.0f)
443         {
444             p->y += elv->vy;
445         }
446     }
447 }
448
449 void button_init(button_t *btn, float tile_x, float tile_y, uint8_t
        sprite_index_base)
450 {
451     float x = tile_x * 16;
452     float y = tile_y * 16 - 16; // Top position of button sprite's upper-
            left corner
453
454     btn->x = x;
455     btn->y = y;
456     btn->pressed = false;
457     btn->sprite_index_base = sprite_index_base;
458
459     // Fixed frame numbers
460     btn->frame_top = BUTTON_PURPLE_FRAME;         // 55
461     btn->frame_base_left = LEVER_BASE_FRAME;      // 57
462     btn->frame_base_right = LEVER_BASE_FRAME + 1; // 60
463
464     // Upper button
465     sprite_set(&btn->top_sprite, sprite_index_base + 0, 0);
466     btn->top_sprite.x = (uint16_t)x;
467     btn->top_sprite.y = (uint16_t)y + 2;
468     btn->top_sprite.frame_id = btn->frame_top;
469     btn->top_sprite.enable = true;
470     sprite_update(&btn->top_sprite);
471
472     // Left base
473     sprite_set(&btn->base_left_sprite, sprite_index_base + 1, 0);
474     btn->base_left_sprite.x = (uint16_t)x - 8;
475     btn->base_left_sprite.y = (uint16_t)(y + 13);
476     btn->base_left_sprite.frame_id = btn->frame_base_left;
477     btn->base_left_sprite.enable = true;
478     sprite_update(&btn->base_left_sprite);
479
480     // Right base
481     sprite_set(&btn->base_right_sprite, sprite_index_base + 2, 0);
482     btn->base_right_sprite.x = (uint16_t)(x + 7);
483     btn->base_right_sprite.y = (uint16_t)(y + 13);
484     btn->base_right_sprite.frame_id = btn->frame_base_right;
485     btn->base_right_sprite.enable = true;
486     sprite_update(&btn->base_right_sprite);
487 }
488
489 void button_update(button_t *btn, const player_t *players)
490 {
491     btn->pressed = false;
492     float max_depth = 0.0f;
493
494     for (int i = 0; i < NUM_PLAYERS; ++i)
```

```c
    {
        float px_center = players[i].x + SPRITE_W_PIXELS / 2.0f;
        float foot_y = players[i].y + PLAYER_HEIGHT_PIXELS - 15;
        // Player must be within button area horizontally
        if (px_center >= btn->x && px_center <= btn->x + 16)
        {
            // Vertical distance must be close to button top (ground level)
            if (fabsf(foot_y - btn->y) <= 4.0f)
            {
                float dx = fabsf(px_center - (btn->x + 8.0f)); // Center
                    offset
                float depth = 8.0f - dx; // Depression value: maximum 8px
                if (depth > max_depth)
                    max_depth = depth;

                if (dx <= 5.0f) // WARNING: Center 3 pixels -> total 6px
                    btn->pressed = true;
            }
        }
    }

    btn->press_offset = max_depth;

    // Visual sprite downward movement
    btn->top_sprite.y = (uint16_t)(btn->y + 2 + btn->press_offset);
    sprite_update(&btn->top_sprite);
}
```

Listing 15: sprite.c

```c
// tilemap.c
// Terrain map data and tile collision detection implementation
#include "tilemap.h"
#include <math.h> // For floor()
#include "player.h"
#include "sprite.h" // If missing, include for item_t definition
#include "type.h"
#include <stdio.h>
// === Example map data ===
// 0: Empty  1: Wall  2: Fire pit  3: Water pool  4: Goal
const int tilemap[30][40] = {
    //                x                1                5                2
                     5                3                5                4
    {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1}, //
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6, 1, 1, 5, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 10, 0, 9, 0, 0, 1},
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 1, 1, 1, 5, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 10, 0, 9, 0, 0, 1},
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
    {1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
    {1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1}, // 1
    {1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1,
        1, 1, 1, 1, 1, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
    {1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1,
```

```
        1, 1, 1, 1, 1, 1, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
    {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1},
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 7, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1},
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1}, //
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
    {1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 5, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 1, 1, 1, 1, 1, 1, 1},
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 1, 1, 1, 1, 1}, // 2
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 1, 1, 1, 1},
    {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 5, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 1, 1, 5, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 1, 1, 1, 1, 1, 1,
        1, 4, 4, 4, 4, 4, 1, 1, 1, 1, 1, 5, 0, 0, 0, 0, 1},
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1}, //
    {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6, 1, 1, 1},
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1},
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1},
    {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2,
        2, 1, 1, 1, 3, 3, 3, 3, 3, 1, 1, 1, 1, 1, 1, 1, 1}};

#define COLLISION_MARGIN 1.0f

bool is_tile_blocked(float x, float y, float width, float height)
{
    float center_x = x + width / 2.0f;

    for (int i = PLAYER_HITBOX_OFFSET_Y; i < (int)height; ++i)
    {
        float sx = center_x;
        float sy = y + i + COLLISION_MARGIN;

        int tx = (int)(sx / TILE_SIZE);
        int ty = (int)(sy / TILE_SIZE);

        if (tx < 0 || tx >= MAP_WIDTH || ty < 0 || ty >= MAP_HEIGHT)
            return true;

        int tile = tilemap[ty][tx];

        // Normal wall
        if (tile == TILE_WALL)
            return true;

        if (tile == TILE_WATER || tile == TILE_POISON || tile == TILE_FIRE)
        {
            float y_in_tile = fmod(sy, TILE_SIZE);
            if (y_in_tile >= 8.0f)
                return true;
```

```
72          }
73          // Sloped ceiling handling (character head collision)
74          if (tile == TILE_CEIL_L || tile == TILE_CEIL_R)
75          {
76              float x_in_tile = fmod(sx, TILE_SIZE);
77              float y_in_tile = fmod(sy, TILE_SIZE);
78
79              int x_local = (int)x_in_tile;
80              int y_local = (int)y_in_tile;
81
82              int max_y = (tile == TILE_CEIL_L)
83                              ? TILE_SIZE - 1 - x_local // Left low, right
                                    high
84                              : x_local;               // Right low, left
                                    high
85
86              if (y_local <= max_y)
87                  return true;
88          }
89
90          // Sloped floor handling (character foot collision)
91          if (tile == TILE_SLOPE_L_UP || tile == TILE_SLOPE_R_UP)
92          {
93              float x_in_tile = fmod(sx, TILE_SIZE);
94              float y_in_tile = fmod(sy, TILE_SIZE);
95
96              int x_local = (int)x_in_tile;
97              int y_local = (int)y_in_tile;
98
99              int min_y = (tile == TILE_SLOPE_L_UP)
100                             ? x_local                  // \ Left high,
                                    right low
101                             : TILE_SIZE - 1 - x_local; // /  Right high,
                                    left low
102
103             if (y_local >= min_y)
104                 return true;
105         }
106     }
107
108     return false;
109 }
110 int get_tile_at_pixel(float x, float y)
111 {
112     int tx = (int)(x / TILE_SIZE);
113     int ty = (int)(y / TILE_SIZE);
114
115     if (tx < 0 || tx >= MAP_WIDTH || ty < 0 || ty >= MAP_HEIGHT)
116         return TILE_WALL; // Treat out of bounds as wall
117
118     return tilemap[ty][tx];
119 }
120
121 void item_place_on_tile(item_t *item, int tile_x, int tile_y)
122 {
123     item->x = tile_x * TILE_SIZE + (TILE_SIZE - item->width) / 2.0f;
124     item->y = tile_y * TILE_SIZE + (TILE_SIZE - item->height) / 2.0f;
125
126     item->sprite.x = (uint16_t)item->x;
127     item->sprite.y = (uint16_t)item->y;
128 }
129
130 bool is_death(float x, float y, float width, float height, player_type_t p)
131 {
132
133     float center_x = x + width / 2.0f;
```

```
134     for (int i = PLAYER_HITBOX_OFFSET_Y; i < (int)height; ++i)
135     {
136         float sx = center_x;
137         float sy = y + i + COLLISION_MARGIN;
138 
139         int tx = (int)(sx / TILE_SIZE);
140         int ty = (int)(sy / TILE_SIZE);
141 
142         if (tx < 0 || tx >= MAP_WIDTH || ty < 0 || ty >= MAP_HEIGHT)
143             return false;
144 
145         int tile = tilemap[ty][tx];
146 
147         // Dangerous terrain detection (death)
148         if (p == PLAYER_FIREBOY && (tile == TILE_WATER || tile ==
                TILE_POISON))
149             return true;
150         if (p == PLAYER_WATERGIRL && (tile == TILE_FIRE || tile ==
                TILE_POISON))
151             return true;
152     }
153     return false;
154 }
155 
156 bool check_both_players_goal()
157 {
158     bool fireboy_goal = false;
159     bool watergirl_goal = false;
160 
161     for (int i = 0; i < NUM_PLAYERS; ++i)
162     {
163         float center_x = players[i].x + SPRITE_W_PIXELS / 2.0f;
164         float center_y = players[i].y + SPRITE_H_PIXELS / 2.0f;
165         int tile = get_tile_at_pixel(center_x, center_y);
166 
167         if (players[i].type == PLAYER_FIREBOY && tile == TILE_GOAL2)
168             fireboy_goal = true;
169         else if (players[i].type == PLAYER_WATERGIRL && tile == TILE_GOAL1)
170             watergirl_goal = true;
171     }
172 
173     return fireboy_goal && watergirl_goal;
174 }
```

Listing 16: tilemap.c