CSEE4840 Embedded Systems

Final Report: Geometry Dash

Riju Dey, Rachinta Marpaung, Charles Chen, Sasha Isler

Columbia University | Spring 2024

Table of Contents

Introduction

Geometry Dash is a platformer game known for its rhythmic gameplay (Figure 1). The game challenges players to navigate a geometric sprite through a scrolling obstacle course, synchronized background music.

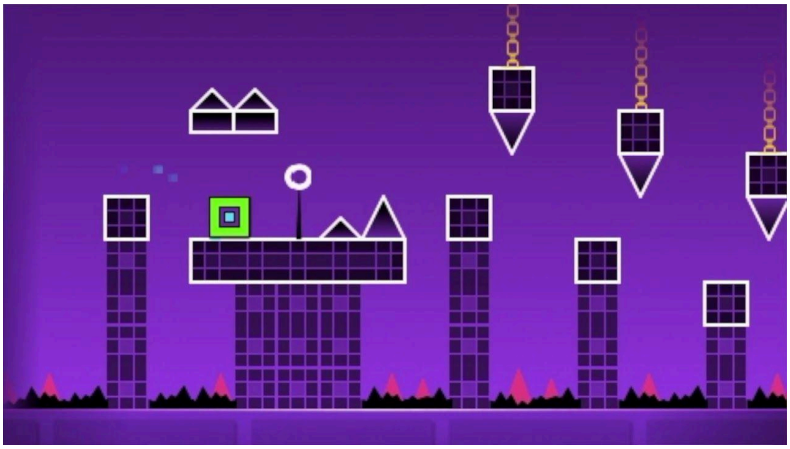

Figure 1: Geometry Dash interface

For our final project, we aimed to implement a simplified version of Geometry Dash on the DE1-SoC FPGA development board. Our system aimed to recreate core features of the game— including continuous scrolling, obstacle generation, collision detection, and player controlled jumping— using hardware and software integration. Graphics are rendered through a video graphics array (VGA) display using a custom tile-based rendering system, while game logic is driven by a finite state machine. A connected joypad is used for player input, replicating the original tap-to-jump interaction in a responsive way. To preserve the game's rhythm-centric feel, we attempted to incorporate MIDI-based audio.

System Overview

The game implementation is organized into three main categories: software, hardware, and external peripherals. Figure 2 shows a high-level block diagram of the full system. The USB joypad acts as the primary external input device, communicating controller data to the software running on the HPS. A C-based control loop processes player input, manages game state, and issues commands to hardware modules through a custom kernel driver. This software communicates with the FPGA over the Avalon bus, writing key variables such as player position and scroll offset into memory-mapped registers.

On the hardware side, a VGA rendering engine fetches data from memory-mapped RAM blocks for the TileMap, TileSet, and Palette to display game visuals in real time on the VGA monitor. Simultaneously, a MIDI-based audio subsystem was designed with an Altera dual-clock FIFO to buffer note events from the HPS and drive square-wave signals into the WM8731 audio codec, which outputs to an external speaker.

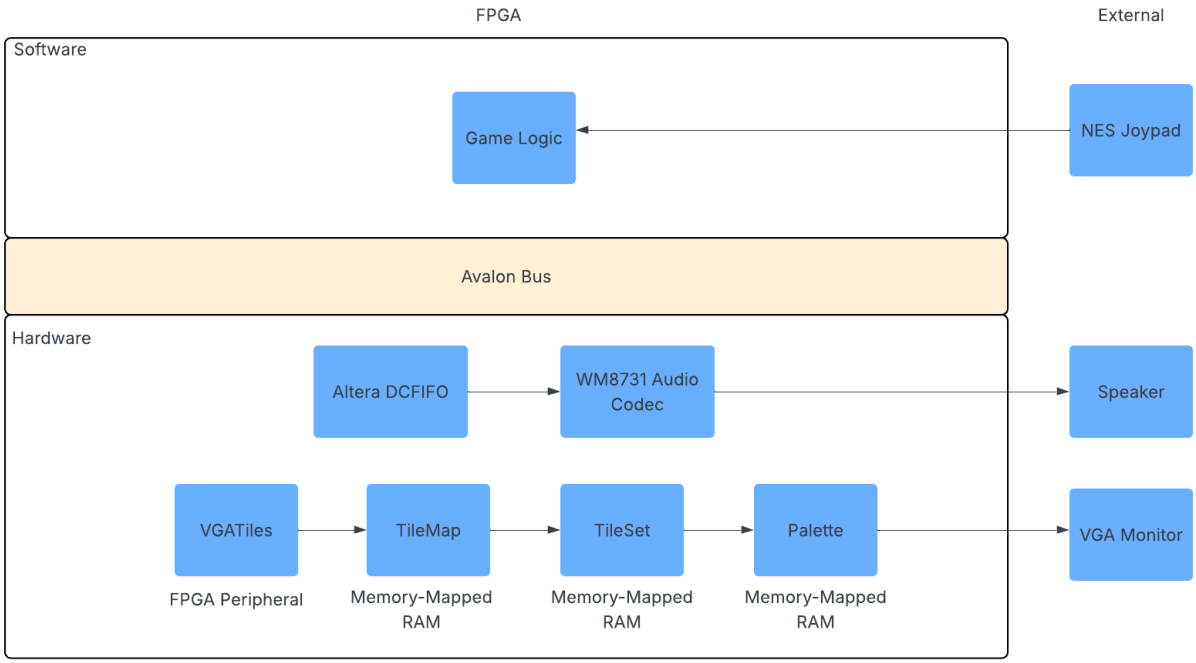

Figure 2. System block diagram

Design and Implementation

VGA Display and Map Rendering

To render the scrolling environment, we implemented a tile-based graphics system. Our tile engine divides the screen into 32×32 pixel tiles, arranged in a virtual grid. Each tile corresponds to a number in a TileMap array. This number indexes into a TileSet—a memory structure containing the pixel data for all tile types (e.g., ground, spikes, platforms). Each tile uses indexed color: a palette maps 4-bit color codes to 24-bit RGB values for output to the VGA display.

The tile rendering system uses pixel coordinates (hcount, vcount) generated by the **VGA counters**, then maps them through three steps:

- **Tile Lookup**
- **Pixel Color Code**
- **RGB Mapping**

This is fully pipelined in hardware and synchronized to the 25 MHz VGA pixel clock. Pipeline registers align the VGA sync signals (hsync, vsync) with the correct color data to ensure glitch-free video output.

Memory Structure

- **TileMap**: 4096 × 8-bit RAM (128×64 tiles)
- **TileSet**: 16384 × 4-bit RAM (256 tiles × 64 pixels)
- **Palette**: 16 × 24-bit RGB entries, 4-byte aligned

All three memories are implemented using **dual-port BRAM**, enabling independent simultaneous access by the VGA rendering logic (read-only) and the HPS (read/write via Avalon MM).

Horizontal Scrolling

To simulate player movement, we shift the visible portion of the TileMap to the left by incrementing an x_shift value each frame. Once x_shift crosses a tile boundary, the tile column index increments, creating a continuous scrolling effect. This method avoids redrawing the entire frame or modifying pixel values directly.

Software Access and Initialization

Through the Avalon Memory-Mapped (MM) interface, the HPS loads tile graphics and layout data into memory-mapped RAM regions:

- tilemap1.hex → TileMap
- tileset1.hex → TileSet
- palette1.hex → Palette

Audio Interface

We attempted to implement a hardware-based audio system that plays music using the DE1-SoC's WM8731 audio codec. Music was preprocessed in software and then written to hardware.

Music Processing

The raw music file was processed using fluidsynth and sox:

- Downloaded the .mp3 for the audio we wanted to use.
- Trimed and converted .mp3 to .wav (48kHz stereo 16-bit):

ffmpeg -ss 44 -t 124 -i Monody.mp3 -ar 48000 -ac 2 -sample_fmt s16 monody_trimmed.wav

- Convert .wav to .raw PCM (interleaved stereo, 16-bit little-endian):

ffmpeg -i monody_trimmed.wav -f s16le -acodec pcm_s16le monody_stereo_48k.raw

This .raw file contains a stream of 32-bit samples: Left channel (16 bits) + Right channel (16 bits). Because it's stereo, we read two channels (Left + Right) — so we read every other sample.

Audio Connections in Hardware

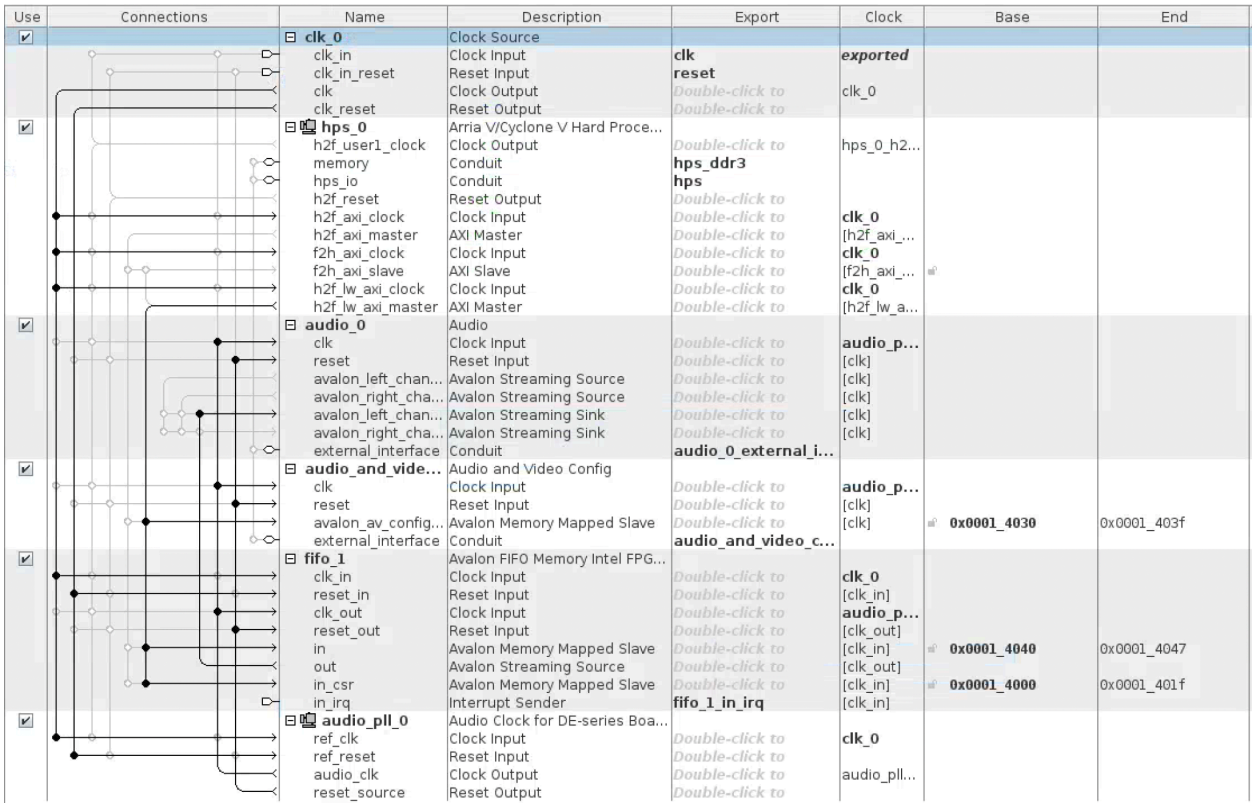

Figure 3: Platform Designer Hardware Connections

The data is written from software to a dual-clock FIFO (Altera DCFIFO) fifo_1, and then fed from fifo_1 to the audio CODEC component audio_0. The input to fifo_1 is clocked on the clk_0, and the output is clocked on the audio_pll configured to 12.288MHz which is the required frequency for the codec to operate at a sampling rate of 48kHz. The audio_0 and the audio configuration are both clocked on the audio_pll as well.

Software

We wrote a separate kernel module audio_fifo.c which looks for "ALTR,fifo-21.1" on the device tree. This module supports:

- WRITE_AUDIO_FIFO: write 16 bit data sample to the FIFO
- READ_AUDIO_FILL_LEVEL: reads the fill level from csr of the FIFO
- READ_AUDIO_STATUS: reads status from the csr of the FIFO

The audio.c userspace code attempts to initialize the audio CODEC using I2C and write data to the FIFO using the ioctl, however we were not successful at producing sound. We believe this is because the codec was not configured to receive initialization configurings over hps.

USB Joypad Controller

Player control is provided by a NES-style USB joypad. The usbjoypad.c driver initializes the device using libusb, polls for input via an interrupt transfer, and uses a shared structure (ControllerState) to reflect the current button state. This input is used to:

- Trigger jump events
- Adjust scrolling offset (x_shift) based on left/right movement
- Set game flags such as PLAYER_JUMPING or PLAYER_INVERTED

Software Design

Game Logic

Game logic is executed in a C userspace program (game_loop.c) that models player movement, jump physics, gravity inversion, and collision detection. The player's vertical velocity is governed by Newtonian motion, with obstacles like spikes, blocks, platforms, jump pads, and gravity portals influencing state transitions. The game operates through a four-state FSM (LOADING, READY, PLAYING, GAME_OVER), each with distinct responsibilities and transitions.

Game state variables (e.g., player_y, x_shift, map_block, flags) are sent to the kernel module (geo_dash.c) through ioctl() calls, triggering hardware-level updates.

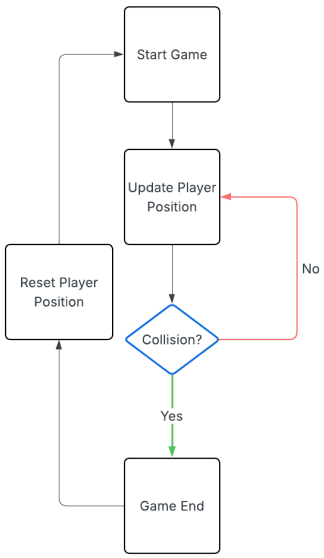

Figure 4. FSM Control Path

Figure 5. Game in Progress

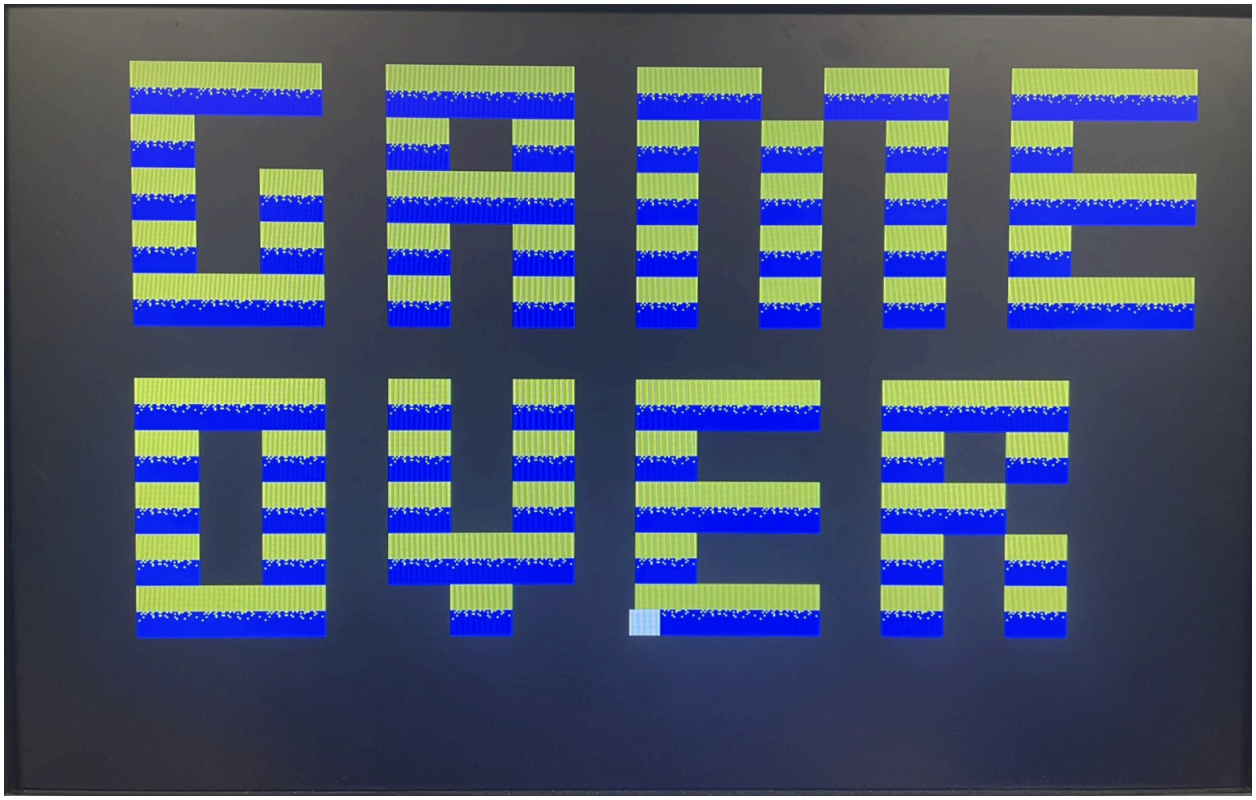

Figure 5. Game over screen

Kernel Driver

We developed a custom kernel module (geo_dash.ko) to map software-level game variables into hardware-visible memory locations using memory-mapped registers. Commands like WRITE_PLAYER_Y_POS, WRITE_ SCROLL_OFFSET, and WRITE_FLAGS are defined via ioctl codes in geo_dash.h, and handled in geo_dash_ioctl(). To control the hardware tilemaps, tilesets and palette, we implemented ioctls WRITE_TILE, WRITE_TILESET, and WRITE_PALETTE.

The driver exposes /dev/geo_dash and coordinates tightly with the FPGA to update player state, scroll position, background color gradients, and game status flags.

Control Registers

We implemented two 8-bit control registers in hardware, y_pos and scroll_offset, which control the y position of the player square and the offset of the tilemap on the vga monitor respectively.

Game Controller

The player interacts with the game using a iNNEXT joypad. The controller sends and receives over the USB connection and the software uses the libusb library to communicate. We only used button A to enable the sprite to jump.

Figure 4. iNNEXT Joypad

The joypad was configured through libusb, which follows the USB protocol. The protocol message is 8 bytes for each event.

Resource Budget

	Dimensions	Size (bits)
TileMap	512 x 8	4096
TileSet	16384 x 8	131072
Palette	16 x 24	384
Control Regs	2 x 8	16

Conclusions

Our final game successfully implemented tile rendering, level scrolling, external input button peripheral for jumping, overall game logic including physics and collision detection.

We were unsuccessful in implementing hardware scrolling and audio. In the future, we would give ourselves the advice to take more time to read about each component, and develop a solid plan to test every single component in both hardware and software.

Appendix

A. Hardware

1. twoportbram.sv

```
C/C++
module twoportbram
 #(parameter int DATA_BITS = 8,  ADDRESS_BITS = 10)
  (input logic                    clk1,  clk2,
   input logic [ADDRESS_BITS-1:0] addr1, addr2,
   input logic [DATA_BITS-1:0]    din1,  din2,
   input logic                    we1,   we2,
   output logic [DATA_BITS-1:0]   dout1, dout2);

  localparam WORDS = 1 << ADDRESS_BITS;

  /* verilator lint_off MULTIDRIVEN */
  logic [DATA_BITS-1:0]       mem [WORDS-1:0];
  /* verilator lint_on MULTIDRIVEN */

  always_ff @(posedge clk1)
    if (we1) begin
            mem[addr1] <= din1;
            dout1 <= din1;
    end else dout1 <= mem[addr1];

  always_ff @(posedge clk2)
    if (we2) begin
            mem[addr2] <= din2;
            dout2 <= din2;
    end else dout2 <= mem[addr2];

endmodule
```

2. vga_counter.sv

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

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

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

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

  always_ff @(posedge VGA_CLK or posedge VGA_RESET)
    if (VGA_RESET)     vcount <= 10'd 524;
    else if (endOfLine)
      if (endOfFrame)   vcount <= 10'd 0;
      else             vcount <= vcount + 10'd 1;

  // 656 <= hcount <= 751
  assign VGA_HS = !( hcount[9:7] == 3'b101 &
                  hcount[6:4] != 3'b000 & hcount[6:4] != 3'b111 );
  assign VGA_VS = !( vcount[9:1] == 9'd 245 ); // Lines 490 and 491

  // hcount < 640 && vcount < 480
  assign VGA_BLANK_n = !( hcount[9] & (hcount[8] | hcount[7]) ) &
                       !( vcount[9] | (vcount[8:5] == 4'b1111) );
endmodule
```

3. vga_tiles.sv

```
C/C++
/*
 * Avalon memory-mapped agent peripheral that produces a VGA tile display
 *
 * Stephen A. Edwards
 * Columbia University
 *
 * Memory map:
 *
 * 0000 - 1FFF Tilemap (8K, tile number is 8 bits per byte)
 * 2000 - 203F Palette (64, 24 bits every 4 bytes)
 * 4000 - 7FFF Tileset (16K, color index is lower 4 bits of each byte)
 *
 * 00m mmmm mmmm mmmm  Tilemap
 * 010 0000 00pp ppbb  Palette
 * 1ss ssss ssss ssss  Tileset
 *
 * In the 64-byte palette region, every color occupies 4 bytes, although
 * only 24 bits are stored.  Writing to the first 3 bytes in each group
 * writes a byte into the 24-bit color register.  Writing to the fourth
 * byte writes the color register to the palette memory; any data written to
 * these addresses is ignored; they always read 0.
 *
 * | Offset |    On Write          |  On Read          |
 * +--------+----------------------+-------------------+
 * |   0    | creg[7:0] <- data    | palette[0].red    |
 * |   1    | creg[15:8] <- data   | palette[0].green  |
 * |   2    | creg[23:16] <- data  | palette[0].blue   |
 * |   3    | palette[0] <- creg   | Always 0          |
 * |   4    | creg[7:0] <- data    | palette[1].red    |
 * |   5    | creg[15:8] <- data   | palette[1].green  |
 * |   6    | creg[23:16] <- data  | palette[1].blue   |
 * |   7    | palette[1] <- creg   | Always 0          |
 *  ...
 * |   60   | creg[7:0] <- data    | palette[15].red   |
 * |   61   | creg[15:8] <- data   | palette[15].green |
 * |   62   | creg[23:16] <- data | palette[15].blue  |
 * |   63   | palette[15] <- creg | Always 0          |
```

```
 *
 */
module vga_tiles
  (input logic              clk, reset,              // Avalon MM Agent port
   input logic              chipselect, write,       // read == chipselect & !write
   input logic [14:0] address,                 // 32K window
   input logic [7:0]  writedata,               // 8-bit interface
   output logic [7:0] readdata,

   input logic       vga_clk_in, VGA_RESET,        // VGA signals
   output logic [7:0] VGA_R, VGA_G, VGA_B,
   output logic      VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n);

   logic [2:0]              creg_write;              // Latch enable per byte
   logic          tm_we, ts_we, palette_we;     // Memory write enables
   logic [7:0]              tm_dout;                 // Data from tilemap
   logic [3:0]              ts_dout;                 // Data from tileset
   logic [23:0]      creg, palette_dout;          // Data to/from palette


   // Sprite y position register
            logic [8:0] y_pos = 9'd40;

            // Colors
            logic [7:0] tile_R, tile_G, tile_B;
            logic [7:0] final_R, final_G, final_B;

            logic [9:0] hcount;
            logic [8:0]  vcount; // filled in by tiles, which is filled in by vga_counters.

            logic [7:0] scroll_offset;

            assign VGA_R = final_R;
            assign VGA_G = final_G;
            assign VGA_B = final_B;

   tiles tiles(.mem_clk       ( clk          ),
                  .tm_address     ( address[12:0] ),
                        .tm_din     ( writedata      ),
```

```
                .ts_address     ( address[13:0] ),
                        .ts_din     ( writedata[3:0] ),
                .palette_address( address[5:2]  ),
                        .palette_din( creg          ),
                        .hcount(hcount),
            .vcount(vcount),
                        .VGA_R(tile_R),
                        .VGA_G(tile_G),
                        .VGA_B(tile_B),
                        .scroll_offset (scroll_offset),
                        .*);
assign VGA_CLK = vga_clk_in;

always_comb begin                               // Address Decoder
  {tm_we, ts_we, palette_we, creg_write, readdata } = { 6'b 0, 8'h xx };
  if (chipselect)
        if (address[14] == 1'b 1) begin             // Tileset 1----------
          ts_we    = write;                     //  Write to tileset mem
          readdata = { 4'h 0, ts_dout };          //  Read lower 4 bits; pad upper
        end else if (address[13] == 1'b 0) begin    // Tilemap 00----------
          tm_we    = write;                      //  Write to tilemap mem
          readdata = tm_dout;                    //  Read 8 bits
        end else if ( address[12:6] == 7'b 0_0000_00 ) // Palette 010000000------
          case (address[1:0])
            2'h 0 : begin readdata = palette_dout[7:0];   // Read red byte
                        creg_write[0] = write;          // creg <- red
            end
            2'h 1 : begin readdata = palette_dout[15:8];  // Read green byte
                creg_write[1] = write;          // creg <- green
            end
       2'h 2 : begin readdata = palette_dout[23:16]; // Read blue byte
                creg_write[2] = write;          // creg <- blue
            end
            2'h 3 : begin readdata = 8'h 00;            // Always reads as 00
                palette_we = write;             // mem <- creg
            end
          endcase
end
```

```
always_ff @(posedge clk)
begin
                if (write && address == 15'h3002)
        y_pos <= writedata + 113;
end

always_ff @(posedge clk or posedge reset)
begin
                if (reset) scroll_offset = 8'd0;
                else if (write && address == 15'h3010)
        scroll_offset <= writedata; // sadly our scroll_offset is limited to < 255 because of our writedata witdth.
end

always_ff @(posedge clk or posedge reset)
 if (reset) creg <= 24'b 0; else begin
        if (creg_write[0]) creg[7:0]   <= writedata;    // Write byte (color)
        if (creg_write[1]) creg[15:8]  <= writedata;    // to creg according to
        if (creg_write[2]) creg[23:16] <= writedata;    // creg_write bits
 end


        always_comb begin
                if (VGA_BLANK_n) begin
                        if ((hcount >= 320) && (hcount < 336) &&  // sprite width = 16 pixels
                                (vcount >= y_pos) && (vcount < y_pos + 16))
                                {final_R, final_G, final_B} = 24'hFFFFFF;  // white sprite
                        else
                                {final_R, final_G, final_B} = {tile_R, tile_G, tile_B}; // tiles
                end else begin
                        {final_R, final_G, final_B} = 24'h000000;
                end
        end
endmodule
```

4. tiles.sv

```
C/C++
module tiles
 (input logic        VGA_CLK, VGA_RESET,
  output logic [7:0]  VGA_R, VGA_G, VGA_B,
  output logic        VGA_HS, VGA_VS, VGA_BLANK_n,

  input logic                mem_clk,        // Clock for memory ports

  input logic [12:0]  tm_address,     // Tilemap memory port
  input logic                tm_we,
  input logic [7:0]   tm_din,
  output logic [7:0]  tm_dout,

  input logic [13:0]  ts_address,     // Tileset memory port
  input logic                ts_we,
  input logic [3:0]   ts_din,
  output logic [3:0]  ts_dout,

  input logic [3:0]   palette_address, // Palette memory port
  input logic                palette_we,

  input logic [7:0]  scroll_offset,
  input logic [23:0]  palette_din,
  output logic [23:0] palette_dout,
  output logic [9:0]        hcount, // filled in by vga counters
  output logic [8:0]        vcount
  );

  logic [9:0] effective_hcount;
 /* Pads and wraps using modulo */
  assign effective_hcount = (hcount + {2'b0, scroll_offset}) % 1024;

  logic [4:0]                hcount1;        // Pipeline registers (5 bits for 32 pixels)
  logic          VGA_HS0, VGA_HS1, VGA_HS2;
  logic          VGA_BLANK_n0, VGA_BLANK_n1, VGA_BLANK_n2;

  logic [7:0]                tilenumber;     // Memory outputs
  logic [3:0]                colorindex;
```

```
/* verilator lint_off UNUSED */
logic               unconnected; // Extra vcount bit from counters
/* verilator lint_on UNUSED */

vga_counters cntrs(.vcount( {unconnected, vcount} ), // VGA Counters
                   .VGA_BLANK_n( VGA_BLANK_n0 ),
                   .VGA_HS( VGA_HS0 ),
                   .*);

twoportbram #(.DATA_BITS(8), .ADDRESS_BITS(13))  // Tile Map
tilemap(.clk1  ( VGA_CLK ), .clk2 ( mem_clk ),
        .addr1 ( { vcount[8:5], effective_hcount[9:5] } ), // Changed from [8:3],[9:3] to [8:5],[9:5] for 32 pixel tiles
        .we1   ( 1'b0 ), .din1( 8'h X ), .dout1( tilenumber ),
        .addr2 ( tm_address ),
        .we2   ( tm_we ), .din2( tm_din ), .dout2( tm_dout ));

always_ff @(posedge VGA_CLK)                  // Pipeline registers
  { hcount1, VGA_BLANK_n1, VGA_HS1 } <=
    { effective_hcount[4:0], VGA_BLANK_n0, VGA_HS0 };      // Changed from [2:0] to [4:0] for 32 pixel tiles

twoportbram #(.DATA_BITS(4), .ADDRESS_BITS(14))  // Tile Set
tileset(.clk1  ( VGA_CLK ), .clk2 ( mem_clk ),
        .addr1 ( { tilenumber, vcount[4:0], hcount1 } ), // Changed from [2:0] to [4:0] for local coordinates
        .we1   ( 1'b0 ), .din1( 4'h X), .dout1( colorindex ),
        .addr2 ( ts_address ),
        .we2   ( ts_we ), .din2( ts_din ), .dout2( ts_dout ));

always_ff @(posedge VGA_CLK)                  // Pipeline registers
  { VGA_BLANK_n2, VGA_HS2 } <= { VGA_BLANK_n1, VGA_HS1 };

twoportbram #(.DATA_BITS(24), .ADDRESS_BITS(4))  // Palette
palette(.clk1  ( VGA_CLK ), .clk2 ( mem_clk ),
        .addr1 ( colorindex ),
        .we1   ( 1'b0 ), .din1( 24'h X ), .dout1( { VGA_B, VGA_G, VGA_R } ),
        .addr2 ( palette_address ),
        .we2   ( palette_we ), .din2( palette_din ), .dout2( palette_dout ));

always_ff @(posedge VGA_CLK)                  // Pipeline registers
```

```
    { VGA_BLANK_n, VGA_HS } <= { VGA_BLANK_n2, VGA_HS2 };

endmodule
```

B. Software

1. audio_fifo.c

```
C/C++
#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/ioctl.h>
#include "audio_fifo.h"

// ================================================
// ===== audio_fifo structures and constants =====
// ================================================

#define AUDIO_FIFO_NAME "audio_fifo"
#define FIFO_ISTATUS_OFFSET    0x4 // relative to CSR base
```

```
struct audio_fifo_dev {
    struct resource res;
            struct resource res_fifo;
    struct resource res_csr;
    void __iomem *virtbase;
            void __iomem *virtbase_csr;
} audio_dev;

static void write_audio_fifo(uint16_t sample) {
    iowrite16(sample, audio_dev.virtbase);
}

static uint32_t read_fifo_fill_level(void) {
    return ioread16(audio_dev.virtbase_csr);
}

static uint32_t read_fifo_status(void) {
    return ioread16(audio_dev.virtbase_csr + FIFO_ISTATUS_OFFSET) & 0x3F; // Only i_status
bits
}

static long audio_fifo_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
    printk(KERN_INFO "audio_fifo_ioctl called with cmd 0x%x\n", cmd);
            if (!audio_dev.virtbase) {
                    pr_err("audio_fifo_ioctl: virtbase is NULL\n");
                    return -EIO;
            }
    switch (cmd) {
                    case WRITE_AUDIO_FIFO: {
      printk(KERN_INFO "WRITE_AUDIO_FIFO");
```

```
                audio_fifo_arg_t vla;
                if (copy_from_user(&vla, (audio_fifo_arg_t __user *)arg, sizeof(vla)))
                        return -EFAULT;
                write_audio_fifo(vla.audio);
                break;
        }

        case READ_AUDIO_STATUS: {
    printk(KERN_INFO "READ_AUDIO_STATUS");
                uint32_t status = read_fifo_status();
                if (copy_to_user((uint32_t __user *)arg, &status, sizeof(status)))
                        return -EFAULT;
                break;
        }

        case READ_AUDIO_FILL_LEVEL: {
    printk(KERN_INFO "READ_AUDIO_FILL_LEVEL");
                uint32_t level = read_fifo_fill_level();
                if (copy_to_user((uint32_t __user *)arg, &level, sizeof(level)))
                        return -EFAULT;
                break;
        }

        default:
                return -EINVAL;
    }

  return 0;
}


static const struct file_operations audio_fifo_fops = {
```

```
    .owner = THIS_MODULE,
    .unlocked_ioctl = audio_fifo_ioctl
};

static struct miscdevice audio_fifo_misc_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = AUDIO_FIFO_NAME,
    .fops = &audio_fifo_fops
};

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

    pr_info("audio_fifo: probe started\n");

    // Register misc device
    ret = misc_register(&audio_fifo_misc_device);
    if (ret) {
        pr_err("audio_fifo: failed to register misc device\n");
        return ret;
    }

    // Get FIFO memory resource
    ret = of_address_to_resource(pdev->dev.of_node, 0, &audio_dev.res_fifo);
    if (ret) {
        pr_err("audio_fifo: failed to get FIFO resource\n");
        goto out_deregister;
    }
    if (!request_mem_region(audio_dev.res_fifo.start, resource_size(&audio_dev.res_fifo),
AUDIO_FIFO_NAME)) {
        ret = -EBUSY;
        goto out_deregister;
```

```
    }

    // Get CSR memory resource
    ret = of_address_to_resource(pdev->dev.of_node, 1, &audio_dev.res_csr);
    if (ret) {
        pr_err("audio_fifo: failed to get CSR resource\n");
        goto out_release_fifo;
    }
    if (!request_mem_region(audio_dev.res_csr.start, resource_size(&audio_dev.res_csr),
AUDIO_FIFO_NAME "_csr")) {
        ret = -EBUSY;
        goto out_release_fifo;
    }

    // Map FIFO base
    audio_dev.virtbase = of_iomap(pdev->dev.of_node, 0);
    if (!audio_dev.virtbase) {
        pr_err("audio_fifo: failed to map FIFO registers\n");
        ret = -ENOMEM;
        goto out_release_csr;
    }

    // Map CSR base
    audio_dev.virtbase_csr = of_iomap(pdev->dev.of_node, 1);
    if (!audio_dev.virtbase_csr) {
        pr_err("audio_fifo: failed to map CSR registers\n");
        ret = -ENOMEM;
        goto out_unmap_fifo;
    }

    pr_info("audio_fifo: probe successful\n");
```

```
    pr_info("audio_fifo: FIFO mapped to %p, CSR mapped to %p\n", audio_dev.virtbase,
audio_dev.virtbase_csr);
    return 0;

// Cleanup paths
out_unmap_fifo:
    iounmap(audio_dev.virtbase);
out_release_csr:
    release_mem_region(audio_dev.res_csr.start, resource_size(&audio_dev.res_csr));
out_release_fifo:
    release_mem_region(audio_dev.res_fifo.start, resource_size(&audio_dev.res_fifo));
out_deregister:
    misc_deregister(&audio_fifo_misc_device);
    return ret;
}


static int __exit audio_fifo_remove(struct platform_device *pdev) {
            iounmap(audio_dev.virtbase);
            iounmap(audio_dev.virtbase_csr);
            release_mem_region(audio_dev.res_fifo.start, resource_size(&audio_dev.res_fifo));
            release_mem_region(audio_dev.res_csr.start, resource_size(&audio_dev.res_csr));
            misc_deregister(&audio_fifo_misc_device);
    pr_info("audio_fifo: removed\n");
    return 0;
}

#ifdef CONFIG_OF
static const struct of_device_id audio_fifo_of_match[] = {
    { .compatible = "ALTR,fifo-21.1" },
    { .compatible = "ALTR,fifo-1.0" },
    {},
```

```
};
MODULE_DEVICE_TABLE(of, audio_fifo_of_match);
#endif

static struct platform_driver audio_fifo_driver = {
    .probe = audio_fifo_probe,
    .remove = audio_fifo_remove,
    .driver = {
        .name = AUDIO_FIFO_NAME,
        .owner = THIS_MODULE,
        .of_match_table = of_match_ptr(audio_fifo_of_match),
    },
};

/* Called when the module is loaded: set things up */
static int __init audio_fifo_init(void)
{
    pr_info("audio_fifo: init\n");
    return platform_driver_register(&audio_fifo_driver);
}

static void __exit audio_fifo_exit(void)
{
    platform_driver_unregister(&audio_fifo_driver);
    pr_info("audio_fifo: exit\n");
}

module_init(audio_fifo_init);
module_exit(audio_fifo_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Stephen A. Edwards, Columbia University");
```

```
MODULE_DESCRIPTION("audio FIFO driver");
```

2. audio_fifo.h

```
C/C++
#ifndef _AUDIO_FIFO_H
#define _AUDIO_FIFO_H

#ifndef __KERNEL__
#include <stdint.h>
#endif

#include <linux/ioctl.h>

typedef struct {
    uint32_t audio; // or any structure matching what audio_fifo_ioctl expects
} audio_fifo_arg_t;

#define AUDIO_FIFO_MAGIC 'r'
#define WRITE_AUDIO_FIFO       _IOW(AUDIO_FIFO_MAGIC, 1, audio_fifo_arg_t *)
#define READ_AUDIO_FILL_LEVEL  _IOR(AUDIO_FIFO_MAGIC, 2, uint32_t *)
#define READ_AUDIO_STATUS      _IOR(AUDIO_FIFO_MAGIC, 3, uint32_t *)

#endif
```

3. audio.c

```
C/C++
#include <stdio.h>
#include <stdint.h>
#include <fcntl.h>
```

```c
#include <unistd.h>
#include <sys/ioctl.h>
#include <linux/i2c-dev.h>

#include "audio_fifo.h"


#define I2C_DEV "/dev/i2c-0"  // Might be /dev/i2c-1 on some boards
#define WM8731_ADDR 0x1A

// Utility to send a 9-bit register: 7 bits address + 9 bits data
int wm8731_write(int i2c_fd, uint8_t reg, uint16_t data) {
    uint8_t buf[2];
    buf[0] = (reg << 1) | ((data >> 8) & 0x01);  // reg address (7 bits) + D8
    buf[1] = data & 0xFF;                        // D7..D0
    return write(i2c_fd, buf, 2);
}

void init_wm8731() {
    int i2c_fd = open(I2C_DEV, O_RDWR);
    if (i2c_fd < 0) {
        perror("open i2c");
        return;
    }

    if (ioctl(i2c_fd, I2C_SLAVE, WM8731_ADDR) < 0) {
        perror("ioctl I2C_SLAVE");
        close(i2c_fd);
        return;
    }

    // Soft reset
    wm8731_write(i2c_fd, 0x0F, 0x000);

    // Left headphone out: volume = 0x79
    wm8731_write(i2c_fd, 0x02, 0x179);
    // Right headphone out: volume = 0x79
    wm8731_write(i2c_fd, 0x03, 0x179);
```

```
    // Analog audio path: DAC selected
    wm8731_write(i2c_fd, 0x04, 0x012);

    // Digital audio path: disable soft mute
    wm8731_write(i2c_fd, 0x05, 0x000);

    // Power down control: everything on
    wm8731_write(i2c_fd, 0x06, 0x000);

    // Digital audio interface format: I2S, 16-bit, MCLK slave (left justified)
    wm8731_write(i2c_fd, 0x07, 0x000);

    // Sampling control: normal mode, 48kHz
    wm8731_write(i2c_fd, 0x08, 0x000);

    // Activate digital interface
    wm8731_write(i2c_fd, 0x09, 0x001);

    close(i2c_fd);
}

void print_fifo_status(uint32_t status) {
    printf("FIFO i_status: 0x%08x\n", status);

    if (status & (1 << 0)) printf(" - FULL: FIFO is full\n");
    if (status & (1 << 1)) printf(" - EMPTY: FIFO is empty\n");
    if (status & (1 << 2)) printf(" - ALMOSTFULL: Fill level >= almostfull threshold\n");
    if (status & (1 << 3)) printf(" - ALMOSTEMPTY: Fill level <= almostempty threshold\n");
    if (status & (1 << 4)) printf(" - OVERFLOW: Write occurred when FIFO was full\n");
    if (status & (1 << 5)) printf(" - UNDERFLOW: Read occurred when FIFO was empty\n");

    // Sanity check
    if ((status & 0x3F) == 0)
        printf(" - FIFO is somewhere between ALMOSTEMPTY and ALMOSTFULL, not full or empty\n");
}

int main() {
            printf("Initializing Audio CODEC\n");
            init_wm8731();
```

```
        usleep(1000);

        int fd = open("/dev/audio_fifo", O_RDWR);
if (fd < 0) {
                perror("Failed to open audio_fifo");
    return 1;
}

FILE *audio = fopen("monody_stereo_48k.raw", "rb");
if (!audio) {
    perror("Failed to open audio file");
    close(fd);
    return 1;
}

printf("Opened audio_fifo device and audio file\n");
audio_fifo_arg_t arg = {0};
uint16_t dummy;


        //uint32_t fill_level;
        //if (ioctl(fd, READ_AUDIO_FILL_LEVEL, &fill_level) == -1) {
        //      perror("ioctl READ_AUDIO_FILL_LEVEL failed");
        //      close(fd);
        //      return 1;
        //}

        //printf("Initial FIFO fill level: %u\n", fill_level);

        int i = 0;
while (fread(&arg.audio, sizeof(uint16_t), 1, audio) == 1) {
                // printf("Skipping right channel\n");
    // Skip right channel
    fread(&dummy, sizeof(uint16_t), 1, audio);  // skip right

                // Shift left sample into upper 16 bits, right = 0
                arg.audio = (uint32_t)arg.audio;
```

```
                if (ioctl(fd, WRITE_AUDIO_FIFO, &arg) == -1) {
                        perror("ioctl WRITE_AUDIO_FIFO failed");
                        break;
                }
        /*
                uint32_t status;
                if ((i++ % 1000) == 0) {

    if (ioctl(fd, READ_AUDIO_STATUS, &status) == -1) {
                                perror("ioctl READ_AUDIO_STATUS failed");
                                close(fd);
                                return 1;
                        } else {
                                print_fifo_status(status);
                        }

                        uint32_t fill_level;
                        if (ioctl(fd, READ_AUDIO_FILL_LEVEL, &fill_level) == -1) {
                                perror("ioctl READ_AUDIO_FILL_LEVEL failed");
                                close(fd);
                                return 1;
                        }

                        printf("FIFO fill level: %u\n", fill_level);
                }
*/
                usleep(100);
    }

    printf("cleaning up\n");
    fclose(audio);
    close(fd);
    return 0;
}
```

4. geo_dash.c

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

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

```
// ============================================
// ====== geo_dash structures and constants ======
// ============================================

#define DRIVER_NAME "geo_dash"
// Assuming that we have 16-bit registers.
#define X_SHIFT(base)       ((base) + 0x02)  // 16-bit

#define TILEMAP(base)   ((base))
#define PALETTE(base)   ((base) + 0x2000)
#define TILESET(base)   ((base) + 0x4000)

#define FLAGS(base)         ((base) + 0x0C)  // lower 8 bits used
#define OUTPUT_FLAGS(base)   ((base) + 0x0E)  // lower 8 bits used
#define SCROLL_OFFSET(base)  ((base) + 0x10)  // 16-bit  - added for tile scroll
#define PLAYER_Y_POS(base) ((base) + 0x3002)  // 16-bit register
#define SCROLL_OFFSET(base) ((base) + 0x3010)  // 16-bit register

/*
Information about our geometry_dash device. Acts as a mirror of hardware state.
*/

struct geo_dash_dev {
    struct resource res; /* Our registers. */
    void __iomem *virtbase; /* Where our registers can be accessed in memory. */
    short x_shift;
    short scroll_offset;   /* Added to keep track of scroll offset */
} geo_dash_dev;

static void write_tile(uint8_t *value, int row, int col)
{
    pr_info("writing %d to tile map at row %d, col %d", *value, row, col);
    void *tilemap_location = TILEMAP(geo_dash_dev.virtbase) + row * 32 + col;
    iowrite8(*value, tilemap_location);
}

static void write_palette(uint32_t *rgb, int color_index)
{
    pr_info("writing color %x at index %d to palette", *rgb, color_index);
```

```
    uint8_t r = (*rgb >> 16) & 0xFF;
    uint8_t g = (*rgb >> 8) & 0xFF;
    uint8_t b = (*rgb) & 0xFF;

    uintptr_t addr = (uintptr_t) PALETTE(geo_dash_dev.virtbase) + color_index * 4;

    iowrite8(r, addr);
    iowrite8(g, addr + 1);
    iowrite8(b, addr + 2);
    iowrite8(0x01, addr + 3);
}

static void write_tileset(uint8_t *value, int tile_no, int pixel_no)
{
    /*writing to the pixel in that specific tile. */
    //pr_info("writing to tile");
    iowrite8(*value, TILESET(geo_dash_dev.virtbase) + tile_no * 32 * 32 + pixel_no);

}

static uint8_t read_tile(int row, int col)
{
     uintptr_t tilemap_location = (uintptr_t) TILEMAP(geo_dash_dev.virtbase) + row * 40 + col;
     return ioread8(tilemap_location);
}

static uint32_t read_palette(int color_index)
{
     void *rgb_location = PALETTE(geo_dash_dev.virtbase) + 4 * color_index;
     return ioread32(rgb_location);
}

static uint8_t read_tileset(int tile_no, int pixel_no)
{
     void *pixel_location = TILESET(geo_dash_dev.virtbase) + tile_no * 32 * 32 + pixel_no;
     return ioread8(pixel_location);
}

static void write_player_y_position(uint8_t *value) {
```

```
        iowrite8(*value, PLAYER_Y_POS(geo_dash_dev.virtbase));
}

static void write_x_shift(unsigned short *value) {
    iowrite16(*value, X_SHIFT(geo_dash_dev.virtbase));
    geo_dash_dev.x_shift = *value;
}

static void write_flags(uint8_t *value) {
    iowrite16((uint16_t)(*value), FLAGS(geo_dash_dev.virtbase));
}

static void write_output_flags(uint8_t *value) {
    iowrite16((uint16_t)(*value), OUTPUT_FLAGS(geo_dash_dev.virtbase));
}

static void write_scroll_offset(uint8_t *value) {
        iowrite8(*value, SCROLL_OFFSET(geo_dash_dev.virtbase));
}

static long geo_dash_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
    geo_dash_arg_t vla;
        printk(KERN_INFO "geo_dash_ioctl called with cmd 0x%x\n", cmd);

    // Copy user struct into kernel space
    if (copy_from_user(&vla, (geo_dash_arg_t *) arg, sizeof(vla)))
        return -EFAULT;

    switch (cmd) {
        case WRITE_TILE:
            pr_info("writing %d to tilemap at %d, %d", vla.tile_value, vla.tilemap_row, vla.tilemap_col);
            write_tile(&vla.tile_value, vla.tilemap_row, vla.tilemap_col);
            break;

        case READ_TILE:
        {
            uint8_t tile_value = read_tile(vla.tilemap_row, vla.tilemap_col);
            vla.tile_value = tile_value;
```

```
        if (copy_to_user((geo_dash_arg_t *) arg, &vla, sizeof(vla)))
            return -EFAULT;
        break;
    }
    case WRITE_PALETTE:
        pr_info("writing to palette\n");
        write_palette(&vla.rgb, vla.color_index);
        break;

    case READ_PALETTE:
    {
        uint32_t rgb = read_palette(vla.color_index);
        vla.rgb = rgb;
        if (copy_to_user((geo_dash_arg_t *) arg, &vla, sizeof(vla)))
            return -EFAULT;
        break;
    }
    case WRITE_TILESET: ;
//      pr_info("writing to tile set\n");
        /* this should write 32x32 bytes to the location requested*/
        int i = 0;
        for (i = 0; i < 32; i++) {
            int j = 0;
            for (j = 0; j < 32; j++) {
                write_tileset(&vla.tileset[i][j], vla.tile_no, 32*i+j);
            }
        }
        break;
    case READ_TILESET:
        {
         int i, j;

         for (i = 0; i < 32; i++) {
           for (j = 0; j < 32; j++) {
             vla.tileset[i][j] = read_tileset(vla.tile_no, 32*i+j);
           }
         }
         if (copy_to_user((geo_dash_arg_t *) arg, &vla, sizeof(vla)))
           return -EFAULT;
```

```
        }
        break;
    case WRITE_X_SHIFT:
        write_x_shift(&vla.x_shift);
        break;

    case WRITE_PLAYER_Y_POS:
        write_player_y_position(&vla.player_y);
        break;

    case WRITE_FLAGS:
        write_flags(&vla.flags);
        break;

    case WRITE_OUTPUT_FLAGS:
        write_output_flags(&vla.output_flags);
        break;

    case WRITE_SCROLL_OFFSET:
        write_scroll_offset(&vla.scroll_offset);
        break;

    default:
        return -EINVAL;  // Unknown command
  }

  return 0;
}

static const struct file_operations geo_dash_fops = {
  .owner = THIS_MODULE,
  .unlocked_ioctl = geo_dash_ioctl
};

static struct miscdevice geo_dash_misc_device = {
  .minor = MISC_DYNAMIC_MINOR,
  .name = DRIVER_NAME,
  .fops = &geo_dash_fops
};
```

```
/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init geo_dash_probe(struct platform_device *pdev)
{
	//cat_invaders_color_t beige = { 0xf9, 0xe4, 0xb7 };
			//audio_t audio_begin = { 0x00, 0x00, 0x00 };
	int ret;
	pr_info("geo_dash: probe successful\n");

	/* Register ourselves as a misc device: creates /dev/geo_dash */
	ret = misc_register(&geo_dash_misc_device);

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

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

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

	return 0;

out_release_mem_region:
```

```
        release_mem_region(geo_dash_dev.res.start, resource_size(&geo_dash_dev.res));
out_deregister:
        misc_deregister(&geo_dash_misc_device);
        return ret;
}

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

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

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver geo_dash_driver = {
        .probe = geo_dash_probe, // move it here
        .remove = __exit_p(geo_dash_remove),
        .driver = {
                .name = DRIVER_NAME,
                .owner = THIS_MODULE,
                .of_match_table = of_match_ptr(geo_dash_of_match),
        },
};


// ===========================================
// ============ module init and exit ============
// ===========================================
```

```
/* Called when the module is loaded: set things up */
static int __init geo_dash_init(void)
{
   int ret;

   pr_info(DRIVER_NAME ": init\n");

   ret = platform_driver_register(&geo_dash_driver);
   if (ret)
      return ret;

   return 0;
}

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

module_init(geo_dash_init);
module_exit(geo_dash_exit);

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

5. geo_dash.h

```
C/C++
#ifndef _GEO_DASH_H
#define _GEO_DASH_H

#ifndef __KERNEL__
#include <stdint.h>
```

```
#endif

#include <linux/ioctl.h>

// Game settings and constants
#define MAX_LEVEL_LENGTH 1024  // Maximum level length in blocks
#define MAX_OBSTACLES 16       // Maximum number of different obstacle types

// Obstacle type definitions
#define OBS_NONE 0             // Empty space
#define OBS_SPIKE 1            // Spike (instant death)
#define OBS_BLOCK 2            // Square block (collision)
#define OBS_PLATFORM 3         // Platform (can land on)
#define OBS_JUMP_PAD 4         // Jump pad (extra boost)
#define OBS_GRAVITY_PORTAL 5   // Gravity portal (flip gravity)

// Player state flags
#define PLAYER_NORMAL 0x00     // Default state
#define PLAYER_JUMPING 0x01    // Player is jumping
#define PLAYER_DEAD 0x02       // Player is dead
#define PLAYER_INVERTED 0x04   // Gravity is inverted

// Game state flags
#define GAME_LOADING 0x01      // Game is loading
#define GAME_READY 0x02        // Game is ready to start
#define GAME_PLAYING 0x04      // Game is in progress
#define GAME_OVER 0x08         // Game is over

// Structure for communicating with the device driver
typedef struct {
    uint16_t x_shift;          // Pixel offset for scrolling
    uint8_t player_y;          // Player Y position
    uint8_t  bg_r;             // Background color (R)
    uint8_t  bg_g;             // Background color (G)
    uint8_t  bg_b;             // Background color (B)
    uint8_t  map_block;        // Current map block
    uint8_t  flags;            // Game flags
    uint8_t  output_flags;     // Output status flags
    uint32_t audio;            // Audio sample
```

```
    uint8_t scroll_offset;    // Tile scrolling offset (new field)
    uint8_t  tile_value;
    int tilemap_row;
    int tilemap_col;
    uint8_t tileset[32][32];
    uint32_t rgb;
    int color_index;
    int tile_no;
} geo_dash_arg_t;

// IOCTL commands
#define GEO_DASH_MAGIC 'q'

#define WRITE_X_SHIFT         _IOW(GEO_DASH_MAGIC, 0, geo_dash_arg_t *)
#define WRITE_PLAYER_Y_POS    _IOW(GEO_DASH_MAGIC, 1, geo_dash_arg_t *)
#define WRITE_BACKGROUND_R    _IOW(GEO_DASH_MAGIC, 2, geo_dash_arg_t *)
#define WRITE_BACKGROUND_G    _IOW(GEO_DASH_MAGIC, 3, geo_dash_arg_t *)
#define WRITE_BACKGROUND_B    _IOW(GEO_DASH_MAGIC, 4, geo_dash_arg_t *)
#define WRITE_MAP_BLOCK       _IOW(GEO_DASH_MAGIC, 5, geo_dash_arg_t *)
#define WRITE_FLAGS           _IOW(GEO_DASH_MAGIC, 6, geo_dash_arg_t *)
#define WRITE_OUTPUT_FLAGS    _IOW(GEO_DASH_MAGIC, 7, geo_dash_arg_t *)
#define WRITE_SCROLL_OFFSET   _IOW(GEO_DASH_MAGIC, 8, geo_dash_arg_t *)
#define WRITE_TILE            _IOW(GEO_DASH_MAGIC, 9, geo_dash_arg_t *)
#define WRITE_PALETTE         _IOW(GEO_DASH_MAGIC, 10, geo_dash_arg_t *)
#define WRITE_TILESET         _IOW(GEO_DASH_MAGIC, 11, geo_dash_arg_t *)

#define READ_TILE             _IOWR(GEO_DASH_MAGIC, 12, geo_dash_arg_t *)
#define READ_PALETTE          _IOWR(GEO_DASH_MAGIC, 13, geo_dash_arg_t *)
#define READ_TILESET          _IOWR(GEO_DASH_MAGIC, 14, geo_dash_arg_t *)
#endif
```

6. game_loop.c

```
C/C++
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
```

```c
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <time.h>
#include <string.h>
#include <signal.h>
#include <math.h>
#include <pthread.h>
#include "geo_dash.h"
#include "../controller/usbjoypad.h"

#define SCREEN_WIDTH 20
#define SCREEN_HEIGHT 15
#define TILE_HEIGHT 32

// Player sprite constants
#define PLAYER_TILE 8           // Tile index for player sprite
#define PLAYER_START_X 10       // Starting X position (screen coordinate)
#define PLAYER_START_Y 11       // Starting Y position (screen coordinate)
#define GRAVITY 2               // Gravity force
#define JUMP_VELOCITY -20       // Initial jump velocity (negative means upward)
#define MAX_FALL_SPEED 5.0      // Maximum falling speed
#define GROUND_Y 11             // Ground Y position
#define REG_GROUND 255          // Ground Y position

#define GAMEOVER_BIN  "../hw/gd-tiles/gameover.bin"

// Game state
typedef struct {
    float player_x;             // Player X position in screen coordinates
    float player_y;             // Player Y position in screen coordinates
    float player_vy;            // Player vertical velocity
    bool is_jumping;            // Is the player currently jumping?
    bool is_dead;               // Is the player dead?
    int level_x;                // Current level position (scroll position)
} GameState;

GameState game;
```

```
// Level buffer - allocated dynamically based on level size
uint8_t *level_buffer = NULL;
int level_width = 0;

// Flag to control the game loop
volatile int keep_running = 1;

static int      map_origin   = 0;  // which device column (0–31) is the leftmost tile
static uint8_t  pixel_offset = 0;  // 0–31 pixel scroll inside a tile

// Thread for game logic
pthread_t game_thread;
pthread_mutex_t game_mutex = PTHREAD_MUTEX_INITIALIZER;

static int load_gameover(void);
static void show_gameover(int fd);


/**
 * Signal handler for clean termination
 */
void handle_signal(int sig) {
    keep_running = 0;
}

/**
 * Get tile value at specified row and column in the level
 */
 uint8_t get_level_tile(int row, int col) {
    if (row < 0 || row >= SCREEN_HEIGHT || col < 0 || col >= level_width) {
        return 0; // Return empty tile for out of bounds
    }
    return level_buffer[row * level_width + col];
}

// A 20×15 tile image that spells out “GAME OVER”
static uint8_t gameover_map[SCREEN_HEIGHT][SCREEN_WIDTH];

// Read the 20×15 tile indices from disk into gameover_map[][]
```

```
static int load_gameover(void) {
    FILE *f = fopen(GAMEOVER_BIN, "rb");
    if (!f) { perror("Opening gameover.bin"); return -1; }
    size_t want = SCREEN_HEIGHT * SCREEN_WIDTH;
    size_t got  = fread(gameover_map, 1, want, f);
    fclose(f);
    if (got != want) {
        fprintf(stderr, "Expected %zu bytes in %s, got %zu\n", want, GAMEOVER_BIN, got);
        return -1;
    }
    return 0;
}

static void show_gameover(int fd) {
    geo_dash_arg_t arg;
    pthread_mutex_lock(&game_mutex);
    for (int row = 0; row < SCREEN_HEIGHT; row++) {
        for (int col = 0; col < SCREEN_WIDTH; col++) {
            arg.tilemap_row = row;
            arg.tilemap_col = col;
            arg.tile_value  = gameover_map[row][col];
            if (ioctl(fd, WRITE_TILE, &arg) < 0) {
                perror("WRITE_TILE gameover");
                pthread_mutex_unlock(&game_mutex);
                return;
            }
        }
    }
    pthread_mutex_unlock(&game_mutex);
                sleep(5);
}


void initial_fill(int fd) {
    map_origin = 0;
    for (int col = 0; col < 32; col++) {
        for (int row = 0; row < SCREEN_HEIGHT; row++) {
            geo_dash_arg_t arg = {
                .tilemap_row = row,
```

```
            .tilemap_col = col,
            .tile_value   = get_level_tile(row, game.level_x + col)
        };
        if (ioctl(fd, WRITE_TILE, &arg) < 0) {
            perror("Error writing tile");
            pthread_mutex_unlock(&game_mutex);
            return;
        }
    }
  }

  pthread_mutex_unlock(&game_mutex);
}

/**
 * Load a level from a binary file
 * @param filename The level file path
 * @param width The width of the level (in tiles)
 * @return 0 on success, negative value on error
 */
int load_level(const char *filename, int width) {
  FILE *file = fopen(filename, "rb");
  if (file == NULL) {
    perror("Error opening level file");
    return -1;
  }

  // Set the level width
  level_width = width;

  // Allocate memory for level buffer
  level_buffer = (uint8_t *)malloc(SCREEN_HEIGHT * level_width * sizeof(uint8_t));
  if (level_buffer == NULL) {
    perror("Failed to allocate memory for level");
    fclose(file);
    return -2;
  }

  // Initialize to zeros
```

```
    memset(level_buffer, 0, SCREEN_HEIGHT * level_width * sizeof(uint8_t));

    // Read the level data from file (row major format)
    size_t bytes_read = fread(level_buffer, 1, SCREEN_HEIGHT * level_width, file);
    printf("Read %zu bytes from level file\n", bytes_read);

    fclose(file);
    return 0;
}

/**
 * Clean up allocated memory for the level
 */
void cleanup_level() {
    if (level_buffer != NULL) {
        free(level_buffer);
        level_buffer = NULL;
    }
}



/**
 * Check if a tile is solid (obstacle/ground)
 */
bool is_solid_tile(uint8_t tile) {
    // Tiles 1 (ground), 2 (platform), and 3 (obstacle) are considered solid
    return (tile == 1 || tile == 2 || tile == 3);
}

/**
 * Check if player is colliding with an obstacle
 */
bool check_collision(int player_screen_x, int player_screen_y) {
    // Get the absolute level position
    int level_x_pos = game.level_x + player_screen_x;

    // Check the tile at the player's position
    uint8_t tile = get_level_tile(player_screen_y, level_x_pos);
```

```
            printf("player_screen_y: %d, level_x_pos %d, tile is: %d\n", player_screen_y, level_x_pos, tile);

    // Obstacle is tile type 3 (red)
    return (tile == 1 || tile == 2);
}

/**
 * Update the visible tilemap on the device
 * @param fd The device file descriptor
 */
void update_screen(int fd) {
    geo_dash_arg_t arg;

    pthread_mutex_lock(&game_mutex);

    // Write each visible tile to the device
    for (int row = 0; row < SCREEN_HEIGHT; row++) {
        for (int col = 0; col < SCREEN_WIDTH; col++) {
            // Get the level column
            int level_col = game.level_x + col;

            // Get the tile from our level buffer
            uint8_t tile = get_level_tile(row, level_col);

            // Write the tile to the device
            arg.tilemap_row = row;
            arg.tilemap_col = col;
            arg.tile_value = tile;

            if (ioctl(fd, WRITE_TILE, &arg) < 0) {
                perror("Error writing tile");
                pthread_mutex_unlock(&game_mutex);
                return;
            }
        }
    }

    pthread_mutex_unlock(&game_mutex);
}
```

```c
/**
 * Initialize the palette colors
 * @param fd The device file descriptor
 */
void initialize_palette(int fd) {
    geo_dash_arg_t arg;

    // Set up some basic colors
    uint32_t colors[] = {
        0x00000000,  // Color 0: Black/transparent
        0x00663300,  // Color 1: Brown (ground)
        0x00888888,  // Color 2: Gray (platforms)
        0x00FF0000,  // Color 3: Red (obstacles)
        0x00FFFFFF,  // Color 4: White (clouds)
        0x0000FF00,  // Color 5: Green
        0x000000FF,  // Color 6: Blue
        0x00FFFF00,  // Color 7: Yellow
        0x00FF00FF   // Color 8: Purple (player)
    };

    // Write the palette colors to the device
    for (int i = 0; i < 9; i++) {
        arg.color_index = i;
        arg.rgb = colors[i];

        if (ioctl(fd, WRITE_PALETTE, &arg) < 0) {
            perror("Error writing palette");
            return;
        }
    }
}

/**
 * Load a tileset from file
 * @param fd The device file descriptor
 * @param filename The tileset file path
 * @return 0 on success, negative value on error
 */
```

```
int load_tileset(int fd, const char *filename) {
    geo_dash_arg_t arg;
    int tile_count = 0;

    // Read tileset from file
    FILE *file = fopen(filename, "rb");
    if (file == NULL) {
        perror("Error opening tileset file");
        return -1;
    }

    // Continue reading tiles until we reach EOF
    while (!feof(file)) {
        // Initialize the current tile with zeros
        for (int row = 0; row < 32; row++) {
            for (int col = 0; col < 32; col++) {
                arg.tileset[row][col] = 0;
            }
        }

        // Read a 32x32 tile from the file
        for (int row = 0; row < 32; row++) {
            for (int col = 0; col < 32; col++) {
                uint8_t byte;
                if (fread(&byte, 1, 1, file) != 1) {
                    if (feof(file)) {
                        // End of file reached
                        goto end_of_file;
                    } else {
                        perror("Error reading from file");
                        fclose(file);
                        return -2;
                    }
                }
                arg.tileset[row][col] = byte;
            }
        }

        // Write this tile to the device
```

```
            arg.tile_no = tile_count;
            if (ioctl(fd, WRITE_TILESET, &arg) < 0) {
                perror("Error writing tileset");
                fclose(file);
                return -3;
            }

            tile_count++;
        }

    end_of_file:
        fclose(file);

        // Create a player sprite tile (if we have enough tile slots)
        // We'll use tile index 8 for our player
        if (tile_count <= 8) {
            // Initialize tile with zeros
            for (int row = 0; row < 32; row++) {
                for (int col = 0; col < 32; col++) {
                    arg.tileset[row][col] = 0;
                }
            }

            // Create a simple player sprite - a colored square with eyes
            for (int row = 8; row < 24; row++) {
                for (int col = 8; col < 24; col++) {
                    if (row == 8 || row == 23 || col == 8 || col == 23) {
                        // Border
                        arg.tileset[row][col] = 8;  // Purple outline
                    } else if ((row == 12 && (col == 12 || col == 19)) ||
                               (row == 13 && (col == 12 || col == 19))) {
                        // Eyes
                        arg.tileset[row][col] = 0;  // Black eyes
                    } else {
                        // Body fill
                        arg.tileset[row][col] = 8;  // Purple fill
                    }
                }
            }
```

```
        // Write the player tile to the device
        arg.tile_no = PLAYER_TILE;
        if (ioctl(fd, WRITE_TILESET, &arg) < 0) {
            perror("Error writing player tile");
            return -4;
        }

        printf("Player sprite created as tile %d\n", PLAYER_TILE);
    } else {
        printf("Warning: Not enough tile slots for player sprite\n");
    }

    printf("Tileset successfully loaded and written to device (%d tiles)\n", tile_count);
    return 0;
}

/**
 * Initialize the game state
 */
void initialize_game() {
    game.player_x = PLAYER_START_X;
    game.player_y = REG_GROUND;
    game.player_vy = 0;
    game.is_jumping = false;
    game.is_dead = false;
    game.level_x = 0;
}

/**
 * Update player physics and game state
 */
void update_game_state(int fd) {
    pthread_mutex_lock(&game_mutex);
    // Get controller state
    ControllerState controller = controller_get_state();

    // Handle jump input
    if (controller.buttonAPressed && !game.is_jumping && game.player_y >= REG_GROUND) {
```

```
    game.player_vy = JUMP_VELOCITY;
    game.is_jumping = true;
}

// Update player vertical position with gravity
game.player_vy += GRAVITY;

// Clamp fall speed
if (game.player_vy > MAX_FALL_SPEED) {
    game.player_vy = MAX_FALL_SPEED;
}

// Update player Y position

// assume player_y and player_vy are uint32_t
game.player_y += game.player_vy;

        if (game.player_y >= REG_GROUND) {
                game.player_y = REG_GROUND;
                game.player_vy = 0;
                game.is_jumping = false;
        }

        // Update hardware sprite position via ioctl
        geo_dash_arg_t arg;
        arg.player_y = (uint8_t)game.player_y;
        // printf("Writing player y pos to pos %d\n", arg.player_y);
        if (ioctl(fd, WRITE_PLAYER_Y_POS, &arg) < 0) {
                perror("Failed to write player Y position");
        }

// Check for ground collision
if (game.player_y >= REG_GROUND) {
    game.player_y = REG_GROUND;
    game.player_vy = 0;
    game.is_jumping = false;
}

        int tile_x = (int)game.player_x;
```

```
            int tile_y = (int)((game.player_y+113) / TILE_HEIGHT);

    // Check for obstacle collision
    if (check_collision(tile_x, tile_y)) {
                        pthread_mutex_unlock(&game_mutex);
        game.is_dead = true;
                        keep_running = 0;
        printf("\nGame Over! Collided with an obstacle.\n");
                        show_gameover(fd);
                        return;
    }

            /*
            // THEN: look one tile below our feet for ground (tile == 1)
    uint8_t below = get_level_tile(tile_y + 1, game.level_x + tile_x);
    if (below == 1) {
        // snap to the *top* of that block
        float ground_px = (tile_y + 1) * TILE_HEIGHT - TILE_HEIGHT - 113;
        game.player_y   = ground_px;
        game.player_vy  = 0;
        game.is_jumping = false;
        // push to hardware
        geo_dash_arg_t arg = { .player_y = (uint8_t)game.player_y };
        ioctl(fd, WRITE_PLAYER_Y_POS, &arg);
        pthread_mutex_unlock(&game_mutex);
        return;
    }
            */

    pthread_mutex_unlock(&game_mutex);
}

/**
 * Game loop thread function
 */
void* game_loop(void* arg) {
    int fd = *((int*)arg);
    struct timespec sleep_time;
```

```
// Set up timing
int fps = 60;  // Target frames per second
int frame_time_ms = 1000 / fps;

// Sleep time structure for consistent frame rate
sleep_time.tv_sec = 0;
sleep_time.tv_nsec = frame_time_ms * 1000000; // Convert to nanoseconds
printf("player y pos before running game loop: %d\n", game.player_y);
// Main game loop
while (keep_running && game.level_x < level_width - SCREEN_WIDTH) {
    // Skip processing if game is over
    if (!game.is_dead) {
        // Update game state (player physics, etc.)
        update_game_state(fd);

        // Advance level position every few frames for scrolling
        static int scroll_counter = 0;
        if (++scroll_counter >= 15) {  // Scroll every 15 frames (4 times per second at 60fps)
            scroll_counter = 0;

            pthread_mutex_lock(&game_mutex);
            game.level_x++;
            pthread_mutex_unlock(&game_mutex);

            // Display some debug info
            printf("Level position: %d/%d | Player: (%.1f, %.1f) | %s\r",
                   game.level_x, level_width - SCREEN_WIDTH,
                   game.player_x, game.player_y,
                   game.is_jumping ? "Jumping" : "Grounded");
            fflush(stdout);
        }
    }



                    // 2) smooth pixel scroll
    pixel_offset = (pixel_offset + 1) & 0x1F;
    geo_dash_arg_t a = { .scroll_offset = pixel_offset };
    if (ioctl(fd, WRITE_SCROLL_OFFSET, &a) < 0) {
        perror("WRITE_SCROLL_OFFSET");
```

```
        }

        // 3) if we just wrapped a full tile (32px), reload one new column
        if (pixel_offset == 0) {
            pthread_mutex_lock(&game_mutex);
            game.level_x++;
            int dead_col  = map_origin;
            int new_tilec = game.level_x + 31;
            for (int r = 0; r < SCREEN_HEIGHT; r++) {
                geo_dash_arg_t x = {
                    .tilemap_row = r,
                    .tilemap_col = dead_col,
                    .tile_value   = get_level_tile(r, new_tilec)
                };
                if (ioctl(fd, WRITE_TILE, &x) < 0)
                    perror("WRITE_TILE (repaint)");
            }
            map_origin = (map_origin + 1) & 31;
            pthread_mutex_unlock(&game_mutex);
        }

        // Update screen
        update_screen(fd);

        // Sleep to maintain frame rate
        nanosleep(&sleep_time, NULL);
    }

    printf("\nGame complete!\n");
    return NULL;
}

int main(int argc, char *argv[]) {
    int fd;

    // Check for required arguments
    if (argc < 4) {
        fprintf(stderr, "Usage: %s <tileset_filename> <level_filename> <level_width>\n", argv[0]);
        fprintf(stderr, "  tileset_filename: Binary file containing tileset data\n");
```

```
    fprintf(stderr, "  level_filename: Binary file containing level data in row-major format\n");
    fprintf(stderr, "  level_width: Width of the level in tiles\n");
    return -1;
}

// Parse the level width
level_width = atoi(argv[3]);
if (level_width <= 0) {
    fprintf(stderr, "Error: Level width must be a positive integer\n");
    return -1;
}

// Set up signal handlers for clean termination
signal(SIGINT, handle_signal);
signal(SIGTERM, handle_signal);

// Open the geo_dash device
fd = open("/dev/geo_dash", O_RDWR);
if (fd < 0) {
    perror("Error opening device");
    return -1;
}

// Load the tileset
if (load_tileset(fd, argv[1]) < 0) {
    fprintf(stderr, "Failed to load tileset\n");
    close(fd);
    return -1;
}

// Initialize the palette
initialize_palette(fd);

// Load the level
if (load_level(argv[2], level_width) < 0) {
    fprintf(stderr, "Failed to load level\n");
    close(fd);
    return -1;
}
```

```
        // Load game over
        if (load_gameover() < 0) {
                fprintf(stderr, "Failed to load %s\n", GAMEOVER_BIN);
                return 1;
        }

// Initialize the controller
controller_init();
printf("player y pos at beginning: %d\n", game.player_y);
// Initialize game state
initialize_game();

printf("Starting Geometry Dash. Press Ctrl+C to exit.\n");
printf("Level width: %d tiles\n", level_width);
printf("Controls: Press A button to jump\n");
printf("player y pos after initialization: %d\n", game.player_y);

// initial game state
game.player_x = PLAYER_START_X;
game.player_y = PLAYER_START_Y;
game.player_vy = 0;
game.is_jumping = false;
game.is_dead    = false;
game.level_x    = 0;

// prime the tilemap ring
initial_fill(fd);

        // Start game thread
if (pthread_create(&game_thread, NULL, game_loop, &fd) != 0) {
    perror("Failed to create game thread");
    cleanup_level();
    close(fd);
    return -1;
}

// Wait for game thread to finish
pthread_join(game_thread, NULL);
```

```
    // Clean up
    cleanup_level();
    close(fd);
    return 0;
}
```