CSEE4840 Embedded Systems
Final Report: ScreamJump

Ananya Maan Singh (am6542)
Sharwari Bhosale (sb5032)
Kamala Vennela Vasireddy (kv2446)

Contents:

1. System Overview

ScreamJump is a hardware-accelerated platformer game for the DE1-SoC board, featuring real-time VGA graphics and USB gamepad input. The game uses a tilemap background and animated sprite rendering via a custom SystemVerilog VGA driver. The ARM CPU handles user input and game logic, while the FPGA handles fast graphical output. The libusb is used to read USB game controller inputs.

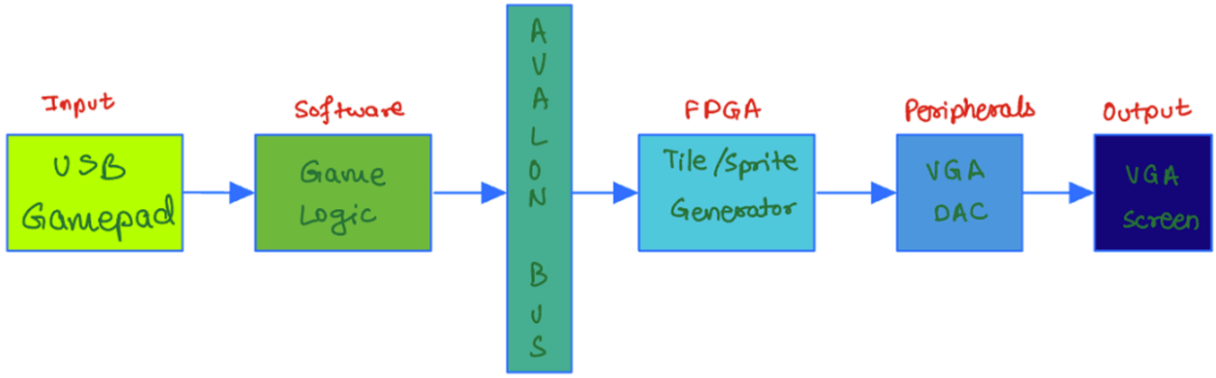


Figure 1 - Overall System Block Diagram

2. Hardware Design

2.1 VGA Controller Logic and Timing

The DE1-SoC's VGA controller uses a 50 MHz clock to drive a standard 640×480 @ 60 Hz display. Because the pixel clock for 640×480 VGA is 25 MHz, the controller outputs one pixel every two 50 MHz cycles. A horizontal counter (hcount) runs up to 1599 (which corresponds to 800 pixel cycles, since each pixel is two clock ticks), and a vertical counter (vcount) runs up to 524 (for 525 total scan lines) in each frame. The counters generate the sync pulses and blanking signals at the appropriate counts. For example, HACTIVE = 1280 counts (640 pixels), with additional counts for front porch, sync pulse, and back porch to total 1600 horizontal counts. Similarly, VACTIVE = 480 lines, plus porch and sync, for a total of 525 lines per frame. The VGA_counters module produces VGA_HS and VGA_VS pulses for sync and an active low blanking signal (VGA_BLANK_n) during the active display periods. The VGA_CLK output is generated by toggling the 50 MHz clock's phase (using VGA_CLK = hcount[0]) to produce a 25 MHz pixel clock.
During active display time, the controller reads pixel color data from a line buffer and drives the 8-bit DAC signals for red, green, and blue. The design uses 16-bit color (5-6-5 RGB). In hardware, the 5-bit red and blue and 6-bit green components are scaled to 8-bit by shifting left (multiplying by 8) before output. For instance, a pixel value with red bits r[4:0] is driven on VGA_R as r << 3 (e.g., 5-bit value 0x1F becomes 0xF8 on the DAC). The blanking signal (VGA_BLANK_n) ensures the DAC outputs are disabled during porch intervals so that the beam is off-screen. In summary, the VGA controller ensures correct timing for each scan line and frame, outputting pixels at 25 MHz and generating sync pulses so the monitor locks onto 640×480 @ 60 Hz timing.

2.1.1 Tiles and Sprites Configuration

- **Resolution**: 640×480 @ 60Hz
- **Tile Size**: 16×16 pixels, grid of 40×30 tiles = 1200 tiles
- **Sprite Size**: 32×32 pixels

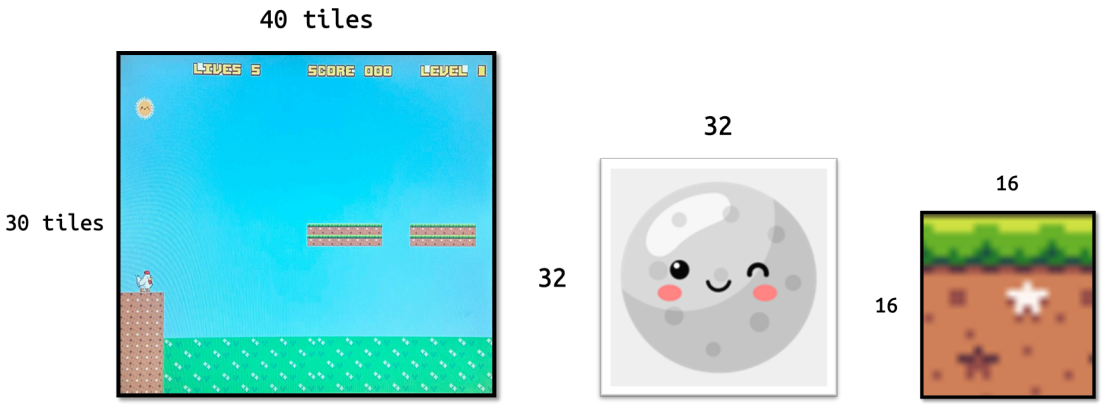


Figure 2.1.1 - Graphics and their specifications

2.2 Linebuffer Architecture

To generate each line of pixels on the fly, the design uses a dual-ported line buffer with double buffering. This allows one scanline of pixel data to be prepared while the previous line is sent to the VGA. The line buffer is a two on-chip RAMs named ram0 and ram1, each capable of holding one line of pixel data, 640 pixels. At any given time, one RAM is the display buffer feeding the VGA shift register for the current line, while the other is the draw buffer written with the next line's pixels. After a line is completed, the roles of the two buffers are swapped or toggled for the next line. This ping-pong buffering is controlled by a switch signal that pulses near the end of each line. When the switch is asserted, the hardware flips the display_index and draw_index pointers, so the just-filled buffer becomes the new display source, and the other buffer becomes the target for drawing the upcoming line. This swap occurs during the brief horizontal blanking interval, ensuring no contention between drawing and reading.

Each line buffer RAM has two write ports: one for tile-wide writes, 16 pixels at once, and one for single-pixel writes. Internally, the RAM is configured with two port widths: a 256-bit port, which is 16 pixels × 16 bits addressed by tile index 6-bit address, supporting up to 64 tiles per line, and a 16-bit port addressed by pixel index 10-bit address for up to 1024 pixels, though only 640 are used. The hardware takes advantage of this by writing an entire 16-pixel tile segment in one cycle on the wide port, and by writing individual sprite pixels on the narrow port. The line buffer module (linebuffer.sv) ties these ports to the draw or display side depending on the context: for the display buffer, the VGA controller supplies a pixel address to read out q_pixel_display the color for the current pixel, while for the draw buffer, the tile and sprite loader modules supply write addresses and data to build the next line's pixel values. The double-port, double-buffer scheme guarantees that pixel generation for the next line happens in

parallel with output of the current line, avoiding flicker or tearing. By the time the beam reaches the end of a line, the next line's buffer is fully drawn and ready to be swapped in.

- tile_loader: writes 40 tiles per line into the linebuffer
- sprite_loader: overlays active sprites per frame, skipping transparent pixels
- vga_top: synchronizes draw/display buffers with the VGA scanout timing

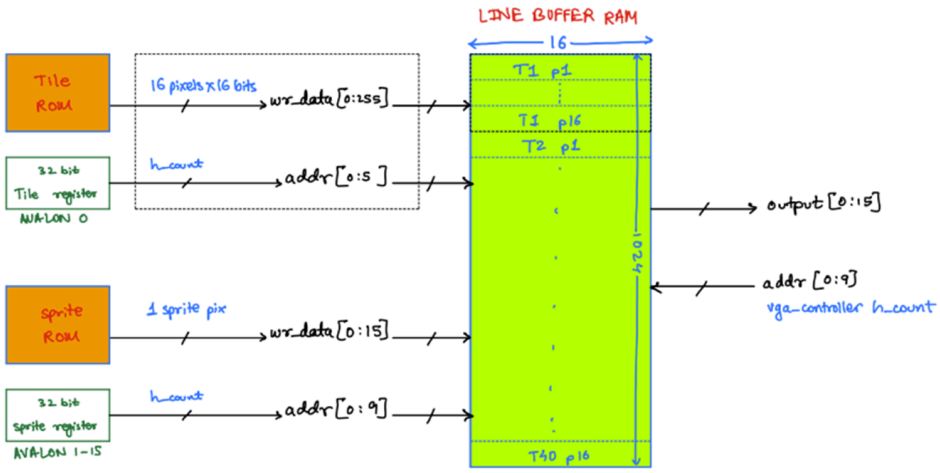


Figure 2.2.1 - Line Buffer Block Diagram

2.3 Tile Rendering

The tile background forms the static or scrolling backdrop of the game (e.g., ground, sky, platforms that are part of the level). The tile map is stored in a dedicated on-chip RAM called tile_array. Conceptually, this is a 2D array (e.g., 30 rows × 40 columns for a 640×480 screen of 16×16 tiles) that holds an 8-bit tile index for each tile position. The CPU (game software) updates this tile map by writing to a memory-mapped register at address offset 0. A 32-bit write to address 0 uses the lower 19 bits as:

- [18:14] = tile row
- [13:8] = tile column
- [7:0] = tile image number

This interface lets the software update any tile on the map, for example, to change the ground as the character moves or to place a new platform, by writing the tile's coordinates and index. When the write occurs, the hardware in tile_loader captures the data and writes it into the tile_array

RAM at the appropriate location. The tile_array is a dual-ported RAM so that the CPU, port B, can update tile entries asynchronously, while the tile loader, port A, can read the map to draw the screen.

During each active scan line, the Tile Loader module streams out the background pixels for that line. At the very start of a line (when hcount == 0 and if the line is in the visible region), the VGA controller asserts tile_start for one cycle. This triggers the tile loader state machine to run through 40 tile columns for the next line. The loader computes the starting address in tile_array corresponding to the upcoming line: it takes the vertical count of the next line (since we are drawing one line ahead due to buffering) and divides by 16 (>>4) to get the tile row, then multiplies by 40 and adds the tile column index. This gives the address of the tile's index in the map for the current column. It then reads the tile index (tile_img_num) from the tile map RAM. Using this index, the loader computes an address for the tile image ROM: tile_rom_address = tile_img_num * 16 + (line_in_tile). Each tile image is 16 pixels tall, so the tile image number is offset by the row within the tile (the lower 4 bits of the line number). The tile_rom is a ROM containing all tile graphics (for example, if there are 256 tile images, the ROM stores 256×16×16 pixels). It outputs 256 bits at a time, which is a 16-pixel row of the tile. The tile loader takes this 256-bit data and writes it into the line buffer's draw RAM via the wide port.

Bits	15 - 11b	10 - 6b	5 - 0b
	Red	Green	Blue

The tile loader iterates through columns 0 to 39, requesting each tile's pixel row in turn. Internally, a small counter column increments from 0 up to 39 for each tile segment. For each column, the module waits for the necessary cycles for the tile_rom to output the data, then asserts the write-enable (wren_tile_draw) to store the 16-pixel chunk at the appropriate position in the line buffer. The address for the line buffer's tile-port write (address_tile_draw) corresponds to the tile column index (0–39) being written. Because of pipelining, the first couple of cycles are spent initiating the first ROM read; the loader keeps address_tile_draw at 0 until valid data is ready, then proceeds to increment it each time a new 16-pixel block is written. By the time col reaches the last tile of the line, the entire 640-pixel line of background is in the draw buffer. The tile loader then raises a tile_finish flag to signal completion.

Some care is taken around the end of the active frame. The design skips drawing during the vertical blanking lines. In the tile loader (and sprite loader) logic, if the current vcount is 479 (last visible line index) or in the blanking interval, the loader simply sets finish immediately without drawing. Additionally, at the very bottom of the frame (when vcount == 524, the last line of the total frame), the logic wraps around and prepares the first line of the next frame (line 0). This ensures the draw buffer is ready when the new frame starts. In summary, the tile loader

provides a 40×16 pixel slice of background for each line by fetching tile IDs from tile_array and tile graphics from tile_rom and writing them into the line buffer. The result is the base pixel layer for that scanline, onto which sprites will later be overlaid.

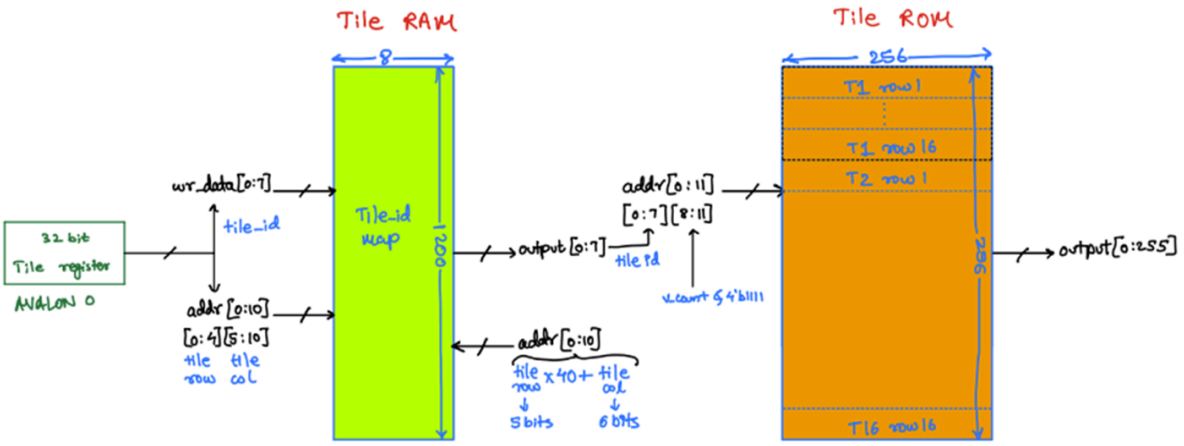


Figure 2.3.1 - Tile Rendering Logic Block Diagram

2.4 Sprite Rendering

Sprites represent moving objects (like the player's character "chick" or any enemies/obstacles) that are drawn on top of the background tiles. The design supports up to 16 sprites, and their properties are controlled by the CPU via 32-bit memory-mapped registers. Each sprite has a 25-bit control register (written at address offsets 1 through 16) with the following format: bit 24 = active flag, bits 23-15 = Y position, bits 14-5 = X position, bits 4-0 = image index. The active flag enables or disables the drawing of that sprite. X and Y specify the sprite's top-left coordinate on the screen with Y in [0,479] and X in [0,639]. The image index selects which sprite graphic to use. There can be up to 32 different 32×32 sprite images, since 5 bits are available for the index. When the CPU writes to a sprite register, for example, writes a 25-bit value to address 5 for sprite #4, the hardware captures the data and stores it in an array of registers, sprite_array, indexed by sprite number. In the hardware, sprite_number_write - 1 is used because the address 1 corresponds to index 0. The game software updates these registers every frame or as needed to move the sprites. For instance, updating the Y position to make the chick jump or fall, and toggling active flags for spawning or removing objects.

During the drawing of each scanline after the tile loader has finished the background, the Sprite Loader module overlays sprites onto the line buffer. When tile_finish is asserted, the VGA top-level logic raises sprite_start for one cycle, starting the sprite loader for that line. The sprite

loader will examine each sprite in order (from sprite 0 up to sprite 15) to see if it needs to draw it on the current line. It performs a loop: for each sprite index, first do a quick visibility check, then if the sprite is visible on this line, fetch its pixels and write them into the buffer.

Sprite visibility check:
This is handled by the sprite_active sub-module. The sprite loader sends the current sprite number and the actual line being drawn, the next line's index, similar to the tile loader's use of vcount+1, into sprite_active by pulsing sprite_active_start. The sprite_active module then reads that sprite's register from the sprite_array, which is populated by CPU writes, and checks if the sprite is active and within the vertical range of the line. Specifically, it checks sprite_array[sprite_num] active and compares the sprite's Y coordinate to the line: if actual_vcount >= sprite_y && actual_vcount < sprite_y + 32. If the current line lies between the sprite's top Y coordinate and bottom Y coordinate (sprite height is 32), then the sprite covers this line. In that case, sprite_active asserts is_active, the row_in_sprite, which is actual_vcount of sprite_y, indicates which row of the sprite's image to draw, the sprite's X position, sprite_column, and the sprite's image index (img_num). If the sprite is not active or not in range, it outputs is_active = 0. The sprite loader waits one cycle for this result using a small handshake where it raises sprite_active_start and then waits for sprite_active_finish. Once the check is done, the sprite loader has all the information needed for that sprite.

Sprite drawing:
If is_active is asserted for sprite N, the sprite loader will invoke the sprite_draw module to draw the sprite's pixels for this line. It sets sprite_draw_start to a one-cycle pulse to initiate drawing for that sprite. The sprite draw unit then begins reading the sprite image from the sprite ROM and writing pixels to the line buffer. Each sprite image is 32×32 pixels. The design uses a ROM that stores all sprite images, up to 32 images of 1024 pixels each. On start, the sprite draw computes the starting address in the sprite ROM for the needed row: sprite_rom_address = img_num * 1024 + row_in_sprite * 32. This points to the first pixel of that sprite's row. The module also initializes an internal column counter and sets the target X coordinate: on the first cycle of drawing, it loads pixel_hcount = sprite_column, the starting screen X position for the sprite.

As the sprite_draw module runs, it reads one pixel at a time from the sprite ROM, and the ROM outputs 16-bit pixel data. Each clock, it outputs a pixel color RGB encoding data along with a pixel address (pixel_hcount) to the line buffer draw port. It also increments the sprite ROM address to fetch the next pixel in the row and increments pixel_hcount unless the screen edge is reached. This continues for up to 32 pixels (the sprite's width). The sprite draw will stop early if it reaches the end of the line (X = 639) to avoid writing beyond the visible area. An important feature sprites have is transparency; sprite images can have transparent pixels, allowing the background or other sprites behind to show through. In hardware, the least significant bit of each sprite pixel is used as a transparency flag. By convention, if data[0] == 1, it means the pixel is

transparent, and if data[0] == 0, the pixel is opaque. The sprite draw logic only writes the pixel to the line buffer if it's not transparent: wren = (!finish) && (col > 0) && (data[0] == 0).(Here, col > 0 just skips the very first cycle while data is being fetched. Thus, for each pixel, if the sprite pixel is opaque, the module asserts wren_pixel_draw and writes the 16-bit color into the line buffer at the appropriate X position; if the pixel is marked transparent, the write enable remains 0 and the previous pixel value in the buffer which could be a tile or an earlier sprite is left untouched. In this way, the sprite is composed over the background with per-pixel transparency. After it has written all applicable pixels for sprite N's current line, the sprite draw asserts the sprite_draw_finish signal.

Bits	15 - 11b	10 - 6b	5 - 1b	0b
	Red	Green	Blue	Transparency

The sprite loader orchestrates these steps for each sprite in turn. It increments through sprite_number = 0 to 15 in a loop. For each sprite, it does the active check, then, if needed, the draw routine, moves to the next. The design processes sprites in increasing order. This effectively sets a priority: sprites with higher indices will be drawn later and thus appear on top of sprites with lower indices in overlapping regions. The time the sprite loader has checked all 16, it raises sprite_finish, indicating the line's draw buffer now contains the final pixel data (background tiles plus any opaque sprite pixels on top) for the next line.

A corner case is if a sprite's X or Y position causes it to be partially or fully off-screen. The sprite_active logic already ignores sprites whose Y range is outside the current line. If a sprite is above or below the visible frame entirely (Y < 0 or Y >= 480), the software simply marks it inactive. Horizontally, the sprite draw logic handles clipping at the right edge ($x \geq 640$) by checking pixel_hcount < 639 before incrementing and writing. The game engine would ensure sprites stay within bounds or deactivate them to avoid such cases.

In a nutshell, the sprite system takes each active sprite that intersects the scanline, fetches the appropriate row of its image from sprite_rom, and overlays those pixels into the line buffer over the background. Transparency bits in the pixel data allow selective merging so that non-rectangular sprites, like a character with a transparent background, render correctly. Once all sprites are processed, the draw buffer is complete for that line. When the VGA controller's counter reaches the end of the line, it triggers the buffer switch to swap the prepared buffer into the display slot, and the cycle repeats for the next line.

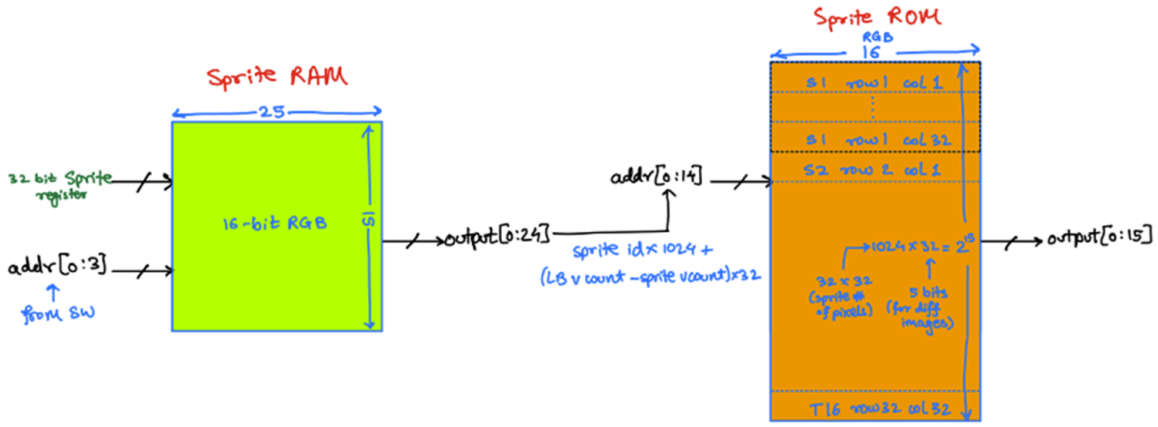


Figure 2.4.1 - Sprite Rendering Logic Block Diagram

2.5 Register Mapping (HW-SW Interface)

Tile and sprite writes are issued via ioctl() from userspace.

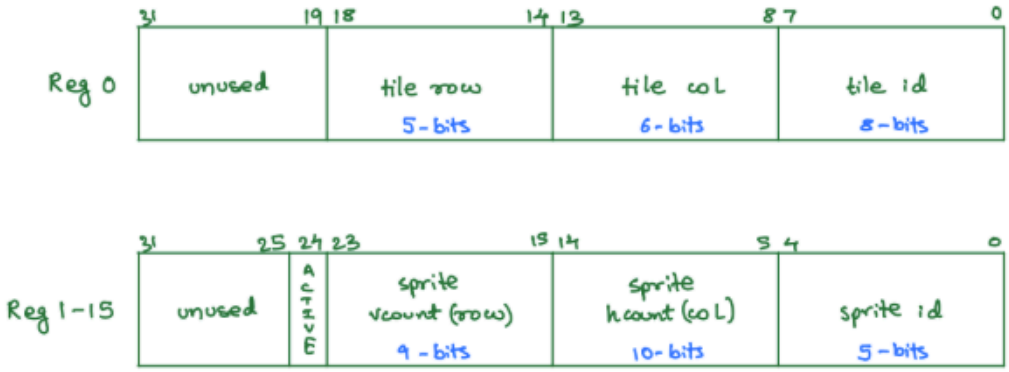


Figure 2.5.1 - Tile register (0) and Sprite registers (1 - 15)

2.6 Timing Diagram

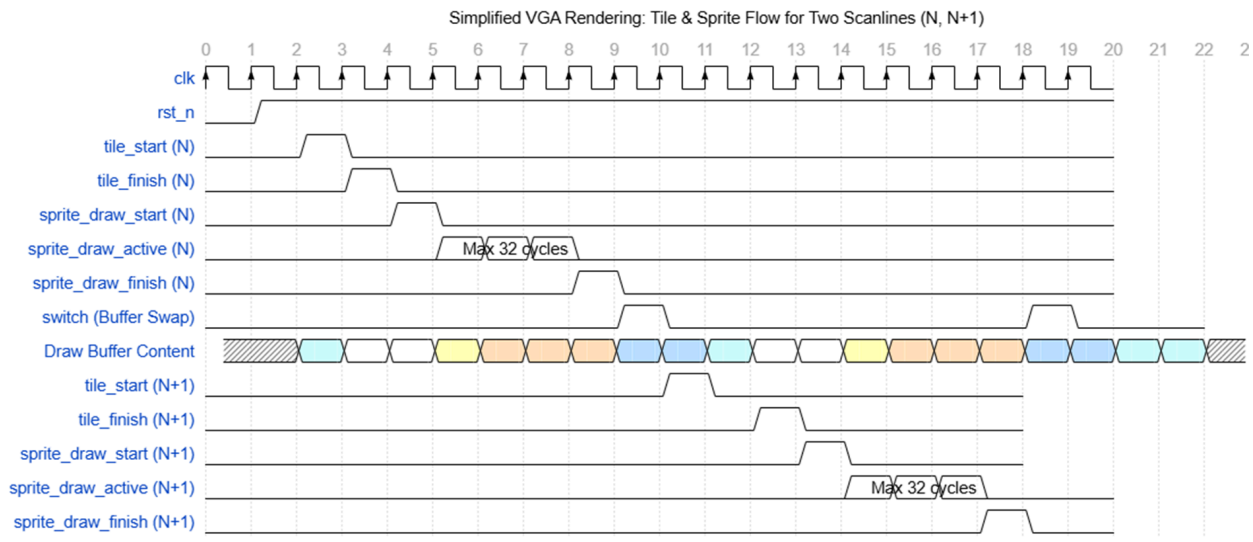


Figure 2.6.1

3. Software Design

3.1 Overview

Screamjump is a game where the player controls a chicken navigating through a vertically layered world of moving platforms. The objective is simply to survive as long as possible. The player can also collect coins and rack up points by successfully landing on the moving platforms as they progress towards the chicken. As the game progresses, the level difficulty increases, evoking a faster reaction time from the player. The player uses a USB gamepad controller connected to our DE1-SOC board. By pressing X, the player can start (or restart) the game and jump onto the moving platforms by pressing B.

3.2 Game State Machine Overview

The game consists of three major states: the start screen, the gameplay loop, and the end screen. In the start screen, the game displays a title using write_text() and waits for the player to press the X button (detected via controller_state.x). Once pressed, the game transitions into the gameplay state via a goto game_restart_point label. In the gameplay loop, the system continues to execute until the player loses all lives. Upon death, the end screen displays the final score and coin count, again waiting for a key press to restart the game.

3.3 USB Controller input

The player can play the game via the USB HID gamepad controller that is connected to the DE1-SoC board via its USB port. The controller communicates with the SoC using an 8-byte protocol.
The inputs are handled in a background thread using the function controller_input_thread(), and the 8-byte reports are read using libusb_interrupt_transfer(). For our game, we mapped the button inputs:

- B - For jumping
- X - To start/restart the game

3.4 Start Screen

At the beginning of execution, the game calls init_vga_interface() to initialize the VGA hardware interface. This sets up the tile and sprite buffers, clears any previous display data, and prepares the system for rendering. Simultaneously, the game spawns a new thread using pthread_create() to launch controller_input_thread(), which continuously polls a connected USB gamepad for input using the libusb library.

Once initialized, the game enters the start screen phase, during which it displays a title and prompt using write_text() calls — showing “scream jump” and “press X key to start.” The background tiles are drawn with either a daytime sky (fill_sky_and_grass()) or a night sky (fill_nightsky_and_grass()), depending on the current game level. The game remains in this idle state, looping until it detects a button press (controller_state.x), which transitions the player into the main game.

3.2 Main Game Logic

The player character is represented by a Chicken structure, which includes its X/Y position, vertical velocity, and flags for jumping and coin collection. The character can jump when on the ground or on a platform by pressing the B button, which triggers an upward velocity. Each frame, gravity is applied via vy += GRAVITY, and moveChicken() updates the position accordingly. If the player lands on a platform (handled via handleBarCollision()), the character stops falling and sets jumping = false. Coin collection is initiated when the player lands on a bar that has an active coin. A short delay counter (on_bar_collect_timer_us) is incremented until it reaches a defined threshold, the coin is deactivated, and the score is incremented.

The platforms, or "bars," are divided into two groups: barsA[] and barsB[]. Each group contains 10 bars and alternates in spawning waves. These bars scroll from right to left at a speed defined by the current level. Spawning is done by the main loop, which uses a level-dependent configuration to determine the number, size, and spacing of bars. For higher levels (Level 3 and

above), bar Y-positions are randomized within a range and clamped to screen limits. Bar movement is handled each frame by move_all_active_bars(), and collision detection is performed using handleBarCollision().

Coins are stored in the active_coins[] array and are spawned probabilistically on bars starting from Level 3. If a bar has a coin and the player lands on it, the coin enters a collection timer state. When the timer reaches the required delay, the coin is marked inactive, the score increases, and the coin is removed from the bar it was associated with.

Game difficulty increases dynamically through a level system. The game starts at Level 1 and increases every 10 points. Each level alters the gameplay by reducing bar sizes, increasing their speed, tightening their spacing, and adjusting coin spawn probabilities. It also shortens the jump initiation delay (usleep()) to challenge the player's reaction time. The background sky also changes from bright to night-themed at Level 3 and above.

Rendering is split between background tiles and dynamic sprites. The background is drawn using 16×16 pixel tiles updated with write_tile_to_kernel(). Each frame, the functions fill_sky_and_grass() or fill_nightsky_and_grass() fill the static elements, while draw_all_active_bars_to_back_buffer() places the active platforms. HUD text, such as "score," "lives," and "level," is written via write_text() and write_number() functions. These updates are finalized each frame by calling vga_present_frame().

Sprites are rendered on top of the tile layer using write_sprite_to_kernel_buffered(). This includes the chicken (with different sprites for jumping and standing), animated sun/moon sprites (updated each level with update_sun_sprite_buffered()), and coins. Before drawing, all previous sprites are cleared using clearSprites_buffered() to avoid ghosting. Only active sprites are redrawn per frame, improving performance and reducing flicker.

Scoring and HUD updates are done each frame. The score is displayed using digit tiles with write_number(), and coins collected are also shown upon game over. If the player's Y-position drops below the screen (i.e., falls off all platforms), one life is lost. When all lives are gone, the game transitions to the end screen. There, the "game over" text is shown along with the final score and coins collected, and the game waits for the player to press the X button to restart from the beginning.

3.3 Game Over

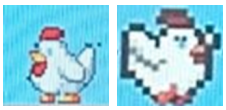
When the chicken's Y-position exceeds the bottom of the screen, the player loses one life. This is checked via:

```
if (chicken.y + CHICKEN_H > WIDTH - WALL) lives--;
```

if (lives == 0) → end screen

Thus, “game over” is displayed once all 5 lives of the player are over. The player can now view a report on their progress consisting of their final score and the number of coins they collected. The player is also shown the option to restart the game over here.

4. Resource Allocation

Feature	Graphics	Number	Size (pixels)	Total size
Chicken		2	32 x 32	32 x 32 x 2 x 16 = 32,768
Sun		1	32 x 32	32 x 32 x 16 = 16,384
Moon		1	32 x 32	32 x 32 x 16 = 16,384
Coin		1	32 x 32	32 x 32 x 16 = 16,384
Characters		35	16 x 16	16 x 16 x 35 x 16 = 143,360
Platforms		2	16 x 16	16 x 16 x 2 x 16 = 8,192
Skies		4	16 x 16	16 x 16 x 4 x 16 = 16,384
Grass		3	16 x 16	16 x 16 x 3 x 16 = 12,288
Total				= 262,144

5. Conclusion

ScreamJump successfully demonstrates a real-time interactive platformer game built from the ground up using hardware-software co-design principles. The project showcases how a custom VGA graphics pipeline, paired with an ARM processor running C code, can be used to deliver smooth, responsive gameplay on the DE1-SoC board.

Through the integration of tile-based rendering, sprite buffering, and USB gamepad input, we were able to build a complete game system that features dynamic difficulty scaling, platform collision detection, coin collection logic, and visual polish such as smooth scrolling and animated sprites. The modularity of the software, combined with careful timing and buffer management in the hardware, allowed for efficient rendering and responsive user control.

One of the key strengths of this project was the clear separation between the graphical pipeline and the game logic, allowing us to experiment with gameplay mechanics such as coin collection timers, jump delays, and randomized bar spawning without requiring changes to the underlying hardware modules.

In completing this project, we gained hands-on experience working at the boundary between software and hardware, deepened our understanding of memory-mapped I/O, double-buffered rendering, and real-time input polling, and developed an appreciation for the challenges of synchronization, timing, and performance in embedded game development.

ScreamJump is not just a demonstration of technical integration — it’s a playable, visually coherent game that reflects our team's creativity, attention to detail, and ability to work across abstraction layers. We hope future teams can build upon this foundation to explore even more advanced graphics systems, richer physics, or multiplayer interaction on FPGA-CPU hybrid platforms.

Team Member	**Contributions**
Ananya am6542	- Sprite and tile building, level-based background logic, testing
Sharwari sb5032	- Bar generation logic, difficulty scaling, testing
Kamala kv2446	- Tile scrolling logic, USB controller thread, coin mechanics, testing, (playing the game repeatedly)

5. Code Listing

C Files:

Demo.c

```C/C++
// Main game logic for ScreamJump.
// Handles player movement, level progression, obstacles, scoring, and VGA
output.
// Authored by: Ananya Maan Singh (am6542), Kamala Vennela Vasireddy (kv2446),
Sharwari Bhosale (sb5032)

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>
#include <pthread.h>
#include <time.h>
#include <fcntl.h>
#include <string.h>
#include <math.h> // For ceil and floor

#include "usbcontroller.h"
#include "vga_interface.h"

// Screen and physics
#define SCR_W              640   // VGA width
#define SCR_H              480   // VGA height
#define MARGIN             8     // Top/bottom margin (px)
#define GRAVITY            +1

// Sprite dimensions
#define CKN_W              32
#define CKN_H              32
#define COIN_W             32
#define COIN_H             32

// MIF indices
#define CKN_STAND_IDX      8
#define CKN_JUMP_IDX       9
#define TOWER_TILE_IDX     40
#define SUN_TILE_IDX       20
#define MOON_TILE_IDX      21
#define COIN_SPRITE_IDX    22
```

```
// Tower
#define TOWER_TOP_ROW      21
#define CKN_TOWER_Y        ((TOWER_TOP_ROW * TILE_SIZE) - CKN_H)

// Game settings
#define INIT_LIVES         5
#define JUMP_VEL           -20   // Initial jump velocity Y
#define BASE_JUMP_DELAY    2000  // us
#define LONG_JUMP_DELAY    4000  // us
#define PTS_PER_LVL        10
#define MAX_LVL            5
#define MAX_SCORE_DIGITS   3
#define MAX_COIN_DIGITS    2

// Bar properties
#define MAX_BARS           10    // Size of bar arrays
#define WAVE_SWITCH_OFF    70    // Offset px to trigger next wave spawn
#define BAR_H_ROWS         2     // Bar height in tiles
#define BAR_TILE_IDX       39
#define BAR_OFFSCREEN_X    -1000 // X pos for inactive bars
#define BAR_X_STAGGER_B    96    // Initial X stagger for bar group B

// Bar Y clamping
#define BAR_MIN_Y          180   // Min Y for bar top
#define BAR_MAX_Y          400   // Max Y for bar top

// Fixed Y for L1/L2 bars
#define L12_BAR_Y_A        240
#define L12_BAR_Y_B        200

// Relative Y offset for random bar groups (L3+)
#define BAR_Y_REL_OFF      150

// Coin properties
#define MAX_COINS          5     // Max on screen
#define COIN_POINTS        2
#define COIN_SPAWN_LVL     3     // Level when coins start spawning
#define COIN_SPAWN_CHANCE  100   // Percent chance to spawn coin on eligible
bar
#define COIN_COLLECT_DELAY (500) // Microseconds on bar to collect
#define FIRST_COIN_SPR_REG 2     // First sprite hardware register for coins

// Global variables
int vga_fd;
```

```
//int g_audio_fd;
struct controller_output_packet g_ctrl_state;
bool g_tower_on = true;
int  g_coins_total = 0;
bool g_do_restart = true;
int  g_level; // Current game level

// Structures
typedef struct {
    int x, y, vy;
    bool jumping;
    int coin_idx; // Index of coin being collected, or -1
    int coin_timer_us; // Timer for coin collection
} Chicken;

typedef struct {
    int x, y;     // Top-left pixel position
    int len;      // Length in tiles
    bool has_coin;
    int coin_idx; // Index in active_coins array, or -1
} Bar;

typedef struct {
    int bar_idx;    // Index of bar it's on
    int bar_grp;    // 0 for group A, 1 for group B
    bool active;
    int spr_reg;    // Sprite hardware register
} Coin;

Coin g_coins[MAX_COINS]; // Active coins on screen

// Function Prototypes
void draw_bars_buffered(Bar bars_a[], Bar bars_b[], int size);
void move_bars(Bar bars_a[], Bar bars_b[], int size, int speed);
bool check_bar_collision(Bar bars[], int grp_id, int size, int prev_y_ckn,
Chicken *ckn, int *score, bool *landed);
void *ctrl_thread(void *arg);
void init_ckn(Chicken *c);
void move_ckn(Chicken *c);
void update_sun_moon_sprite(int current_level);
void reset_bars(Bar bars[], int size);
void init_coins(void);
void draw_coins_buffered(Bar bars_a[], Bar bars_b[]);
```

```
void reset_for_death(Chicken *c, Bar bA[], Bar bB[], bool *tower_on, bool
*grpA_act, bool *needs_A, bool *needs_B, int *wA_idx, int *wB_idx, int
*next_sA, int *next_sB, int *last_y_A, int *last_y_B, bool *first_rand_wave);

void draw_bars_buffered(Bar bars_a[], Bar bars_b[], int size) {
    Bar* cur_grp;
    for (int grp = 0; grp < 2; grp++) {
        cur_grp = (grp == 0) ? bars_a : bars_b;
        for (int b = 0; b < size; b++) {
            if (cur_grp[b].x == BAR_OFFSCREEN_X || cur_grp[b].len == 0)
continue;
            int bar_px_w = cur_grp[b].len * TILE_SIZE;
            // Only draw if bar is somewhat on screen
            if (cur_grp[b].x < SCR_W && cur_grp[b].x + bar_px_w > 0) {
                int col0 = cur_grp[b].x / TILE_SIZE;
                int row0 = cur_grp[b].y / TILE_SIZE;
                int row1 = row0 + BAR_H_ROWS - 1;
                for (int r_tile = row0; r_tile <= row1; r_tile++) {
                    if (r_tile < 0 || r_tile >= TILE_ROWS) continue; // Bounds
check row
                    for (int i = 0; i < cur_grp[b].len; i++) {
                        int c_tile = col0 + i;
                        if (c_tile >= 0 && c_tile < TILE_COLS) // Bounds check
col
                            write_tile_to_kernel(r_tile, c_tile, BAR_TILE_IDX);
                    }
                }
            }
        }
    }
}

void move_bars(Bar bars_a[], Bar bars_b[], int size, int speed) {
    Bar* cur_grp;
    for (int grp = 0; grp < 2; grp++) {
        cur_grp = (grp == 0) ? bars_a : bars_b;
        for (int b = 0; b < size; b++) {
            if (cur_grp[b].x == BAR_OFFSCREEN_X) continue;
            cur_grp[b].x -= speed;
            int bar_px_w = cur_grp[b].len * TILE_SIZE;
            if (cur_grp[b].x + bar_px_w <= 0) { // Bar off-screen
                cur_grp[b].x = BAR_OFFSCREEN_X;
                if (cur_grp[b].has_coin && cur_grp[b].coin_idx != -1) {
                    int c_idx = cur_grp[b].coin_idx;
```

```
                    if (c_idx >= 0 && c_idx < MAX_COINS) { // Safe access
                        g_coins[c_idx].active = false;
                    }
                    cur_grp[b].has_coin = false;
                    cur_grp[b].coin_idx = -1;
                }
            }
        }
    }
}

bool check_bar_collision(Bar bars[], int grp_id, int size, int prev_y_ckn,
Chicken *ckn, int *score, bool *landed) {
    if (ckn->vy <= 0) return false; // Not falling or moving up

    for (int b = 0; b < size; b++) {
        if (bars[b].x == BAR_OFFSCREEN_X) continue;

        int bar_top = bars[b].y;
        int bar_bottom = bars[b].y + BAR_H_ROWS * TILE_SIZE;
        int bar_left = bars[b].x;
        int bar_right = bars[b].x + bars[b].len * TILE_SIZE;

        int ckn_bottom_prev = prev_y_ckn + CKN_H;
        int ckn_bottom_curr = ckn->y + CKN_H;
        int ckn_right = ckn->x + CKN_W;

        // Collision: passed through bar top, within horizontal and vertical
bounds
        if (ckn_bottom_prev <= bar_top && ckn_bottom_curr >= bar_top &&
            ckn_bottom_curr <= bar_bottom &&
            ckn_right > bar_left && ckn->x < bar_right) {
            ckn->y = bar_top - CKN_H; // Land on bar
            ckn->vy = 0;
            ckn->jumping = false;
            if (!(*landed)) {
                (*score)++;
                *landed = true;
            }
            // Check for coin on this bar
            if (bars[b].has_coin && bars[b].coin_idx != -1 &&
g_coins[bars[b].coin_idx].active) {
                ckn->coin_idx = bars[b].coin_idx;
                ckn->coin_timer_us = 0;
```

```
            } else {
                ckn->coin_idx = -1;
            }
            return true; // Collision detected
        }
    }
    return false; // No collision
}

void *ctrl_thread(void *arg) {
    uint8_t endpt_addr;
    struct libusb_device_handle *ctrl_handle = opencontroller(&endpt_addr);
    if (!ctrl_handle) {
        fprintf(stderr, "Ctrl thread: opencontroller() failed.\n");
        pthread_exit(NULL);
    }

    unsigned char buf[GAMEPAD_READ_LENGTH];
    int actual_len;
    while (g_do_restart) { // Loop while game session may restart
        int status = libusb_interrupt_transfer(ctrl_handle, endpt_addr, buf,
GAMEPAD_READ_LENGTH, &actual_len, 1000);
        if (status == LIBUSB_SUCCESS && actual_len == GAMEPAD_READ_LENGTH) {
            usb_to_output(&g_ctrl_state, buf);
        } else if (status == LIBUSB_ERROR_TIMEOUT) {
            continue; // Timeout is expected, just retry
        } else if (status == LIBUSB_ERROR_INTERRUPTED) {
            fprintf(stderr, "Ctrl thread: Transfer interrupted. Exiting.\n");
            break; // Interrupted, likely by shutdown
        } else {
            fprintf(stderr, "Ctrl thread: Read error: %s\n",
libusb_error_name(status));
            if (status == LIBUSB_ERROR_NO_DEVICE) {
                fprintf(stderr, "Ctrl thread: Device disconnected.\n");
                break; // Device gone, critical error
            }
            usleep(100000); // Brief pause on other errors
        }
    }
    printf("Ctrl thread: Cleaning up...\n");
    libusb_release_interface(ctrl_handle, 0);
    libusb_close(ctrl_handle);
    pthread_exit(NULL);
}
```

```
void init_ckn(Chicken *c) {
    c->x = 32; c->y = CKN_TOWER_Y; c->vy = 0; c->jumping = false;
    c->coin_idx = -1; c->coin_timer_us = 0;
}

void move_ckn(Chicken *c) {
    if (!c->jumping && g_tower_on) return; // Don't move if on tower and not
jumping
    c->y += c->vy;
    c->vy += GRAVITY;
}

void update_sun_moon_sprite(int current_level) {
    const int start_x = 32, end_x = 608, base_y = 64;
    double frac = (current_level > 1) ? (double)(current_level - 1) / (MAX_LVL
- 1) : 0.0;
    if (current_level >= MAX_LVL) frac = 1.0;

    int sprite_x = start_x + (int)((end_x - start_x) * frac + 0.5);
    // Sprite reg 1 is for sun/moon
    write_sprite_to_kernel_buffered(1, base_y, sprite_x, (g_level >=4 ?
MOON_TILE_IDX : SUN_TILE_IDX), 1);
}

#include <stdlib.h>
#include <time.h>
#include <stdbool.h>

// … your other includes …

#include <stdlib.h>
#include <time.h>
#include <stdbool.h>

// … your other includes …


void reset_bars(Bar bars[], int size) {
    for (int i = 0; i < size; i++) {
        bars[i].x = BAR_OFFSCREEN_X; bars[i].len = 0;
        bars[i].has_coin = false; bars[i].coin_idx = -1;
    }
}
```

```
void init_coins(void) {
    for (int i = 0; i < MAX_COINS; i++) {
        g_coins[i].active = false; g_coins[i].bar_idx = -1;
        g_coins[i].bar_grp = -1;
        g_coins[i].spr_reg = FIRST_COIN_SPR_REG + i;
    }
}

void draw_coins_buffered(Bar bars_a[], Bar bars_b[]) {
    for (int i = 0; i < MAX_COINS; i++) {
        if (g_coins[i].active) {
            Bar *parent_bars = (g_coins[i].bar_grp == 0) ? bars_a : bars_b;
            int bar_idx = g_coins[i].bar_idx;

            // Check if the bar this coin is on is still active and valid
            if (bar_idx != -1 && bar_idx < MAX_BARS && parent_bars[bar_idx].x
!= BAR_OFFSCREEN_X && parent_bars[bar_idx].coin_idx == i) {
                int bar_center_x = parent_bars[bar_idx].x +
(parent_bars[bar_idx].len * TILE_SIZE) / 2;
                int coin_x = bar_center_x - (COIN_W / 2);
                int coin_y = parent_bars[bar_idx].y - COIN_H - (TILE_SIZE / 4);
// Position above bar

                bool on_screen_x = (coin_x + COIN_W > 0) && (coin_x < SCR_W);
                bool on_screen_y = (coin_y + COIN_H > 0) && (coin_y < SCR_H);

                if (on_screen_x && on_screen_y) {
                    write_sprite_to_kernel_buffered(1, coin_y, coin_x,
COIN_SPRITE_IDX, g_coins[i].spr_reg);
                } else { // Active but off-screen
                    write_sprite_to_kernel_buffered(0, 0, 0, 0,
g_coins[i].spr_reg); // Hide
                }
            } else { // Coin active, but its parent bar is gone/invalid.
Deactivate.
                write_sprite_to_kernel_buffered(0, 0, 0, 0,
g_coins[i].spr_reg); // Hide
                g_coins[i].active = false;
            }
        } else { // Coin not active, ensure sprite is hidden
            write_sprite_to_kernel_buffered(0, 0, 0, 0, g_coins[i].spr_reg);
        }
    }
```

```
}


void reset_for_death(Chicken *c, Bar bA[], Bar bB[], bool *tower_on,
                     bool *grpA_act, bool *needs_A, bool *needs_B,
                     int *watch_idx_A, int *watch_idx_B, int *next_slot_A, int
*next_slot_B,
                     int *last_y_A, int *last_y_B, bool *first_rand_wave) {
    init_ckn(c);
    *tower_on = true;
    reset_bars(bA, MAX_BARS); reset_bars(bB, MAX_BARS);
    init_coins(); // Reset all coins

    *grpA_act = true; *needs_A = true; *needs_B = false;
    *watch_idx_A = -1; *watch_idx_B = -1;
    *next_slot_A = 0; *next_slot_B = 0;

    *last_y_A = L12_BAR_Y_A;
    *last_y_B = L12_BAR_Y_B;
    *first_rand_wave = true;

    cleartiles();
    fill_sky_and_grass();
    clearSprites_buffered();
}


int main(void) {
    // Static vars for bar Y positions, persist across deaths, reset on new
game.
    static int s_last_y_a = L12_BAR_Y_A;
    static int s_last_y_b = L12_BAR_Y_B;
    static bool s_first_rand_wave = true;

    int score, lives;

    if ((vga_fd = open("/dev/vga_top", O_RDWR)) < 0) { perror("VGA open");
return -1; }
    init_vga_interface();

    pthread_t ctrl_tid;
    g_do_restart = true; // Controller thread runs as long as game can restart
    if (pthread_create(&ctrl_tid, NULL, ctrl_thread, NULL) != 0) {
        perror("Ctrl thread create"); close(vga_fd); return -1;
```

```
    }

game_restart_point: // Label for full game restart
    score = 0;
    g_level = 1;
    lives = INIT_LIVES;
    g_coins_total = 0;
    init_coins(); // Clear any existing coins from previous game

    // Reset static Y tracking for new game
    s_last_y_a = L12_BAR_Y_A;
    s_last_y_b = L12_BAR_Y_B;
    s_first_rand_wave = true;
    g_tower_on = true; // Reset tower state for new game start

    cleartiles(); clearSprites_buffered();
    fill_sky_and_grass(); // Initial screen
    write_text("scream", 6, 13, 16); write_text("jump", 4, 13, 22);
    write_text("press", 5, 16, 12); write_text("x", 1, 16, 18);
    write_text("key", 3, 16, 20); write_text("to", 2, 16, 24);
    write_text("start", 5, 16, 27);
    vga_present_frame(); present_sprites();

    while (!g_ctrl_state.x) { usleep(10000); } // Wait for X press
    usleep(200000); // Debounce
    while (g_ctrl_state.x) { usleep(10000); } // Wait for X release

    cleartiles(); clearSprites_buffered();
    fill_sky_and_grass(); // Background for level 1 start

    srand(time(NULL));
    int jump_vy = JUMP_VEL;
    const int hud_col = TILE_COLS / 2; const int hud_off = 12;

    Bar bars_a[MAX_BARS], bars_b[MAX_BARS];
    reset_bars(bars_a, MAX_BARS); reset_bars(bars_b, MAX_BARS);

    int min_bar_len, max_bar_len, bar_count, bar_speed, bar_spacing_px;
    int fixed_y_a, fixed_y_b; // For L1 & L2
    int jump_delay;

    Chicken ckn; init_ckn(&ckn);
    bool landed_jump = false;
    bool grp_a_spawns = true; bool spawn_a = true; bool spawn_b = false;
```

```
    int next_slot_a = 0, next_slot_b = 0;
    int watch_idx_a = -1, watch_idx_b = -1;

    while (lives > 0) {
        // Update game level based on score
        int old_level = g_level;
        g_level = 1 + (score / PTS_PER_LVL);
        if (g_level > MAX_LVL) g_level = MAX_LVL;

                // If level changed, maybe update background (already handled
by general draw)
        if (old_level != g_level) {
           if (g_level > 3 && old_level < 3) fill_nightsky_and_grass(); //
Transition to night
               else if (g_level == 3 ) fill_evesky_and_grass(); //transition to
eve
           else if (g_level < 3 && old_level >3) fill_sky_and_grass(); //
Transition to day
        }

        // Level-specific settings
        switch (g_level) {
            case 1:
                min_bar_len = 7; max_bar_len = 8; bar_count = 4; bar_speed = 3;
                bar_spacing_px = 170; fixed_y_a = L12_BAR_Y_A; fixed_y_b =
L12_BAR_Y_B;
                jump_delay = LONG_JUMP_DELAY;
                break;
            case 2:
                min_bar_len = 6; max_bar_len = 8; bar_count = 3; bar_speed = 3;
                bar_spacing_px = 180; fixed_y_a = L12_BAR_Y_A; fixed_y_b =
L12_BAR_Y_B;
                jump_delay = LONG_JUMP_DELAY;
                break;
            case 3:
                min_bar_len = 5; max_bar_len = 7; bar_count = 3; bar_speed = 4;
                bar_spacing_px = 160; /* Y is random */ jump_delay =
LONG_JUMP_DELAY;
                break;
            case 4:
                min_bar_len = 5; max_bar_len = 6; bar_count = 2; bar_speed = 4;
                bar_spacing_px = 190; /* Y is random */ jump_delay =
BASE_JUMP_DELAY;
                break;
```

```
            case 5: default:
                min_bar_len = 5; max_bar_len = 6; bar_count = 2; bar_speed = 5;
                bar_spacing_px = 190; /* Y is random */ jump_delay =
BASE_JUMP_DELAY;
                break;
        }

        update_grass_scroll(bar_speed); // Scroll grass based on effective bar
speed

        // Handle jump input
        if (g_ctrl_state.b && !ckn.jumping) {
            ckn.vy = jump_vy; ckn.jumping = true;
            landed_jump = false; g_tower_on = false; // play_sfx(0);
            if(ckn.coin_idx != -1) {
                ckn.coin_timer_us = 0; ckn.coin_idx = -1;
            }
            usleep(jump_delay);
        }

        int prev_y_ckn = ckn.y;
        move_ckn(&ckn);
        move_bars(bars_a, bars_b, MAX_BARS, bar_speed);

        // Spawn bar wave A
        if (grp_a_spawns && spawn_a) {
            int y_a;
            if (g_level >= 3) { // Random Y for L3+
                if (s_first_rand_wave) {
                    y_a = L12_BAR_Y_A; // First random wave starts at a known Y
                    s_first_rand_wave = false;
                } else {
                    y_a = s_last_y_b + (rand() % (2 * BAR_Y_REL_OFF + 1)) -
BAR_Y_REL_OFF;
                }
            } else { // Fixed Y for L1/L2
                y_a = fixed_y_a;
            }
            y_a = (y_a < BAR_MIN_Y) ? BAR_MIN_Y : (y_a > BAR_MAX_Y) ? BAR_MAX_Y
: y_a; // Clamp Y
            y_a = (y_a / TILE_SIZE) * TILE_SIZE; // Align to tile grid
            s_last_y_a = y_a;

            int spawned = 0, last_idx = -1;
```

```
            for (int i = 0; i < bar_count; i++) {
                int slot = -1; // Find available slot in bars_a
                for (int j = 0; j < MAX_BARS; j++) {
                    int cur = (next_slot_a + j) % MAX_BARS;
                    if (bars_a[cur].x == BAR_OFFSCREEN_X) { slot = cur; break;
}
                }
                if (slot != -1) {
                    bars_a[slot].x = SCR_W + (i * bar_spacing_px);
                    bars_a[slot].y = y_a;
                    bars_a[slot].len = rand() % (max_bar_len - min_bar_len + 1)
+ min_bar_len;
                    bars_a[slot].has_coin = false; bars_a[slot].coin_idx = -1;
                    if (g_level >= COIN_SPAWN_LVL && (rand() % 100) <
COIN_SPAWN_CHANCE) {
                        for (int c_idx = 0; c_idx < MAX_COINS; c_idx++) { //
Find inactive coin
                            if (!g_coins[c_idx].active) {
                                g_coins[c_idx].active = true;
g_coins[c_idx].bar_idx = slot;
                                g_coins[c_idx].bar_grp = 0; // Group A
                                bars_a[slot].has_coin = true;
bars_a[slot].coin_idx = c_idx;
                                break;
                            }
                        }
                    }
                    last_idx = slot; spawned++;
                } else break; // No slot
            }
            if (spawned > 0) { watch_idx_a = last_idx; next_slot_a = (last_idx
+ 1) % MAX_BARS; }
            spawn_a = false;
        }
        // Spawn bar wave B
        else if (!grp_a_spawns && spawn_b) {
            int y_b;
            if (g_level >= 3) { // Random Y for L3+
                 // s_first_rand_wave only applies to group A's first random
wave in a session/reset
                y_b = s_last_y_a + (rand() % (2 * BAR_Y_REL_OFF + 1)) -
BAR_Y_REL_OFF;
            } else { // Fixed Y for L1/L2
                y_b = fixed_y_b;
```

```
            }
            y_b = (y_b < BAR_MIN_Y) ? BAR_MIN_Y : (y_b > BAR_MAX_Y) ? BAR_MAX_Y
: y_b; // Clamp Y
            y_b = (y_b / TILE_SIZE) * TILE_SIZE; // Align to tile grid
            s_last_y_b = y_b;

            int spawned = 0, last_idx = -1;
            for (int i = 0; i < bar_count; i++) {
                int slot = -1;
                for (int j = 0; j < MAX_BARS; j++) {
                    int cur = (next_slot_b + j) % MAX_BARS;
                    if (bars_b[cur].x == BAR_OFFSCREEN_X) { slot = cur; break;
}
                }
                if (slot != -1) {
                    bars_b[slot].x = SCR_W + BAR_X_STAGGER_B + (i *
bar_spacing_px);
                    bars_b[slot].y = y_b;
                    bars_b[slot].len = rand() % (max_bar_len - min_bar_len + 1)
+ min_bar_len;
                    bars_b[slot].has_coin = false; bars_b[slot].coin_idx = -1;
                     if (g_level >= COIN_SPAWN_LVL && (rand() % 100) <
COIN_SPAWN_CHANCE) {
                        for (int c_idx = 0; c_idx < MAX_COINS; c_idx++) { //
Find inactive coin
                            if (!g_coins[c_idx].active) {
                                g_coins[c_idx].active = true;
g_coins[c_idx].bar_idx = slot;
                                g_coins[c_idx].bar_grp = 1; // Group B
                                bars_b[slot].has_coin = true;
bars_b[slot].coin_idx = c_idx;
                                break;
                            }
                        }
                    }
                    last_idx = slot; spawned++;
                } else break;
            }
            if (spawned > 0) { watch_idx_b = last_idx; next_slot_b = (last_idx
+ 1) % MAX_BARS; }
            spawn_b = false;
        }

        // Switch active spawner group
```

```
        if (grp_a_spawns && watch_idx_a != -1 && bars_a[watch_idx_a].x !=
BAR_OFFSCREEN_X) {
            if (bars_a[watch_idx_a].x < SCR_W - WAVE_SWITCH_OFF) {
                grp_a_spawns = false; spawn_b = true; watch_idx_a = -1;
            }
        } else if (!grp_a_spawns && watch_idx_b != -1 && bars_b[watch_idx_b].x
!= BAR_OFFSCREEN_X) {
            if (bars_b[watch_idx_b].x < SCR_W - WAVE_SWITCH_OFF) {
                grp_a_spawns = true; spawn_a = true; watch_idx_b = -1;
            }
        }

        // Coin collection logic
        if (ckn.coin_idx != -1 && !ckn.jumping) {
            ckn.coin_timer_us += 16666;
            if (ckn.coin_timer_us >= COIN_COLLECT_DELAY) {
                Coin* coin = &g_coins[ckn.coin_idx];
                if (coin->active) {
                    score += (COIN_POINTS -1);
                    g_coins_total++;
                    coin->active = false; // Deactivate coin itself

                    // Remove coin from bar
                    Bar* parent_bars = (coin->bar_grp == 0) ? bars_a : bars_b;
                    if(coin->bar_idx != -1 && coin->bar_idx < MAX_BARS &&
parent_bars[coin->bar_idx].coin_idx == ckn.coin_idx) {
                        parent_bars[coin->bar_idx].has_coin = false;
                        parent_bars[coin->bar_idx].coin_idx = -1;
                    }
                }
                ckn.coin_idx = -1; ckn.coin_timer_us = 0;
            }
        } else if (ckn.coin_idx != -1 && ckn.jumping) {
            ckn.coin_timer_us = 0; ckn.coin_idx = -1;
        }

        // Check collision with bars
        if (ckn.vy > 0) { // Only check if falling
            bool landed_a = check_bar_collision(bars_a, 0, MAX_BARS,
prev_y_ckn, &ckn, &score, &landed_jump);
            if (!landed_a) check_bar_collision(bars_b, 1, MAX_BARS, prev_y_ckn,
&ckn, &score, &landed_jump);
        }
```

```
        // Boundary checks for chicken
        if (ckn.y < MARGIN && ckn.jumping) { ckn.y = MARGIN; if (ckn.vy < 0)
ckn.vy = 0; } // Hit ceiling
        if (ckn.y + CKN_H > SCR_H - MARGIN) {
            lives--;
            if (lives > 0) {
                reset_for_death(&ckn, bars_a, bars_b, &g_tower_on,
                                &grp_a_spawns, &spawn_a, &spawn_b,
                                &watch_idx_a, &watch_idx_b, &next_slot_a,
&next_slot_b,
                                &s_last_y_a, &s_last_y_b, &s_first_rand_wave);
                vga_present_frame(); present_sprites();
                usleep(1000000); // Pause after death (1 sec)
                continue;
            }
        }
        if (g_level > 3) fill_nightsky_and_grass();
            else if (g_level == 3) fill_evesky_and_grass();
        else fill_sky_and_grass();
        //fill_sky_and_grass();
        draw_bars_buffered(bars_a, bars_b, MAX_BARS);

        write_text("lives", 5, 1, hud_col - hud_off);
        write_number(lives, 1, hud_col - hud_off + 6);
        write_text("score", 5, 1, hud_col - hud_off + 12);
        write_numbers(score, MAX_SCORE_DIGITS, 1, hud_col - hud_off + 18);
        write_text("level", 5, 1, hud_col - hud_off + 24);
        write_number(g_level, 1, hud_col - hud_off + 30);

        if (g_tower_on) {
            for (int r = TOWER_TOP_ROW; r < TOWER_TOP_ROW + 9; ++r) {
                 if (r >= TILE_ROWS ) break;
                for (int c_tower = 0; c_tower < 5; ++c_tower) {
write_tile_to_kernel(r, c_tower, TOWER_TILE_IDX); }
            }
        }
        vga_present_frame(); // Push tilemap buffer to screen

        clearSprites_buffered(); // Prepare sprite buffer
        write_sprite_to_kernel_buffered(1, ckn.y, ckn.x, ckn.jumping ?
CKN_JUMP_IDX : CKN_STAND_IDX, 0); // Chicken is sprite 0
        update_sun_moon_sprite(g_level); // Sun/Moon is sprite 1
        draw_coins_buffered(bars_a, bars_b);
        present_sprites();
```

```
        usleep(16666); // ~60 FPS
    }


    cleartiles();
    fill_sky_and_grass();

    clearSprites_buffered();
    write_text("game", 4, 13, 16); write_text("over", 4, 13, 21);
    write_text("score", 5, 15, 16); write_numbers(score, MAX_SCORE_DIGITS, 15,
22);
    write_text("coins", 5, 17, 11); write_text("collected", 9, 17, 17);
write_numbers(g_coins_total, MAX_COIN_DIGITS, 17, 27);
    write_text("press", 5, 19, 9); write_text("x", 1, 19, 15);
    write_text("key", 3, 19, 17); write_text("to", 2, 19, 21);
    write_text("restart", 7, 19, 24);
    vga_present_frame(); present_sprites();

    memset(&g_ctrl_state, 0, sizeof(g_ctrl_state));
    usleep(100000); // Debounce

    while(1) { // Wait for restart command
        if (g_ctrl_state.x) {
            goto game_restart_point; // Restart the game
        }
        usleep(50000);
    }


    close(vga_fd);
    return 0;
}
```

vga_interface.c

```C/C++
//LAST WORKING REVERT 2.3 of A
#include "vga_top.h"
#include "vga_interface.h"
#include <stdio.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>        // For rand()
#include <stdbool.h>

// --- Software Double Buffering for Tiles ---
static unsigned char display_buffer_A[TILE_ROWS][TILE_COLS];
static unsigned char display_buffer_B[TILE_ROWS][TILE_COLS];
static unsigned char (*current_back_buffer)[TILE_COLS] = display_buffer_A;
static unsigned char (*current_front_buffer)[TILE_COLS] = display_buffer_B;

// Shadow map for actual hardware tile writes
static unsigned char shadow_hardware_map[TILE_ROWS][TILE_COLS];
static bool vga_initialized = false;

// --- Sprite State Buffering ---
static SpriteHWState desired_sprite_states[MAX_HARDWARE_SPRITES];
static SpriteHWState actual_hw_sprites[MAX_HARDWARE_SPRITES];

// MODIFICATION: Variables for scrolling grass effect
static int grass_pixel_scroll_accumulator = 0;
static int grass_current_tile_shift = 0; // How many full tiles the grass
pattern has shifted


static void vga_hardware_write_tile(unsigned char r, unsigned char c, unsigned
char n) {
    if (r >= TILE_ROWS || c >= TILE_COLS) return;
    if (shadow_hardware_map[r][c] == n) return;

    vga_top_arg_t vla;
    vla.r = r; vla.c = c; vla.n = n;
    if (ioctl(vga_fd, VGA_TOP_WRITE_TILE, &vla)) {
        perror("ioctl(VGA_TOP_WRITE_TILE) failed in vga_hardware_write_tile");
```

```
        return;
    }
    shadow_hardware_map[r][c] = n;
}

static void vga_hardware_write_sprite(unsigned char active, unsigned short r,
unsigned short c, unsigned char n, unsigned short register_n) {
  if (register_n >= MAX_HARDWARE_SPRITES) return;

  vga_top_arg_s vla;
  vla.active = active; vla.r = r; vla.c = c; vla.n = n; vla.register_n =
register_n;
  if (ioctl(vga_fd, VGA_TOP_WRITE_SPRITE, &vla)) {
    // perror("ioctl(VGA_TOP_WRITE_SPRITE) failed in
vga_hardware_write_sprite");
  }
}


void init_vga_interface(void) {
    if (vga_initialized) return;

    for (int r = 0; r < TILE_ROWS; r++) {
        for (int c = 0; c < TILE_COLS; c++) {
            display_buffer_A[r][c] = BLANKTILE;
            display_buffer_B[r][c] = BLANKTILE;
            shadow_hardware_map[r][c] = 0xFF;
        }
    }
    current_back_buffer = display_buffer_A;
    current_front_buffer = display_buffer_B;

    for (int r = 0; r < TILE_ROWS; r++) {
        for (int c = 0; c < TILE_COLS; c++) {
            vga_hardware_write_tile(r, c, BLANKTILE);
        }
    }

    for (int i = 0; i < MAX_HARDWARE_SPRITES; i++) {
        desired_sprite_states[i] = (SpriteHWState){.active = false, .r = 0, .c
= 0, .n = 0};
        actual_hw_sprites[i]   = (SpriteHWState){.active = false, .r = 0, .c =
0, .n = 0};
        vga_hardware_write_sprite(0, 0, 0, 0, i);
```

```
    }
    grass_pixel_scroll_accumulator = 0; // Initialize grass scroll variables
    grass_current_tile_shift = 0;
    vga_initialized = true;
}

void write_tile_to_kernel(unsigned char r, unsigned char c, unsigned char n) {
  if (!vga_initialized) return;
  if (r >= TILE_ROWS || c >= TILE_COLS) return;
  current_back_buffer[r][c] = n;
}

void vga_present_frame(void) {
    if (!vga_initialized) return;
    for (int r = 0; r < TILE_ROWS; r++) {
        for (int c = 0; c < TILE_COLS; c++) {
            if (current_back_buffer[r][c] != shadow_hardware_map[r][c]) {
                 vga_hardware_write_tile(r, c, current_back_buffer[r][c]);
            }
        }
    }
    unsigned char (*temp_buffer)[TILE_COLS] = current_front_buffer;
    current_front_buffer = current_back_buffer;
    current_back_buffer = temp_buffer;
}

void write_sprite_to_kernel_buffered(unsigned char active, unsigned short r,
unsigned short c, unsigned char n, unsigned short register_n) {
    if (!vga_initialized || register_n >= MAX_HARDWARE_SPRITES) return;
    desired_sprite_states[register_n].active = active;
    desired_sprite_states[register_n].r = r;
    desired_sprite_states[register_n].c = c;
    desired_sprite_states[register_n].n = n;
}

void present_sprites(void) {
    if (!vga_initialized) return;
    for (int i = 0; i < MAX_HARDWARE_SPRITES; i++) {
        if (desired_sprite_states[i].active != actual_hw_sprites[i].active ||
            (desired_sprite_states[i].active &&
             (desired_sprite_states[i].r != actual_hw_sprites[i].r ||
              desired_sprite_states[i].c != actual_hw_sprites[i].c ||
              desired_sprite_states[i].n != actual_hw_sprites[i].n))) {
```

```
            vga_hardware_write_sprite(
                desired_sprite_states[i].active,
                desired_sprite_states[i].r,
                desired_sprite_states[i].c,
                desired_sprite_states[i].n,
                i
            );
            actual_hw_sprites[i] = desired_sprite_states[i];
        }
    }
}

void cleartiles() {
  if (!vga_initialized) return;
  for(int i=0; i < TILE_ROWS; i++) {
     for(int j=0; j < TILE_COLS; j++) {
        current_back_buffer[i][j] = BLANKTILE;
     }
  }
}

void clearSprites_buffered(){
  if (!vga_initialized) return;
  for(int i = 0; i < MAX_HARDWARE_SPRITES; i++){
    desired_sprite_states[i].active = false;
  }
}

void write_number(unsigned int num, unsigned int row, unsigned int col) {
  if (num > 9) num = 9;
  write_tile_to_kernel((unsigned char) row, (unsigned char) col,
NUMBERTILE(num));
}

void write_letter(unsigned char letter, unsigned int row, unsigned int col) {
  if (letter >= 'a' && letter <= 'z') {
    letter = letter - 'a';
    write_tile_to_kernel(row, col, LETTERTILE(letter));
  } else {
    write_tile_to_kernel(row, col, BLANKTILE);
  }
}
```

```c
void write_numbers(unsigned int nums, unsigned int digits, unsigned int row,
unsigned int col) {
  if ((col + digits) > TILE_COLS) return;
  if (digits == 0 || digits > 10) digits = 1;
  char temp_num_str[11];
  sprintf(temp_num_str, "%0*u", digits, nums);
  for (unsigned int i = 0; i < digits; i++) {
      if ((col + i) < TILE_COLS) {
          write_number(temp_num_str[i] - '0', row, col + i);
      } else { break; }
  }
}

void write_score(int new_score) {
  if (new_score < 0) new_score = 0;
  write_numbers((unsigned int) new_score, SCORE_MAX_LENGTH, SCORE_COORD_R,
SCORE_COORD_C);
}

void write_text(unsigned char *text, unsigned int length, unsigned int row,
unsigned int col) {
  if (!text) return;
  unsigned int write_len = length;
  if ((col + length) > TILE_COLS) {
    write_len = TILE_COLS - col;
  }
  for (unsigned int i = 0; i < write_len; i++) {
    if ((col + i) < TILE_COLS) {
        write_letter(*(text + i), row, col + i);
    } else { break; }
  }
}

// NEW: Updates the grass scroll offset based on speed.
// This should be called once per game frame from main.c
void update_grass_scroll(int scroll_speed_px) {
    if (!vga_initialized) return; // Ensure interface is ready

    grass_pixel_scroll_accumulator += scroll_speed_px;

    // Calculate how many full tiles we need to shift the pattern
    int tiles_to_shift_this_frame = grass_pixel_scroll_accumulator / TILE_SIZE;
```

```
    if (tiles_to_shift_this_frame != 0) { // Check if it's non-zero to handle
negative speeds correctly too
        grass_current_tile_shift += tiles_to_shift_this_frame;
        // Keep the remainder of pixels for the next frame's calculation
        grass_pixel_scroll_accumulator %= TILE_SIZE;

        // Wrap grass_current_tile_shift around TILE_COLS to make the pattern
repeat
        // Handle potential negative results from modulo with negative numbers
if speed can be negative
        grass_current_tile_shift %= TILE_COLS;
        if (grass_current_tile_shift < 0) {
            grass_current_tile_shift += TILE_COLS;
        }
    }
}


// MODIFICATION: Fills the sky and randomly distributes three types of grass
tiles,
// taking into account the current scroll offset.
void fill_sky_and_grass(void) {
    if (!vga_initialized) {
        init_vga_interface();
        if (!vga_initialized) return;
    }
    int r, c;
    unsigned char grass_tile_to_use;

    // Draw sky tiles to the back buffer.
    for (r = 0; r < GRASS_ROW_START; ++r) {
        for (c = 0; c < TILE_COLS; ++c) {
            write_tile_to_kernel(r, c, SKY_TILE_IDX);
        }
    }
    // Draw grass tiles randomly to the back buffer, considering the scroll.
    for (r = GRASS_ROW_START; r < TILE_ROWS; ++r) {
        for (c = 0; c < TILE_COLS; ++c) {
            // Calculate the effective column in the "world" grass pattern
            int pattern_col = (c + grass_current_tile_shift) % TILE_COLS;
            if (pattern_col < 0) pattern_col += TILE_COLS; // Ensure positive
for modulo-based pattern
```

```
            // Choose grass tile type based on the scrolled pattern column and
row
            // This creates a diagonal-like repeating pattern that scrolls.
            int rand_choice = (pattern_col + r) % 3; // Deterministic based on
scrolled position

            switch (rand_choice) {
                case 0:
                    grass_tile_to_use = GRASS_TILE_1_IDX;
                    break;
                case 1:
                    grass_tile_to_use = GRASS_TILE_2_IDX;
                    break;
                case 2:
                default:
                    grass_tile_to_use = GRASS_TILE_3_IDX;
                    break;
            }
            write_tile_to_kernel(r, c, grass_tile_to_use);
        }
    }
}


void fill_nightsky_and_grass(void) {
    if (!vga_initialized) {
        init_vga_interface();
        if (!vga_initialized) return;
    }

    int r, c;
    unsigned char sky_tile_to_use;
    unsigned char grass_tile_to_use;

    // Draw sky tiles to the back buffer.
    for (r = 0; r < GRASS_ROW_START; ++r) {
        for (c = 0; c < TILE_COLS; ++c) {
           int rand_choice = rand() % 6;
            sky_tile_to_use = (rand_choice == 0) ? NIGHTSKY_TILE_IDX :
STAR_TILE_IDX;
            write_tile_to_kernel(r, c, sky_tile_to_use);
        }
    }
```

```
    // Draw grass tiles randomly to the back buffer, considering the scroll.
    for (r = GRASS_ROW_START; r < TILE_ROWS; ++r) {
        for (c = 0; c < TILE_COLS; ++c) {
            // Calculate the effective column in the "world" grass pattern
            int pattern_col = (c + grass_current_tile_shift) % TILE_COLS;
            if (pattern_col < 0) pattern_col += TILE_COLS; // Ensure positive
for modulo-based pattern

            // Choose grass tile type based on the scrolled pattern column and
row
            // This creates a diagonal-like repeating pattern that scrolls.
            int rand_choice = (pattern_col + r) % 3; // Deterministic based on
scrolled position

            switch (rand_choice) {
                case 0:
                    grass_tile_to_use = GRASS_TILE_1_IDX;
                    break;
                case 1:
                    grass_tile_to_use = GRASS_TILE_2_IDX;
                    break;
                case 2:
                default:
                    grass_tile_to_use = GRASS_TILE_3_IDX;
                    break;
            }
            write_tile_to_kernel(r, c, grass_tile_to_use);
        }
    }
}

void fill_evesky_and_grass(void) {
    if (!vga_initialized) {
        init_vga_interface();
        if (!vga_initialized) return;
    }

    int r, c;
    unsigned char sky_tile_to_use;
    unsigned char grass_tile_to_use;

    // Draw sky tiles to the back buffer.
    for (r = 0; r < GRASS_ROW_START; ++r) {
        for (c = 0; c < TILE_COLS; ++c) {
```

```
          // int rand_choice = rand() % 6;
            sky_tile_to_use = EVE_TILE_IDX;
            write_tile_to_kernel(r, c, sky_tile_to_use);
        }
    }

    // Draw grass tiles randomly to the back buffer, considering the scroll.
    for (r = GRASS_ROW_START; r < TILE_ROWS; ++r) {
        for (c = 0; c < TILE_COLS; ++c) {
            // Calculate the effective column in the "world" grass pattern
            int pattern_col = (c + grass_current_tile_shift) % TILE_COLS;
            if (pattern_col < 0) pattern_col += TILE_COLS; // Ensure positive
for modulo-based pattern

            int rand_choice = (pattern_col + r) % 3; // Deterministic based on
scrolled position

            switch (rand_choice) {
                case 0:
                    grass_tile_to_use = GRASS_TILE_1_IDX;
                    break;
                case 1:
                    grass_tile_to_use = GRASS_TILE_2_IDX;
                    break;
                case 2:
                default:
                    grass_tile_to_use = GRASS_TILE_3_IDX;
                    break;
            }
            write_tile_to_kernel(r, c, grass_tile_to_use);
        }
    }
}
```

vga_interface.h

```
C/C++
//last workimng revert 2.3 of A


#ifndef VGA_INTERFACE_H
#define VGA_INTERFACE_H
```

```c
#include <stdbool.h> // For bool type

// Define screen dimensions in tiles
#define TILE_ROWS 30
#define TILE_COLS 40
#define TILE_SIZE 16

#define BLANKTILE 0 // img number of blank tile
#define NUMBEROFFSET 1
#define NUMBERTILE(x) (unsigned char) (x+NUMBEROFFSET)
#define LETTEROFFSET 11
#define LETTERTILE(x) (unsigned char) (x+LETTEROFFSET)

#define SCORE_COORD_R 1
#define SCORE_COORD_C 20
#define SCORE_MAX_LENGTH 4

#define MAX_HARDWARE_SPRITES 12

extern int vga_fd;

void init_vga_interface(void);

typedef struct {
    bool active;
    unsigned short r;
    unsigned short c;
    unsigned char n;
} SpriteHWState;


void write_tile_to_kernel(unsigned char r, unsigned char c, unsigned char n);
void write_sprite_to_kernel_buffered(unsigned char active, unsigned short r,
unsigned short c, unsigned char n, unsigned short register_n);
void vga_present_frame(void);
void present_sprites(void);


void write_number(unsigned int num, unsigned int row, unsigned int col);
void write_letter(unsigned char letter, unsigned int row, unsigned int col);
void write_numbers(unsigned int nums, unsigned int digits, unsigned int row,
unsigned int col);
void write_score(int new_score);
```

```
void write_text(unsigned char *text, unsigned int length, unsigned int row,
unsigned int col);

void cleartiles();
void clearSprites_buffered();


#define SKY_TILE_IDX 37
#define NIGHTSKY_TILE_IDX 0
#define STAR_TILE_IDX 44
#define GRASS_TILE_1_IDX 38 // User-defined 260 - grass plain
#define GRASS_TILE_2_IDX 42 // User-defined 2a0 - flowers on grass
#define GRASS_TILE_3_IDX 41 // User-defined 290 - grass with dark green thingy
#define GRASS_ROW_START 25
#define EVE_TILE_IDX 43

void fill_sky_and_grass(void);

void fill_nightsky_and_grass(void);

void fill_evesky_and_grass(void);
// NEW: Declaration for updating grass scroll offset
void update_grass_scroll(int scroll_speed_px);

#endif // VGA_INTERFACE_H
```

usbcontroller.c

```
C/C++
//Last working edit
#include "usbcontroller.h"

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


// find and return a usb controller via the argument, or NULL if not found
struct libusb_device_handle *opencontroller(uint8_t *endpoint_address) {
    //initialize libusb
       int initReturn = libusb_init(NULL);
```

```
      if(initReturn < 0) {
            printf("libusb initialization error!\n");
            exit(1);
      }

    // for searching for descriptor info
    struct libusb_device_descriptor desc;
    struct libusb_device_handle *controller = NULL;
    libusb_device **devs;
    ssize_t num_devs, d;
    uint8_t i, k;

    if ( (num_devs = libusb_get_device_list(NULL, &devs)) < 0 ) {
        fprintf(stderr, "Error: libusb_get_device_list failed\n");
        exit(1);
    }

    // iterate over all devices list to find the one with the right protocol
    for (d = 0; d < num_devs; d++) {
        libusb_device *dev = devs[d];
        if ( libusb_get_device_descriptor(dev, &desc) < 0 ) {
            fprintf(stderr, "Error: libusb_get_device_descriptor failed\n");
            libusb_free_device_list(devs, 1);
            exit(1);
        }
        if (desc.bDeviceClass == LIBUSB_CLASS_PER_INTERFACE) {
            struct libusb_config_descriptor *config;
            libusb_get_config_descriptor(dev, 0, &config);
            for (i = 0 ; i < config->bNumInterfaces ; i++) {
                for ( k = 0 ; k < config->interface[i].num_altsetting ; k++ ) {
                    const struct libusb_interface_descriptor *inter =
                    config->interface[i].altsetting + k;
                    if (inter->bInterfaceClass == LIBUSB_CLASS_HID &&
                        inter->bInterfaceProtocol == GAMEPAD_CONTROL_PROTOCOL)
{
                        int r;
                        if ((r = libusb_open(dev, &controller)) != 0) {
                            fprintf(stderr, "libusb_open failed: %s\n",
libusb_error_name(r));
                            exit(1);
                        }
                        if (libusb_kernel_driver_active(controller,i)) {
                            libusb_detach_kernel_driver(controller, i);
```

```
                        }
                        libusb_set_auto_detach_kernel_driver(controller, i);
                        if ((r = libusb_claim_interface(controller, i)) != 0) {
                            fprintf(stderr, "claim interface failed: %s\n",
libusb_error_name(r));
                            exit(1);
                        }
                        // endpoint address
                        *endpoint_address =
inter->endpoint[0].bEndpointAddress;
                        goto found;
                    }
                    // printf("d:%zd  i:%d  k:%d  interface class:%x  interface
protocol: %x  endpoint address: %x\n", d, i, k, inter->bInterfaceClass,
inter->bInterfaceProtocol, inter->endpoint[0].bEndpointAddress);
                }
            }
        }
    }

found:
    libusb_free_device_list(devs, 1);

    return controller;

}


struct controller_output_packet *usb_to_output(struct controller_output_packet
*packet,
                                               unsigned char* output_array) {
    /* check up and down arrow */
    switch(output_array[IND_UPDOWN]) {
        case 0x0: packet->updown = 1;
            break;
        case 0xff: packet->updown = -1;
            break;
        default: packet->updown = 0;
            break;
    }
    /* check left and right arrow */
    switch(output_array[IND_LEFTRIGHT]) {
        case 0x0: packet->leftright = 1;
            break;
```

```
        case 0xff: packet->leftright = -1;
            break;
        default: packet->leftright = 0;
            break;
    }

    /* check select and start with bitshifting */
    switch(output_array[IND_SELSTARIB] >> 4) {
        case 0x03: packet->select = packet->start = 1;
            break;
        case 0x02: packet->start = 1;
            packet->select = 0;
            break;
        case 0x01: packet->start = 0;
            packet->select = 1;
            break;
        case 0x00: packet->start = 0;
            packet->select = 0;
            break;
    }

    /* check left and right rib with bitmasking */
    switch(output_array[IND_SELSTARIB] & 0x0f) {
        case 0x03: packet->left_rib = packet->right_rib = 1;
            break;
        case 0x02: packet->right_rib = 1;
            packet->left_rib = 0;
            break;
        case 0x01: packet->right_rib = 0;
            packet->left_rib = 1;
            break;
        case 0x00: packet->right_rib = 0;
            packet->left_rib= 0;
            break;
    }

    packet->x = packet->y = packet->a = packet->b = 0;

    /* check if x, y, a, b is pressed */
    if ((output_array[IND_XYAB] >> 4) & 0x01) { // x
        packet->x = 1;
    }
    if ((output_array[IND_XYAB] >> 4) & 0x02) { // a
        packet->a = 1;
```

```
    }
    if ((output_array[IND_XYAB] >> 4) & 0x04) { // b
        packet->b = 1;
    }
    if ((output_array[IND_XYAB] >> 4) & 0x08) { // y
        packet->y = 1;
    }

    return packet;
}
```

usbcontroller.h

```
C/C++
#ifndef _USBKEYBOARD_H
#define _USBKEYBOARD_H

#include "libusb.h"

#define VENDOR_ID 0x079
#define PRODUCT_ID 0x011
#define GAMEPAD_ENDPOINT_ADDRESS 0x81
#define GAMEPAD_CONTROL_PROTOCOL 0
#define GAMEPAD_READ_LENGTH 8

#define IND_UPDOWN 4
#define IND_LEFTRIGHT 3
#define IND_SELSTARIB 7
#define IND_XYAB 5

#define UP 1
#define DOWN -1
#define LEFT 1
#define RIGHT -1



struct controller_output_packet {
    short updown; // 0 for no change, 1 for up, -1 for down
    short leftright; // 0 for no change, 1 for left, -1 for right
```

```
    uint8_t select; // for the rest, 1 is true/active, 0 is false/not active
    uint8_t start;
    uint8_t left_rib;
    uint8_t right_rib;
    uint8_t x;
    uint8_t y;
    uint8_t a;
    uint8_t b;
};


/* Find and open a USB controller device, argument should
point to space to store an endpoint address. Returns NULL
if no controller was found */
extern struct libusb_device_handle *opencontroller(uint8_t *);


/* convert the usb controller output into a packet for access */
extern struct controller_output_packet *usb_to_output(struct
controller_output_packet *, unsigned char*);

#endif
```

vga_top.c

```
C/C++
// adapted from 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>
```

```c
#include <linux/uaccess.h>
#include "vga_top.h"

#define DRIVER_NAME "vga_top"

/* Device registers */
#define WRITE_TILE(x) (x)
#define WRITE_SPRITE(x) (x+4) // it's byte addressed

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


static void write_tile(unsigned char r, unsigned char c, unsigned char n)
{
  // 5bit r, 6bit c, 8bit n
	iowrite32(((unsigned int) r << 14) + ((unsigned int) c << 8) + n,
WRITE_TILE(dev.virtbase) );
}

static void write_sprite(unsigned char active, unsigned short r, unsigned short
c, unsigned char n, unsigned short register_n)
{
	unsigned int r_mask = (1 << 9) - 1;
	unsigned int c_mask = (1 << 10) - 1;
	unsigned int n_mask = (1 << 5) - 1;
	 // printk("act:%i   r:%i  c:%i  n:%i  register_n:%i   address:%i\n",
active, r, c, n, register_n, WRITE_SPRITE(dev.virtbase + register_n) -
dev.virtbase);
	 // printk("Hex form: %x\n",   ((unsigned int) active << 24) +
		//           ((unsigned int) ((r & r_mask) << 15)) +
		//           ((unsigned int) ((c & c_mask) << 5)) +
		//           ((unsigned int) (n & n_mask)));
	// 1bit active, 9bit r, 10bit c, 5bit n
	iowrite32( ((unsigned int) active << 24) +
			((unsigned int) ((r & r_mask) << 15)) +
			((unsigned int) ((c & c_mask) << 5)) +
			((unsigned int) (n & n_mask)),
WRITE_SPRITE(dev.virtbase + register_n*4)); // byte addressed
```

```
}

/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
 */
static long vga_top_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
  vga_top_arg_t vlat;
  vga_top_arg_s vlas;
       switch (cmd) {
         case VGA_TOP_WRITE_TILE:
                if (copy_from_user(&vlat, (vga_top_arg_t *) arg,
sizeof(vga_top_arg_t)))
                       return -EACCES;
                write_tile(vlat.r, vlat.c, vlat.n);
                break;
           case VGA_TOP_WRITE_SPRITE:
                 //   printk("writing sprite:  ");
             if (copy_from_user(&vlas, (vga_top_arg_s *) arg,
sizeof(vga_top_arg_s)))
                       return -EACCES;
                write_sprite(vlas.active, vlas.r, vlas.c, vlas.n,
vlas.register_n);
                break;
         default:
                return -EINVAL;
       }

       return 0;
}

/* The operations our device knows how to do */
static const struct file_operations fops = {
       .owner       = THIS_MODULE,
       .unlocked_ioctl = vga_top_ioctl,
};

/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice misc_device = {
       .minor       = MISC_DYNAMIC_MINOR,
       .name        = DRIVER_NAME,
       .fops        = &fops,
```

```
};

/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init probe(struct platform_device *pdev)
{
	int ret;

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

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

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

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

	return 0;

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

/* Clean-up code: release resources */
```

```
static int vga_top_remove(struct platform_device *pdev)
{
	iounmap(dev.virtbase);
	release_mem_region(dev.res.start, resource_size(&dev.res));
	misc_deregister(&misc_device);
	return 0;
}

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

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver driver = {
	.driver	= {
		.name  = DRIVER_NAME,
		.owner = THIS_MODULE,
		.of_match_table = of_match_ptr(device_of_match),
	},
	.remove	= __exit_p(vga_top_remove),
};

/* Called when the module is loaded: set things up */
static int __init vga_top_init(void)
{
	pr_info(DRIVER_NAME ": init\n");
	return platform_driver_probe(&driver, probe);
}

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

module_init(vga_top_init);
module_exit(vga_top_exit);
```

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("zhz");
MODULE_DESCRIPTION("vga_top driver");
```

vga_top.h

```C/C++
#ifndef _VGA_TOP_H
#define _VGA_TOP_H

#include <linux/ioctl.h>

// def of argument for tiles
typedef struct {
  unsigned char r;
  unsigned char c;
  unsigned char n;
} vga_top_arg_t;

// def of argument for sprites
typedef struct {
  unsigned char active;
  unsigned short r;
  unsigned short c;
  unsigned char n;
  unsigned short register_n; // the corresponding sprite register, start from 0
} vga_top_arg_s;

// function top dec
void write_tile_to_kernel(unsigned char r, unsigned char c, unsigned char n);

void write_sprite_to_kernel(unsigned char active,
                            unsigned short r,
                            unsigned short c,
                            unsigned char n,
                            unsigned short register_n);


#define VGA_TOP_MAGIC 'q'
```

```
/* ioctls and their arguments */
#define VGA_TOP_WRITE_TILE _IOW(VGA_TOP_MAGIC, 1, vga_top_arg_t *)
#define VGA_TOP_WRITE_SPRITE _IOW(VGA_TOP_MAGIC, 2, vga_top_arg_s *)

#endif
```

tile_generator.py

Python

```
from google.colab import files
uploaded = files.upload()

from PIL import Image
import matplotlib.pyplot as plt

img = Image.open("I.jpg")
plt.imshow(img)
plt.axis("off")
plt.title("Uploaded Tile Preview")
plt.show()

#for specific tiles where I want to change background colour

from PIL import Image
import numpy as np

def rgb_to_565(r, g, b):
    r = int(r)
    g = int(g)
    b = int(b)
    red = (r * 31) // 255
    green = (g * 31) // 255
    blue = (b * 63) // 255
    return (red << 11) | (green << 6) | blue

# Common nearly-white/light-grey RGB565 values to treat as background
light_bg_codes = {
    0xEF7B,  # RGB(255, 255, 255) - pure white
    0xE73A,  # light grey
    0xE739,  # slightly darker light grey
    0xE73B   # another light grey shade
}
```

```
A
# Load and prepare image
img = Image.open("E.png").convert("RGB").resize((16, 16), Image.NEAREST)
pixels = np.array(img, dtype=np.uint8)

# Convert each row of pixels to 16-bit MIF line with horizontal flip
start_address = 0x1A0  # Adjust as needed
for y in range(16):
    row = pixels[y][::-1]  # reverse the row for horizontal mirroring
    words = []
    for r, g, b in row:
        color565 = rgb_to_565(r, g, b)
        if color565 in light_bg_codes:
            words.append("867A")
        else:
            words.append(f"{color565:04X}")
    print(f"\t{start_address + y:03x}  :   {''.join(words)};")


#general tile generator

from PIL import Image
import numpy as np

def rgb_to_565(r, g, b):
    r = int(r)
    g = int(g)
    b = int(b)
    red = (r * 31) // 255
    green = (g * 31) // 255
    blue = (b * 63) // 255
    return (red << 11) | (green << 6) | blue

# Load and prepare image
img = Image.open("I.jpg").convert("RGB").resize((16, 16), Image.NEAREST)
pixels = np.array(img, dtype=np.uint8)

# Convert each row of pixels to 16-bit MIF line with horizontal flip
start_address = 0x020  # adjust as needed
for y in range(16):
    row = pixels[y][::-1]  # reverse the row for mirroring
    words = [f"{rgb_to_565(r, g, b):04X}" for r, g, b in row]
    print(f"\t{start_address + y:03x}  :   {''.join(words)};")
```

SystemVerilog Files (reused or modified from Bubble Bobble):

vga_top.sv

```
Unset
/*
 * Avalon memory-mapped peripheral that generates VGA
 *
 * Stephen A. Edwards
 * Columbia University
 *
 * Majorly based on BubbleBobble code
 */

module vga_top(input logic        clk,
               input logic          reset,
              input logic [31:0]  writedata,
              input logic     write,
              input           chipselect,
              input logic [4:0]  address, //KV2446

              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);

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

   vga_counters counters(.clk50(clk), .*);

    // line buffer
    logic    [5:0]  address_tile_display;
         logic      [9:0]  address_pixel_display;
         logic      [5:0]  address_tile_draw;
         logic      [9:0]  address_pixel_draw;

         logic      [255:0]  data_tile_display;
    logic    [15:0]  data_pixel_display;
    logic    [255:0]  data_tile_draw;
    logic    [15:0]  data_pixel_draw;


         logic        wren_tile_display;
         logic        wren_pixel_display;
```

```
    logic      wren_tile_draw;
        logic        wren_pixel_draw;

        logic      [255:0]  q_tile_display;
    logic    [15:0]  q_pixel_display;
    logic    [255:0]  q_tile_draw;
    logic    [15:0]  q_pixel_draw;

    logic switch;

    linebuffer(.*);

    // tile loader
    logic tile_start;
    logic tile_finish;
    logic tile_write;
    assign tile_write = (chipselect && write && (address == 0)); // address =
0: write tile
    // low 19 bit of writedata: row(5b), column(6b), tile image number(8b)
    tile_loader(clk, reset, tile_start, tile_write, writedata[18:0], vcount,
address_tile_draw, data_tile_draw, tile_finish);

    // sprite loader
    logic sprite_start;
    logic sprite_finish;
    logic sprite_write;
    //assign sprite_write = (chipselect && write && (address >= 1)); // address
>= 1: write sprite
       localparam int NUM_SPRITES = 16;
       assign sprite_write = chipselect  && write && (address >= 1)  && (address
< 1 + NUM_SPRITES);



    sprite_loader(clk, reset, sprite_start, sprite_write, address,
writedata[24:0], vcount, address_pixel_draw, data_pixel_draw, sprite_finish,
wren_pixel_draw);

    logic drawing_sprite;

   // draw line buffer
   always_ff @(posedge clk) begin
      if (reset) begin
        wren_tile_display <= 0;
```

```
        wren_pixel_display <= 0;
        wren_tile_draw <= 0;
        tile_start <= 0;
        sprite_start <= 0;
        drawing_sprite <= 0;

        tile_start <= 0;
      // only draw active lines
      end else if (vcount < 479 || vcount == 524) begin
        // start the tile loader to write 40 tiles
        if(hcount == 0) begin
          tile_start <= 1;
          wren_tile_draw <= 1;
          drawing_sprite <= 0;
        end else begin
          tile_start <= 0;
        end

        // wait for tile loader to finish
        // careful, setting start=1 takes 1 cycle and resetting tile_finish
takes another
        // so tile start becomes 1 at hcount=1 and tile_finish becomes 0 at
hcount=2
        if(hcount > 1 && tile_finish && (drawing_sprite == 0)) begin
          wren_tile_draw <= 0;
          // tile draw done, start sprite
          sprite_start <= 1;
          drawing_sprite <= 1;
        end
        // pull sprite_start back to 0 since it only needs 1 cycle pulse
        if (drawing_sprite) begin
          sprite_start <= 0;
        end

      end
    end


   // output
   always_comb begin
      if (hcount[10:1] < 639)
        address_pixel_display = hcount[10:1] + 1; // account for memory delay
      else
        address_pixel_display = 0;
```

```
      if(hcount == 1598 && (vcount < 479 || vcount == 524)) // 2 cycles early:
1 cycle for switch, another for reading memory
        switch = 1;
      else
        switch = 0;

      VGA_R = q_pixel_display[15:11] << 3;
      VGA_G = q_pixel_display[10:6] << 3;
      VGA_B = q_pixel_display[5:1] << 3;
   end

endmodule

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

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

   // Parameters for vcount
   parameter VACTIVE      = 10'd 480,
             VFRONT_PORCH = 10'd 10,
             VSYNC        = 10'd 2,
```

```
          VBACK_PORCH  = 10'd 33,
          VTOTAL       = VACTIVE + VFRONT_PORCH + VSYNC +
                         VBACK_PORCH; // 525

logic endOfLine;

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

assign endOfLine = hcount == HTOTAL - 1;

logic endOfField;

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

assign endOfField = vcount == VTOTAL - 1;

// Horizontal sync: from 0x520 to 0x5DF (0x57F)
// 101 0010 0000 to 101 1101 1111
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
```

linebuffer.sv

```
Unset
module linebuffer(
       clk,
  reset,
       switch,
       address_tile_display,
       address_pixel_display,
       address_tile_draw,
       address_pixel_draw,

       data_tile_display,
       data_pixel_display,
  data_tile_draw,
  data_pixel_draw,

  wren_tile_display,
  wren_pixel_display,
  wren_tile_draw,
  wren_pixel_draw,

  q_tile_display,
  q_pixel_display,
  q_tile_draw,
  q_pixel_draw
);
       input  clk;
  input reset;
  input switch;     // switch drawing and displaying buffer
  input      [5:0]  address_tile_display;
       input  [9:0]  address_pixel_display;
       input  [5:0]  address_tile_draw;
       input  [9:0]  address_pixel_draw;

       input  [255:0]  data_tile_display;
```

```
  input      [15:0]  data_pixel_display;
  input      [255:0]  data_tile_draw;
  input      [15:0]  data_pixel_draw;


      input    wren_tile_display;
      input    wren_pixel_display;
  input        wren_tile_draw;
      input    wren_pixel_draw;

      output [255:0]  q_tile_display;
  output     [15:0]  q_pixel_display;
  output     [255:0]  q_tile_draw;
  output     [15:0]  q_pixel_draw;

  wire [5:0] address_tile[1:0];
  wire [9:0] address_pixel[1:0];

  wire [255:0] data_tile[1:0];
  wire [16:0] data_pixel[1:0];

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

  wire [255:0] q_tile[1:0];
  wire [16:0] q_pixel[1:0];

  logic display_index;
  logic draw_index;

  linebuffer_ram ram0(address_tile[0], address_pixel[0], clk, data_tile[0],
data_pixel[0], wren_tile[0], wren_pixel[0], q_tile[0], q_pixel[0]);
  linebuffer_ram ram1(address_tile[1], address_pixel[1], clk, data_tile[1],
data_pixel[1], wren_tile[1], wren_pixel[1], q_tile[1], q_pixel[1]);

  always_ff @(posedge clk) begin
    if(reset) begin
      draw_index <= 0;
      display_index <= 1;
    end else if (switch) begin // currently only accept single-cycle pulse on
switch
      draw_index <= display_index;
      display_index <= draw_index;
    end
```

```
  end

  always_comb begin
      address_tile[display_index] = address_tile_display;
      address_pixel[display_index] = address_pixel_display;
      address_tile[draw_index] = address_tile_draw;
      address_pixel[draw_index] = address_pixel_draw;

      data_tile[display_index] = data_tile_display;
      data_pixel[display_index] = data_pixel_display;
      data_tile[draw_index] = data_tile_draw;
      data_pixel[draw_index] = data_pixel_draw;

      wren_tile[display_index] = wren_tile_display;
      wren_pixel[display_index] = wren_pixel_display;
      wren_tile[draw_index] = wren_tile_draw;
      wren_pixel[draw_index] = wren_pixel_draw;

      q_tile_display = q_tile[display_index];
      q_pixel_display = q_pixel[display_index];
      q_tile_draw = q_tile[draw_index];
      q_pixel_draw = q_pixel[draw_index];
  end

endmodule
```

tile_loader.sv

```
Unset
module tile_loader(input logic clk,
                   input logic reset,
                   input logic start,
                   input logic write,
                   input logic [18:0] writedata,
                   input logic [9:0] vcount,
                   output logic [5:0] address_tile_draw, // = column of a tile
                   output logic [255:0] data_tile_draw,
                   output logic finish
);

  logic [5:0] col;
```

```
  logic [10:0] tile_array_address_read;
  logic [10:0] tile_array_address_write;
  assign tile_array_address_write = writedata[18:14]*40 + writedata[13:8];

  logic [9:0] actual_vcount; // the next line to draw
  logic [7:0] tile_img_num;
  logic [11:0] tile_rom_address; // adjusted according to the actual rom size

  // row = actual_vcount / 16 = actual_vcount >> 4
  // address of the beginning of that row = row * 40
  assign tile_array_address_read = (actual_vcount >> 4)*40 + col;

  tile_array(.address_a(tile_array_address_read), // port a for read
           .address_b(tile_array_address_write), // port b for write
           .clock(clk),
           .data_b(writedata[7:0]),
           .wren_a(1'b0),
           .wren_b(write),
           .q_a(tile_img_num));

  // 16 rows (words) per image, vcount%16 is the row (word) in the tile
  // tile_img_num * 16 + actual_vcount % 16
  assign tile_rom_address = (tile_img_num << 4) + actual_vcount[3:0]; // + has
higher priority than <<

  tile_rom(tile_rom_address, clk, data_tile_draw);

  always_ff @(posedge clk) begin
    if(reset) begin
      finish <= 1;
    end else if (start) begin
      finish <= 0;
      col <= 0;
      // calculate actual vcount
      if (vcount < 479) begin
        actual_vcount <= vcount + 1;
      end else if (vcount >= 479 && vcount < 524) begin
        finish <= 1; // inactive lines, nothing to draw
      end else if (vcount == 524) begin
        actual_vcount <= 0; // draw the first line
      end
    end else if (!finish) begin
      if (col < 39)
        col <= col+1;
```

```
      if (address_tile_draw == 38)
        finish <= 1;
    end
  end

  // output linebuffer address to draw
  always_ff @(posedge clk) begin
    if (col <= 1) begin // wait 2 cycles to sync with data_tile_draw
      address_tile_draw <= 0;
    end else if (!finish) begin
      address_tile_draw <= address_tile_draw + 1;
    end
  end

endmodule
```

sprite_loader.sv

```
Unset
module sprite_loader(input logic clk,
                     input logic reset,
                     input logic start,
                     input logic write,
                     input logic [4:0] sprite_register_number, //kv2446
                     input logic [24:0] writedata,
                     input logic [9:0] vcount,
                     output logic [9:0] address_pixel_draw,
                     output logic [15:0] data_pixel_draw,
                     output logic finish,
                     output logic wren_pixel_draw
);
    logic sprite_active_start;
    logic [4:0] sprite_number; //kv2446
    logic [9:0] actual_vcount;
    logic [4:0] row_in_sprite;
    logic [9:0] sprite_column;
    logic [4:0] img_num;
    logic is_active;
    logic sprite_active_finish;
    logic checking;
    logic checked;
```

```
    sprite_active(clk, reset, write, sprite_register_number, sprite_number,
writedata, sprite_active_start, actual_vcount, row_in_sprite, sprite_column,
img_num, is_active, sprite_active_finish);


    logic sprite_draw_start;
    logic sprite_draw_finish;
    logic drawing;

    sprite_draw(clk, reset, sprite_draw_start, row_in_sprite, sprite_column,
img_num, wren_pixel_draw, address_pixel_draw, data_pixel_draw,
sprite_draw_finish);


    always_ff @(posedge clk) begin
        if (reset) begin
            finish <= 1;
            drawing <= 0;
            checked <= 0;
            checking <= 0;
            sprite_active_start <= 0;
            sprite_draw_start <= 0;
        end else if (start) begin
            sprite_number <= 0;
            finish <= 0;
            checked <= 0;
            checking <= 0;
            drawing <= 0;
            // calculate actual vcount
            if (vcount < 479) begin
                actual_vcount <= vcount + 1;
            end else if (vcount >= 479 && vcount < 524) begin
                finish <= 1; // inactive lines, nothing to draw
            end else if (vcount == 524) begin
                actual_vcount <= 0; // draw the first line
            end
        end else if (!finish) begin
          if (sprite_number < 16) begin //12) begin
                // check sprite active
                if (!checked) begin
                    if (!checking) begin
                        sprite_active_start <= 1;
                        checking <= 1;
                    end else begin
```

```
                        sprite_active_start <= 0;
                        if (sprite_active_finish) begin
                            checking <= 0;
                            checked <= 1;
                        end
                    end
                end else begin
                    // start drawing if active
                    if (is_active) begin
                        if (!drawing) begin
                            drawing <= 1;
                            sprite_draw_start <= 1;
                        end else begin
                            sprite_draw_start <= 0;
                        end

                        // sprite_draw_start = 0 happens in the same cycle as
sprite_draw_finish = 0
                        if (drawing && (sprite_draw_start == 0) &&
sprite_draw_finish) begin
                            drawing <= 0;
                            sprite_number <= sprite_number + 1;
                            checked <= 0;
                        end
                    // skip if not active
                    end else begin
                        sprite_number <= sprite_number + 1;
                        checked <= 0;
                    end
                end
            end else begin
                finish <= 1;
            end
        end
    end

endmodule
```

sprite_active.sv

```
Unset
```

```
module sprite_active(input logic clk,
                     input logic reset,
                     input logic write_sprite,
                     input logic [4:0] sprite_number_write,
                     input logic [4:0] sprite_number, //kv2446
                     input logic [24:0] sprite_register,
                     input logic write_vcount,
                     input logic [9:0] actual_vcount, // the line being drawn
                     output logic [4:0] row_in_sprite, // the row needed to be
drawn in the sprite
                     output logic [9:0] sprite_column, // where the sprite is
located
                     output logic [4:0] img_num,
                     output logic is_active, // input sprite is indeed, active
                     output logic finish // finish the sprite
);


  // sprite array access
  logic [31:0][24:0] sprite_array;
  // indexing: 24 is active, 23-15 is v/row, 14-5 is h/col 4-0 is image number


  always_ff @(posedge clk) begin
    if(reset) begin
      finish <= 1;
      is_active <= 0;
    end else if (write_vcount) begin
      finish <= 0;
      if (actual_vcount < 480) begin
        // check corresponding sprite register
        if (sprite_array[sprite_number][24] == 1) begin
          // if this sprite is active at somewhere on the screen
          if (actual_vcount >= sprite_array[sprite_number][23:15] &&
              actual_vcount - sprite_array[sprite_number][23:15] < 32) begin
            // is within range, output sprite
            row_in_sprite <= (actual_vcount -
sprite_array[sprite_number][23:15]);
            sprite_column <= sprite_array[sprite_number][14:5];
            img_num <= sprite_array[sprite_number][4:0];
            is_active <= 1;
            finish <= 1;
          end else begin
```

```
            // is not within range
            is_active <= 0;
            finish <= 1;
          end
        end else begin
          // sprite not active
          is_active <= 0;
          finish <= 1;
        end
      end else if (actual_vcount >= 480) begin
        is_active <= 0;
        finish <= 1; // inactive lines, nothing to do
      end
    end
  end

  // for when needing to change sprite_register value
  // -1 because incoming number because sprite register base is base + 1
  always_ff @(posedge clk) begin
    if (write_sprite) begin
      sprite_array[sprite_number_write - 1][24:0] <= sprite_register[24:0];
    end
  end


endmodule
```

sprite_draw.sv

```
Unset


module sprite_draw(input logic clk,
                   input logic reset,
                   input logic start,
                   input logic [4:0] row_in_sprite, // the row needed to be
drawn in the sprite
                   input logic [9:0] sprite_column, // where the sprite is
located
                   input logic [4:0] img_num,
                   output logic wren,
```

```
                    output logic [9:0] pixel_hcount, // where the pixel goes on
the row
                    output logic [15:0] data, // pixel data
                    output logic finish
);

  logic [15:0] sprite_rom_address;
  sprite_rom(sprite_rom_address, clk, data);

  logic [5:0] col;
  assign wren = (!finish) && (col > 0) && (data[0] == 0); // write only not
transparent

  always_ff @(posedge clk) begin
    if (reset) begin
      finish <= 1;
    end else if (start) begin
      // img_num * 1024 (# of pixels per img) + row_in_sprite * 32 (# of pixels
per row)
      sprite_rom_address <= (img_num << 10) + (row_in_sprite << 5);
      col <= 0;
      finish <= 0;
      pixel_hcount <= 0;
    end else if (!finish) begin
      // get pixel data from rom
      if (col < 32 && pixel_hcount < 639) begin
        col <= col + 1;
        sprite_rom_address <= sprite_rom_address + 1;
      end else
        finish <= 1;

      // output linebuffer address to draw
      if (col == 0) begin // wait 1 cycle to sync with data_tile_draw
        pixel_hcount <= sprite_column;
      end else if (pixel_hcount < 639) begin
        pixel_hcount <= pixel_hcount + 1;
      end
    end
  end

endmodule
```

Assets:

- combined_tile.mif (sky, grass, platform, letters)
- combined_sprite.mif (chicken, coin, sun/moon)