Jump Jump Report

29th March 2025

Abstract

This project is our final submission for *ECE243: Computer Organization* at the University of Toronto. It is a **PvP (Player vs. Player) game** that integrates VGA rendering, audio processing, and PS/2 keyboard input, all implemented in **C**. In this game, **Player 1 controls Pac-Man** using a PS/2 keyboard, while **Player 2 controls jumping ghosts** through microphone input. By combining real-time graphics and audio-based interactions, the game offers an engaging and competitive experience for players to enjoy with friends.

1. Introduction

1.1 Overview

This project is the final assignment for ECE243: Computer Organization at the University of Toronto. It is a **two-player PVP game** that integrates **VGA rendering, audio processing,** and **PS/2 keyboard input**, all implemented using **C programming** on the **DE1-SoC FPGA board**. The game combines traditional controls with an innovative audio-based input, allowing players to engage in a fun and interactive experience.

1.2 Objectives

Our project aims to apply low-level programming techniques and hardware interaction concepts learned in ECE243 to create a fully functional game. This project demonstrates key aspects of embedded systems and real-time computing by utilizing VGA graphics, PS/2 input handling, and real-time audio processing.

Additionally, the game was designed to provide a competitive and enjoyable multiplayer experience, where:

- Player 1 controls Pac-Man using a PS/2 keyboard.
- Player 2 controls jumping ghosts using voice input from a microphone.

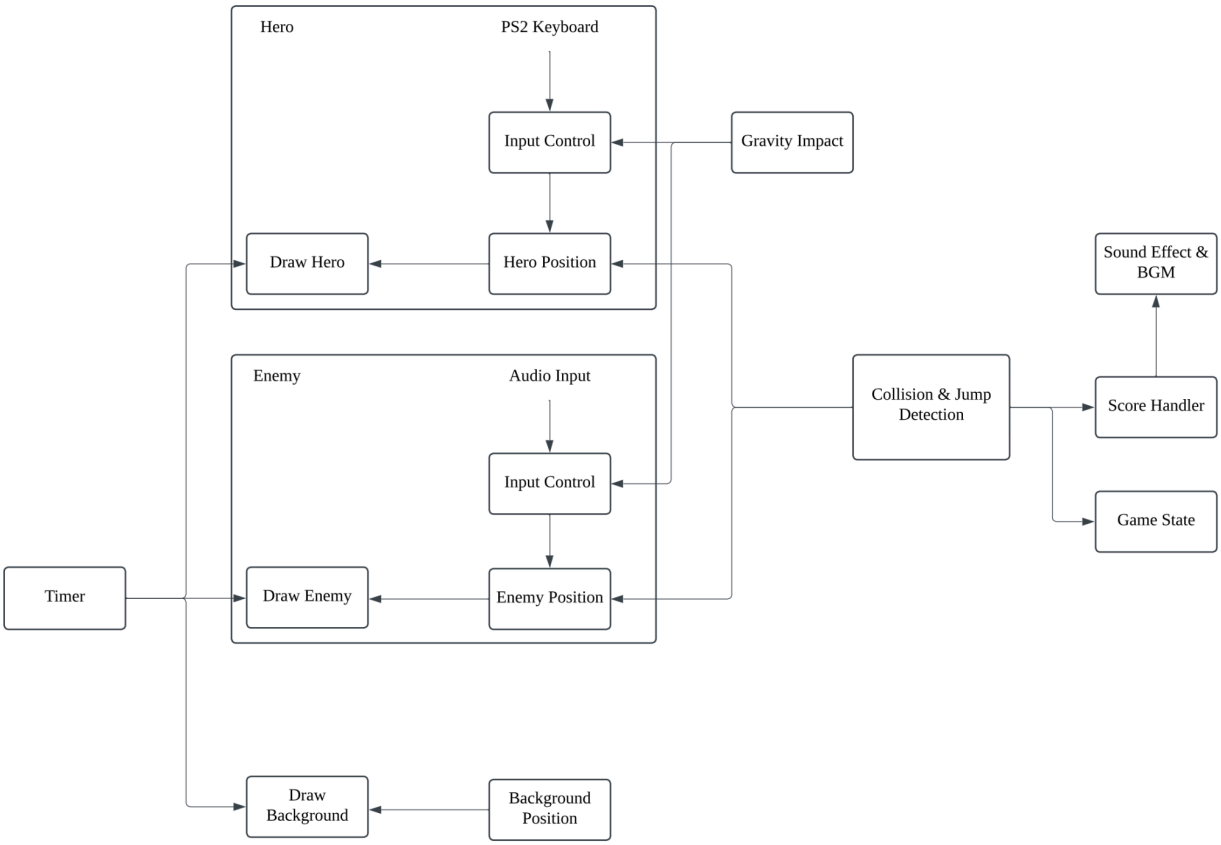

Figure 1: Block Diagram for our project

1.3 Course Relevance

This project integrates several core **computer organization** topics, including:

- Memory-mapped I/O: VGA rendering, double buffering for smooth rendering.
- Polling-driven programming: Processing multiple inputs at the same time.
- Various inputs processing: PS/2 keyboard, audio.

2. Code Description

2.1 Game Logic

Below are the core game logics for detecting game state and calculating scores:

- Character Updating: Updating characters' location due to user input and score
- Gravity setting: Constant gravity to keep the characters falling
- Parallel World Design: Two worlds are parallel to each other
- Collision Detection: When the hero (Player 1) touches the ghosts (Player 2) or other enemies, the game is over.

2.2 Game-state switching

Our game has 5 game states:

- Starting page
- Tutorial for new players
- Game start
- Game over animation
- Game restart state

2.3 Audio Input

The game utilizes **audio input** to enable Player 2 to control the ghosts to jump by detecting sound amplitude through a microphone. This feature is implemented using the **DE1-SoC audio input FIFO**, where the microphone captures real-time audio signals.

To process real-time sound amplitude triggering, we applied the **Root-Mean-Square** method to take a fixed amount of input signals and calculate the max amplitude.

```c
#define AUDIO_BASE 0xFF203040
#define MAX_SAMPLES 512

// get pointer to audio fifo
struct audio_t {
  unsigned int control;
  unsigned char rarc;
  unsigned char ralc;
  unsigned char wsrc;
  unsigned char wslc;
  unsigned int ldata;
  unsigned int rdata;
};

volatile struct audio_t *const audiop = ((struct audio_t *)AUDIO_BASE);

int main() {
  while (1) {
    float amplitude = 0;
    float max_amplitude = 0;

    int count = 0;
    while (audiop->rarc > 0 && count < MAX_SAMPLES) {
      int sample = audiop->ldata;
      if (sample < 0) {
        sample = -sample;
      }
      if (sample > max_amplitude) {
        max_amplitude = sample;
      }
      count++;
    }

    // Convert to normalized amplitude
    amplitude = fabs((float)max_amplitude / (float)(0x7FFFFF));
    if (amplitude > 2) {
      printf("Amplitude: %.2f\n", amplitude);
    }

    // Update Enemy Positions
    update_enemy_positions(amplitude);
  }
}
```

Figure 2: Code for taking audio input and triggering the jumping for ghosts

2.4 PS/2 Input

The game utilizes **PS/2 input** to enable Player 1 to control the Pac-Man.
To implement smooth keyboard usage, we used some global variables to detect the key-released and key-holding states.

```
volatile int *ps2_ptr = (volatile int *)0xFF200100;
char key_held = 0;

int main() {
  int PS2_data, RAVLID;
  char byte1 = 0, byte2 = 0, byte3 = 0;
  *ps2_ptr = 0xff;
  char key = 0;
  int move = 0;
  while (1) {
    PS2_data = *ps2_ptr;
    RAVLID = PS2_data & 0X8000;
  }

  if (RAVLID) {
    byte1 = byte2;
    byte2 = byte3;
    byte3 = PS2_data & 0xff;
    if (byte2 == 0xf0) {
      if (byte3 == key_held) {
        key_held = 0;
        key = byte3;
        move = 0;
      }
    } else {
      key_held = byte3;
      key = key_held;
      move = 1;
    }
  }
}

void update_hero_position(char key, int move) {
  if (move) {
    // Down Button
    if (key == 0x1B) {
      heroY = groundHeight + heroRadius + 1;
    }
  }
}
```

Figure 3: Code for taking PS/2 input and triggering moving for Pac-Man

2.5 VGA Output

2.5.1 Double Buffering VGA Rendering Logic

To avoid frequent blinkings of the screen, we implemented double buffering, which draws all the background, characters and score on the back buffer and then swaps the two buffers to render on the front buffer.

```
// Read the starting address of the pixel buffer from the pixel controller
volatile int *pixel_ctrl_ptr = (int *)0xFF203020;

// Properly align buffers in SDRAM
short int Buffer1[SCREEN_HEIGHT][512];  // Front buffer
short int Buffer2[SCREEN_HEIGHT][512];  // Back buffer

int pixel_buffer_start;

int main() {
  // Set front buffer to Buffer1
  *(pixel_ctrl_ptr + 1) = (int)Buffer1;
  wait_for_vsync();
  pixel_buffer_start = *pixel_ctrl_ptr;  // Set front buffer pointer
  clear_screen(Buffer1);

  // Set back buffer to Buffer2
  *(pixel_ctrl_ptr + 1) = (int)Buffer2;
  pixel_buffer_start = *(pixel_ctrl_ptr + 1);
  clear_screen(Buffer2);

  while (1) {
    // Update Enemy Positions
    update_enemy_positions(amplitude);
    // check_mutation();
    update_hero_position(key, move);
    update_background_position(key, move);
    draw_background();

    // Erase prev. drawings.
    for (int i = 0; i < 6; i++) {
      clear_character(x, y);
    }

    // Draw enemies.
    draw_character(x, y);

    wait_for_vsync();
    pixel_buffer_start = *(pixel_ctrl_ptr + 1);  // Swap buffer
  }
}
```

Figure 4: Code for implementing double buffering

The clean logic only cleans the previous position of the character, which achieves a clear character display even with moving characters and backgrounds.

Figure 5: Game screen display of clear characters.

2.5.2 Character Rendering

Converting the character.png to a 2d short int array which contains the position and colour enables the program to draw the character on the VGA screen.

```
void draw_slime(int x, int y, int length, int height, short int color) {
    for (int i = 0; i < CHARACTER_SIZE; i++) {
      for (int j = 0; j < CHARACTER_SIZE; j++) {
        if (slime[i][j] != 0x0000) {
          plot_pixel(x + j, y + i, slime[i][j]);
        }
      }
    }
  }
```

Figure 6: Example code for drawing slime

2.6 Animation Rendering

Purpose:

To create visually dynamic effects such as:

- Hero death
- Blood splatter
- Experience (XP) absorption
- Revival animation

Key Features:

- It uses double buffering (`Buffer1`, `Buffer2`) to ensure flicker-free animation.
- Each animation is handled in dedicated functions:
    - `animate_ball_crack(...)`
    - `animate_blood_splash(...)`
    - `animate_blood_drops(...)`
    - `animate_revive(...)`
    - `increase_xp(...)`
    - `animate_reverse_blood_spring(...)`

Key Code Snippet:

```
for (frame = 0; frame < numFrames; frame++) {
    clear_screen();  // Clear back buffer
    for (i = 0; i < NUM_PARTICLES; i++) {
        particles[i].x += particles[i].vx;
        particles[i].y += particles[i].vy;
```

```
            plot_pixel((int)particles[i].x, (int)particles[i].y, color);
        }
        wait_for_vsync();  // Swap to display the new frame
        pixel_buffer_start = *(pixel_ctrl_ptr + 1);  // Update buffer pointer
        delay(1);
    }
```

Techniques:

- Particles use physics-like properties (`vx`, `vy`) to simulate movement.
- `plot_pixel(...)` places each frame's updates.
- Synchronization via `wait_for_vsync()` to avoid screen tearing.

2.7 Score Storing & Display

Purpose:

To track the player's score and manage a **leaderboard** system.

Key Features:

- Real-time **score updating** (`score` variable)
- Display:
    - `Score: [current]` at top-right
    - `History Max: [max]` at top-left
- Upon death:
    - `GAME OVER` screen
    - Name input and leaderboard update

- Top 5 entries shown

Key Code Snippet:

```
char scoreStr[16];

sprintf(scoreStr, "Score: %d", score);

draw_string(320 - strlen(scoreStr) * CHAR_WIDTH, 0, scoreStr, GREEN);


void update_leaderboard() {

    if (score >= 0) {

        ...

        // Insert or update player

        // Sort descending

    }

}
```

Leaderboard Structure:

```
typedef struct {

    char name[20];

    int score;

} LeaderboardEntry;

LeaderboardEntry leaderboard[MAX_LEADERBOARD_ENTRIES];
```

Name Input:

- Controlled via PS/2 input (`check_name_input`)

- Uses custom scancode-to-ASCII mapping (`ps2_to_ascii`)
- Pressing **Enter** starts the game

2.8 Collision Detection

Purpose:

To detect when the **hero character** touches an **enemy**, triggering death and transitioning to the game-over animation and leaderboard screen.

How It Works:

- Every frame during the game loop, the function `check_collision(...)` is called.
- It compares the **hero's position and radius** with each enemy's **rectangle bounds**.
- If overlap is detected, it:
    - Triggers `animate_ball_crack(...)`
    - Sets `gameOn = 2` (to go to death screen)
    - Flags leaderboard update (`leaderboardUpdated = 0`)

Key Function:

```
void check_collision(int enemyX[], int enemyY[], int heroX, int heroY)
```

Key Code Snippet:

```
for (int i = 0; i < 6; i++) {
    int e_x1 = enemyX[i] + 3;
    int e_x2 = enemyX[i] + enemyLenght - 3;
    int e_x3 = enemyX[i] + 0.5 * enemyLenght;

    int x1 = heroX - e_x1;
    int x2 = heroX - e_x2;
    int x3 = heroX - e_x3;
```

```
    int y = heroY - enemyY[i];
    int y_half = heroY - enemyY[i] - enemyHeight * 0.5;
    int y_full = heroY - enemyY[i] - enemyHeight;

    double l1 = x1 * x1 + y * y;
    double l2 = x2 * x2 + y * y;
    double l3 = x3 * x3 + y_half * y_half;
    double l4 = x1 * x1 + y_full * y_full;
    double l5 = x2 * x2 + y_full * y_full;

    if (l1 < heroRadius * heroRadius || l2 < heroRadius * heroRadius
||
        l3 < heroRadius * heroRadius || l4 < heroRadius * heroRadius
||
        l5 < heroRadius * heroRadius) {

        animate_ball_crack(heroX, heroY, heroRadius);
        leaderboardUpdated = 0;
        gameOn = 2;
    }
}
```

Logic Breakdown:

- It approximates collision between a **circular hero** and **rectangular enemies** by checking:
    1. Distance from the hero center to multiple **enemy edge points**.
- It uses the **Pythagorean distance formula** (`dx² + dy²`) and compares with `heroRadius²`.
- Checks 5 key points on each enemy:
    1. Left edge
    2. Right edge

3. Center-top
4. Top-left
5. Top-right

Called In:

```
case 1:
    ...
    check_collision(enemyX, enemyY, heroX, heroY);
```

Result:

On detection:

- Collision animation (`animate_ball_crack`)
- Stops gameplay
- Prepares to show leaderboard and revival prompt

3. Testing

3.1 CPU-Lator simulator

For rendering VGA and simple game logic, we used CPU-Lator to simulate each part and avoid going to the lab for testing.

3.2 FPGA Board

After checking each part works fine, we used the FPGA Board to modify the logic and settings for a smoother user interface.

4. Reflection

4.1 Contents balancing for limited memory

After creating a fixed background, we implemented a parallax background with a closer background moving faster and a farther background moving slower to create a 3D feeling.

Figure 7: Closer Background with (255, 0, 255) as transparency

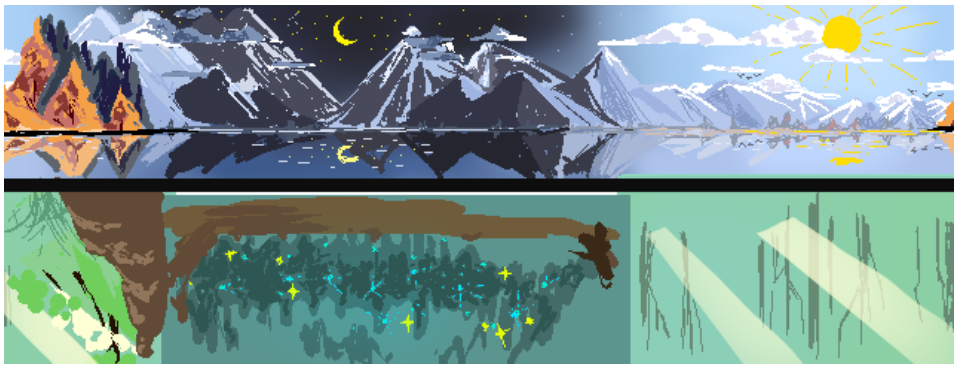

Figure 8: Farther Background

However, with two huge 2d arrays (640*240), the hardware memory was taken up too much and the character-moving lags a lot. We have to balance the memory and visual display effects, so we just keep only one background.

4.2 Task Planning

With two people working on this final project, most of the time we need to work individually and test our specific parts on CPU-Lator.

In addition, at first, we thought the PS/2 connection was easy which just received keyboard input signals and reacted accordingly. However, we didn't know that the PS/2 would send the key pressed and released right after the break signal (0xf0) is sent, causing continuous moving after one press. We tried various methods to detect the break signal and avoid the feedback signal messing up triggering logic.

4.3 Interest Combination

We applied our personal interests to this project. The design of characters and backgrounds creates a unique game visual.
Some game logic like the following move also used our previous experience with our other projects.

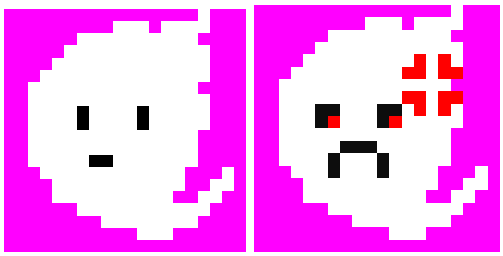

Figure 9: Designed changing ghosts for normal and jumping states

4.4 Improvements

- PS/2 inputs limit the player inputs of pressing two buttons at the same time, reducing a little bit user interface. May improve by detecting holding a key while pressing another key.
- Due to limited hardware memory, more features like parallax background moving, BGM and changing characters align moving logic are reduced for a smooth experience. May improve by using an outer RAM connection.
- Rolling Background: To achieve a better display and visual depth, for the 320*240 VGA screen, we prepared a 640*240 background array to achieve a rolling background moving when with user inputs.

```
short int background[240][640];  // Farther background (moves slower)
int background_offset = 0;       // Offset for farther background

/* Function to draw the background with parallax scrolling */
void draw_farther_background() {
    for (int y = 0; y < SCREEN_HEIGHT; y++) {
      for (int x = 0; x < SCREEN_WIDTH; x++) {
        // Compute background positions with wrap-around
        int farther_x = (x + background_offset) % BG_WIDTH;

        // Draw the farther background first
        plot_pixel(x, y, background[y][farther_x]);
      }
    }
  }
```

Figure 10: Code for rolling background

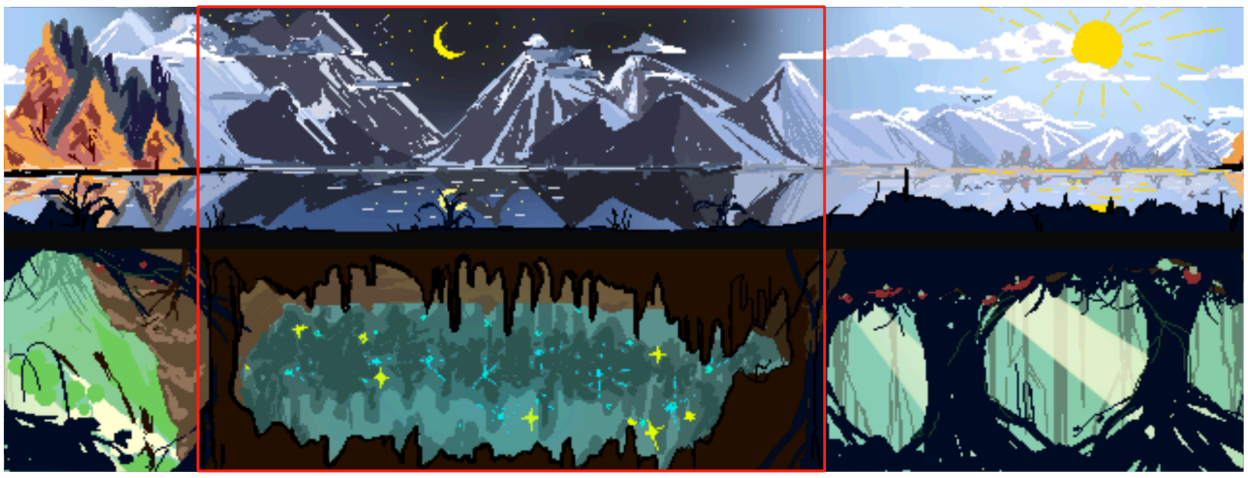

Figure 11: Long Background(640*240) for rolling showing on the VGA screen(320*240)

5 Attribution Table

Task	**Zhongyi Wang**	**Jiayi Xu**
Game Logic	√	
Game-State switching	√	√
Audio input		√
PS/2 input		√
VGA double buffering	√	√
Background & character rendering		√
Characters Moving Logic	√	√
Animation rendering	√	
Score Storing & Display	√	
Leaderboard Storing & Display	√	
Texting Display	√	
Character Design		√