





CPong


Vignesh N



C project

 [Synopsis](#)


 [System requirements](#)

 [System Design](#)

 [Demonstration](#)

 [Source code](#)

 [Output](#)

 [Scope for improvement](#)

 [Bibliography](#)



Synopsis

Pong is one of the earliest and most influential video games, often credited with launching the video game industry. Created by Atari in 1972, it simulated table tennis with two paddles and a bouncing ball. Players scored points by hitting the ball past their opponent. Despite simple graphics and gameplay, Pong's addictive and straightforward design made it a classic, laying the foundation for future gaming.

The main goal of **CPong** is to recreate a classic arcade game using modern programming tools and techniques. Developed in C with SDL2, CPong demonstrates how precise control over system resources can be used to build efficient, responsive real-time applications. The project leverages SDL2 for handling 2D rendering and user input, providing a lightweight yet dynamic graphical environment.

CPong is designed with a clean, modular codebase that uses key programming constructs such as structures, pointers, conditional logic, and function-based organization. The core game loop efficiently manages input, updates game state, performs collision detection, and handles frame rendering to ensure smooth, interactive gameplay. The inclusion of a computer-controlled paddle introduces simple AI behavior that tracks the ball's movement, providing a challenging and engaging single-player experience.

CPong offers multiple benefits beyond entertainment. The fast-paced gameplay promotes:

- **Improved hand-eye coordination:** Players must react quickly and accurately to control the paddle and intercept the ball.
- **Enhanced focus and concentration:** Sustained attention is required to track the ball and anticipate its movement.
- **Sharper reflexes and response time:** The game trains the player to make quick decisions and movements under time pressure.
- **Pattern recognition:** Players begin to recognize trajectories and optimize reactions based on experience.

From a development perspective, CPong highlights real-time input handling, graphics rendering, and basic automation logic through AI. The use of C ensures efficient performance and deeper insight into how low-level systems interact with high-level game mechanics.

SDL2 supports flexible rendering and input capture, enabling the creation of interactive applications with minimal overhead.

In conclusion, CPong blends classic gameplay with modern implementation, offering not just an engaging game, but also a platform that enhances cognitive skills and showcases real-time application development.



System requirements

Hardware Requirements

- **Processor (CPU):** A modern multi-core processor (ex. Intel Core i3 or AMD Ryzen 3)
- **Memory (RAM):** At least 1 GB of RAM is recommended
- **Storage:** 50 MB free disk space for executable and assets
- **Graphics:** A dedicated graphics card or integrated graphics with OpenGL 2.1+ support
- **Display:** Monitor with a minimum resolution 800x600 pixels
- **Input:** Keyboard + mouse for optimal experience

Software Requirements

- **Operating System:**
 - **Linux:** Ubuntu 18.04+, Debian 9+, or any modern Linux distribution
 - **Windows:** Windows 7/8/10/11 (32-bit or 64-bit)
 - **macOS:** macOS 10.12 Sierra or later
- **Required Libraries:**
 - **SDL2:** SDL Version 2.0.0 (Simple DirectMedia Layer)
 - **glibc:** Standard C library (typically pre-installed on Linux)
- **Development Tools (for compilation):**
 - **GCC:** GNU Compiler Collection 4.8+
 - **Make:** Build automation tool
 - **pkg-config:** For library configuration
- **Installation Commands:**
 - **Ubuntu/Debian:** <https://github.com/libsdl-org/SDL/blob/main/docs/README-linux.md>

- *MacOS*: <https://github.com/libsdl-org/SDL/blob/main/docs/README-macos.md>
- *Windows*: <https://github.com/libsdl-org/SDL/blob/main/docs/README-windows.md>



System Design

Libraries and Headers

Library/Header	Purpose	Functions Used
<code>SDL2/SDL.h</code>	Core SDL2 library for graphics, input, window management	SDL_Init, SDL_CreateWindow, SDL_GetKeyboardState, etc.
<code>stdlib.h</code>	Standard C library for memory allocation and utilities	Standard memory functions
<code>stdio.h</code>	Standard input/output operations	printf, snprintf, fprintf

Core Data Structures

Structure	Members	Purpose
<code>ball_t</code>	x, y (position), w, h (dimensions), dx, dy (velocity)	Represents the game ball
<code>paddle_t</code>	x, y (position), w, h (dimensions)	Represents player and computer paddles
<code>button_t</code>	x, y, w, h (bounds), visible (state)	UI button elements

Global Variables

Variable	Type	Purpose
<code>ball</code>	<code>ball_t</code>	Main game ball instance
<code>paddle[2]</code>	<code>paddle_t</code> array	Player (index 1) and computer (index 0) paddles
<code>score[2]</code>	int array	Score tracking for both players
<code>width, height</code>	int	Current window dimensions
<code>scale_x, scale_y</code>	float	Scaling factors for window resizing
<code>window</code>	<code>SDL_Window*</code>	Main game window
<code>renderer</code>	<code>SDL_Renderer*</code>	SDL rendering context

Variable	Type	Purpose
game_paused	int	Game pause state flag

Key Functions Analysis

Function	Parameters	Return Type	Purpose
main()	argc, argv	int	Program entry point, main game loop
init()	width, height, argc, args	int	Initialize SDL, create window/renderer, load assets
init_game()	void	void	Initialize game objects with proper scaling
update_scaling()	void	void	Recalculate scaling factors when window resizes
move_ball()	void	void	Update ball position, handle collisions and scoring
move_paddle_comp()	void	void	Computer logic for computer paddle movement
move_paddle()	direction	void	Handle player paddle movement
check_collision()	ball_t, paddle_t	int	Detect collision between ball and paddle
check_score()	void	int	Check win conditions (score limit reached)
draw_ball()	void	void	Render ball to screen
draw_paddle()	void	void	Render both paddles to screen
draw_net()	void	void	Draw center court divider
draw_pause_button()	void	void	Render pause button UI element
draw_pause_menu()	void	void	Render pause menu overlay
draw_game_over()	player	void	Display win/lose screen
load_digit_images()	void	void	Load score digit bitmap files
restart_game()	void	void	Reset game state and scores

SDL2 Functions Used

SDL Function	Purpose	Usage Context
<code>SDL_Init()</code>	Initialize SDL subsystems	Program startup
<code>SDL_CreateWindowAndRenderer()</code>	Create game window and rendering context	Window creation
<code>SDL_LoadBMP()</code>	Load bitmap image files	Asset loading
<code>SDL_CreateRGBSurfaceWithFormat()</code>	Create software rendering surface	Screen buffer creation
<code>SDL_GetKeyboardState()</code>	Get current keyboard input state	Input handling
<code>SDL_PollEvent()</code>	Process SDL events	Event loop
<code>SDL_FillRect()</code>	Draw filled rectangles	Rendering game objects
<code>SDL_BlitterScaled()</code>	Copy and scale surface to another	Image rendering
<code>SDL_UpdateTexture()</code>	Update hardware texture from surface	Screen refresh
<code>SDL_RenderCopy()</code>	Copy texture to renderer	Final rendering
<code>SDL_RenderPresent()</code>	Display rendered frame	Screen presentation
<code>SDL_Delay()</code>	Pause execution for specified milliseconds	Frame rate control
<code>SDL_GetTicks()</code>	Get milliseconds since SDL initialization	Timing calculations
<code>SDL_Quit()</code>	Clean up SDL subsystems	Program cleanup

Game Features

Feature	Implementation	Key Functions
Ball Physics	Velocity-based movement with collision detection	<code>move_ball()</code> , <code>check_collision()</code>
Computer Opponent	Predictive movement based on ball trajectory	<code>move_paddle_comp()</code>
Resizable Window	Dynamic scaling of all game elements	<code>update_scaling()</code> , scaling functions
Pause System	Overlay menu with resume/restart/exit options	<code>draw_pause_menu()</code> , pause handling
Score Display	Bitmap-based digit rendering	<code>draw_player_0_score()</code> , <code>draw_player_1_score()</code>

Feature	Implementation	Key Functions
Game States	Menu, gameplay, game over, pause states	State machine in main loop

Asset Dependencies

Asset File	Purpose	Format
title.bmp	Main menu title image	24-bit BMP
digits/0.bmp - digits/9.bmp	Score display numbers	24-bit BMP
player_win.bmp	Player victory screen	24-bit BMP
comp_win.bmp	AI victory screen	24-bit BMP
resume.bmp	Pause menu resume option	24-bit BMP
restart.bmp	Pause menu restart option	24-bit BMP
exit.bmp	Pause menu exit option	24-bit BMP

Compilation Instructions

```
# Linux/macOS
gcc pong.c `sdl2-config --cflags --libs` -o pong

# Alternative with manual linking
gcc pong.c -ISDL2 -ISDL2main -o pong

# Windows (MinGW)
gcc pong.c -lmingw32 -ISDL2main -ISDL2 -o pong.exe
```

Performance Characteristics

- **Frame Rate:** Locked to 60 FPS
- **Memory Usage:** Low (~5-10 MB typical)
- **Processor Usage:** Minimal, single-threaded
- **Scalability:** Fully scalable UI supporting window resizing



Demonstration

Pre-requisites

Required Assets

- `title.bmp` - Main menu background
- `digits/0.bmp` through `digits/9.bmp` - Score display numbers
- `player_win.bmp` - Player victory screen
- `ai_win.bmp` - AI victory screen
- `resume.bmp` , `restart.bmp` , `exit.bmp` - Pause menu options

Compilation

```
gcc -o pong pong3.c -ISDL2 -ISDL2main
```

Interactions

Main Menu Screen

Visual Elements:

- Game title displayed centrally
- "Press SPACE to start" instruction

User Interactions:

- `SPACE` → Start game
- `ESC` → Exit application
- Window resize → Interface scales proportionally

Gameplay Screen

Visual Elements:

- Left paddle (AI-controlled, white rectangle)
- Right paddle (player-controlled, white rectangle)
- Ball (white square)
- Dotted center line
- Score display (AI score | Player score) at top
- Pause button (top-right corner with pause symbol)

User Interactions:

- **UP Arrow** → Move player paddle up
- **DOWN Arrow** → Move player paddle down
- **Mouse click on pause button** → Open pause menu
- **ESC** → Exit to desktop
- Window resize → Game elements rescale automatically

Pause Menu

Visual Elements:

- Semi-transparent dark overlay
- Centered menu box with options:
 - RESUME (highlighted in blue when selected)
 - RESTART
 - EXIT

User Interactions:

- **UP/DOWN Arrows** → Navigate menu options
- **ENTER** or **SPACE** → Select highlighted option
- Resume → Return to game
- Restart → Reset scores and ball position

- Exit → Return to main menu

Game Over Screen

Visual Elements:

- Victory image (player_win.bmp or comp_win.bmp)
- Centered display showing winner

User Interactions:

- **SPACE** → Return to main menu

Sample Use Cases

Use Case 1 - Complete Game Session

1. Launch application → Main menu appears
2. Press SPACE → Game starts with ball moving
3. Use UP/DOWN arrows → Control right paddle
4. Ball bounces off paddles → Speed increases slightly
5. Ball exits screen → Opponent scores
6. Score reaches 2 → Game over screen shows winner
7. Press SPACE → Return to main menu

Use Case 2 - Pause/Resume Functionality

1. Start game → Ball and paddles active
2. Click pause button → Game freezes, menu appears
3. Use arrow keys → Navigate to "RESUME"
4. Press ENTER → Game continues from exact position
5. Pause again → Navigate to "RESTART"
6. Press ENTER → Scores reset to 0-0, ball resets

Use Case 3 - Window Resizing

1. Start game in windowed mode
2. Drag window corner → Resize window
3. Observe → All elements scale proportionally
4. Paddle controls remain responsive
5. Ball physics adjust to new dimensions



Source code

```
#include <SDL2/SDL.h>
#include <stdlib.h>
#include <stdio.h>

#define SCREEN_WIDTH 800
#define SCREEN_HEIGHT 600

//function prototypes

//initilise SDL
int init(int w, int h, int argc, char *args[]);

typedef struct ball_s {
    int x, y; /* position on the screen */
    int w,h; // ball width and height
    int dx, dy; /* movement vector */
} ball_t;

typedef struct paddle {
    int x,y;
    int w,h;
} paddle_t;

typedef struct button {
    int x, y, w, h;
    int visible;
} button_t;

// Program globals
static ball_t ball;
static paddle_t paddle[2];
```

```

int score[] = {0,0};
int width, height;    //current window size
float scale_x = 1.0f, scale_y = 1.0f; //scaling factors for resizable window

SDL_Window* window = NULL; //The window where SDL will be render
SDL_Renderer *renderer;    //The renderer SDL will use to draw to the screen

//surfaces
static SDL_Surface *screen;
static SDL_Surface *title;
//static SDL_Surface *numbermap;
//static SDL_Surface *end;

//score's numbers
static SDL_Surface* digits[11];

//game over results
static SDL_Surface* player_win_img ;
static SDL_Surface* comp_win_img ;

// Pause menu image surfaces
static SDL_Surface *resume_img;
static SDL_Surface *restart_img;
static SDL_Surface *exit_img;

//textures
SDL_Texture *screen_texture;

//pause system
button_t pause_button;
int game_paused = 0;
int pause_menu_selection = 0; // 0 = resume, 1 = restart, 2 = exit

//mouse state
int mouse_x, mouse_y;
int mouse_clicked = 0;

```



```

//calculate scaled size based on current window size
int scale_size(int original_size, float scale_factor) {
    return (int)(original_size * scale_factor);
}

//calculate scaled position
int scale_pos_x(int original_pos) {
    return (int)(original_pos * scale_x);
}

int scale_pos_y(int original_pos) {
    return (int)(original_pos * scale_y);
}

//initialize pause button
static void init_pause_button() {
    pause_button.w = scale_size(30, scale_x);
    pause_button.h = scale_size(30, scale_y);
    pause_button.x = width - pause_button.w - scale_size(10, scale_x);
    pause_button.y = scale_size(10, scale_y);
    pause_button.visible = 1;
}

//check if point is inside button
int point_in_button(int px, int py, button_t btn) {
    return (px >= btn.x && px <= btn.x + btn.w && py >= btn.y && py <= btn.y +
}

//update scaling factors based on current window size
static void update_scaling() {
    SDL_GetWindowSize(window, &width, &height);
    scale_x = (float)width / SCREEN_WIDTH;
    scale_y = (float)height / SCREEN_HEIGHT;

    //recreate screen surface with new size
    if (screen) {
        SDL_FreeSurface(screen);
    }
}

```

```

screen = SDL_CreateRGBSurfaceWithFormat(0, width, height, 32,
                                         SDL_PIXELFORMAT_RGBA3

//recreate screen texture
if (screen_texture) {
    SDL_DestroyTexture(screen_texture);
}
screen_texture = SDL_CreateTextureFromSurface(renderer, screen);

//update pause button position and size
init_pause_button();
}

//initialize starting position and sizes of game elements
static void init_game() {
    ball.x = scale_pos_x(SCREEN_WIDTH / 2);
    ball.y = scale_pos_y(SCREEN_HEIGHT / 2);
    ball.w = scale_size(10, scale_x);
    ball.h = scale_size(10, scale_y);
    ball.dy = scale_size(1, scale_y);
    ball.dx = scale_size(1, scale_x);

    paddle[0].x = scale_size(20, scale_x);
    paddle[0].y = scale_pos_y(SCREEN_HEIGHT / 2) - scale_size(50, scale_y);
    paddle[0].w = scale_size(10, scale_x);
    paddle[0].h = scale_size(50, scale_y);

    paddle[1].x = width - scale_size(20, scale_x) - scale_size(10, scale_x);
    paddle[1].y = scale_pos_y(SCREEN_HEIGHT / 2) - scale_size(50, scale_y);
    paddle[1].w = scale_size(10, scale_x);
    paddle[1].h = scale_size(50, scale_y);

    init_pause_button();
}

//restart game function - resets scores and reinitializes game
static void restart_game() {
    score[0] = 0;

```

```

    score[1] = 0;
    init_game();
}

int check_score() {
    int i;

    //loop through player scores
    for(i = 0; i < 2; i++) {
        //check if score is @ the score win limit
        if (score[i] == 2 ) {
            //reset scores
            score[0] = 0;
            score[1] = 0;

            //return 1 if computer wins
            if (i == 0) {
                return 1;
            }
            //return 2 if player wins
            else {
                return 2;
            }
        }
    }

    //return 0 if no one has reached a score of 10 yet
    return 0;
}

//if return value is 1 collision occurred. if return is 0, no collision.
int check_collision(ball_t a, paddle_t b) {
    int left_a, left_b;
    int right_a, right_b;
    int top_a, top_b;
    int bottom_a, bottom_b;

    left_a = a.x;
    right_a = a.x + a.w;

```

```

top_a = a.y;
bottom_a = a.y + a.h;

left_b = b.x;
right_b = b.x + b.w;
top_b = b.y;
bottom_b = b.y + b.h;

if (left_a > right_b) {
    return 0;
}

if (right_a < left_b) {
    return 0;
}

if (top_a > bottom_b) {
    return 0;
}

if (bottom_a < top_b) {
    return 0;
}

return 1;
}

/* This routine moves each ball by its motion vector. */
static void move_ball() {
    /* Move the ball by its motion vector. */
    ball.x += ball.dx;
    ball.y += ball.dy;

    /* Turn the ball around if it hits the edge of the screen. */
    if (ball.x < 0) {
        score[1] += 1;
        init_game();
    }
}

```

```

if (ball.x > width - ball.w) {
    score[0] += 1;
    init_game();
}

if (ball.y < 0 || ball.y > height - ball.h) {
    ball.dy = -ball.dy;
}

//check for collision with the paddle
int i;

for (i = 0; i < 2; i++) {
    int c = check_collision(ball, paddle[i]);

    //collision detected
    if (c == 1) {
        //ball moving left
        if (ball.dx < 0) {
            ball.dx -= scale_size(1, scale_x);
        }
        //ball moving right
        } else {
            ball.dx += scale_size(1, scale_x);
        }

        //change ball direction
        ball.dx = -ball.dx;

        //change ball angle based on where on the paddle it hit
        int hit_pos = (paddle[i].y + paddle[i].h) - ball.y;
        int paddle_segment = paddle[i].h / 9;

        if (hit_pos >= 0 && hit_pos < paddle_segment) {
            ball.dy = scale_size(4, scale_y);
        } else if (hit_pos >= paddle_segment
            && hit_pos < paddle_segment * 2) {
            ball.dy = scale_size(3, scale_y);
        }
    }
}

```

```

    } else if (hit_pos >= paddle_segment * 2
               && hit_pos < paddle_segment * 3) {
        ball.dy = scale_size(2, scale_y);
    } else if (hit_pos >= paddle_segment * 3
               && hit_pos < paddle_segment * 4) {
        ball.dy = scale_size(1, scale_y);
    } else if (hit_pos >= paddle_segment * 4
               && hit_pos < paddle_segment * 5) {
        ball.dy = 0;
    } else if (hit_pos >= paddle_segment * 5
               && hit_pos < paddle_segment * 6) {
        ball.dy = -scale_size(1, scale_y);
    } else if (hit_pos >= paddle_segment * 6
               && hit_pos < paddle_segment * 7) {
        ball.dy = -scale_size(2, scale_y);
    } else if (hit_pos >= paddle_segment * 7
               && hit_pos < paddle_segment * 8) {
        ball.dy = -scale_size(3, scale_y);
    } else {
        ball.dy = -scale_size(4, scale_y);
    }

    //ball moving right
    if (ball.dx > 0) {
        //teleport ball to avoid multi collision glitch
        if (ball.x < scale_size(30, scale_x)) {
            ball.x = scale_size(30, scale_x);
        }
    }
    //ball moving left
    } else {
        //teleport ball to avoid multi collision glitch
        if (ball.x > width - scale_size(40, scale_x)) {
            ball.x = width - scale_size(40, scale_x);
        }
    }
}
}
}
}
}

```

```

static void move_paddle_comp() {
    int center = paddle[0].y + paddle[0].h / 2;
    int screen_center = height / 2;
    int ball_speed = ball.dy;

    if (ball_speed < 0) {
        ball_speed = -ball_speed;
    }

    if (ball_speed == 0) {
        ball_speed = scale_size(1, scale_y);
    }

    //ball moving right
    if (ball.dx > 0) {
        //return to center position
        if (center < screen_center) {
            paddle[0].y += ball_speed;
        } else {
            paddle[0].y -= ball_speed;
        }
    }
    //ball moving left
    } else {
        //ball moving down
        if (ball.dy > 0) {
            if (ball.y > center) {
                paddle[0].y += ball_speed;
            } else {
                paddle[0].y -= ball_speed;
            }
        }
    }

    //ball moving up
    if (ball.dy < 0) {
        if (ball.y < center) {
            paddle[0].y -= ball_speed;
        } else {

```

```

        paddle[0].y += ball_speed;
    }
}

//ball moving straight across
if (ball.dy == 0) {
    if (ball.y < center) {
        paddle[0].y -= scale_size(5, scale_y);
    } else {
        paddle[0].y += scale_size(5, scale_y);
    }
}

//keep computer paddle on screen
if (paddle[0].y < 0) {
    paddle[0].y = 0;
}
if (paddle[0].y > height - paddle[0].h) {
    paddle[0].y = height - paddle[0].h;
}
}

static void move_paddle(int d) {
    int speed = scale_size(5, scale_y);

    // if the down arrow is pressed move paddle down
    if (d == 0) {
        if(paddle[1].y >= height - paddle[1].h) {
            paddle[1].y = height - paddle[1].h;
        } else {
            paddle[1].y += speed;
        }
    }

    // if the up arrow is pressed move paddle up
    if (d == 1) {
        if(paddle[1].y <= 0) {

```



```

        paddle[1].y = 0;
    } else {
        paddle[1].y -= speed;
    }
}
}

static void draw_game_over(int p) {

    SDL_Surface* img = (p == 1) ? player_win_img : comp_win_img;
    if (!img) return;

    SDL_Rect dest;
    dest.w = scale_size(img->w, scale_x);
    dest.h = scale_size(img->h, scale_y);

    dest.x = (width/2) - (dest.w/2);
    dest.y = (height/2) - (dest.h/2);

    SDL_BlittedScaled(img, NULL, screen, &dest);

    /*SDL_Rect p1, cpu, dest;

    p1.x = 0;
    p1.y = 0;
    p1.w = end->w;
    p1.h = 75;

    p2.x = 0;
    p2.y = 75;
    p2.w = end->w;
    p2.h = 75;

    cpu.x = 0;
    cpu.y = 150;
    cpu.w = end->w;
    cpu.h = 75;

```

```

dest.x = (width / 2) - scale_size(end→w / 2, scale_x);
dest.y = (height / 2) - scale_size(75 / 2, scale_y);
dest.w = scale_size(end→w, scale_x);
dest.h = scale_size(75, scale_y);

switch (p) {
    case 1:
        SDL_BlitScaled(end, &p1, screen, &dest);
        break;
    case 2:
        SDL_BlitScaled(end, &p2, screen, &dest);
        break;
    default:
        SDL_BlitScaled(end, &cpu, screen, &dest);
}*/
}

static void draw_menu() {
    SDL_Rect src, dest;

    src.x = 0;
    src.y = 0;
    src.w = title→w;
    src.h = title→h;

    dest.x = (width / 2) - scale_size(src.w / 2, scale_x);
    dest.y = (height / 2) - scale_size(src.h / 2, scale_y);
    dest.w = scale_size(title→w, scale_x);
    dest.h = scale_size(title→h, scale_y);

    SDL_BlitScaled(title, &src, screen, &dest);
}

static void draw_pause_button() {
    if (!pause_button.visible) return;

    SDL_Rect button_rect;

```

```

button_rect.x = pause_button.x;
button_rect.y = pause_button.y;
button_rect.w = pause_button.w;
button_rect.h = pause_button.h;

//draw button background
SDL_FillRect(screen, &button_rect, 0xffffffff);

//draw button border
SDL_Rect border = button_rect;
border.x += 2;
border.y += 2;
border.w -= 4;
border.h -= 4;
SDL_FillRect(screen, &border, 0x000000ff);

//draw pause symbol (two vertical bars)
SDL_Rect bar1, bar2;
int bar_width = scale_size(4, scale_x);
int bar_height = scale_size(16, scale_y);
int center_x = pause_button.x + pause_button.w / 2;
int center_y = pause_button.y + pause_button.h / 2;

bar1.x = center_x - bar_width - scale_size(2, scale_x);
bar1.y = center_y - bar_height / 2;
bar1.w = bar_width;
bar1.h = bar_height;

bar2.x = center_x + scale_size(2, scale_x);
bar2.y = center_y - bar_height / 2;
bar2.w = bar_width;
bar2.h = bar_height;

SDL_FillRect(screen, &bar1, 0xffffffff);
SDL_FillRect(screen, &bar2, 0xffffffff);
}

// Draw a pause menu option using BMP image with selection highlight

```

```

static void draw_menu_option(SDL_Surface* image, int x, int y, int selected){
    if (image == NULL) return;

    SDL_Rect src, dest;

    // Source rectangle covers the entire image
    src.x = 0;
    src.y = 0;
    src.w = image→w;
    src.h = image→h;

    // Destination rectangle with scaling applied
    dest.x = x;
    dest.y = y;
    dest.w = scale_size(image→w, scale_x);
    dest.h = scale_size(image→h, scale_y);

    // Draw highlight background if this option is selected
    if (selected) {
        SDL_Rect highlight_rect;
        highlight_rect.x = dest.x - scale_size(10, scale_x);
        highlight_rect.y = dest.y - scale_size(5, scale_y);
        highlight_rect.w = dest.w + scale_size(20, scale_x);
        highlight_rect.h = dest.h + scale_size(10, scale_y);

        // Draw semi-transparent highlight background

        Uint32 bluee = SDL_MapRGB(screen→format, 173, 216, 230);
        SDL_FillRect(screen, &highlight_rect, bluee);
    }

    // Blit the scaled image to the screen
    SDL_BlittedScaled(image, &src, screen, &dest);
}

static void draw_pause_menu() {
    //draw semi-transparent overlay
    SDL_Rect overlay;

```

```

overlay.x = 0;
overlay.y = 0;
overlay.w = width;
overlay.h = height;
SDL_FillRect(screen, &overlay, 0x00000080);

//draw menu background
SDL_Rect menu_bg;
menu_bg.w = scale_size(300, scale_x);
menu_bg.h = scale_size(250, scale_y);
menu_bg.x = (width - menu_bg.w) / 2;
menu_bg.y = (height - menu_bg.h) / 2;
SDL_FillRect(screen, &menu_bg, 0x333333ff);

//draw menu border
SDL_Rect menu_border = menu_bg;
menu_border.x += 3;
menu_border.y += 3;
menu_border.w -= 6;
menu_border.h -= 6;

Uint32 white = SDL_MapRGBA(screen->format, 255, 255, 255, 255);
SDL_FillRect(screen, &menu_border, white);

// SDL_FillRect(screen, &menu_border, 0x0000ffaa);

// Calculate positions for menu options
int option_y_start = menu_bg.y + scale_size(45, scale_y);
int option_spacing = scale_size(60, scale_y);
int option_x = menu_bg.x + scale_size(50, scale_x);

// Draw "RESUME" option
draw_menu_option(resume_img, option_x, option_y_start,
                 (pause_menu_selection == 0));

// Draw "RESTART" option
draw_menu_option(restart_img, option_x, option_y_start + option_spacing,
                 (pause_menu_selection == 1));

```

```

// Draw "EXIT" option
draw_menu_option(exit_img, option_x, option_y_start + (option_spacing * 2),
                  (pause_menu_selection == 2));

}

static void draw_net() {
    SDL_Rect net;

    net.x = width / 2 - scale_size(2, scale_x);
    net.y = scale_size(20, scale_y);
    net.w = scale_size(5, scale_x);
    net.h = scale_size(15, scale_y);

    //draw the net
    int i, r;
    int segments = height / scale_size(37, scale_y);

    for(i = 0; i < segments; i++) {
        r = SDL_FillRect(screen, &net, 0xffffffff);
        if (r != 0) {
            printf("fill rectangle failed in func draw_net()");
        }
        net.y += scale_size(37, scale_y);
    }
}

static void draw_ball() {
    SDL_Rect src;

    src.x = ball.x;
    src.y = ball.y;
    src.w = ball.w;
    src.h = ball.h;

    int r = SDL_FillRect(screen, &src, 0xffffffff);

```

```

    if (r != 0){
        printf("fill rectangle failed in func drawball()");
    }
}

static void draw_paddle() {
    SDL_Rect src;
    int i;

    for (i = 0; i < 2; i++) {
        src.x = paddle[i].x;
        src.y = paddle[i].y;
        src.w = paddle[i].w;
        src.h = paddle[i].h;

        int r = SDL_FillRect(screen, &src, 0xffffffff);

        if (r != 0){
            printf("fill rectangle failed in func draw_paddle()");
        }
    }
}

void load_digit_images() {
    for (int i = 0; i <= 10; ++i) {
        char filename[32];
        snprintf(filename, sizeof(filename), "digits/%d.bmp", i);
        // e.g., digits/0.bmp ... digits/9.bmp

        digits[i] = SDL_LoadBMP(filename);
        if (!digits[i]) {
            fprintf(stderr, "Failed to load %s: %s\n", filename, SDL_GetError());
        }
    }
}

void free_digit_images() {

```

```

    for (int i = 0; i <= 10; ++i) {
        if (digits[i]) SDL_FreeSurface(digits[i]);
    }
}

static void draw_player_0_score() {
    if (score[0] < 0 || score[0] > 10) return;

    SDL_Surface* digit_img = digits[score[0]];
    if (!digit_img) return;

    SDL_Rect dest;

    dest.w = scale_size(digit_img->w, scale_x);
    dest.h = scale_size(digit_img->h, scale_y);

    dest.x = (width / 2) - dest.w - scale_size(12, scale_x);
    dest.y = 0;

    SDL_BlittedScaled(digit_img, NULL, screen, &dest);
}

static void draw_player_1_score() {
    if (score[1] < 0 || score[1] > 10) return;

    SDL_Surface* digit_img = digits[score[1]];
    if (!digit_img) return;

    SDL_Rect dest;

    dest.w = scale_size(digit_img->w, scale_x);
    dest.h = scale_size(digit_img->h, scale_y);

    dest.x = (width / 2) + scale_size(12, scale_x);
    dest.y = 0;

    SDL_BlittedScaled(digit_img, NULL, screen, &dest);
}

```



```

int main (int argc, char *args[]) {
    //SDL Window setup
    if (init(SCREEN_WIDTH, SCREEN_HEIGHT, argc, args) == 1) {
        return 0;
    }

    update_scaling();

    int sleep = 0;
    int quit = 0;
    int state = 0;
    int r = 0;
    Uint32 next_game_tick = SDL_GetTicks();
    SDL_Event e;

    // Initialize the ball position data.
    init_game();

    //render loop
    while(quit == 0) {
        //handle events
        while(SDL_PollEvent(&e) != 0) {
            if(e.type == SDL_QUIT) {
                quit = 1;
            }
            else if(e.type == SDL_WINDOWEVENT) {
                if(e.window.event == SDL_WINDOWEVENT_RESIZED) {
                    update_scaling();
                    init_game(); //reinitialize game elements with new scaling
                }
            }
            else if(e.type == SDL_MOUSEBUTTONDOWN) {
                if(e.button.button == SDL_BUTTON_LEFT) {
                    mouse_x = e.button.x;
                    mouse_y = e.button.y;
                    mouse_clicked = 1;
                }
            }
        }
    }
}

```

```

    }
}

const Uint8 *keystate = SDL_GetKeyboardState(NULL);

if (keystate[SDL_SCANCODE_ESCAPE]) {
    quit = 1;
}

//handle mouse clicks on pause button
if (mouse_clicked && state == 1 && !game_paused) {
    if (point_in_button(mouse_x, mouse_y, pause_button)) {
        game_paused = 1;
        pause_menu_selection = 0;
    }
    mouse_clicked = 0;
}

//handle pause menu
if (game_paused && state == 1) {
    if (keystate[SDL_SCANCODE_UP]) {
        pause_menu_selection--;
        if (pause_menu_selection < 0) {
            pause_menu_selection = 2; //wrap to bottom option
        }
        SDL_Delay(200); //prevent rapid selection changes
    }

    if (keystate[SDL_SCANCODE_DOWN]) {
        pause_menu_selection++;
        if (pause_menu_selection > 2) {
            pause_menu_selection = 0; //wrap to top option
        }
        SDL_Delay(200); //prevent rapid selection changes
    }

    if (keystate[SDL_SCANCODE_RETURN] || keystate[SDL_SCANCODE_SI

```

```

        if (pause_menu_selection == 0) {
            game_paused = 0; //resume
        } else if (pause_menu_selection == 1) {
            restart_game(); //restart
            game_paused = 0;
        } else {
            state = 0; //exit to main menu
            game_paused = 0;
            restart_game(); //reset game state
        }
        SDL_Delay(200);
    }
}

//game controls (only when not paused)
if (!game_paused) {
    if (keystate[SDL_SCANCODE_DOWN]) {
        move_paddle(0);
    }

    if (keystate[SDL_SCANCODE_UP]) {
        move_paddle(1);
    }
}

//draw background
SDL_RenderClear(renderer);
SDL_FillRect(screen, NULL, 0x000000ff);

//display main menu
if (state == 0 ) {
    if (keystate[SDL_SCANCODE_SPACE]) {
        state = 1;
    }
    draw_menu();
}

//display gameover
} else if (state == 2) {

```

```

    if (keystate[SDL_SCANCODE_SPACE]) {
        state = 0;
        SDL_Delay(500);
    }

    if (r == 1) {
        draw_game_over(0); //player wins
    } else {
        draw_game_over(1); //computer wins
    }

//display the game
} else if (state == 1) {
    //only update game logic when not paused
    if (!game_paused) {
        //check score
        r = check_score();

        //if either player wins, change to game over state
        if (r == 1 || r == 2) {
            state = 2;
        }

        //paddle computer movement
        move_paddle_comp();

        //Move the balls for the next frame.
        move_ball();
    }

//draw game elements
draw_net();
draw_paddle();
draw_ball();
draw_player_0_score();
draw_player_1_score();
draw_pause_button();

```

```

        //draw pause menu if paused
        if (game_paused) {
            draw_pause_menu();
        }
    }

    SDL_UpdateTexture(screen_texture, NULL,
                      screen->pixels, screen->w * sizeof (Uint32));
    SDL_RenderCopy(renderer, screen_texture, NULL, NULL);

    //draw to the display
    SDL_RenderPresent(renderer);

    //time it takes to render frame in milliseconds
    next_game_tick += 1000 / 60;
    sleep = next_game_tick - SDL_GetTicks();

    if( sleep >= 0 ) {
        SDL_Delay(sleep);
    }

    mouse_clicked = 0; //reset mouse click state
}

//free loaded images
SDL_FreeSurface(screen);
SDL_FreeSurface(title);
//SDL_FreeSurface(numbermap);
//SDL_FreeSurface(end);

free_digit_images();

if (player_win_img) SDL_FreeSurface(player_win_img);
if (comp_win_img) SDL_FreeSurface(comp_win_img);

// Free pause menu images
if (resume_img) SDL_FreeSurface(resume_img);
if (restart_img) SDL_FreeSurface(restart_img);

```

```

    if (exit_img) SDL_FreeSurface(exit_img);

    //free renderer and all textures used with it
    SDL_DestroyRenderer(renderer);

    //Destroy window
    SDL_DestroyWindow(window);

    //Quit SDL subsystems
    SDL_Quit();

    return 0;
}

int init(int width, int height, int argc, char *args[]) {
    //Initialize SDL
    if (SDL_Init(SDL_INIT_VIDEO) < 0) {
        printf("SDL could not initialize! SDL_Error: %s\n", SDL_GetError());
        return 1;
    }

    int i;

    for (i = 0; i < argc; i++) {
        //Create window
        if(strcmp(args[i], "-f")) {
            SDL_CreateWindowAndRenderer(width, height,
                                         SDL_WINDOW_SHOWN | SDL_WINDOW_RESIZABLE, &window,
            } else {
                SDL_CreateWindowAndRenderer(width, height,
                                             SDL_WINDOW_SHOWN | SDL_WINDOW_RESIZABLE | SDL_WIN
                                             &window, &renderer);
            }
        }
    }

    // Check if window and renderer were created successfully
    if (window == NULL || renderer == NULL) {
        printf("Window or Renderer could not be created! SDL Error: %s\n",

```

```

        SDL_GetError());
    return 1;
}

// Load BMP images for the game
screen = SDL_CreateRGBSurfaceWithFormat(0, width, height, 32,
                                         SDL_PIXELFORMAT_RGBA32);

title = SDL_LoadBMP("title.bmp");
//numbermap = SDL_LoadBMP("numbermap.bmp");
//end = SDL_LoadBMP("gameover.bmp");
load_digit_images();
player_win_img = SDL_LoadBMP("player_win.bmp");
comp_win_img = SDL_LoadBMP("comp_win.bmp");

// Load pause menu BMP images - this is what was missing!
resume_img = SDL_LoadBMP("resume.bmp");
restart_img = SDL_LoadBMP("restart.bmp");
exit_img = SDL_LoadBMP("exit.bmp");

// Check if critical images loaded successfully
if (title == NULL /*|| numbermap == NULL || end == NULL */) {
    printf("Failed to load game images! SDL Error: %s\n", SDL_GetError());
    return 1;
}

// Check if pause menu images loaded (warn but don't fail)
if (resume_img == NULL || restart_img == NULL || exit_img == NULL) {
    printf("Warning: Failed to load pause menu images! SDL Error: %s\n",
          SDL_GetError());
    printf("Make sure resume.bmp, restart.bmp,
          and exit.bmp are in the same directory\n");
}

// Create screen texture
screen_texture = SDL_CreateTextureFromSurface(renderer, screen);
if (screen_texture == NULL) {
    printf("Failed to create screen texture! SDL Error: %s\n", SDL_GetError());
    return 1;
}

```

```
}

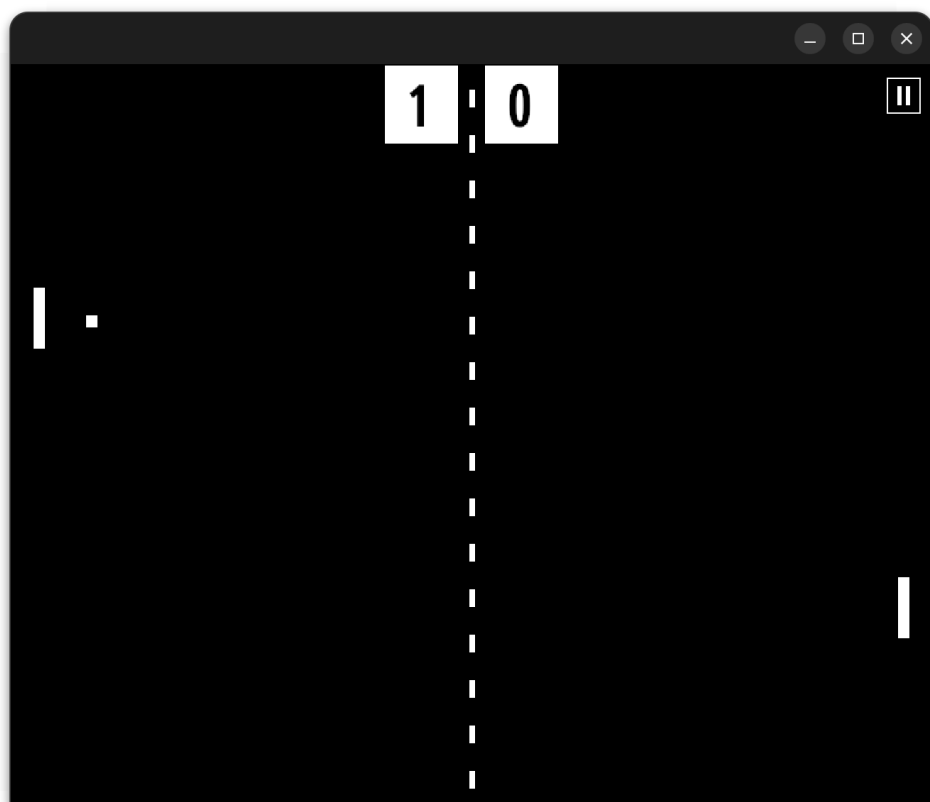
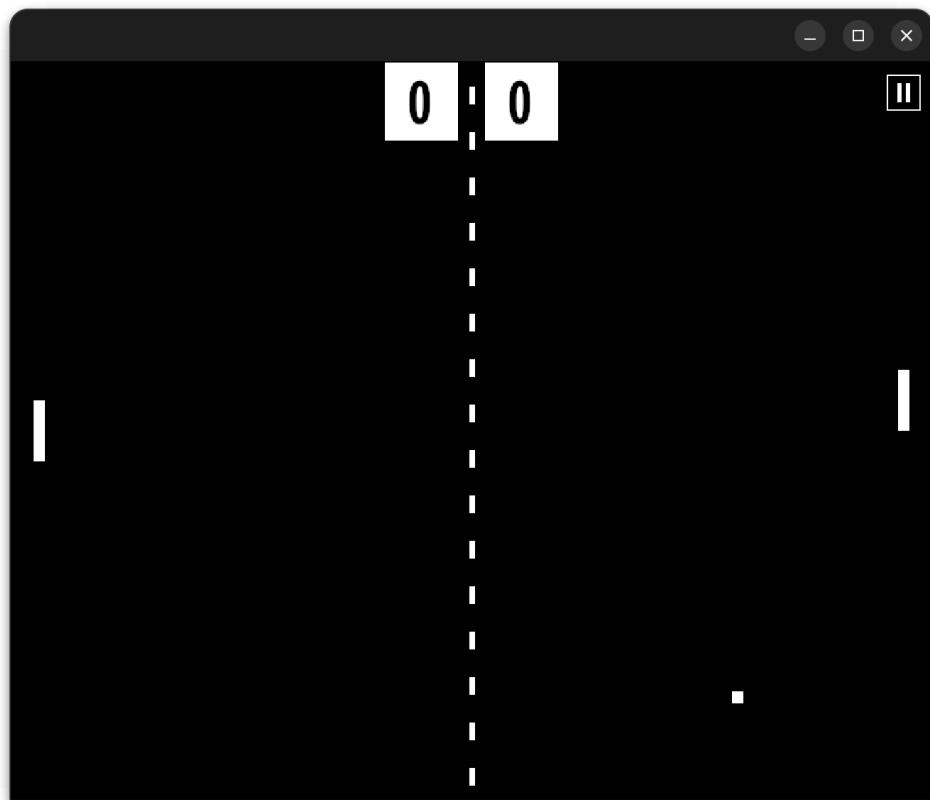
if (!player_win_img || !comp_win_img) {
    printf("Warning: Failed to load game over images! SDL Error: %s\n",
          SDL_GetError());

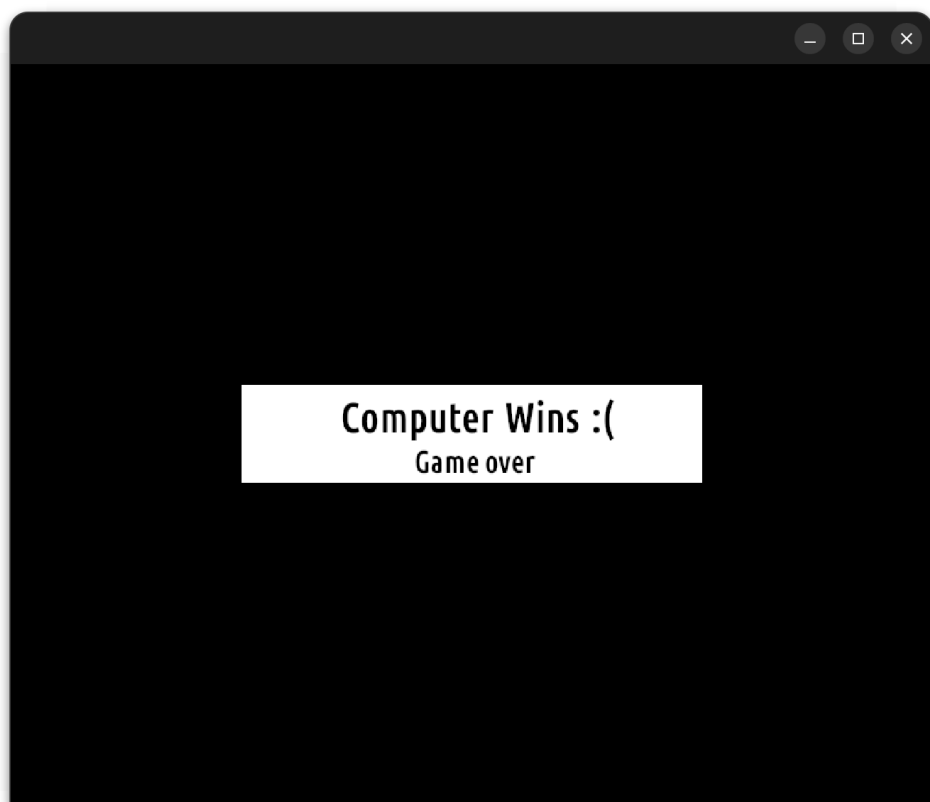
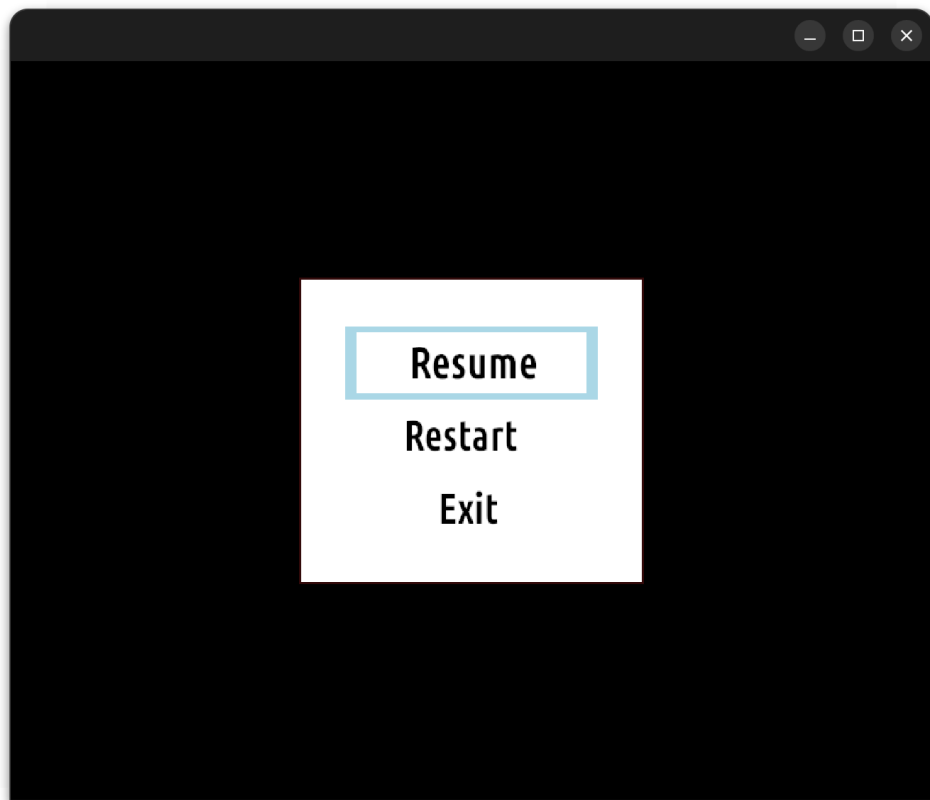
    return 0;}
}
```

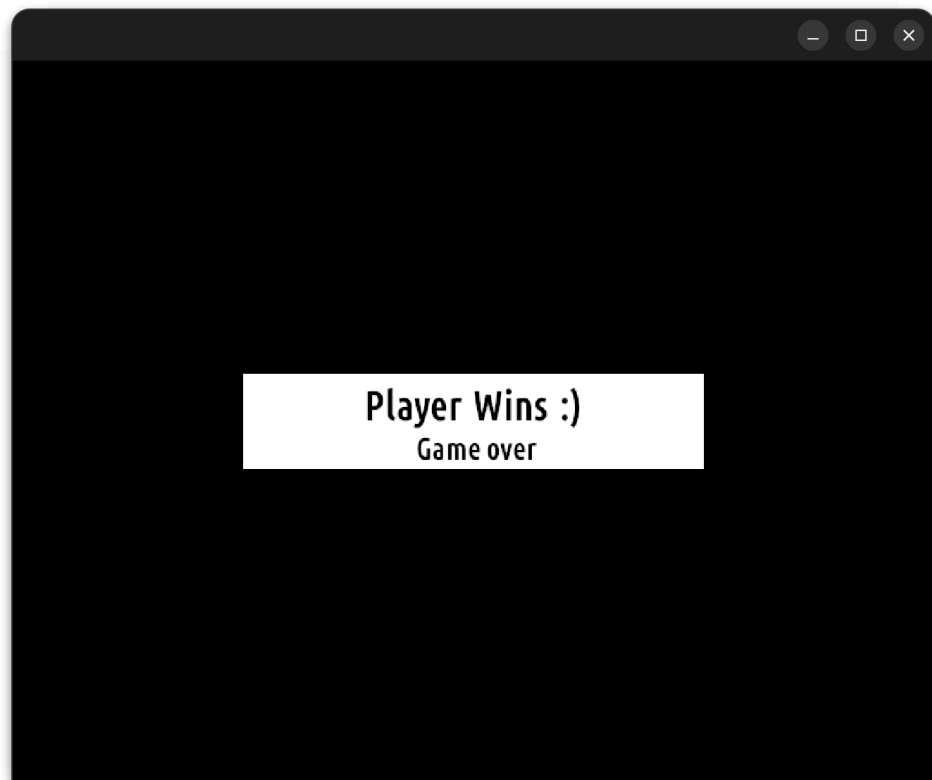



Output











Scope for improvement

Graphics & Visual Polish

- Replace basic rectangles with textured and rounded sprites for paddles and ball
- Add particle effects for ball collisions and scoring
- Implement smooth animations and transitions between game states
- Add background themes and visual customization options

Gameplay Features

- Multiple difficulty levels with adjustable AI intelligence
- Two-player local multiplayer mode
- Power-ups (speed boost, larger paddle, multi-ball)
- Different game modes
- Tournament bracket system

Audio System

- Sound effects for paddle hits, scoring, and menu navigation
- Background music with volume controls
- Audio feedback for game events

User Interface

- Settings menu for game configuration
- High score tracking and leaderboards
- Improved menu system with mouse support throughout
- In-game statistics display (rally count, ball speed)

Code Quality

- Modular code structure with separate files for different systems
- Configuration file for easy gameplay tweaking
- Improved memory management and resource cleanup



Bibliography

<https://www.w3schools.com/>

<https://github.com/>

<https://github.com/libsdl-org/SDL/>

<https://wiki.libsdl.org/SDL2/>

<https://lazyfoo.net/tutorials/SDL/index.php>