EEET2490 – Embedded System: OS and Interfacing, Semester 2023-2

Assessment 3 – Group Work Report

SCREEN DISPLAY & APPLICATION DEVELOPMENT FOR BARE METAL OS

Lecturers: Mr Linh Tran – linh.tranduc@rmit.edu.vn,

Mr. Phuc Nguyen - phuc.nguyenhoangthien@rmit.edu.vn

Team Number: 11

Team Members:

Tran Truong Son (s3818468)

Vu Thien Nhan (s3810151)

Phan Ngoc Quang Anh (s3810148)

Hoang Dinh Tri (s3877818)

Date : 26/09/2023

TABLE OF CONTENTS

I. INTRODUCTION

In this assignment, we are given 2 tasks, the first is to continue developing our bare metal OS, using the experience we learned from the last assignment, the OS now has to be able to display colored images and videos on screen, as well as display messages on the screen using a custom font, done by our own team. Secondly, we need to develop a small game of our choosing to run on our bare metal OS. The OS will also be running on a physical Raspberry Pi board instead of running virtually on our PCs like in the last assignment.

This report serves as a documentation of the work we have completed, explaining how the work was done as well as discussing the work done.

II. SCREEN DISPLAY

BACKGROUND

Before we do any work on the project, we must first implement some of the basic functions, which are utilized by various methods. In our math.c, we created our own mathematical functions, such as square root (sqrt), power (pow), rounding to the closest integer (round), rounding up (ceil), max, min, absolute (abs), float absolute (fabs), and lastly the trigonometry calculations sin cos tan. Then, we need to configure UART, mailbox, kernel, etc. for the system to work, these refer closely to the given instructions from the lecturers.

IMPLEMENTATION

Image and Video

The OS's Image and Video must be displayed with colored pixels, as opposed to black and white.

For both image and video to be displayed, we first needed to set the screen's size, which is 1024x768, done in framebf.c, then all of the necessary data are converted using following the tutorial provided in the assignment details document [https://javl.github.io/image2cpp/] and stored in the image.c, then defined in the following structure, where each image data is stored in an array:

```
Image image_name = { width, height, image_data };
```

And the video's data is done similarly, where each frame of the video is stored like a picture, then put in a larger video_data array, simulated as this:

```
video_data[] = {frame1_data[], … , frameN_data[]}
```

To display the images and video, we use the display_image() and display_video() functions in framebf.c. The details of each function are as written below.

For images, to ensure that each image is correctly positioned, since each image can have different width and height, we first need to locate the image's center coordinates and its bounds. Then we can start printing the image by moving the position of the pixel pointer to the (0,0) coordinates and start drawing the image pixel by pixel, line by line and fill the pixel's color using drawPixelARGB32() function. The rotational parts of the function are implemented for the game section of the assignment; it is not explained here since we do not rotate the images for this section.

```
void display_image(int x, int y, int image_width, int image_height, const
unsigned int *image_data, int transparency, int rotation) {

    // Calculate the rotation in radians

    double rotation_rad = -rotation * DEG_TO_RAD;


    // Calculate the maximum rotated dimensions

    double cos_rotation = cos(rotation_rad);

    double sin_rotation = sin(rotation_rad);


    // Calculate the center coordinates of the image

    int center_x = x + image_width / 2;

    int center_y = y + image_height / 2;


    // Calculate the maximum bounds of the rotated image

    int half_bbox_w = ceil((fabs(image_width * cos_rotation) +
fabs(image_height * sin_rotation)) / 2.0);

    int half_bbox_h = ceil((fabs(image_width * sin_rotation) +
fabs(image_height * cos_rotation)) / 2.0);


    // Adjust the start and end points of the loops

    int start_x = max(0, center_x - half_bbox_w);

    int start_y = max(0, center_y - half_bbox_h);

    int end_x = min(SCREEN_WIDTH, center_x + half_bbox_w);

    int end_y = min(SCREEN_HEIGHT, center_y + half_bbox_h);


    for (int j = start_y; j < end_y; j++) {
```

```
        for (int i = start_x; i < end_x; i++) {
            // Translate the pixel coordinates relative to the center
            int translated_i = i - center_x;
            int translated_j = j - center_y;

            // Apply rotation to the translated coordinates
            int rotated_i = round(translated_i * cos_rotation - translated_j
* sin_rotation + center_x);
            int rotated_j = round(translated_i * sin_rotation + translated_j
* cos_rotation + center_y);

            // Check if the rotated coordinates are within the image bounds
            if (rotated_i >= x && rotated_i < x + image_width && rotated_j >=
y && rotated_j < y + image_height) {
                unsigned int color = image_data[(rotated_j - y) * image_width
+ (rotated_i - x)];
                if (!transparency || color)
                    drawPixelARGB32(i, j, color);
            }
        }
    }
}
```

```
void drawPixelARGB32(int x, int y, unsigned int attr)
{
    int offs = (y * pitch) + (COLOR_DEPTH / 8 * x);
    // Access 32-bit together
    *((unsigned int *)(fb + offs)) = attr;
}
```

Displaying video is basically the same as displaying the images, we just need to set a frame data array for the video using VIDEO_FRAMES and draw each frame of the video as an image and delay the drawing process by a duration calculated using VIDEO_FPS.

```
void display_video(int x, int y)
{
    int delay_ms = (1.0 / VIDEO_FPS) * 1000;

    for (int i = 0; i < VIDEO_FRAMES; i++) {
        set_wait_timer(1, delay_ms);
        if (uart_isReadByteReady()) return;
        display_image(x, y, VIDEO_WIDTH, VIDEO_HEIGHT, video_data[i], 0, 0);
        set_wait_timer(0, delay_ms);
    }
}
```

Font

The OS's Font will be drawn in the form of glyphs, where each pixel data of the character is stored in a bitmap to display our team members' names, each name should be in different colors.

For our Font, first, we had to draw out each character available, using the provided .ttf file in [https://www.cufonfonts.com/font/segoe-ui-4], in our font.c. The font we used was Segoe UI, converted using the online font converter tool [https://lvgl.io/tools/font_conv_v5_3]. Then we stored all of the characters in a bitmap, a gave each of the characters a description using glyph_desc, which includes the character's width and the starting bitmap index. The universal height for all of the characters is 48 pixels.

```
struct glyph_desc {
  unsigned int width_px;
  unsigned int glyph_index;
};
```

```
{.width_px = 13,   .glyph_index = 0},
```

Similar to how image and video are displayed, the font will also be displayed using the draw_char() and draw_string() functions in framebf.c. The draw_char() function is used to draw a character line by

line using the provided data while the draw_string() function is used to loop through a string to draw each character until the string ends. Then to color each text, we use the drawRectARGB32() function, which selects and colors a whole block of pixels based on the scale, instead of each individual pixel like the drawPixelARGB32() function by finding the starting and ending pixel points and filling all of the ON pixels with a selected color.

```
void draw_char(int x, int y, unsigned char ch, unsigned int attr, float
scale)
{
    if (ch < UNICODE_FIRST || ch > UNICODE_LAST) return;
    int char_index = ch - UNICODE_FIRST;
    int glyph_index = glyph_desc[char_index].glyph_index;
    int glyph_width = glyph_desc[char_index].width_px;

    for (int row = 0; row < HEIGHT_PX; row++) {
        for (int col = 0; col < glyph_width; col++) {
            int pixel_x = (int)(x + col * scale);
            int pixel_y = (int)(y + row * scale);
            int bitmap_byte = glyph_index + row * ((glyph_width + 7) / 8) +
col / 8;

            if (bitmap[bitmap_byte] & (1 << (7 - col % 8))) {
                drawRectARGB32(pixel_x, pixel_y, pixel_x + (int)scale,
pixel_y + (int)scale, attr, 1);
            }
        }
    }
}
```

```
void draw_string(int x, int y, char *s, unsigned int attr, int spacing, float
scale)
{
    if (s == 0) return;
```

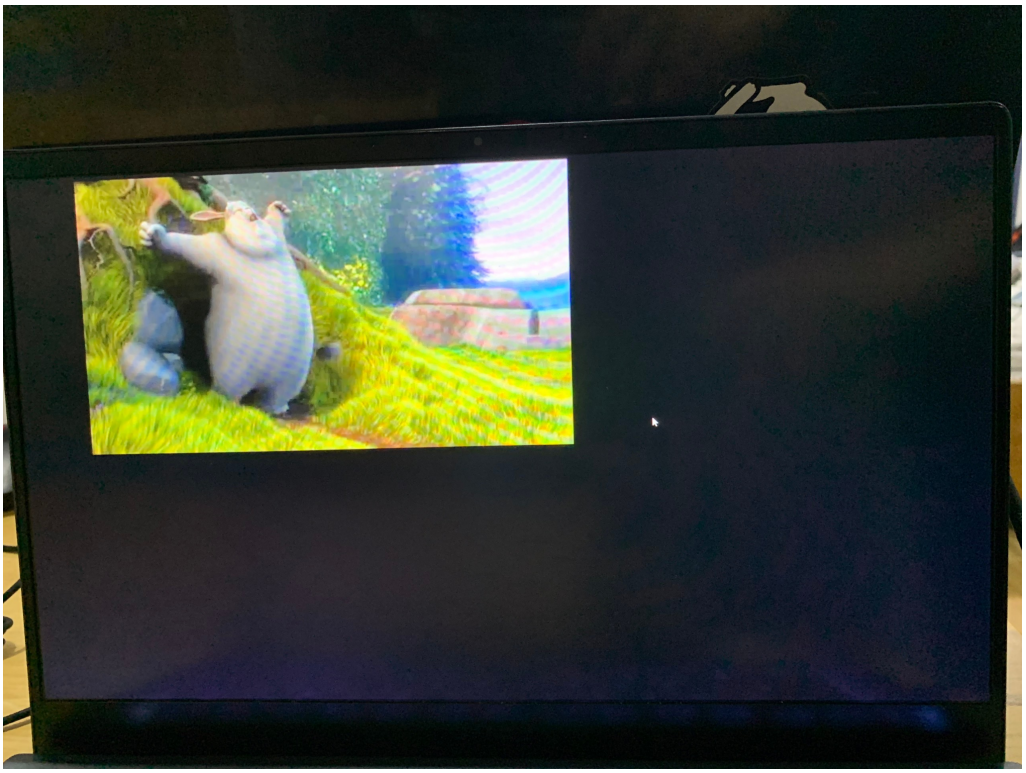
```
    int curr_x = x;


    while (*s != '\0') {
        draw_char(curr_x, y, *s, attr, scale);
        curr_x += (glyph_desc[*s - UNICODE_FIRST].width_px + spacing) *
scale;
        s++;
    }
}
```

```
void drawRectARGB32(int x1, int y1, int x2, int y2, unsigned int attr, int
fill)
{
    x1 = max(0, x1);
    y1 = max(0, y1);
    x2 = min(SCREEN_WIDTH, x2);
    y2 = min(SCREEN_HEIGHT, y2);
    for (int y = y1; y <= y2; y++) {
        for (int x = x1; x <= x2; x++) {
            if (fill)
                drawPixelARGB32(x, y, attr);
            else if ((x == x1 || x == x2) || (y == y1 || y == y2))
                drawPixelARGB32(x, y, attr);
        }
    }
}
```

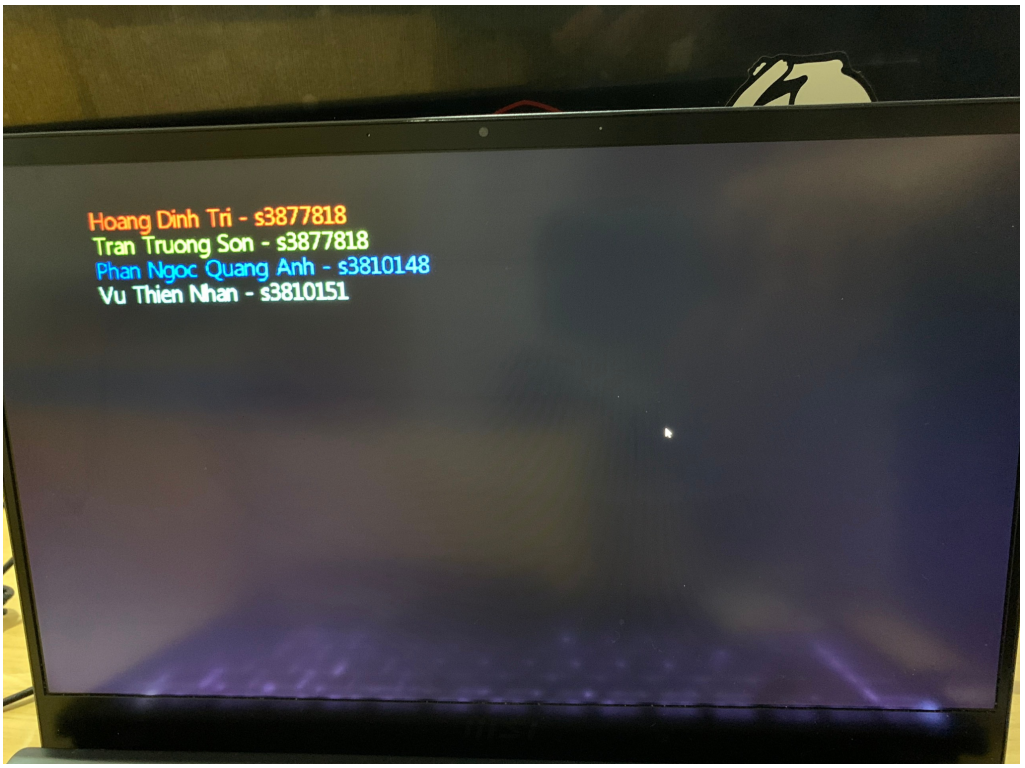
RESULT DISCUSSION

In terms of the results, our OS successfully displayed the images, video and strings with the custom font applied. Concerning the images, the user can switch between the pictures by pressing A or D and can move the current picture up or down with W and S.

The video is displayed properly on the monitor, too.



Finally, the strings with the custom font applied are displayed on the screen as expected.



III. APPLICATION DEVELOPMENT

GAMEPLAY

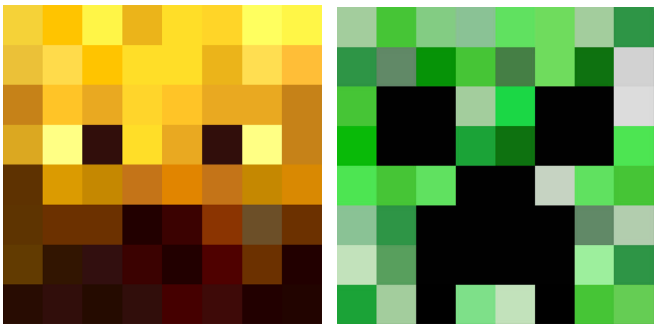
In our game development task, our team decided to build an action game named Doomsday. As the name suggests, the player will find himself in a doomsday scenario in which he is being hunted by monsters that spawn unpredictably. The primary objective of the game is to stay alive by evading the hostile creatures while also inflicting damage to them. To move around in the environment, the user will use W, A, S, D to go UP, LEFT, DOWN and RIGHT in that order. Concerning the combat mechanism, the player is capable of shooting bullets at the monsters by pressing the SPACE button. Furthermore, rotation is also enabled by pressing Q or E to rotate 45 degrees LEFT or RIGHT respectively, thus assisting our survivor in dealing with multiple enemies coming from various directions. Nevertheless, our protagonist only has 3 lives in total which diminishes every time he gets caught by the enemies. Fortunately, if a bullet hits its targets, their health bars drop gradually until the targets disappear. The game offers progress features as the player can advance to subsequent rounds whenever all of the monsters are slain. Additionally, the difficulty is increased each round corresponding to the increment of enemies. Last but not least, the game ends when the player has no life left.

GAME DESIGN

Resources

- **Display functions:** The functions that we have developed for screen displays are reused to present the game. The rotating feature of the display_image function is useful since the game requires rotation to indicate the movement directions of the game characters. The function calculates the rotation angle in radians based on the provided rotation value. Subsequently, it computes the maximum dimensions (width and height) that the rotated image can occupy by considering the image's original dimensions and the rotation angle. It calculates the center coordinates (center_x and center_y) of the image based on the top-left corner coordinates (x and y) and the image dimensions. It adjusts the start and end points of the loops (start_x, start_y, end_x, and end_y) to define a bounding box that contains the rotated image within the screen boundaries. It then iterates through the pixels within this bounding box, applies the rotation transformation to each pixel, and checks if the transformed coordinates are within the bounds of the original image.
- **Sprites:** Similar to the images in the first part of the project, the sprites of the game are arrays of unsigned integers converted from image files. The sprites include the player sprite, enemy sprites, the bullet sprite and the heart symbol sprite.

Player Sprite



Enemy sprites

Heart symbol sprite

Bullet sprite

Implementation

- **Game State Machine:** We initiated a state machine to manage the game's flow. States like "new_game," "next_lv," "play," "quit," and "win" represent different phases of the game.
- **Input Handling:** Inside the "play" state, user input is read from the UART (Universal Asynchronous Receiver-Transmitter) to control the player's movement and actions. It responds to 'w', 's', 'a', 'd' for movement, 'q' and 'e' for rotation, and space (' ') for shooting bullets.
- **Player Movement:** The "move_player" and "rotate_player" functions adjust the player's position and rotation based on the input.
- **Bullet Management:** The code keeps track of bullets in the "bullets" array and updates their positions over time. The "shoot_bullet" function is used to create new bullets.
- **Enemy Management:** Similarly, the code maintains an array of enemies and updates their positions as well behaviors.
- **Round Progression:** The code checks if the current round is over (no enemies left) and progresses to the next round or declares a win if the maximum rounds are reached.
- **User Interface:** The game's user interface elements are updated in real time, such as health, round, and score display.
- **Game Over and Win States:** In the "quit" and "win" states, the code displays messages and waits for the user to press 'Enter' before returning to the main menu.

```
26
27  void game()
28  {
29      char c = '\0';
30
31      init_enemy_sprites();
32
33      while (1) {
34          switch (state)
35          {
36          case new_game:
37          {
38              game_round = 1;
39              game_score = 0;
40              enemy_count = 0;
41              bullet_count = 0;
42              player_hits = 0;
43
44              set_screen_color(BACKGROUND_COLOR);
45              draw_string(SCREEN_WIDTH / 2 - 70, 262, "DOOMDAYS", TEXT_COLOR, 2, 0.5);
46              draw_string(SCREEN_WIDTH / 2 - 70, 300, "Press enter to play", TEXT_COLOR, 2, 0.5);
47              c = uart_getc();
48              if (c == '\n')       // check if the user pressed enter
49                  state = next_lv; // change the state to play
50
51              break;
52          }
53
```

"new_game" state

```
53
54          case next_lv:
55          {
56              state = play; // change the state to play
57
58              // Initialize level
59              player.x = SCREEN_WIDTH / 2 - player_sprite.width;
60              player.y = SCREEN_HEIGHT / 2 - player_sprite.height;
61              player.w = player_sprite.width;
62              player.h = player_sprite.height;
63              player.bbox_w = player_sprite.width;
64              player.bbox_h = player_sprite.height;
65              player.angle = 0;
66              bullet_count = 0;
67              shoot_delay_set = 0;
68              shoot_delay_cycles = 0;
69              iframes_set = 0;
70              iframes_cycles = 0;
71
72              set_screen_color(BACKGROUND_COLOR);
73
74              // Place player in the center of the screen
75              display_image(player.x, player.y, player.bbox_w, player.bbox_h, player_sprite.data, 0, player.angle);
76
77              // Update UI elements
78              update_health();
79              update_round();
80              update_score();
81
82              // Spawn enemies
83              spawn_enemies(game_round);
84
85              break;
```

"next_lv" state

```
    case play:
    {
        // Start the polling timer
        set_wait_timer(1, POLLING_TIME_MS);

        c = get_uart(); // read the user input

        switch (c) // perform different actions based on the input
        {
        case 'w': // move the player up
            move_player(0, -PLAYER_SPEED);
            break;

        case 's': // move the player down
            move_player(0, PLAYER_SPEED);
            break;

        case 'a': // move the player left
            move_player(-PLAYER_SPEED, 0);
            break;

        case 'd': // move the player right
            move_player(PLAYER_SPEED, 0);
            break;

        case 'q': // rotate player 45 degrees counter-clockwise
            rotate_player(-PLAYER_ROTATION_STEP);
            break;

        case 'e': // rotate player 45 degrees clockwise
            rotate_player(PLAYER_ROTATION_STEP);
            break;

        case ' ': // shoot a bullet
            shoot_bullet();
            break;
```

```
        case 'e': // rotate player 45 degrees clockwise
            rotate_player(PLAYER_ROTATION_STEP);
            break;

        case ' ': // shoot a bullet
            shoot_bullet();
            break;

        case 'p':         // quit the game
            state = quit; // change the state to quit
            break;

        default: // Do nothing for any other input
            break;
        }

        // Update bullets on screen
        update_bullets();

        // Update enemies on screen
        update_enemies();

        // Check if round is over
        if (enemy_count == 0) {
            if (game_round < MAX_ROUNDS) {
                game_round++;
                state = next_lv;
            } else {
                state = win;
            }
        }

        // Wait for the polling timer to expire
        set_wait_timer(0, POLLING_TIME_MS);

        break;
    }
```

“play” state

```
155         case quit:
156         {
157             set_screen_color(BACKGROUND_COLOR);
158             draw_string(SCREEN_WIDTH / 2 - 70, 262, "Game Over", TEXT_COLOR, 2, 0.5);
159             draw_string(SCREEN_WIDTH / 2 - 70, 300, "Press enter to exit", TEXT_COLOR, 2, 0.5);
160             c = uart_getc();
161             if (c == '\n') {
162                 view = '0';         // change to main menu
163                 state = new_game;   // reset game
164                 return;             // exit the function and return to the main menu
165             }
166             break;
167         }
168 
169         case win:
170         {
171             set_screen_color(BACKGROUND_COLOR);
172             draw_string(SCREEN_WIDTH / 2 - 70, 262, "You Win", TEXT_COLOR, 2, 0.5);
173             draw_string(SCREEN_WIDTH / 2 - 70, 300, "Press enter to exit", TEXT_COLOR, 2, 0.5);
174             c = uart_getc();
175             if (c == '\n') {
176                 view = '0';         // change to main menu
177                 state = new_game;   // reset game
178                 return;             // exit the function and return to the main menu
179             }
180             break;
181         }
182 
183         default:
184             set_screen_color(BACKGROUND_COLOR);
185             break;
186         }
187 
188         rand();
189     }
190 }
191
```

"quit" state

Functions implementation

- *move_player(int x_move, int y_move):*

This function is responsible for moving the player's character on the game screen. It takes two parameters, x_move and y_move, which represent the intended horizontal and vertical movement, respectively. The function calculates the new potential position of the player (player_move_x and player_move_y) based on the input. It checks whether the calculated position is within the screen boundaries and not colliding with any boundaries defined by SCREEN_WIDTH, SCREEN_HEIGHT, and TOP_PADDING_PX. If the new position is valid, it updates the player's position, erases the player's previous position on the screen, and redraws the player at the new position. This function ensures that the player cannot move outside the screen boundaries.

```
void move_player(int x_move, int y_move)
{
    int player_move_x = player.x + x_move;
    int player_move_y = player.y + y_move;

    if (player_move_x >= 0 &&
        player_move_x + player.w <= SCREEN_WIDTH &&
        player_move_y >= TOP_PADDING_PX &&
        player_move_y + player.h <= SCREEN_HEIGHT)
    {
        // Calculate the coordinates of the player's bounding box
        int player_bbox_x = player.x - (player.bbox_w - player.w) / 2;
        int player_bbox_y = player.y - (player.bbox_h - player.h) / 2;

        // Draw a rectangle to remove the player from the previous position
        drawRectARGB32(player_bbox_x, player_bbox_y, player_bbox_x + player.bbox_w, player_bbox_y + player.bbox_h, BACKGROUND_COLOR, 1);

        // Update player's position
        player.x = player_move_x;
        player.y = player_move_y;

        // Draw the player in the new position
        display_image(player.x, player.y, player.w, player.h, player_sprite.data, 0, player.angle);
    }
}

```

- *rotate_player(int rotation):*

This function is for rotating the character. It takes one parameter, rotation, which represents the amount of rotation (in degrees) to apply to the player. The function calculates the player angle after the rotation is applied. It then recalculates the dimensions of the player's bounding box (player.bbox_w and player.bbox_h) based on the player's current angle. Finally, it erases the player from its previous position, updates the player's angle, and redraws the player with the new angle. This function allows the player character to rotate while maintaining an accurate bounding box.

```
void rotate_player(int rotation)
{
    // Calculate the coordinates of the player's bounding box
    int player_bbox_x = player.x - (player.bbox_w - player.w) / 2;
    int player_bbox_y = player.y - (player.bbox_h - player.h) / 2;

    // Draw a rectangle to remove the player from the previous position
    drawRectARGB32(player_bbox_x, player_bbox_y, player_bbox_x + player.bbox_w, player_bbox_y + player.bbox_h, BACKGROUND_COLOR, 1);

    // Update player's angle
    // Map angle to 0-360 range
    player.angle += rotation;
    player.angle %= 360;
    if (player.angle < 0) player.angle += 360;

    // Convert the angle to radians
    double rad = player.angle * DEG_TO_RAD;
    double angle_sin = sin(rad);
    double angle_cos = cos(rad);

    // Calculate the dimensions of the bounding box around the player based on the player's current angle
    player.bbox_w = ceil(fabs(player.w * angle_cos) + fabs(player.h * angle_sin));
    player.bbox_h = ceil(fabs(player.w * angle_sin) + fabs(player.h * angle_cos));

    // Display rotated player sprite
    display_image(player.x, player.y, player.w, player.h, player_sprite.data, 0, player.angle);
}

```

- *shoot_bullet():*

This function controls the action of shooting bullets from the player's character. It checks if the maximum number of bullets (MAX_BULLETS) has not been exceeded and if a "shoot delay" is not set which is used to limit the fire rate of bullets. If conditions are met, it calculates the initial position and angle of the bullet based on the player's current angle. It updates the bullet's position, dimensions,

bounding box, and angle. The function then increments the bullet counter (bullet_count) to indicate that a bullet has been fired. Additionally, it sets the "shoot delay" to prevent a large firing rate. This function allows the player to shoot bullets in the direction they are facing.

```
246  void shoot_bullet()
247  {
248      if (bullet_count < MAX_BULLETS && !shoot_delay_set) {
249          int bullet_w = bullet_sprite.width;
250          int bullet_h = bullet_sprite.height;
251
252          // Convert the angle to radians
253          double rad = player.angle * DEG_TO_RAD;
254          double angle_sin = sin(rad);
255          double angle_cos = cos(rad);
256
257          // Set the bullet's initial position to the player's center
258          bullets[bullet_count].x = player.x + (player.w / 2) - (bullet_w / 2);
259          bullets[bullet_count].y = player.y;
260
261          // Set bullet dimensions
262          bullets[bullet_count].w = bullet_w;
263          bullets[bullet_count].h = bullet_h;
264
265          // Calculate the coordinates of the bullet's bounding box based on the player's current angle
266          bullets[bullet_count].bbox_w = ceil(fabs(bullet_w * angle_cos) + fabs(bullet_h * angle_sin));
267          bullets[bullet_count].bbox_h = ceil(fabs(bullet_w * angle_sin) + fabs(bullet_h * angle_cos));
268
269          // Set bullet angle
270          // Map angle to 0-360 range
271          bullets[bullet_count].angle = player.angle - 90;
272          bullets[bullet_count].angle %= 360;
273          if (bullets[bullet_count].angle < 0) bullets[bullet_count].angle += 360;
274
275          // Increment the bullet counter to fire the bullet
276          bullet_count++;
277
278          // Begin shoot delay
279          shoot_delay_set = 1;
280      }
281  }
```

- *update_bullets():*

This function manages the behavior and movement of bullets in the game. .It first checks if a "shoot delay" is set (shoot_delay_set) which was implemented to limit the firing rate of bullets. If the delay is set, it increments the "shoot delay" counter (shoot_delay_cycles). When the "shoot delay" duration (SHOOT_DELAY) is reached, the delay is reset, allowing the player to shoot again. Next, the function iterates through each active bullet in the bullets array. It calculates the new position of each bullet based on its angle and speed. The bullet's position is updated by adding the calculated horizontal (move_x) and vertical (move_y) components to its current position. The bullet's bounding box coordinates are also updated accordingly. Before drawing the bullet in its new position, the function checks if the bullet is within the screen bounds and not colliding with any boundaries (SCREEN_WIDTH, SCREEN_HEIGHT, and TOP_PADDING_PX). If the bullet is within bounds, it calculates the hitbox coordinates (bullet_hbox_x and bullet_hbox_y) to detect collisions with enemy objects. The function then checks for collisions between the bullet and enemy objects. It iterates through the active enemy objects in the enemies array. For each enemy, it calculates the collision detection area based on the enemy's position and dimensions. If a bullet collides with an enemy, it increments the enemy's hit count (enemy->hits). If the enemy's hit count exceeds a threshold (ENEMY_HP), the enemy is considered defeated. The enemy is removed from the game screen by drawing over its position with the background color. The enemy is removed from the enemies array by

replacing it with the last active enemy and decrementing the enemy_count. The player's score is increased, and the UI is updated with the new score. If a bullet hits an enemy or leaves the screen bounds, it is removed from the game. The bullet is removed from the bullets array by replacing it with the last active bullet and decrementing the bullet_count. The loop counter i is decremented to ensure that the next bullet in the array is processed. If a bullet is within bounds and hasn't hit an enemy, it is drawn in its new position using the display_image function. The angle of the bullet sprite is adjusted based on the bullet's angle.

```
283  void update_bullets()
284  {
285      // Handle shoot delay
286      if (shoot_delay_set && ++shoot_delay_cycles >= SHOOT_DELAY) {
287          shoot_delay_set = 0;
288          shoot_delay_cycles = 0;
289      }
290
291      // Update bullet positions
292      for (int i = 0; i < bullet_count; i++) {
293          struct Object* bullet = &bullets[i];
294
295          // Calculate bullet bounding box coordinates
296          int bullet_bbox_x = bullet->x - (bullet->bbox_w - bullet->w) / 2;
297          int bullet_bbox_y = bullet->y - (bullet->bbox_h - bullet->h) / 2;
298
299          // Remove bullet from the previous position
300          int bullet_start_x = player.x + (player.w / 2) - (bullet->w / 2);
301          if (bullet->y != player.y || bullet->x != bullet_start_x)
302              drawRectARGB32(bullet_bbox_x, bullet_bbox_y, bullet_bbox_x + bullet->bbox_w, bullet_bbox_y + bullet->bbox_h, BACKGROUND_COLOR, 1);
303
304          // Calculate the movement in x and y directions based on the angle and speed
305          int move_x = round(cos(bullet->angle * DEG_TO_RAD) * BULLET_SPEED);
306          int move_y = round(sin(bullet->angle * DEG_TO_RAD) * BULLET_SPEED);
307
308          // Update bullet position
309          bullet->x += move_x;
310          bullet->y += move_y;
311
312          // Update bullet bounding box coordinates
313          bullet_bbox_x += move_x;
314          bullet_bbox_y += move_y;
315
```

```
312          // Update bullet bounding box coordinates
313          bullet_bbox_x += move_x;
314          bullet_bbox_y += move_y;
315
316          // Ensure that the bullet is within screen bounds
317          if (bullet_bbox_x >= 0 && bullet_bbox_x + bullet->bbox_w <= SCREEN_WIDTH &&
318              bullet_bbox_y >= TOP_PADDING_PX && bullet_bbox_y + bullet->bbox_h <= SCREEN_HEIGHT) {
319
320              // Calculate bullet hit box coordinates
321              int min_dimension = min(bullet->w, bullet->h) / 2;
322              int bullet_hbox_x = (bullet->x + bullet->w / 2) - min_dimension;
323              int bullet_hbox_y = (bullet->y + bullet->h / 2) - min_dimension;
324
325              // Detect enemy hit
326              int hit = 0;
327              for (int j = 0; j < enemy_count; j++) {
328                  struct Enemy* enemy = &enemies[j];
329
330                  // Calculate the collision detection area for the enemy
331                  if (bullet_hbox_x >= enemy->x - enemy->w &&
332                      bullet_hbox_x <= enemy->x + enemy->w &&
333                      bullet_hbox_y >= enemy->y - enemy->h &&
334                      bullet_hbox_y <= enemy->y + enemy->h) {
335
336                      // Bullet hit an enemy
337                      enemy->hits++;
338                      hit = 1;
339
340                      // Check if enemy HP is depleted
341                      if (enemy->hits >= ENEMY_HP) {
342                          // Remove the enemy from the game
343                          drawRectARGB32(enemy->x, enemy->y - 9, enemy->x + enemy->w, enemy->y + enemy->h, BACKGROUND_COLOR, 1);
344                          enemy_count--;
345                          enemies[j] = enemies[enemy_count];
346
```

```
                    // Check if enemy HP is depleted
                    if (enemy->hits >= ENEMY_HP) {
                        // Remove the enemy from the game
                        drawRectARGB32(enemy->x, enemy->y - 9, enemy->x + enemy->w, enemy->y + enemy->h, BACKGROUND_COLOR, 1);
                        enemy_count--;
                        enemies[j] = enemies[enemy_count];

                        // Increment score and update UI
                        game_score += POINTS_PER_ENEMY;
                        update_score();
                    }

                    // Remove the bullet from active bullets
                    bullet_count--;
                    bullets[i] = bullets[bullet_count];
                    i--;
                    break; // Exit the loop since each bullet can only hit one enemy
                }
            }
            // Do not redraw the bullet if a hit is detected
            if (hit) continue;

            // drawRectARGB32(bullet_bbox_x, bullet_bbox_y, bullet_bbox_x + bullet->bbox_w, bullet_bbox_y + bullet->bbox_h, 0x0000FF00, 1);

            // Draw the bullet in the new position
            display_image(bullet->x, bullet->y, bullet->w, bullet->h, bullet_sprite.data, 1, bullet->angle + 90);

            // drawRectARGB32(bullet_hbox_x, bullet_hbox_y, bullet_hbox_x + bullet->w, bullet_hbox_y + bullet->w, 0x00FF0000, 1);
        } else {
            // Remove the bullet from active bullets
            bullet_count--;
            bullets[i] = bullets[bullet_count];
            i--;
        }
    }
}
```

- *update_enemies():*

The update_enemies function controls the enemy objects in the game, including their movement, collision detection, and interaction with the player. At first, It calculates the bounding box coordinates for the player based on his position and dimensions. This is used later for collision detection between enemies and the player. If "i-frames" are set (invincibility frames after the player is hit by an enemy), the function increments a counter (iframes_cycles) to keep track of the elapsed time. When the i-frame duration (PLAYER_IFRAMES) is reached, the i-frames are reset.Then, the function iterates through each active enemy in the enemies array. It draws a rectangle over the old position of the enemy to remove it from the previous position on the screen. It calculates the direction vector from each enemy to the player to determine the angle at which the enemy should move. Based on the calculated angle and enemy speed (ENEMY_SPEED), it calculates the new position (new_x and new_y) for the enemy. Furthermore, the function checks for collisions with other enemies. If an enemy is within a certain range (ENEMY_MIN_DISTANCE), the function adjusts the enemy's movement direction to avoid collisions. It also ensures that the new position of the enemy is within the screen bounds before updating the enemy's position. After that, the function checks for collisions between enemies and the player. If there is a collision and i-frames are not set, it indicates that the player has been hit so the player's hit count (player_hits) is incremented. The player's health is updated and displayed. If the player's hit count exceeds their maximum health (PLAYER_HP), the game state is set to "quit," indicating the end of the game. Apart from that, the function draws an enemy health bar based on the remaining health (ENEMY_HP - enemy->hits) of each enemy.

```c
void update_enemies()
{
    // Calculate player bounding box coordinates
    int player_bbox_x = player.x - (player.bbox_w - player.w) / 2;
    int player_bbox_y = player.y - (player.bbox_h - player.h) / 2;

    // Handle i-frames after the player is hit by an enemy
    if (iframes_set && ++iframes_cycles >= PLAYER_IFRAMES) {
        iframes_cycles = 0;
        iframes_set = 0;
    }

    for (int i = 0; i < enemy_count; i++) {
        struct Enemy* enemy = &enemies[i];

        // Draw a rectangle to remove the enemy from the old position
        drawRectARGB32(enemy->x, enemy->y - 9, enemy->x + enemy->w, enemy->y + enemy->h, BACKGROUND_COLOR, 1);

        // Calculate the direction vector towards the player
        double dx = (player.x + player.w / 2) - (enemy->x + enemy->w / 2);
        double dy = (player.y + player.h / 2) - (enemy->y + enemy->h / 2);

        // Calculate the angle between the enemy and the player
        double angle = atan2(dy, dx);

        // Calculate the movement in x and y directions based on the angle and speed
        int new_x = enemy->x + ceil(cos(angle) * ENEMY_SPEED);
        int new_y = enemy->y + ceil(sin(angle) * ENEMY_SPEED);
```

```c
        // Calculate the movement in x and y directions based on the angle and speed
        int new_x = enemy->x + ceil(cos(angle) * ENEMY_SPEED);
        int new_y = enemy->y + ceil(sin(angle) * ENEMY_SPEED);

        // Check for collision with obstacles (other enemies)
        for (int j = 0; j < enemy_count; j++) {
            if (j == i) continue;

            // Calculate the direction vector away from the obstacle
            double obstacle_dx = enemy->x - enemies[j].x;
            double obstacle_dy = enemy->y - enemies[j].y;

            // If the obstacle is within a certain range, adjust the movement direction away from it
            if (pow(obstacle_dx, 2) + pow(obstacle_dy, 2) < pow(ENEMY_MIN_DISTANCE, 2)) {
                double avoid_angle = atan2(obstacle_dy, obstacle_dx);
                new_x += ceil(cos(avoid_angle));
                new_y += ceil(sin(avoid_angle));
            }
        }

        // Ensure that the new position is within the screen bounds
        if (new_x >= 0 && new_x <= SCREEN_WIDTH - enemy->w && new_y >= TOP_PADDING_PX && new_y <= SCREEN_HEIGHT - enemy->h) {
            enemy->x = new_x;
            enemy->y = new_y;
        }

        // Check for player collision
        if (!iframes_set &&
            enemy->x < player_bbox_x + player.bbox_w &&
            enemy->x + enemy->w > player_bbox_x &&
            enemy->y < player_bbox_y + player.bbox_h &&
            enemy->y + enemy->h > player_bbox_y)
        {
```

```
        // Check for player collision
        if (!iframes_set &&
            enemy->x < player_bbox_x + player.bbox_w &&
            enemy->x + enemy->w > player_bbox_x &&
            enemy->y < player_bbox_y + player.bbox_h &&
            enemy->y + enemy->h > player_bbox_y)
        {
            // Enemy collided with player
            player_hits++;

            update_health();
            if (player_hits >= PLAYER_HP) {
                state = quit;
                return;
            }

            // Begin i-frames counter
            iframes_set = 1;
        }

        // Display the updated enemy position
        display_image(enemy->x, enemy->y, enemy->w, enemy->h, enemy_sprites[enemy->index].data, 0, 0);

        // Draw enemy health bar
        int health_bar_x = enemy->x + (int)(enemy->w * ((ENEMY_HP - enemy->hits) / (float)ENEMY_HP));
        drawRectARGB32(enemy->x, enemy->y - 9, health_bar_x, enemy->y - 4, HEALTH_BAR_COLOR, 1);
        drawRectARGB32(enemy->x, enemy->y - 9, enemy->x + enemy->w, enemy->y - 4, TEXT_COLOR, 0);
    }
}
```

- *update_health():*

The function calculates the number of hearts to display based on the player's remaining health (PLAYER_HP - player_hits). It clears all the heart icons by drawing rectangles with the background color over them. It then displays the appropriate number of heart icons to represent the player's remaining health.

```
// Update player's health
void update_health()
{
    // Calculate number of hearts to display
    int health = PLAYER_HP - player_hits;

    // Clear all hearts
    for (int i = 0; i < PLAYER_HP; i++) {
        int x = 12 + 40 * i;
        drawRectARGB32(x, 12, x + heart_sprite.width, 12 + heart_sprite.height, BACKGROUND_COLOR, 1);
    }

    // Draw hearts
    for (int i = 0; i < health; i++) {
        display_image(12 + 40 * i, 12, heart_sprite.width, heart_sprite.height, heart_sprite.data, 1, 0);
    }
}
```

- *update_round():*

This function is responsible for updating the display of the current round label on the screen. It takes the current game_round variable, converts it to a string, and then uses the draw_string function to display "Round:" followed by the round number on the screen.

```
// Update round text
void update_round()
{
    char round_str[10];
    int_to_str(game_round, round_str);

    int start_x = (SCREEN_WIDTH / 2) - 38;
    drawRectARGB32(start_x, 12, start_x + 120, 36, BACKGROUND_COLOR, 1);
    draw_string(start_x, 12, "Round:", TEXT_COLOR, 2, 0.5);
    draw_string(start_x + 71, 12, round_str, TEXT_COLOR, 2, 0.5);
}
```

- *update_score():*

This function updates the display of the player's score on the screen. Similar to update_round, it takes the current game_score variable, converts it to a string, and then uses the draw_string function to display the label "Score:" as well as the player's score value on the screen.

```
// Update score text
void update_score()
{
    char score_str[10];
    int_to_str(game_score, score_str);

    int start_x = (SCREEN_WIDTH / 2) + 390;
    drawRectARGB32(start_x, 12, start_x + 120, 36, BACKGROUND_COLOR, 1);
    draw_string(start_x, 12, "Score:", TEXT_COLOR, 2, 0.5);
    draw_string(start_x + 66, 12, score_str, TEXT_COLOR, 2, 0.5);
}

```

- *spawn_enemies():*

This function is responsible for spawning enemy objects in the game. It takes an integer n as an argument for the number of enemies to spawn. The function attempts to spawn up to n enemies, each with a random position along the edges of the screen (x and y coordinates). It randomly selects an enemy sprite index, width, and height. The function also checks if the newly generated enemy is too close to any existing enemy to avoid clustering. If the enemy is not too close to existing enemies, it is spawned with its attributes (position, dimensions, hits, and sprite index). The enemy_count variable is updated to reflect the number of active enemies in the game.

```
void spawn_enemies(int n)
{
    int max_enemies = min(MAX_ENEMIES, n);
    int spawned_enemies = 0;

    while (spawned_enemies < max_enemies) {
        int x, y;
        int rand_enemy_index = rand_int(0, MAX_ENEMY_SPRITES - 1);
        int enemy_w = enemy_sprites[rand_enemy_index].width;
        int enemy_h = enemy_sprites[rand_enemy_index].height;

        // Generate random positions for the enemy along the edges of the screen
        int side = rand_int(0, 3);
        switch (side)
        {
        case 0: // Spawn enemy on the top edge
            x = rand_int(0, SCREEN_WIDTH - enemy_w);
            y = TOP_PADDING_PX;
            break;

        case 1: // Spawn enemy on the right edge
            x = SCREEN_WIDTH - enemy_w;
            y = rand_int(TOP_PADDING_PX, SCREEN_HEIGHT - enemy_h);
            break;

        case 2: // Spawn enemy on the bottom edge
            x = rand_int(0, SCREEN_WIDTH - enemy_w);
            y = SCREEN_HEIGHT - enemy_h;
            break;

        default: // Spawn enemy on the left edge
            x = 0;
            y = rand_int(TOP_PADDING_PX, SCREEN_HEIGHT - enemy_h);
            break;
        }
```

```
        default: // Spawn enemy on the left edge
            x = 0;
            y = rand_int(TOP_PADDING_PX, SCREEN_HEIGHT - enemy_h);
            break;
        }

        // Check if the new enemy is too close to any existing enemy
        int too_close = 0;
        for (int i = 0; i < spawned_enemies; i++) {
            int dist_x = abs(x - enemies[i].x);
            int dist_y = abs(y - enemies[i].y);
            int min_dist_x = (enemy_w + ENEMY_MIN_DISTANCE) / 2;
            int min_dist_y = (enemy_h + ENEMY_MIN_DISTANCE) / 2;

            if (dist_x < min_dist_x && dist_y < min_dist_y) {
                too_close = 1;
                break;
            }
        }

        if (too_close) continue;

        // Spawn the enemy
        enemies[spawned_enemies].x = x;
        enemies[spawned_enemies].y = y;
        enemies[spawned_enemies].w = enemy_w;
        enemies[spawned_enemies].h = enemy_h;
        enemies[spawned_enemies].hits = 0;
        enemies[spawned_enemies].index = rand_enemy_index;
        spawned_enemies++;
    }
    enemy_count = max_enemies;
}
```

- *int_to_str(int num, char *str):*

This utility function converts an integer num to a string representation stored in the str array. It handles both positive and negative integers and performs the conversion by repeatedly dividing num by 10 and storing the digits as characters in the array. If num is negative, it is turned into a positive integer by multiplying with -1. Finally, it reverses the order of the characters in the array to obtain the correct string representation.

```
void int_to_str(int num, char *str)
{
    int i = 0;
    int is_negative = 0;

    if (num < 0) {
        is_negative = 1;
        num = -num;
    }

    do {
        str[i++] = num % 10 + '0';
        num /= 10;
    } while (num);

    if (is_negative)
        str[i++] = '-';

    for (int j = 0; j < i / 2; j++) {
        char tmp = str[j];
        str[j] = str[i - j - 1];
        str[i - j - 1] = tmp;
    }

    str[i] = '\0';
}
```

- *rand():*

This function generates a pseudo-random unsigned integer using a linear congruential generator (LCG) algorithm. The value of rand_seed is updated using the LCG formula while the function ensures that

the result is within the range of a 32-bit unsigned integer. The updated rand_seed is returned as the pseudo-random value.

```
596  unsigned int rand()
597  {
598      rand_seed = (1664525 * rand_seed + 1013904223) % 4294967296;
599      return rand_seed;
600  }
601
```

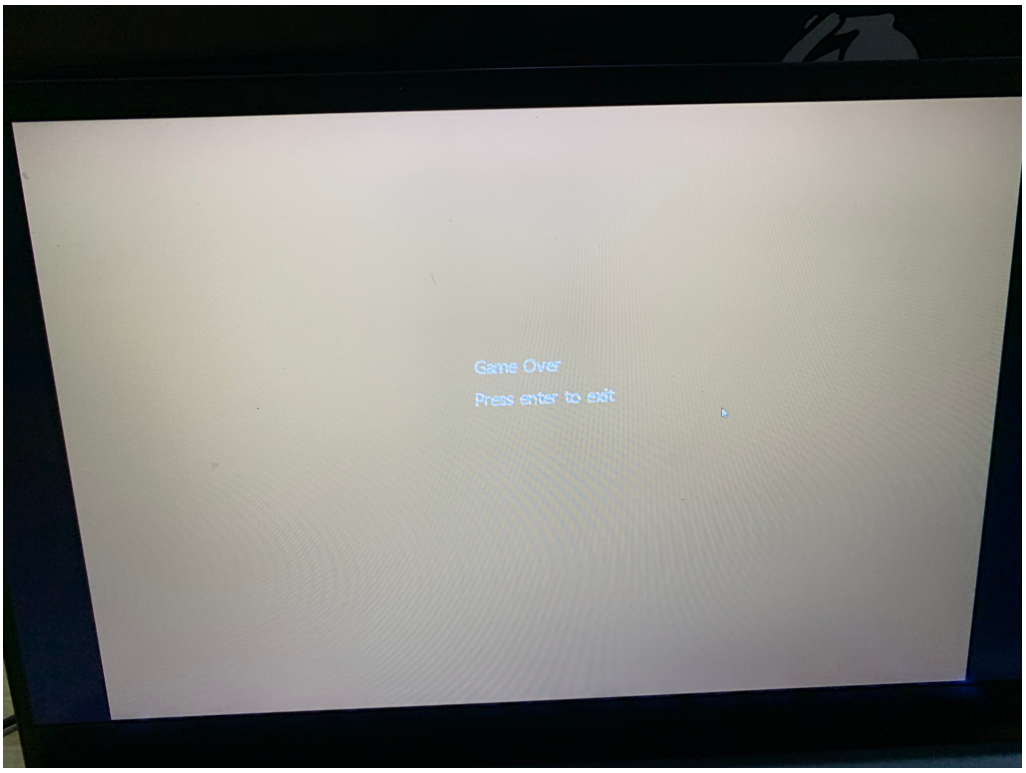
- *rand_int(int min, int max)*

This utility function generates a pseudo-random integer within a specified range. It takes two arguments: min (the minimum value of the range) and max (the maximum value of the range). It refers to the rand function earlier to generate a pseudo-random value and scales it to fit within the specified range. The resulting random integer within the specified range is returned.

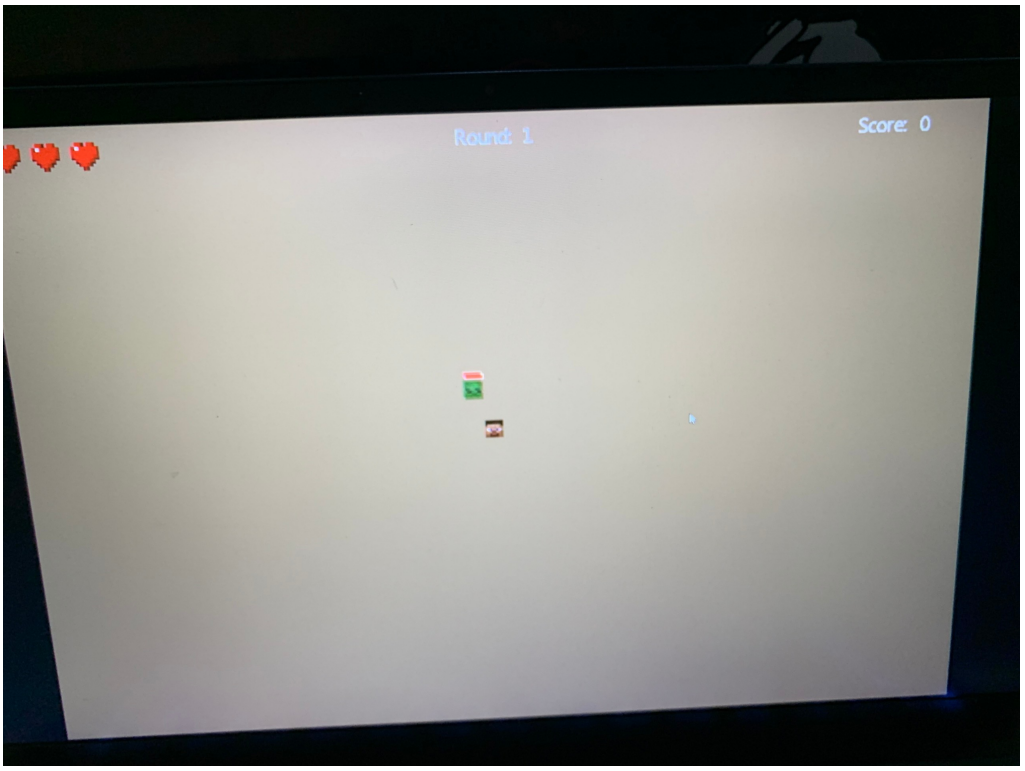
```
602  int rand_int(int min, int max)
603  {
604      return (rand() % (max - min + 1)) + min;
605  }
606
```

RESULT DISCUSSION

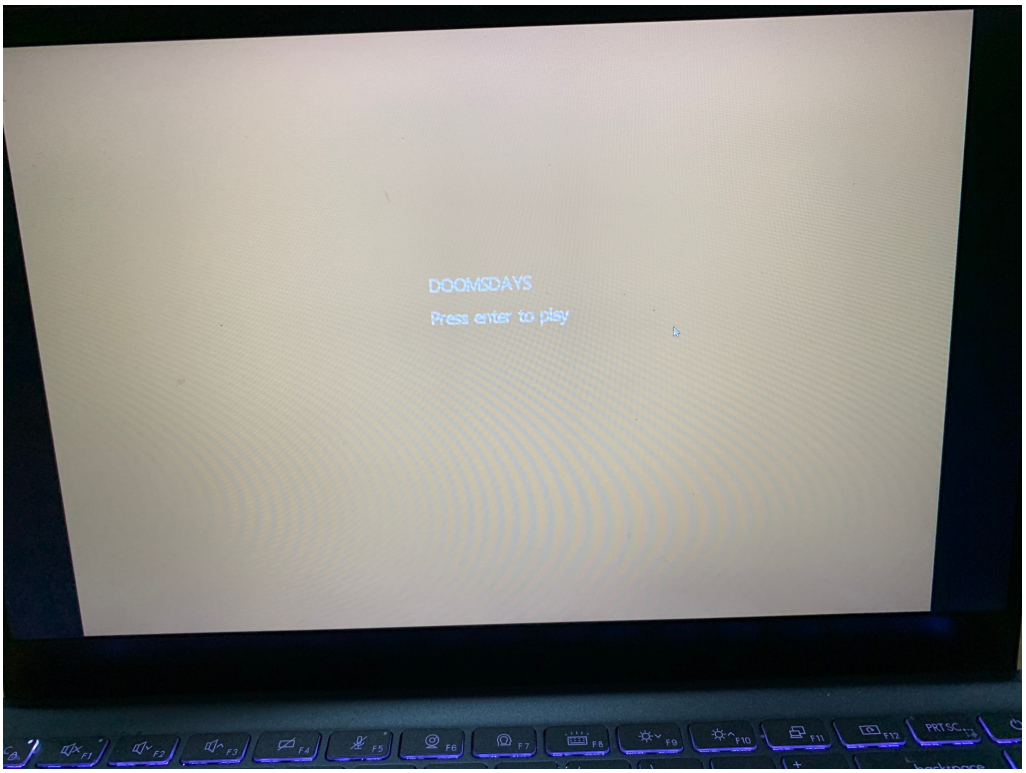
The game ran as intended and the controls worked smoothly.



Game Welcome screen



Gameplay screen



Game over screen

We have updated the Game Welcome screen to display instructions to the user as follows.

IV. CONCLUSION

In conclusion, our group has developed a system that satisfies the requirements of the project. This includes a bare metal OS that is capable of displaying images, videos and custom fonts. Additionally, we have also successfully built a small 2D game that has dynamic interaction among the objects as well as captivating visual presentation. These tasks necessitate each member to have a firm grasp of the embedded subjects that have been taught in this course which are UART communication, mailbox interface, framebuffer, ... Moreover, it is crucial that we understand the features of a Raspberry Pi board by referring to the board documentation so as to write embedded programs for it.

Throughout the project all of the members have acquired essential skills and knowledge in the field of embedded systems and software engineering. Firstly, we delved into the intricacies of interfacing with hardware components, which is fundamental in embedded systems development. This involved understanding the hardware specifications, communication protocols, and ensuring interaction between software and hardware components. The core of this project revolved around software coding and software project management. Hence, we have practiced our programming skills extensively in the C language, which is widely used in embedded systems due to its efficiency and low-level control.

Another crucial aspect of this project is teamwork and communication. We collaborated with each other, learned to distribute tasks efficiently, and coordinated efforts to achieve project milestones. Effective teamwork not only ensures project success but also promotes a positive work environment.

V. REFERENCES (USE IEEE STYLES)

J. Loenen. “image2cpp”. github.io Date accessed (Sep. 24, 2023) [Online]. Available: https://javl.github.io/image2cpp/

P. Levis and P. Hanrahan. “Lab 6: Drawing into the Framebuffer”. CS107e Winter 2023. Date accessed (Sep. 25, 2023). [Online]. Available: http://cs107e.github.io/labs/lab6/

Rpi4-osdev. “Writing a “bare metal” operating system for Raspberry Pi 4 (Part 5)”. rpi4os.com. Date accessed (Sep. 26, 2023). [Online]. Available: https://www.rpi4os.com/part5-framebuffer/