Game Report - Super Mario: The Adventure Quest

Problem Description

Super Mario: The Adventure Quest is a MarioBros-style platformer where players navigate through aseries of obstacles, collect coins, and avoid enemies and hazards. The objective is to reach the endpoint of each level after collecting all available coins, represented by a flag or specific location. The game includes standard platformer mechanics, such as moving left and right, jumping, and interacting with objects in the environment.

The game world is filled with classic platformer elements, including breakable and unbreakable blocks, moving platforms, and hidden passages, adding layers of complexity and exploration to each level. Players must master timing and precision to dodge enemies and hazards, which are strategically placed to challenge their progress. Enemies have unique behaviors, such as patrolling specific areas, following set paths, or actively pursuing the player when in close range. Hazards like fire, spikes, or pits add an extra layer of difficulty, often requiring the player to pause and observe patterns before advancing.

Features of Game

1) Collecting All Coins

The primary objective of the game is to collect all coins present in each level. Coins are strategically scattered throughout the environment, often guarded by non-player character (NPC) enemies or positioned in challenging locations that require precise jumping and movement to access. Successfully gathering every coin is essential for player progression and completion of the level.

2) Navigating NPC Enemies

Players must also navigate and overcome various NPC enemies that exhibit distinct behaviors. These enemies may patrol specific areas, chase the player, or block pathways, each requiring different strategies to either avoid or engage. Mastering the interactions with these enemies is crucial for advancing through the levels and achieving success.

3) Environmental Obstacles

The game features a variety of environmental obstacles that challenge the player’s platforming skills. Players must maneuver around platforms, walls, pits, and spike traps while avoiding harmful hazards such as fire, spikes, and falling rocks. Success in this aspect of the game hinges on careful timing and precise movements to dodge these dangers effectively.

4) Reaching the Level Endpoint

Ultimately, the game culminates in the player reaching the endpoint of each level. After successfully collecting all coins, navigating obstacles, and overcoming hazards, the player must arrive at this endpoint, which could be represented by a flag, door, or designated location. The completion of this objective signifies the successful conclusion of the level.

Classes Implemented

1. Player Class

The Player class is responsible for managing the player's behavior and interactions within the game world. It handles the player's movement, gravity effects, collisions, and animation states (e.g., jumping, walking, idle). This class is central to the game, as it directly represents the character the player controls.

•Movement: The class controls the player's movement left and right using keyboard input. It ensures that the player can only move within the boundaries of the game environment.

•Jumping: The player can jump by applying an upward velocity. The class handles jump initiation and gravity effects, ensuring the player moves up and falls back down.

•Collision Detection: The player detects and responds to collisions with other objects, such as platforms or walls. If the player lands on a platform, it stops falling and can walk on it.

•Animation: Based on the player’s state (e.g., walking, jumping), the class updates the sprite to show the correct animation.

Health and Coin Collection: The class tracks the player's health and coin count. The player's health decreases if they hit an obstacle or enemy, and coins are collected when the player touches them.

- Variables:
    - o COLOR: Color of the player for debugging or other purposes.
    - o GRAVITY: Gravity constant for calculating fall speed.
    - o SPRITES: Holds the loaded animations for the player.
    - o ANIMATION_DELAY: Delay between frames in animation.
    - o rect: Position and dimensions of the player.
    - o x_vel, y_vel: Horizontal and vertical velocities.
    - o mask: Used for collision detection.
    - o direction: Direction the player is facing (left or right).
    - o animation_count: Counter for tracking animation frames.
    - o fall_count: Tracks the duration of falling.
    - o jump_count: Tracks the number of jumps.
    - o hit: Boolean indicating if the player is hit.
    - o hit_count: Counter for how long the player remains in the hit state.
    - o coins: Count of coins collected.
    - o health: Player's health points.

- Methods:
    - o jump(): Initiates a jump, updating the vertical velocity and resetting fall counters.
    - o make_hit(hit_type): Reduces health based on the type of hit (fire or other).
    - o move(dx, dy): Moves the player by dx and dy.
    - o move_left(vel), move_right(vel): Moves the player left or right, changing the direction and resetting animation if necessary.
    - o loop(fps): Updates the player’s position, checks for hits, and animates per frame.
    - o update(): Updates the player’s collision mask and position.
    - o landed(): Resets fall and jump counters when the player lands.
    - o hit_head(): Inverts vertical velocity if the player hits a ceiling.
    - o update_sprite(): Updates the current sprite based on the player's state (e.g., running, jumping).
    - o all_coins_collected(coins): Checks if all coins have been collected.
    - o draw(win, offset_x): Draws the player on the screen at the current position with an

2. Object Class

The Object class serves as the foundation for all in-game objects like platforms, blocks, coins, etc. It provides a common structure for defining the properties and behaviors of objects in the game world.
•Positioning: The class tracks an object’s position using a rectangle (rect), which defines where the object is located on the screen and its size.
•Rendering: Objects can be drawn to the screen using an image or shape associated with them.

Collision Detection: The rect of each object is used to detect collisions with the player or other objects in the game.
- Variables:
    - o rect: Position and dimensions.
    - o image: Surface to represent the object visually.
    - o width, height: Dimensions of the object.
    - o name: Identifier for the object type.

- Methods:
    - o draw(win, offset_x): Draws the object on the screen with a horizontal offset.

3. Block Class

The Coin class is a specialized version of the Object class, designed to represent collectible coins within the game. When the player collects a coin, their coin count increases.

•Rendering: Coins are drawn on the screen in their respective positions.

•Collection: The class detects when the player touches or collects the coin. Upon collection, it adds to the player's coin total.

•Display: The coin's visual representation may be an image that shows up as a coin icon or shape within the game world.

- Variables:
    - o Inherits variables from Object.
    - o image: Surface with block texture from the get_block() function.
    - o mask: Mask for collision detection.
- Methods:
    - o Inherits draw() from Object.

4. Fire Class

Represents a fire hazard that can harm the player.

- Variables:
    - o ANIMATION_DELAY: Delay between frames in animation.
    - o fire: Contains fire animation frames.
    - o image: Current frame of the fire animation.
    - o mask: Mask for collision detection.
    - o animation_count: Counter to track frames for animation.
    - o animation_name: Indicates whether fire is "on" or "off".
- Methods:
    - o on(): Turns the fire animation to the "on" state.
    - o off(): Turns the fire animation to the "off" state.
    - o loop(): Updates the fire animation based on animation_name, resets animation_count if necessary.

5. Coins Class

Represents collectible coins in the game.

- Variables:
    - o ANIMATION_DELAY: Delay between frames for coin animation.
    - o rect: Position and dimensions of the coin.
    - o coin_sprites: List of coin animation frames.
    - o image: Current image for the coin.
    - o mask: Mask for collision detection.
    - o animation_count: Counter for tracking animation frames.
    - o collected: Boolean indicating if the coin is collected.
- Methods:
    - o collect(): Sets the coin as collected.
    - o loop(): Animates the coin if it hasn't been collected, resets animation_count if necessary.
    - o draw(win, offset_x): Draws the coin on the screen if it hasn't been collected.

6. Algorithms:

Local Search:

```
Input: current position (node), target position (goal_state), speed, player,
obstacles (objects)
Output: Updated position

function move_local(node, goal_state, speed, objects):
    moves ← {
        "right": (node.x + speed, node.y),
        "left": (node.x - speed, node.y),
        "down": (node.x, node.y + speed),
        "up": (node.x, node.y - speed)
    }

    best_move ← None
    current_heuristic ← ∞

    for each move, position in moves do
        if no collision at position with objects then
            heuristic ← calculate_heuristic(position, goal_state)

            if heuristic < current_heuristic then
                best_move ← position
                current_heuristic ← heuristic

    if best_move is not None then
        update node.x, node.y to best_move

    return node
```

A* Path Finding

```
Input: Start position, goal position, movement step, obstacle list (objects)
Output: Path from start to goal

function heuristic(pos1, pos2):
    return abs(pos1.x - pos2.x) + abs(pos1.y - pos2.y)


function find_path(start, goal_state, objects):
    frontier ← PriorityQueue with (0, start, start)
    explored ← empty set
    path ← empty

    while frontier is not empty do
        g_cost, node, path ← frontier.pop_with_lowest_cost()
        add node to explored
        neighbours ← [
            (node.x + step, node.y),
            (node.x - step, node.y),
            (node.x, node.y + step),
            (node.x, node.y - step)
        ]

        for each neighbour in neighbours do
            if neighbour equals goal_state then
                add neighbour to path
                return path

            else if neighbour in explored or collision(neighbour, objects) or
is_out_of_bounds(neighbour) then
                continue

            else
                score ← g_cost + heuristic(node, goal_state)
                new_path ← path + (node)
                add (score, neighbour, new_path) to frontier

    return failure
```

Output:

Image 1 : Start Page

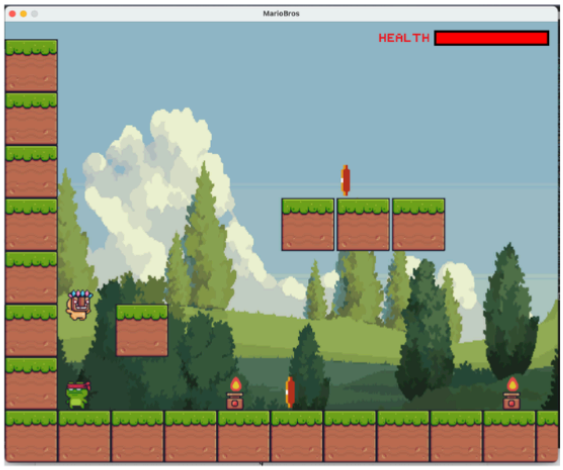

Image 2 : Initial Page - This is shown once the "START" button is clicked

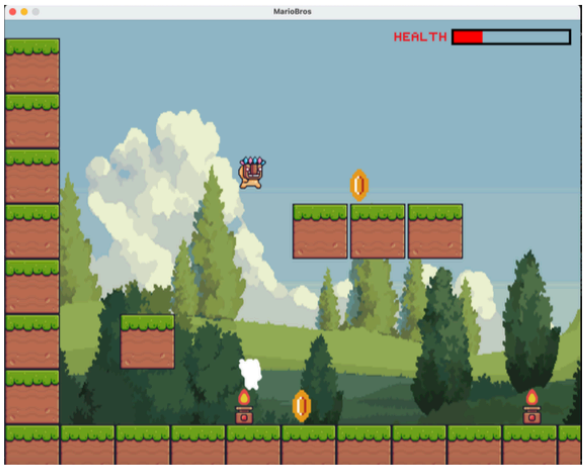

Image 3 : If Player collides with fire, the player turns to white and health meter decreases

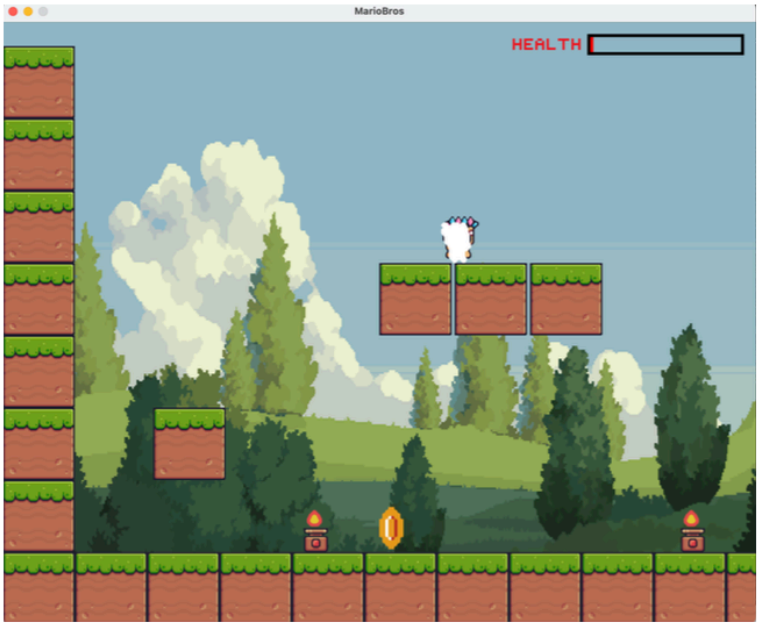

Image 4 : If Enemy touches the player, the player turns white and health meter decreases in a faster rate

Image 5 : If player misses to jump when there's a pit, the game ends

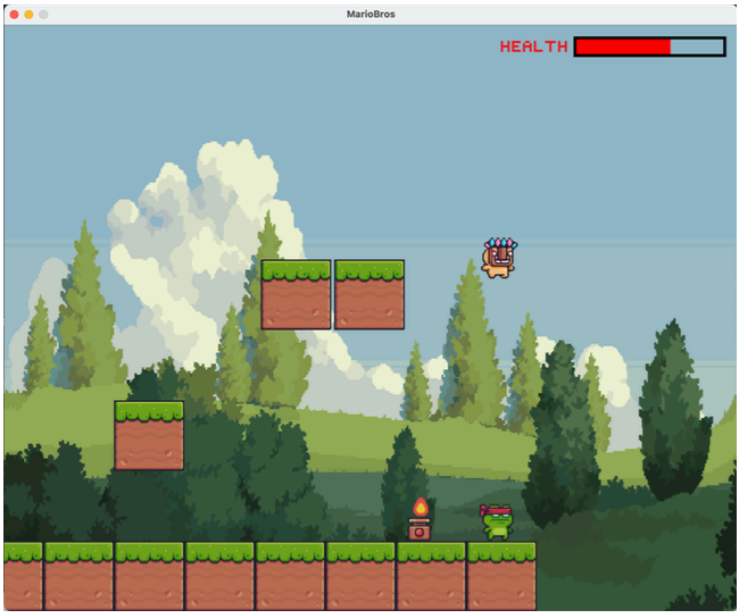

Image 6 : When player reaches the last block of floor after collecting all the coins, then the End page is displayed. If not, the player has missed coins to collect and has to collect all of them for the quest to get over

---

Image 7 : End Page - This is displayed only when all the coins are collected and the player should reach the last block of the floor.
