Processing Game: Code Analysis and Explanation

Introduction

This document provides a detailed explanation of the core functions, classes, and algorithms used in the Processing game. It highlights the purpose, behavior, and implementation details of each key component, enabling a deeper understanding of the code structure and functionality.

Key Variables

The following variables serve as the foundation of the game, controlling its state and behavior:

- **gameScreen**: Tracks the current screen (e.g., menu, gameplay, game over).
- **difficulty**: Stores the difficulty level chosen by the player (easy, medium, hard).
- **backgroundX** & **backgroundImage**: Manage the background scrolling effect.
- **score**: Tracks the player's current score.
- **walls**: An array holding the active wall obstacles.
- **ball and racket**: Represent the main game elements the player interacts with.
- **currentWeather**: Tracks the active weather effect (none, rain, wind, or both).
- **healthBoost**: Manages health-boosting items that the player can collect.
- **gravity, airfriction, friction**: Physical constants controlling ball movement.

Class Definitions

Ball Class

- Represents the player-controlled ball.
- **Key Functions:**
    - **draw**: Renders the ball on the screen.
    - **decreaseHealth**: Reduces the ball's health when it collides with obstacles. Triggers a game over if health reaches zero.
    - **applyGravity**: Adds a downward force to simulate gravity.
    - **keepInScreen**: Keeps the ball within screen boundaries.
    - **bounce**: Calculates the ball's response when it hits a surface (e.g., walls, paddle).
    - **applyWeatherEffects**: Adjusts the ball's movement based on current weather conditions.

Racket Class

- Represents the paddle controlled by the player's mouse.
- **Key Functions:**
    - **draw**: Renders the paddle on the screen.
    - **watchBounce**: Detects collisions between the paddle and the ball, and adjusts the ball's trajectory.

Wall Class

- Represents the obstacles in the game.
- **Key Functions:**
    - **draw**: Renders the walls with a gap for the ball to pass through.
    - **move**: Moves the walls horizontally and adjusts their vertical position dynamically.
    - **collides**: Checks for collisions between the wall and the ball.

HealthBoost Class

- Represents collectible health items that restore the ball's health.
- **Key Functions:**
    - **draw**: Renders the health boost on the screen.
    - **move**: Moves the health boost toward the player.
    - **checkCollection**: Detects when the ball collects the health boost and increases health accordingly.

BackgroundImage Class

- Manages the scrolling background visuals.
- **Key Functions:**
    - **setImage**: Sets the background image based on the selected difficulty.
    - **scroll**: Creates a seamless scrolling effect by looping the background image.
    -
    - `scroll`: Creates a seamless scrolling effect by looping the background image.

WEATHER EFFECTS

- **createRain and drawRain**: Generate and animate raindrops for the "rain" weather effect.
- **createWindParticles and drawWindParticles**: Add visual cues for wind effects, including directional particles.
- **updateWind**: Changes the wind force and direction at regular intervals.
- **applyWindEffect**: Applies wind force to the ball.

GAMEPLAY LOGIC

Difficulty Settings

- **setDifficultySettings**: Adjusts wall gap height and speed based on the selected difficulty.

Walls and Health Boosts

- **wallAdder**: Adds new walls at intervals and occasionally places a health boost within gaps.

Health Bar

- **drawHealthBar**: Visualizes the ball's current health using a color-coded bar.

GAME SCREENS

Main Screens

- **selectDifficulty**: Displays the menu for selecting the difficulty level.
- **gameplayScreen**: Handles the main gameplay loop, including drawing and updating all game elements.
- **gameOverScreen**: Displays the "Game Over" screen and score.
- **displayTopScoresScreen**: Shows the top scores for the selected difficulty.

Input and Validation

- **createInputForm**: Displays input fields for username and PIN when the player qualifies for the leaderboard.
- **validateInputs**: Validates the player's username and PIN against defined rules.
- **submitTopScore**: Submits the player's score to the server and transitions to the appropriate screen.

TOP SCORES AND API INTEGRATION

- **fetchTopScores**: Retrieves the top scores from the API.
- **submitTopScore**: Sends the player's username, PIN, and score to the server.

UTILITY FUNCTIONS

- **resetGame**: Resets all variables and elements to their initial states.
- **updateGame**: Handles periodic updates like adding walls, updating weather, and creating health boosts.

Ball Class

Represents the central gameplay element, with algorithms designed for realistic physics and dynamic interactions.

Key Algorithms:

1. **Gravity Application:**

```
applyGravity() {
  this.speedVert += gravity;
  this.y += this.speedVert;
  this.speedVert -= this.speedVert * airfriction;
}

keepInScreen() {
if (this.y + this.size / 2 > height) {
  this.bounce(height - this.size / 2, "vertical"); // Bottom bounce
}
if (this.y - this.size / 2 < 0) {
  this.bounce(this.size / 2, "vertical"); // Top bounce
}
if (this.x - this.size / 2 < 0) {
  this.bounce(this.size / 2, "horizontal"); // Left bounce
}
if (this.x + this.size / 2 > width) {
  this.bounce(width - this.size / 2, "horizontal"); // Right bounce
}
}
```

Explanation: This simulates real-world gravity by increasing the ball's vertical velocity (speedVert) over time. The air resistance (airfriction) ensures the velocity doesn't grow infinitely, mimicking how air slows objects.

2. **Collision with Screen Edges:**

```
keepInScreen() {
  if (this.y + this.size / 2 > height) {
    this.bounce(height - this.size / 2, "vertical"); // Bottom bounce
  }
  if (this.y - this.size / 2 < 0) {
    this.bounce(this.size / 2, "vertical"); // Top bounce
  }
  if (this.x - this.size / 2 < 0) {
    this.bounce(this.size / 2, "horizontal"); // Left bounce
  }
  if (this.x + this.size / 2 > width) {
    this.bounce(width - this.size / 2, "horizontal"); // Right bounce
  }
}
```

Explanation: The ball is checked against screen boundaries. If it exceeds any edge, it "bounces" by reversing the velocity and applying a damping effect (explained below).

3. **Bounce Calculation:**

```
bounce(surface, direction) {
const maxSpeed = 10; // Limit maximum bounce speed
const minSpeed = 0.5; // Ignore very small movements to stop jittering
let multiplier = 1.02; // Default multiplier for difficulty

if (difficulty === "medium") {
  multiplier = 1.10;
} else if (difficulty === "hard") {
  multiplier = 1.15;
}

if (direction === "vertical") {
  this.y = surface + (this.size / 2) * (this.speedVert > 0 ? -1 : 1); // Correct position
  this.speedVert *= -multiplier * (1 - friction); // Reverse and apply friction
  this.speedVert = constrain(this.speedVert, -maxSpeed, maxSpeed); // Clamp speed
  if (Math.abs(this.speedVert) < minSpeed) {
    this.speedVert = 0; // Stop very small movements
  }
} else if (direction === "horizontal") {
  this.x = surface + (this.size / 2) * (this.speedHorizon > 0 ? -1 : 1); // Correct position
  this.speedHorizon *= -multiplier * (1 - friction); // Reverse and apply friction
  this.speedHorizon = constrain(this.speedHorizon, -maxSpeed, maxSpeed); // Clamp speed
  if (Math.abs(this.speedHorizon) < minSpeed) {
    this.speedHorizon = 0; // Stop very small movements
  }
}
```

Explanation: The bounce applies a multiplier based on the difficulty, making the game harder at higher levels. The friction ensures the bounce loses energy over time, preventing endless motion. The speed clamping keeps the game visually coherent.

4. **Weather Effects:**

```
applyWeatherEffects() {
  if (currentWeather === "none"){
    return; // Skip if no weather effect
   }
  // Apply rain effects
  if (currentWeather === "rain" || currentWeather === "both") {
    this.speedVert *= 0.98; // Dampen vertical speed
    this.speedHorizon *= 0.98; // Dampen horizontal speed
  }
  // Apply wind effects
  if (currentWeather === "wind" || currentWeather === "both") {
    this.speedHorizon += windForce; // Push ball horizontally
  }
 }
}
```

Explanation: Rain dampens both vertical and horizontal speeds, mimicking increased drag. Wind modifies horizontal velocity directly, creating directional movement.

Racket Class

The paddle serves as a secondary interactive element. Its algorithms focus on detecting collisions and redirecting the ball.

Key Algorithms:

1. **Collision with Ball:**

```
watchBounce(ball) {
  const overhead = mouseY - pmouseY;
  if (ball.x + ball.size / 2 > mouseX - this.width / 2 &&
      ball.x - ball.size / 2 < mouseX + this.width / 2) {
    if (dist(ball.x, ball.y, ball.x, mouseY) <= ball.size / 2 + abs(overhead)) {

      ball.bounce(mouseY - ball.size / 2, "vertical"); // Bounce vertically off the racket
      ball.speedHorizon = (ball.x - mouseX) / 10;

      if (overhead < 0) {

        ball.y += overhead / 2;
        ball.speedVert += overhead / 2;

      }
```

Explanation: Collision detection uses the distance formula to determine if the ball is close enough to the paddle. The ball's horizontal speed is influenced by its relative position to the paddle, introducing spin-like behavior. If the paddle moves upward during collision, it adds additional vertical velocity to the ball, simulating an "active hit."

Wall Class

Walls introduce obstacles that the player must navigate. The algorithms ensure dynamic motion and collision detection.

Key Algorithms:

1. **Wall Movement:**

```
move() {
  this.x -= wallSpeed;
  this.y += this.moveDirection * 2;
  if (this.y <= 0 || this.y + this.height >= height){
    this.moveDirection *= -1;
  }
}
```

Explanation: Walls move left at a constant speed (wallSpeed) and oscillate vertically. The oscillation creates dynamic, harder-to-predict gaps.

2. **Collision Detection:**

```
collides(ball) {
  return (
    ball.x + ball.size / 2 > this.x &&
    ball.x - ball.size / 2 < this.x + this.width &&
    (ball.y - ball.size / 2 < this.y || ball.y + ball.size / 2 > this.y + this.height)
  );
}
```

Explanation: The algorithm checks if the ball overlaps the wall horizontally while being outside the gap. This simple but efficient check ensures quick collision detection.

HealthBoost Class

Health boosts restore the player's health, adding an element of recovery.

Key Algorithms:

1. **Collection Logic:**

```
// Check if the ball has collected the health boost
checkCollection(ball) {
  // Check if the health boost is close enough to the ball to be "collected"
  if (dist(this.x, this.y, ball.x, ball.y) < (this.size + ball.size) / 2) {
    this.collected = true; // Mark as collected
    let healthIncrease = maxHealth * 0.8; // 80% of maxHealth
    ball.health = min(ball.health + healthIncrease, maxHealth); // Increase the ball's health, capped at maxHealth
    //console.log("Health boost collected. New health:", ball.health);
  }
 }
}
```

Explanation: Similar to paddle collision, the **distance formula** determines if the ball collects the boost. If collected, the health is increased by 80% of the maximum, capped to ensure no overflow.

BackgroundImage Class

Handles the visual scrolling effect to give the illusion of motion.

Key Algorithms:

1. **Scrolling:**

```
scroll(speed = 2) {
  // Move the background to the left by the specified speed
  this.x -= speed;
  if (this.x <= -width) {
    this.x = 0; // Reset position for seamless scroll
  }

  // Draw two copies of the image side-by-side for a seamless scroll effect
  image(this.currentImage, this.x, 0, width, height);
  image(this.currentImage, this.x + width, 0, width, height);
}
```

Explanation: The background shifts left by a fixed speed. When it scrolls past the screen width, it resets to the starting position, ensuring a continuous flow.

Entry Form for Top Scores

This form appears when the player qualifies for the Top-5 leaderboard, allowing them to enter their username and PIN. It includes validation for input correctness.

Key Features and Algorithms:

1. **Dynamic Form Creation:**
    - The **createInputForm** function dynamically generates HTML elements for the form. These elements include:
        - **Input fields** for username and PIN.
        - **Error messages** for invalid input.
        - **Reminder message** to retain username and PIN.
    - Styling ensures that the form integrates seamlessly with the game screen.
2. **Validation Logic:**
    - **Username**: Must be 8-15 alphanumeric characters.
    - **PIN**: Must be a 6-digit number and cannot be common values like 123456, 000000, or 111111.
    - **Validation** is handled by the validateInputs function, which updates the error messages dynamically.

3. **Debounced Input Handling:**

    - Event listeners ensure input is sanitized and only valid characters are accepted. For example:

```
usernameInput.addEventListener("input", (e) => {
  playerName = e.target.value.replace(/[^a-zA-Z0-9]/g, ""); // Sanitize input
  //console.log("Username updated:", playerName);
});

pinInput.addEventListener("input", (e) => {
  playerPin = e.target.value.replace(/[^0-9]/g, ""); // Sanitize input
  //console.log("PIN updated:", playerPin);
});
```

Score Display Screen

Once a game ends, the **Top Score Display Screen** lists the top scores for the current difficulty level.

Key Features and Algorithms:

1. **Dynamic Score Rendering:**

    - Scores are fetched from the server and displayed in a visually organized table, with rank, username, and score.

```
topScores.forEach((score, index) => {
  textAlign(LEFT);

  // Truncate long usernames
  const maxUsernameLength = 12;
  const displayUsername =
    score.username.length > maxUsernameLength
      ? score.username.substring(0, maxUsernameLength - 3) + "..."
      : score.username;

  // Render rank, username, and score
  text(`${index + 1}.`, rankX, startY + index * lineHeight); // Rank
  text(displayUsername, nameX, startY + index * lineHeight); // Username
  text(`${score.score}`, scoreX, startY + index * lineHeight); // Score
});
```

How it Maintains Alignment

Fixed Truncation:

- If a username exceeds 12 characters, it will be shortened to the first 9 characters followed by ..., ensuring it fits within the allotted space.
- This truncation does not affect the overall alignment because the length is predictable.

Left Alignment:

- Since the usernames are aligned using textAlign(LEFT) and positioned consistently (nameX), truncating longer usernames will not disturb the spacing or alignment of other elements (like scores or ranks).

Score and Rank Positions:

- The positions (rankX, nameX, scoreX) are fixed and independent of the username length, ensuring they stay aligned even if usernames are truncated.

2. **Responsive Updates:**
    - The fetchTopScores function ensures the latest scores are displayed. If the player qualifies, they are redirected to the Entry Form.
3. **Fallback for No Scores:**
    - If no scores are available, a placeholder message is displayed:

```
// Check if `topScores` is available
if (!topScores || topScores.length === 0) {
  textSize(20);
  fill(255, 255, 0); // Yellow text
  textStyle(BOLD);
  text("No scores available.", width / 2, height / 2);
  return;
}
```

Network Error Screen

This screen appears when API calls (fetch or submission) fail, indicating a network issue.

Key Features:

1. **Error Messaging:**
    - Displays an error message explaining the failure, such as

```
textAlign(CENTER, CENTER);

// Display the error message
fill(255, 0, 0); // Red text for high visibility
textSize(24);
text("Connection Error", width / 2, height / 2 - 100);
```

2. **Retry Option:**

- Offers instructions for the player to restart or try again

```
// Display the user's score
fill(236, 240, 241); // Light gray text
textSize(20);
text(`Your Score: ${score}`, width / 2, height / 2);

// Display reset instructions
textSize(16);
text("Press R to Return to the Main Menu", width / 2, height / 2 + 50);
```

API Fetch and Submission

Handles leaderboard integration, including retrieving and submitting player scores.

Fetching Top Scores:

1. **fetchTopScores Function:**

   Prevents redundant API calls if a previous fetch has already failed. This saves resources and avoids overwhelming the server. Sends a GET request to the API. If the response status is not OK (response.ok), it throws an error with a descriptive message. This ensures any HTTP errors (like 404 or 500) are caught early. Extracts the data field from the response. If data is not an array (as expected for the leaderboard), it throws an error. This safeguards against unexpected API changes or malformed responses. Updates the local topScores array with the retrieved data, ensuring the UI reflects the latest scores. Determines if the player's score qualifies for the Top-5: If there are fewer than 5 scores, the player automatically qualifies. If the player's score is higher than the lowest score on the leaderboard, they qualify. If any error occurs during the fetch or processing, the fetchFailed flag is set to prevent further attempts. The error is then propagated for higher-level handling (e.g., showing a network error screen).

Submitting a Score:

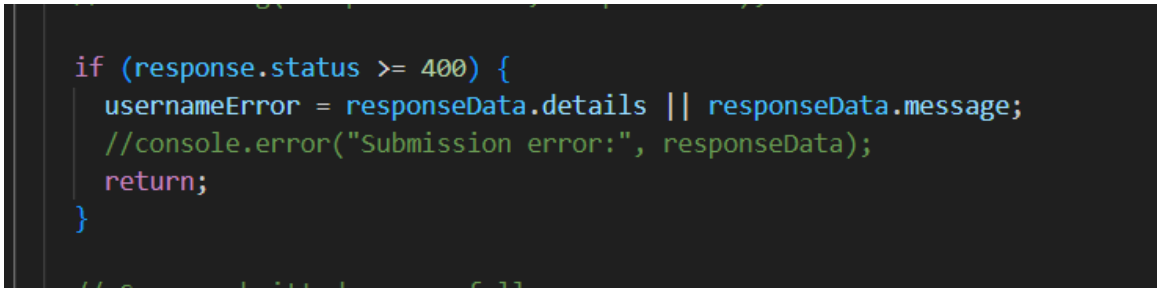
1. **submitTopScore Function:**

   - Sends a POST request to the API with the player's username, PIN, score, and difficulty level.

```
const data = { username: playerName, pin: playerPin, score, difficulty_level: difficulty };

//console.log("Submitting Score Data:", data);

try {
  const response = await fetch(url, {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(data),
  });

  const responseData = await response.json();
```

2. **Validation and Error Handling:**

- If the server rejects the score (e.g., invalid input or duplicate username), an error message is displayed:



```
if (response.status >= 400) {
  usernameError = responseData.details || responseData.message;
  //console.error("Submission error:", responseData);
  return;
}
```

3. **Successful Submission:**

- Redirects to the **Top Score Display Screen** to show the updated leaderboard.

Conclusion

This detailed breakdown of the game's classes and algorithms offers a comprehensive understanding of its implementation. Each component is designed to work together seamlessly, creating a dynamic and engaging gaming experience.