Player Movements and Event Handling in the BattleField Game

Introduction

In this chapter, we delve into the key elements of the game logic that manage player movements within the `BattleField` game. These mechanics are pivotal to how players interact with the game world, especially during gameplay when they are trying to navigate, avoid obstacles, and compete with each other. This includes an exploration of how the game responds to player input via keyboard events and how those inputs translate to movements on the screen.

We will break down the essential algorithms that govern player movement, focusing on the key event handling, the control structures for player actions, and the logic to detect collisions with obstacles or other players. Understanding these mechanisms is crucial for enhancing gameplay interaction and ensuring fluid and intuitive controls.

1. Event Handling and Key Listeners

The game provides a grid where two players can move, each with a specific key binding for their controls. Handling these key presses efficiently is essential for providing a responsive and enjoyable gameplay experience. In Java, this can be accomplished using key event listeners, which are components of the Java AWT event-handling framework.

KeyEvent and KeyAdapter

The `KeyEvent` class in Java provides constant values for the different keys on the keyboard, such as `KeyEvent.VK_LEFT` for the left arrow key, `KeyEvent.VK_RIGHT` for the right arrow key, and so on. These constants are used in the event handler to map specific key presses to player movements.

The `KeyAdapter` class allows us to handle key events with minimal boilerplate. It is used here to intercept the key press events that are fired when the user presses any of the defined keys for movement. By extending `KeyAdapter` and overriding the `keyPressed` method, we can define the specific actions that should occur when a key is pressed.

Example of KeyEvent Handler Implementation

```
gamePanel.addKeyListener(new KeyAdapter() {
    @Override
    public void keyPressed(KeyEvent e) {
        int keyCode = e.getKeyCode();
        if (keyCode >= KeyEvent.VK_LEFT && keyCode <=
KeyEvent.VK_DOWN) {
            game.setKeyCodePlayer1(keyCode);
        } else if (keyCode == KeyEvent.VK_W || keyCode ==
KeyEvent.VK_A || keyCode == KeyEvent.VK_S || keyCode ==
KeyEvent.VK_D) {
            game.setKeyCodePlayer2(keyCode);
        }
    }
});
```

This code snippet listens for key presses and sets the corresponding key code for `player1` or `player2`, which is later used to decide how the player moves.

Key Mapping and Player Movement

Each player has a defined set of keys to move on the battlefield. Player 1 is mapped to the arrow keys (`Up`, `Down`, `Left`, `Right`), and Player 2 is mapped to the WASD keys (`W`, `A`, `S`, `D`). This allows both players to control their movements independently and simultaneously.

Handling Key Inputs

When a key is pressed, the game determines which direction the player should move in. Based on the key pressed, the corresponding movement is made. Here's a breakdown of the movement handling logic:

- **Player 1 Movement**: Player 1 uses the arrow keys. The following checks are performed:
    - `Up`: Moves the player one step up (decreases the x-coordinate of the player's position).
    - `Down`: Moves the player one step down (increases the x-coordinate).
    - `Left`: Moves the player one step left (decreases the y-coordinate).
    - `Right`: Moves the player one step right (increases the y-coordinate).

Similarly, for **Player 2**, the WASD keys are used to control their movement on the battlefield.

Example of Movement Logic for Player 1

```
public void movePlayer1(BattleFieldGUI gui) {
    if (blueFlag) {
        System.out.println(player1.getName() + " Died...");
        gameState = false;
        player2.incrementScore();
        player2RoundWins++;
        return;
    }

    gui.markTrail(posOfPlayer_1, Colors.LIGHT_RED);
    switch (keyCodePlayer_1) {
        case KeyEvent.VK_UP:
            if (posOfPlayer_1[0] > 0) posOfPlayer_1[0]--;
            else blueFlag = true; break;
        case KeyEvent.VK_DOWN:
            if (posOfPlayer_1[0] < battleFieldHeight - 1)
posOfPlayer_1[0]++;
            else blueFlag = true; break;
        case KeyEvent.VK_LEFT:
            if (posOfPlayer_1[1] > 0) posOfPlayer_1[1]--;
            else blueFlag = true; break;
        case KeyEvent.VK_RIGHT:
            if (posOfPlayer_1[1] < battleFieldWidth - 1)
posOfPlayer_1[1]++;
            else blueFlag = true; break;
    }

    if
(gui.getCellColor(posOfPlayer_1).equals(Color.decode(Colors.LIGHT_BL
UE.getColorCode()))
        || blueFlag
        || gui.getCellColor(posOfPlayer_1).equals(Color.DARK_GRAY))
{
        blueFlag = true;
        gameState = false;
        System.out.println(player1.getName() + " Died!");
        player2RoundWins++;
```

```
    } else {
        gui.updatePlayer1Position(posOfPlayer_1, blueFlag);
    }
}
```

This method handles player movement based on the key input. It checks whether the new position is valid and updates the player's position if valid. If the player moves out of bounds or collides with a trail, the player "dies," and the game state changes accordingly.

2. Collision Detection and Game State

When players move, the game needs to check if they collide with anything, either the other player's trail or an obstacle on the battlefield. This is critical to making the game challenging and preventing players from moving recklessly.

Collision Detection for Player 1

When Player 1 moves, the game checks whether their new position overlaps with any of the following:

- **Player 2's Trail**: This is represented by the color of the cell. If Player 1 moves into a cell that has Player 2's trail (or color), Player 1 dies.
- **Walls or Obstacles**: The game also checks for obstacles (represented by the color DARK_GRAY).
- **Out of Bounds**: If Player 1 tries to move outside the battlefield, the game will stop their movement and mark them as "dead."

Player Death and Game State

If a player collides with another player's trail, the game ends the current round and increments the opposing player's score. This is handled by the following condition:

```
if
(gui.getCellColor(posOfPlayer_1).equals(Color.decode(Colors.LIGHT_BL
UE.getColorCode()))
    || blueFlag
    || gui.getCellColor(posOfPlayer_1).equals(Color.DARK_GRAY)) {
    blueFlag = true;
    gameState = false;
    System.out.println(player1.getName() + " Died!");
```

```
    player2RoundWins++;
}
```

This logic checks if Player 1's new position overlaps with Player 2's trail (`LIGHT_BLUE`) or a wall (`DARK_GRAY`). If true, Player 1 dies, and the round ends.

3. Managing Multiple Players

Two Players, Two Control Schemes

The game supports two players with different control schemes:

- **Player 1** uses the arrow keys.
- **Player 2** uses the WASD keys.

Each player has their own set of movement logic and checks, and the game constantly updates their positions on the battlefield. To allow for smooth, simultaneous player actions, the game runs two independent movement checks within the same round—one for each player.

Round Wins and Game Progression

The `BattleField` class also manages round wins and keeps track of who wins each round. After each player's movement is processed, if one player dies, the other player wins the round and their score is updated. This logic is handled by the following code:

```
player1RoundWins++;
```

Similarly, the game can reset after each round to prepare for the next.

Conclusion

This chapter explored the underlying logic that powers player movement within the `BattleField` game. By using key event listeners, handling player input, and updating the game state based on collisions and player actions, the game ensures an interactive and immersive experience. Players can control their characters with precise inputs, and the game dynamically responds to their movements in real-time. The algorithms discussed here form the core of the game's movement mechanics, ensuring both challenge and excitement for the players.