**Algorithmic Approach for the Platform Game**

**1. Define the Game Level Structure**

* Store the level layout as a string (simpleLevelPlan).
* Convert it into a **2D grid array (rows)** by splitting the string into lines and mapping each character to an object type.
* Store important attributes:
  + width and height of the level.
  + startActors → A list of game entities (Player, Lava, Coins) and their initial positions.
* Implement **collision detection (touches())**:
  + Check if a given position and size overlap with a wall or lava.
  + If an entity is outside the level bounds, consider it as touching a wall.

**2. Define Game Entities (Actors)**

Each entity has:

* **Position (pos)** → Current coordinates.
* **Size (size)** → Defines the space occupied.
* **Update logic (update())** → Determines movement per frame.
* **Collision handling (collide())** → Defines how it interacts with other objects.

Actors include:

1. **Player**
   * Moves left or right based on key input.
   * Affected by gravity.
   * Can jump when pressing ArrowUp.
   * Updates position while checking collisions.
   * If collides with lava, game status changes to "lost".
2. **Lava (Hazard)**
   * Moves horizontally (=) or vertically (|).
   * If it reaches a wall, it reverses direction.
   * Some lava objects reset to a starting position when they hit a wall (v).
   * If player touches lava, the game ends ("lost").
3. **Coin (Collectible)**
   * Coins slightly move up and down (wobble effect).
   * When collected, it is removed from the level.
   * If all coins are collected, game status changes to "won".

**3. Manage the Game State**

* **Stores**:
  + level → The current level layout.
  + actors → All entities in the game (Player, Lava, Coins).
  + status → Can be "playing", "lost", or "won".
* **Methods**:
  + update() → Updates all actors based on time elapsed.
  + player() → Returns the Player object.
  + If the player touches lava, change status to "lost".
  + If no coins remain, change status to "won".

**4. Handle User Input**

* Track arrow key presses using trackKeys().
* Store key states (true for pressed, false for released).
* In Player.update(), apply movement based on key states.
* Prevent movement into walls using touches().

**5. Render the Game**

* **Create HTML elements dynamically**:
  + Use <table> for the background grid.
  + Use <div> elements for actors (Player, Lava, Coins).
* **Update actor positions (syncState())**:
  + Remove the old actor layer and redraw new positions.
  + Update game status visually (e.g., losing or winning effects).
* **Scroll camera to follow the player (scrollPlayerIntoView())**:
  + Ensure the player is always within view by adjusting the scroll position dynamically.

**6. Game Loop & Animation**

* **runAnimation()** → Repeatedly calls the update function.
  + Ensures smooth frame transitions.
  + Limits time step to prevent extreme movement changes.
* **runLevel()** → Runs a single level:
  + Creates the game display.
  + Continuously updates the state while the game is active.
  + Ends the level when status changes to "lost" or "won".
* **runGame()** → Runs multiple levels in sequence:
  + Starts at level 0.
  + Advances to the next level if the player wins.
  + Resets the current level if the player loses.

**7. Implementation Steps**

1. **Parse Level Layout** → Convert simpleLevelPlan to a Level object.
2. **Initialize Game State** → Create State.start(level).
3. **Listen for Key Events** → Use trackKeys() to capture movement keys.
4. **Run the Game Loop** → Update game state and render the new positions.
5. **Check for Collisions** → Determine interactions (wall blocking, hazards, collectibles).
6. **Handle Winning Condition** → Progress to the next level when all coins are collected.

This structured plan helps you implement the game using your own syntax while maintaining proper logic. You can adapt it to any programming style or game engine you prefer.

**Algorithmic Approach for the Platform Game**

**1. Define the Game Level Structure**

* Store the level layout as a string (simpleLevelPlan).
* Convert it into a **2D grid array (rows)** by splitting the string into lines and mapping each character to an object type.
* Store important attributes:
  + width and height of the level.
  + startActors → A list of game entities (Player, Lava, Coins) and their initial positions.
* Implement **collision detection (touches())**:
  + Check if a given position and size overlap with a wall or lava.
  + If an entity is outside the level bounds, consider it as touching a wall.
  + Different objects have different collision behaviors.

**2. Define Game Entities (Actors)**

Each entity has:

* **Position (pos)** → Current coordinates.
* **Size (size)** → Defines the space occupied.
* **Velocity (speed or wobble)** → Determines movement over time.
* **Update logic (update())** → Determines movement per frame.
* **Collision handling (collide())** → Defines how it interacts with other objects.

Actors include:

1. **Player**
   * Moves left or right based on key input.
   * Affected by gravity.
   * Can jump when pressing ArrowUp.
   * Updates position while checking collisions.
   * If collides with lava, game status changes to "lost".
   * Movement is affected by acceleration and friction.
2. **Lava (Hazard)**
   * Moves horizontally (=) or vertically (|).
   * If it reaches a wall, it reverses direction.
   * Some lava objects reset to a starting position when they hit a wall (v).
   * If player touches lava, the game ends ("lost").
3. **Coin (Collectible)**
   * Coins slightly move up and down (wobble effect) to create an animation.
   * When collected, it is removed from the level.
   * If all coins are collected, game status changes to "won".
   * Uses a sine wave function to create smooth oscillation.

**3. Manage the Game State**

* **Stores**:
  + level → The current level layout.
  + actors → All entities in the game (Player, Lava, Coins).
  + status → Can be "playing", "lost", or "won".
* **Methods**:
  + update() → Updates all actors based on time elapsed.
  + player() → Returns the Player object.
  + If the player touches lava, change status to "lost".
  + If no coins remain, change status to "won".
  + Ensures smooth physics updates by handling time steps.

**4. Handle User Input**

* Track arrow key presses using trackKeys().
* Store key states (true for pressed, false for released).
* In Player.update(), apply movement based on key states.
* Prevent movement into walls using touches().
* Handle jump mechanics by checking if the player is on solid ground.

**5. Render the Game**

* **Create HTML elements dynamically**:
  + Use <table> for the background grid.
  + Use <div> elements for actors (Player, Lava, Coins).
* **Update actor positions (syncState())**:
  + Remove the old actor layer and redraw new positions.
  + Update game status visually (e.g., losing or winning effects).
* **Scroll camera to follow the player (scrollPlayerIntoView())**:
  + Ensure the player is always within view by adjusting the scroll position dynamically.
  + Keeps the player centered for better visibility in large levels.

**6. Game Loop & Animation**

* **runAnimation()** → Repeatedly calls the update function.
  + Ensures smooth frame transitions.
  + Limits time step to prevent extreme movement changes.
  + Handles frame skipping to maintain consistent physics updates.
* **runLevel()** → Runs a single level:
  + Creates the game display.
  + Continuously updates the state while the game is active.
  + Ends the level when status changes to "lost" or "won".
* **runGame()** → Runs multiple levels in sequence:
  + Starts at level 0.
  + Advances to the next level if the player wins.
  + Resets the current level if the player loses.
  + Allows restarting after completion.

**7. Implementation Steps**

1. **Parse Level Layout** → Convert simpleLevelPlan to a Level object.
2. **Initialize Game State** → Create State.start(level).
3. **Listen for Key Events** → Use trackKeys() to capture movement keys.
4. **Run the Game Loop** → Update game state and render the new positions.
5. **Check for Collisions** → Determine interactions (wall blocking, hazards, collectibles).
6. **Handle Winning Condition** → Progress to the next level when all coins are collected.
7. **Optimize Performance** → Minimize unnecessary DOM updates for smooth rendering.

This structured plan helps you implement the game using your own syntax while maintaining proper logic. You can adapt it to any programming style or game engine you prefer.