Bomberman

Software Technology

Contributors:

Mohammed Efaz (ZTFK53)

Md Rakibul Hasan (JR9RXP)

Isroilbek Jamolov (DXFV5Y)

Diyorbek Uktamov (A0SJHR)

Problem Description:

A competitive multiplayer Bomberman game, is played on a 2-dimensional field, consisting of square-shaped tiles. The game is played by at most 3 players, each controlling a Bomberman character with the goal of being the last one standing. The game field contains wall elements, boxes, monsters, and the players' characters. Players can place bombs to blow up boxes, monsters, and players (including themselves). A player loses (and thus their opponent wins and gains one point) if they explode or are caught by a monster. There are 4 rounds.

Framework and Tools:

Java Swing

Apache Maven

Subtasks:

- **Intelligent Monsters:**
  Implementing diverse monster behaviors to challenge players.
- **Advanced Power-ups:**
  Introducing additional power-ups for strategic gameplay advantages.
- **Three Players Mode:**
  Expanding the game to support a three-player mode.
- **Continuous Movement:**
  Improving character movement for a smoother gaming experience.

Main Code:

Entity.java:

Constructor: Entity(GamePanel gp)

Initializes a new entity with a reference to the game panel for accessing game-wide properties and methods.

Method: setAction()

Intended for overriding; should specify entity-specific behaviors like movement or interactions.

Method: update()

Processes movement based on direction, handles collision checks with tiles, monsters, and players, and manages animation cycles.

Method: draw(Graphics2D g2)

Determines and draws the current sprite image based on the entity's direction and animation state.

Method: setup(String imagePath)

Loads, scales, and returns an image from a given path to use as the entity's sprite.

Player.java:

Constructor: Player(GamePanel gp, KeyHandler keyH, int playerNum)

- Initializes a player with a specific number within a game, linking to the game's control handling system. Sets up player-specific images and default properties.

Method: setDefaultValues()

- Sets the default values for player properties such as position, speed, life, and bomb limits based on the player number.

Method: getPlayerImage()

- Loads and sets up the sprite images for the player based on their specific color, which is determined by the player number.

Method: placeBomb()

- Allows the player to place a bomb if they haven't reached their bomb limit, adds it to the game, and updates the bomb count.

Method: detonateBombs()

- Triggers the detonation of all placed bombs by the player, clears the bombs list, and resets the detonator status.

Method: placeObstacle()

- Places an obstacle on the game map at the player's last position if the obstacle limit hasn't been reached and the tile is not already occupied.

Method: makeInvincible(int duration)

- Grants the player temporary invincibility for a set duration, preventing damage from collisions or attacks.

Method: endInvincibility()

- Ends the player's invincibility and stops any blinking effects that indicate invincibility.

Method: enableGhost(int duration)

- Makes the player a ghost, allowing them to move without collisions for a set duration.

Method: disableGhost()

- Disables the ghost state, restoring normal collision behavior.

Method: bombExploded()

- Decreases the count of active bombs when one explodes, managing the player's capacity to place more bombs.

Method: updateBombInteraction()

- Checks and updates interactions between the player and their bombs, primarily managing bomb collision avoidance rules.

Method: update()

- Overrides the Entity.update() method to incorporate player-specific controls and behaviors such as movement handling based on key inputs, collision checks, and power-up interactions.

Method: pickUpPowerUp(int index)

- Activates and removes a power-up from the game when the player collides with it.

Method: pickUpObject(int index)

- Picks up an object if the player collides with it, removing it from the game.

Method: contactMonster(int index)

- Handles interactions with monsters, setting the player's life to zero if they collide with a monster while not invincible.

Method: updateInvincibility()

- Manages the countdown of the player's invincibility duration and blinking effect as it nears expiration.

Method: updateGhost()

- Manages the countdown and potential deactivation of the ghost mode, including the blinking effect as it nears expiration.

Method: draw(Graphics2D g2)

- Extends the drawing capabilities of Entity.draw() to incorporate visual effects related to invincibility and ghost states, including semi-transparency and positional adjustments based on overlapping effects.

AssetSetter.java:

Constructor: AssetSetter(GamePanel gp)

- Initializes an instance of AssetSetter with a reference to the GamePanel object to access and manipulate game elements.

Method: setObject()

- Places ghost objects at predefined positions on the game map. This method initializes two ghost entities and positions them using the game panel's tile size for precise placement.

Method: setMonster()

- Spawns different types of monsters at specific locations on the game map. This method sets up an array of monsters, including red slime, green slime, orc, and skeleton, by positioning them using the game panel's tile size.

Method: setPowerUp()

- Randomly distributes various types of power-ups in available box locations on the game map. It uses the Random class to select random box locations for placing power-ups such as extra bombs, blast expansion, detonators, roller skates, invincibility, ghost abilities, and obstacles. If there are no more boxes available, the loop breaks early.

CollisionChecker.java:

Constructor: CollisionChecker(GamePanel gp)

- Initializes the CollisionChecker with a reference to the GamePanel to access game environment details and entity positions.

Method: checkTile(Entity entity)

- Handles collision checks for a given entity against tiles in the game map. Differentiates between general entities and specific types like MON_orc for custom collision logic.

Method: isEdgeTile(int col, int row)

- Determines if a tile at the specified column and row is an edge tile of the game map.

Method: defaultCollisionCheck(Entity entity)

- Performs a collision check for the specified entity against the game's tiles based on the entity's direction and speed. Includes logic to ignore collisions for ghost-mode players unless at map edges.

Method: orcCollisionCheck(MON_orc orc)

- Specialized collision check for MON_orc type entities that involves checking intersections with game objects like bombs.

Method: checkBombCollision(Entity entity, boolean player)

- Checks for collisions between the specified entity and bombs in the game, adjusting collision status based on entity properties like ghost mode.

Method: collisionIndex(Entity entity, boolean player, int index, SuperObject obj)

- Helper method used during bomb collision checks to determine if an entity collides with an object and returns the index of the colliding object.

Method: checkObject(Entity entity, boolean player)

- Checks for collisions between the entity and non-bomb objects in the game, setting collision flags and possibly returning the index of the colliding object.

Method: checkPowerUpCollision(Player player)

- Checks for intersections between the player and power-up objects, returning the index of the colliding power-up for potential collection.

Method: checkEntity(Entity entity, Entity[] target)

- Checks for collisions between the specified entity and an array of target entities, such as monsters, updating collision flags as necessary.

Method: checkPlayer(Entity entity)

- Checks for collisions between a non-player entity and all player entities in the game, updating collision flags as necessary.

Method: directionChange(Entity entity)

- Adjusts the position of the entity's collision area based on its movement direction and speed, preparing for collision checks.

Method: canMove(Entity entity, String direction)

- Determines if an entity can move in a specified direction without colliding with tiles, using the game's tile management system to check collidable tiles.

Method: isCollidable(int col, int row)

- Checks if a tile at a specific column and row is collidable, aiding movement checks and collision handling.

EventHandler.java:

Constructor: EventHandler(GamePanel gp)

- Initializes the EventHandler with a reference to the GamePanel. Sets up a default rectangular area (eventRect) that will be used to detect events. This rectangle's position and size can be adjusted to target specific areas for event detection.

Method: checkEvent()

- This method serves as a placeholder within the EventHandler class. It is intended to be overridden or filled in with specific logic to check for and handle various game events based on the game state and interactions detected by the hit method.

Method: hit(int eventCol, int eventRow, String reqDirection)

- Checks if any player has interacted with the specified event area based on their position and direction.

Parameters:

- eventCol: The column of the tile where the event is located.
- eventRow: The row of the tile where the event is located.
- reqDirection: The required direction the player needs to be facing to trigger the event. This can also be set to "any" to allow the event to be triggered from any direction.

GamePanel.java:

Constructor: GamePanel()

- Initializes the game panel, setting up dimensions based on tile size and screen grid. It also loads essential game resources and initializes player settings, UI components, and game state management.

Method: initPlayers()

- Initializes or re-initializes the array of player objects based on the number of players configured in the game UI.

Method: updatePlayerSettings()

- Resets player settings, typically called when game settings are changed, like the number of players.

Method: setupGame()

- Calls methods from the AssetSetter to populate the game environment with objects, monsters, and power-ups.

Method: startGameThread()

- Starts the game loop running in a new thread.

Method: run()

- Contains the game loop, which handles game state updates and rendering at a fixed time interval determined by the FPS setting.

Method: update()

- Updates game logic, including player and monster updates, checking for power-up interactions, and handling game state transitions based on gameplay events such as all players dying or all monsters being cleared.

Method: nextRound(Player winner)

- Advances the game to the next round, updating player scores and resetting the game environment as necessary.

Method: resetRound()

- Resets the game to its initial state for a new round, reinitializing game assets and settings.

Method: endGame()

- Handles the end of the game, displaying final scores and transitioning to the game over state.

Method: loadRandomMap()

- Loads a random map from available resources, adding variability to game rounds.

Method: isTileOccupied(int x, int y, Player placingPlayer)

- Checks if a specific tile is occupied by any player other than the one specified, used to validate placements of objects or movement.

Method: displayWinner(Player player) and displayNoWinner()

- Utility methods to display the game winner or indicate a draw, respectively.

Method: resetGame()

- Resets the entire game to default values, typically called after a game over or when restarting the game from the menu.

Method: paintComponent(Graphics g)

- Overrides JPanel's paintComponent to render the game elements on the panel, handling both UI and game object rendering.

Mouse Event Handlers (mouseClicked, mousePressed, mouseReleased, mouseEntered, mouseExited)

- Implements methods from MouseListener to handle mouse interactions, primarily used to detect clicks on UI elements like buttons in the game over screen.

KeyHandler.java:

Constructor: KeyHandler(GamePanel gp)

- Initializes the KeyHandler with a reference to GamePanel.
- Enables the handling of key inputs across different parts of the game.

Method: keyTyped(KeyEvent e)

- Currently, this method does not perform any operation.

Method: keyPressed(KeyEvent e)

- Determines the game state and calls the corresponding method to handle key presses for that state.

Method: keyReleased(KeyEvent e)

- Updates boolean flags for key states to false when keys are released, indicating the end of the corresponding movement or action.

Method: titleScreenInput(int code)

- Processes key inputs specific to the title screen, such as navigating menus or starting the game.

Method: playStateInput(int code)

- Handles key inputs during active gameplay, including player movement, bomb placement, and pausing the game.

Method: pauseStateInput(int code)

- Manages key inputs when the game is paused, allowing navigation through pause menu options or resuming the game.

UI.java:

Constructor: UI(GamePanel gp)

- Initializes the UI with a reference to the GamePanel and sets up initial UI-related configurations, such as fonts and player heart icons.

Method: updateUI()

- Updates UI components based on the current game state, delegating to specific drawing methods like drawTitleScreen or drawPlayerLife.

Method: setOkButtonCoordinates(int x, int y)

- Sets the x and y coordinates for the OK button used in various UI screens.

Method: getOkButtonX() and getOkButtonY()

- Returns the x and y coordinates of the OK button, respectively.

Method: setGraphics(Graphics2D g2)

- Sets the Graphics2D object for the UI, which is used for drawing on the GamePanel.

Method: showMessage(String text)

- Displays a message on the screen temporarily. This uses a separate thread to manage visibility duration.

Method: draw(Graphics2D g2)

- Central drawing method that updates the Graphics2D object and delegates drawing based on the game state.

Method: drawPlayerLife()

- Draws the life indicators (hearts) for players, showing current and maximum life.

Method: drawPlayerNumberSelection(Graphics2D g2)

- Draws the player number selection screen, allowing the user to choose how many players will participate.

Method: drawPlayerNameInputs(Graphics2D g2)

- Provides input fields for players to enter their names, typically used in the game setup phase.

Method: drawMapSelection(Graphics2D g2)

- Allows players to select a game map from available options.

Method: drawTitleScreen()

- Draws the main title screen of the game, which includes menu options like start game, settings, and exit.

Method: drawPauseScreen()

- Draws the pause screen, offering options like resuming the game or going to the settings.

Method: drawFinishScreen(Graphics2D g2)

- Displays the game over screen with the final scores and a message indicating the game's completion.

Method: showEndGameScores(int[] scores)

- Shows the end game scores on a custom-designed game over screen with an OK button to acknowledge the end of the game.

Method: drawTextWithBorder(Graphics2D g2, String text, int x, int y)

- Utility method to draw text with a border for better visibility against varied backgrounds.

Method: getGameOverOkButtonY()

- Computes and returns the y-coordinate for the OK button on the game over screen.

Method: getXForCentreText(String text)

- Calculates the x-coordinate for centering text on the screen based on its length.

UtilityTool.java:

Method: scaleImage(BufferedImage original, int width, int height)

- This method creates a new BufferedImage of the specified width and height, then draws the original image onto the new image at the new dimensions. The Graphics2D object used for drawing is then disposed to release system resources.

MON_greenSlime.java:

Constructor: MON_greenSlime(GamePanel gp)

- Initializes a new instance of MON_greenSlime with a reference to the game panel (GamePanel).
- Sets initial attributes such as name, speed, life, and the collision area of the slime.
- Calls the getImage() method to load specific images for different movement directions.

Method: getImage()

- Loads and assigns sprite images for different directions (up, down, left, right) from specified paths. This method assumes the same image is used for all directions but differentiates between two states (e.g., up1 and up2 for animation).

Method: setAction()

- Determines the slime's next action based on a random decision-making process.
- Randomly chooses a direction and updates the slime's movement speed.
- Uses an actionLockCounter to introduce variability in movement changes, which provides more dynamic and unpredictable movements.
- Checks if the new direction is feasible using the CollisionChecker's canMove() method before updating the direction, ensuring the slime does not move into obstacles.

MON_orc.java:

Constructor: MON_orc(GamePanel gp)

- **Purpose**: Initializes a new instance of an orc with base settings for movement speed, life, and collision areas.
- **Details**: Sets specific attributes like name, movement speed, life statistics, and initializes the sprite images for various movement directions.

Method: getImage()

- **Purpose**: Loads sprite images for the orc from specified file paths for different directions (up, down, left, right).
- **Details**: Each direction has two sprite states (e.g., up1 and up2) to facilitate animation.

Method: setAction()

- **Purpose**: Determines the next action for the orc based on its current situation, such as facing obstacles or reaching map edges.
- **Details**: Uses random direction choices and checks whether moving in a new direction is feasible without encountering obstacles.

Helper Methods:

- **isPathBlocked()**: Checks if the orc's next move leads to a collision with bombs or map boundaries.
- **isNextStepAtEdge()**: Determines if the next move would bring the orc too close to the map's edge, preventing movement in that direction.
- **changeDirectionRandomly()**: Changes the orc's movement direction randomly, ensuring it does not move into blocked paths.

Method: noGrassTileAhead(String direction)

- **Purpose**: Checks if moving in the specified direction leads to a non-walkable tile (e.g., a tile without grass).
- **Details**: This method is part of the decision-making process for changing movement direction.

Method: wouldCollideWithEdge(String testDirection)

- **Purpose**: Tests if moving in a certain direction would result in a collision with the map edge.
- **Details**: Useful in the decision-making process to avoid paths leading directly into boundaries.

Method: isNextStepAtEdge(int testX, int testY)

- **Purpose**: Overloaded version of isNextStepAtEdge() that checks for boundary collisions based on hypothetical next positions.
- **Details**: Supports the wouldCollideWithEdge method by providing a way to predict collisions before making a move.

Method: update()

- **Purpose**: Regularly updates the orc's state, including its position based on the current speed and direction.
- **Details**: Also manages the animation cycle by updating sprite frames.

Method: updateSprite()

- **Purpose**: Manages sprite animation by cycling through sprite images.
- **Details**: Adjusts spriteNum based on spriteCounter to switch between animation frames.

MON_redSlime.java:

Constructor: MON_redSlime(GamePanel gp)

- Initializes the Red Slime with specific attributes such as name, speed, life, and collision area.
- Calls getImage() to load images for different movement animations.

Method: update()

- Updates the state of the Red Slime each frame, deciding on stopping or moving based on the proximity of the nearest player.

Method: getImage()

- Loads images for different movement states (up, down, left, right) from resource files, supporting two animation states for each direction.

Method: setAction()

- Regularly updates the slime's behavior, choosing either to move towards the nearest player or to wander randomly if direct movement is not possible.

Method: moveInCurrentDirection()

- Moves the slime in its current direction if there are no obstacles, using collision checking from the GamePanel.

Method: shouldStop(Player player)

- Determines if the Red Slime should stop moving based on its distance to the nearest player, primarily to align itself with the player within a certain range.

Method: stopMovement()

- Stops the slime's movement by setting its movement flag to false.

Method: attemptMoveTowardsPlayer(Player player)

- Attempts to move directly towards the nearest player by evaluating which direction will decrease the distance to the player the most, considering obstacles.

Method: moveInDirection(String dir)

- Moves the slime in a specified direction, adjusting its coordinates based on its current speed.

Method: wanderRandomly()

- Chooses a random direction to move when no clear path towards the player is available, ensuring the slime remains active.

Method: findNearestPlayer()

- Searches for the closest player by calculating distances from all available players, returning the nearest one.

Method: relevance(String direction, int dx, int dy)

- Calculates the relevance of a direction based on how directly it leads towards a player, used to sort potential movement directions.

Utility Methods for Movement Decision:

- getNextStepTowardsPlayer(Player player): Calculates the next step towards the player if a direct path is clear.
- rankDirection(String dir, int dx, int dy): Ranks movement directions based on their effectiveness in reducing distance to a target player, used for sorting possible movements.

MON_skeleton.java:

Constructor: `MON_skeleton(GamePanel gp)`

- **Purpose**: Initializes a new instance of a skeleton monster, inheriting behaviors and properties from the `MON_redSlime` class but changing the name to "Skeleton" to reflect its unique identity.

Method: `setAction()`

- **Purpose**: Overrides the behavior logic to occasionally make incorrect decisions, simulating error in the skeleton's movement.
- **Details**: The method modifies the regular action cycle by introducing a probability (`errorProbability`) where the skeleton might choose a random, potentially suboptimal

direction to move. This is designed to make the skeleton's behavior less predictable and more challenging for the player.
- **Execution**: Every few cycles (determined by `actionLockCounter`), the skeleton may choose a random direction from those available (not blocked by obstacles) or follow its inherited behavior from `MON_redSlime`.

Method: `getPossibleDirections()`

- **Purpose**: Filters and returns a list of viable movement directions that are not currently blocked by obstacles.
- **Details**: Utilizes the `CollisionChecker` from `GamePanel` (`gp.cChecker`) to determine which directions are currently feasible for movement.

Method: `getImage()`

- **Purpose**: Loads specific sprite images for the skeleton's different movement directions.
- **Details**: Assigns two different images for each direction (up, down, left, right) to support animation. The images are specifically designated for the skeleton, distinguishing its appearance from the red slime.
-

SuperObject.java:

The `SuperObject` class is a foundational component in a game, designed to represent interactive or non-interactive objects within the game world. It provides essential attributes for image handling, position, and collision detection, along with basic methods for drawing these objects on the game panel and an empty `update()` method for subclasses to implement specific behaviors. The class simplifies the creation and management of diverse game elements by centralizing common functionalities needed across various object types.

BlastExpansion.java:

Constructor: `BlastExpansion(GamePanel gp, int x, int y)`

- Initializes a `BlastExpansion` power-up at specific coordinates, setting its type to `BLAST_EXPANSION`.

Method: `applyEffect(Player player)`

- Increases the player's bomb blast radius by one, enhancing their bomb's effective area.

Method: `removeEffect(Player player)`

- Reduces the player's bomb blast radius by one, but not below a minimum threshold, ensuring a base level of enhanced capability remains.
-

Detonator.java:

Constructor: `Detonator(GamePanel gp, int x, int y)`

- Initializes a `Detonator` power-up at specified coordinates, setting its type to `DETONATOR` and a duration of 5 (presumably 5 seconds or game cycles).

Method: `applyEffect(Player player)`

- Grants the player the ability to detonate bombs at will, enabling strategic gameplay enhancements.

Method: `removeEffect(Player player)`

- Reverses the detonator ability, removing the player's capacity to manually detonate bombs.

ExtraBomb.java:

Constructor: `ExtraBomb(GamePanel gp, int x, int y)`

- Initializes an `ExtraBomb` power-up at specified coordinates, specifically setting its type to `EXTRA_BOMB` and indicating no inherent duration for the effect (denoted by `0`).

Method: `applyEffect(Player player)`

- Increases the player's bomb capacity by one, but only if their current bomb limit is 1, allowing them to carry multiple bombs.

Method: `removeEffect(Player player)`

- Decreases the player's bomb limit by one upon removal of the power-up, ensuring that the limit does not fall below 1, thereby reverting to the initial bomb-carrying capacity.

Ghost.java:

Constructor: `Ghost(GamePanel gp, int x, int y)`

- Initializes the `Ghost` power-up at the specified coordinates within the game panel and sets a duration for the effect (5, which could be in seconds or game cycles depending on how `duration` is used in the game logic).

Method: `applyEffect(Player player)`

- Activates the ghost mode for the player, translating the duration from seconds to game frames using `convertSecondsToFrames(duration)`, allowing the player to pass through obstacles.

Method: `removeEffect(Player player)`

- Deactivates the ghost mode, returning the player to normal collision interactions with obstacles.

Invincibility.java:

Constructor: `Invincibility(GamePanel gp, int x, int y)`

- Initializes an `Invincibility` power-up at the specified coordinates on the game panel, setting it to a type of `INVINCIBILITY` and a fixed duration of 5 units, likely translating to seconds.

Method: `applyEffect(Player player)`

- Confers temporary invincibility to the player if they are not already invincible. The duration of the effect is calculated by converting seconds into frames (a common method in game development for handling time-based effects).

Method: `removeEffect(Player player)`

- Terminates the invincibility effect, making the player vulnerable again to damage and other game mechanics that had been temporarily suspended.

Obstacle.java:

Constructor: `Obstacle(GamePanel gp, int x, int y)`

- Initializes an `Obstacle` power-up at the specified coordinates within the game panel. It sets this power-up type to `OBSTACLE` and assigns no duration (`0`), indicating that the effect may be permanent or not time-based.

Method: `applyEffect(Player player)`

- Increases the player's limit for placing obstacles by 3, allowing them to strategically alter the game environment by adding more barriers that can block or reroute other players and enemies.

Method: `removeEffect(Player player)`

- This method is overridden but not implemented, suggesting that the effects of this power-up are intended to be permanent or that the game does not require the removal of this type of effect under normal gameplay circumstances.

PowerUp.java:

Constructor: `PowerUp(GamePanel gp, Type type, int x, int y, int duration)`

- Initializes a power-up object at a specified location with a defined duration (converted to frames) and type (such as `DETONATOR`, `OBSTACLE`, etc.). It also initializes the power-up's image based on its type.

Abstract Methods:

- **`applyEffect(Player player)`**: Applies the specific effect of the power-up to the player. This must be implemented by subclasses to define what happens when a power-up is activated.
- **`removeEffect(Player player)`**: Reverses the effect of the power-up on the player. This is typically called when the power-up's duration expires or under certain game conditions.

Method: `getBufferedImage()`

- Loads the image associated with the power-up's type from resource files, which visually represents the power-up on the game panel.

Method: `activate(Player player)`

- Activates the power-up, applying its effect to the player and resetting its duration counter. It sets the power-up as active.

Method: `deactivate()`

- Deactivates the power-up, removing its effect from the player who activated it and setting the power-up as inactive.

Method: `convertSecondsToFrames(int seconds)`

- Converts a time duration from seconds to frames, based on the game's frames per second (`FPS`), facilitating time-based effects in terms of game update cycles.

Method: `draw(Graphics2D g2)`

- Draws the power-up's image at its position on the game panel if it is set to be visible.

Method: `update()`

- Updates the state of the power-up each game cycle; decrements the remaining active frames and deactivates the power-up if its time runs out.

Visibility Control:

- **`setVisible(boolean visible)`**: Sets whether the power-up is visible on the game panel.
- **`isVisible()`**: Returns the visibility status of the power-up.

Utility and Accessor Methods:

- **`isActive()`**: Returns whether the power-up is currently active.
- **`getType()`**: Returns the type of the power-up.
- **`getX()` and `getY()`**: Return the x and y coordinates of the power-up on the game panel.

RollerSkate.java:

Constructor: `RollerSkate(GamePanel gp, int x, int y)`

- **Purpose**: Initializes a `RollerSkate` power-up at specified coordinates, setting its type to `ROLLER_SKATE`. This type of power-up has no inherent duration (`0`), suggesting that the speed boost might be permanent until explicitly removed or the game session ends.

Method: `applyEffect(Player player)`

- **Purpose**: Enhances the player's speed attribute by a fixed amount (`speedIncrease`), which is set to 2. This increase in speed allows the player to move faster on the game board, facilitating quicker navigation and potentially evading opponents or dangers more effectively.

Method: `removeEffect(Player player)`

- **Purpose**: Reverses the speed boost applied by the RollerSkate power-up, decreasing the player's speed by the same amount it was increased. This method ensures that the player's speed returns to normal once the power-up's effect is no longer needed or desired.

Tile.java:

Attributes:

- **image**: A `BufferedImage` object that holds the graphical representation of the tile. This could be anything from a section of the ground to a wall or any other visual element in your game's world.
- **collision**: A `boolean` that indicates whether the tile is passable or not. If set to `true`, this tile will block movement or certain types of interactions, acting as an obstacle.

Usage:

This class is typically used within a tile map system where each tile represents a piece of the game's terrain or environment. The `collision` attribute is crucial for gameplay mechanics involving movement, as it determines which areas of the map are navigable and which are barriers.

TileManager.java:

Constructor: `TileManager(GamePanel gp)`

- **Purpose**: Initializes a new `TileManager` object, setting up tiles and loading a map.
- **Details**: The constructor sets up the `Tile` array with basic tiles, loads specific tile images, and initializes the game map from a text file representing tile indices.

Method: `getTileImage()`

- **Purpose**: Loads and assigns images to the types of tiles handled by the `TileManager`.
- **Details**: Sets up common tiles like grass, wall, and destructible wall, specifying which tiles are passable (collision false) and which are not (collision true).

Method: `setup(int index, String imageName, boolean collision)`

- **Purpose**: Configures a tile at a specified index with an image and collision property.
- **Details**: Uses `UtilityTool` to load and scale the tile image appropriately to the game's tile size. It marks the tile as collidable or not based on the `collision` parameter.

Method: `loadMap(String filePath)`

- **Purpose**: Loads a map layout from a specified file, populating the `mapTileNum` array which dictates the tile types and locations on the game panel.
- **Details**: Reads a text file where each number corresponds to a tile type, arranging them according to the structure expected in the game's world. Special handling is noted for tiles like destructible walls by storing their locations for gameplay mechanics.

Method: `draw(Graphics2D g2)`

- **Purpose**: Renders the tiles onto the game panel.
- **Details**: Iterates over the `mapTileNum` grid, drawing each tile at its appropriate screen position, thereby constructing the visible game map for the player.

TESTING:

TestAdditional.java:

Test Setup with `@BeforeEach`

- **Method**: `setUp()`
- **Purpose**: Initializes shared test components before each test method runs. It sets up a new `GamePanel` instance and a `Player` object, and it also loads a map.
- **Actions**:
  - Instantiate `GamePanel`.
  - Load a game map which likely sets up the initial conditions for the test environment.

Test: `testPlayerPickUpPowerUp`

- **Purpose**: Verifies that a player can pick up an `ExtraBomb` power-up correctly and that the power-up affects the player's bomb limit as expected.
- **Expected Outcome**: The player's bomb limit should increase by 1 after picking up the power-up.

Test: `testPlayerLifeAfterDamage`

- **Purpose**: Checks the functionality that handles the player's life decrementing after taking damage.
- **Expected Outcome**: The player's current life count should decrease by 1.

Test: `testPlayerDetonateBombs`

- **Purpose**: Tests whether bombs placed by a player are correctly accounted for and can be detonated, affecting the bomb count appropriately.
- **Expected Outcome**: The bomb count should return to 0 after detonation, assuming all bombs explode.

Test: `testGamePausesAndResumes`

- **Purpose**: Ensures the game can transition between play and pause states correctly.
- **Expected Outcome**: The game's state should reflect accurately whether it is paused or resumed based on actions simulated in the test.

Test: `testPlayerResetAfterRoundEnds`

- **Purpose**: Confirms that player states (specifically life totals) are reset appropriately at the end of a game round.
- **Expected Outcome**: Both players should have their life counts reset to initial values as a new round starts.

TestBomb.java:

Test Setup with `@BeforeEach`

- **Method**: `setUp()`
- **Purpose**: Prepares the testing environment by initializing the `GamePanel` and a `Player` instance before each test method runs, ensuring a clean setup for every test case.
- **Actions**:
    - Instantiate `GamePanel` and `Player`, ensuring that each test begins with a default state for these objects.

Test: `testUpdateBombInteraction`

- **Purpose**: Verifies that bombs correctly recognize when a player has moved away sufficiently to enable collision.
- **Procedure**:
    - Simulate a player placing a bomb and then moving away from the bomb's location.
    - Check whether the bomb's `ignoreCollisionWithOwner` flag is set to false, which should occur once the player is no longer immediately adjacent to the bomb.
- **Expected Outcome**: After the player moves away, the bomb should update its collision status to not ignore collisions with the player, ensuring realistic interactions such as the bomb exploding upon contact or proximity.

Test: `testBombPlanting`

- **Purpose**: Tests the player's ability to place a bomb and ensure it's added to the game objects in the `GamePanel`.
- **Procedure**:
    - Trigger the player's bomb placement action.
    - Check the game panel's object list to confirm the presence of the newly placed bomb.
- **Expected Outcome**: The list of game objects should include exactly one bomb after the action, indicating successful bomb planting.

Test: `testBombIgnoringOwner`

- **Purpose**: Ensures that a newly placed bomb initially ignores collision with its owner, which is crucial for allowing the player to move away after placing a bomb.
- **Procedure**:
  - Directly create a bomb object at the player's location.
  - Verify that the bomb's `ignoreCollisionWithOwner` flag is initially set to true.
- **Expected Outcome**: The bomb should start with collision ignoring enabled to prevent immediate and unintended explosions that could penalize the player unfairly.

TestGame.java:

Test Setup with `@BeforeEach`

- **Method**: `setUp()`
- **Purpose**: Prepares the testing environment by initializing a `GamePanel` instance and a `Player` object, and also loads a game map.
- **Details**: This ensures that each test starts with a fresh game state, preventing interference from previous tests.

Test: `testBombExplosionChangesTile`

- **Purpose**: Tests whether a bomb explosion correctly alters the game map's tile type, simulating the destruction of destructible walls or obstacles.
- **Details**: Places a bomb at a specific location, triggers its explosion, and then checks if the tile has been changed to grass (represented as tile number `0`), indicating the bomb's effect on the environment.

Test: `testPowerUpExtraBomb`

- **Purpose**: Verifies that the `ExtraBomb` power-up correctly increases the player's bomb limit.
- **Details**: Applies the `ExtraBomb` effect to the player and asserts that the bomb limit is incremented.

Test: `testPowerUpBlastExpansion`

- **Purpose**: Ensures the `BlastExpansion` power-up properly extends the bomb blast radius.
- **Details**: Increases the player's bomb blast radius and checks that it has incremented as expected.

Test: `testPowerUpDetonator`

- **Purpose**: Confirms that the `Detonator` power-up grants the player the ability to detonate bombs on command.
- **Details**: Tests the application of the detonator effect and checks if the player can now trigger bombs at will.

Test: `testPowerUpRoller`

- **Purpose**: Tests whether the `RollerSkate` power-up successfully increases the player's speed.
- **Details**: Applies the roller skate effect and verifies that the player's speed is increased by 2 units.

Test: `testPowerUpGhost`

- **Purpose**: Evaluates both the activation and deactivation of the `Ghost` power-up.
- **Details**: Activates ghost mode (making the player able to pass through obstacles), then deactivates it and checks both states.

Test: `testInvincibilityActivationAndDeactivation`

- **Purpose**: Tests the functionality of the `Invincibility` power-up by checking its activation and subsequent deactivation.
- **Details**: Makes the player invincible and then removes invincibility, verifying both conditions.

Test: `estPowerUpObstacle`

- **Purpose**: Ensures the `Obstacle` power-up increases the player's obstacle placement limit.
- **Details**: Applies the obstacle power-up and confirms that the obstacle limit increases by 3.

TestPlayer.java:

Test Setup with `@BeforeEach`

- **Method**: `setUp()`
- **Purpose**: Initializes common elements used across tests, such as `GamePanel`, `Player`, and `KeyHandler`, to ensure each test starts with a consistent environment.
- **Details**: Sets up a new game panel, initializes player settings, and configures the game, ensuring that the player is ready for interaction in the tests.

Test: `testPlayerInit`

- **Purpose**: Ensures that the player is initialized correctly with default settings.
- **Details**: Tests initial values like position, speed, direction, and life, verifying that the player starts with the expected default values.
- **Expected Outcome**: The player's initial properties should match the predefined settings (position at (50, 50), speed of 4, direction "down", and full life).

Test: `testPlayerMovement`

- **Purpose**: Verifies that the player moves correctly in response to input.
- **Details**: Simulates a key press for moving up and checks if the player's position updates accordingly.
- **Expected Outcome**: After the update, the player's y-coordinate should decrease by the speed value (4 units), indicating movement upwards.

Test: `testPlayerCollision`

- **Purpose**: Tests the player's collision detection system by simulating a scenario where an obstacle blocks the player's path.
- **Details**: Places an obstacle directly in the player's movement path to the right and simulates a frame update where the player attempts to move right.
- **Expected Outcome**: The player should not overlap or pass through the obstacle, stopping just before it.

TestPlayerDamage.java:

Test Setup with `@BeforeEach`

- **Method**: `setUp()`
- **Purpose**: Prepares the testing environment by setting up a `GamePanel` instance, loading a game map, and initializing player settings.
- **Details**: This ensures a controlled environment where the player's initial state is consistent for each test, facilitating accurate assessments of changes due to game events.

Test: `testPlayerDamageByBomb`

- **Purpose**: Evaluates the damage mechanism by simulating a bomb explosion near the player to confirm that it correctly affects the player's life.
- **Procedure**:
  - Retrieves the player from the game panel.
  - Records the player's initial life for a baseline comparison.

    - Creates a bomb at the player's location and triggers its explosion.
    - Calls the game panel's update method to process the explosion's effect on the player.
- **Expected Outcome**: The player's life should decrease following the bomb explosion, demonstrating that the bomb damage is applied correctly.

Key Aspects Tested:

- **Bomb Damage Application**: The test checks if the game logic correctly reduces player health when a bomb explodes nearby. This is essential for gameplay integrity, ensuring that bombs represent a real hazard in the game.
- **Game Update Mechanics**: By invoking `gamePanel.update()`, the test also implicitly checks that the game's update loop is capable of processing events like bomb explosions and applying their effects correctly, which includes updating player health states.

TestUI.java:

Framework Setup with `AssertJSwingJUnitTestCase`

- **Base Class**: `AssertJSwingJUnitTestCase` provides a testing framework for writing GUI tests using AssertJ-Swing. It handles setup and teardown of the GUI testing context, making it easier to write robust GUI tests.

Test Environment Setup with `onSetUp()`

- **Method**: `onSetUp()`
- **Purpose**: Prepares the testing environment by creating a `GamePanel` inside a JFrame and wrapping it with a `FrameFixture` for interaction testing.
- **Details**: This method initializes the UI components within a frame and makes it visible, ensuring it is ready for interaction and rendering tests.

Test: `testTitleScreenDrawing`

- **Purpose**: Ensures that the title screen of the game is visible when the game is in the appropriate state.
- **Procedure**:
    - Uses `GuiActionRunner` to set the game panel's title screen state and triggers a repaint.
    - Uses `FrameFixture` to check that the panel designated as the "TitleScreenPanel" is visible.
- **Expected Outcome**: The title screen panel should be visible, confirming that the game's UI responds correctly to state changes and draws the title screen when expected.

Test: `testPlayerLifeDrawing`

- **Purpose**: Tests that the player's life display is visible and correctly updated during the game.
- **Procedure**:
  - Initializes a player with specific life values.
  - Sets the game state to `playState` to simulate the game running.
  - Triggers a repaint and checks if the panel showing the player's life is visible.
- **Expected Outcome**: The player's life panel should be visible, and it should accurately reflect the player's life state, indicating that the UI correctly represents game state changes related to the player's health.

Cleanup with `onTearDown()`

- **Method**: `onTearDown()`
- **Purpose**: Cleans up the GUI test environment after each test to prevent memory leaks and ensure that each test starts with a clean slate.
- **Details**: The cleanup method closes the window and releases any graphical resources, maintaining test isolation and performance.

.gitlab-ci.yml:

- **`build_job`**:
  - `stage: build`: This job is part of the `build` stage.
  - `script`: Contains commands that the CI server will execute for this job. Here, `mvn compile` compiles the source code of the project but does not package or install it. This step is crucial to ensure that the code is syntactically correct and all dependencies are resolved before moving to the testing phase.

- **`test_job`**:
  - `stage: test`: This job runs in the `test` stage, after the `build` stage.
  - `script`: Executes `mvn test`, which runs unit tests for the project using a suitable testing framework configured in the Maven project (like JUnit). This step is vital to ensure that the software behaves as expected.

Pom.xml:

Dependencies

Defines external libraries that the project depends on:

- **JUnit Jupiter Engine**: For running tests with JUnit 5.
- **JUnit**: For running tests with JUnit 4.
- **AssertJ Swing JUnit**: Provides facilities to write and test Java GUI components.

Build Configuration

- **Plugins**: Tools that provide additional capabilities to the build process:
    - **Maven Compiler Plugin**: Configures the Java compiler used by Maven.
        - `<source>` and `<target>`: Also set to "21" here, which should be corrected to match a valid JDK version.