Mehmet Arda Kutlu

Assignment 2 Report (YouTube link: https://youtu.be/wtylzdB4K0A)

Introduction

About the Game: This is the Only Level is a Flash game created in 2009. The game has a simple concept: the player controls a blue elephant in the game area which involves obstacles that restrict the player movement using the arrow keys and tries to reach the exit pipe. However, the exit is blocked by a door, and to open it, the player must press the button on the map. While doing this, the player must avoid spikes. Touching a spike causes the elephant to respawn at the start pipe, and it also increases the death count by one.

Although the game has only one level, it is divided into 30 different stages. Each stage has its own unique challenge or change in gameplay. After completing one stage by reaching the exit pipe, the player moves on to the next. The game finishes when all 30 stages are completed.

In this assignment, we were asked to recreate the game using five stages from the original version. Four of these stages were chosen for us, and I selected the fifth one. Below is a short explanation of each stage included in my version of the game:

Arrow Keys Are Required: This is the basic version of the game. The player moves the elephant using the arrow keys, and there are no changes or tricks.

Not Always Straight Forward: In this stage, the controls are reversed. Pressing the right arrow key moves the elephant to the left, and pressing the left key moves it to the right.

A Bit Bouncy Here: In this stage, the up-arrow key does not work. Instead, the elephant jumps automatically whenever it touches the surface of an obstacle.

Never Gonna Give You Up: Here, the player needs to press the button five times to open the door instead of just once. This adds an extra challenge.

Inbetween Gravitii (My choice): In this stage, gravity changes depending on the elephant's position on the x-axis. In some areas, gravity pulls the elephant downward, and in others, it pulls upward. These areas are shown by vertical stripes in different colors on the obstacles.

Implementation Details

The game is implemented using object-oriented approach. It has 4 additional classes except the main class. These classes are **Stage**, **Player**, **Map**, and **Game**. Each of these classes plays a significant role to ensure the integrity of the code.

Stage

Stage class stores the properties of the stages such as the gravity value, the clue and help message, and provides methods that allow other classes to interact with the stage properties.

Stage	
- stageNumber: int	The index of the stage.
- gravity: double	The gravity value at the stage.
- velocityX: double	The x distance that the player can take in one frame.
- velocityY: double	The y distance that the player can take in one frame.
- rightCode: int	Indicates which key moves player to right.
- leftCode: int	Indicates which key moves player to left.

- upCode: int	Indicates which key makes the player jump.
- clue: String	Clue string for the stage.
- help: String	Help string for the stage.
- color: Color	Color of the obstacles.
- gravityStripColor: Color	Color of the strips (only useful in stage 5).
- isHelpDisplaying: boolean	Whether the help message is displaying.
+ Stage (gravity: double, velocityX: double, velocityY: double, stageNumber: int, rightCode: int, leftCode: int, upCode: int, clue: String, help: String)	Constructor of the class.
+ getStageNumber(): int	Returns the stage number.
+ getGravity(): double	Returns the gravity value.
+ getVelocityX(): double	Returns the velocity in x direction.
+ getVelocityY(): double	Returns the velocity in y direction.
+ getKeyCodes(): int[]	Returns the key codes of the stage in the form [rightCode, leftCode, upCode]
+ getClue(): String	Returns the clue for the stage.
+ isHelpDisplaying(): boolean	Returns the displaying status of the help message.
+ setHelpDisplaying(isHelpDisplaying: boolean): void	Allows the modification of isHelpDisplaying from other classes.
+ getHelp(): String	Returns the help for the stage.
+ getColor(): Color	Returns the color of the obstacles.
+ setColor(color: Color): void	Sets a Color object for the obstacles.
+ getGravityStripColor(): Color	Returns the color of the gravity strips.
+ setGravityStripColor(gravityStripColor: Color): void	Sets the color of the gravity strips.

Figure 1: UML Diagram of the Stage Class

Player

Player class stores the information concerning the player. The information includes the player's coordinates, its dimensions and the facing direction of the player.

Player	
- x: double	X coordinate of the player.
- y: double	Y coordinate of the player.
- nextX: double	X coordinate of the player in the next frame.
- nextY: double	Y coordinate of the player in the next frame.
- width: double	Width of the elephant character.
- height: double	Height of the elephant character.
- velocityY: double	Velocity of the player in y direction.
- facingDirection: char	The facing direction of the elephant character.
+ Player(x: double, y: double)	Constructor of the class.
+ setX(x: double): void	Sets x position of the player.
+ setY(y: double): void	Sets y position of the player.
+ getX(): double	Returns x position of the player.
+ getY(): double	Returns y position of the player.
+ getNextX(): double	Returns x position of the player in the next frame.
+ setNextX(nextX: double): void	Sets x position of the player in the next frame.
+ getNextY(): double	Returns y position of the player in the next frame.
+ setNextY(nextY: double): void	Sets y position of the player in the next frame.
+ getVelocityY(): double	Returns the velocity of the player in y direction.
+ setVelocityY(velocityY: double): void	Sets the velocity of the player in y direction.

+ getWidth(): double	Returns the height of the elephant character.
+ getHeight(): double	Returns the width of the elephant character.
+ getFacingDirection(): char	Returns facing direction of the elephant character.
+ setFacingDirection(direction: char): void	Sets facing direction of the elephant character.
+ respawn(spawnPoint: int[]): void	Respawns the player.
+ draw(facingDirection: char): void	Draws the player based on its facing direction.

Figure 2: UML Diagram of the Player Class

Map

Map class contains the elements of the game area. The visuals of the game are primarily related to this class. This class provides methods that move the player in the map using gravity and collision mechanisms based on the stage properties, draw the map elements and create the animations such as door opening and button pressing.

Map	
- stage: Stage	Stage object of the map.
- player: Player	Player object of the map.
- collidingObstacles: ArrayList<int[]>	Stores the colliding obstacles in each frame.
- obstacles: int[][]	List of the coordinates of the obstacles in the format [xLeftDown, yLeftDown, xRightUp, yRightUp]
- button: int[]	Coordinates of the button in the same format.
- buttonColor: Color	Color of the button.
- buttonFloor: int[]	Coordinates of the button floor.
- buttonFloorColor: Color	Color of the button floor.
- startPipe: int[][]	Coordinate list of the start pipe that consists of two rectangles.
- exitPipe: int[][]	Coordinate list of the exit pipe that consists of two rectangles.
- pipeColor: Color	Color of the pipes.
- spikes: int[][]	Coordinate list of the spike areas.
- door: int[]	Coordinates of the door.
- doorColor: Color	Color of the door.
- buttonPressNum: int	How many times is the button pressed.
- isButtonPressing: boolean	Whether the button is being pressed.
- isDoorOpen: boolean	Whether the player pressed to button sufficient times to open the door.
- isSpikeHit: boolean	Whether the player hit a spike.
+ Map(stage: Stage, player: Player)	Constructor of the class.
- xCenter(coordinates: int[]): double	Finds the x-center of the rectangular obstacle.
- yCenter(coordinates: int[]): double	Finds the y-center of the rectangular obstacle.
- halfWidth(coordinates: int[]): double	Finds the halfwidth of the rectangular obstacle.
- halfHeigth(coordinates: int[]): double	Finds the halfheight of the rectangular obstacle.
- isTouchingGround(x: double, y: double): boolean	Checks if the player touches the surface of an obstacle from above.
- istouchingTop(x: double, y: double): boolean	Checks if the player touches the surface of an obstacle from below. Only used in stage 5.
- gravity(): void	Implements the gravity mechanism by updating velocityY.
- reverseGravity(): void	Reverses gravity mechanism for stage 5.
+ updateXCoordinate(direction: char): void	Calculates the x coordinate of the successive frame.
+ updateYCoordinate(direction: char): void	Calculates the y coordinate of the successive frame. Handles the jump mechanism.

- rightBoundary(obstacle: int[]): double	Finds the left collision boundary of the rectangular obstacle.
- leftBoundary(obstacle: int[]): double	Finds the right collision boundary of the rectangular obstacle.
- topBoundary(obstacle: int[]): double	Finds the upper collision boundary of the rectangular obstacle.
- bottomBoundary(obstacle: int[]): double	Finds the lower collision boundary of the rectangular obstacle.
- checkCollision(): void	Detects the collision with obstacles and temporarily saves these obstacles to **collidingObstacles**.
- spikeCollision(): Boolean	Checks the collision with spikes.
- twoDCollisionType(obstacle: int[]): String	Checks if the collision is happened from sides or form top/bottom surface.
- twoDCollision(obstacle: int[]): void	Handles the collision mechanism when the player is moving in both directions.
- horizontalCollision(obstacle: int[]): void	Handles the collision mechanism when the player is only moving in x direction.
- verticalCollision(obstacle: int[]): void	Handles the collision mechanism when the player is only moving in y direction.
+ movePlayer(): void	Moves the player in the map. Harnesses the collision methods.
+ changeState(): boolean	Checks if the player reached the exit.
+ pressButton(): boolean	Presses the button and increases **buttonPressNum**.
- doorCheck(): void	Handles the door opening animation.
+ restartStage(): void	Restarts the stage.
+ getIsSpikeHit(): boolean	Returns **isSpikeHit**.
+ getStage(): Stage	Returns the stage of the map.
+ getPlayer(): Player	Returns the player.
+ draw(): void	Draws the components of the map.

Figure 3: UML Diagram of the Map Class

Game

Game class is the engine of the game. It provides methods that are responsible for running the game with the help of the other classes, implementing the control and information center below the map and receiving and processing the user input.

Game	
- stageIndex: int	Indicates the current stage of the game.
- stages: ArrayList<Stage>	Stores all the stages of the game.
- deathNumber: int	Tracks the total number of spike hits and restarts.
- gameTime: double	Elapsed time in the game.
- pauseDuration: int	The duration between two successive frames.
- milliseconds: int	Split-second fraction of the time.
- seconds: int	Second fraction of the time.
- minutes: int	Minute fraction of the time.
- startTime: double	The beginning time of the game.
- resetTime: double	The last reset time of the game.
- resetGame: boolean	Indicates the game reset status.
- isClickingRestart: boolean	A flag to prevent constant presses to restart button.
- isClickingReset: boolean	A flag to prevent constant presses to reset button.
- isEmptyPressing: boolean	A flag to prevent triggering of the buttons by hovering the cursor while the mouse is being pressed.

+ Game(stages: ArrayList<Stage>)	Constructor of the class.
+ play(map: Map): void	Starts and runs the game.
- handleInput(map: Map): void	Handles player input and interactions (keyboard and mouse).
- isResetClicked(): void	Checks if the player pressed the mouse when the cursor is on the reset button.
- isRestartClicked(): boolean	Checks if the player pressed the mouse when the cursor is on the restart button.
- isHelpClicked(): boolean	Checks if the player pressed the mouse when the cursor is on the help button.
- isEmptyClicked(): void	Checks if the cursor is on one of the buttons.
+ setStartTime(startTime: double): void	Sets the starting time of the game.
+ getStageIndex(): int	Returns the index of the current stage.
+ getCurrentStage(): Stage	Returns the current stage of the game.
+ getDeathNumber(): int	Returns how many times did the player die.
+ getMinutes(): int	Returns the minute fraction of the time.
+ getSeconds(): int	Returns the second fraction of the time.
+ getMilliseconds(): int	Returns the split-second fraction of the time.

Figure 4: UML Diagram of the Game Class

Main Class

In the main class, all five stages are first created and then stored in an ArrayList. Each stage is assigned a random color using the Random class. The game is controlled using two nested while loops. The outer loop is responsible for creating the game object and allowing the player to either reset the game or restart after completing all stages. The inner loop runs the game by calling the play method, which progresses to the next stage once the current one is completed. The play method ends either when the player successfully finishes a stage or chooses to reset the game. When all stages are completed, the inner loop exits, and the player is taken to the end-game screen within the outer loop. On this screen, the player has the option to restart the game by pressing the "A" key or to exit by pressing the "Q" key.

Implementation of the Reset Mechanism: As mentioned above, the play method can end in one of two ways: either the player completes the stage, or he/she presses the reset button. In the first case, the **stageIndex** is incremented by one, allowing the game to move on to the next stage. In the second case, the **stageIndex** is set to zero, indicating that the player has chosen to restart the game. This distinction is used to determine the player's action after the play method ends. If the **stageIndex** is zero, it means the player pressed the reset button. As a result, the inner loop breaks, and a new game begins by creating a fresh game object. Before resetting, a green ***"RESETTING THE GAME…"*** banner is displayed for 2 seconds. If the **stageIndex** is not zero, the inner loop continues, and the player proceeds to the next stage.

```
    mainGame.play(map);
    // If the play method ends when the stage index is 0, the player must have restarted the game
    // Inner while loop breaks, and the game continues from the beginning
    if(mainGame.getStageIndex() == 0){
        break;
    }
}
```

Figure 5: The code snippet that controls the reset mechanism

More Information About the 5th Stage

We were asked to select the fifth stage from the original game. For my version, I chose the 21st stage titled *"Inbetween gravitii"* because I found its gameplay mechanics unique and visually interesting, especially with the colorful stripes on the map.

The key feature of this stage is the gravity reversal mechanic. The direction of gravity changes based on the player's **x-coordinate**. The map is divided into a number of vertical intervals of equal width, and gravity reverses whenever the player crosses from one interval into the next.

This mechanic increases the difficulty, as spikes on the ceiling—which are usually not a threat—now become just as dangerous as those on the ground. To help players adapt to this challenge, each gravity zone is visually marked with stripes of different colors. This way, players can better plan their movements and react to gravity shifts more effectively.

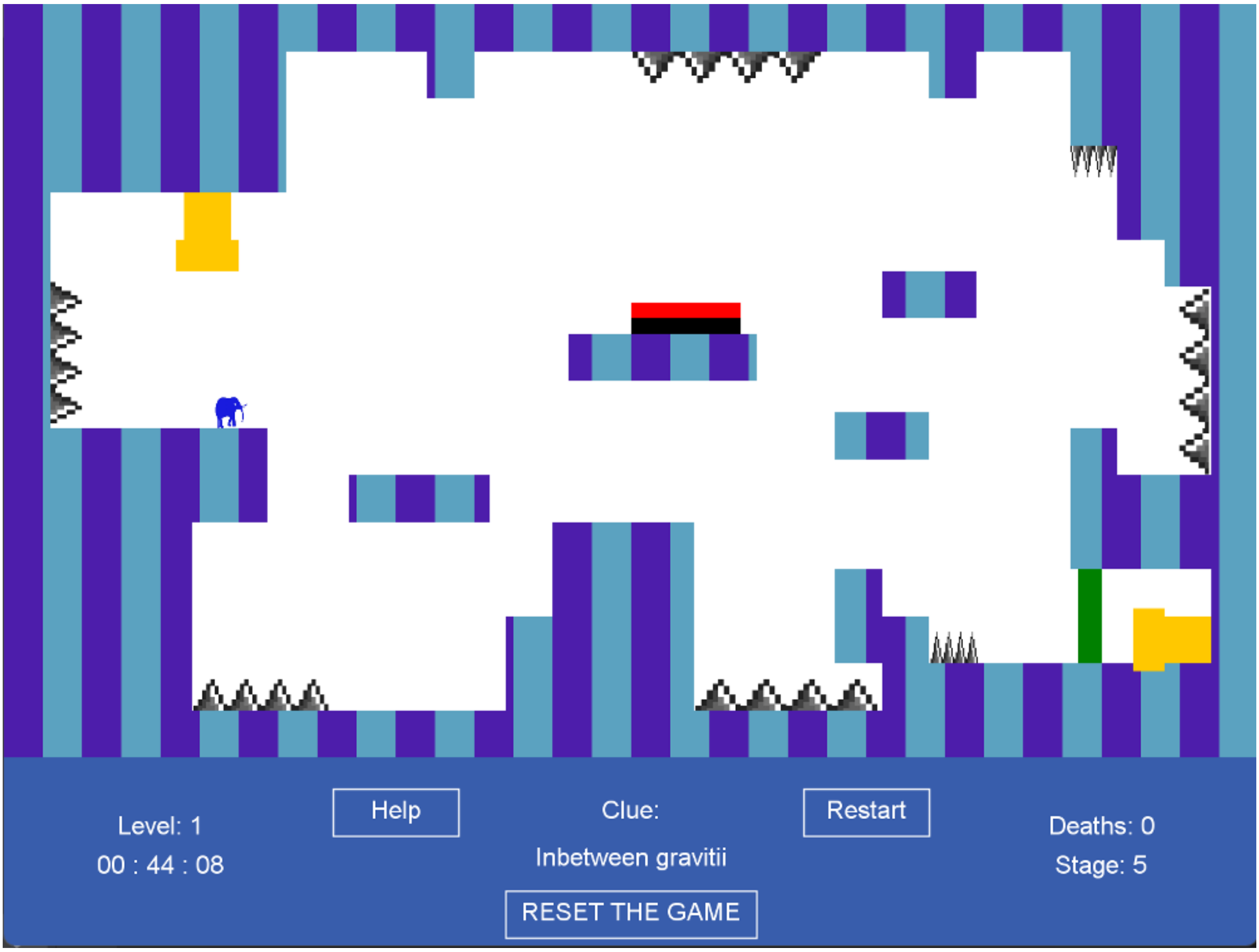

Figure 6: Appearance of the stage 5

Mechanics of the Game

Movement and Collisions: The player's interaction with obstacles, spikes, and the door forms a core part of the game mechanics. When handling movement and collision in each frame, the game does not directly update the player's x and y coordinates. Instead, it first calculates potential new positions using nextX and nextY. These values are updated as if there are no obstacles, using the **updateX** and **updateY** methods.

Collision detection and handling are managed in the Map class through the **movePlayer** method. This method checks for possible collisions from different directions by calling **horizontalCollision**, **verticalCollision**, and **twoDCollision**. If a collision is detected, it adjusts the **nextX** and **nextY** values accordingly and then updates the player's actual position using the **setX** and **setY** methods in the Player class.

Respawning: If the player touches a spike, he/she is instantly killed and respawns at the starting pipe. Each death increases the death counter by one. Spike collisions are detected by the **spikeCollision** method in the Map class. The player can also manually restart the level by clicking the restart button below the map. If the **isRestartClicked** method in the Game class confirms the button press, the respawn method in the Player class is triggered. As with spike deaths, using the restart button also increments the death count by one.

```java
public void respawn(int[] spawnPoint){  15 usages
    this.x = spawnPoint[0];
    this.y = spawnPoint[1];
    this.nextX = this.x;
    this.nextY = this.y;
    // Resets the velocityY variable to ensure that the elephant respawns at rest
    this.velocityY = 0;
}
```

Figure 7: **respawn** method

Passing the Stage: A stage is completed when the player reaches the exit pipe. The **changeStage** method in the Map class checks whether the player has reached this point. If so, the play method in the main class returns, and the player advances to the next stage by incrementing **stageIndex** by one. Before transitioning, a green banner displaying the message ***"You passed the stage But is the level over?!"*** appears for 2 seconds.

The Clue and Help Texts: Each stage includes a unique clue and help message. The clue provides a subtle hint about the stage, while the help message offers a more direct explanation. At the beginning of each stage, the clue is displayed by default below the map. If the player clicks the help button, and the **isHelpPressed** method in the Game class confirms the action, the clue will be replaced by the help text. If the player restarts the stage or resets the game, the clue message is shown again.

Displaying the Time: The time counter is displayed at the bottom-left corner of the game screen. The elapsed time is tracked using the variables **startTime**, **currentTime**, and **resetTime**, along with the *System.currentTimeMillis()* method. At the beginning of each stage, **startTime** is set to the current system time. During each frame, the elapsed time is calculated by subtracting **startTime** from **currentTime**.

However, when the player restarts a stage, the timer should not reset to zero. Instead, it must continue counting from the total time accumulated before the restart. For example, if stage 3 ends when the timer shows *3 : 14 : 89* (minutes : seconds : split-seconds), and the player restarts stage 4, the timer should continue from *3 : 14 : 89*, not from zero.

To achieve this behavior, resetTime is updated by adding the duration of each completed stage whenever a stage is finished. The final elapsed game time, referred to as **gameTime**, is calculated using the formula: **gameTime = resetTime + (currentTime - startTime)**

Since **gameTime** is measured in milliseconds, it needs to be converted into a more readable format: minutes, seconds, and split-seconds. This conversion is done by extracting the respective time components and storing them in the variables **minutes**, **seconds**, and **milliseconds**.

```java
gameTime = resetTime + currentTime - startTime;
// Converting raw data in milliseconds to more useful m : s : ms format
milliseconds = ((int) gameTime % 1000) / 10;
seconds = (int)(gameTime % 60000) / 1000;
minutes = (int) gameTime / 60000;
```

Figure 8: Extracting the time components

```java
StdDraw.text( x: 100,  y: 50, String.format("%02d : %02d : %02d",minutes,seconds,milliseconds));//timerText
```

Figure 9: Displaying the time counter using the string format method

Finishing the Game: When the player completes the final stage, the **stageIndex** becomes equal to the size of the stages ArrayList. Since the condition for the outer while loop is **stageIndex < stages.size()**, this condition is no longer met, and the inner loop exits. At this point, the end-game screen is displayed, featuring a green banner that reads: **"CONGRATULATIONS YOU FINISHED THE LEVEL PRESS A TO PLAY AGAIN"**.

This end screen is controlled by an infinite while loop, which waits for the player's input. The loop is broken if the player presses the **"A"** key, in which case a new Game object is created and the game restarts from the beginning. Alternatively, if the player presses the **"Q"** key, the program terminates by calling the **System.exit()** method.

Screenshots from the Game

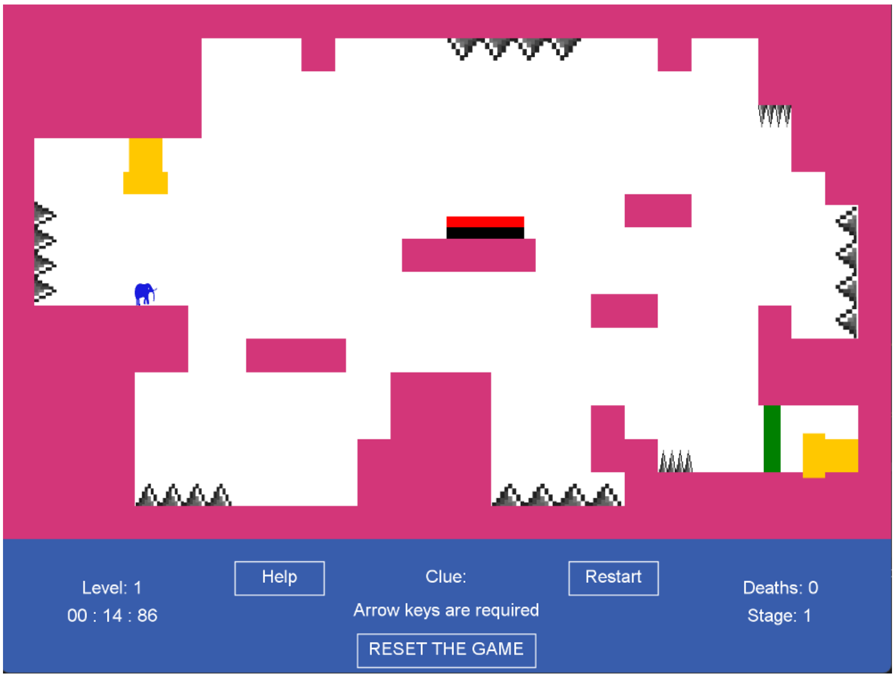

Figure 10: Game Instance

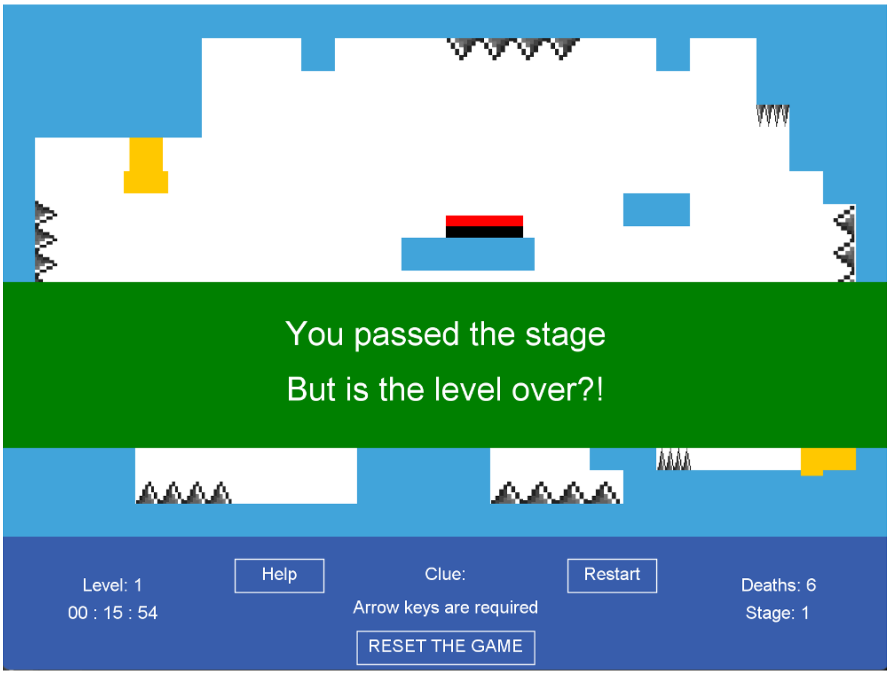

Figure 11: Passing Stage Screen

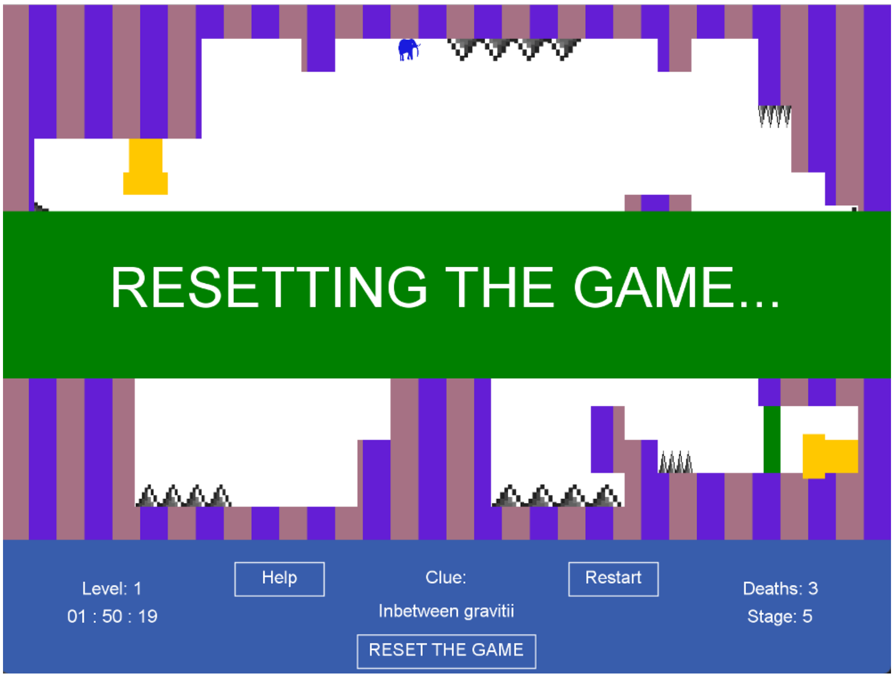

Figure 12: Resetting Screen

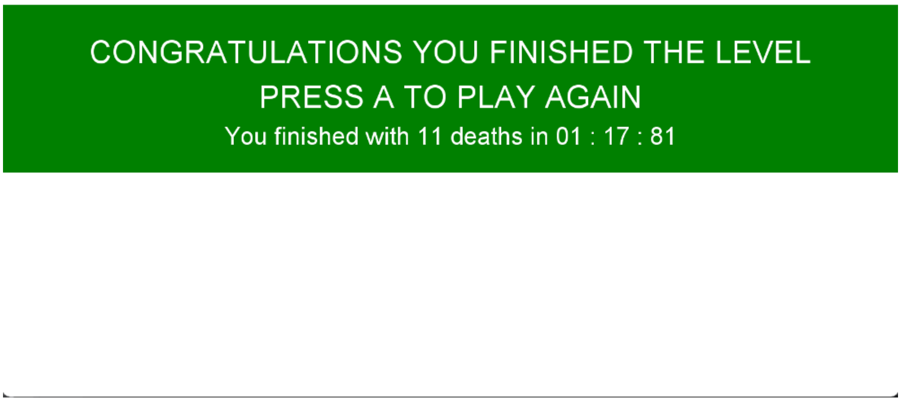

Figure 13: End Game Screen