

**MARMARA UNIVERSITY**

**FACULTY OF ENGINEERING**

**CSE1242 Computer Programming II, Spring 2022**

|  |
| --- |
| **CSE 1242, Spring 2022: TERM PROJECT**  **PIPE PUZZLE** |

**Authors:**Aksanur Konuk  
…

**Title:** CSE1242, Spring 2022: Term Project

**Project Name:** Pipe Puzzle

**Date Submitted:** May 09, 2022

**Submitted to:** …

Table of Contents

1. PROBLEM DEFINITION…………………………………………………………………..3

TILE DETAILS…………………………………………………………….…………..3

2. IMPLEMENTATION DETAILS…………………………………………………..………..4

PipePuzzleClass…………………………………………………………….………….4

TileClass……………………………………………………………………………..…6

CheckTiles Class……………………………………………………………...………..7

CheckSwap Class…………………………………………………………………..…..8

SwapTiles Class……………………………………………………………………..….9

CheckLevel Class……………………………………………………………….…….10

AnimateBall Class…………………………………………………………….………11

Additionally……………………………………………………………………..…….11

Contribution by Teammates……………………………………………….…………..11

3. TEST CASES……………………………………………………………………...……….12

1. PROBLEM DEFINITION

This project aims to implement a game with JavaFX where the player is asked to form a path by moving tiles which are in forms of different pipes, so that the ball can roll through its way. When the player successfully forms a path with the given pipes, ball starts rolling from where it’s located at the start to the destination tile at the end by following the built path. The main structure of the project is formed by different images that represents each tile which are determined with a text input file.

About the gameplay, player is asked to form a path by moving the tiles. While performing this act, some rules must not be forgotten: the starter and end tiles cannot be moved. These tiles can be determined by their difference in shape and colour. Both have only one direction available and starter tiles are blue, meanwhile the end tiles are red. Other blue tiles other than starter tile are also known to be static type, which again determines that the tile cannot be moved. Moveable tiles are specified with the colour yellow and can be in any shape such as straight or curved. Direction of any tile can differ. All kinds of pipes can be used during the forming of the path, although it’s not a must to use all the tiles given by the level. Tiles only can move horizontally and vertically with one row or one column in each move.

## **TILE DETAILS**

|  |  |
| --- | --- |
|  | **Starter Tiles**  There are two types: ‘Horizontal’ and ‘Vertical’. Starter tiles are incapable of moving. Indicates where the path starts. |
|  | **End Tiles**  There are two types: ‘Horizontal’ and ‘Vertical’. End tiles are incapable of moving. Indicates where the path ends. |
|  | **Empty Tiles**  There are two options: ‘Free’ and ‘None’. Free tiles determine the locations that non-static tiles can move to. ‘Empty-None’ tiles can move; however, they cannot take part in forming a path. |
|  | **PipeStatic Tiles**  There are four options: ‘Vertical’, ‘Horizontal’, ’01’, ‘10’, ’10’, and ‘11’. Numbers indicates the direction of the curve. Pipe static tiles are incapable of moving. However, they can take part in forming the path. |
|  | **Pipe Tiles**  There are four options: ‘Vertical’, ‘Horizontal’, ’01’, ‘10’, ’10’, and ‘11’.  Numbers indicates the direction of the curve. Pipe tiles can move, and they can also take part in forming the path. |

2. IMPLEMENTATION DETAILS

## **PipePuzzle Class**

|  |  |
| --- | --- |
| **PipePuzzle** | |
| -  -  -  +  +  +  +  + | nmbOfMoves: Text  levelCompleted: boolean  tiles: Tile[][]  startRow: int startCol: int  move: int  levelNumber: int  ball: Circle |
| +  +  +  +  + | PipePuzzle()  main (args: String[]): void  start(stage: Stage): void  readInputFile(lineParts: String[], pane: GridPane): GridPane  createBall():Circle |

* **PipePuzzle** is the main class of the project. It includes the main and start methods and few more implementations (which are explained down below) as well.
* **nmbOfMoves** keeps the Text field of the number of the moves, which is printed on the pane.
* **levelCompleted** indicates whether the level is completed by the player or not; has the initial value of true but is set to false right after a new level is displayed.
* **tiles** array keeps the information of what kind of tile remains in the specific row and column.
* **startRow** holds the int value of the position of where the starter tile is located by its row.
* **startCol** holds the int value of the position of where the starter tile is located by its column.
* **move** variable shows how many moves are done by the player in each level. This number is updated in each swap and displayed on the screen simultaneously.
* **levelNumber** variable initializes where the game starts, which is the first level every time the program is run. This number is added by one whenever the level is completed, so the next level can be displayed.
* **ball** is a Circle type object which is placed on top of the starter tile and performs an animation of roll along the path after the level is completed by the player.
* **PipePuzzle** constructor with no arguments has no implementation inside the body.
* **main** method launches the application for JavaFX to be run.
* **start** method is the essential method for JavaFX projects. In this method:
* pane of *GridPane* type is created to provide an opportunity for player to play the game. Some essentials such as “*Number of Moves:* “ is added to the pane and the ball is created. Later, with the help of a scanner, the input files are read each time and related lines are taken to form the pane. While performing this act, some cases are taken into consideration such as skipping the empty lines of the input file. After each line, the *readInputFile* method is invoked to fill the related cell of the pane with the matching tile. This is performed until the input file has no line left and new levels are displayed after the current level is completed if there are input files left un-played. After successfully forming the pane, *getRowCol* method from the *CheckSwap* class is invoked to enable the mouse as an input for player to start playing. Finally, scene and stage are set and ready to be shown.
* **readInputFile** method is responsible for taking the tile information from the given inputs. As the line is read word by word from the array it was assigned to, ‘*equal’* controls match the right type of tile with the given input:
* The first word determines where to place the given tile. Second word determines the main type of the tile and the third word either specializes the direction or indicates whether the tile is ‘*empty’* or ‘*none’*. Meanwhile the tiles are being added to pane, the information of these tiles are also added to tiles array by defining them based on the constructor in the Tile class: id, direction and moveability information are first set in this method.
* **createBall** method creates a ball of *Circle* type with the radius of 20 and colour of dark salute blue and whenever it’s invoked, returns an object with these properties.

## **Tile Class**

|  |  |
| --- | --- |
| **Tile** | |
| -  -  -  -  -  -  - | id: String  up: boolean  down: boolean  right: boolean  left: boolean  moveable: boolean  image: Image |
| +  +  +  +  +  +  +  +  +  + | Tile()  Tile(id: String, up: boolean, down: boolean, right: boolean, left: boolean, moveable: boolean,  pane: GridPane)  addPane(pane: GridPane, image: Image, row: int, column: int): void  getImage(): Image  getId(): String  isUp(): boolean  isDown(): boolean  isRight(): boolean  isLeft(): boolean  isMoveable(): boolean |

* Each tile on the pane is formed from the **Tile** class. Each tile should have an **id**, an **up**, a **right**, a **left,** a **moveable,** an **image**.
* The data field **id** represents the type of tiles. The data field **up**, **down**, **right**, **left** represents the available directions of the pipe. The data field **moveable** represents whether tile can move or not. The data field **image** contains images of  each composed tile.
* **Tile** can be created with one of two constructors.
* If **Tile** is created with a second constructor, the data fields assigned according to coming information with parameters.
* There are getter methods of id, image, up, down, right, left, moveable. **getId** returns String value; **getImage** getter returns Image; **isUp**, **isDown**, **isRight**, **isLeft, isMoveable** return boolean value.
* **addPane** method adds the tiles with the given information to the pane. This method is mainly responsible for forming the visual representation of each level. After the tiles are correctly added, ball is also placed on where the starter tile is.

## **CheckTiles Class**

|  |  |
| --- | --- |
| **CheckTiles** | |
| -  -  -  -  - | tileRow: int  tileCol: int  freeRow: int  freeCol: int  tiles: Tile[][] |
| +  +  + | CheckTiles()  CheckTiles(tiles: Tile[][], freeRow: int, freeCol: int, tileRow: int, tileCol: int)  checkTiles(): boolean |

* **CheckTiles** class checks the specific properties of the two tiles that are chosen by the player.
* **tileRow** keeps the int value of the row of the tile which is previously selected first by the player with *MousePressed*. **tileCol** keeps the int value of the column of the same tile.
* **freeRow** keeps the int value of the row of the tile which is previously selected second by the player with *MouseReleased*. **freeCol** keeps the int value of the column of the same tile.
* **tiles** array keeps the information of what kind of tile remains in the specific row and column. While the pane is responsible for the visuality, tiles array works in synchronization.
* **CheckTiles with no arguments** constructor doesn’t implement anything. Meanwhile the second **CheckTiles constructor with arguments** matches the values of the variables.
* **checkTiles** method performs the following check: The previously taken values of row and column of the two cells are keys to determine the type of the tile situated in the cell. This method checks whether the first tile is ‘*moveable’* and second is the type of a ‘*free’* tile.

## **CheckSwap Class**

|  |  |
| --- | --- |
| **CheckSwap** | |
| -  -  -  -  -  -  -  -  -  - | nmbOfMoves: Text  pane: GridPane  tiles: Tile[][]  stage: Stage  tileDeclared: boolean  freeDeclared: boolean  tileRow: int  tileCol: int  freeRow: int  freeCol: int |
| +  +  +  +  + +  +  + | CheckSwap()  CheckSwap(pane: GridPane, tiles: Tile[][], stage: Stage, nmbOfMoves: Text)  getRowCol(): void  checkDeclared(): void  checkAvailable(): void  printMoveNumber(pane: GridPane): void  setFreeDeclared(freeDeclared: boolean): void  setTileDeclared(tileDeclared: boolean): void |

* **CheckSwap** class controls whether tiles chosen by mouse are available for swapping.
* **nmbOfMoves** is the text of information about player’s number of moves in a level.
* **pane** is the pane that tiles and text nmbOfMoves are placed.
* **tiles** contains the reference of **Tile class** of each tile on the pane.
* **stage** hosts a scene and scene graph.
* **tileDeclared** indicates whether the mouse was pressed and whether the row and column information of the pressed tile was received.
* **freeDeclared** indicates whether the mouse released and when the mouse has been released, whether the row and column information of tile was received.
* **tileRow** hold the row value of selected tile’s by pressing mouse button and **tileCol** hold column value of this tile.
* **freeRow** holds the row value of selected tile’s by releasing the mouse button and **freeCol** holds the column value of this tile.
* **CheckSwap** can be created with one of two constructors. If **CheckSwap** is created with second constructor, the data fields assign according to coming information with parameters.
* **getRowCol** method takes the information of selected tiles. When the mouse button is pressed, the information of x and y value of the point is taken, and the row and column information are calculated by dividing the coordinates by the tile’s image size.
* **tiledeclared** returns to true and call the **checkDeclared** method.  After pressed, when the mouse is released, the information of second selected tile is reached in the same way.
* **freedeclared** returns to true and call the **checkDeclared** method.
* **checkDeclared** method checks whether the row and column information of two selected tiles is obtained and whether these values of row and column are available to swap.
* **checkAvailable** method controls whether the location of selected tiles is available. Tiles must be right near each other by horizontally or vertically. Crossed swap is not allowed.
* **printMoveNumber** method increases the number of moves and ensures that the increased number is shown on the pane.
* There are setter methods: **setFreeDeclared**, **setTileDeclared**.

## **SwapTiles Class**

|  |  |
| --- | --- |
| **SwapTiles** | |
| -  -  -  -  -  -  - | tileRow: int  tileCol: int  freeRow: int  freeCol: int  tiles: Tile[][]  pane: GridPane  primaryStage: Stage |
| +  +  + | SwapTiles()  SwapTiles(pane: GridPane, tiles: Tile[][], primaryStage: Stage, freeRow: int, freeCol: int,  tileRow: int, tileCol: int)  swap(): GridPane |

* **SwapTiles** class is the actual class that keeps the synchronization by switching the tile information in swapped cells.
* **tileRow** keeps the int value of the row of the tile which is previously selected first by the player with *MousePressed*. **tileCol** keeps the int value of the column of the same tile.
* **freeRow** keeps the int value of the row of the tile which is previously selected second by the player with *MouseReleased*. **freeCol** keeps the int value of the column of the same tile.
* **tiles** array keeps the information of what kind of tile remains in the specific row and column.
* **pane** and **primaryStage** keeps the actual reference of the main pane and stage.
* **swap** method performs the swap by following these steps:
* Firstly, a smooth animation occurs whenever player performs a successful swap. This animation is done by forming a path between the two tiles. Moreover, for further usage the tiles array needs to be updated after each swap. So, as soon as the animation ends, the information in chosen cells are also swapped. Right after these acts are done, updated pane is returned.

## **CheckLevel Class**

|  |  |
| --- | --- |
| **CheckLevel** | |
| -  -  -  -  -  - | levelCompleted: boolean  pathRow: ArrayList<Integer>  pathCol: ArrayList<Integer>  tiles: Tile[][]  stage: Stage  pane: GridPane |
| +  +  +  +  + +  +  +  + | CheckLevel()  CheckLevel()  control(): boolean  getPathRow(): ArrayList<Integer>  setPathRow(): void  getPathCol(): ArrayList<Integer>  setPathCol(): void  isLevelCompleted(): boolean  setLevelCompleted(): void |

* The **CheckLevel** class controls whether the path is complete or not.
* **levelCompleted** represents whether the level is completed or not.
* **pathRow** ArrayList holds the row values of the path. **pathCol** ArrayList holds the column values of the path. These row and column values will be used for animation.
* **tiles** contains reference of **Tile** class of each tile on the pane. The data field **stage** hosts a scene and scene graph. The data field **pane** is pane that tiles and text nmbOfMoves are placed.
* **CheckLevel** can be created with one of two constructors.
* If **CheckLevel** is created with second constructor, the data fields assign according to coming information with parameters.
* The **control** method controls whether the path is completed or not. The control method is invoked every time the tiles are successfully swapped. The basic idea of this control is:
* Firstly, enter a loop and control the type of tiles, determine the direction of tile (direction is the indicated direction of coming path). Detect the next tile according to the type and direction of current tiles.
* Check whether there is the harmony of the current and next tiles(the direction of tiles is right). If they are compatible, keep the row and column value  at the **pathRow** and **pathCol**. Then update the coordinate of the current tile. If they are not, break the loop and clear the **pathRow** and **pathCol.**
* Update direction information (Direction is determined according to the previous column , row and current column, row.)
* The loop continues until the right path is reached or incompatible tiles are encountered.
* There are getters and setters: **getPathRow**, **getPathCol** , **isLevelCompleted** ; **setPathRow**, **setPathCol**, **setLevelCompleted**.

## **AnimateBall Class**

|  |  |
| --- | --- |
| **AnimateBall** | |
| -  - | pathRow: ArrayList<Integer>  pathCol: ArrayList<Integer> |
| +  +  + | AnimateBall()  AnimateBall(pathRow: ArrayList<Integer>, pathCol: ArrayList<Integer>)  animateBall(pane: GridPane, primaryStage: Stage): void |

* **AnimateBall** class helps the ball to roll through the formed path.
* **pathRow** ArrayList keeps the rows of the tiles that forms the path in order. Same way:
* **pathCol** ArrayList keeps the columns of the tiles that forms the path in order.
* **AnimateBall constructor with no arguments** doesn’t implements anything. Meanwhile the second **AnimateBall constructor with arguments** matches the values of the variables.
* **animateBall** method performs the animation of the ball when the level is completed:
* To form the animation path, the values stored in *ArrayList* are taken with their next value to form horizontal and vertical lines between the tiles that form the path. These lines between each tile are connected to create the main path. Later, this path is set alongside with the ball that will perform the animation. As soon as the animation ends, start method is invoked to upload the next level on screen.

## **Additionally**

All the expected implementations are done. There are no parts left incomplete or runs different than it should have.

During the implementation, some of the notable difficulties we encountered were figuring out how to perform the animation and coming out with an idea of a method that can check any level whether it’s completed or not. But eventually all these problems were solved whether with a brainstorm or with the help of subject related resources.

In addition, the program is adjusted to read as many levels as possible if the next input file can be found. Thus, there will be no level restriction and users can play the desired number of levels if the input files are provided.

## **Contribution by Teammates**

In general, we thought together about how to start and continue our game project and determine a plan. When we encountered problems with our own codes, we asked each other for help. We worked together in all codes. However, one of us was more interested in some parts of the project. To show the proper output on the panel, we worked together. Duru Baştunalı was more interested about the swap tiles, and animation of the ball after the level is completed. Aksanur Konuk was more interested about control of whether this level is completed or not.

Text

Description automatically generatedGraphical user interface, application

Description automatically generated3. TEST CASES

**Graphical user interface, application

Description automatically generated**

Graphical user interface, application

Description automatically generatedGraphical user interface, application

Description automatically generatedWhen the photos are examined, it is proven that the game is successful at  obtaining the correct view according to the input file. By looking at the third photo, we can be sure that the game can swap the tiles correctly and increase the move’s number.   
  
  
A picture containing graphical user interface

Description automatically generatedGraphical user interface

Description automatically generated with medium confidence

In these photos, we can show that the game can control whether a level is completed or not. After the game detects that the player finds the right way for the ball, it starts to animate the ball. Additionally, it can be noticed that the game can drag the tiles correctly in the fifth photo*.*

A picture containing application

Description automatically generated

In the last photo, we can see that the game passes the next level, After ball animation is finished. It should be drawn attention that the value of move number is reset by  the game, when passing the next