CSE 1242 - COMPUTER PROGRAMMING II TERM PROJECT

PIPE BALL MAZE

Sena Ektiricioğlu – 150120047

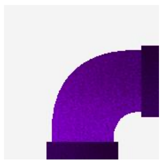
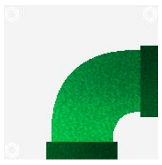
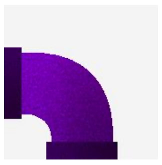
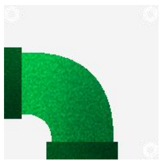
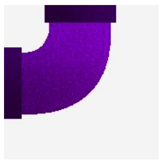
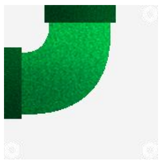
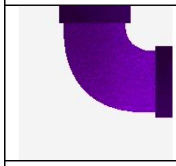
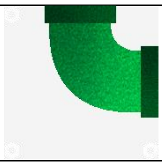
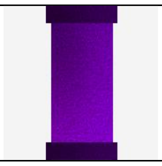
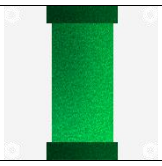
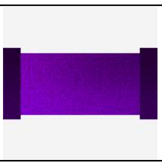
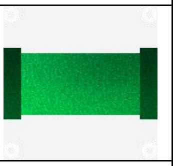
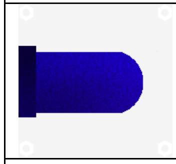
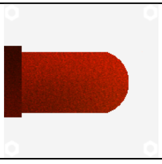
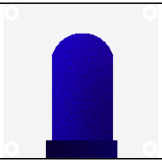
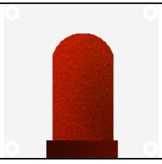
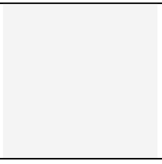
Beyza Nur Kaya - 150120077

CSE1242 Computer Programming II, Spring 2022

Date Submitted: May 09, 2022

PROBLEM DEFINITION:

The game consists of 16 tiles in the gameboard total, some of which can move and some that cannot. The aim of the game is to create a path by connecting the pipes on these tiles and make the ball which is on the first tile slide.

					
Curve1	Curve1 Static	Curve2	Curve2 Static	Curve3	Curve3 Static
					
Curve4	Curve4Static	Vertical Linear	Vertical Linear Static	Horizontal Linear	Horizontal Linear Static
					
Starter Horizontal	End Horizontal	Starter Vertical	End Vertical	Empty	EmptyFree

Ball

There are 7 types of tiles in the game:

Starter: It is the starting tile of the ball's path. It cannot be moved, and every game board has one. Also, starter tiles' pipe can be vertical or horizontal.

End: It is the end tile of the ball's path. It cannot be moved, and every game board has one. Also, end tiles' pipe can be vertical or horizontal.

Empty Free: Empty free tiles are tiles without pipe. They exist to be dragged other movable tiles to where this tile is located. They can also be thought of as spaces.

Empty: Empty tiles are tiles without pipe. These tiles can be dragged to where the empty free tiles are.

Pipe Static: Pipe static tiles cannot be moved, which means they are static. Pipe inside these tiles can be vertical or horizontal.

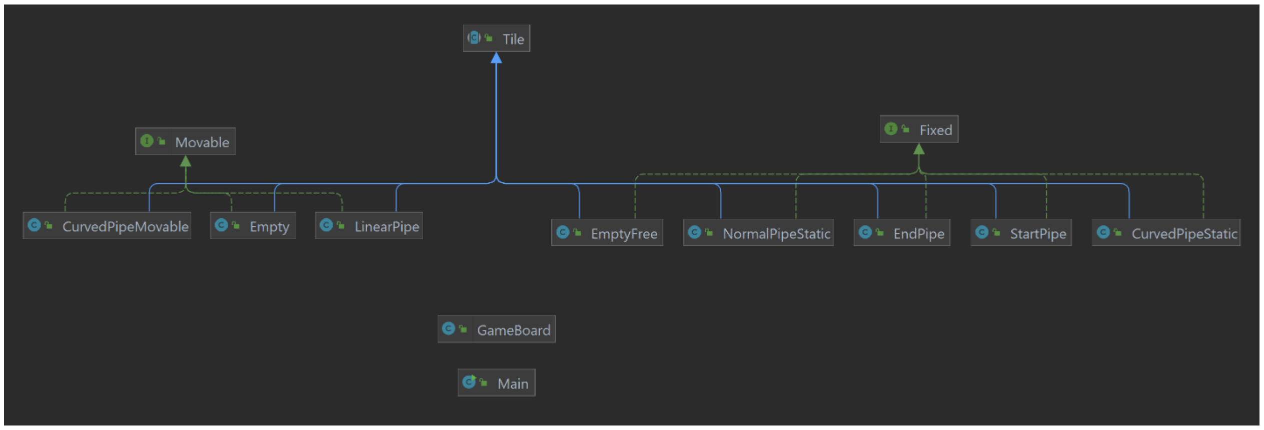
Pipe: Pipe tiles can be dragged to where the empty free tiles are. Pipe inside these tiles can be vertical or horizontal.

Curved Pipe: Curved pipe tiles divided into static ones and movable ones. Static curved pipe tiles cannot move, in contrast, movable curved pipe tiles can be dragged to empty free tiles' location. Curved pipes shape types are represented by the values of "00, 01, 10 and 11".

The game includes levels of different challenge. These levels are shown in the game by reading from the input file.

When gamer starts the game, s/he will meet a start screen with a "Play the Game" button and a sweet soundtrack that will continue throughout the game. When the gamer clicks the button, the first level of the game will appear. At the top of game screen, there is a counter showing the number of moves made in that level. The center of the game screen holds tiles. Movable tiles can be moved to the empty free tiles' locations, if they are dragged by holding them on top by mouse. Also, movable tiles can be dragged one unit vertically or horizontally, not diagonally. On the lower left corner of the game, there are passive two buttons: 'Check' and 'Next'. 'Check' becomes active after the gamer completes the path and 'Next' becomes active after check button is pressed. If the gamer clicks on the 'Check' button, ball starts to move. Additionally, 'Next' button is to move to the next level if current level is completed. When the gamer clicks next button, new screen will appear with "Level… is completed in … moves!" text and "Next Level" button. Pressing the next level button will open the next level. When the gamer completed all levels, end screen will show up with "The game is completed." text.

IMPLEMENTATION DETAILS:



Above the text, we have the class relationship of our project.

Levels Folder: We have a folder named Levels and it contains the txt files for our game inputs. The naming convention for the levels is level(level number).txt.

Assets Folder: We have a folder named Assets and it contains the images and sounds for the game.

Tiles Package: We have a package named Tiles and it contains the classes except the GameBoard and Main classes.

Tile

Tile
- image Image - status String
+ getters and setters for each attribute

Tile class is an abstract class, and it is the superclass of the 8 following tile type classes. It has two attributes “image” which has a return type of Image and “status” which has a return type of String. “image” holds the image of the tile we are using, and “status” holds the “vertical”, “horizontal”, “00”, “01”, “10”, “11”. This “status” is for the rotation of the tiles that we will be using for the subclasses later. Also, we have getter and setter methods to access and change them.

StartPipe

StartPipe
+ StartPipe(status: String) + isFixed(): boolean

StartPipe is the subclass of the Tile class, and it implements the Fixed interface. Fixed interface states that this tile is static – which means it cannot move. It has a constructor with the parameters of String “status”. It is created as new StartPipe(“Vertical”) or new StartPipe(“Horizontal”). According to the given status, in this class’s constructor we set the image of that tile and set the status of that tile.

LinearPipe

LinearPipe
+ LinearPipe(status: String) + isMovable(): boolean

LinearPipe is the subclass of the Tile class, and it implements the Movable interface. Movable interface states that this tile is can be draggable to the empty spaces on the gameboard. It has a constructor with the parameters of String “status”. It is created as new LinearPipe(“Vertical”) or new LinearPipe(“Horizontal”). According to the given status, in this class’s constructor we set the image of that tile and set the status of that tile.

NormalPipeStatic

NormalPipeStatic
+ NormalPipeStatic (status: String) + isFixed(): boolean

NormalPipeStatic is the subclass of the Tile class, and it implements the Fixed interface. Fixed interface states that this tile is static – which means it cannot move. It has a constructor with the parameters of String “status”. It is created as new NormalPipeStatic(“Vertical”) or new NormalPipeStatic(“Horizontal”). According to the given status, in this class’s constructor we set the image of that tile and set the status of that tile.

CurvedPipeMovable

CurvedPipeMovable
+ CurvedPipeMovable(status: String) + isMovable(): boolean

CurvedPipeMovable is the subclass of the Tile class, and it implements the Movable interface. Movable interface states that this tile is can be draggable to the empty spaces on the gameboard. It has a constructor with the parameters of String “status”. It is created as new CurvedPipeMovable (“00”), new CurvedPipeMovable (“01”), new CurvedPipeMovable (“10”), new CurvedPipeMovable (“11”). These statuses are for the direction of the curved pipe’s arc, and it is shown also in the Test Cases section. According to the given status, in this class’s constructor we set the image of that tile and set the status of that tile.

CurvedPipeStatic

CurvedPipeStatic
+ CurvedPipeStatic(status: String) + isFixed(): boolean

CurvedPipeStatic is the subclass of the Tile class, and it implements the Fixed interface. Fixed interface states that this tile is static – which means it cannot move. It has a constructor with the parameters of String “status”. It is created as new CurvedPipeStatic (“00”), new CurvedPipeStatic (“01”), new CurvedPipeStatic (“10”), new CurvedPipeStatic (“11”). These statuses are for the direction of the curved pipe’s arc, and it is shown also in the Test Cases section. According to the given status, in this class’s constructor we set the image of that tile and set the status of that tile.

EndPipe

EndPipe
+ EndPipe(status: String) + isFixed(): boolean

EndPipe is the subclass of the Tile class, and it implements the Fixed interface. Fixed interface states that this tile is static – which means it cannot move. It has a constructor with the parameters of String “status”. It is created as new EndPipe (“Vertical”) or new EndPipe (“Horizontal”). According to the given status, in this class’s constructor we set the image of that tile and set the status of that tile.

Empty

Empty
+ Empty() + isMovable(): boolean

Empty is the subclass of the Tile class, and it implements the Movable interface. Movable interface states that this tile is can be draggable to the empty spaces on the gameboard. It has a no-arg constructor and an instance of that is created by using new Empty(), and this constructor sets the image and the status – as Empty. We cannot drag tiles to the Empty.

EmptyFree

EmptyFree
+ EmptyFree() + isMovable(): boolean

EmptyFree is the subclass of the Tile class, and it implements the Fixed interface. Fixed interface states that this tile is static – which means it cannot move. It has a no-arg constructor and an instance of that is created by using new EmptyFree(), and this constructor sets the image and the status – as EmptyFree. We can drag tiles to the EmptyFree only.

Fixed

<<interface>> *Fixed*
+ *isFixed(): boolean*

Fixed is the interface that we've created for using in the main part of the project. We used this interface to access our Tile's static attribute. It has one method to implement and it is boolean isFixed(). Since it does not have any purpose other than sorting the Tile according to its movability, while implementing this we just return true.

Movable

<<interface>> *Movable*
+ *isMovable(): boolean*

Movable is the interface that we've created for using in the main part of the project. We used this interface to access our Tile's movable attribute. It has one method to implement and it is boolean isMovable(). Since it does not have any purpose other than sorting the Tile according to its movability, while implementing this we just return true.

GameBoard

GameBoard
- numberOfMoves: int - borderPane: BorderPane - pane: Pane - boardScene: Scene - tiles: Tile[][] - imageViews: ImageView[] - checkButton: Button - nextButton: Button - totalLevelNumber: int - levels: ArrayList<File>
+ GameBoard() # makeScene(): Scene - createTiles(): Tile[][] - createImageViews(): ImageView[] - edgePane(label: Label): StackPane - gameControlsPane(button: Button, button2: Button): HBox - createBall(): ImageView # displayNumberOfMoves(): void + getters and setters for each attribute

ATTRIBUTES

GameBoard is the class we create the gameboard and the tiles according to the input files. Basically, it sets up the Scene for each level according to the level which is being played at that moment. It has 11 attributes and all of them are in private. First, we have the numberOfMoves which has the type of int, and it is also defined as static. This int keeps our number of moves for the game and later it will be displayed on the Scene. Next, we have a borderPane which has the type of BorderPane. In our game, we used a BorderPane and we will use the side parts for game controls and some info about the game – like number of moves. We will put our puzzle maze to the center of the BorderPane. Then, we have the pane as the type of Pane. Since it is more movement free than the other type of panes, we decided to use it. It has our tiles' ImageView inside. After that, we have the boardScene which has a type of Scene, and it is the Scene instance for our whole tile and controls. We use this so that while changing the level we can change the Scene too – so that it would be visible. Later on, we have the tiles which has the type of Tile[][] which is a two dimension Tile array. We keep our puzzle's details such as in which row and column we have the tile and what is the type of the tile. We will use these tiles array in our main game controls like checking, dragging, and making the Path for our ball. Next, we have the imageViews which has the same functionality as the tiles array. We will use them later for dragging. Next, we have the ball which has the type of ImageView and we set its coordinates later on. After that, we have the checkButton and nextButton which has the type of Button. They are our game control buttons and simply the checkButton is activated when a solution is found and when it is clicked our animation starts. Moreover, nextButton is activated when the animation is over and when it is clicked it takes the user to the Level Completed Screen which we will introduce later. Then, we have the totalLevelNumber which has the type of int and is also static. This keeps the level number we have in our “Levels” folder. Lastly, we have the levels which has the type of ArrayList<File>, it has our levels in a File format and we will be able to use them later to create levels.

CONSTRUCTOR

In the constructor, we have no parameters. It creates a folder - which is a File, and it creates an ArrayList for the levels which we have in the data field. After that, we have a for loop and in the loop a String levelNo is created and it keeps the level number. Then, a file as File is created in the loop and its path will be src/Levels/level(level number here as it is in the code).txt. After creating the file, we add this to the levels ArrayList.

Also, we create the checkButton and nextButton here and we disable them first with .setDisable(true) so that they will not be clickable first. Then, we set totalLevelNumber with the levels ArrayList’s size and lastly, we set the boardScene with the parameters being calling the makeScene() method.

METHODS

- makeScene()

This method is protected, and its return type is Scene. It calls the createTiles() and createImageViews() methods. Then, it creates a BorderPane, Pane and Scene – boardScene which was in data fields. It sets the background of the BorderPane, sets the right, and left with the edgePane method with the parameters of a Label with a text like this “”. The center is set as the pane and we add all the imageViews to the pane here. For the bottom, we have the gameControlsPane with parameters of two buttons – check and next buttons. We set the top part in another method – displayNumberOfMoves later. After completing the pane and borderPane, we create the boardScene with the borderPane with 950 x 780 pixels. In the end, this method returns the boardScene.

- createTiles()

This method is private, and its return type is Tile[][]. It starts with reading the data from the files. We create a Scanner with getLevels().get(Main.getLevelNumber()) – which gets the right level for each Scene. After that, we create a lines ArrayList<String> and add the input.next() to that ArrayList. Then, we create a words ArrayList<String> and add the split lines to that words ArrayList. The split separates the lines by the comma.

After these, we create a Tile two-dimensional array which has 4 rows and 4 columns. We have an int named queue which has the tiles index in one dimension and getting the mod and integer division gives us the column and row of the Tile. According to the name we create the Tile and add them to the array here. Lastly, it returns the tiles array.

- createImageViews()

This method is private, and its return type is ImageView[]. It creates an ArrayList for that first and adds them to the array list with setting the coordinates – 140x140 pixels for the tiles and 3 pixels for the spaces, and sizes. In the end, it converts the ArrayList to the array and returns that array.

- edgePane(Label label)

This method is private, and it creates a StackPane called edgePane and sets padding by using the label that with some text(in some cases we don't use a text). In the end, it returns the edgePane.

- gameControlsPane(Button button1, Button button2)

This method is private, and it creates a HBox with two buttons called gameControlsPane, adds them to the pane and sets padding for the pane. In the end, it returns the gameControlsPane.

- createBall()

This method is private, and its return type is ImageView. It creates and ImageView ball and then finds the Starter Tile. According to the Starter Tile's status and coordinates, it sets the coordinates for our balls. For the size of the ball, we used 55 x 55 pixels. In the end, it returns the ball.

- displayNumberOfMoves()

This method is protected, and it is a void method. It creates a Label with the text of "Number of Moves" and the number next to it. It is in white color, and we set some font and size for the text. Then, we use the edgePane method for this one and set the top of the BorderPane which was told earlier.

GETTERS AND SETTERS

Also, we have some getters and setters. But we will not be talking about these since they are default setters and getters.

Main

Main
- gameBoard: GameBoard - levelCompleted: boolean - levelNumber: int - pipesInOrder: ArrayList<Tile> - path: Path - draggable: boolean
+ main(args: String[]): void + start(primaryStage: Stage): void - startPage(primaryStage: Stage, startButton: Button): void - clickedOnStart(startButton: Button, primaryStage: Stage): void - clickedOnCheck(): void - clickedOnNext(primaryStage: Stage, nextLevelButton: Button, closeButton: Button): void - levelCompletedScene(nextLevelButton: Button): Scene - clickedOnNextLevel(primaryStage: Stage, nextLevelButton: Button): void - clickedOnClose(closeButton: Button, primaryStage: Stage): void - drag(): void - swapImages(imageView1: ImageView, imageView2: imageView): void - swapTiles(index1x: int, index1y: int, index2x: int, index2y: int): void - dragAnimation(imageView1: ImageView, imageView2: imageView): void - checkForSolution(): boolean - directionFinder(statusCurve: String, statusPrevious: String): String - setWholePath(): void - reverseDirection(moveTo: MoveTo, lineTo: LineTo): void - reverseDirection(moveTo: MoveTo, arcTo: ArcTo): void - indexFinder(tile: Tile): int - animate(): void + getters and setters for each attribute

ATTRIBUTES

Main method is the subclass of Application and in Main, we create methods for the game controlling mechanisms – like drag, checking, clicking methods which will be introduced in some lines after this. We have 6 attributes this time. First, we have the “gameBoard” in the type of GameBoard. Since our Scene is in gameBoard, we access that by using the data field in game mechanisms which makes accessing easier. Also, we have a boolean “levelCompleted” that stores whether the current level is completed or not. Then, we have a static int “levelNumber” that stores the current level number. Later, we have an ArrayList of Tile named pipesInOrder and it stores the pipes that makes a path and is checked while adding the pipes. After that, we have the path which is in the type of Path and it creates our path for the ball’s animation. Lastly, we have a boolean named draggable and it stores if the gameBoard can be draggable or not – when the solution is found we cannot drag them anymore.

METHODS

- main(String[] args)

This method only contains the launch(args) as default and as usual it is a static void method.

- start(Stage primaryStage)

It is the overridden start method because this class extends from the Application class, and it has the parameter primaryStage which is a Stage type. It is public and void. Firstly, this method creates the three buttons which are startButton, nextLevelButton and closeButton. Then it creates the gameBoard here. After that, it calls the methods startPage, clickedOnStart, clickedOnCheck, clickedOnNextLevel, clickedOnClose.

- startPage(Stage primaryStage, Button startButton)

This method is private and void. Also, it has the primaryStage as Stage and startButton as Button for the parameters. First, we set the game sound with creating a media in the type of Media. For this, we create file first and get its URI as a String. Then we create an audioClip as AudioClip with getting the source of the media we have created. After, we set the volume to 20 and play it.

After the sound, we set our primaryStage as not resizable. Then, we create our welcoming Scene. We create a Label with the game's name and set its font, size and color. Then, we set our startButton's size and we create a VBox named startPane with spacing 50. Later, we add the Label and Button to the startPane with the alignment to the center. We set the background and then create the startScene with the startPane and 950 x 780 pixels. Lastly, we set the Scene to that startScene in the primaryStage and set the title with the game's name. Then show it to the users.

- clickedOnStart(Button startButton, Stage primaryStage)

This method is private and void with the parameters of startButton as Button and primaryStage as Stage. This method has only the lambda expression of the startButton. When the user clicked on the startButton, it activates and sets the first level. Firstly, it resets the numberOfMoves to 0 which has in GameBoard, sets the scene of primaryStage by using the gameBoard's makeScene method and shows them to the user. After it is shown, we set the draggable to true and and check if it is true at first and call the drag method as well.

- clickedOnCheck()

This method is private and void with no parameters. This method has only the lambda expression for the gameBoard's checkButton. When it is clicked by the user, the animation of the ball starts.

- clickedOnNext(Stage primaryStage, Button nextLevelButton, Button closeButton)

This method is private and void with parameters of primaryStage as Stage, nextLevelButton as Button and closeButton as Button. This method has only the lambda expression for the gameBoard's nextButton. When it is clicked, it takes the user to a new Scene.

If the level is not the last level, it takes the user to the levelCompletedScene method. Otherwise, it creates the levelCompleted Scene with the Finish button and when the user clicked Finish, it takes the user to the Scene which says the game is completed. For that, we create a VBox named endPane with spacing 50, set the background and creating a Label with the text of "The game is completed." with some fonts and sizes. Also, we set the closeButton's size and add them to the endPane with the alignment as the center. Then we create an endScene with endPane and 950 x 780 pixels.

- levelCompletedScene(Button nextLevelButton)

This method is private and returns Scene with the parameters of nextLevelButton as Button. We use this method when a level is completed and pressed the next button. For that, we create a VBox named levelCompletedPane with spacing 50, set the background and creating a Label with the text of "Level ... is completed in ... moves." with some fonts and sizes. Also, we set the nextLevelButton's size and add them to the levelCompletedPane with the alignment as the center. Then we create a levelCompletedScene with levelCompletedPane and 950 x 780 pixels. In the end, we return that levelCompletedScene.

- clickedOnNextLevel(Stage primaryStage, Button nextLevelButton)

This method is private and void with parameters of primaryStage as Stage and nextLevelButton as Button. This method has only the lambda expression for the nextLevelButton. When it is clicked, it resets the disability of the checkButton and nextButton, and it assigns the numberOfMoves with 0 again. It sets the primaryStage's scene with the gameBoard's makeScene method and shows it. Also, it makes it draggable again by making it true. After these, it checks if the level is completed and calls the drag method.

- clickedOnClose(Button closeButton, Stage primaryStage)

This method is private and void with parameters of closeButton as Button and primaryStage as Stage. This method has only the lambda expression for the closeButton. When it is clicked, it makes the primaryStage closed.

- drag()

This method is private and void. To be simple, it makes the tile drag. First, it creates imageViews array and gets them from the gameBoard and same for the tiles. Then, it creates a loop for imageViews and it creates and sets the imageView1 to the imageView in the loop when that imageView is released. For imageView2, we just create it firstly as null. Then, we get the result of the event e and if its intersected Node is ImageView, set that intersected Node to the imageView2. Then, we check if imageView is not null. If it is not null, then we search their indexes for the tiles and we set them to index1x, index1y for imageView1 and index2x, index2y for imageView2. In the end, we check if the second Tile is EmptyFree and first Tile is not EmptyFree. If it is okay, we check the first Tile is an instance of Movable interface. After that if the tiles can be draggable and it is in the area of the tile, we start the drag's instructions. First, we make the first Tile come to the front and start the dragging animation by calling the dragAnimation. Then we swap the imageViews and the tiles. Lastly, we display the number of moves since they will be increased and will need to be updated. Also, we do set the levelCompleted after every drag by using the checkForSolution method.

- swapImages(int index1x, int index1y, int index2x, int index2y)

This method is private and void with the parameters of imageView1 as ImageView and imageView2 as ImageView. It swaps them by creating a temp and changing the x and y coordinates to each other's. In the end, it increments the numberOfMoves.

- swapTiles(int index1x, int index1y, int index2x, int index2y)

This method is private and void with the parameters of index1x as int, index1y as int, index2x as int and index2y as int. It swaps the tiles with the help of a temp Tile.

- dragAnimation(ImageView imageView1, ImageView imageView2)

This method is private and void with the parameters of imageView1 as ImageView and imageView2 as ImageView. It creates a pathTransition as PathTransition. It creates a Line for the orbit of the dragging as the parameters being the imageViews' centers. Then, it sets the Path as the line we have created and sets the Node as the imageView1. We set the duration as 0.50 seconds and we play the animation. Also, while the animation occurs, there is a sound effect, and we create it by creating a Media with the sound effect's File's URI as a String. Then, we create an AudioClip by getting source of the media we have created and set the volume and play it. When the animation is finished, we take the imageView1 to back again.

- checkForSolution()

This method is private and returns a boolean with no parameters. First, we create the ArrayList for the pipesInOrder and we will add them our pipes while doing the check. In our check algorithm, we decided to find the Starter Tile first and count how many CurvedPipeMovables or CurvedPipeStatics we have. Then, we check the next tile has the same status as previous tile – for example Starter, until a CurvedPipeMovable or CurvedPipeStatic is found. When the curved pipe is found, we call the directionFinder method to get the current direction we are going into. After, we do the same process for each CurvedPipeMovables or CurvedPipeStatics.

In the code, we have conditionals for Starter Tile first we check it and when a CurvedPipeMovable or CurvedPipeStatic it breaks. After that, we have a loop for the other CurvedPipeMovables or CurvedPipeStatics. Since the first one is found, we start our loop with 1 here instead of 0. As in the StartTile we do the same procedure with the new direction we got from the directionFinder method. We have 4 ifs in the loop according to the directionFinder – as if it is "up", "down", "right" or "left". According to that we check the tiles as the Starter Tile and also add them to the pipesInOrder ArrayList from the Starter Tile to the End Tile . If the EndPipe is found our code makes check button enabled and increases the level number. If the direction is wrong or no EndPipe is found, we return false and break. Also, it makes the table not draggable and returns true right there.

- directionFinder(String statusCurve, String statusPrevious)

This method is private and returns a String with the parameters of statusCurve as String and statusPrevious as String. According to these, we find the next tile's direction. For example, if the statusPrevious is Horizontal and statusCurve is 00, the only gap will be the up direction, so we need to continue from there. In this case, it returns "up" as a String. We have the ifs and else for each case to keep it more readable we will not be mentioning here.

- setWholePath()

This method is private and void with no parameters. It creates a path as in the data field. Then, it creates a loop with the pipesInOrder and starts to create the path for them one by one. According to the Tile's type – Start, End, Linear, etc., it creates a moveTo and a lineTo or a moveTo and an arcTo.

For StartPipe, we check its status and according to the status we create a moveTo and a lineTo to from the path. Since the ball only can go down or left, we do not use the reverseDirection method here. We create the moveTo for the starting point and lineTo for the ending point. Then, we add them to the path's elements, and it forms a line for us, but the path is not closed, and we can add more lines to these.

For EndPipe, we check its status and according to the status we create a moveTo and a lineTo to from the path. Since the ball only can come from down or left, we do not use the reverseDirection method here. We create the moveTo for the starting point and lineTo for the ending point. Then, we add them to the path's elements, and it forms a line for us, but the path is not closed, and we can add more lines to these.

For LinearPipe and NormalPipeStatic, we check its status and according to the status we create a moveTo and a lineTo to form a path. Since the ball can come from up, down, right, or left, we use reverseDirection this time. We check if their difference in x or y is negative, and if it is negative, we call the reverseDirection method. We create the moveTo for the starting point and lineTo for the ending point. Then, we add them to the path's elements, and it forms a line for us, but the path is not closed, and we can add more lines to these.

For CurvedPipeMovable and CurvedPipeStatic, we check its status and according to the status we create a moveTo and an arcTo to form a path. Since the ball can come from up, down, right, or left, we use reverseDirection this time too. We check if their difference in x or y is negative, and if it is negative or positive according to the status of the CurvedPipeMovable or CurvedPipeStatic, we call the reverseDirection method. We create the moveTo for the starting point and arcTo for the ending point. For arcTo, we set the radius which is different from lineTo. Then, we add them to the path's elements, and it forms a line for us, but the path is not closed, and we can add more lines to these.

In the end, we set the stroke to white.

- reverseDirection(MoveTo moveTo, LineTo lineTo)

This method is private and void with the parameters of moveTo as MoveTo and lineTo as LineTo. It swaps them by creating a temp and changes their x and ys. This method is for only linear like pipes.

- reverseDirection(MoveTo moveTo, ArcTo arcTo)

This method is private and void with the parameters of moveTo as MoveTo and arcTo as ArcTo. It swaps them by creating a temp and changes their x and y. This method is for only arced like pipes.

- indexFinder(Tile tile)

This method is private, and its return type is int. It has parameters tile as Tile. According to tile, it finds the imageView by first finding the tile's index and comparing the tile's image to the imageViews array's image. After finding the index, it returns its index as an int.

- animate()

This method is private and void with no parameters. This method calls the setWholePath() method and then sets the opacity of the path from the data field to 0. Then, it adds the path to the gameBoard's pane and makes the ball's ImageView come to the front. Later, it creates a path transition and sets the path with the path in the data field. It sets the Node as the ball and sets the duration as 2 seconds. Then, it starts to play it. When the animation is finished, it enables the gameBoard's nextButton.

GETTERS AND SETTERS

Also, we have some getters and setters. But we will not be talking about these since they are default setters and getters.

QUESTIONS:

i) Which parts are complete/incomplete in your project?

All the parts are completed. It is working perfectly.

ii) What are the difficulties you have encountered during the implementation?

We had problems while we are coding the part of check. Firstly, we had a hard time finding what was wrong and then we completely changed that part into something more widely covered method that passes every test we have ever made.

iii) What are the additional functionalities of your project added by your team?

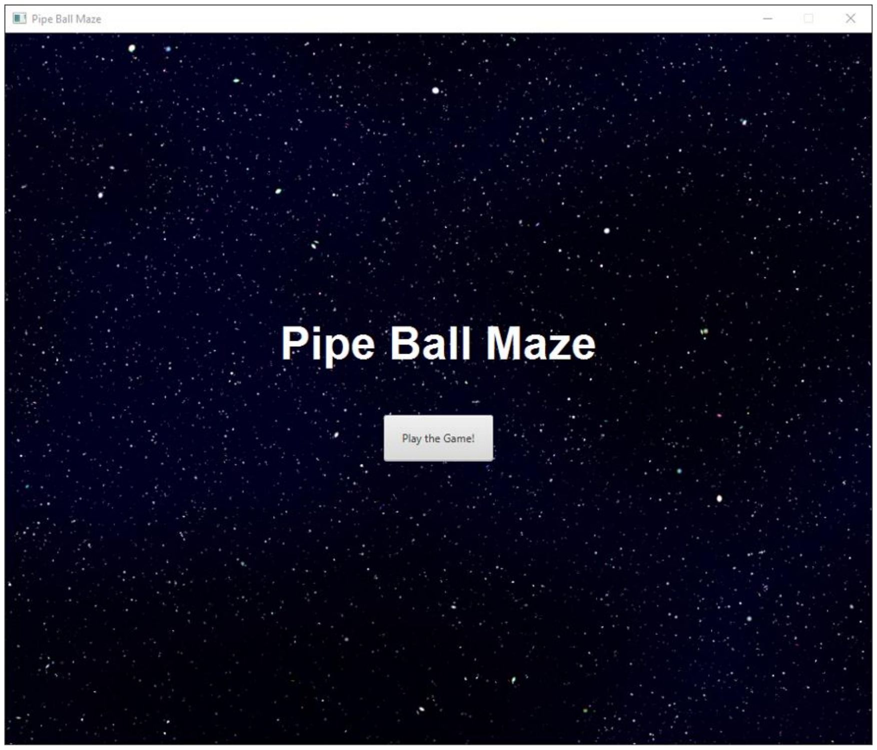
We put a start menu, a check and next button, level completed screen and game finished screen with an usable exit button inside. Also, we added some music and sound effects to enjoy more while gaming. We designed our tiles and make it more space themed so that it will capture attention of younger gamers.

TEST CASES

This part consists of some screenshots of our project execution for the given test cases and explanation about of them.

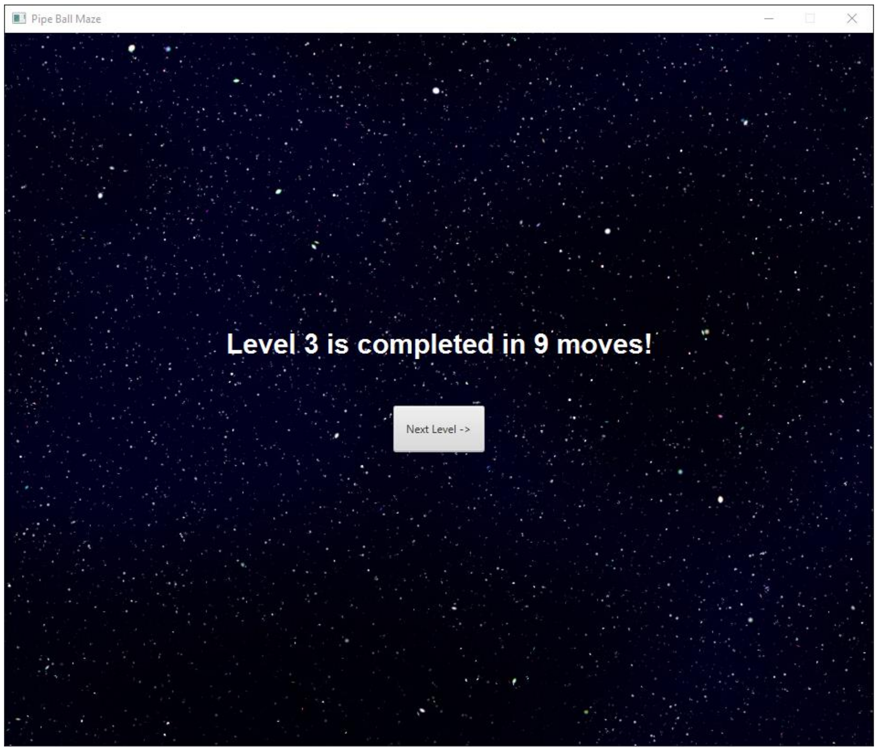
Start Screen

This image was taken from the start screen. There is label for the game name, a "Play the Game" button and a lovely music that will continue throughout the game. The gamer should click the button to start the game.



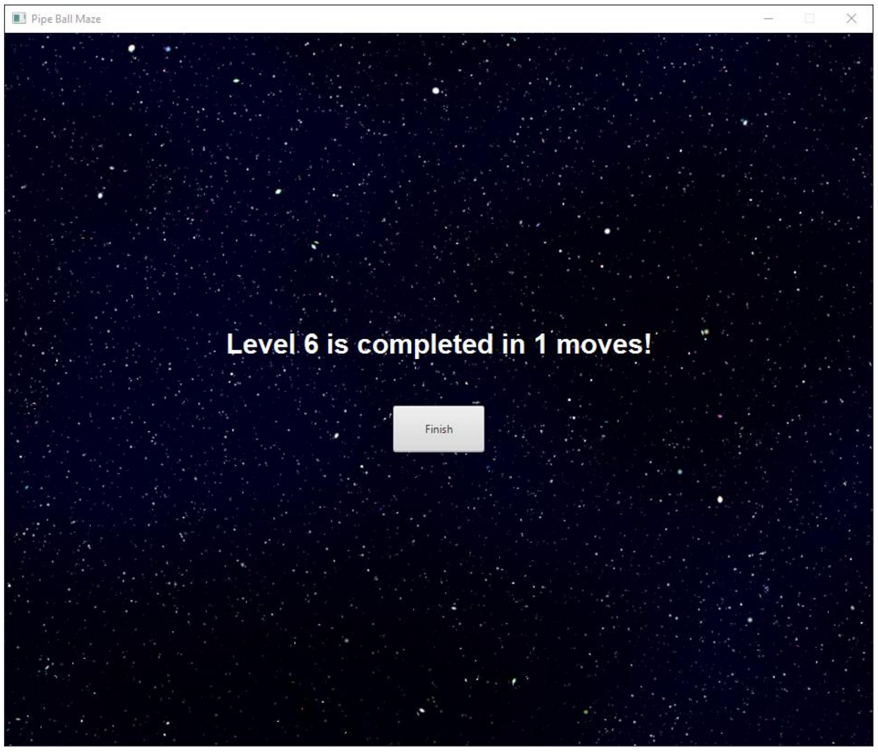
Level Completed Screen

This image was taken from the level completed screen. It shows how many moves the player has completed the level. Besides, it has "Next Level" button to move next level.



The Last Level's Level Completed Screen – with the Finish button.

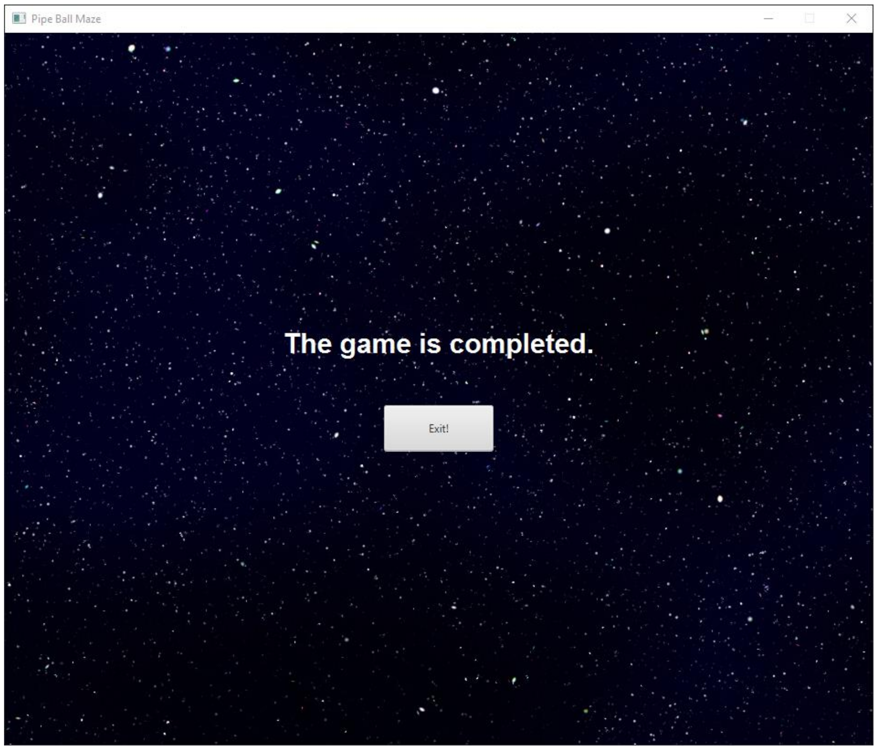
This image was taken from the level completed screen of last level. It shows how many moves the player has completed the level as normal level completed screen. In contrast, since there is no more level, it has "Finish" button to move end screen.



Level 6 is completed in 1 moves!

The End Screen – with Exit button.

This image was taken from the end screen. There is a message to the gamer which is "The game is completed". Additionally, it has exit button to exit the game.

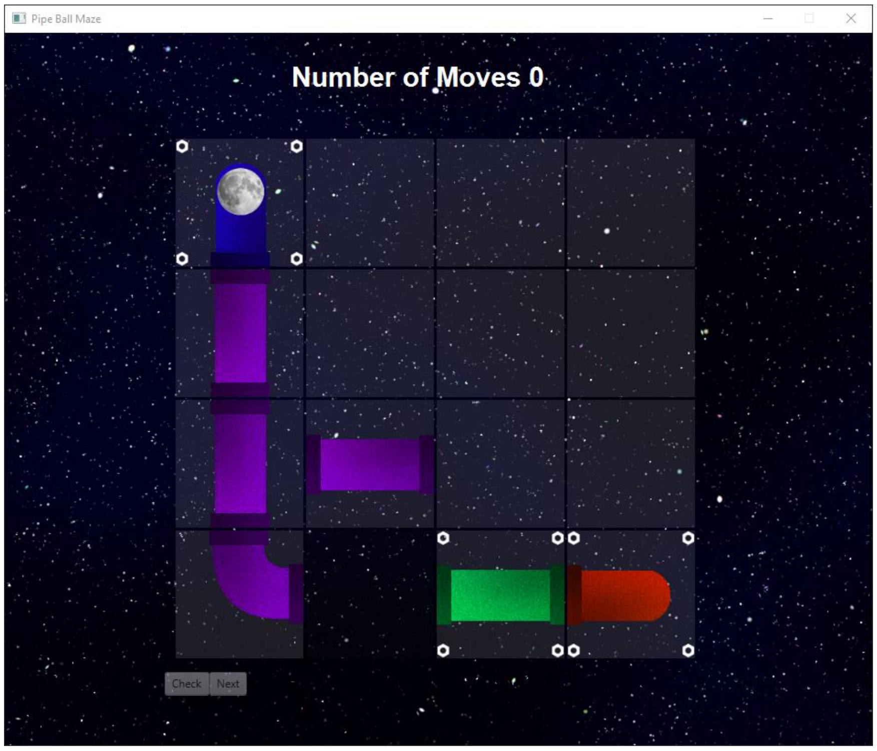


The game is completed.

1) 1st Level

This image was taken from the first level.

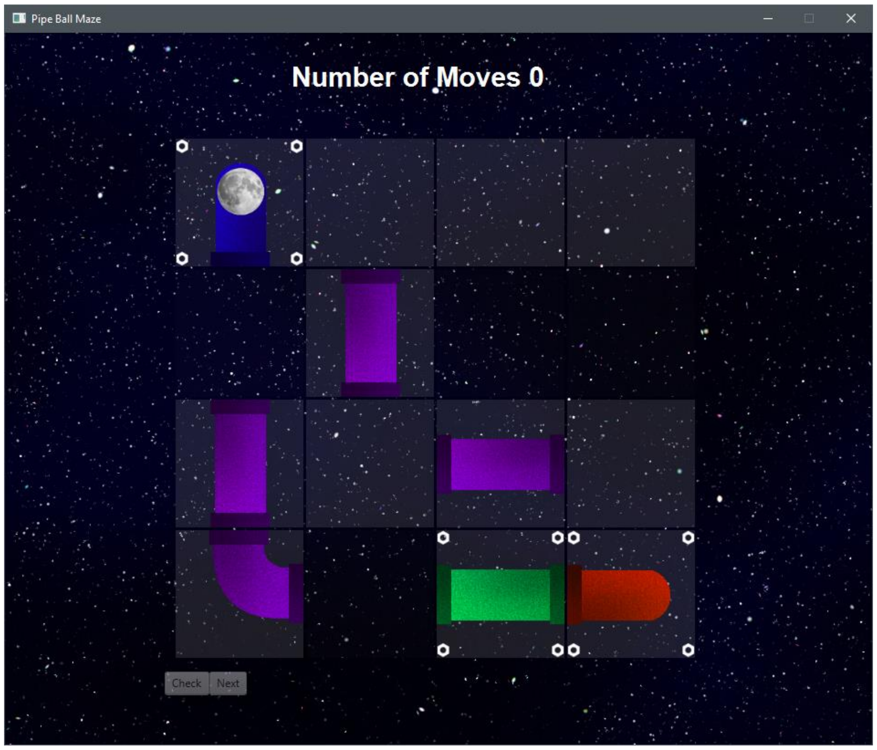
At the center, main game is located. At the top, there is counter that holds number of moves made by the gamer. Also at the lower left corner, two buttons are placed which are check and next. Since the gamer hasn't joined the pipes properly yet, both are passive.



2) 2nd Level

This image was taken from the second level.

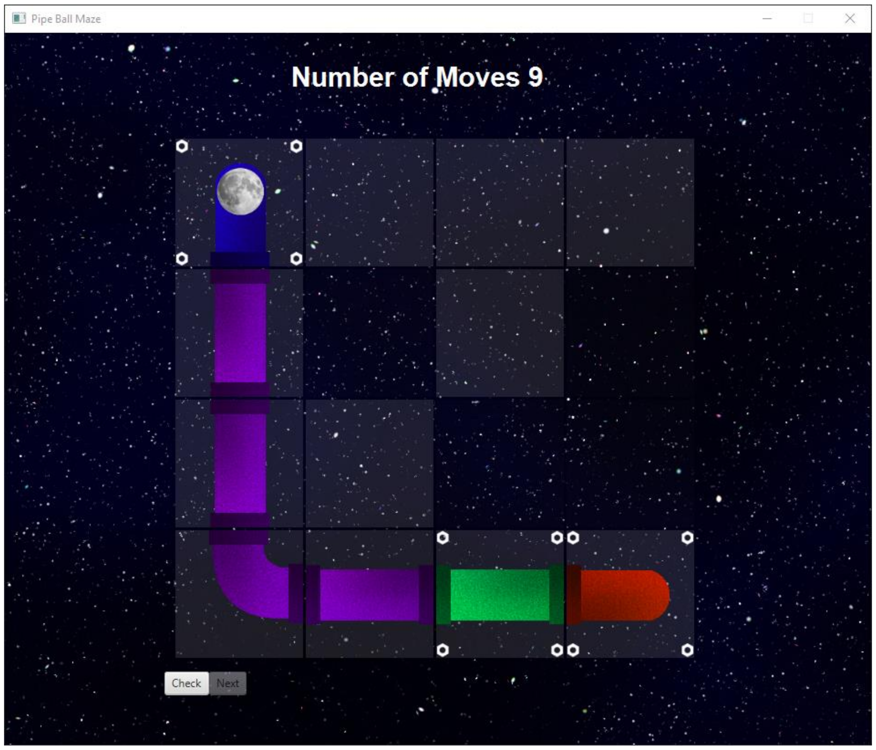
As it is seen from the image, while all draggable pipes are purple, not-draggable ones have different color. Draggable tiles can be dragged to the location of empty free tiles which are illustrated with just space.



3) 3rd Level

This image was taken from the third level.

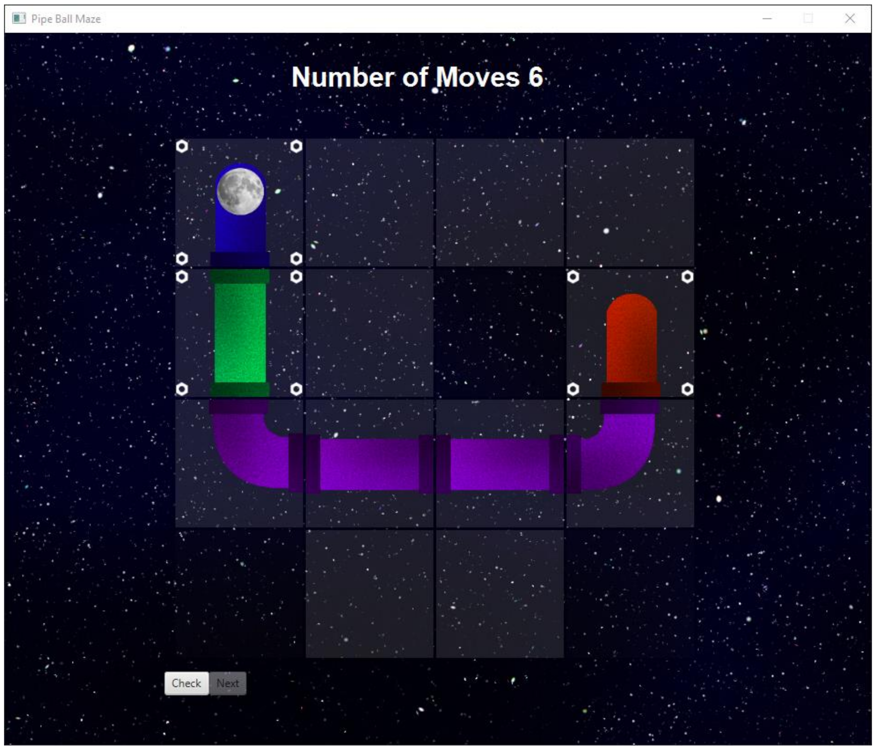
Since the gamer has correctly located the pipes through which the ball will roll, check button has been activated. Moreover, as it is seen, the gamer has completed this level in 9 moves.



4) 4th Level

This image was taken from the fourth level.

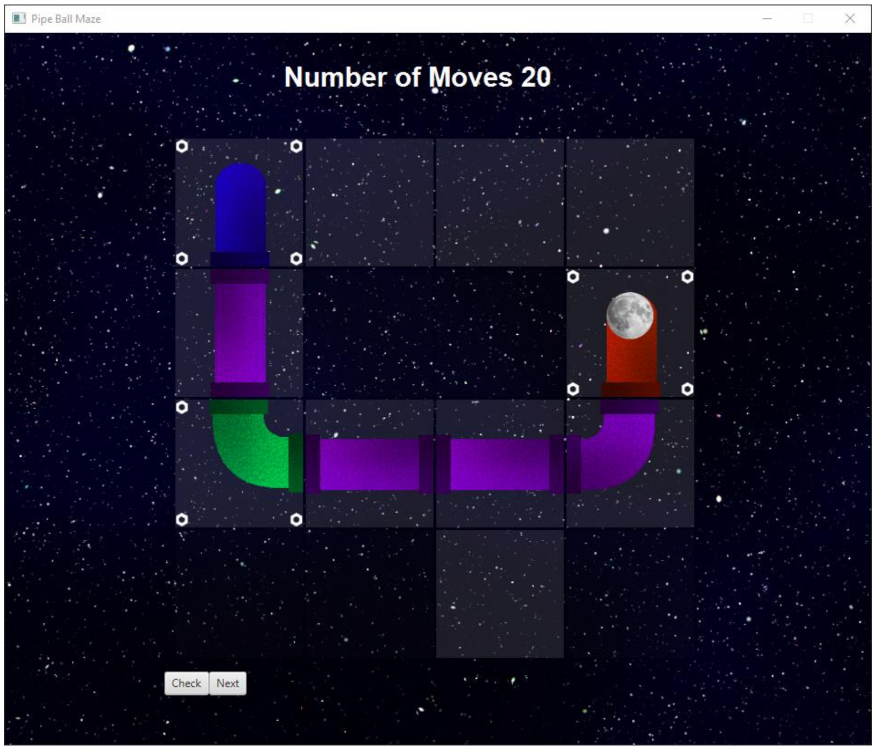
Since the path is completed, gamer cannot move tiles now. If gamer clicks the activated next button, the ball will start to roll. When the ball reaches the end pipe, next button will be active.



5) 5th Level

This image was taken from the fifth level.

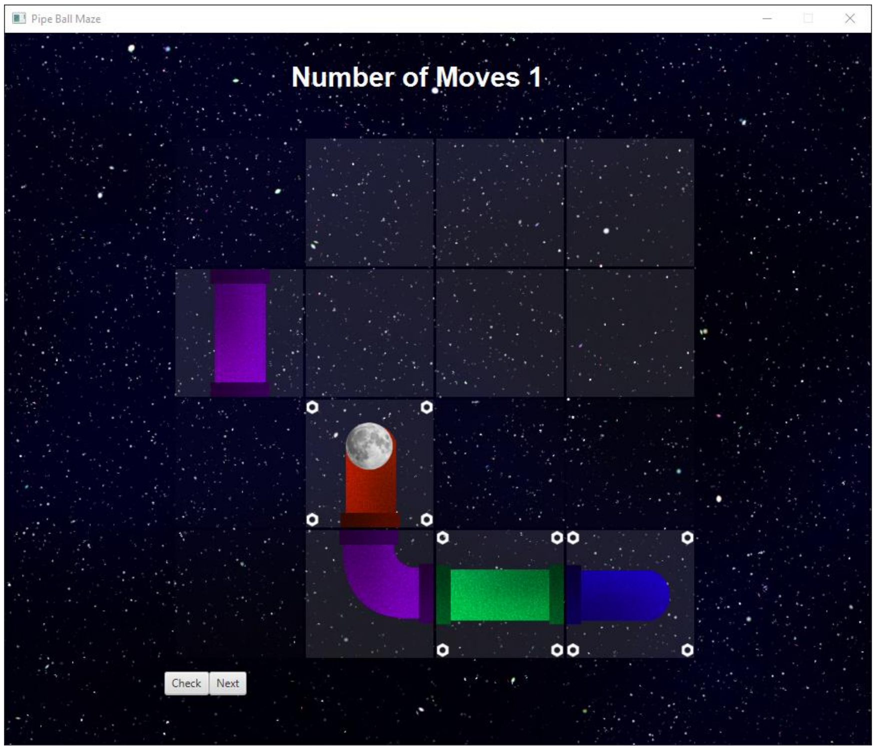
Since the ball has reached the end pipe, next button is become active. When it is clicked, level completed screen will show up.



6) 6th Level

This image was taken from the sixth level.

In that level, there is extra pipe which is not include the ball's path. Since that is the last level, when the next button is clicked, the last level's level completed screen with the finish button will appear.


Pipe Ball Maze
Number of Moves 1
Check
Next