YouTube video link: https://youtu.be/nh6PDpsOJZU

Intoduction

The game consists of 1 level, which is made of 5 stages. Every stage has its own unique rules and the aim is passing all stages. The animations are created using StdDraw library.

The objective for each stage is to pass through the exit pipe. The pipe is blocked by a door which opens by pressing the button. There are spikes in map which kills the player and restarts the stage upon touching them. When all the stages are passed, the user can press A to play again or Q to quit.

Game Components

The game consists of the main components: map, the player, and the bottom screen. These components are implemented with an OOP approach.

Classes

There are five classes in the code: Game, Map, Stage, Player and the main class MustafaSahin

MustafaSahin

The main class, this creates the Game object and starts the game after setting canvas dimentions.

Player

The class that holds the player's height, width, position, y-velocity and direction variables. Has setters and getters available. Also has a method that draws player on map using the image file of the player.

Stage

This class holds the stage's unique values such as stage number, gravity, x and y velocites, keycodes for movements and the hint strings clue and help. Has getters available.

Map

This class holds all the obstacles present on the map — including the walls, the button, the door, the start and exit pipes, and the spikes. It also holds assistant variables for these obstacles, for example isDoorOpen variable holds the information of the door's status. The Map class also has a Stage and a Player data field for referencing and changing these classes objects. This class has many methods used for checking collisions, drawing the obstacles and more.

Game

The Game class handles all user input and draws the output animations. The class has its ArrayList<Stage> data field holding the stages that will be played. It also includes the stageIndex variable, several others for calculating game time, and some int[] coordinate arrays for processing input from buttons. Additionally, the class maintains variables for tracking gameplay statistics, such as the deathNumber.

Some methods defined in this class are handleInput for processing input, convertTime for converting game time to a string, drawing methods for drawing the bottom screen, stage passed, resetting and winning animations, and most importantly, the play method which calls every other method created. The play method is called in the main MustafaSahin class to start the game. The Game class has some setters and getters available as well.

UML Diagrams

Player	
- x: double	Position's x component
- y: double	Position's y component
+ WIDTH: double	Player's width
+ HEIGHT: double	Player's height
- velocityY: double	Player's vertical velocity
- lookDirection: int	The direction player is turned towards, -1 or 1
- xMoveDirection: int[]	The x direction player is set to move
- yMoveDirection: int[]	The y direction player is set to move
+ Player(x: double, y: double)	Constructs player with given x, y
+ respawn(spawnPoint: double[]): void	Resets all data fields to initial values
+ draw(stageIndex: int): void	Draws player on map at (x, y)
//setters	Sets the following data fields to input value: x, y, velocityY, lookDirection, xMoveDirection, yMoveDirection
//getters	Returns the following data fields: x, y, velocityY, xMoveDirection, yMoveDirection

Game	
- stageIndex: int	Current stage index of the game
- stages: ArrayList<Stage>	Stages list of the game
- deathNumber: int	Number of deaths in the game
- resetGame: boolean	Indicates game reset status
- startTime: long	The current time when the game starts(milliseconds)
- frameDuration: int	The pause duration for each frame(ms)
- gameTime: int	Overall passed time, (current time - startTime) (ms)
- resetTime: long	Last time the game got resetted(ms)
+ TIME_STEP: double	The delta time between each frame for player(s)
- restartButton: int[]	The coordinates of the restart button
- resetButton: int[]	The coordinates of the reset button
- helpButton: int[]	The coordinates of the help button
- isRestartPressed: boolean	Indicates restart button's status
- isResetPressed: boolean	Indicates reset button's status
- isHelpPressed: boolean	Indicates help button's status
- clueMessage: String	The clue message displayed on bottom screen
+ Game(stages: ArrayList<Stage>)	Constructs game object with the playable stages
+ play(): void	Starts the game and calls all other methods
- handleInput(map: Map, currentStage: Stage, player: Player): void	Changes variables according to input
- drawBottomScreen(clue: String): void	Draws the bottom screen
- drawResetScreen(): void	Draws resetting banner for two seconds
- drawStagePassed(): void	Draws stage passed banner for two seconds
- drawWinScreen(): void	Draws the winning screen
- convertTimeToString(timeElapsed: int): String	Converts given elapsed time into a string with format: mm : ss : msms
+ getStageIndex(): int	Returns stageIndex
+ getCurrentStage(): Stage	Returns the current stage being played
+ setStageIndex(stageIndex: int): void	Sets stageIndex to input integer
- setResetTime(): void	Sets reset time to current time
- resetDeathNumber(): void	Sets deathNumber to 0
- increaseDeathNumber(): void	Increases deathNumber by 1

Stage	
- stageIndex: int	Stage number(1-5)
- gravity: double	Gravity applied to the player
- velocityX: double	Player's x velocity
- velocityY: double	Player's maximum y velocity
- rightCode: int	Key code for right movement
- leftCode: int	Key code for left movement
- upCode: int	Key code for up movement
- clue: String	Stage clue
- help: String	Help for stage
- color: Color	Color of the obstacles, randomly selected
- SPAWN_POINT: double[]	The spawn point for the stage
Stage(gravity: double, velocityX: double, velocityY: double, stageNumber: int, rightCode: int, leftCode: int, upCode: int, clue: String, help: String)	Creates stage with given parameters
getKeyCodes(): int[]	returns new int[]{rightCode, leftCode, upCode}
//getters	Returns the following data fields: stageNumber, gravity, velocityX, velocityY, clue, help, color, SPAWN_POINT

Map	
- obstacles: int[][]	Coordinates of obstacles
- button: int[]	Coordinates of button
- buttonFloor: int[]	Coordinates of button floor
- startPipe: int[][]	Coordinates of start pipe
- exitPipe: int[][]	Coordinates of exit pipe
- spikes: int[][]	Coordinates of spikes
- spikeAngles: int[][]	The amount of degrees for each spike to rotate
- door: double[]	Coordinates of door
- doorFloor: double	Y coordinate of door's floor
- DOOR_SPEED: double	Door's speed
- stage: Stage	Stage of the map
- player: Player	Player object in the map
-TIME_STEP: double	The delta time between each frame for player(s)
- X_SCALE: int[]	The x boundaries of playable area
- Y_SCALE: int[]	The y boundaries of playable area
- buttonPressNum: int	Number of times the button is pressed
- isButtonPressed: boolean	Indicates status of button
- isDoorOpen: boolean	Indicates status of door
- doorColor: Color	The door's color
- pipeColor: Color	The pipe's color
- buttonFloorColor: Color	The button floor's color
- buttonColor: Color	The button's color
- HALFWIDTH: double	Half of player's width
- HALFHEIGHT: double	Half of player's height
+ DIRECTIONS: double	The direction array holding 4 direcitions and the stationary status
+ Map(stage: Stage, player: Player)	Constructs Map object with given stage and player
+ restartStage(): void	Resets the map objects to their initial values
+ movePlayer(): void	Moves player depending on player's move directions and stage's gravity
+ fall(): void	Adds gravity*TIME_STEP to player's velocityY
+ jump(): void	Sets player's velocityY to its max value
+ slideDoor(): void	Decreases the door's y coordinates by DOOR_SPEED*TIME_STEP if the door is set open and is not fully opened yet
+ checkSpikeCollision(nextX: double, nextY: double): boolean	Checks if the next position of the player intersects with a spike
+ checkDoorCollision(nextX: double, nextY: double): boolean	Checks if the next position of the player intersects with the door
+ changeStage(): boolean	Checks if the next position of the player intersects with exitPipe[1]
+ checkButtonPressed(): void	Sets isButtonPressed true if player's position intersects with the button
+ checkCollision(nextX: double, nextY: double, obstacle: int[]): boolean	Checks if the next position of the player will intersect with the given obstacle
+ checkCollision(nextX: double, nextY: double, obstacle: double[]): boolean	Checks if the next position of the player will intersect with the given obstacle
+ checkDirectionCollision(nextX: double, nextY: double, direction: int[]): boolean	Checks if the player will intersect an obstacle from the given direction
+ isIn(x: double, y: double, obstacle: int[]): boolean	Checks if the given coordinates are inside the given obstacle
+ isIn(x: double, y: double, obstacle: double[]): boolean	Checks if the given coordinates are inside the given obstacle
+ draw(): void	Draws all the map elements and player
- drawArrayedRectangles(array: int[][]): void	Draws multiple rectangles whose coordinates are given in array
- drawRectangle(rectangle: int[]): void	Draws the rectangle whose coordinates are given in rectangle array
- drawRectangle(rectangle: double[]): void	Draws the rectangle whose coordinates are given in rectangle array
- drawSpikes(): void	Draws the spikes
+ setDoorStatus(isStageFour: boolean): void	Sets the door open depending on the stage's requirements
+ pressButton(): void	increases buttonPressNum by 1
//setters	Returns the following data fields: stage, player, DIRECTIONS

figures 1, 2, 3 and 4, UML diagrams

How the Game Works

main

The programs starts by executing the main method in MustafaSahin class. After the stages to be played are created and added to stages ArrayList; the canvas size, xScale and yScale are set. Then the Game object “game” is created with the stages ArrayList, and game is started with game.play() call after StdDraw.enableDoubleBuffering().

stages

Five Stage objects are created, here are the rules that makes them unique:
s1: standart gameplay
s2: the arrow controls for left and right are reversed
s3: the player is constantly jumping with a higher jump speed and gravity
s4: the door opens after 5 presses to button
s5: the controls are now done using F, T, H keys

play()

First things first, startTime is set to the current time using System.currentTimeMillis(), so we can refer the starting time in the time calculations. Then, we need to create the Player and Map objects. For this, we need to use the stages ArrayList. We use a for loop to access the stages inside the ArrayList:

```
for (Stage currentStage : stages) { … }
```

Inside the for loop, we set stageIndex and clueMessage variables to their currentStage values. We also initiate Map and Player objects as seen in *figure 5*.

```
Player player = new Player(getCurrentStage().getSpawnPoint()[0], getCurrentStage().getSpawnPoint()[1]);

Map map = new Map(getCurrentStage(), player);
```

figure 5, Player and Map objects are initiated

Then, the main while loop starts, updating each frame. The loop breaks if the stage is passed, checked by map.changeStage() (the helper methods are explained later). In the while loop, the following are done in the order:

1. Checking if reset button is pressed
2. Checking if button is pressed
3. Taking input(setting variables accordingly)
4. Checking if player will hit a spike
5. Movements
6. Drawing
7. Time updates

Checking if reset button is pressed

The game is resetted if the reset button is pressed, checked by resetGame. When the game is reset, the data fields that are changed, and initiated outside the play() method, which are just deathNumber and resetGame, are set to their original value. Also resetTime is set to the current time using setResetTime(). map.drawResetScreen() is called to draw reset screen for two seconds and finally play() method is called to start the game from the beginning.

Checking if button is pressed

The button’s status and press number are stored in isButtonPressed and buttonPressNum data fields. buttonPressNum is updated using the unupdated and updated isButtonPressed values; if isButtonPressed turns from false to true, it means the button got pressed. We create the lastButtonStatus variable to store isButtonPressed and then call map.checkButtonPressed() to update isButtonPressed. Then we increase buttonPressNum with map.pressButton() if !lastButtonStatus && isButtonPressed is true.

We finally use map.setDoorStatus(stageIndex == 4) with our new buttonPressNum and isButtonPressed values. This method sets isDoorOpen true depending on the stage's requirements. If the stage is the fourth stage, the door is opened when buttonPressNum>=5, and for any other stages, the door is opened if isButtonPressed is true.

Taking input

Throughout the gameplay, there are 2 ways of input taking: control keys and mouse input. We use the handleInput(Map map, Stage currentStage, Player player) method to check for these inputs and set the associated variables. First we create the variables, keyCodes and directions and assign them their values using the methods currentStage.getKeyCodes() and map.getDirections(). Then we check for control keys input with StdDraw.isKeyPressed(keyCodes[i]), and set movement directions by using player.setXDirection(keyCodes[i]) and player.setYDirection(keyCodes[i]) in accordance. We also set the y movement direction to directions[2] if the keyCode for jumping is -1(meaning it is stage 3 and player constantly jumps).

For the mouse inputs, the input taking from mouse checks for clicking, meaning the mouse should be pressed on the button and then be released. If the button is pressed on top of button area, the related isPressed value(isResetPressed, isRestartPressed or isHelpPressed) is set true. When the button is released and isPressed is true, we set the isPressed false and necessary action is taken as shown in *figure 6*.

```
if (StdDraw.isMousePressed()) {

    if (map.isIn(mouseX, mouseY, helpButton)) {isHelpPressed = true;}.

    if (map.isIn(mouseX, mouseY, restartButton)) {isRestartPressed = true;}

    if (map.isIn(mouseX, mouseY, resetButton)) {isResetPressed = true;}.

} else {

    if (isRestartPressed) {isRestartPressed = false; deathNumber++; map.restartStage();}

    if (isResetPressed) { isResetPressed = false; resetTime = gameTime; resetGame = true;}

    if (isHelpPressed) {isHelpPressed = false; clueMessage = currentStage.getHelp();}.

}
```

figure 6, the mouse inputs

If the restart button is clicked, we increase the deathNumber and call map.restartStage().

If the reset button is clicked, we update resetTime, and set resetGame true.

If the help button is pressed, we set clueMessage to the current stage's help string.

After all three, the related isPressed value is set false

Checking if player will hit a spike

After input taking, player has updated xMoveDirection and yMoveDirection values, which we will use for detecting the player's next position. Because the player's x velocity is constant and player moves on the x axis only when arrow keys are pressed, the next x is calculated by:

nextX = x + direction * xVelocity * timeStep (i)

Where direction is -1 or 1 depending movement is to left or right, and the timeStep is the delta time between frames.
As for the next y, because the y velocity is updated each frame due to gravity, the next y is calculated by:

nextY = y + yVelocity * timeStep (ii)

We use these equations to assign the new nextX and nextY variables as shown in *figure 7*.

```
double nextX = player.getX() + player.getXMoveDirection()[0]*currentStage.getVelocityX()*TIME_STEP;

double nextY = player.getY() + player.getVelocityY()*TIME_STEP;
```

figure 7, assignment of nextX and nextY

Now we can check for spike collisions using map.checkSpikeCollisions(nextX, nextY). If the return value is true, we kill the player by resetting the stage using map.restartStage() and increasing deathNumber.

Movements
We use map.movePlayer() to move the player to its next position. (Helper methods explained later)

Drawing
We finally have the updated position values, it is time to draw everything. To draw the map, we use the map.draw() and for the bottom screen, drawBottomScreen(clueMessage) methods. We call these methods after clearing the screen, and after calling the methods, we use StdDraw.show() to make all the changes visible.

Time updates
After everything is drawn and updated for the current frame, we update the game time. We set gameTime datafield to the overall time passed from the start, it is set as the following: gameTime = (int) (StdDraw.currentTimeMillis() - startTime). Lastly we pause for frameDuration.

After the time updates, the while loop starts from the beginning, from checking if stage is passed again.

When the while loop breaks
If the stage is passed and the passed stage was not the last stage, the stage passed animation is drawn for two seconds using drawStagePassed(), then the main stages for loop gets to the next currentStage.

When the for loop breaks
If all the stages are passed, the winning screen is drawn until player presses the keys Q or A. We draw the screen with drawWinScreen(), and check input with StdDraw.isKeyPressed(). If Q is pressed, the game ends with System.exit(0) command, and if A is pressed, the game will reset play again, for this, the same actions with the reset button part are taken except the drawing reset screen.

Helper Methods

In the play method, certain methods are called to get information or change the values of the objects in the game. Such methods and the methods they use when they work are now explained in order of their mentions in the play method.

Collision Methods

The first method mentioned in play method is the changeStage method inside Map. This method checks if the player is in the boundaries of the exit pipe, and to understand how this works, we first need to understand how the main collision methods in Map work.

isIn
The isIn method is the most basic collision method, but it is the corner stone of the collision checking. It takes double x and y values with an int[] or double[] coordinates as input and checks if point (x, y) resides in coordinates, which is the array in the format: {x0, y0, x1, y1}, which represents a rectangular region with bottom left corner (x0, y0) and top right corner (x1, y1). The checking is done according to the following inequalities:

x > coordinates[0] (iii)	**x < coordinates[2] (iv)**	isIn returns true if all of these equations are satisfied.
y > coordinates[1] (v)	**y < coordinates[3] (vi)**	

checkCollision

The checkCollision method checks if the player with given double nextX, nextY values will intersect with the given int[] or double[] obstacle region. The given array and all the region array throughout the code obeys the format explained in the previous part. The checking is done as shown in *figure 8*:

```
//indicates if the next position intersects with a region
public boolean checkCollision(double nextX, double nextY, int[] obstacle) {  3 usages
    //if any corner gets into region, return true
    return (isIn( x: nextX+ HALFWIDTH,  y: nextY + HALFHEIGHT, obstacle) ||
            isIn( x: nextX- HALFWIDTH,  y: nextY + HALFHEIGHT, obstacle) ||
            isIn( x: nextX- HALFWIDTH,  y: nextY+HALFHEIGHT, obstacle) ||
            isIn( x: nextX+ HALFWIDTH,  y: nextY-HALFHEIGHT, obstacle)) ||
            nextX<X_SCALE[0] || nextX>X_SCALE[1] ||
            nextY<Y_SCALE[0] || nextY>Y_SCALE[1]; //check if player gets out of stage as well
}
//takes input array as double[]
public boolean checkCollision(double nextX, double nextY, double[] obstacle) {  1 usage
    return (isIn( x: nextX + HALFWIDTH,  y: nextY + HALFHEIGHT, obstacle) ||
            isIn( x: nextX - HALFWIDTH,  y: nextY + HALFHEIGHT, obstacle) ||
            isIn( x: nextX -HALFWIDTH,  y: nextY + HALFHEIGHT, obstacle) ||
            isIn( x: nextX + HALFWIDTH,  y: nextY - HALFHEIGHT, obstacle)) ||
            nextX<X_SCALE[0] || nextX>X_SCALE[1] ||
            nextY<Y_SCALE[0] || nextY>Y_SCALE[1];
}
```

figure 8, checkCollision

Checks if any corner's next position will get inside the region with isIn function. Checks for player getting out of the playable area as well. Two method calls are available, one with the region array in type int[] one with double[].

With these two methods, we can now define the obstacle spesific collision methods.

changeStage

Returns checkCollision(player.getX(), player.getY(), exitPipe[1]), which is the player whereabouts related to exit pipe.

checkDoorCollision

Checks if the input nextX, nextY will collide to the door by returning checkCollision(nextX, nextY, door)

checkSpikeCollision

Checks collision with nextX, nextY for each spike by putting spikes into a for each loop and returning true if checkCollision(nextX, nextY, spike) is true, returns false if no spike is collided.

checkButtonPressed

Sets isButtonPressed to the return value of checkCollision(nextX, nextY, button), returns nothing.

Drawing Methods

The next method mentioned in the play method is drawResetScreen. While at explaining this, it is sensible to explain all other drawing methods as well. The draw methods reside in three classes: Game, Map, Player.

The method in the Player class is draw()

- draw(int stageIndex)

Draws the png file of player on (x, y) with correct width and height. There are two png files of player, one looking right and one looking left. The draw methods draws the suitable file depending on stageIndex(draws the player reversed in stage 2) and lookDirection (states player is looking left or right by having values -1 or 1).

The methods in the Map class are draw(), drawArrayedRectangles(), drawRectangle() and drawSpikes()

- drawRectangle(int[] rectangle) / drawRectangle(double[] rectangle)

Draws the rectangular region which obeys the explained format to the map.

• drawArrayedRectangles(int[][] array)
Calls drawRectangle() for each rectangle in array.

• drawSpikes()
Draws the spike png file to the spike areas in spikes. It also rotates the png file according to spikeAngles and scales the width and height of the image to the spike area.

• draw()
Draws all the objects in map. Uses player.draw to draw player and calls the previous three methods to draw the following objects: door, obstacles(walls), button, buttonFloor, startPipe, exitPipe, spikes.

For the door, the draw function calls slideDoor() to slide the door if needed. slideDoor() decreases door's y values by DOOR_SPEED*TIME_STEP as long as isDoorOpen is true and door's upper y coordinate is bigger than doorFloor. After altering door's coordinates, we draw the door by setting pen color to doorColor and using drawRectangle(door)

As for the obstacles, we call drawArrayedRectangles, but before that, we need to set a new obstacles array because the array at hand contains two blocks residing on top of the exit pipe, blocking the player's entrance, which are not supposed to be drawn, but are useful to be there. Therefore we create a new int[][] obstaclesDrawn, and fill it with all the arrays but the last two in obstacles using System.arraycopy(obstacles, 0, obstaclesDrawn, 0, obstacles.length-2). Then we draw these obstacles after setting the pen color to stage.getColor().

The rule with the button is, the button is drawn only if isButtonPressed is false, as the button should not be seen when pressed.

The button floor, pipes, player and spikes are drawn by setting the color and calling the related methods as usual.

The methods in the Player class are drawBottomScreen(), drawResetScreen(), drawStagePassed() and drawWinScreen().

• drawBottomScreen(String clue)
Draws the bottom screen with help, reset, restart buttons and prints the information regarding deathNumber, stageIndex, level number(always 1), the clue message which is provided by clue and timer on the bottom screen. The timer is drawn using the output of convertTime(gameTime). convertTime(timeElapsed) converts timeElapsed which is in milliseconds to a String of format minutes : seconds : milliseconds.

• drawResetScreen()
Draws the reset banner that has "RESETTING…" written on. Runs a while loop for two seconds, The passage of time is controlled with the boolean value of inequality (vii).

(System.currentTimeMillis() - startTime) - gameTime < 2000 (vii)

Because gameTime is not updated while this loop runs, the amount of time passed since the loop started is equal to the left side of the inequality.

• drawStagePassed()
Uses the same principle as drawResetScreen. Draws "You Passed The Stage" on top of "But is the level over?!" On a green banner. Increases startTime by 2000 to compansate for the time passed during this screen.

• drawWinScreen()
Draws "CONGRATULATIONS YOU FINISHED THE LEVEL" on top of "PRESS A TO PLAY AGAIN" which is on top of "You finished with deathNumber deaths in convertedTime" where convertedTime is the return value of convertTime(gameTime).

movePlayer()

Here is the last helper used in play. This method checks if the player is able to move to nextX and nextY(found in equations (i) and (ii)). We first create the movement variables: xMoveDirection, yMoveDireciton, x, y, velocityX, velocityY, nextY and gravity, assigning them their values from player and stage.

We will first set the velocity y and x position values using the input from moveDirection variables. But first we need to check the player will not collide to an object, or the walls would be passable.

```
public boolean checkDirectionCollision(double nextX, double nextY, int[] direction) {  4 usages
    for (int[] obstacle : obstacles) {
        if (direction[1] != 0) { //vertical check (top-bottom)
            if (
                    isIn( x: nextX - HALFWIDTH,  y: nextY + direction[1]* HALFHEIGHT, obstacle) ||
                    isIn( x: nextX + HALFWIDTH,  y: nextY + direction[1]* HALFHEIGHT, obstacle)
            ) {return true;}

            if (direction[1] == 1) {if (nextY>Y_SCALE[1]) {return true;}} //check y being out of bounds
            else {if (nextY<Y_SCALE[0]) {return true;}} //check y being out of bounds
        }
        if (direction[0] != 0){ //horizontal check (left-right)
            if (
                    isIn( x: nextX + direction[0]* HALFWIDTH,  y: nextY+ HALFHEIGHT, obstacle) ||
                    isIn( x: nextX + direction[0]* HALFWIDTH,  y: nextY- HALFHEIGHT, obstacle)
            ) {return true;}

            if (direction[0] == 1) {if (nextX>X_SCALE[1]) {return true;}} //check x being out of bounds
            else {if (nextX<X_SCALE[0]) {return true;}} //check x being out of bounds
        }
    }
    return false;
}
```

figure 10, checkDirectionCollision

Checks if the player's corners on its given side(direction provides the side) is going to collide with any obstacle. Also checks for the getting out of playable area of the given direction.

```
//if player is falling:
if (velocityY<0) {
    //if player will crash to the ground, set velocityY = 0
    if (checkDirectionCollision(x, nextY, DIRECTIONS[3])) {
        player.setVelocityY(0);
    } else { //if player will not crash to ground, player will move
        player.setY(nextY);
    }
} else { // velocityY >= 0
    //if player will crash to the ceiling, set velocityY = 0
    if (checkDirectionCollision(x, nextY, DIRECTIONS[2])) {
        player.setVelocityY(0);
    } else { //if player will not crash to the ceiling, player will move
        player.setY(nextY);
    }
}
fall(); //gravity's effect
//the velocityY while the player is standing on top of an object will be set 0,
//then fall() sets it to gravity*timeStep
//this is useful as it keeps velocityY negative, so its nextY is not where it stands but a little below
//this nextY will intersect with obstacles,
//so the collision functions will detect before player collides the object
```

figure 11, the assignment of next position/velocity

```
//if player will not crash into walls or the door, set its nextX as its x
if (!checkDirectionCollision(nextX, y, xMoveDirection) && !checkDoorCollision(nextX, y)) {
    player.setX(nextX);
}
//jump input or stage 3:
if (yMoveDirection == DIRECTIONS[2]) {
    //for the player to be able to jump, it must be in touch with and object and its velocityY must be 0
    //for smoother animation, we check if velocityY<0.00001,
    //and also because the velocityY while standing on an object
    //is equal to gravity*timeStep(explained later), we check for velocityY-gravity*timeStep<0.00001
    //also we check standing on an object requirement with bottom direction collision function
    if ((velocityY - gravity * TIME_STEP <= 0.00001) && (checkDirectionCollision(x, nextY, DIRECTIONS[3]))) {
        jump();
    }
}
player.setXMoveDirection(DIRECTIONS[4]); //reset the xMoveDirection for next input
player.setYMoveDirection(DIRECTIONS[4]); //reset the yMoveDirection for next input

//update the y variables after jump
velocityY = player.getVelocityY();
nextY = y + velocityY * TIME_STEP;
```

figure 12, the assignment of next y

For this checking, we need a better collision method, as the one in hand provides us a limited information about the player's interaction with obstacles. We would like the player to be able to move right or left even though it will hit the ceiling in its next position. This means we need a collision method that provides a directional information as well. Define checkDirectionCollision (*figure 10*)

With this method at hand, we check the movabililty of player to assign it new x position and y velocity according to input. We check right and left collisions for x inputs, and for the y inputs, we check if the player is standing on an obstacle and its y velocity is 0. The setting of velocities according to input is shown in *figure 11*.

The x position is updated if there will be no collision to walls or the door. For the y input, the player is set to jump if its yMoveDirection is {0, 1} (Which is DIRECTIONS[2]). We check for standing on obstacles with bottom checkDirectionCollision, and check if the velocityY is near 0 for a smooth animation.

The gravity's effect on velocityY is added as the last step, therefore we need to subtract gravity*TIME_STEP from it when comparing with 0. We finally reset the moveDirections, because we already processed this input and do not want to process it in the next frame.

After assigning x and velocityY, only assigning y is left. We check if the next position of y is possible once again. We check for collisions depending on on the moving direction of player on y-axis, which is the direction of velocityY. We reset the velocity if player will collide into objects, and if not, we update the y position to nextY. These changes are made as shown in *figure 12*.