# SNAKE 3D GAME

**Student's Name:** Anton Smedholm
**Student Number:** 907271
**Degree Program:** ELEC
**Year of Studies:** 4th
**Date:** 22.04.2024

## 1. General Description

The Snake 3D Game is a simple 3D representation of the classic Snake game where the player controls a snake to eat food and grow in size while avoiding collisions with walls or itself. The game is implemented in Scala using the ScalaFX library for the graphical user interface. The project aimed to create a functional game with basic gameplay mechanics and a simple user interface.

## 2. User Interface

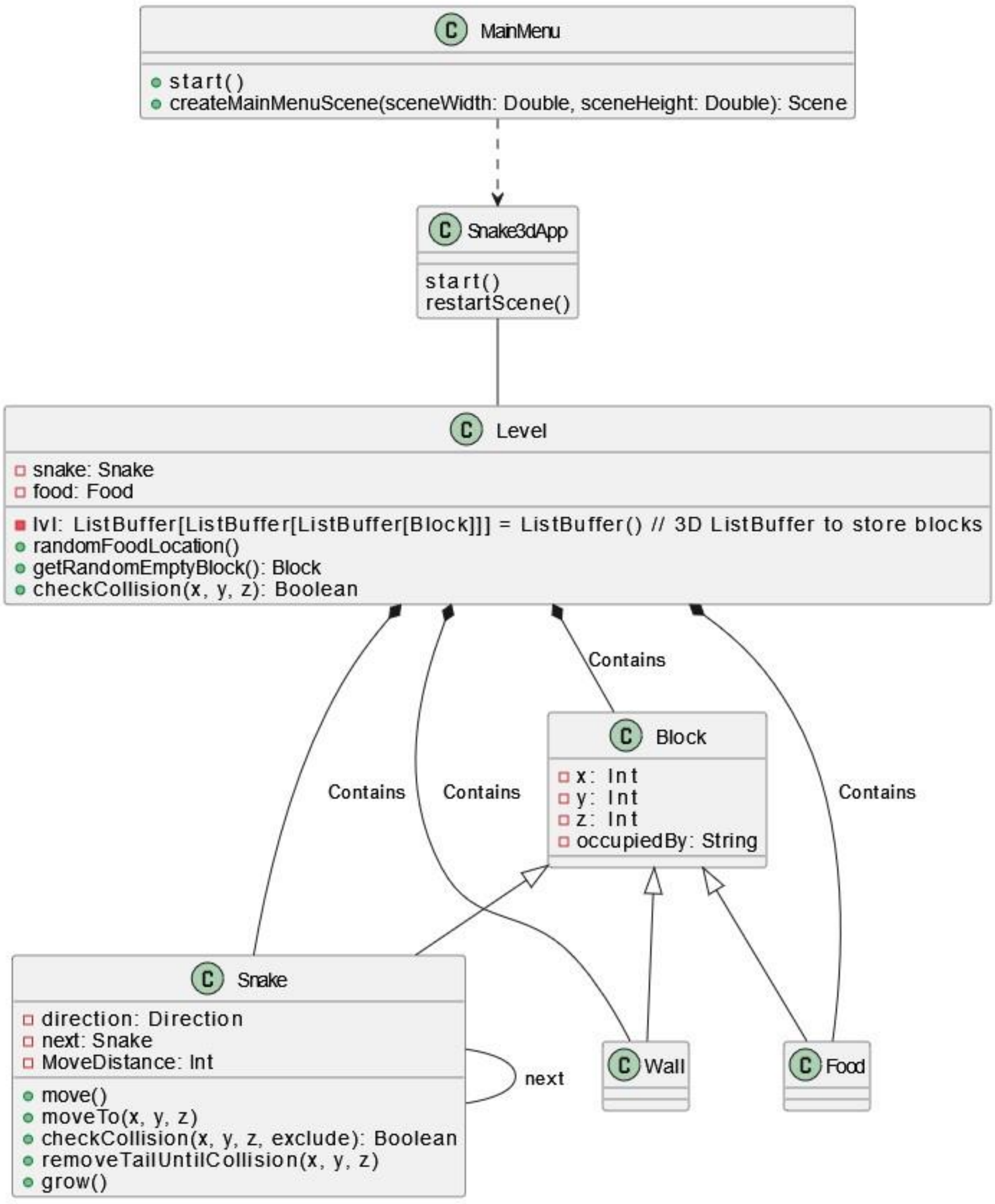
To launch the game, execute the MainMenu object in MainMenu.scala file. The main menu window will appear with "Start" and "Exit" buttons. Clicking the "Start" button will launch the 3D game scene where the snake can be controlled using the W, S, A, D keys for movement in X, Y-axis and Q and E for movement in Z-axis. The goal is to eat food items represented by red cubes to increase the snake's length. The game restarts if the snake collides with game field's edges or the walls.

## 3. Program structure

1. MainMenu.scala
   - 1.1. start() : creates stage with title and sets initial scene width and height.
   - 1.2. createMainMenuScene(sceneWidth: Double, sceneHeight: Double): Scene: Takes width and height as arguments for the future scene that it returns. Creates scene with Start and Exit buttons. Clicking Exit closes the window. Clicking Start launches the game through Snake3dApp.
2. Snake3dApp.scala
   - 2.1. start() : Creates stage and scene using Level-class. Sets location and angle for camera.
   - 2.2. updateFoodCounterLabel() : updates counter label for scene.
   - 2.3. snakeAteFood(): increases food counter by one.
   - 2.4. restartScene(): sets all values to default and restarts scene.
   - 2.5. handleKeyPress(): detects key presses and changes snakes direction. Prevents movement to opposite direction of current direction.
   - 4. Level.scala
     - 4.1. Level(file: String) : Takes text file's name as argument. Reads file and creates level according to it. Adds empty boxes, snake and walls to scene.
     - 4.2. randomFoodLocation() : Places food to random empty location.
     - 4.3. getRandomEmptyBlock() : Chooses different blocks until it finds empty one.
     - 4.4. checkCollision(x: Int, y: Int, z: Int) : Takes coordinates as arguments. Check if these coordinates collide with any snake's part.
   - 5. Block.scala
     - 5.1. Block(nx: Int, ny: Int, nz: Int) : Takes coordinates as parameters and creates empty boxes.
   - 6. Snake.scala
     - 6.1. Snake(var nx: Int, var ny: Int, var nz: Int, var level: Level) : Takes coordinates and level object as parameters. Creates snake's head at given coordinates. Snake is a linked list. Variable next is snake's next part.
     - 6.2. move(): moves snake at given direction.
     - 6.3. moveTo(x: Int, y: Int, z: Int) : Moves the snake to the specified coordinates if possible. If snake moves to food's location, removes food and calls grow(), level.randomFoodLocation() and Snake3dApp.snakeAteFood() functions.

6.4. checkCollision(x: Int, y: Int, z: Int, exclude: Snake = null) : Checks if there is a collision at the specified coordinates.
6.5. removeTailUntilCollision(x: Int, y: Int, z: Int) : Removes tail segments until a collision occurs at specified coordinates.
6.6. grow() : Increases the length of the snake by adding a new segment.

7. Wall.scala
   7.1. Wall(x: Int, y: Int, z: Int): Creates wall in given coordinates as white block.
8. Food.scala
   8.1. Food(x: Int, y: Int, z: Int) : Creates food in given coordinates as red block.
9. level.txt: number represents current deep of level. Symbols until next number form current layer: #-for empty block, W- for wall and S-for Snake's head.



C MainMenu
start()
createMainMenuScene(sceneWidth: Double, sceneHeight: Double): Scene
C Snake3dApp
start()
restartScene()
C Level
snake: Snake
food: Food
lvl: ListBuffer[ListBuffer[ListBuffer[Block]]] = ListBuffer() // 3D ListBuffer to store blocks
randomFoodLocation()
getRandomEmptyBlock(): Block
checkCollision(x, y, z): Boolean
Contains
Contains
Contains
Contains
C Block
x: Int
y: Int
z: Int
occupiedBy: String
C Snake
direction: Direction
next: Snake
MoveDistance: Int
move()
moveTo(x, y, z)
checkCollision(x, y, z, exclude): Boolean
removeTailUntilCollision(x, y, z)
grow()
next
C Wall
C Food

# 10. Algorithms

- **Snake Movement:** The snake moves one step in its current direction on each game tick. Q,W,E,A,S,D button change direction of snake's movement. W, S, A, D are used for movement in x,y – axis and Q, E for movement in z-axis. W changes direction to UP, S – DOWN, A – LEFT, D – RIGHT, Q – FURTHER and E – CLOSER. Snake is a linked list and when it moves, head gets new location and all next parts get previous coordinates of previous parts.
- **Snake's Collision Detection:** Level consists of blocks. Each block has variable occupiedBy and it's value is EMPTY, SNAKE, WALL or FOOD. Before moving snake we check the occupiedBy value of the next block. If it is WALL, game restarts, If it is FOOD, functions grow(), level.randomFoodLocation() and Snake3dApp.snakeAteFood() are called. Grow function goes throw snake as linked list until it finds tail and creates new snake's part in place tail was before while other parts are already mode to new locations.

  To Find collision with itself we use function Snake.checkCollision that goes throw all snake linked nodes until tail and checks that none of nodes has same coordinates as destination. If collision occurs, function removeTailUntilCollision finds colliding part by coordinates and remove all parts from it until tail.

- **Foods at random positions, After eating a food, the location of new food object cannot overlap with snake body:** Function randomFoodlocation searches picks random blocks until it finds block with occupidedBy's value EMPTY and using level.checkCollision checks that it is not colliding with any of snake parts. If it finds such block it places food there and changes its occupiedBy value to FOOD.
- **Switching the snakehead direction to the opposite should be disabled (as it ends the game instantly)** : Function handleKeyPress does nothing if pressed button corresponds to movement in opposite direction of current direction. For example: If current direction is DOWN, pressing W for moving up is ignored.
- **Game area must be bounded:** If next block where snake is going to move does not exist, function moveTo() catches error and restarts the game.

# 11. Data structures

**Linked list:** Snake's body is linked list which simplifies going throw snake and its growing. For example if snake would be a simple list, there would be maximum size for it and it would use more memory to store several empty nodes. It could be fixed by recreating snake list with +1 size every time it eats, but it would be unnecessarily more difficult. Mutable structure.
**3D ListBuffer:** It was used to store all blocks added to level. It simplifies accessing each block using only block's coordinates. Each block can be accessed like this List(z)(y)(x).
1D List could be used, but it would need to check every node to find block with needed coordinates.
Mutable structure

## 12. Files

There is level.txt file using which Level-class creates map. Each number corresponds to layer in z-axis. Symbols after number correspond to blocks in x,y-axis. #- EMPTY, W-WALL, S-Snakehead. For example current level.txt creates 4x4x4 map, but if we want 2x2x2 map, it must be written like this:

```
1

##

##

2

##

##
```

## 13. Testing

Testing involved manual gameplay testing to ensure proper snake movement, collision detection, and food consumption. Testing was done mostly according to plan, but it did not include all, because some things like save files  were not implemented.
Manual tests included:
System testing:
1.User Interface Testing:
• Ensure all user input methods are functional and responsive.
• Verify that controls (keyboard commands) work as expected.
• Check the GUI for proper display and information presentation.
2. Functionality Testing:
• Test snake movement in all directions to ensure proper control.
• Validate snake growth upon eating food.
• Verify collision detection with walls, snake body, and food.
• Check if the game ends correctly upon collisions.
3. File Management Testing:
• Test loading a predefined map from a file to ensure correct placement of walls.
4. Boundary Testing:
• Test the boundaries of the game area to ensure the snake does not move outside.
Unit testing was not used because majority of functions does not return anything and such tests would be hard to implement.

## 14. Known bugs and missing features

All moderate level features were implemented.
Bugs:

1. Sometimes new food does not spawn.
2. Sometimes game just restarts even if snake is not colliding.

It was planned to make project on difficult level, but because there is barely information on scalafx, I decided that grade for difficult level does not correspond needed effort.

## 15. 3 best sides and 3 weaknesses

Best:

1. It is easy to add new classes to game with no need to rewrite game.
2. It is easy to create 3D level using text files.
3. It has eaten food counter.

Weaknesses:

1. Walls block the view. Could be improved by changing their transparency.
2. Game does not have end. Could be improved by comparing amount on empty blocks in the beginning and length of snake in the end.
3. Snake's head can be easily lost. Could be improved by making snake elliptic or by adding edges to snake body.

## 16. Deviations from the plan, realized process and schedule

Schedule:

Day 1: Creating Level from file. Fixing Camera.

Day 2: Adding snake to map. Making snake move.

Day 3: Adding collision detection for snake and food.

Day 63: Fixing bugs and adding comments. Writing document.

Plan's schedule deviates from project and it is biggest difference. During exam week I had no exams or  tasks, so I used this time on project and finished it in 3 days, but completed everything in planned order. During those days I decided to not implement difficult level features as grade differences between moderate and difficult were not equivalent to effort of adding difficult features. During this process I learned that nobody codes in scala.

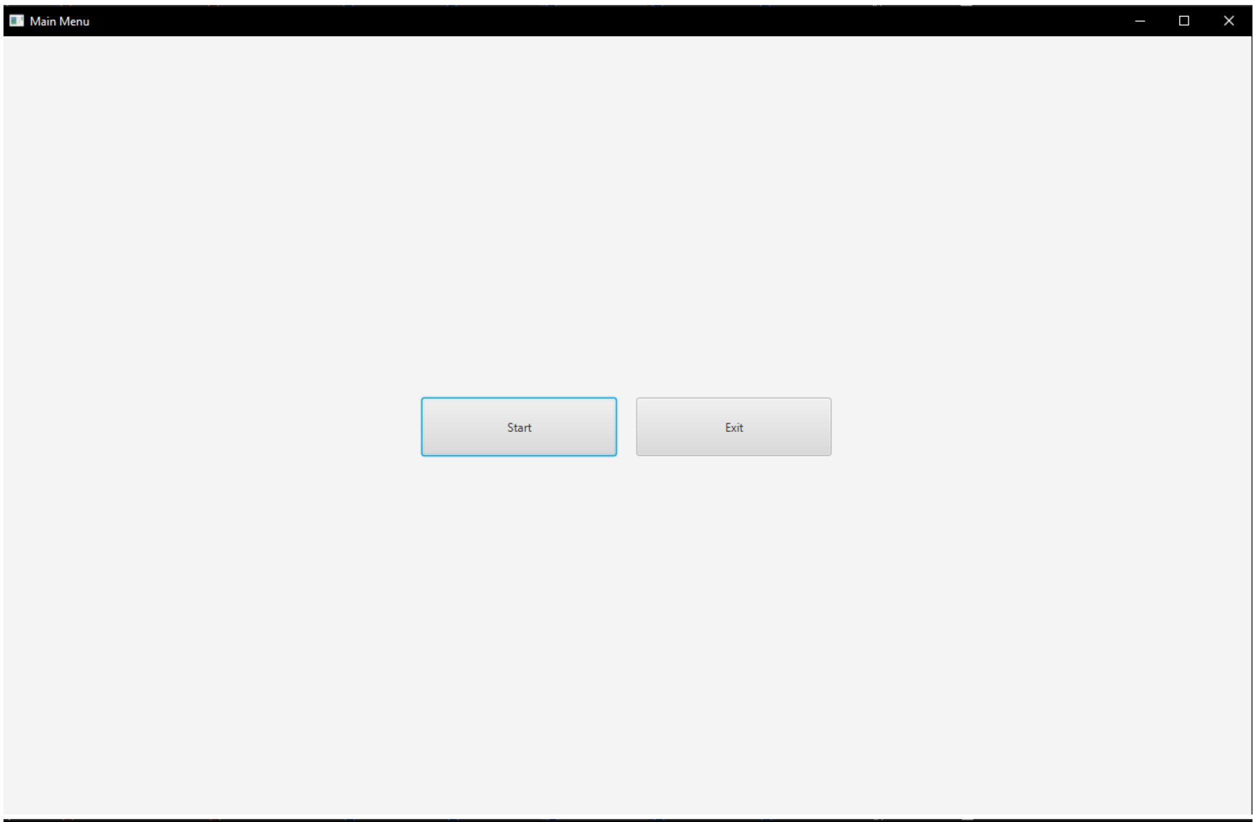
## 17. Final evaluation

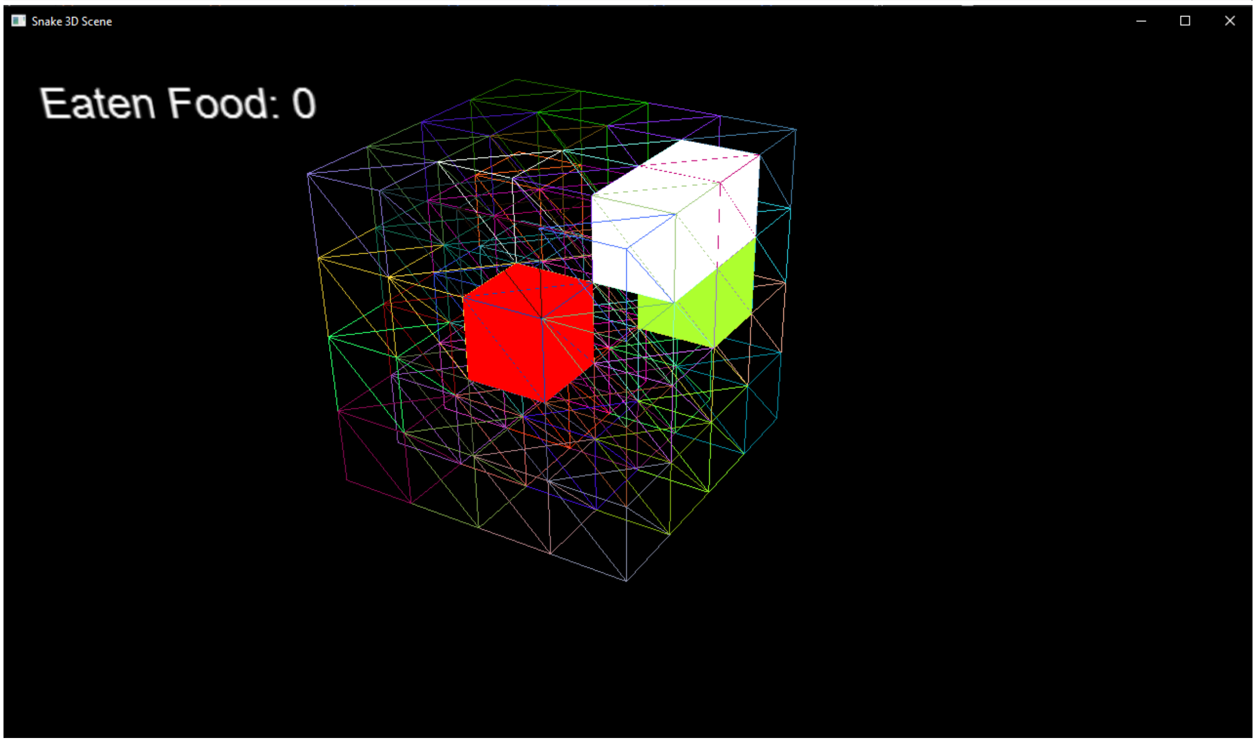
I think that my project deserves the highest grade for moderate difficulty as it has all needed features. I do not think that it has weak aspects. I choosed best data structures for it and finished it in 3 days as I already had picture in my head of how implement all its features when I choosed this topic. Only improvements I can imagine are described in weaknesses. Project is very suitable for changes and extensions as it only needs to add handling case to level class and checkCollision functions.
If I would started the project again from the beginning, I would never use scala.

## 18. References and code written by someone else

I used course materials are examples. Scala.IO was used for reading txt file, scalafx for creating scene and scala.collection was used to create 3DListBuffer.
Code could not be copied from another web site, because no one uses scala. Program and data structures were created by myself. GhatGPT was used as debugger to improve errors and it helped to create scene because I did not find suitable examples in web.

## 19. Appendices





## MainMenu.scala:

```
package snake

import scalafx.application.JFXApp3
import scalafx.application.JFXApp3.PrimaryStage
import scalafx.scene.Scene
import scalafx.scene.control.Button
import scalafx.stage.FileChooser
import scalafx.stage.FileChooser.ExtensionFilter
import scalafx.geometry.{Insets, Pos}
import scalafx.scene.layout.{HBox, VBox}

object MainMenu extends JFXApp3 {

  override def start(): Unit = {
    stage = new PrimaryStage {
      title = "Main Menu"
      scene = createMainMenuScene(1280, 800) // Set initial scene width and
height here
    }
  }

  def createMainMenuScene(sceneWidth: Double, sceneHeight: Double): Scene = {
    val startButton = new Button("Start")
    val exitButton = new Button("Exit")

    // Set button sizes
    val buttonWidth = 200
    val buttonHeight = 60
    startButton.prefWidth = buttonWidth
    startButton.prefHeight = buttonHeight
    exitButton.prefWidth = buttonWidth
    exitButton.prefHeight = buttonHeight

    // Set button actions
    startButton.onAction = _ => Snake3dApp.start()
    exitButton.onAction = _ => stage.close()

    // Create an HBox to center the buttons horizontally
    val buttonBox = new HBox(20) {
      alignment = Pos.Center
      children = Seq(startButton, exitButton)
    }

    // Create a VBox to center the HBox vertically
    val layout = new VBox {
      alignment = Pos.Center
      padding = Insets(50)
      children = Seq(buttonBox)
    }

    new Scene(layout, sceneWidth, sceneHeight) // Create scene with specified
width and height
  }
}
```

## Snake3dApp.scala:

```scala
package snake

import com.sun.javafx.scene.traversal.Direction
import scalafx.Includes._
import scalafx.application.JFXApp3
import scalafx.geometry.Point3D
import scalafx.scene.{AmbientLight, Group, PerspectiveCamera, Scene}
import scalafx.scene.input.{KeyCode, KeyEvent}
import scalafx.scene.paint.{Color, PhongMaterial}
import scalafx.scene.shape.{Box, DrawMode}
import scalafx.scene.transform.{Rotate, Translate}

import scala.util.Random
import scalafx.animation.AnimationTimer
import scalafx.animation.PauseTransition
import scalafx.util.Duration
import scalafx.scene.control.Label
import scalafx.scene.text.Font

// Define the main object for the Snake 3D application
object Snake3dApp extends JFXApp3 {

  // Define constants for box movement
  val MoveDistance = 100
  var canMove = true
  val moveDelay = Duration(1000) // Adjust the delay duration as needed
  var gameStarted = false // Flag to check whether the game has started or
not
  var snake: Snake = _
  var root = new Group()
  var eatenFoodCount = 0 // Counter for eaten food items
  var foodCounterLabel: Label = _ // Label for eaten food count

  // Override the start method of JFXApp3
  override def start(): Unit = {
    // Load level from file
    var level = Level("level.txt")
    root = new Group(level)
    snake = level.snake

    // Create a camera with initial translation and rotation
    val camera = new PerspectiveCamera(false)
    camera.getTransforms.addAll(
      new Translate(-200, -200, 600), // Initial translation
      new Rotate(60, Rotate.YAxis),     // Rotate around Y-axis by 45 degrees
      new Rotate(-30, Rotate.XAxis)     // Rotate around X-axis by -30 degrees
    )

    // Create ambient light
    val ambientLight = new AmbientLight(Color.White)

    // Add the camera and ambient light to the root group
    root.children.addAll(camera, ambientLight)

    // Create the 3D scene
    val scenee = new Scene(root, 1280, 720, true)
    scenee.fill = Color.Black

    // Set the camera for the scene
    scenee.camera = camera

    // Add eaten food counter label
    foodCounterLabel = new Label(s"Eaten Food: $eatenFoodCount")
    foodCounterLabel.layoutX = -600 // Position from left
    foodCounterLabel.layoutY = -350 // Position from top
    foodCounterLabel.font = Font("Arial", 24) // Set font size
    foodCounterLabel.textFill = Color.White // Set text color to white
    foodCounterLabel.transforms += new Rotate(60, Rotate.YAxis)
    root.children.add(foodCounterLabel)

    // Handle keyboard input for box movement
    scenee.onKeyPressed = (event: KeyEvent) => handleKeyPress(event)

    // Set up the stage
    stage = new JFXApp3.PrimaryStage {
      title = "Snake 3D Scene"
      scene = scenee

      // Start the animation timer
      AnimationTimer { currentTime =>
        if (gameStarted && canMove) {
          snake.move()

          // Check for collisions or other game logic here
          updateFoodCounterLabel()


          canMove = false
          new PauseTransition(moveDelay) {
            onFinished = _ => canMove = true
          }.play()
        }
      }.start()
    }
  }

  // Update the eaten food counter label
  def updateFoodCounterLabel(): Unit = {
    foodCounterLabel.text = s"Eaten Food: $eatenFoodCount"
  }

  // Increase food counter
  def snakeAteFood(): Unit = {
    eatenFoodCount += 1
  }

  // Restart the game scene
  def restartScene(): Unit = {
    // Stop the current animation timer
    stage.getScene.getWindow.getOnCloseRequest match {
      case handler: javafx.event.EventHandler[_] => handler.handle(null)
      case _ => // No close request handler found
    }

    // Clear the root group
    root.children.clear()
    eatenFoodCount = 0
    gameStarted = false
    start()
  }

  // Handle keyboard input
  def handleKeyPress(event: KeyEvent): Unit = {
    if (!gameStarted) {
      // If the game hasn't started, start it now
      gameStarted = true
    }
    event.code match {
      case KeyCode.W if snake.direction != Direction.DOWN => snake.direction
= Direction.UP
      case KeyCode.S if snake.direction != Direction.UP => snake.direction =
Direction.DOWN
      case KeyCode.A if snake.direction != Direction.RIGHT => snake.direction
= Direction.LEFT
      case KeyCode.D if snake.direction != Direction.LEFT => snake.direction
= Direction.RIGHT
      case KeyCode.E if snake.direction != Direction.NEXT => snake.direction
= Direction.PREVIOUS
      case KeyCode.Q if snake.direction != Direction.PREVIOUS =>
snake.direction = Direction.NEXT
      case _ =>
    }
  }
}
```

## Level.scala:

```
package snake

import scala.io.Source
import scalafx.scene.{Group, PerspectiveCamera, Scene}
import scala.collection.mutable.ListBuffer
import scala.util.Random

/**
 * Represents the game level, initialized from a file.
 *
 * @param file Path to the level file
 */
class Level(file: String) extends Group {

  // Read the level file to initialize the level
  val bufferedSource = Source.fromFile("src/main/scala/" + file)
  var y = 0 // Initialize y coordinate
  var z = -1 // Initialize z coordinate
  var lvl: ListBuffer[ListBuffer[ListBuffer[Block]]] = ListBuffer() // Create
a 3D ListBuffer to store blocks
  var snake: Snake = null // Initialize snake object
  var food: Food = null // Initialize food object

  // Loop through each line in the level file
  for (line <- bufferedSource.getLines) {
    if (!line.forall(Character.isDigit)) {
      // Ensure the ListBuffer has enough dimensions for z
      while (lvl.length <= z) {
        lvl += ListBuffer()
      }

      // Ensure the ListBuffer(z) has enough dimensions for y
      while (lvl(z).length <= y) {
        lvl(z) += ListBuffer()
      }

      // Loop through each character in the line
      for ((char, x) <- line.zipWithIndex) {
        char match {
          case 'W' =>
            // Create a wall and corresponding block at the specified
coordinates
```

```scala
            val wall = new Wall(x, y, z)
            children.add(wall)
            val block = new Block(x, y, z)
            block.occupiedBy = "WALL"
            children.add(block)
            lvl(z)(y) += block
          case 'S' =>
            // Create the snake and corresponding block at the specified
coordinates
            snake = new Snake(x, y, z, this)
            children.add(snake)
            val block = new Block(x, y, z)
            children.add(block)
            lvl(z)(y) += block
          case _ =>
            // Create an empty block at the specified coordinates for other
characters
            val block = new Block(x, y, z)
            children.add(block)
            lvl(z)(y) += block
        }
      }
      y += 1 // Move to the next y coordinate
    } else if (line.forall(Character.isDigit)) {
      z += 1 // Move to the next z coordinate
      y = 0 // Reset y to 0 for the next row
    }
  }
  bufferedSource.close // Close the file

  // Initialize food at a random empty block location
  randomFoodLocation()

  // Function to generate food at a random empty block location
  def randomFoodLocation(): Unit = {
    var block = getRandomEmptyBlock()
    food = new Food(block.x, block.y, block.z)
    block.occupiedBy = "FOOD"
    children.add(food)
  }

  // Function to get a random empty block
  def getRandomEmptyBlock(): Block = {
    var block =
lvl(Random.nextInt(lvl.length))(Random.nextInt(lvl(0).length))(Random.nextInt
(lvl(0)(0).length))
    while (!block.occupiedBy.equals("EMPTY") || (snake != null &&
checkCollision(block.x, block.y, block.z))) {
      block =
lvl(Random.nextInt(lvl.length))(Random.nextInt(lvl(0).length))(Random.nextInt
(lvl(0)(0).length))
    }
    block
  }

  // Function to check for collisions with the snake
  def checkCollision(x: Int, y: Int, z: Int): Boolean = {
    var current: Snake = snake
    while (current != null) {
      if (current.translateX.toInt == x && current.translateY.toInt == y &&
current.translateZ.toInt == z) {
        return true
      }
      current = current.next
    }
    false
  }
}
```

## Block.scala:

```
package snake

import scalafx.scene.paint.{Color, PhongMaterial}
import scalafx.scene.shape.{Box, DrawMode}
import scala.util.Random

/**
 * Represents a block in the game level grid.
 *
 * @param nx Initial x-coordinate of the block
 * @param ny Initial y-coordinate of the block
 * @param nz Initial z-coordinate of the block
 */
class Block(nx: Int, ny: Int, nz: Int) extends Box {
  // Initialize block coordinates
  var x = nx
  var y = ny
  var z = nz

  // Set block size
  var blockSize = 100
  width = blockSize
  height = blockSize
  depth = blockSize

  // Set block draw mode and position
  drawMode = DrawMode.Line
  translateX = x * blockSize
  translateY = y * blockSize
  translateZ = z * blockSize

  // Set random color for the block
  material = new PhongMaterial(Color.rgb(Random.nextInt(256),
Random.nextInt(256), Random.nextInt(256)))

  // Set initial occupancy status
  var occupiedBy = "EMPTY"
}
```

## Wall.scala:

```
package snake

import scalafx.scene.paint.{Color, PhongMaterial}
import scalafx.scene.shape.DrawMode

/**
 * Represents a Wall entity in the game level.
 *
 * @param x Initial x-coordinate of the wall
```

```scala
 * @param y Initial y-coordinate of the wall
 * @param z Initial z-coordinate of the wall
 */
class Wall(x: Int, y: Int, z: Int) extends Block(x: Int, y: Int, z: Int) {
  drawMode = DrawMode.Fill
  material = new PhongMaterial(Color.White)
  occupiedBy = "WALL"
}
```

## Food.scala:

```scala
package snake

import scalafx.scene.paint.{Color, PhongMaterial}
import scalafx.scene.shape.DrawMode

/**
 * Represents a Food entity in the game level.
 *
 * @param x Initial x-coordinate of the food item
 * @param y Initial y-coordinate of the food item
 * @param z Initial z-coordinate of the food item
 */
class Food(x: Int, y: Int, z: Int) extends Block(x: Int, y: Int, z: Int) {
  drawMode = DrawMode.Fill
  material = new PhongMaterial(Color.Red)
  occupiedBy = "FOOD"
}
```

## Snake.scala:

```scala
package snake

import scalafx.scene.paint.{Color, PhongMaterial}
import scalafx.scene.shape.DrawMode
import com.sun.javafx.scene.traversal.Direction
import snake.Snake3dApp.restartScene

/**
 * Represents the Snake entity in the game.
 *
 * @param nx     Initial x-coordinate of the Snake's head
 * @param ny     Initial y-coordinate of the Snake's head
 * @param nz     Initial z-coordinate of the Snake's head
 * @param level  Reference to the game level
 */
class Snake(var nx: Int, var ny: Int, var nz: Int, var level: Level) extends
Block(nx: Int, ny: Int, nz: Int) {
  drawMode = DrawMode.Fill
  material = new PhongMaterial(Color.GreenYellow)
  occupiedBy = "Snake"
  var next: Snake = null
  var direction = Direction.RIGHT
  val MoveDistance = 100
  /**
   * Moves the snake based on its current direction.
   */
  def move(): Unit = {
    direction match {
      case Direction.RIGHT => moveTo(translateX.toInt + MoveDistance,
translateY.toInt, translateZ.toInt)
      case Direction.LEFT => moveTo(translateX.toInt - MoveDistance,
translateY.toInt, translateZ.toInt)
      case Direction.UP => moveTo(translateX.toInt, translateY.toInt -
MoveDistance, translateZ.toInt)
      case Direction.DOWN => moveTo(translateX.toInt, translateY.toInt +
MoveDistance, translateZ.toInt)
      case Direction.NEXT => moveTo(translateX.toInt, translateY.toInt,
translateZ.toInt + MoveDistance)
      case Direction.PREVIOUS => moveTo(translateX.toInt, translateY.toInt,
translateZ.toInt - MoveDistance)
    }
  }

  /**
   * Moves the snake to the specified coordinates if possible.
   *
   * @param x Target x-coordinate
   * @param y Target y-coordinate
   * @param z Target z-coordinate
   */
  def moveTo(x: Int, y: Int, z: Int): Unit = {
    try {
      val block = level.lvl(z / 100)(y / 100)(x / 100)
      if (block.occupiedBy.equals("EMPTY")) {
        if (next != null && checkCollision(x, y, z, exclude = this)) {
          // Collision with snake body, handle it by removing tail segments
          removeTailUntilCollision(x, y, z)
        } else {
          if (next != null) {
            next.moveTo(translateX.toInt, translateY.toInt, translateZ.toInt)
          }
          translateX = x.toDouble
          translateY = y.toDouble
          translateZ = z.toDouble
        }
      } else if (block.occupiedBy.equals("FOOD")) {
        block.occupiedBy = "EMPTY"
        level.children.removeAll(level.food)
        grow()
        level.randomFoodLocation()
        moveTo(x, y, z)
        Snake3dApp.snakeAteFood()
      } else {
        // Collision with non-empty block (e.g., walls), restart the scene
        restartScene()
      }
    } catch {
      case _: Throwable => // Handle exceptions, for simplicity, restart the
scene
        restartScene()
    }
  }

  /**
   * Checks if there is a collision at the specified coordinates.
   *
   * @param x        X-coordinate to check
   * @param y       Y-coordinate to check
   * @param z       Z-coordinate to check
   * @param exclude Snake segment to exclude from collision check
   * @return True if collision detected, false otherwise
   */
  def checkCollision(x: Int, y: Int, z: Int, exclude: Snake = null): Boolean
= {
    var current: Snake = this
    while (current != null) {
      if (current != exclude && current.translateX.toInt == x &&
current.translateY.toInt == y && current.translateZ.toInt == z) {
        return true // Collision detected
      }
      current = current.next
    }
    false // No collision
  }

  /**
   * Removes tail segments until a collision occurs at specified coordinates.
   *
   * @param x X-coordinate of collision
   * @param y Y-coordinate of collision
   * @param z Z-coordinate of collision
   */
  def removeTailUntilCollision(x: Int, y: Int, z: Int): Unit = {
    var current: Snake = this
    var previous: Snake = null

    // Find the colliding segment
    while (current != null && (current.translateX.toInt != x ||
current.translateY.toInt != y || current.translateZ.toInt != z)) {
      previous = current
      current = current.next
    }

    // Remove tail segments until the colliding segment
    if (current != null) {
      while (current != null) {
        level.children.remove(current)
        current = current.next
      }
      if (previous != null) {
        // Update the 'next' reference of the previous segment
        previous.next = null
      } else {
        // The head itself collided, set 'next' to null
        next = null
      }
    }
  }

  /**
   * Increases the length of the snake by adding a new segment.
   */
  def grow(): Unit = {
    if (next != null) {
      next.grow()
    } else {
      next = new Snake(this.translateX.toInt, this.translateY.toInt,
this.translateZ.toInt, level)
      // Update the position based on the current direction
      direction match {
        case Direction.RIGHT => next.translateX.value -= MoveDistance
        case Direction.LEFT => next.translateX.value += MoveDistance
        case Direction.UP => next.translateY.value += MoveDistance
        case Direction.DOWN => next.translateY.value -= MoveDistance
        case Direction.NEXT => next.translateZ.value -= MoveDistance
        case Direction.PREVIOUS => next.translateZ.value += MoveDistance
      }
      level.children.add(next)
    }
  }
}
```

## level.txt:

```
1
#WW#
##S#
####
####
2
####
####
####
####
3
####
####
####
####
4
####
####
####
####
```