

COMP132: Advanced Programming

Programming Project Report

Food Chain Through Time: Simulation Design and Development

<AHMET ENES MADEN, 86843>

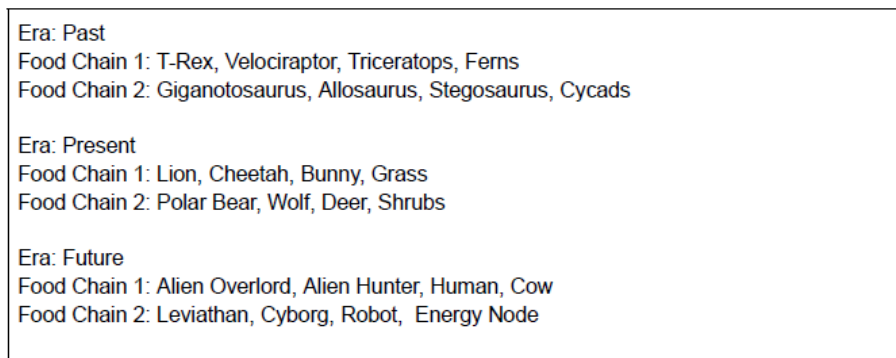
<Fall 2025>



PART 1: General Demo Information

1.1) Food Chain Configuration

The application uses external text files to configure the food chain dynamically for each era. There exist three text files with names: `past_animals.txt`, `present_animals.txt`, `future_animals.txt`. Each one represents a particular game era. Contents of these text files can be found in the Figure 1.



```
Era: Past
Food Chain 1: T-Rex, Velociraptor, Triceratops, Ferns
Food Chain 2: Giganotosaurus, Allosaurus, Stegosaurus, Cycads

Era: Present
Food Chain 1: Lion, Cheetah, Bunny, Grass
Food Chain 2: Polar Bear, Wolf, Deer, Shrubs

Era: Future
Food Chain 1: Alien Overlord, Alien Hunter, Human, Cow
Food Chain 2: Leviathan, Cyborg, Robot, Energy Node
```

Figure 1: Contents of each txt file

As shown in the Figure 1, txt files define the food chain structures for each era. For instance, `past_animals.txt` holds two different food chains, each containing an apex predator, a predator, a prey and a food name.

To implement this dynamic configuration, I implemented a method in `FileManager` class, named `loadRandomFoodChain()`. The following pseudocode summarizes how a random food chain is selected from the text files at the start of each game:

```
FUNCTION loadRandomFoodChain(mode):
    // 1. Determine file path based on selected Era
    fileName = getFileNameByMode(mode)

    // 2. Read and parse the file, take valid food chains
    validChains = []
    OPEN fileName:
        FOR each line in file:
            IF line starts with "Food Chain":
                validChains.add(parseLine(line))

    // 3. Return a random chain
    RETURN validChains[random.nextInt()]
```

The important thing about this method is that, as long as the format is same, differences in the foodchain.txt file does not affect the game execution. So this part is totally dynamic and independent of changes in character names.

1.2) Start Screen

The game launches with a very simple and user friendly start screen interface. This interface enables users to determine gameplay preferences such as era, grid size selection and specifying total number of rounds to play. The start screen is illustrated in Figure 2:

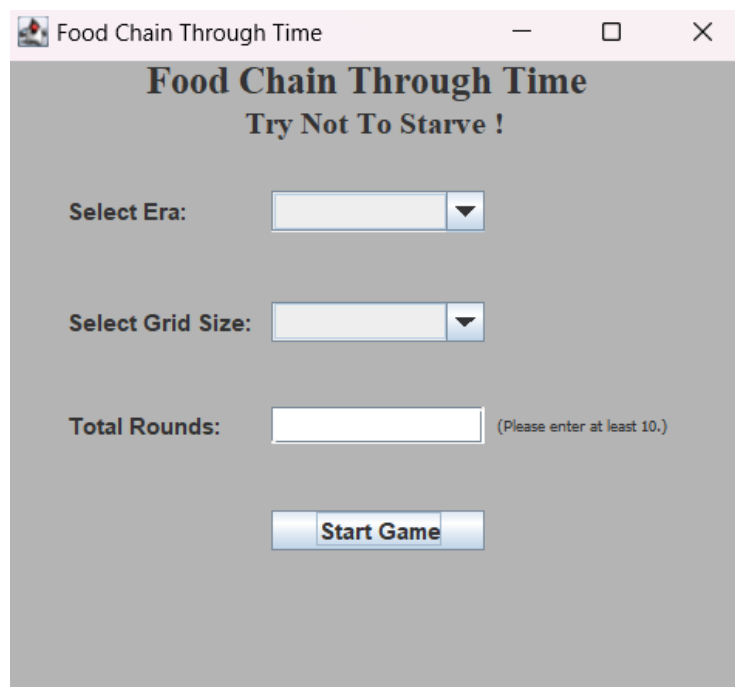


Figure 2: The Start Screen

User can chose desired era from this interface: Past, Present, Future. Also, user can choose grid size with options of 10x10, 15x15, 20x20. And number of total rounds can be entered by user at this interface, with condition of at least 10 rounds entry. The system does not enable to start game with empty or invalid entries, as shown in the following figures.

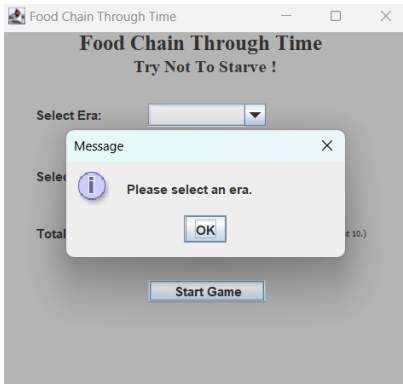


Figure 3: Invalid Era

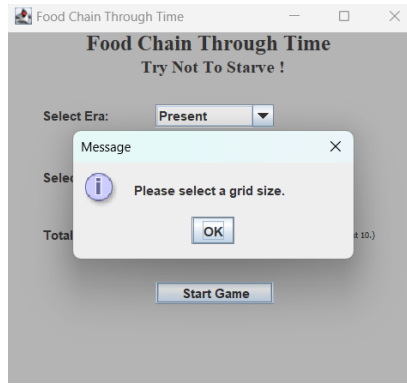


Figure 4: Invalid Grid Size



Figure 5: Invalid Number of Rounds

After selecting preferences, the user needs to click “Start Game” button to switch to gameplay screen. This button calls “startGamePlayScreen(selectedEra, gridSize, rounds)” method from MainFrame class, with taken parameters from user. Here is the pseudocode for this method:

```
FUNCTION startGamePlayScreen(era, gridSize, rounds):
    // 1. Initialize the core Game Engine
    // The engine sets up the grid and populates it with animals
    this.engine = NEW GameEngine(era, gridSize, rounds)

    // 2. Link engine to GUI components and set info panel
    gamePanel.setEngine(this.engine)
    updateLabels()

    // 3. Switch view to gameplay screen and set focus for inputs
    cardLayout.show(mainPanel, "Game ")
    gamePanel.requestFocus()
END FUNCTION
```

So, this processes ensures that the game environment is correctly instantiated without any invalid entries or states. User successfully switches to gameplay screen, the grid and characters are rendered and the game is ready to play.

1.3) Gameplay

After the “Start Game” button is clicked by user, the following interface is displayed:

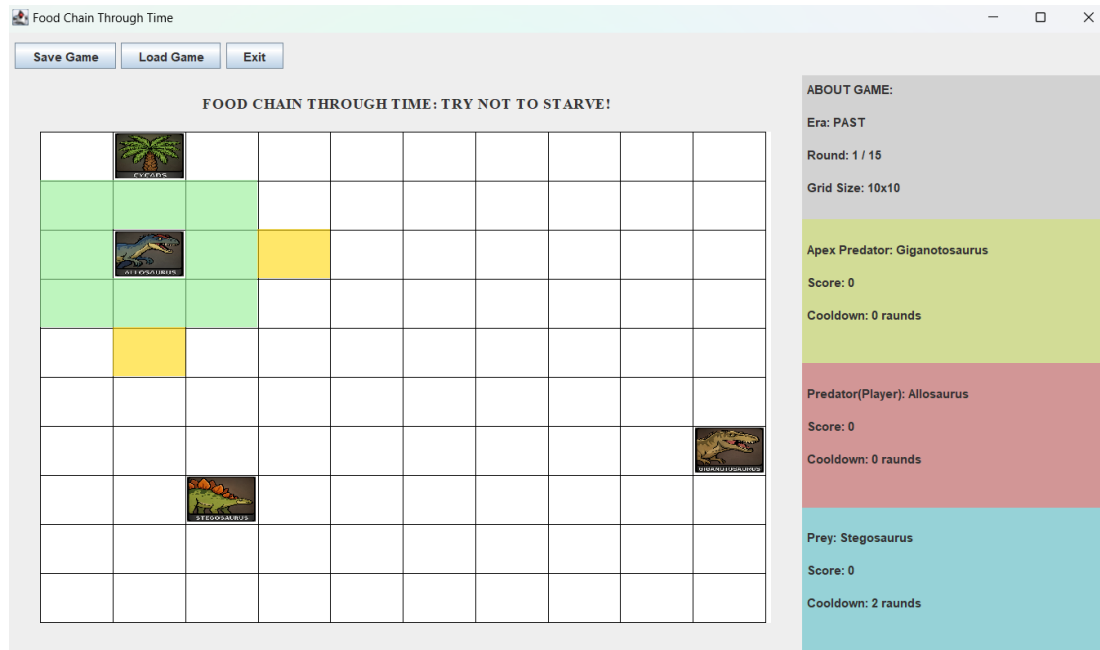


Figure 6: The Gameplay Screen

Introduction of Gameplay Interface

Upon successfully initializing the game, the gameplay screen above welcomes the player. The user controls predator character. The main objective of the predator is running away from the apex predator character and chasing the prey character. Since names and icons of characters are changing, the info panel demonstrates which one is apex predator, predator or prey; by demonstrating the names of characters. So it helps user to understand food chain better and provides a better game experience. Info panel also demonstrates each character's score and cooldown informations. Additionally this panel includes basic game information such as round info, era info and grid size that game is being played.

Gameplay:

Prey and apex predator characters are controlled by computer, each has its own decision mechanism. Predator is controlled by user. Player uses mouse clicks to move predator. Available moves for player is painted: green for standar walk ability and yellow for special movement abilities. Using a special ability cause a cooldown for a particular round time.

When it is the predator's turn, player can move to the available (painted) cells by clicking with mouse. Using `mousePressed(MouseEvent e)` from `GamePanel` class, program successfully handles mouse clicks:

```

FUNCTION mousePressed(mouseEvent):
    // 1. Takes pixel coordinates from event
    mouseX = mouseEvent.getX()
    mouseY = mouseEvent.getY()

    // 2. Determines which row and column were clicked
    clickedCol = mouseX / cellWidth
    clickedRow = mouseY / cellHeight

    // 3. The engine receives grid coordinates
    turnSuccessful = engine.processTurnWhenClicked(clickedCol,
    clickedRow)

    // 4. Update UI if the turn is successful
    IF turnSuccessful IS TRUE:
        repaint()
        MainFrame.updateLabels()
        // Check for game over condition
        IF engine.isGameOver():
            showMsgDialog("Game Over", engine.getGameResult())
        END IF
    END IF
END FUNCTION

```

With this function, mouse clicks are successfully handled. So user can move predator by just clicking as demonstrated below:

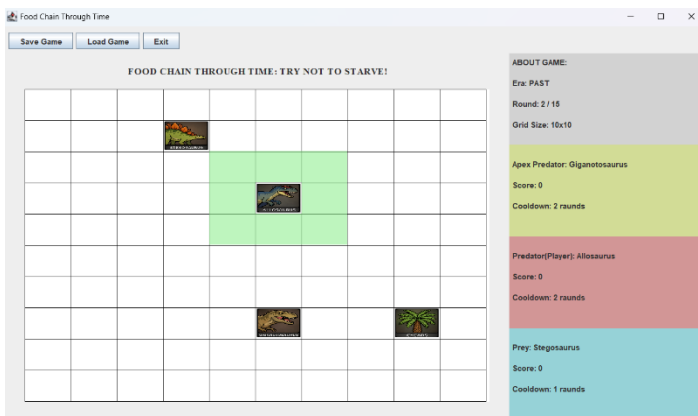


Figure 7: Before Standard Move

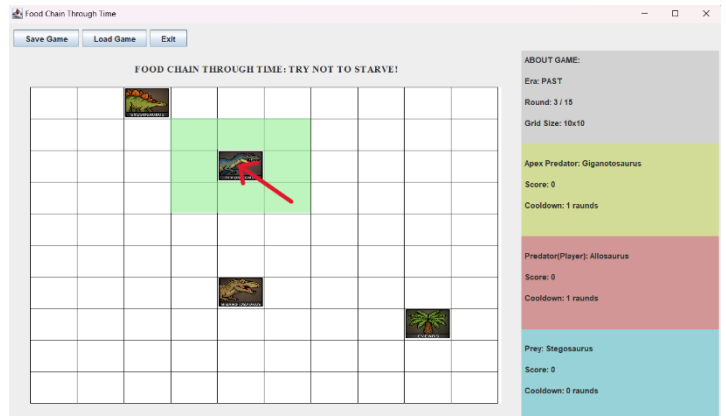


Figure 8: After Standard Move

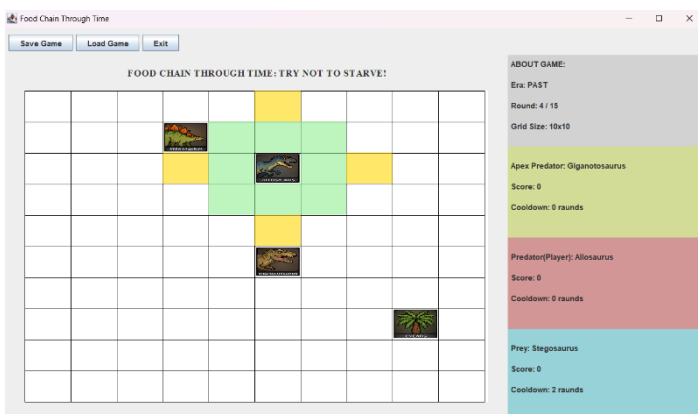


Figure 9: Before Using Special Ability

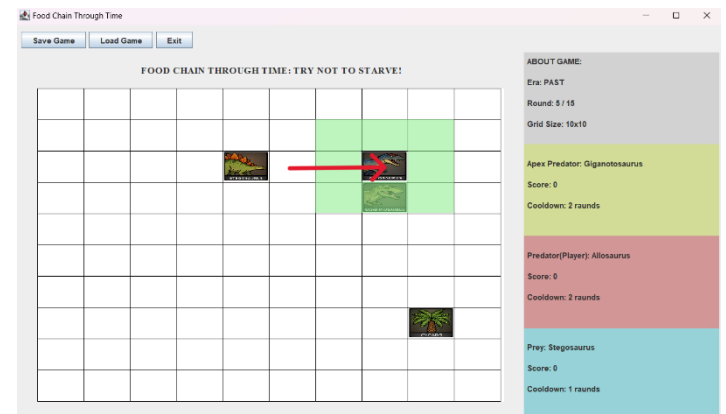
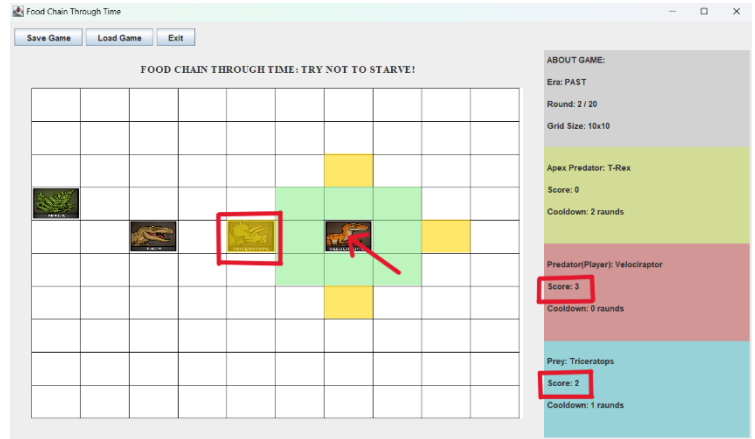
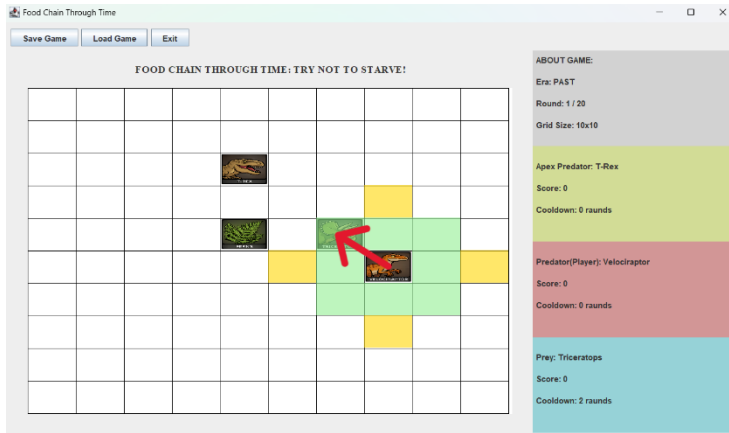


Figure 10: After Using Special Ability

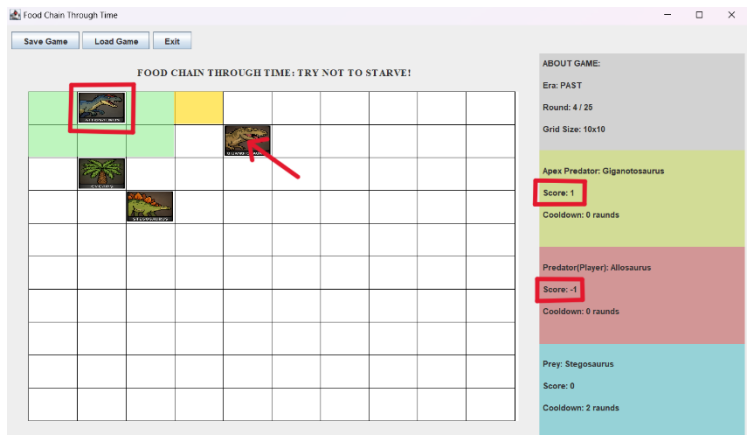
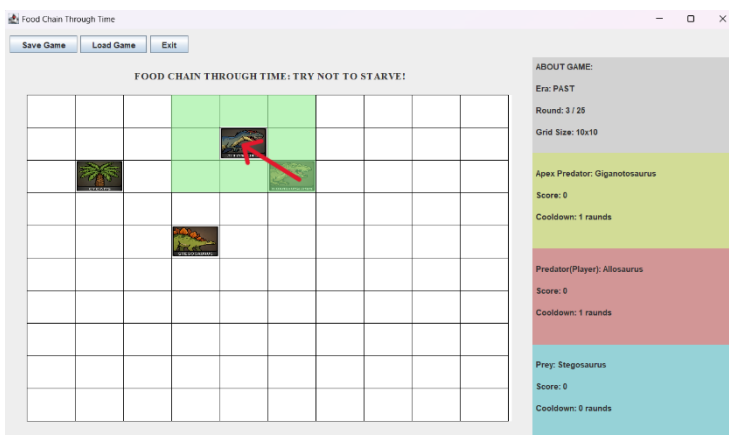
Score and Eating

Player (predator) can eat prey characters. In that case, player gains +3 points and prey loses -1 point. The illustration of this event is below:



As seen in the images above, when predator eats prey character, player gains +3 points and predator loses -1 point. (Prey also eats food at its turn so its score is $3 - 1 = 2$) Also, prey respawns in a random empty cell on the grid.

Also apex predator can eat predator (player). In that case, player loses -1 point and apex predator gains +1 point. Following images demonstrate that event:



As seen in the images above, when apex predator eats predator, player loses -1 point and predator gains +1 point. Also, predator respawns in a random empty cell in grid. The procedure is pretty same when apex predator eats prey character.

Logging

To ensure consistency and assist in debugging, the application records a log file named log.txt that records the entire flow of the game. The GameLogger class handles this operation. It appends every important event happening in the game. Following figure demonstrates log file used in application:

```
3=====
4=== NEW GAME STARTED ===
5The Player is playing predator with name: Wolf
6Mode: PRESENT, Grid: 10x10, Total Rounds: 15
7Food Chain Loaded: Polar Bear -> Wolf -> Deer -> Shrubs
8
9Raund 1
10Deer (Prey) moved to (0, 2)
11Wolf (Player/Predator) moved to (1, 8)
12Polar Bear (Apex) moved to (6, 2)
13Round 1 completed.
14
15-----
16
17Raund 2
18Deer (Prey) ATE FOOD at (0,3)
19Deer (Prey) moved to (0, 3)
20Wolf (Player/Predator) moved to (1, 7)
21Polar Bear (Apex) moved to (5, 3)
22Round 2 completed.
23
24-----
25
26Raund 3
27Deer (Prey) moved to (1, 2)
28Wolf (Player/Predator) moved to (1, 6)
29Polar Bear (Apex) moved to (4, 2)
30Round 3 completed.
31
32-----
33
34Raund 4
35Deer (Prey) moved to (2, 1)
36Wolf (Player/Predator) moved to (1, 5)
37Polar Bear (Apex) moved to (3, 1)
38Round 4 completed.
39
40-----
```

Figure 15: Example Log File

Game Completion

The game is designed to conclude automatically once the predefined number of rounds is reached. Throughout the gameplay, the system continuously checks the round counter after every round. When the final round is processed, game over process executes and game over panel is displayed:

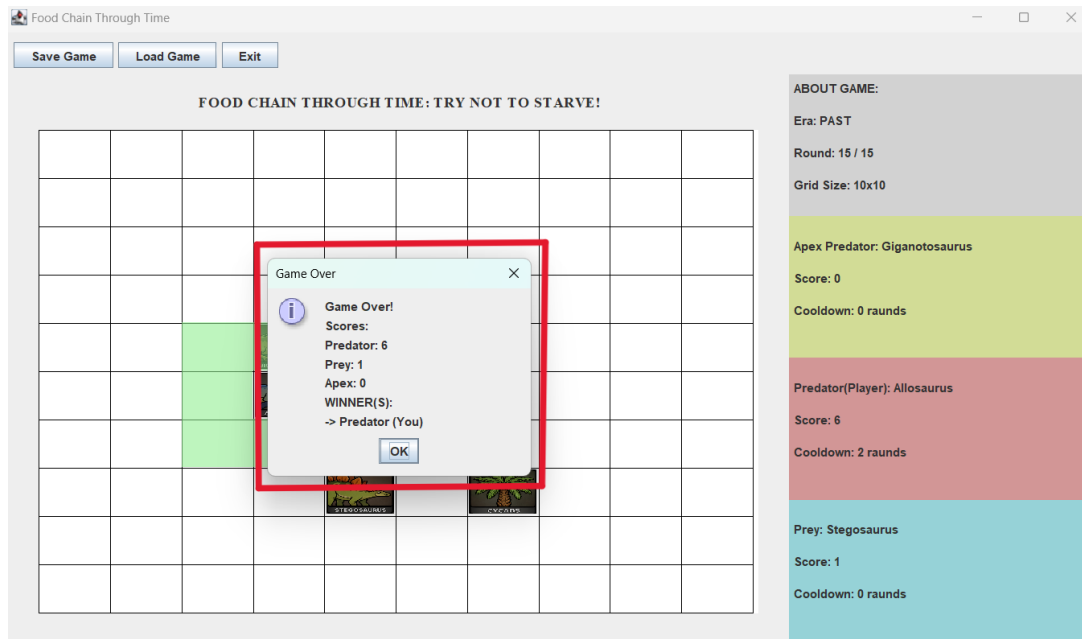


Figure 16: The Game Over screen

User can see the game results at that panel and exit the game using exit button above.

1.4) Saving Game State

The application provides a save feature for users. With this feature, users can save their game progress at any time of the game. It is accessible via the "Save Game" button located on the top panel of the game interface. Upon successful completion of this operation, the user is notified via a confirmation dialog, as shown in the Figure 17.

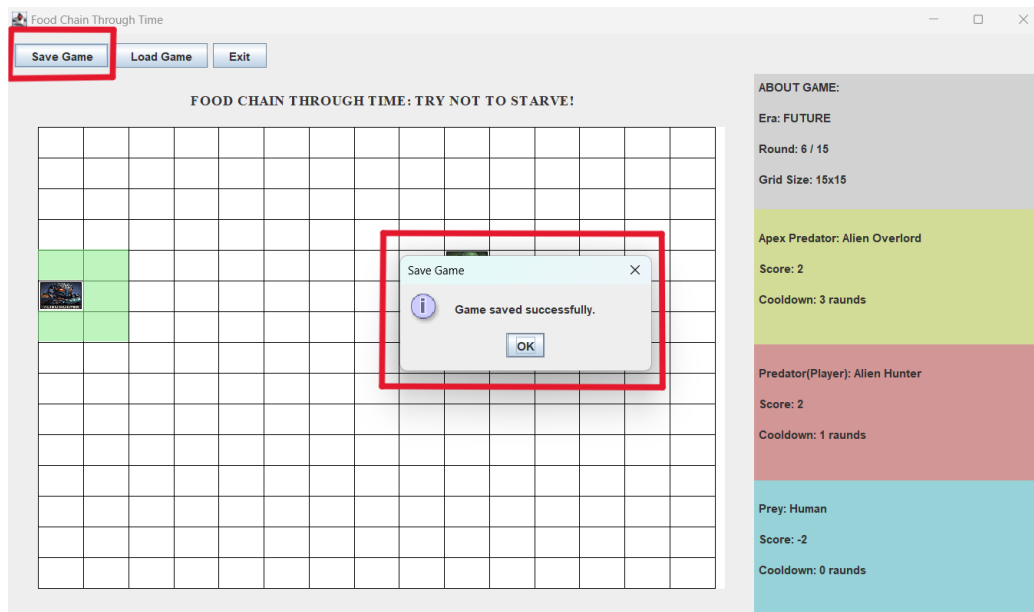


Figure 17: "Save Game" button and notification

After this step is done, the application saves the current game state to the "saved_game.txt" file. This file holds game informations, entities positions, score and cooldown informations, and food's position. The format of this text file is given below in Figure 18.

```
1 MODE:FUTURE
2 ROUND:6/15
3 GRID SIZE:15
4 ENTITY:ApexPredator;Alien Overlord;9;4;2;3
5 ENTITY:Predator;Alien Hunter;0;5;2;1
6 ENTITY:Prey;Human;11;6;-2;0
7 FOOD:Cow;9;7
```

Figure 18: Content of saved_game.txt file

1.5) Load Game

The application allows users to restore a previously saved game using the "Load Game" button. To do this, user needs to start a game first. And then by clicking “Load Game” button, the saved game is going to be uploaded by system. The player can continue playing the saved game. Following figures demonstrate that process.

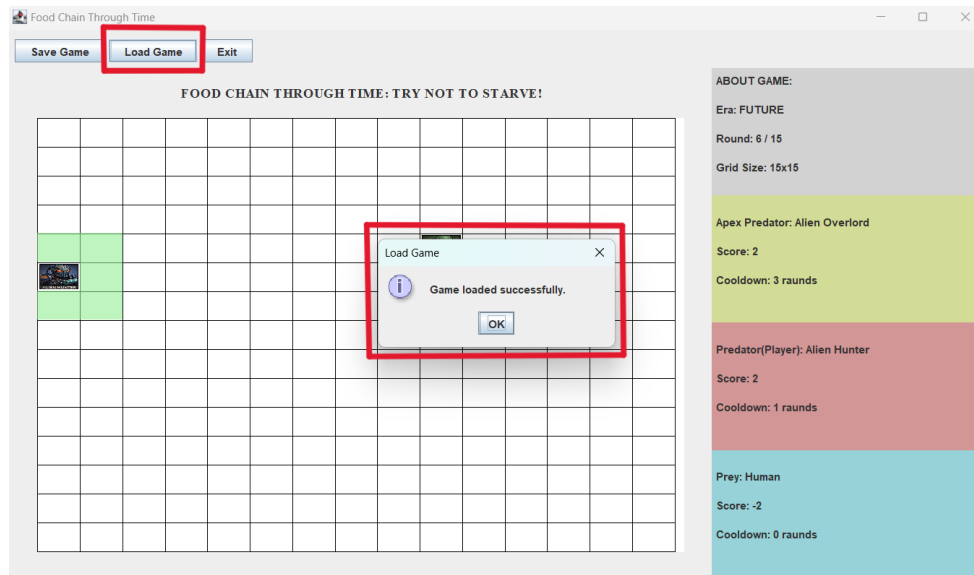


Figure 19: Loading a game saved previously

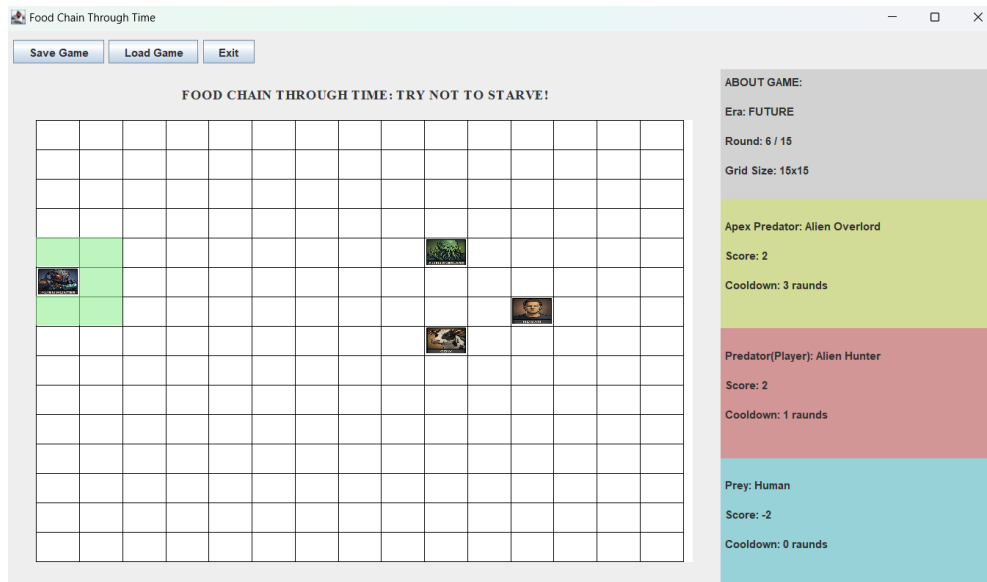


Figure 20: Loaded game is ready to be played

1.6) Different Special Moves and Character Icons

The following figure illustrates a future era game state. The Cyborg character (Predator) has dash special ability available. It can move 2 cells in any direction.

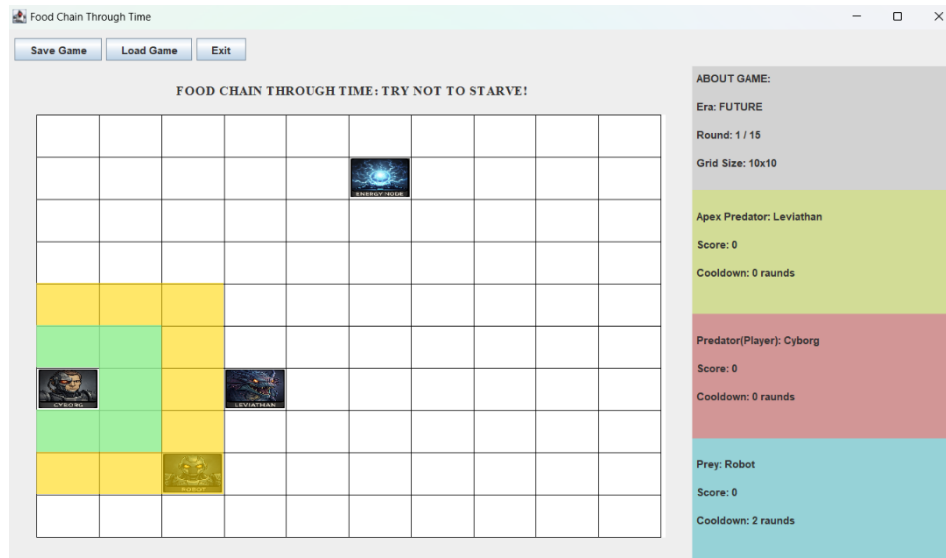


Figure 21: Future era game with predator Cyborg

The following figure illustrates a present era game state. The wolf character (predator) has its dash special ability available, since it is adjacent to polar bear (apex predator). It can move 2 cells in any direction.

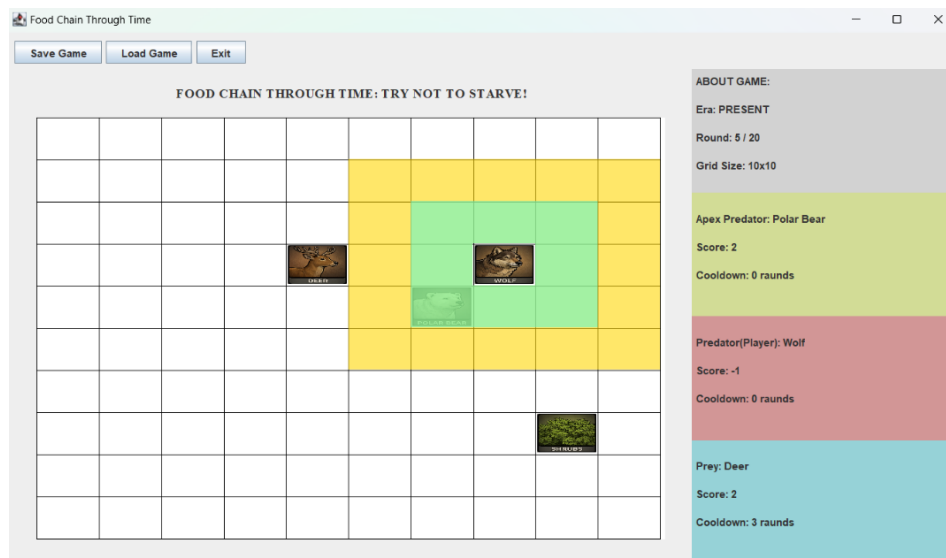


Figure 22: Present era game with predator Wolf

PART 2: Project Design Description

2.1) Class Relations of My Project:

- model package
 - interface IRespawnable
 - abstract class Entity
 - class Food extends Entity implements IRespawnable
 - abstract class Animal extends Entity implements IRespawnable
 - class Prey extends Animal
 - class Predator extends Animal
 - class ApexPredator extends Animal
 - class Grid
- logic package
 - enum GameMode
 - class GameEngine
- util package
 - class FileManager
 - class GameLogger
- gui package
 - class StartScreen extends Jpanel
 - class GamePanel extends Jpanel
 - class MainFrame extends JFrame
- exception package
 - class GameLoadException extends Exception
- main package
 - class Main

2.2) The Implementation Details

model package:

The model package contains core aspects of the game. It holds the attributes and behavior of all characters and objects in the game. It helps to keep character's data/logic separate from game's logic or visual parts.

IRespawnable Interface this interface stands for entities which are able to respawn after being eaten. The Animal class implements this interface, so all animals have this method. Also Food class implements this interface, since food objects also need to be respawnable.

Entity Class is the fundamental abstract base class for all objects that stand on the grid. It instantiates core attributes such as position and name of characters.

Food Class is a concrete class that represents stationary food objects of the game. By extending Entity class, it inherits the coordinate properties required to be placed on the grid. Furthermore, it implements IRespawnable interface, ensuring that food objects are able to respawn when they are eaten.

Animal Class is a common base for all moveable entities in the game. It stores some basic information of characters such as score, isAlive, abilityCooldown and era knowledge of that specific animal object. Era knowledge is crucial for defining special abilities later. It also declares the abstract method makeMove(Grid grid), each subclass (Prey, Predator, ApexPredator) needs to implement this method to define its own movement logic.

Prey Class is a concrete class that represents prey objects, which are at the bottom of the food chain. It is controlled by computer, so it has its own decision making (AI) system. Also this class implements makeMove(Grid grid) method. The following figures demonstrate how AI logic of prey works and implementation of makeMove(Grid grid) method.

```

private int[] pickBestMove(List<int[]> candidates, Entity threat, Entity food, boolean canEat) {
    int[] bestMove = null;
    double bestScore = -Double.MAX_VALUE;

    double panicDistance = 3.0;

    for (int[] move : candidates) {
        double score = 0;
        int targetX = move[0];
        int targetY = move[1];

        if (threat != null) {
            double distToThreat = getDistance(targetX, targetY, threat.getX(), threat.getY());

            if (distToThreat < panicDistance) {
                score += distToThreat * 20.0;
            }
            else {
                score += distToThreat * 0.1;
            }
        }

        if (food != null && canEat) {
            double distToFood = getDistance(targetX, targetY, food.getX(), food.getY());
            score -= distToFood * 6.0;
        }

        if (score > bestScore) {
            bestScore = score;
            bestMove = move;
        }
    }
    return bestMove;
}

```

Figure 23: *pickBestMove(...)* method is main decision making mechanism of prey

The **pickBestMove(...)** method uses a scoring system to evaluate the given candidate moves. It defines a local variable named `panicDistance`. The logic about `panicDistance` is as follows: if prey is three cells or fewer away from a threat, the algorithm prioritizes running away from the threat. Otherwise, the prey prioritizes moving towards food.

```

@Override
public void makeMove(Grid grid) {

    List<Entity> threats = findEntities(grid, Predator.class, ApexPredator.class);
    List<Entity> foods = findEntities(grid, Food.class);

    Entity closestThreat = getClosest(threats);
    Entity closestFood = getClosest(foods);

    if (this.abilityCooldown == 0) {
        boolean usedAbility = tryUseSpecialAbility(grid, closestThreat, closestFood);
        if (usedAbility) {
            setCooldownBasedOnEra();
            return;
        }
    }

    makeStandardMove(grid, closestThreat, closestFood);
}

```

Figure 24: *makeMove(...)* method is main movement method of prey

The **makeMove(...)** method first detects `closestThreat` and `closestFood` entities by using helper methods. The movement logic works in a simple way for prey characters: if special ability available, prey uses it to move a better position. Most of the time this logic is convenient for a greedy mechanism. If special ability is not available, prey does standard move. Both `tryUseSpecialAbility(...)` and `makeStandardMove(...)` methods uses helper method `pickBestMove(...)` method to pick best possible option.

Predator Class is a concrete class that represents predator objects on the grid. Predator character is controlled by player so its implementation differs from prey or apex predator classes. The main movement method for Predator is **performMove(Grid grid, int targetX, int targetY)**. This method takes target position on the grid and executes move of player. But the main concept here is taking user input by clicking. Here is the piece of code from GamePanel class that handles mouse clicks on the game grid:

```
@Override
public void mousePressed(MouseEvent e) {
    int mouseX = e.getX();
    int mouseY = e.getY();

    int rows = engine.getGrid().getRows();
    int cols = engine.getGrid().getCols();

    int cellWidth = getWidth() / cols;
    int cellHeight = getHeight() / rows;

    int clickedCol = mouseX / cellWidth;
    int clickedRow = mouseY / cellHeight;

    boolean turnSuccessfull = engine.processTurnWhenClicked(clickedCol, clickedRow);

    if (turnSuccessfull) {
        System.out.println(clickedCol + ". col and " + clickedRow + ". row is clicked.");
        repaint();

        Window w = SwingUtilities.getWindowAncestor(GamePanel.this);
        if (w instanceof MainFrame) {
            ((MainFrame) w).updateLabels();
        }

        if (engine.isGameOver()) {
            String result = engine.getGameResult();

            // Game over penceresini çıkar
            JOptionPane.showMessageDialog(GamePanel.this, result, "Game Over", JOptionPane.INFORMATION_MESSAGE);
        }
    }
}
```

Figure 25: Handling Mouse Clicks

Once the mouse click input is processed, the method invokes **processTurnWhenClicked(clickedCol, clickedRow)** from GameEngine class. Inside of this method, **performMove(...)** method is called with the target position information. And predator's movement is successfully executed and can be seen simultaneously on the grid thanks to **repaint()** method.

ApexPredator Class is a concrete class that defines apex predator character's attributes and movement logic. It is controlled by computer and has its own decision making system. It uses **pickBestMove(...)** method to move best possible grid cell. Its main movement logic is implemented in **makeMove(...)** method. The contents of these methods can be followed in figures below.

```

private int[] pickBestMove(List<int[]> candidates, Entity target) {
    int[] bestMove = null;
    double minDistance = Double.MAX_VALUE;

    for (int[] move : candidates) {
        double dist = 0;

        if (target != null) {
            dist = getDistance(move[0], move[1], target.getX(), target.getY());
        }

        if (dist < minDistance) {
            minDistance = dist;
            bestMove = move;
        }
    }
    return bestMove;
}

```

Figure 26: pickBestMove(...) method is main decision making mechanism of apex predator

This method first detects its nearest prey (prey or predator). And returns the move that makes it closest to that prey. The decision making algorithm is pretty basic for apex predator character.

```

@Override
public void makeMove(Grid grid) {
    List<Entity> targets = findTargets(grid);
    Entity closestTarget = getClosest(targets);

    boolean abilityUsed = false;
    if (abilityCooldown == 0 && closestTarget != null) {
        abilityUsed = tryUseSpecialAbility(grid, closestTarget);
    }

    if (!abilityUsed) {
        if (abilityCooldown > 0) {
            abilityCooldown--;
        }
        makeStandardMove(grid, closestTarget);
    }
}

```

Figure 27: makeMove(...) is main movement method for apex predator character

This method is main movement method for apex predator character. First, it detects closest target (prey or predator). Then, it tries to use special ability if available. Otherwise, it executes a standard move. Most of the time this logic is convenient for a greedy mechanism. Both tryUseSpecialAbility(...) and makeStandardMove(...) methods uses helper method pickBestMove(...) method to move closest cell to the target entity.

Grid Class represents 2D environment of the game. It contains a two dimensional array to keep track of all entities on the grid. Its main role is to manage positions and provide helper methods such as `isValidPosition(int x, int y)` to check boundaries and `putEntity(Entity e, int x, int y)` to place or move entities on the grid.

logic package

GameMode enum defines the three different time periods in the game: Past, Present and Future. It is used inside the logic package to tell the system which era is currently active. It ensures that the game mode is always set to one of these three specific options.

Game Engine Class acts as the brain of the game execution. Its main job is to manage the flow of the game which includes starting a new match, placing animals in random spots and keeping track of the current round. It also handles the turn order ensuring that the prey moves first, followed by the player (predator) and finally the apex predator. Additionally, it constantly checks if the game is over after every round.

The flowchart on the next page shows the step by step logic of the gameflow. It illustrates the order of moves for the prey, player and apex predator.

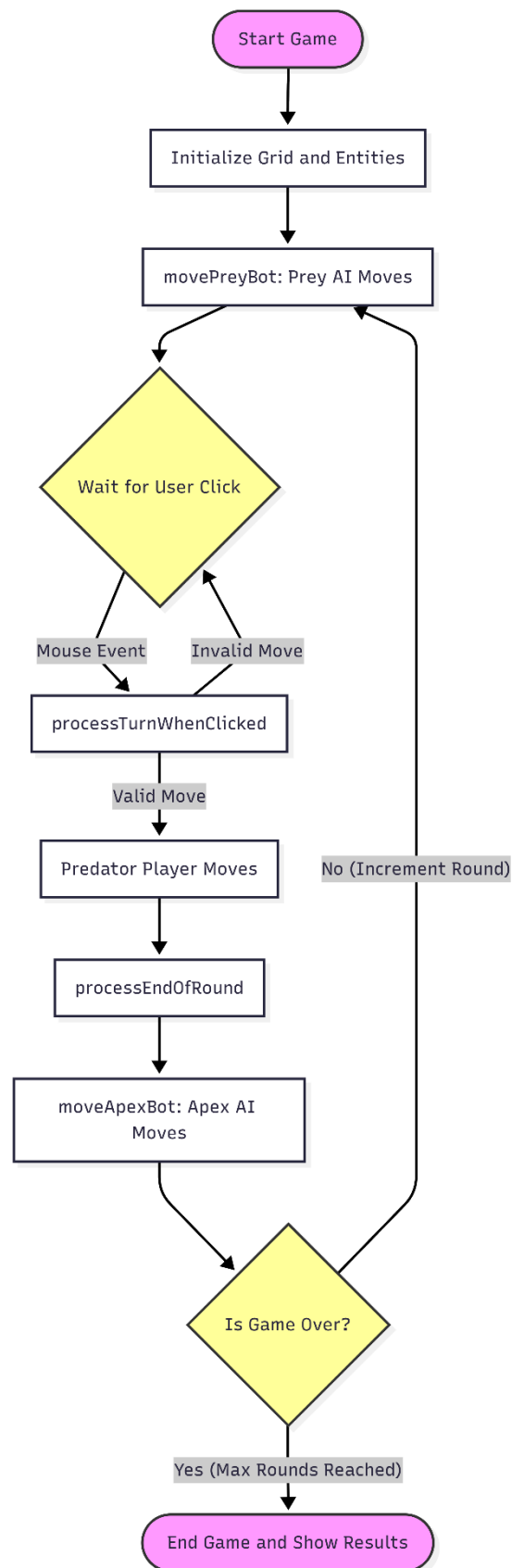


Figure 28: The Gameflow in the Game Engine

2.3) GUI Components

gui package

StartScreen Class extends JPanel and displays the game's start menu. It uses different swing components to create a user friendly interface. A JComboBox is used to select the game era and grid size. Also a JTextField is used to enter the number of rounds. All game settings are collected on a single screen.

This class also controls how the game starts. When the player clicks the "Start Game" button, it checks whether the inputs are valid or not. It makes sure that options are selected and the number of rounds is at least 10. If everything is correct, it calls the `startGamePlayScreen()` method in MainFrame to change the screen and start the game.

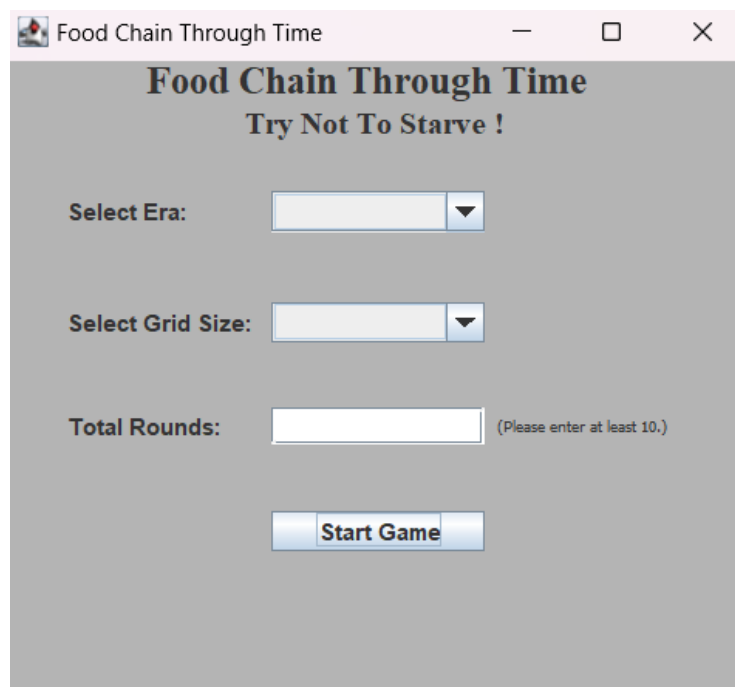


Figure 29: Start Screen

GamePanel Class is the main visualizer of the game grid. It is responsible for drawing the game on the screen. It extends JPanel and overrides the `paintComponent` method to draw the game grid. The size of each cell is calculated using the current window size. So the grid automatically resizes when the window changes. While drawing the grid, it goes through the grid object and displays the correct images for animals and food in each cell. This class also manages user interaction. It uses a `MouseListener` to detect mouse clicks on the panel. The pixel position of a click is converted to row and column values, which are then sent to the `GameEngine` to handle the move.

To help the player, the panel highlights valid moves for the player (predator). Standard walk moves are highlighted in green and special ability moves are shown in yellow. This makes it easier for the player to understand possible moves. Here is the method which is responsible for this operation:

```
private void paintAvailableCells(Graphics g, int cellWidth, int cellHeight) {
    if (engine != null) {
        Predator player = engine.getPredatorPlayer();
        if (player == null) {
            return;
        }

        List<int[]> moves = player.getAvailableMoves(engine.getGrid());

        int mevcutX = player.getX();
        int mevcutY = player.getY();

        for (int[] m : moves) {
            int possibleX = m[0];
            int possibleY = m[1];

            double a = Math.pow(possibleX - mevcutX, 2);
            double b = Math.pow(possibleY - mevcutY, 2);

            double oyuncuyaMesafesi = Math.sqrt(a + b);

            if (oyuncuyaMesafesi > Math.sqrt(2)) {
                g.setColor(new Color(255, 215, 0, 150));
            }
            else {
                g.setColor(new Color(144, 238, 144, 150));
            }

            g.fillRect(possibleX * cellWidth, possibleY * cellHeight, cellWidth, cellHeight);
        }
    }
}
```

Figure 30: paintAvailableCells(...) method to paint possible moves for player

MainFrame Class is the main window of the game. It holds all parts of the user interface. It uses a CardLayout to switch between the start screen and the main game screen. This makes it easy to change screens without opening new windows. When the game starts, the class sets up the main game layout. The GamePanel is placed in the center of the window and a side panel shows game information like the current round, animal names, and scores etc. Additionally it contains a top menu that includes “Save Game”, “Load Game”, and “Exit” buttons. The class also has a helper method called updateLabels(). This method is called after each turn to update the informations on the screen. So the player always sees the latest game status.

2.4) File Processing Details

util package

GameLogger Class is responsible for logging every important event happening in the game. It has a static method that writes text messages to a file called log.txt. The class uses a BufferedWriter in append mode. So new messages are added to the end of the file. This allows the program to keep a record of what happens during the game and helps tracking the game flow.

```
public static void log(String message) {  
    try (BufferedWriter writer = new BufferedWriter(new FileWriter(LOG_FILE, true))) {  
        writer.write(message);  
        writer.newLine();  
    }  
    catch (IOException e) {  
        System.out.println("Log'da bi hata var.");  
    }  
}
```

Figure 31: The log(...) method of GameLogger class

FileManager Class handles all the file reading and writing tasks for the game. Its first important job is to read the game characters. The `loadRandomFoodChain(...)` method looks at specific text files (like `past_animals.txt` or `future_animals.txt`) based on the selected era. It reads these files and randomly picks a food chain line to decide which specific animal characters will appear in the game.

This class is also responsible for saving and loading the player's progress. The `saveGame(...)` method writes all the current game state such as grid size, current round number, exact position and score of every animal into a file named `saved_game.txt`. Later, the `loadGame(...)` method allows the user to continue playing. It reads that file line by line, recreates the grid and puts every animal back in its exact spot with its correct score and cooldwon information.

2.5) Exception Handling Details

In my project, I have a custom exception named **GameLoadException**. It is designed specifically to handle problems that happen when the game tries to load files, such as reading animal data or opening a saved game. By creating this specific exception, the program can easily separate these errors from other types of errors and show a clear message to the user about what went wrong.

References:

- <https://www.youtube.com/watch?v=U28eKSLI7pw>
- <https://www.w3schools.com/java/>
- <https://youtu.be/Kmgo00avvEw>
- <https://docs.oracle.com/en/java/>
- <https://www.baeldung.com/a-guide-to-java-enums>
- [https://www.youtube.com/playlist?list=PLJhTWoCm8I6AAW3EVpZRS
Tvvc-jOI0rfQ](https://www.youtube.com/playlist?list=PLJhTWoCm8I6AAW3EVpZRS
Tvvc-jOI0rfQ)
- <https://www.geeksforgeeks.org/java/introduction-to-java-swing/>