Sprawozdanie

Aplikacja okienkowa Java - Snake

Wykonawcy: *Tomasz Nowak 254021*

1. Opis Zadania

Celem zadania było napisanie aplikacji okienkowej w technologii Java.

Wymagania:

- Aplikacja napisana w języku programowania Java.
- Aplikacja okienkowa z interfejsem graficznym.
- Użycie biblioteki Swing (lub podobnej) do wyświetlania elementów graficznych.
- Aplikacja powinna zapewnić bazową oprawę graficzną typu „kolorowe piksele". Obiekty mogą być przedstawione w postaci prostych kwadratów, linii i kropek. *(**Notka:** za rozbudowaną szatę graficzną można otrzymać dodatkowe punkty, ale nie jest ona wymagana do pełnej oceny.)*
- Wykorzystanie klas i interfejsów.
- Wykorzystanie kolekcji w implementacji własnego algorytmu przetwarzania danych.
- Aplikacja wielowątkowa.
- Wątki powinny komunikować się/wchodzić w interakcje między sobą.
- Sterowanie wybranymi elementami aplikacji za pomocą sztucznej inteligencji (AI).
    - Inteligencja powinna być na takim poziomie by umożliwić podstawową rozgrywkę. Np. wąż powinien zbierać najbliższe owoce i unikać najbliższych przeszkód (nie jest konieczne planowanie najlepszej ścieżki dla wszystkich obiektów na planszy).
- Obsługa zapytań z klawiatury i myszy, np.:
    - Sterowanie elementami na planszy przy pomocy klawiatury.
    - Obsługa menu/okienka za pomocą myszy.
- Zapis i odczyt danych z plików (np. w postaci pliku tablicy najlepszych wyników).
- Wygenerowanie pliku z dokumentacją na podstawie komentarzy w kodzie.
- Udokumentowanie aplikacji przy pomocy diagramu UML. Diagram powinien przedstawiać przynajmniej kluczowe elementy aplikacji jak:
    - Uruchomienie aplikacji – metoda *main*,
    - Wykonanie głównej pętli programu,
    - Rysowanie elementów graficznych,
    - Aktualizacja stanu gry,
    - Odczyt danych wejścia/wyjścia (np. z klawiatury lub myszy).
- Dopuszczalne diagramy UML:
    - diagram klas (*Class Diagram*)
    - diagram aktywności (*Activity Diagram*)
    - diagram stanu (*State Machine Diagram*)

Wybrane technologie

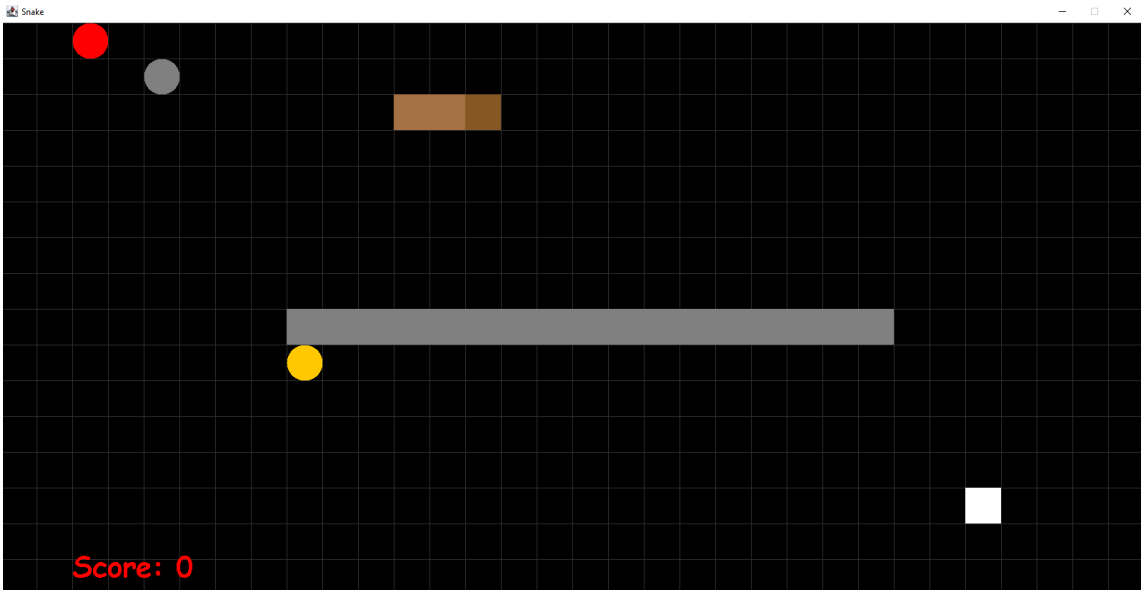
- Edytor kodu Eclipse

- System kontroli wersji Git (GitLab lub GitHub)
- Dokumentacja wygenerowana przy pomocy Javadoc
- Diagram UML (Unified Modeling Language – zunifikowany język modelowania) https://www.draw.io/

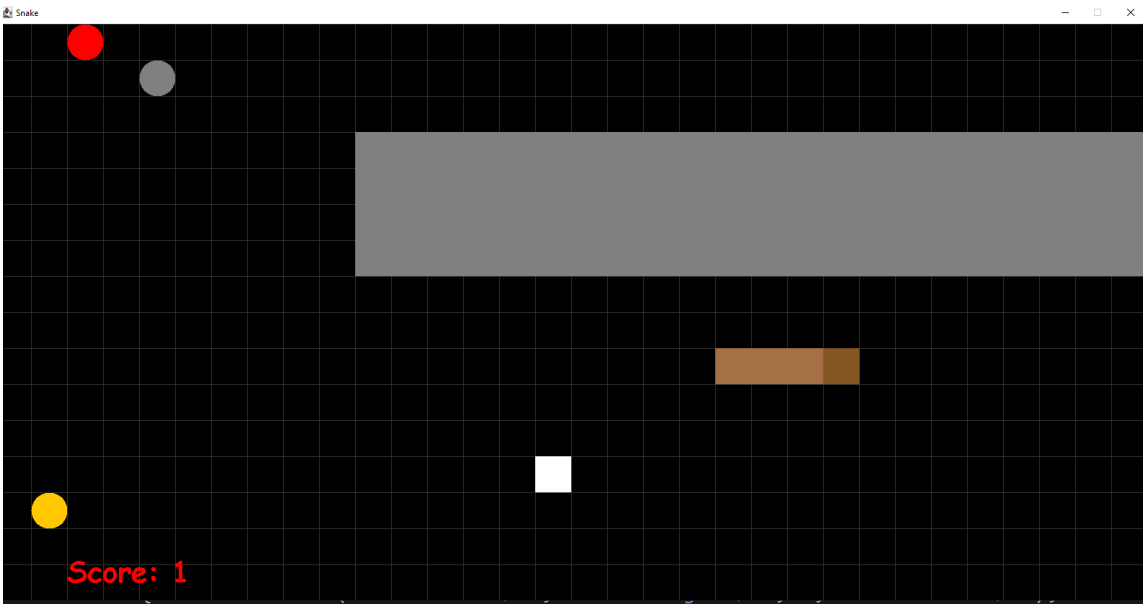
2. Program

2.1. Działanie programu

Po włączeniu programu generuje się mapa, razem z przeszkodami, owocami oraz zającem który ucieka od gracza.



Rysunek 1 Przykładowa mapa 1



Rysunek 2 Przykładowa mapa 2

Elementy gry to:

- Gracz – brązowy wąż
- Zając – biały kwadrat
- Owoce – różnego koloru kółka
- Przeszkody/a – szare kwadraty

„Score” przedstawia ilość punktów które gracz posiada oraz dodatkową długość węża. Zając daje dwa punkty i nie pojawia się ponownie, pomarańczowy i czerwony owoc dają po jednym punkcie, a szary owoc odejmuje jeden punkt. Wyjście poza mapę, zderzenie się z przeszkodą oraz uzyskanie punktów poniżej zera skutkuje zakończenia tej sesji obrazem „Game Over”

Rysunek 3 Ekran Game Over

Z tego ekranu można albo wyjść z gry za pomocą „ESC”, lub zagrać ponownie za pomocą przycisku „Enter”. Ekran ten wyświetla też ilość uzyskanych punktów oraz najwyższy

uzyskany wynik przez gracza ze wszystkich gier, nie tylko tych podczas tego uruchomienia programu.

W każdym momencie gry (poza ekranem Game Over) można zatrzymać grę wciskając przycisk „ESC”. Można ją następnie wznowić przyciskiem „Enter”.

Rysunek 4 Ekran pauzy

2.2. Implementacja w kodzie

2.2.1. Funkcja main:

Jedynie odpala grę, wszystko się dzieje głębiej.

```
public class SnakeGameMain {

    /**
     * 'main' only initializes the GameFrame
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        GameFrame Game = new GameFrame();
        //new GameFrame();
    }

}
```

2.2.2. Klasa GameFrame:

Odpowiada za ustawienie okienka oraz włączenie gry za pomocą klasy GamePanel.
Klasa ta również zamyka okienko po zakończeniu gry.

```
public class GameFrame extends JFrame
{
    GameFrame()
    {
        this.setTitle("Snake");
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setResizable(false);
        GamePanel Game = new GamePanel();;
        this.add(Game);
        this.pack();
        this.setVisible(true);
        this.setLocationRelativeTo(null);
        Game.startGame();
        this.setVisible(false);
        this.dispose();
    }

}
```

2.2.3. Klasa GamePanel:

Konstruktor tej klasy ustawia tło mapy na czarne, rozmiar okienka, oraz włącza wejście klawiatury użytkownika (KeyListener).

```
GamePanel() {
    random = new Random();
    this.setPreferredSize(new Dimension(ScreenWidth, ScreenHeight));
    this.setBackground(Color.black);
    this.setFocusable(true);
    this.addKeyListener(new UserInputAdapter());
}
```

Główny LOOP gry odbywa się w funkcji startGame.
Pierwsza część odpowiada za ustawienie obiektów w grze oraz włączenie wątków.

```
public void startGame() {
    // Creating objects
    PlayerSnake = new Snake(3, 1000, direction, ElementSize);
    FGenerator = new FruitGenerator(3);
    rabbit = new Rabbit(PlayerSnake, ElementSize);

    rabbit.SpawnRabbit();

    // Creating Threads
    PlayerThread = new Thread(PlayerSnake, "PlayerThread");
    RabbitThread = new Thread(rabbit, "RabbitThread");
    FruitGeneratorThread = new Thread(FGenerator, "FruitThread");

    //Generating Obstacle
    Obstacle = Obstacles[0];
    if (random.nextInt() > 50)
        Obstacle = Obstacles[1];

    // starting threads
    PlayerThread.start();
    FruitGeneratorThread.start();
    RabbitThread.start();
```

Druga część odpowiada za LOOP gry. Paint Component rysuje wszystko, CheckCollisions sprawdza kolizje wszystkich obiektów. Jest też oczywiście krótki sleep.

```
running = true;
//Runs until the player wants to quit
while (!QuitGame) {

    //This part is for when the game resets
    //It resets all the values to the default ones
    //and restarts everything that is necessary
    if (ResetGame) {
        ResetGame();
    }
    //main loop
    //checks collisions and draws the graphics

    if (running) {
        paintComponent(getGraphics());
        CheckCollisions();
    }
    repaint();
    try {
        TimeUnit.MILLISECONDS.sleep(GamePanel.Delay);
    } catch (Exception e) {
        System.out.println("Main: " + e.toString());
    }
}
QuitGame = true;
```

Reset gry odbywa się za pomocą funkcji ResetGame, która najpierw ustawia potrzebne wartości na domyślne oraz kończy wątki. Następnie wykonuje to samo co w pierwszej części startGame.

```
private void ResetGame()
{
    System.out.println("Game Reseting\n");
    PlayerSnake.ThreadActive = false;
    rabbit.ThreadActive = false;
    FruitsEaten = 0;
    ResetGame = false;
    running = true;
    SaveHighScore = true;
    direction[0] = 'R';
    // FGenerator.ThreadActive = false;
    try {
        PlayerThread.join();
        RabbitThread.join();
        // FruitGeneratorThread.join();
    } catch (Exception e) {
        System.out.println(e.toString());
    }
```

Funkcja paintComponent istnieje głównie z powodu istnienia super. paintComponent(g) rysowanie elementów gry odbywa się w funkcji draw.

```
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    if (!IsPaused)
        draw(g);
    else if (IsPaused) {
        DrawPauseScreen(g);
    }
}
```

W funkcji draw odbywa się rysowanie głównych elementów, czyli gracza, zająca, owoców oraz przeszkód.

```
// draws lines (the grid)
for (int i = 0; i < ScreenWidth / ElementSize; i++) {
    g.drawLine(i * ElementSize, 0, i * ElementSize, ScreenHeight);
    g.drawLine(0, i * ElementSize, ScreenWidth, i * ElementSize);
}

// Draws The Rabbit
if (rabbit.isAlive)
    g.setColor(Color.white);
else
    g.setColor(Color.black);
g.fillRect(rabbit.PosX, rabbit.PosY, ElementSize, ElementSize);

// Draws the Fruits with proper colors
g.setColor(Color.red);
Color color;
for (int i = 0; i < FGenerator.FruitAmount; i++) {
    try {
        Field field = Class.forName("java.awt.Color").getField(FGenerator.Colors[i]);
        color = (Color) field.get(null);
    } catch (Exception e) {
        color = Color.red; // Not defined
    }
    g.setColor(color);
    g.fillOval(FGenerator.PosX[i], FGenerator.PosY[i], ElementSize, ElementSize);
}
```

```
// Draws the Obstacle
g.setColor(Color.gray);
for (int i = Obstacle.PosX[0]; i <= Obstacle.PosX[1]; i += ElementSize) {
    for (int j = Obstacle.PosY[0]; j <= Obstacle.PosY[1]; j += ElementSize) {
        g.fillRect(i, j, ElementSize, ElementSize);
    }
}

// Draws the Snake
DrawingPlayer = true;
for (int i = 0; i < PlayerSnake.BodyLength; i++) {
    if (i == 0) {
        g.setColor(new Color(133, 87, 35));
        g.fillRect(PlayerSnake.GetX(i), PlayerSnake.GetY(i), ElementSize, ElementSize);
    } else {
        g.setColor(new Color(165, 113, 68));
        g.fillRect(PlayerSnake.GetX(i), PlayerSnake.GetY(i), ElementSize, ElementSize);
    }
}
DrawingPlayer = false;

//Draws the Score
g.setColor(Color.red);
g.setFont(new Font("Comic Sans MS", Font.BOLD, 40));
g.drawString("Score: " + FruitsEaten, 100, ScreenHeight - 25);
```

Wszystko to dzieje się tak długo, jak gra nie jest w stanie GameOver, ponieważ wtedy dzieje się to:

```
try {
    //The Game over screen
    GameOver(g);
} catch (Exception e) {
    System.out.println("GameOver:" + e.toString());
}
```

Oprócz generowania ekranu zakończenia gry funkcja GameOver również zapisuje/odczytuje najlepszy wynik

```
public void GameOver(Graphics g) throws IOException {
    // Score
    if (SaveHighScore) {
        String temp;
        Scanner scan = new Scanner(file);
        // Load it and shit
        while (scan.hasNextLine()) {
            temp = scan.nextLine();
            HighScore = Integer.parseInt(temp);
        }
        if (FruitsEaten > HighScore) {
            HighScore = FruitsEaten;
            FileWriter writer = new FileWriter("HighScore.txt");
            temp = Integer.toString(HighScore);
            writer.write(temp);
            writer.close();

        }

        SaveHighScore = false;
    }
```

Wejście użytkownika jest zaimplementowane za pomocą KeyListener, który potrzebuje KeyAdapter. Po wciśnięciu przycisku, funkcja keyPressed ustawia odpowiednią

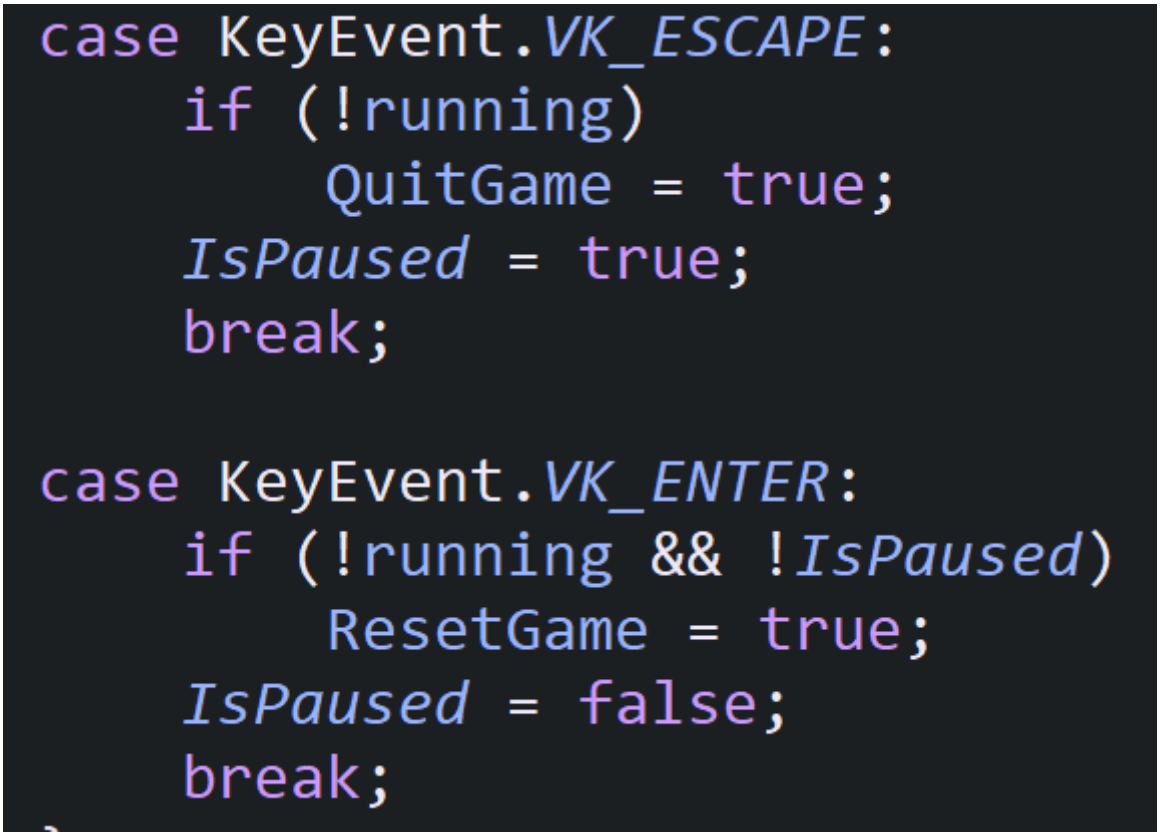
flagę/kierunek.

```
public class UserInputAdapter extends KeyAdapter {

    /**
     *  Function responsible for user input
     *  runs on a separate thread, changes direction[0
     *  also handles the input for stopping, resuming
     */
    @Override
    public void keyPressed(KeyEvent e) {
        switch (e.getKeyCode()) {
        // Left
        case KeyEvent.VK_A:
            if (direction[0] != 'R') {
                direction[0] = 'L';
            }
            break;
        case KeyEvent.VK_LEFT:
            if (direction[0] != 'R') {
                direction[0] = 'L';
            }
            break;

        // Right
        case KeyEvent.VK_D:
            if (direction[0] != 'L') {
                direction[0] = 'R';
            }
            break;
```

Rysunek 5 Przykładowe kierunki



Rysunek 6 Flagi pauzy/zakończenia gry

3. Podsumowanie

Podczas pisania kodu dużo problemów sprawiało brak referencji jak w języku C++. Żeby przekazać referencje do kierunku, trzeba było stworzyć z tego tablicę, ponieważ Java nie robi referencji do typów prymitywnych.

Trochę problemów było też stworzenie kolizji z przeszkodami z powodu wielowątkowości programu. Zdarzało się tak, że funkcja odpowiadająca za kolizje włączyła się po tym, jak gracz zdążył już przeniknąć przez przeszkodę. Rozwiązaniem tego okazało się nie intuitywnie lepsze dopasowanie przeszkód to planszy, czyli jeśli plansza składa się z kwadratów po 30 pikseli, to przeszkody muszą być idealnie dopasowane do tych elementów.

Tworzenie dokumentacji za pomocą Eclipse i Javadoc jest bardzo łatwe, ponieważ wystarczą dwa kliknięcia.

KeyListener oraz KeyAdapter ztrywializowały obsługę wejść użytkownika.

Po nauczeniu się tego, jak działa biblioteka swing, generowanie elementów w okienku okazało się bardzo proste.