WWW.IJRPR.COM

International Journal of Research Publication and Reviews

Journal homepage: www.ijrpr.com ISSN 2582-7421

Integrating AI in Classic Game Design: A Java-Based Snake Game Approach

Arpita Vasudev[1], Vishwajeet Narake[2], Yogesh S[3], Vidyasagar[4], Yanamala Saiharindranath Reddy[5], V Santosh Varma[6], Vedaanth A joshi[7], Shahid UL Islam [8], Vignesh M[9], Vikas[10], Vivek M[11], Raghvendra Chandrashekar Vaddar[12], VijayKumar M[13]

[1] Assistant Professor, Department of Computer Science and Engineering, Dayananda Sagar Academy of Technology and Management
[2,3,4,5,6,7,8,9,10,11,12,13] Students, Department of CSE,Dayananda Sagar Academy of Technology and Management

ABSTRACT

Video game development is the process of creating a video game. The effort is undertaken by a game developer, who may range from a single person to an international team dispersed across the globe. A video game can be developed in many different languages, one of them being Java. Java is one of the best languages in which one can develop a game. Coding in Java allows one to make games that will run on all the desktop operating systems like Windows, OSX and Linux but also it is the native language for making games for all Android devices. A game can be developed in Java by using IDE's. Eclipse is an Integrated Development Environment for Java. We can use Eclipse for coding and executing the games. The Snake Game is one of the games that can be implemented in Java. Here the snake is moved around by the player, for it to reach the apple. It takes us one step forward to learning more about the computer languages.

Keywords: Snakegame, java, eclipse, oop, game development, game loop, rendering, input handling, scoring, collision detection, interface, class design, logic, graphics, grid

Introduction

Video games have evolved from simple, recreational tools into complex systems integrating artificial intelligence (AI), physics, and human-computer interaction. The **Snake game**, introduced in the 1970s and popularized by Nokia in the 1990s, remains a staple in programming education due to its simplicity and potential for complexity. Traditionally, such games are developed using imperative programming languages such as Java, which allows for structured and object-oriented development.

Java, developed by James Gosling at Sun Microsystems in 1995, is widely known for its platform independence, robustness, and simplicity. It is one of the most common languages for teaching game development due to its wide use in both academic and industrial settings. Eclipse, a popular Java IDE, is frequently used for building such applications. Snake games implemented in Java generally involve logic for rendering a board, moving a player-controlled snake, tracking score, and handling collisions.

In contrast, the modern era of game development is increasingly embracing **AI-driven gameplay**, where player decisions are mimicked by intelligent agents. With the integration of AI models such as **genetic algorithms (GAs)** and **neural networks (NNs)**, Snake becomes an ideal sandbox for training autonomous agents capable of learning and adapting strategies over time. This paper explores both the traditional and AI-enhanced approaches for building the Snake game, offering insight into each methodology's strengths and challenges.

Problem Definition

The classic Snake game relies heavily on user input and predetermined logic paths. Such a system cannot dynamically adapt to new environments or learn optimal strategies over time. This limitation motivates the need for an **intelligent Snake agent** that can **learn autonomously**, improve through gameplay iterations, and respond efficiently to game state variations without manual programming of every rule or condition.

Specifically, the problems addressed in this paper include:

- **Static behavior** in traditional implementations
- Lack of **adaptive learning mechanisms**
- Difficulty in crafting optimized decision rules manually

- Need for scalable systems capable of **autonomous improvement**

Objective of the Paper

This paper aims to:

1. Develop a **classic Snake game in Java** using object-oriented programming.
2. Implement an **AI-driven Snake game agent** using a neural network trained with a genetic algorithm.
3. Compare both approaches based on ease of implementation, performance, adaptability, and scalability.
4. Evaluate the impact of various parameters such as population size, mutation rate, and neural architecture on AI performance.
5. Identify key **implementation challenges** and suggest future enhancements for intelligent game agents

Key Challenges in Developing an Optimal or Better snake game model

1. **Environment Representation:**

   Translating a visual grid into meaningful numerical input for a neural network is non-trivial. The snake must sense walls, its body, and apple position relative to its head using mathematical representations such as binary indicators and angles.

2. **Fitness Function Design:**

   Quantifying success in the game requires a well-crafted fitness function. Poorly designed fitness functions can lead to overfitting or ineffective learning, where the snake learns non-generalizable behavior.

3. **Balancing Exploration vs Exploitation:**

   The snake must explore new strategies (via mutation) but also exploit known good ones (via crossover). Tuning mutation rates is critical to avoid premature convergence or endless randomness.

4. **Training Time and Computational Cost:**

   With large populations (e.g., 2000 snakes), training can be resource-intensive and time-consuming, especially on limited hardware.

5. **Algorithm Selection and Parameter Sensitivity:**

   Different selection (roulette, tournament) and crossover (uniform, arithmetic) methods impact the learning curve and diversity of the population.

6. **Game State Generalization:**

   Ensuring the snake doesn’t "memorize" the path but can generalize to new starting positions or apple locations is a major challenge in reinforcement and evolutionary learning.

Overview of existing work:

The traditional version of the Snake game was developed using the Java programming language within the Eclipse Integrated Development Environment (IDE). The project is structured around an object-oriented approach, making use of a package containing six interrelated classes: BoardPanel, SidePanel, Direction, Clock, TileType, and SnakeGame. Each of these classes is designed to handle a specific aspect of the game's functionality, allowing for modular development and easier debugging or future enhancements.

The BoardPanel class is tasked with managing the visual representation of the game area. Upon launching the game, it sets the background to white and overlays a grid on the panel. This grid helps players judge distances and better navigate the snake toward the apple. The class is also responsible for rendering the snake and apple on the screen. The snake appears in green, with its head visually distinct from the body using eye-like indicators that face the current direction of movement—vertical eyes for north/south and horizontal ones for east/west. The apple, on the other hand, is displayed in red and spawns randomly on the grid. Additionally, this class handles the drawing of textual messages that appear at various stages of the game, such as the initial splash screen, the "Game Over" message, and pause/resume notifications, all of which are centrally positioned for visibility.

On the right-hand side of the game window is the SidePanel, which provides the player with important game statistics and control instructions. This panel also has a white background, with black font to ensure clarity. The layout and styling of the headings and text are managed using the setFont() method. The side panel displays real-time updates on the "Total Score", "Total Apples Eaten", and "Apple Score", giving the player immediate feedback on their performance. In addition to statistics, the panel also offers guidance on how to control the game: the snake can be navigated using the arrow keys or the W, A, S, and D keys for up, left, down, and right respectively. To begin the game, the player must press the Enter key, while the P key can be used to pause and resume the game at any point.

The Clock class serves as the timekeeper for the game. It tracks the number of cycles (or frames) that have elapsed since the game began and is responsible for managing the game's timing logic. It can pause the game when necessary and reset the timing variables when the game ends. The Direction class, meanwhile, determines which direction the snake is currently facing and ensures that the snake cannot move in the reverse direction instantly, preventing invalid movements. This class plays a key role in processing user input and translating it into directional logic that the snake follows.

Another supporting class is TileType, which is used to define the type of object occupying each tile on the game grid. This can include the apple, the snake's body, or the snake's head. By categorizing each tile, the game engine can easily detect collisions and manage rendering with precision. The heart of the game lies in the SnakeGame class, which integrates all other classes and acts as the main controller. It handles the overall game loop and core mechanics such as collision detection, apple consumption, score updates, and game resetting. When the snake consumes an apple, its length increases, the fruit counter is incremented, and the player's score is updated based on the "Fruit Score". Notably, the "Fruit Score" starts at 100 points and gradually decreases with time, adding an element of urgency to the gameplay. If the snake collides with the wall or its own body, the game sets a flag indicating game over, halts movement, and displays the corresponding message.

When the game is launched, the user first sees the start screen where the game window is divided into two sections: the main game panel and the side panel. The game panel features the grid where gameplay occurs, while the side panel outlines statistics and control instructions. Once the player presses Enter, the game begins, and the snake, initially of fixed length, starts to move. As it consumes apples, its body grows, making navigation increasingly difficult. The game captures real-time inputs from the player to control movement, and the score dynamically reflects how quickly apples are consumed. Upon collision with a boundary or the snake's own body, the game ends and displays a "Game Over" message. The game can be restarted with another press of the Enter key. Additionally, the game supports pause functionality, where pressing the P key temporarily halts gameplay and pressing it again resumes from the same state.

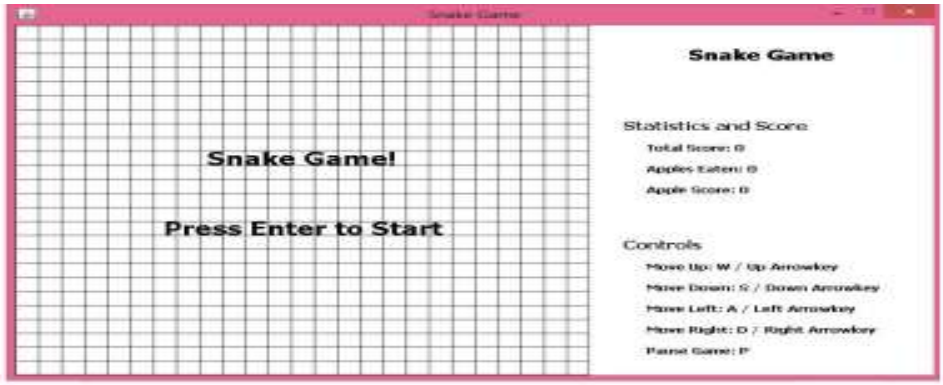

Fig 1. Start window

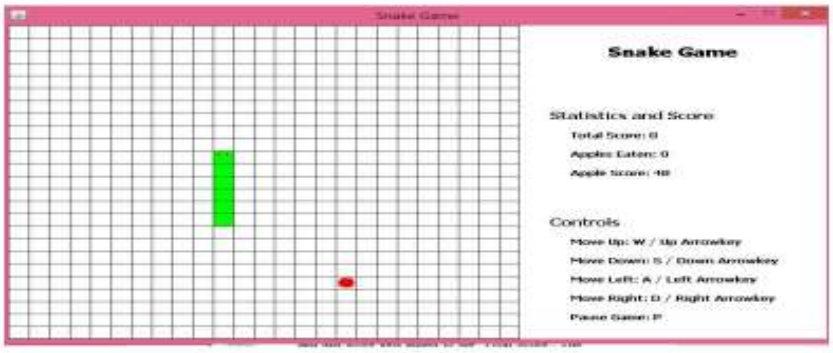

Fig 2. Running Game

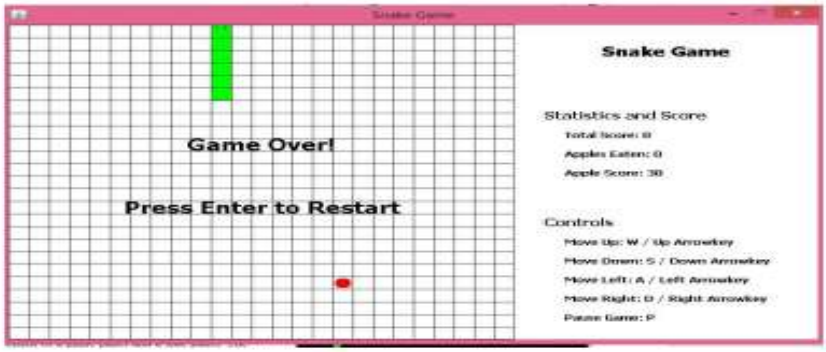

Fig 3. Game Over window

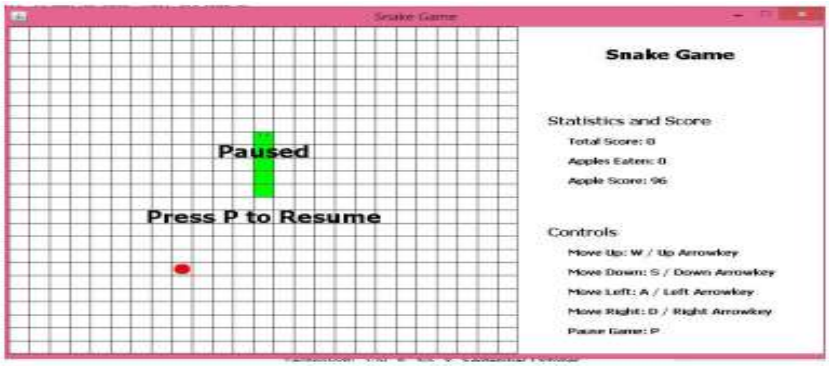

Fig 3. Paused Window

Overall, this implementation of the Snake game demonstrates a complete and functional arcade-style game using core Java principles. It highlights the use of object-oriented programming to manage game state, user input, graphics rendering, and real-time interaction. While relatively simple in design, this structure serves as a strong foundation for future enhancements, such as introducing new game modes, improving the user interface, or integrating artificial intelligence to automate gameplay. The project not only fulfills educational purposes for understanding Java but also presents opportunities for creative and technical exploration**.**

Implementation

Java-Based Snake Game:

- Written using Java SE in Eclipse IDE.
- Snake movement handled via key listeners.
- Grid-based rendering system for apple and snake.
- Collision detection with walls and self.
- Game resets on collision; score and statistics are displayed on side panel.

AI-Based Snake Game:

- Implemented using a custom neural network framework.
- DNA structure encodes the NN with matrices for weights and biases.
- GA Steps:
    1. **Initialize population** with random DNA.
    2. **Evaluate fitness** based on game performance.
    3. **Select parents** using roulette or tournament method.
    4. **Perform crossover** to generate offspring.

5. **Apply mutation** to introduce variation.
6. **Repeat** for multiple generations.

- Fitness function:
  - If apples < 10: steps * steps * 2^score
  - Else: same function with higher thresholds
- Recommends one hidden layer with 6 neurons.
- Common mutation rate: 0.5
- Typical population size: 2000

Results:

Traditional Game:

- Deterministic behaviour based on user input.
- Gameplay starts with a splash screen, transitions to game board.
- Statistics (score, apples eaten) updated in real time.
- Game over message appears on collision; restart via ENTER key.

AI Snake:

- Learning curve observed over generations.
- Initial snakes perform poorly due to random DNA.
- Over time, performance improves: more apples eaten, fewer collisions.
- Statistically, selection methods and mutation rates significantly affect learning speed.
- Visualizations demonstrate snake behavior refinement across generations.

Discussion:

The study reveals that while traditional Snake implementations are easier to code and understand, they lack adaptability and scalability. On the other hand, the AI model offers:

- Autonomous learning with minimal human interference.
- Capability to learn complex behaviors through training.
- Flexibility in experimenting with different AI architectures and hyperparameters.

However, the AI model also presents complexity in implementation and computational demand. Selecting appropriate GA parameters (mutation rate, selection type, crossover method) is critical and often problem-specific.

Conclusion:

The development of the classic Snake game using Java within the Eclipse IDE serves as a compelling demonstration of how fundamental programming concepts can be effectively combined to build a fully functional and interactive game. Through the application of object-oriented programming principles, modular class design, and real-time graphical rendering, this project showcases not only the technical feasibility but also the educational value of game-based learning in software development.

Each component of the game—ranging from user interface rendering and game logic to input handling and timing control—was encapsulated within a specific class, making the codebase well-structured and highly maintainable. The BoardPanel and SidePanel classes were pivotal in creating an intuitive user interface, delivering both visual feedback and game statistics in real-time. Meanwhile, supporting classes such as Direction, Clock, and TileType managed background operations and game state, contributing to a smooth and logical flow of gameplay. The central SnakeGame class served as the core engine, coordinating the actions of all other classes and managing the sequence of events including snake movement, fruit generation, collision detection, score calculation, and game resetting.

From a user experience perspective, the game provides a satisfying and nostalgic arcade feel, with responsive controls and an elegant scoring mechanism. The decreasing "Fruit Score" over time adds a strategic challenge, encouraging players to act quickly and efficiently to maximize points. The implementation also includes essential features such as game pausing, restart capability, and collision-based game termination, which contribute to a complete and user-friendly experience.

From a pedagogical standpoint, this project reinforces key programming skills such as class design, method abstraction, graphical rendering using Java's AWT/Swing libraries, and real-time input processing. It bridges theoretical knowledge with hands-on application, allowing learners to see the immediate effects of their code in a visual and interactive context. The modularity of the system also allows for easy extension—new features such as multiple difficulty levels, additional power-ups, improved graphics, or even AI-based snake control can be integrated without disrupting the existing structure.

In summary, the Java-based Snake game is more than a nostalgic project; it is a rich platform for learning and creativity. It encourages the exploration of software engineering principles in a fun and engaging way and lays the groundwork for transitioning into more complex game development or artificial intelligence integration in the future. This project not only meets the goals of functional game creation but also opens the door to a deeper understanding of system design and user-centered programming in Java.

Future Work:

- **Integrate Reinforcement Learning Techniques**: Incorporate algorithms such as Q-Learning and Deep Q-Networks (DQN) to compare performance and learning efficiency with Genetic Algorithm + Neural Network (GA+NN) methods.
- **Extend to Advanced Game Scenarios**: Expand the traditional Snake game into 3D environments or introduce multi-agent competitive/cooperative gameplay for enhanced complexity and research applicability.
- **Implement Dynamic Difficulty Adjustment (DDA)**: Adapt game difficulty in real time based on the agent's performance to create more robust and generalizable learning agents.
- **Use Convolutional Neural Networks (CNNs)**: Employ CNNs to process raw pixel input from the game environment, enabling vision-based learning and enhancing generalization across different game layouts.
- **Port to Web-Based Platform**: Leverage JavaScript and TensorFlow.js to develop a browser-compatible version of the Snake game, allowing interactive visualization and real-time AI experimentation directly in the web browser.
- **Design a Modular AI Framework**: Create a plug-and-play architecture that allows easy integration, comparison, and visualization of different AI strategies, fostering experimentation and educational use.

References:

1] Mugdhama Patil, R. Neha, Pravachana Mallapu, Rajasekhar Sastry, B.V. Ramana Murthy, and C. Kishor Kumar Reddy. "Snake Game." *Global Journal of Engineering Science and Researches*, ICITAIC-2019.

2] Piotr Białas. "Implementation of Artificial Intelligence in Snake Game Using Genetic Algorithm and Neural Networks." *Faculty of Applied Mathematics, Silesian University of Technology*, 2019.

3] David Brackeen, Bret Barker, and Lawrence Vanhelsuwe. *Developing Games in Java*, New Riders, 2003.

4] Andrew Davison. *Killer Game Programming in Java*, O'Reilly Media, 2005.

5] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. "A Fast Elitist Non-dominated Sorting Genetic Algorithm for Multi-objective Optimization: NSGA-II." *Springer*, 2000.

6] D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989.

7] B. Eckel. *Thinking in Java*, 4th Edition, Prentice Hall, 2006.

8] J. Horn, N. Nafpliotis, and D.E. Goldberg. "A Niched Pareto Genetic Algorithm for Multiobjective Optimization." *IEEE Conference on Evolutionary Computation*, 1994.

9] F. Bonanno, G. Capizzi, A. Gagliano, and C. Napoli. "Optimal Management of Various Renewable Energy Sources by a New Forecasting Method." *IEEE Symposium on Power Electronics and Drives*, 2012.

10] Marcin Woźniak and Dariusz Połap. "Hybrid Neuro-Heuristic Methodology for Simulation and Control of Dynamic Systems." *Neural Networks*, Vol. 93, pp. 45–56, 2017.

11] Edgington, J. and Leutenegger, S. "A Games First Approach to Teaching Introductory Programming." *SIGCSE 2007, Covington, Kentucky*, 2007.

12] Saloni Jain. “Developing Games in Java for Beginners.” *International Journal for Research in Applied Science & Engineering Technology (IJRASET)*, Volume 4, Issue III, March