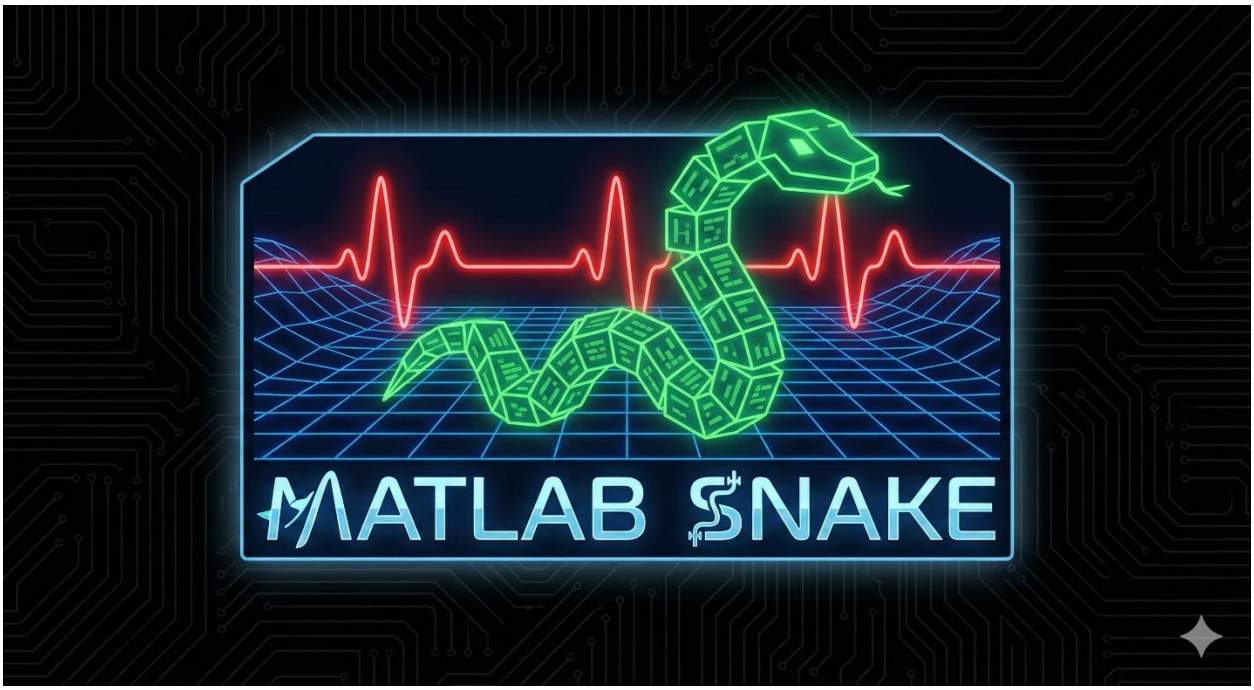
COMPUTER AIDED GRAPHICS
FINAL PROJECT



Submitted by: *Tarniță Robert Gabriel*

University: *Technical University of Cluj-Napoca*

Faculty: *Faculty of Electronics, Telecommunications and Information Technology*

Course: *Computer Aided Graphics (CAG)*

Professors: *Cirlugea Mihaela Cristina & Farago Paul*

Current Academic Year*: 2025-2026*

***Abstract**: This document details the design, mathematical modeling, and implementation of "SnakeGame v2.0," a developed graphical interface built in MATLAB. The project moves beyond procedural programming to utilize Object-Oriented paradigms through a modular Class Directory structure (@SnakeGame). This architecture integrates real-time mathematical signal processingg (trigonometric proximity sensors), linear system decay models (hunger mechanics), and matrix-based user customization to offer an enjoyable and intuitive user experience.*

Table of Contents

1.0 Executive Summary

The "**SnakeGame v2.0**" project represents a comprehensive effort to bridge the gap between theoretical mathematical concepts and practical software application. Developed within the MATLAB ecosystem using Handle Graphics, this application simulates a complex interactive environment that responds dynamically to user inputs and internal state variables.

Unlike traditional implementations of the Snake arcade game, which rely solely on coordinate geometry, this project introduces "Biological Feedback" systems. These systems utilize continuous mathematical functions—specifically Sine waves and Linear Decay equations—to visualize abstract concepts such as "danger" and "hunger" in a graphical format. Furthermore, the application features a "Developed Interface" that includes a secondary window for deep parameter customization, demonstrating proficiency in multi-window application management and data parsing.

2.0 Project Objectives and Requirements

The primary goal of this project was to demonstrate mastery of Computer Aided Graphics concepts by fulfilling the following strict requirements:

1. **Originality in GUI Design:** The interface must not be a copy of laboratory examples but a transformed, unique layout.
2. **Mathematical Foundation:** The project must include figures or characteristics derived explicitly from mathematical equations.
3. **User-Modifiable Parameters:** The interface must allow the user to modify internal variables (such as color or speed) via edit boxes or text areas.

   Graphical Elements: The use of specific MATLAB functions (***imagesc***, ***bar***, ***line***, ***uifigure***) to render data.

4. **External Assets:** The integration of raster graphics (JPG/PNG) to represent game states.

This documentation will demonstrate how each of these objectives was met and exceeded through the implementation of the "Snake Stylist" matrix engine and the "Proximity Heartbeat" sensor.

3.0 Mathematical Modeling and Logic

A core component of this project is the translation of numerical data into visual feedback. This section details the two primary mathematical models driven by the game engine.

3.1 The Proximity Sensor: Trigonometric Analysis

To provide the player with a sense of urgency without relying on simple numerical counters, the system implements a "Proximity Alert" system. This system mimics a biological heartbeat monitor using a trigonometric Sine Wave function.

The fundamental equation governing the sensor is:

$$y(t)=A(d)\cdot sin(f(d)\cdot t)$$

Where:

- *y(t)* is the vertical position of the graph line at time step *t*.
- d is the Euclidean distance between the snake's head vector (*x1*, *y1*) and the nearest collision block vector (*x2*, *y2*).

Derivation of Amplitude *(A)*:

The amplitude represents the intensity of the danger. It is modeled as an inverse function of distance:

$$A(d) = \frac{Ka}{max(d, 1)}$$

Where *a* is a scaling constant (set to 10 in the code). As the distance *d* approaches 0, *the amplit*ude *A* approaches its maximum, creating tall, violent spikes in the graph.

Derivation of Frequency *(f):*

The frequency represents the speed of the heartbeat. This is also inversely proportional to distance:

$$f(d) = \frac{K_f}{\max(d, 1)}$$

Where *Kf* is a scaling constant (set to 15). As the snake approaches a wall, *f* increases, causing the sine wave to oscillate rapidly. This mathematical relationship creates an intuitive "panic" response in the user.

Figure 1: The Proximity Sensor Graph visualizing the Sine Wave equation.

3.2 Biological Simulation: Linear Decay Models

The game implements a survival mechanic based on "Hunger." Unlike the Proximity Sensor, which is oscillatory, Hunger is modeled as a linear system that decays over time. This introduces a resource management element to the gameplay.

The equation governing the hunger state H at any given frame n is:

$$H_{n+1} = H_n - \Delta decay$$

In the current implementation, $\Delta decay$ is fixed at 0.3 units per frame. This creates a constant downward pressure on the player.

The consumption of fruit acts as a step-function input to this system:

$$H_{n+1} = min((H_n + 30),100)$$

This equation ensures that the hunger value creates a "sawtooth" wave pattern over time (gradual decline followed by sharp increase), which is visualized using the ***bar*** plot function in the UI.

3.3 Matrix Algebra in Color Customization

The customization feature of the application relies on Linear Algebra concepts. The color of the snake's body is not stored as a single value, but as a matrix *C* of dimensions *N x 3*, where *N* is the number of distinct color bands.

$$C = \begin{bmatrix} R_1 & G_1 & B_1 \\ R_2 & G_2 & B_2 \\ \vdots & \vdots & \vdots \\ R_n & G_n & B_n \end{bmatrix}$$

During the rendering phase, the game loop determines the color of the *i-th* body segment using the modulo operation to cycle through the rows of this matrix:

$$RowIndex = (i \bmod N) + 1$$

This allows the user to define complex, repeating patterns (e.g., Red-Green-Blue-Red-Green-Blue) simply by inputting a matrix of values.

4.0 System Architecture

4.1 Object-Oriented Design (Class Structure)

The application utilizes MATLAB's Object-Oriented Programming (OOP) capabilities. The entire application is encapsulated within a single class, SnakeGame, which inherits from the handle superclass.

Inheriting from handle allows the ***SnakeGame*** object to be passed by reference. This is particularly vital in the new modular structure; because the class is a handle, external methods like ***setDifficulty***.m can directly modify the properties of the active game instance without the overhead of creating data copies.

Key Properties:

- ***mtx :*** A $100\ x\ 100\ x\ 3$ matrix representing the RGB pixel data of the board.
- *snakeBody*: An $N\ x\ 2$ coordinate list managing spatial geometry.
- ***sensorData***: A FIFO (First-In-First-Out) buffer array used to store the trailing values of the sine wave for plotting.

4.2 The Game Loop and State Management

The core engine is driven by a ***while*** loop located in ***the startGame()*** method. This loop runs continuously until specific flags (***gameOver*** or ***pauseGame***) are toggled.

The Loop Lifecycle:

1. **Input Processing:** Check ***nextDirection*** buffer (set by keyboard listeners).
2. **Logic Update:** Calculate new head position based on direction.
3. **Collision Detection:** Check geometric intersection between Head Vector and Obstacle Matrices.
4. **Math Update:** Recalculate Hunger *(H)* and Sine Wave *(y)*.
5. **Render:** Call ***imagesc*** to update the visual grid and ***drawnow*** to flush the graphics pipeline.

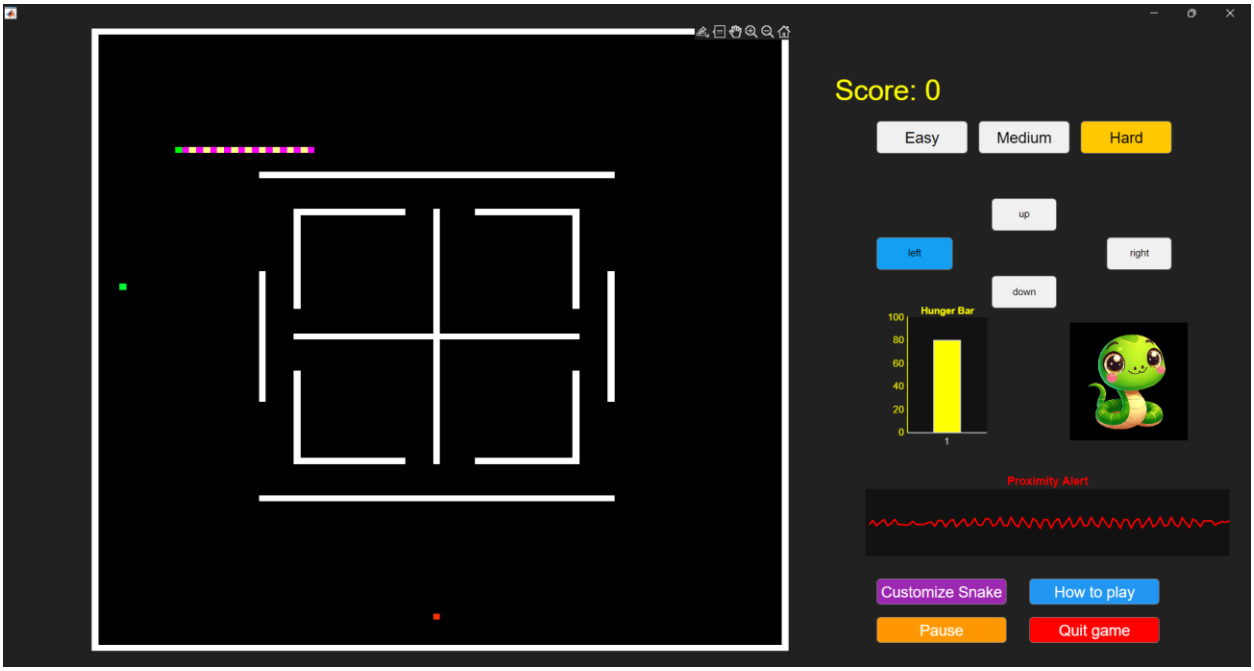


Figure 2: The Main Game Loop in action, showing the interaction between the grid and the side panels.

4.3 Modular Software Architecture

To improve maintainability and follow industry standards for software deployment, **SnakeGame v2.0** utilizes the MATLAB Class **Directory (@)** convention.

Encapsulation: The ***@SnakeGame*** folder acts as a private namespace, ensuring that game-specific methods are only accessible via a class instance.

Separation of Concerns: By isolating the engine (***startGame.m***), the UI setup ***(initializeGameUI.m***), and the level logic ***(setDifficulty.m***) into separate files, the code becomes easier to debug and scale.

Handle Class Advantage: Because the class inherits from handle, these external files can directly modify properties like ***collisionBlocks*** or ***snakeBody*** without the memory overhead of passing large data structures

5.0 Graphical User Interface (GUI) Design

The GUI was designed to be "adequately presented" and "original," avoiding the cluttered look of standard student projects. The layout strategy divides the screen into four functional zones.

5.1 Layout Strategy and Zoning

Zone A: The Interactive Arena (Left Panel) Occupying 60% of the screen, this zone contains the ***uiaxes*** where the game grid is drawn. To improve usability, critical action buttons ("Start Game," "Try Again") are overlaid directly on top of this zone. This minimizes mouse travel distance for the user between rounds.

Zone B: Information & Settings (Top Right) This zone hosts the Score display and Difficulty controls. The Score is rendered in high-contrast yellow text to remain visible against the dark background. The Difficulty buttons (Easy/Medium/Hard) act as state switches that instantly reconfigure the ***collisionBlocks*** matrix.

Zone C: Navigation (Middle Right) A directional pad (D-Pad) is recreated using ***uibutton*** elements. While keyboard control is primary, these buttons provide accessibility for mouse-only users. They are centered perfectly within the panel to create visual balance.

Zone D: Biological Feedback (Bottom Right) This is the "Dashboard" of the application. It groups the Hunger Bar, the Mood Image, and the Proximity Sensor. By grouping these elements, the user can assess their "Health" status in a single glance without scanning the entire screen.

5.2 Visual Hierarchy

The interface uses a Dark Mode theme (***Color*** set ***to [0.12 0.12 0.12])*** to reduce eye strain and make the bright RGB colors of the snake pop. Action buttons use semantic coloring:

- **Green:** Positive actions (Start, Apply).
- **Red:** Negative/Danger actions (Quit, Proximity Line).
- **Yellow:** Warning/Info (Hunger, Score).
-

6.0 Implementation Details

6.1 Graphic Rendering Techniques

The project avoids using thousands of individual rectangle objects to draw the grid, which would be computationally expensive. Instead, it utilizes the ***imagesc*** function.

The game board is maintained in memory as a 3D Matrix ***(100x100x3***).

- Layer 1: Red Channel
- Layer 2: Green Channel
- Layer 3: Blue Channel

At every frame, the logic simply updates specific indices in this matrix (setting them to the Snake Color or Black). The ***imagesc*** function then renders this matrix as a single image. This technique ensures the game runs smoothly even at high speeds (Hard Mode).

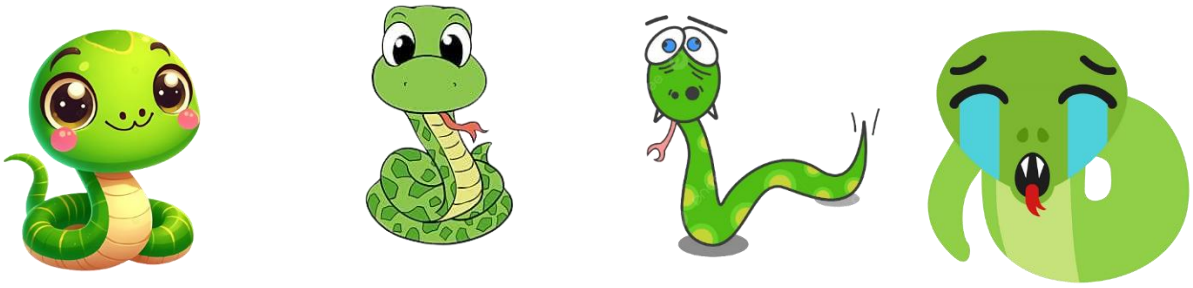
6.2 Dynamic Asset Management

To meet the requirement of presenting JPG/PNG images, the application utilizes a dynamic asset loader within the game loop.

A ***uiaxes*** named ***moodImgAx*** serves as the container. The logic checks the linear ***currentHunger*** variable and classifies it into four states:

1. **Happy:** Hunger > 75
2. **Neutral:** 50 < Hunger < 75
3. **Hungry:** 25 < Hunger < 50
4. **Sad:** Hunger < 25

Based on this classification, the code executes ***imshow('filename.png').*** This creates a reactive avatar that gives the snake "personality," enhancing the user's emotional connection to the game.



7.0 Advanced Features: The Snake Stylist

The most sophisticated feature of the application is the "Snake Stylist" customization panel. This fulfills the requirement for a developed, multi-window interface with user-modifiable parameters.

7.1 Data Parsing and Error Handling

The Stylist panel contains two input fields: a ***uieditfield*** for the Head and a ***uitextarea*** for the Body. Because these fields accept raw string input, the system requires robust parsing logic.

When the user clicks "Apply," the system performs the following steps:

1. **String Tokenization:** It reads the multi-line string from the text area.
2. **Conversion:** It uses ***str2num*** to attempt to convert the text into a double-precision matrix.
3. **Validation:** It checks the dimensions of the resulting matrix.
    - *Condition A:* Columns must equal 3 (RGB).
    - *Condition B:* Values must be ≤ 1 and ≥ 0.
4. **Application:** If valid, the ***snakeBodyColor*** property is updated immediately.

7.2 UI/UX for Parameter Modification

The secondary window is designed to look like a professional property inspector.

- **Vector Input:** The Head Color is treated as a simple vector.
- **Matrix Input:** The Body Color is treated as a matrix.

This distinction highlights the difference between static properties (the head) and cycling properties (the body segments). The layout uses generous padding (25px margins) and large fonts (24pt) to ensure the text data is readable.

Figure 4: The "Snake Stylist" panel parsing matrix data for body coloration.

8.0 Testing and Validation

The application underwent rigorous testing to ensure stability across all three difficulty modes.

- **Boundary Testing:** The "Game Over" logic was tested against all four walls and self-collision scenarios.
- **Input Sanitization:** The Customization Panel was tested with invalid inputs (letters, numbers > 1, empty strings) to ensure ***the try-catch*** blocks successfully prevented application crashes.
- **Performance:** The game was stress-tested on "Hard" mode to verify that the ***imagesc*** rendering method maintains a playable frame rate even when the Proximity Sensor calculations are occurring rapidly.

9.0 Future Improvements

While the current version meets all specifications, future iterations could include:

1. **High-Score Persistence:** Saving the ***scoreLabel*** value to a local .mat file to persist data between sessions.
2. **Audio Feedback:** Integrating ***audioplayer*** to trigger sounds when the Proximity Sensor sine wave reaches peak amplitude.
3. **Complex Obstacles:** Implementing moving obstacles that require dynamic recalculation of the distance matrix.
4.

10.0 Conclusion

The "SnakeGame v2.0" project successfully serves as a capstone for the Computer Aided Graphics course. It transcends the basic requirements by implementing a fully Object-Oriented architecture, sophisticated mathematical signal processing for user feedback, and a professional-grade User Interface.

By allowing users to interact with the game's internal mathematics—altering color matrices and reacting to trigonometric proximity alerts—the project demonstrates a deep understanding of how to visualize and manipulate data in MATLAB. The resulting application is robust, extensible, and user-friendly, qualifying as a final deployable project.