SNAKE GAME IN MATLAB

Bianca-Elena Raicu

Table of Contents

Introduction 3

Theoretical Background 3

MATLAB GUI 3

Grid representation 4

Event Handling 4

Dark mode 4

Dynamic Updates 5

Collision Detection 5

Scoring System 6

Game State Management 6

Difficulty Levels 6

Main functionalities and key points 7

Design 7

Start/pause/stop menu 8

The start button 8

The pause button 8

The stop button 9

Game implementation 10

Creating the head of the snake 10

Creating the food 10

Applying the grid and plotting the snake and the food 10

Moving the snake 11

Wall collision and self collision 11

Eating 13

Score and speed increasing 13

Arrows control 13

More functionalities 14

Difficulty choosing 14

Dark mode 14

Game play 15

References 17

Introduction

The Snake Game is a classic arcade game that has been enjoyed by players worldwide for decades. In this project, a modern adaptation of the Snake Game is developed in MATLAB using MATLAB's graphical user interface (GUI) capabilities to create an interactive and visually engaging experience. The objective of the game is to navigate the snake to consume food items while avoiding collisions with the walls or itself, given the fact that more food eaten means increasing the size of the snake (increasing also the difficulty level of the game).

The aim of this project is to create a reenactment of the beloved game using MATLAB using the graphical programming and logical functionality in order to create a simple and effective gaming application. It requires a wide range of programming concepts that are fundamental to software development. These include real-time event handling, collision detection, and graphical updates, all of which are applicable to more complex software systems. Working on this project provides an opportunity to explore these concepts in a focused and enjoyable context.

Theoretical Background

The Snake Game is a classic example of a real-time application involving dynamic graphics and event-driven programming. For implementing it, I used tools and concepts such as:

MATLAB GUI

GUIs (graphical user interfaces) allow for simple mouse-based control of software applications.

MATLAB apps are stand-alone MATLAB programs with a graphical user interface (GUI) that automate a task or calculation. The GUI typically includes controls such as menus, toolbars, buttons, and sliders.

Many MATLAB products, such as Curve Fitting Toolbox, Signal Processing Toolbox, and Control System Toolbox, include apps with custom user interfaces. You can also create your own custom apps, including corresponding user interfaces, for other people to use.

MATLAB provides a robust platform for developing custom graphical user interfaces (GUIs), making it an excellent choice for creating interactive applications like a Snake game. The flexibility of MATLAB's GUI development tools allows for the integration of graphical elements and game logic in a single environment, enabling developers to design intuitive and engaging user experiences.

MATLAB's plotting and drawing capabilities enable the creation of visually appealing game elements, such as the snake, food, and game area. These can be rendered dynamically within axes or graphical panels. MATLAB GUIs can include elements such as buttons, sliders, and keypress callbacks, which are ideal for handling player input. For a Snake game, keypress callbacks can be used to detect arrow key inputs to control the snake's movement.

Grid representation

The game board is represented by a grid with cells corresponding to positions. The coordinates of the grid are used to place the food and to move the snake using the commands from the arrows. For the implementation of this, I chose a grid of 20x20, with square sized cells.

Event Handling

Event handling is a cornerstone of the Snake Game, enabling real-time interaction between the user and the application. The snake's movement is controlled via keyboard inputs, requiring the application to capture and respond to user events efficiently.

MATLAB's APP Designer provides a powerful platform for creating interactive applications. One of the key features to enhance user interaction is the KeyPressFcn and WindowKeyPressFcn callbacks. These functions allow developers to specify actions that occur when a user presses a key while interacting with the app.

The KeyPressFcn callback is triggered whenever a key is pressed while a specific UI component is in focus. For instance, if assigned to a UIAxes or a button, it will capture the keypress events only when that component has user focus. This targeted functionality makes KeyPressFcn ideal for scenarios where key-based interactions are limited to a particular control or component, such as navigating within a specific table or responding to a key press for a single control element.

The WindowKeyPressFcn callback, on the other hand, applies to the entire application window. It captures key presses regardless of which UI component is in focus, making it more suitable for handling global keyboard interactions. This is especially useful in gaming applications, like the Snake Game, where the user needs to control the snake's movement without worrying about which part of the app is active or in focus.

In a game designed in App Designer, WindowKeyPressFcn is more effective than KeyPressFcn for capturing directional inputs (e.g., arrow keys) because the player's focus is on gameplay rather than UI elements. By binding the WindowKeyPressFcn to directional key events, the app ensures seamless interaction even during intense gameplay. For example, when the player presses an arrow key, the snake's movement updates immediately, irrespective of where the player's focus lies in the UI.

Moreover, both callbacks can be combined for advanced functionalities. For instance, KeyPressFcn can be used for component-specific shortcuts, while WindowKeyPressFcn manages overarching gameplay controls.

Dark mode

Giving the fact that more and more people use dark mode for their applications, I decided to implement one as well, where the colors of the snake and grid change to magenta and black. This mode is better for the eyes of the player, especially after a long session of playing the game.

Dynamic Updates

Smooth gameplay requires continuous updates to the graphical display and game logic. This is achieved using MATLAB's game loop in which the game updates the snake's position, it checks for food consumption and collisions and it updates the graphic of the snake, or it shows the end of the game if the collisions happened.

The snake's position is updated at regular intervals within the game loop, creating the perception of smooth motion. This involves incrementally modifying the position of the snake's head based on the player's input (e.g., arrow keys) while simultaneously adjusting the positions of the body segments to follow the head.

Dynamic updates also check whether the snake's head has reached the location of the food. If the positions match, the snake grows, the score gets updated and a new food is randomly placed on the game board, ensuring it does not overlap with the snake's body.

Within the game loop, continuous checks are performed for wall and self-collisions. If a collision is detected, the game responds dynamically: stopping the game, showing the game over message and displaying the score.

An essential feature of engaging gameplay is the incorporation of dynamic difficulty scaling, which ensures the game remains challenging as the player progresses. In the Snake Game, this is achieved by gradually increasing the snake's speed as the score rises, making it harder for the player to control the snake and avoid collisions.

Collision Detection

Collision detection ensures the game's integrity by ending the game when the snake collides with itself or the boundaries of the game board. This requires logical checks implemented within the game loop for wall collision and self-collision. If any of the condition proves to be true, then the game will finish with the message "Game over!"

Wall collision occurs when the snake's head moves beyond the defined boundaries of the game board. The position of the snake's head is continuously monitored, and each movement is checked against the limits of the play area (e.g., top, bottom, left, and right edges). If the snake's head crosses these limits, the game loop identifies the condition as true and terminates the game.

Self-collision detection adds a layer of complexity and requires careful tracking of the snake's body. In each frame of the game loop, the current position of the snake's head is compared against the positions of all other segments of the snake's body. If the head's position matches that of any body segment, a self-collision is flagged, and the game terminates.

Also through the collision detection, the player can become better and better by comparing the score with the previous ones he had while playing the game.

Scoring System

The scoring system is a fundamental element of the Snake Game, providing players with a clear sense of progression and accomplishment as they achieve higher scores by consuming food. This system not only tracks the player's performance but also directly impacts the gameplay dynamics, including difficulty scaling and end conditions.

In the Snake Game, the score increments each time the snake consumes a food item. This process is implemented using straightforward arithmetic operations. Every instance of food consumption adds a predefined value (usually one point) to the score counter. The updated score is then reflected in the graphical user interface (GUI) or display to keep the player informed of their progress.

In the classic Nokia Snake game, the maximum achievable score was capped at 781 due to the fixed grid size and the limitation of space for new food to appear. The score was directly tied to the number of cells on the game board, with each food item consumed occupying one additional cell. In this version of the Snake Game, the grid size may be larger, and the implementation might allow for a slightly higher maximum score.

Game State Management

Managing the state of the game is a critical aspect of designing a robust and user-friendly Snake Game. By implementing features such as pausing, restarting, and terminating the game, players are given more control over their experience, and the game becomes more adaptable to different situations. This requires careful integration of logical checks and event handling mechanisms within the game loop and user interface.

In this case we have the following states: active (playing), paused (temporar break), restart (reseting the game including the score, snake speed and size) and terminated (game over after a collision or when the player presses the stop button)

Difficulty Levels

The concept of adjustable difficulty levels adds depth to the Snake Game by scaling the challenge as the player progresses. This scalability ensures prolonged engagement and tests the player's reflexes and decision-making skills under increasing pressure. By systematically reducing the timer interval as the score rises, the snake's speed increases, making the game more demanding. This dynamic progression is a hallmark of well-designed games and enhances player satisfaction.

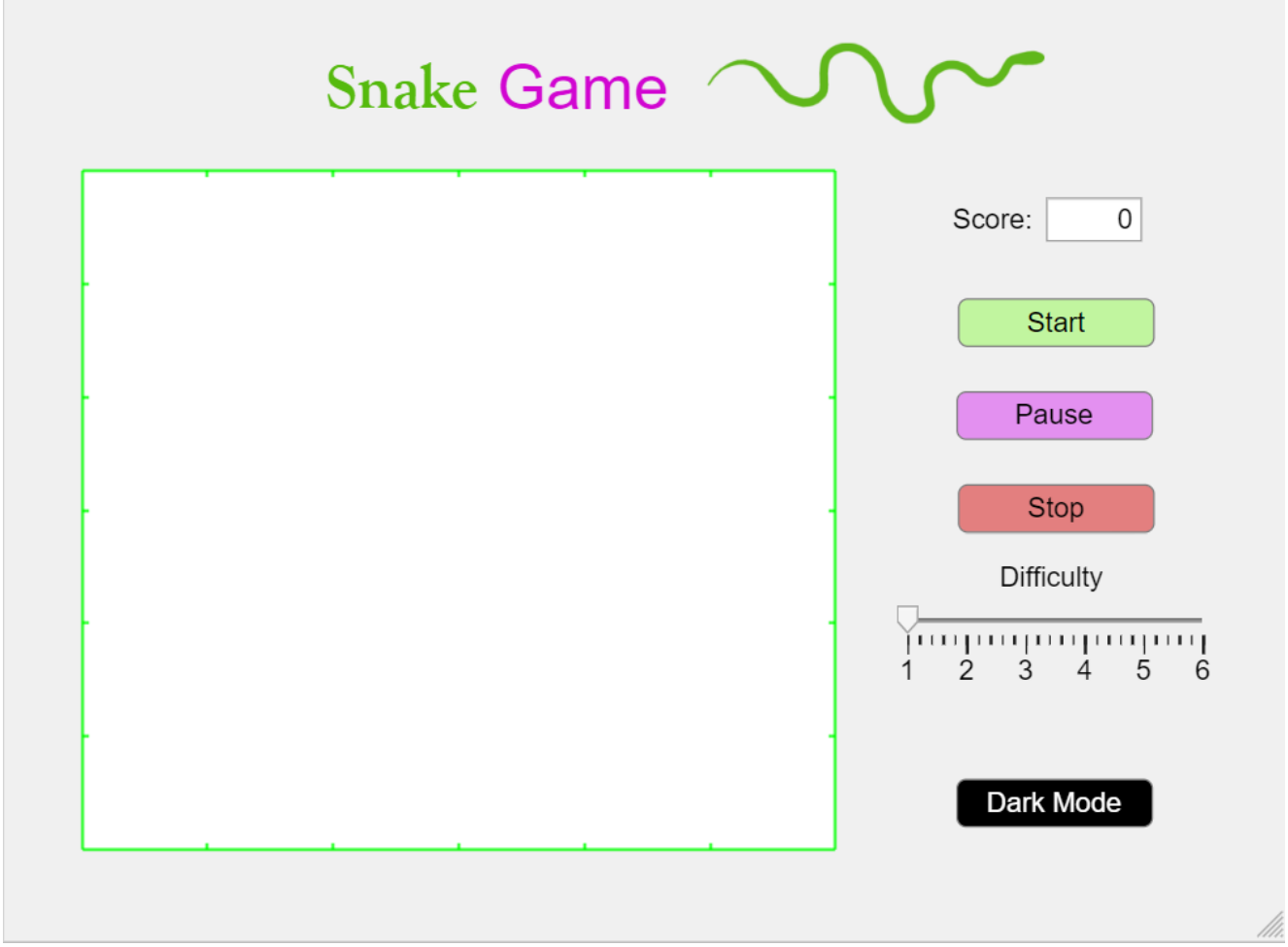
By integrating difficulty scaling into the Snake Game, the gameplay experience becomes richer, offering a rewarding challenge that grows alongside the player's skill level. This scalability ensures that the game remains fresh and compelling, even after multiple playthroughs.

Main functionalities and key points

Design

For the design of the game, I chose the colors green and magenta because they are complementary giving a nice touch with both the light mode (where the snake is green) and the dark mode (where the snake is magenta). The game consists of the following elements:

- Labels (for the title)
- The plot (for the game board)
- 4 buttons
  - Start (for starting the game)
  - Pause (for temporarily taking a break)
  - Stop (for terminating the game)
  - Dark mode (for changing the colors of the game)
- A slider (for choosing the difficulty)
- An edit field (for the score)
- A picture of a snake (for a more suggestive look, created by me in Figma)

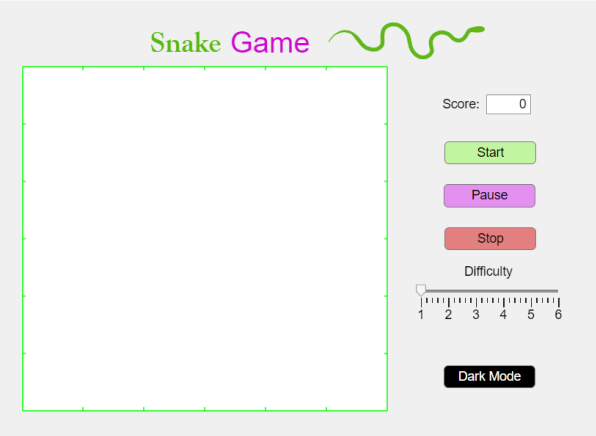


Start/pause/stop menu

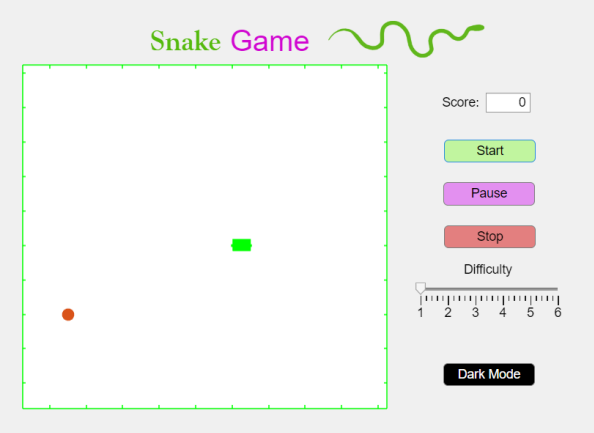
The buttons for start, pause and stop were configured with callbacks. I will show here a few of the functionalities.

The start button

In order for the game to start, the player has to play start. I put here the before and after. The whole code and functionality of it will be discussed separately in the game implementation section



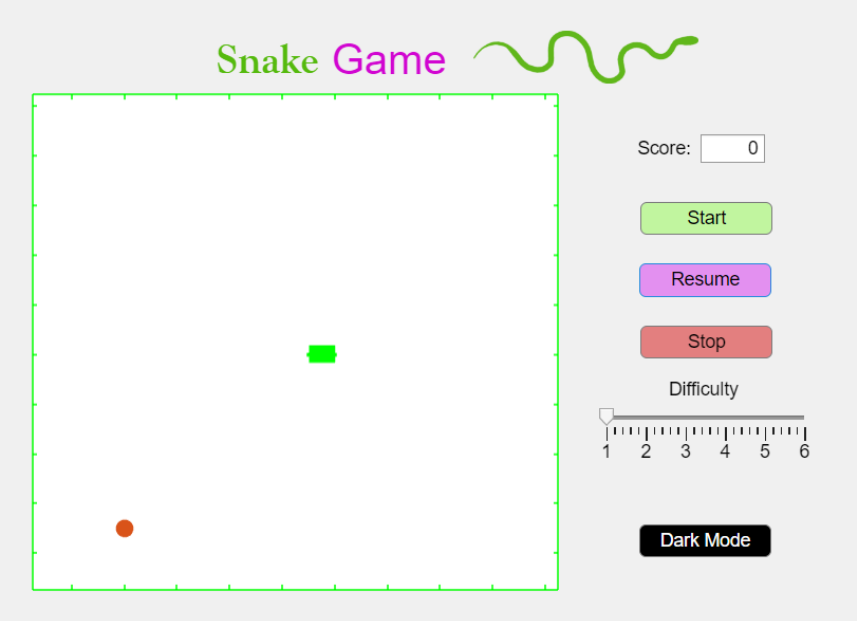
Before pressing play



After pressing play

The pause button

The pause button temporarily puts a break to the game, and lets the player decides when they want to resume the game, by changing the text on the Pause button with “Resume”

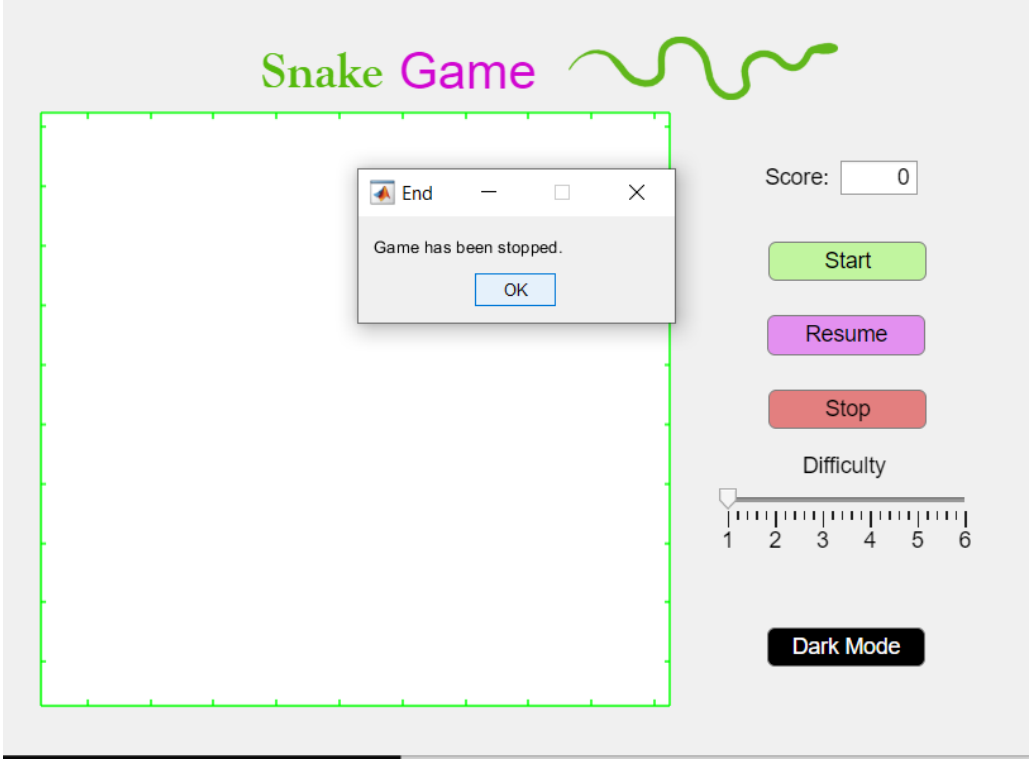


This was implemented with the help of the PauseButtonPushed callback

```
% Button pushed function: PauseButton
        function PauseButtonPushed(app, event)
             app.isPaused = ~app.isPaused; % invert pause state
            if app.isPaused
                app.PauseButton.Text = 'Resume'; % update button text for when
the game is paused
            else
                app.PauseButton.Text = 'Pause'; % update button text for when
the game is not paused
            end
        end
```

The stop button

It terminates the game, deleting the snake from the game board and also showing a message to let the player know the game ended.



It has been done using the StopButtonPushed callback

```
  % Button pushed function: StopButton
        function StopButtonPushed(app, event)
            app.isPaused = true; % pause the game
            msgbox('Game has been stopped.', 'End'); % show a message box
            delete(app.snakeHandle); % delete the snake
            delete(app.foodHandle); % delete the food
        end
    end
```

Game implementation

Creating the head of the snake

```
  s = zeros(100,2);

  s(1,1) = round(dim/2); %find the middle of the screen on the x coordinate
  s(1,2) = round(dim/2); %find the middle of the screen on the y coordinate
  s(2,1) = 5;
  s(2,2) = 6;
```

Creating the food

While doing that I was careful to check that the coordinates of the food do not overlap with the coordinates of the snake head

```
f = [0 0]; %food appearance
f(1) = randi([1, dim]); % Random x position
f(2) = randi([1, dim]); % Random y position

% Ensure food does not overlap with the snake's initial position
while any(ismember(s(1:length(nonzeros(s(:, 1))), :), f, 'rows')) || isequal(f,
s(1, :)) %condition for snake and food overlapping
      f(1) = randi([1, dim]); % regenerate x position
      f(2) = randi([1, dim]); % regenerate y position
end
```

Applying the grid and plotting the snake and the food

While doing this I used the variables gridColor, snakeColor and foodColor that modify depending on the mode used: dark mode or light mode.

```
% Apply grid background color
    set(app.UIAxes, 'Color', gridColor);

%Plot snake
app.snakeHandle = plot(app.UIAxes, s([1 2], 1), s([1 2], 2), 'Marker', ".",
'LineWidth', 10, 'Color', snakeColor);
hold(app.UIAxes, "on");
%Plot food
app.foodHandle = plot(app.UIAxes, f(1, 1), f(1, 2), 'Marker', ".",
'MarkerSize', 30, 'MarkerFaceColor', foodColor);
```

Moving the snake

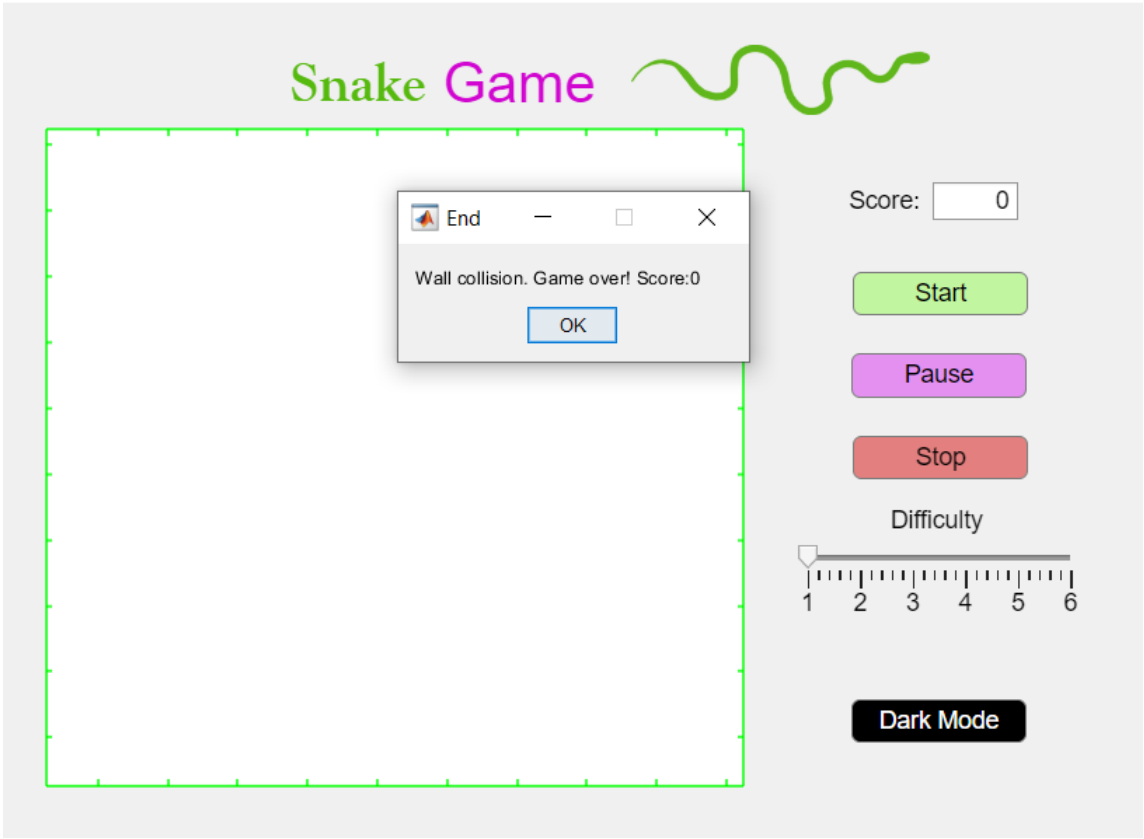
For moving the snake we need to move the head depending on the direction we got from the arrows, and the body to follow the last position.

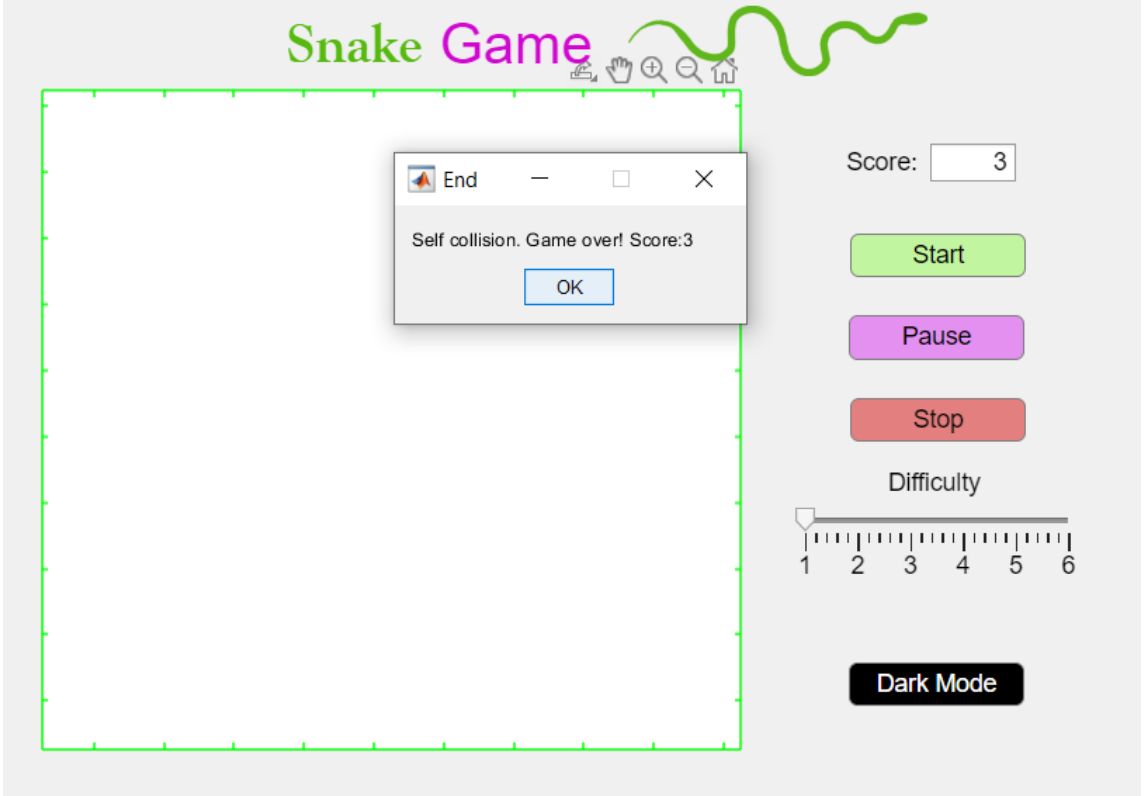
```
s_old = s;

%Update head position
if(app.dir == 0) %the snake moves left
    s(1,1) = s_old(1, 1) - 1;
    s(1,2) = s_old(1, 2);
elseif(app.dir == 1) %the snake moves up
    s(1,1) = s_old(1, 1);
    s(1,2) = s_old(1, 2) + 1;
elseif(app.dir == 2) %the snake moves right
   s(1,1) = s_old(1, 1) + 1;
   s(1,2) = s_old(1, 2);
elseif(app.dir == 3) %the snake moves down
   s(1,1) = s_old(1, 1);
   s(1,2) = s_old(1, 2) - 1;
end

%Update body position
for i = 2:1:length(nonzeros(s(:,1)))
    s(i,1) = s_old(i-1, 1);
    s(i,2) = s_old(i-1, 2);
end
```

Wall collision and self collision

In the case that the snake hits a wall, or it's own body, the game ends meaning that the snake stopps, it is showed a message that says the cause of the death, Game over! and the score. And after that the snake and food are deleted from the game board.

```
%Wall collisions
if(s(1,1)>dim||s(1,1)<1||s(1,2)>dim||s(1,2)<1)
       msgbox(strcat('Wall collision. Game over! Score: ', int2str(app.score)),
'End') %showing a message
       delete(app.snakeHandle);
       delete(app.foodHandle);
       break
end

%Self collisions
if(isempty(find(prod(s(1,1:2)==s(2:end,1:2), 2), 1))==0)
       msgbox(strcat('Self collision. Game over! Score: ', int2str(app.score)),
'End') %showing a message
       delete(app.snakeHandle);
       delete(app.foodHandle);
       break
end
```


Snake Game
End
Wall collision. Game over! Score:0
OK
Score:
0
Start
Pause
Stop
Difficulty
1 2 3 4 5 6
Dark Mode


Snake Game
End
Self collision. Game over! Score:3
OK
Score:
3
Start
Pause
Stop
Difficulty
1 2 3 4 5 6
Dark Mode

Eating

If the head of the snake has the same coordinates as the food, then we move the snake forward while mentaining the last square so that the snake grows. After that, new food is generated, while ensuring that the it is not overlapping with the snake's head or body.

```
  %Eating
  if(s(1,[1 2])==f)
       s(length(nonzeros(s(:,1)))+1,1) = s_old(length(nonzeros(s_old(:,1))),1);
%moving the snake forward
       s(length(nonzeros(s(:,2)))+1,2) = s_old(length(nonzeros(s_old(:,2))),2);

       % Food generation logic
       f(1) = randi([1, dim]); % random x position
       f(2) = randi([1, dim]); % random y position

        % Ensure food does not overlap with the snake's position
        while any(ismember(s(1:length(nonzeros(s(:, 1))), :), f, 'rows')) ||
isequal(f, s(1, :)) %condition for snake and food overlapping
             f(1) = randi([1, dim]); % regenerate x position
             f(2) = randi([1, dim]); % regenerate y position
        end
```

Score and speed increasing

The score grows after eating a food. To raise the difficulty of the game, I also implemented a speed increase by decreasing the pause time, keeping it bigger than 0.1 so it does not become impossible to play.

```
app.score = app.score+1; %rising the score
v = max(v-0.02, 0.1); %ensures the v does now go lower than 0.1
```

Arrows control

For the arrows control I used the WindowsKeyPress function to get key that has been pressed so that I can change the direction accordingly. I used a system where 0 is left, 1 is up, 2 is right and 3 is down.

```
% Window key press function: UIFigure
function UIFigureWindowKeyPress(app, event)
       key = event.Key;
       newDir = app.dir; % Store the current direction
       if(strcmp(key, 'leftarrow') && app.dir~=2) %left arrow pressed
             newDir = 0;
       elseif(strcmp(key, 'rightarrow') && app.dir~=0) %right arrow pressed
             newDir = 2;
       elseif(strcmp(key, 'uparrow') && app.dir~=3) %up arrow pressed
             newDir = 1;
       elseif(strcmp(key, 'downarrow') && app.dir~=1) %down arrow pressed
             newDir = 3;
       end
```

More functionalities

Difficulty choosing

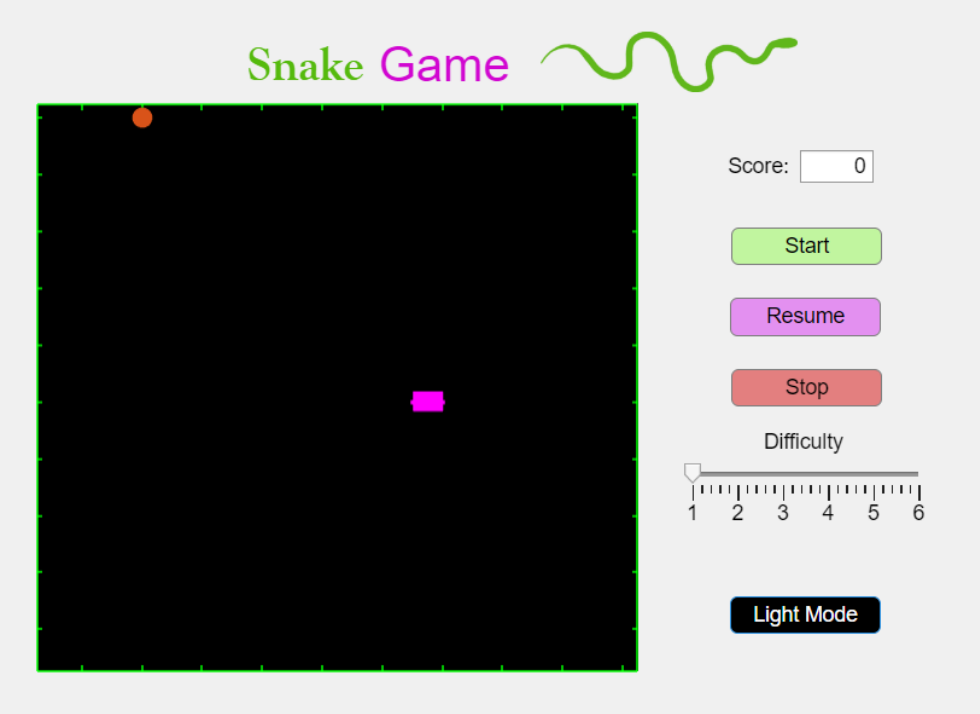
The players can chose what difficulty they want using the slider with 6 steps of difficulty from 1 to 6. In order for the integers and fractional numbers to be able to be used and also for the fact that the speed I wanted to present for this game was between 0.7 and 0.2, I came up with this formula to calculate the speed and I used in in the callback of the Slider.

```
% Value changed function: DifficultySlider
        function DifficultySliderValueChanged(app, event)
            difficulty = app.DifficultySlider.Value;
            app.gameSpeed = (8-difficulty)/10; %formula that ensures that the
speed will be between 0.7 and 0.2
        end
```

Dark mode

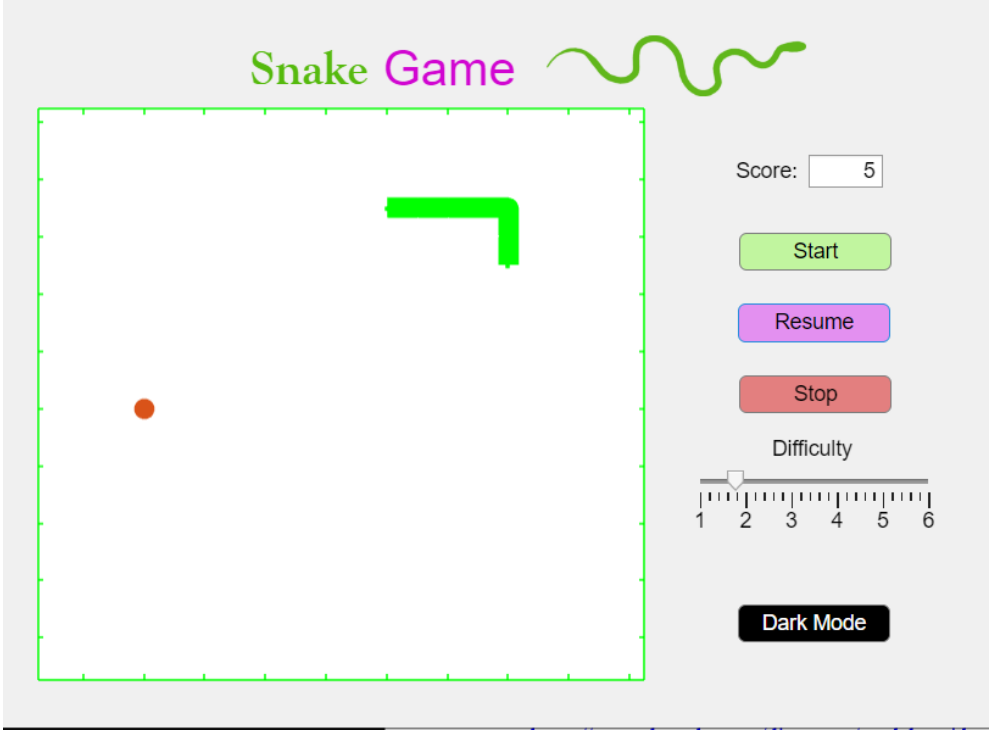
Because it is increasingly popular to use applications in dark mode, I also implemented that functionality, where the grid becomes black and the snake magenta. Also the button will display the writing “Light Mode” when changed to the dark mode. The mode is also checked when the start button is pressed.

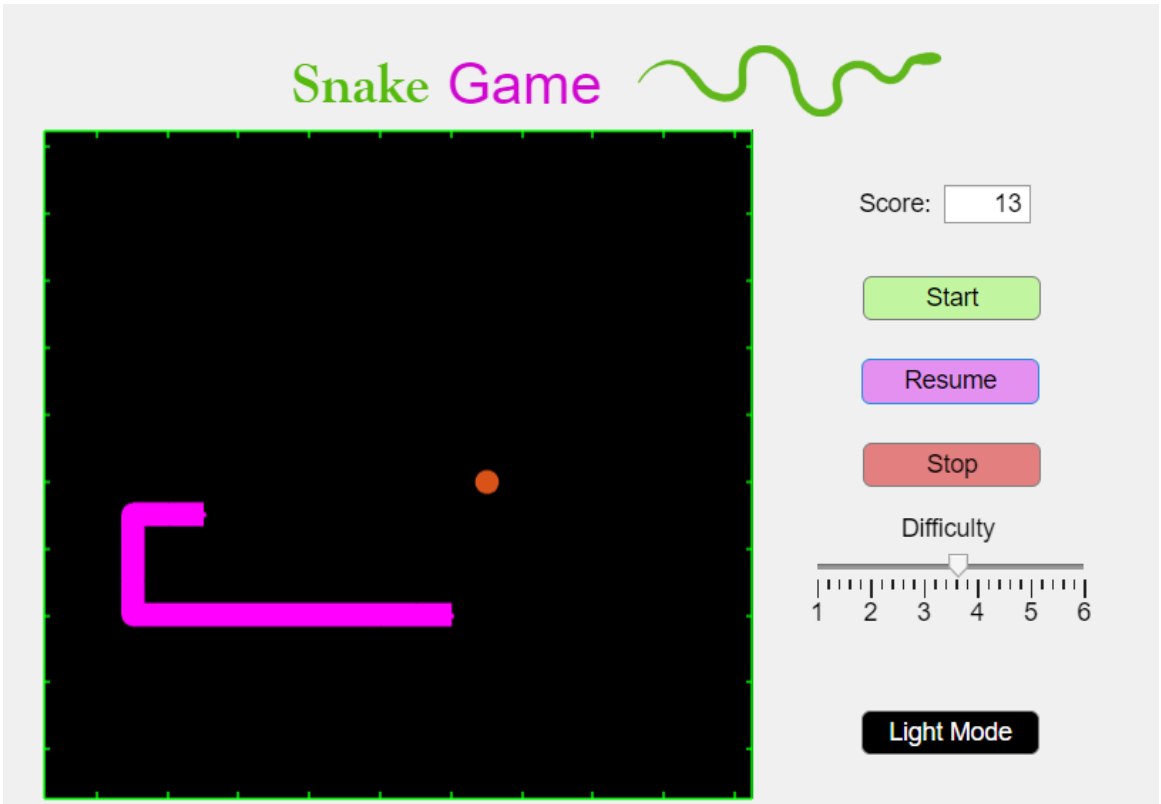
```
% Button pushed function: DarkModeButton
        function DarkModeButtonPushed(app, event)
            app.isDarkMode = ~app.isDarkMode; % Toggle dark mode

            if app.isDarkMode
                % Set colors for dark mode
                snakeColor = [1, 0, 1]; % magenta snake
                foodColor = [1, 0, 0]; % red food
                gridColor = [0, 0, 0]; % black grid
                app.DarkModeButton.Text = 'Light Mode'; % update button text
for when in dark mode
            else
                % Set colors for normal mode
                snakeColor = [0, 1, 0]; % green snake
                foodColor = [1, 0, 0]; % red food
                gridColor = [1, 1, 1]; % white grid
                app.DarkModeButton.Text = 'Dark Mode'; % update button text for
when in light mode
            end

            % Update the grid color
            set(app.UIAxes, 'Color', gridColor);

            % Update the snake color
            if isgraphics(app.snakeHandle)
                set(app.snakeHandle, 'Color', snakeColor);
            end

            % Update the food color
            if isgraphics(app.foodHandle)
                set(app.foodHandle, 'MarkerFaceColor', foodColor);
            end
        end
```

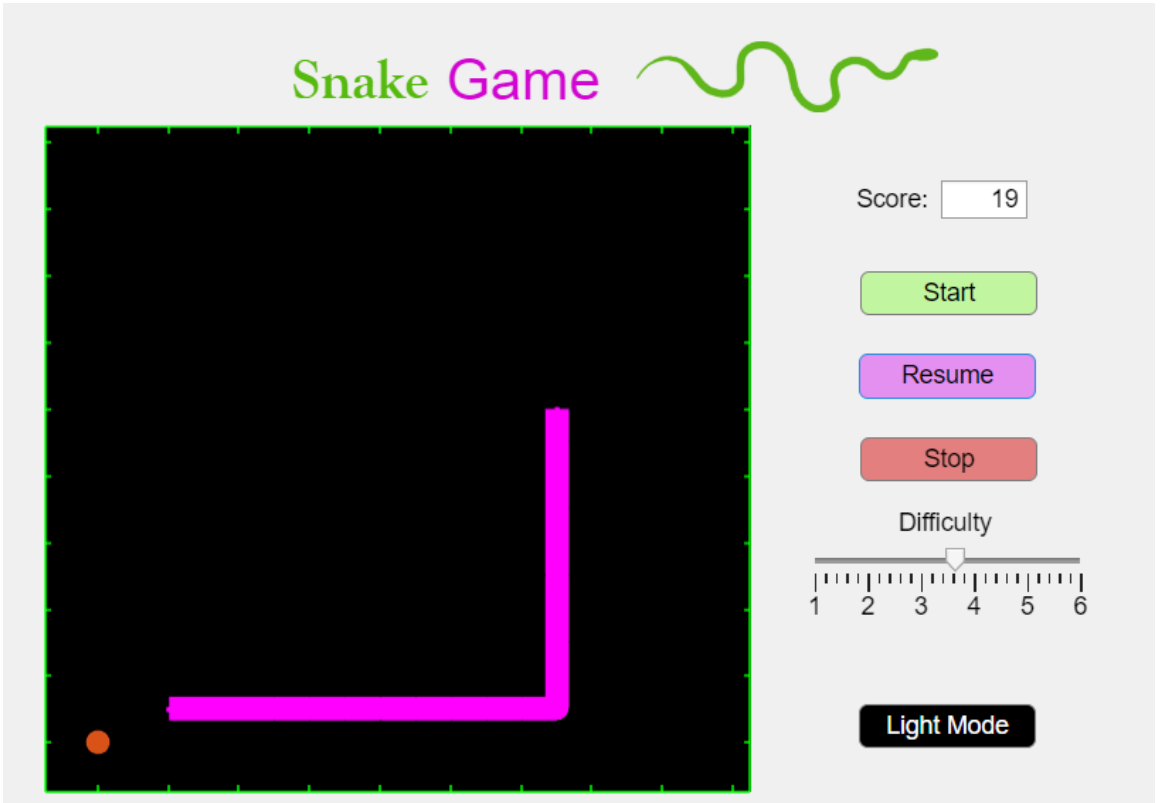


Game play

Here I will play a few games and add screenshots of the actual gameplay and functionalities so that you can observe the score increasing and the dark mode being used.




Snake Game
Score: 13
Start
Resume
Stop
Difficulty
1 2 3 4 5 6
Light Mode


Snake Game
Score: 19
Start
Resume
Stop
Difficulty
1 2 3 4 5 6
Light Mode

References

https://es.mathworks.com/discovery/matlab-gui.html

https://es.mathworks.com/help/matlab/ref/msgbox.html

https://es.mathworks.com/help/matlab/ref/pause.html

https://www.youtube.com/watch?v=wL9yGb86Q2g

https://www.youtube.com/watch?v=09WC5yIHRmQ&list=LL&index=2

https://www.youtube.com/watch?v=dSJsZYwZL4Q

https://es.mathworks.com/matlabcentral/answers/461166-how-to-use-windowkeypressfcn