BEYKOZ ÜNİVERSİTESİ

Term Project Assignment

CME0437

Software Engineering Processes

2023-2024

Lecturer : Alp SARDAĞ

Student ID: Elif Nur Aslıhan Celepoğlu - 1904010023

Contents

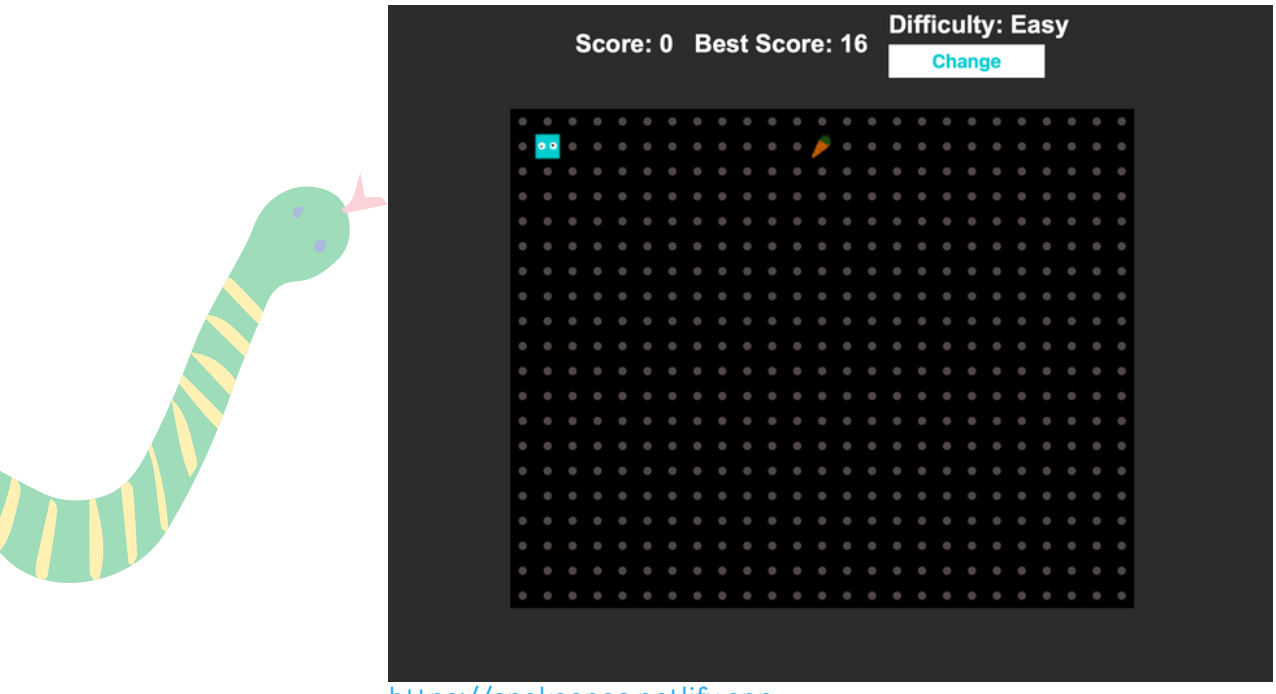
Introduction

This project is an example of software development aiming to modernize the classic Snake game and enhance code quality using **design patterns.**
The game involves the basic Snake mechanics, where a simple snake eats food on the screen to grow in size. However, behind this simple game, **there is a set of features designed and implemented in accordance with software engineering principles.**

Toolbox

The tools used in this project are listed below.

- Javascript(Programming Language)
- html and css (Web Technologies)
- HTML Canvas(Drawing and Animation)
- SVG (Graphics and Visual Assets)
- Web Storage API (LocalStorage)(Data Storage)
- requestAnimationFrame()(Game Loop and Timing)
- Web Audio API(Sound and Sound Effects)
- Console API(Debugging and Error Tracking)



https://snakeenac.netlify.app

Hint: https://github.com/elif1906/Snake_Game_design_pattern/tree/main

Functional Requirements

Game Initialization:

- The game should initialize with the snake at a default position and size.
- The initial score should be set to zero.

Player Controls:

- The player should be able to control the snake using keyboard inputs (arrow keys or W, A, S, D).
- The snake should respond to control inputs and move accordingly.

Game Mechanics:

- The snake should be able to consume food items.
- Upon consuming food, the snake's length should increase, and the player's score should be updated.
- The game should end if the snake collides with the game border or itself.
- The player should have the option to pause and resume the game.
- The player should be able to change the game difficulty (Easy/Hard).

Score Tracking:

- The game should track the player's score.
- The best score should be stored and displayed to the player.
- The player should have the ability to view the best score.

User Interface:

- The game should have a responsive and user-friendly interface.
- Important information such as the current score, best score, and game difficulty should be displayed to the player.

Nonfunctional Requirements

Performance:

- Smooth gameplay with minimal lag.
- Quick response time to user inputs.

Scalability:

- Adaptability to different screen sizes.

Compatibility:

- Compatibility with popular web browsers (Chrome, Firefox, Safari).
- Platform independence for various devices.

UML Sequence Diagram

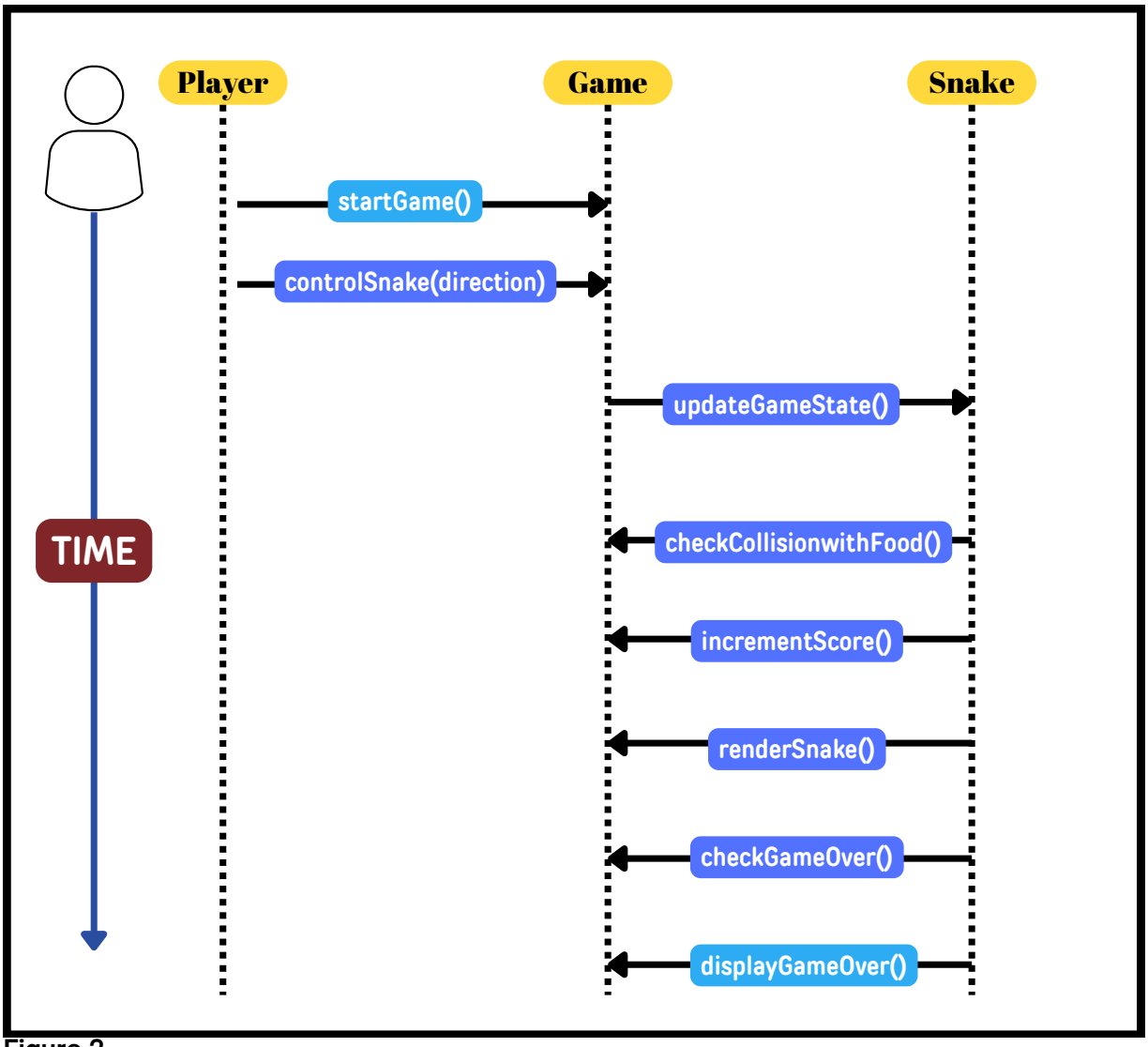


Figure 2

1. The sequence starts with the Player calling the **startGame()** method.
2. The Player sends a message to the Game object to start the game.
3. The Game object controls the Snake based on user input by calling **controlSnake(direction).**
4. The Game object updates the game state by calling **updateGameState().**
5. The Game object checks if the snake collides with food by calling **checkCollisionWithFood().**
6. If a collision with food is detected, the Game object increments the score by calling **incrementScore()**.
7. The Snake object is rendered on the game screen by calling **renderSnake()**.
8. The Game object checks if the game is over by calling **checkGameOver().**
9. If the game is over, the Game object displays the game-over message by calling **displayGameOver().**

Design Patterns

Singleton Pattern

The **audioNames** array contains instances of the Audio class that are created once for different sound elements and then reused. This allows for more efficient resource utilization and centralized management of sound elements.

```
//Singleton Pattern
const audioNames = [
    audioEat = new Audio(),
    audioTurn = new Audio(),
    audioDead = new Audio(),
    audioHit = new Audio(),
];
for ( let i = 0; i < audio.length; i++ ) {
    audioNames[i].src = audio[i];
}
```

Factory Pattern

The **snakeImages array** creates instances of the Image object representing different snake skins. The factory pattern is used to create and easily manage multiple similar objects.

```
//Factory Pattern
const snakeImages = [
    new Image(),
];
for ( let i = 0; i < snakeImages.length; i++ ) {
    snakeImages[i].src = snakeSkins[i];
}
```

Observer Pattern

The functions triggered when the user presses keyboard keys, through **document.addEventListener,** are associated with the observer pattern. It is used to listen to user interactions and perform appropriate actions.

```
//Observer Pattern
document.addEventListener('keydown', (e) => {
    if ( e.keyCode == 87 || e.keyCode == 38 ) { // W (up) or arrow up
        turnUp();
    }
```

Strategy Pattern

Different drawing and game mechanics are applied based on the difficulty level determined by the **diff variable.** This situation resembles the strategy pattern, where different algorithms represent the same functionality.

```
    //Strategy Pattern
    if (diff === 'Hard') {
        ctx.strokeStyle = '#f00';
        ctx.lineWidth = 5;
        ctx.strokeRect(0, 0, canvas.width, canvas.height);
        drawBomb();
    };
}
```