A

Learning project-I Report

On

“Gaming Platform on Pac-Man”

Submitted in partial fulfilment of

The requirements for the 3rd Semistar Sessional Examination of

BACHELOR OF TECHNOLOGY

IN

**COMPUTER SCIENCE & ENGENEERING**

BY

***NAME***

***ANSUMAN MAHAPATRA***

***BISWAJEET PATRA***

***TAPAN KUMAR SAHOO***

Registration No.

23UG010605

23UG010614

23UG010632

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

GIET UNIVERSITY , Gunupur

2023-24

ACKNOWLEDGEMENT

We express our sincere gratitude to Nalinikanta Panda of Computer Science and Engineering

for giving us an opportunity to accomplish the project. Without his active support and guidance, this project report has not been successfully completed.

We also thank our class teacher Mr. Bani Prasad Nayak for his constant support and guidance provided during this project period.

We also thank Mr. Bhavani Sankar Panda, Project Coordinator, Dr. Sachikanta Dash, Head of the Department of Computer Science and Engineering, Prof. Dr. Sanjay Kumar Kunar, Dy. Dean, Computational Science, SOET for their consistent support, guidance and help.

Thanks to all my team members Aman Kumar Sahu and Rahul Gouda for going strength with me without them, this learning project is incomplete.

ANSUMAN MAHAPATRA(23UG010605)

BISWAJEET PATRA(23UG010614)

TAPAN KUMAR SAHOO(23UG010632)

DECLARATION

We hereby declare that project titled **“BLOOD BANK MANAGEMENT SYSTEM”** is a bonafide original record done by us at GIET University, Gunupur towards the partial fulfillment of the requirement of Learning Project – 01, 3rd Semester, School of Engineering and Technology, Department of Computer Science and Engineering during the period of 2023–2024 in GIET University, Gunupur, also we state that this project has not been submitted anywhere in the partial fulfillment for any degree of this or any other University.

ANSUMAN MAHAPATRA(23UG010605)

BISWAJEET PATRA(23UG010614)

TAPAN KUMAR SAHOO(23UG010632)

INDEX

CHAPTER-I

ABSTRACT

INTRODUCTION

CHAPTER-II

WORKDONE IN RELATED AREA

CHAPTER-III

SYSTEM ANALYSIS

CHAPTER-IV

SYSTEM DESIGN AND APPLICATION

CHAPTER-V

CODE OF THE GAMING PLATFORM

CHAPTER-VI

TESTING

CHAPTER-VII

CONCLUSION AND LIMITATION

CHAPTER-VIII

REFERENCE

***CHAPTER-I***

***ABSTRACT***

This project presents an interactive, web-based Pac-Man game developed as a part of my learning journey at GIET University. Designed as an engaging and visually appealing gaming experience, this Pac-Man game is created using foundational web development technologies—HTML, CSS, and JavaScript. The project’s main objective was to implement a functional and dynamic game that brings the classic arcade experience of Pac-Man to the web, allowing users to enjoy it directly from their browsers without needing any additional downloads or installations.

The game layout features a grid-style map representing the familiar Pac-Man maze. The walls of the maze, Pac-Man character, ghosts, and other game elements are represented through custom-designed images. HTML structures the webpage, defining the game interface and necessary elements, while CSS adds style, enhancing the visual appeal of the maze, characters, and background to mimic the authentic arcade feel. JavaScript powers the interactivity and game logic, controlling Pac-Man's movements, ghost AI, and collision detection, creating a smooth and responsive user experience.

***KEY FEATURES :***

This Pac-Man game is a web-based recreation of the classic arcade game, designed using HTML, CSS, JavaScript, and various custom images. It offers a straightforward yet engaging gaming experience where players navigate Pac-Man through a maze, collecting pellets, avoiding ghosts, and using a special super pellet for strategic advantages. The game features one Pac-Man character, three ghost enemies, and a unique super pellet that adds an exciting twist to the classic gameplay.

1. **Maze and Game Environment**
   * The game maze is structured with a grid design reminiscent of the classic arcade layout. The maze’s walls are represented using custom images, and the layout guides Pac-Man’s path and limits his movement.
   * The game area is designed to be visually engaging, with a background and elements styled to give an authentic arcade feel, combining nostalgic elements with a fresh look.
2. **Pac-Man Character and Movements**
   * Pac-Man is the main character controlled by the player, using arrow keys for navigation. Pac-Man moves around the maze, collecting pellets while avoiding enemies.
   * The character animation mimics the iconic “mouth-chomping” motion, achieved through JavaScript animations and CSS styling, adding a touch of realism and fun.
3. **Ghost Enemies with AI Behavior**
   * There are three ghost enemies in the maze, each with unique movement patterns. The ghosts are programmed with basic AI to make the game challenging and engaging.
   * The AI behavior includes random movement patterns, along with periods of aggression where they actively chase Pac-Man, creating tension and requiring strategic play from the user.
4. **Pellets and Super Pellet**
   * Regular pellets are scattered throughout the maze, and Pac-Man scores points as he collects each one. The player’s objective is to collect all pellets in the maze without getting caught by a ghost.
   * A special "super pellet" is strategically placed within the maze. When Pac-Man collects it, he gains temporary powers, allowing him to chase and capture ghosts for additional points. This super pellet adds an exciting element of strategy, allowing the player to turn the tables on the ghosts temporarily.
5. **Score Tracking and High Scores**
   * The game includes a scoring system that updates in real-time as Pac-Man collects pellets and captures ghosts after consuming a super pellet.
   * An optional feature to display high scores can be added, enabling players to compete for top scores in subsequent rounds.
6. **Collision Detection and Game Mechanics**
   * JavaScript handles the core mechanics, including collision detection between Pac-Man and ghosts, as well as Pac-Man’s interactions with the walls, pellets, and super pellet.
   * The game’s mechanics ensure smooth gameplay and responsive control for the player, providing an enjoyable experience and allowing for challenging gameplay.

**Additional Game Features and Functionalities**

1. **Sound Effects and Game Over Screen**
   * Sound effects can be added to enhance the immersive experience. Sounds for pellet collection, ghost encounters, and game events such as the “super pellet” activation improve the gameplay experience.
   * A “Game Over” screen is displayed if Pac-Man is caught by a ghost, prompting the player to either retry the game or exit.
2. **Game Animation and Visuals**
   * Animations are implemented with CSS and JavaScript, providing fluid movement for both Pac-Man and the ghosts. Pac-Man’s iconic chomping action is designed using CSS animations, while ghost movements are animated to reflect their personalities and styles.
   * Ghosts have different colors and animations to make each enemy distinctive, helping players easily recognize them during gameplay.
3. **Dynamic Difficulty Levels (Optional)**
   * To keep players engaged, the game can be programmed with increasing difficulty as they progress. For example, the speed and aggression of the ghosts could increase over time or based on the number of pellets collected.
   * With each level, new mazes or challenges could be introduced, adding variety to the game and allowing players to experience a range of difficulty levels.
4. **Mobile and Desktop Compatibility**
   * The game is designed to be responsive, allowing it to function on both mobile devices and desktop computers. This ensures a wider audience can access and enjoy the game, regardless of device.
5. **Enhanced User Interface**
   * The user interface (UI) elements are styled to provide a clean, easy-to-understand layout. The game controls are intuitive, with arrow keys for movement, and clear indicators for scores, levels, and lives.
   * UI features include a scoring display at the top of the game, showing the player's current score and any special indicators, such as the super pellet's effects.
6. **Code Structure and Modularity**
   * The code for this game is organized in a modular structure, making it easy to read, understand, and expand upon. Each functionality—such as player movement, ghost AI, collision detection, and scoring—is organized into separate functions.
   * The modular code structure allows for future enhancements, such as adding more characters, maze designs, or difficulty settings without needing to rewrite the entire codebase.
7. **Learning Experience and Technological Skills**
   * This project served as a learning platform, helping me apply the concepts of HTML, CSS, JavaScript, and web design in a practical, enjoyable way. By developing this game, I learned about DOM manipulation, event handling, and game logic, as well as CSS animations and styling techniques.
   * Additionally, this project introduced me to essential software engineering principles, including modular coding, debugging, and iterative development, helping me build a strong foundation for future projects.

This online Pac-Man game project exemplifies the application of core web development skills in a fun and interactive format. By combining custom animations, JavaScript-based game logic, and responsive design, the game brings the classic Pac-Man experience to modern web browsers. The project demonstrates my ability to design, develop, and implement a fully functional game, and it highlights the skills I have gained in web development and basic game design. The inclusion of unique elements like the super pellet, three distinct ghost enemies, and animated visuals adds complexity and excitement, making this project a significant milestone in my journey as a web developer.

***CHAPTER-II***

**WORKDONE IN RELATED AREA**

Our Pac-Man game project, developed over a span of 25 days, involved three distinct phases that allowed us to systematically build, test, and refine each aspect of the game. We began by creating the maze structure and its surroundings, followed by integrating the core game characters Pac-Man and the ghosts and concluded with adding the gameplay elements, such as pellets and power pellets. Through these phases, we were able to bring our initial concept to life. The team was led by Ansuman Mahapatra, who completed 50% of the work, with Biswajit Patra contributing 30% and Tapan Kumar Sahoo providing the remaining 20%.

**Phase 1: Maze and Surroundings Development**

The first phase focused on developing the fundamental structure of the game the maze and its surroundings. This phase required significant attention to detail and precision, as the maze is central to Pac-Man’s gameplay experience, dictating Pac-Man's movement path, walls, and escape routes.

**Designing the Maze Layout**

We began by sketching the layout of the maze on paper, using the classic Pac-Man arcade game as inspiration while adding our unique touch. The layout had to ensure that Pac-Man’s movement paths and ghost routes created the right balance of challenge and playability. We used HTML to create the grid-based layout of the maze and CSS to style its walls, paths, and background elements. Careful adjustments to dimensions were necessary to maintain consistency across different screen sizes.

**Adding Visual Surroundings**

Once the maze structure was in place, we turned our attention to the visual surroundings, which involved choosing appropriate colors, patterns, and textures to give the game a familiar yet fresh look. We selected color schemes and images that enhanced the aesthetic appeal of the game, ensuring that it felt immersive and true to the classic arcade experience. Each component of the surroundings, such as wall borders and background, was coded with CSS styles to ensure consistency.

**Testing and Adjustments**

After setting up the maze and surroundings, we conducted initial tests to evaluate the overall layout and styling. Small adjustments were made to the maze width and pathways to ensure smooth movement. The maze layout was saved as a template, which served as a base for the next phases. This initial stage took approximately eight days and was crucial to the development process, as it set the foundation for the remaining features.

**Phase 2: Character Integration – Adding Pac-Man and Ghosts**

With the maze template in place, we proceeded to integrate the primary characters—Pac-Man and the three ghosts. This phase presented challenges in ensuring accurate positioning, animation, and interaction between Pac-Man and the ghosts. Implementing the ghost AI and Pac-Man’s controls required several iterations to perfect.

**Implementing Pac-Man’s Character and Movement**

Our first task was to introduce Pac-Man’s character to the maze. Using JavaScript, we programmed Pac-Man’s movement to respond to arrow key inputs, enabling the player to navigate through the maze smoothly. CSS was used to animate Pac-Man’s chomping motion, which involved coordinating the opening and closing of Pac-Man's mouth in sync with his movements. The coding here required precise timing to ensure that Pac-Man's animation appeared fluid and realistic.

**Adding Ghosts and Programming Basic AI**

Once Pac-Man was moving within the maze, we added three ghost enemies, each with distinct colors to differentiate them. The ghosts were programmed with basic AI logic to create movement patterns within the maze. Initially, the ghosts moved randomly, but we refined their behavior to simulate periods of active pursuit, creating an element of tension and excitement for the player.

**Challenges and Iterations**

Throughout this phase, we encountered several challenges. The positioning of Pac-Man and the ghosts needed constant adjustment, as characters often collided or got stuck in maze walls. We conducted eleven trials, each time troubleshooting issues related to positioning, collision detection, and animation timing. After many adjustments and much collaboration, we achieved the desired behavior in the twelveth trial. This phase took approximately ten days and required intense focus and collaboration, as we repeatedly adjusted code and tested functionality.

**Phase 3: Integrating Pellets and Power Pellets**

The final phase was adding gameplay elements, specifically the pellets and power pellets that Pac-Man would collect to score points. These elements were crucial to creating a rewarding player experience, as they added both scoring and strategy components to the game.

**Adding Standard Pellets**

Standard pellets were placed throughout the maze, serving as Pac-Man’s primary objective. Using JavaScript, we created functions to detect when Pac-Man collected a pellet and updated the player’s score in real-time. The pellets were spaced evenly to encourage Pac-Man to navigate the entire maze, increasing player engagement and creating a more immersive experience.

**Introducing the Super Pellet**

We then added the super pellet, a unique power-up that temporarily changes the game dynamics. When Pac-Man collects the super pellet, he gains the ability to chase and “eat” ghosts for additional points. This feature required additional coding to trigger a “fear” response in the ghosts, causing them to flee from Pac-Man temporarily. Programming the super pellet’s effect was challenging, as it required coordinating animations, changing ghost behavior, and updating the score tracking simultaneously.

**Testing and Final Adjustments**

Once all the elements were in place, we conducted thorough testing to identify any lingering bugs. We focused on ensuring that Pac-Man’s interactions with pellets and power pellets triggered the correct responses and score updates. Minor adjustments were made to optimize performance, including code cleanup and refactoring.

This final phase, taking about seven days, brought together all the elements, making the game fully playable and ready for presentation.

**Team Contributions and Reflection**

The project was a collaborative effort involving three team members:

* **Ansuman (Ansu) Mahapatra** contributed 50% of the work, focusing on character animation, ghost AI programming, and overall project coordination.
* **Biswajit Patra** contributed 30%, primarily handling the maze design, CSS styling, and score tracking.
* **Tapan Kumar Sahoo** contributed the remaining 20%, assisting with testing and debugging in each phase.

Each phase presented unique challenges, requiring trial, error, and numerous code adjustments. Despite these obstacles, our combined efforts enabled us to complete the project successfully, creating a fully functional and engaging Pac-Man game. This experience allowed each of us to develop our technical skills and deepen our understanding of the game development process. The teamwork and perseverance demonstrated throughout the project were essential to our success, highlighting the importance of collaboration and dedication in achieving complex goals.

***CHAPTER-III***

***SYSTEM ANALYSIS AND REQUIREMENTS***

The development of the Pac-Man game project required a structured approach to system analysis and requirements gathering. This process allowed us to outline the technical specifications, gameplay features, and system dependencies necessary to create an interactive and user-friendly gaming experience. The project primarily targets players using Windows operating systems and emphasizes simplicity and accessibility, making it an ideal game for both casual play and as an educational tool for introducing basic gaming concepts.

**1. System Requirements**

System requirements describe the essential technical and operational specifications for the Pac-Man game. These ensure that users can run the game smoothly on compatible systems without needing additional hardware.

**Supported Operating Systems**

The Pac-Man game is optimized to run on the following Windows operating systems:

* **Windows 8**
* **Windows 10** and **Windows 10 Pro**
* **Windows 11**

This support range was selected based on the wide usage of these operating systems, ensuring accessibility to a broad audience. Each of these systems provides compatibility with web-based applications, which is essential for running the game, developed using HTML, CSS, and JavaScript.

**Hardware Requirements**

The game was designed to be lightweight, requiring minimal hardware resources:

* **No Graphics Card Required**: The game’s graphics are simple and do not require a dedicated graphics card, making it accessible to users with basic system configurations.
* **Processor and RAM**: Any modern processor and at least 2GB of RAM are recommended to ensure smooth performance, although the game can run on lower specifications given its low resource demands.

**Software Requirements**

The game is playable through modern web browsers (such as Chrome, Firefox, Edge) installed on the Windows operating systems mentioned. This allows for easy access without needing additional gaming software or installations.

**2. Functional Requirements**

Functional requirements specify the core features and interactions that are essential to the game’s design and user experience. These features are fundamental to delivering an engaging and cohesive gameplay experience.

* **Maze Structure**: The game includes a grid-based maze layout that serves as Pac-Man’s play area, with walls, paths, and junctions. The structure of the maze must guide Pac-Man’s movement while creating challenging paths that require strategic planning.
* **Character Movement**: Pac-Man’s character should be controlled by the user, responding to arrow keys for smooth navigation throughout the maze. The movement should be intuitive and responsive to player input, allowing for an enjoyable gameplay experience.
* **Enemy Ghost AI**: The game includes three ghosts with basic AI to create different movement patterns, such as random wandering and direct pursuit. This adds complexity to the game, making it challenging for players to evade capture while collecting pellets.
* **Collision Detection**: Collision detection is critical for handling interactions between Pac-Man, walls, ghosts, and pellets. When Pac-Man collides with walls, he should stop, and when colliding with ghosts, the game should detect it as a loss of life. Pellet collection should trigger score updates.
* **Pellets and Power Pellets**: Standard pellets are spread across the maze, and collecting them adds points to the player’s score. Power pellets provide a temporary power-up that allows Pac-Man to chase and capture ghosts, adding a strategic element to the game.
* **Score Tracking**: The game should include a real-time score-tracking system that displays the player’s progress and achievements as they collect pellets and capture ghosts.
* **Game Restart and Level End**: The game should reset upon loss of all lives or upon Pac-Man clearing the maze, creating a complete gameplay loop.

**3. Non-Functional Requirements**

Non-functional requirements focus on the quality, performance, and usability aspects of the game. These requirements ensure that the game provides a stable and engaging experience across compatible devices.

* **Performance**: The game must load quickly and maintain consistent frame rates, ensuring that animations, movements, and interactions happen smoothly without lag or delays.
* **Usability and Accessibility**: Controls should be intuitive and simple to operate, requiring only arrow key input. The game’s interface should display the score clearly, and visual indicators should provide feedback for events, such as Pac-Man collecting pellets or losing a life.
* **Reliability**: The game must be reliable, able to handle long play sessions without crashing or freezing. The score tracking and collision systems should work consistently, ensuring data integrity and preventing errors.
* **Compatibility**: The game is designed to run within modern web browsers on the specified Windows operating systems, with CSS and JavaScript handling all visual and interactive elements. This eliminates the need for additional hardware or software dependencies.
* **Maintainability**: Code should be modular and well-documented to facilitate future updates, bug fixes, or expansions. This includes structuring code for the maze, character movement, score tracking, and collision detection separately for easier troubleshooting and improvements.

**4. System Modeling**

System modeling provides a visual and conceptual representation of the Pac-Man game’s functionality, gameplay flow, and interactions between components. The modeling of this system includes several types of diagrams:

* **Use Case Diagram**: Outlines the player’s interactions, such as starting the game, navigating the maze, collecting pellets, using power-ups, and scoring points.
* **Activity Diagram**: Depicts the sequence of actions, such as beginning a game, controlling Pac-Man, encountering ghosts, and either winning or losing.
* **Data Flow Diagram (DFD)**: Shows how data moves within the game, such as score tracking and the collision detection process, which updates as Pac-Man collects pellets and interacts with ghosts.

**5. Feasibility Studies**

Conducting feasibility studies helps ensure that the project can be successfully completed within given constraints:

* **Technical Feasibility**: Given that the game uses HTML, CSS, and JavaScript, it is technically feasible, especially with no requirement for advanced hardware. These programming languages are well-suited for creating a lightweight, web-based game.
* **Operational Feasibility**: The project’s game design, including maze structure, ghost AI, and power-ups, aligns with user expectations and is achievable within the Windows environment specified.
* **Economic Feasibility**: This project is primarily educational, with low cost requirements limited to basic development tools and resources for testing. No extensive financial resources are necessary, as the game is web-based and does not require additional software or equipment.

**HARDWARE REQUIREMENTS:**

* I. **Processor**: Multi-core processor (e.g., Intel i3/i5, AMD Ryzen).
* II. **RAM**: Minimum 2 GB for basic gameplay; 4 GB recommended for optimal performance.
* III. **Storage**: Basic HDD or SSD with at least 500 MB free space for browser cache.

**SOFTWARE REQUIREMENTS:**

* I. **Operating System**: Windows 8, 10, 10 Pro, or 11.
* II. **Browser**: Modern web browser (e.g., Chrome, Firefox, Edge).
* III. **Additional Software**: None required; runs as a browser-based game without additional installations.

***CHAPTER-IV***

***SYSTEM DESIGN AND SPECIFICATION***

The system design phase focuses on planning the architecture, components, and interactions of the Pac-Man game to create a structured, interactive, and efficient gaming experience. This section provides an overview of the game’s design, detailing both visual and functional elements, along with the internal logic governing player interactions and gameplay flow.

**1. Overview of System Architecture**

The architecture of the Pac-Man game is web-based, utilizing **HTML, CSS, and JavaScript**. This structure supports lightweight, cross-platform gameplay directly accessible through a web browser. The system is designed to be modular, enabling easy management and updates to individual components like the maze layout, character movement, and score tracking.

**Key Components of the Architecture:**

* **Frontend Interface**: Created using HTML and CSS, defining the visual layout, color scheme, and overall look of the game, including the maze structure and character designs.
* **Game Logic (JavaScript)**: JavaScript handles all interactive elements, including character movement, collision detection, score tracking, and game controls.
* **Data Management**: All score and status updates are handled locally within the session, without persistent storage, ensuring lightweight functionality suitable for an educational project.

**2. Game Components**

Each component of the game is carefully designed to provide an engaging and consistent gameplay experience.

**a. Maze Structure**

* The maze is designed as a grid, with paths and walls arranged to form a challenging layout.
* CSS is used to style the walls, paths, and pellets, while JavaScript manages the interactions within this grid.

**b. Characters**

* **Pac-Man**: The player-controlled character moves through the maze collecting pellets. The character's movement responds to arrow key input, allowing for smooth and intuitive controls.
* **Ghosts**: Three enemy characters move around the maze with basic AI that either follows random patterns or pursues Pac-Man when certain conditions are met.
* **Power-Ups (Super Pellet)**: The super pellet provides a temporary boost, allowing Pac-Man to chase and capture ghosts for additional points. JavaScript handles the temporary power-up effect, changing ghost behavior when Pac-Man collects a super pellet.

**c. Pellets and Power Pellets**

* **Regular Pellets**: Spread throughout the maze and collected by Pac-Man to earn points.
* **Power Pellets**: A few special pellets provide a power-up effect, allowing Pac-Man to chase ghosts for additional points.

**d. Score and Lives Display**

* A simple scoreboard tracks the player's score, and a life counter displays the number of lives remaining.
* The score and lives update in real-time as Pac-Man collects pellets or encounters ghosts.

**3. Functional Specifications**

The game’s functionality is determined by several core specifications that govern interactions, behaviors, and user feedback.

* **Player Controls**: Arrow keys control Pac-Man’s movement within the maze.
* **Collision Detection**: JavaScript handles collision detection between Pac-Man, ghosts, walls, and pellets, ensuring that characters respond appropriately to their surroundings.
* **Enemy AI**: Ghosts exhibit basic AI behaviors, either following random movement or, when activated by the super pellet, fleeing Pac-Man.
* **Score Tracking**: Scores update instantly as Pac-Man collects pellets or captures ghosts, providing real-time feedback for the player.
* **Game Over Conditions**: The game ends when the player runs out of lives, resetting the maze for a new attempt.

**4. Non-Functional Specifications**

In addition to functionality, non-functional specifications define the game's quality and usability attributes:

* **Performance**: The game should load quickly and run without lag, even on basic systems without dedicated graphics hardware.
* **Usability**: Controls are kept simple and intuitive, requiring only arrow keys for navigation.
* **Reliability**: The system should be stable, with consistent collision detection, movement controls, and score updates.
* **Compatibility**: The game is compatible with Windows 8, 10, 10 Pro, and 11 operating systems and is accessible via modern browsers (Chrome, Firefox, Edge).
* **Accessibility**: The design is lightweight and accessible on systems without advanced hardware, ensuring a broad user base.

**5. Interaction Flow**

The interaction flow in the Pac-Man game involves several key states that define the player’s experience, from starting the game to navigating the maze and reaching game-over conditions.

1. **Game Start**: The player loads the game in a browser and is presented with the initial maze.
2. **Player Movement**: Using arrow keys, the player navigates Pac-Man through the maze to collect pellets while avoiding ghosts.
3. **Ghost Encounters**: If Pac-Man collides with a ghost, he loses a life; if he has collected a super pellet, he can temporarily chase and capture the ghost for extra points.
4. **Score Updates**: Each pellet and ghost capture adds to the player’s score, which updates in real-time.

***CHAPTER-V***

***CODE OF THE GAMING PLATFORM***

**• HTML AND CSS CODE**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Pac-Man</title>

<style>

@keyframes backgroundAnimation {

0% {

background: linear-gradient(0deg, rgb(17, 51, 161), rgb(136, 34, 195));

}

25% {

background: linear-gradient(0deg, rgb(255, 94, 0), rgb(255, 195, 0));

}

50% {

background: linear-gradient(0deg, rgb(0, 128, 255), rgb(0, 255, 255));

}

75% {

background: linear-gradient(0deg, rgb(255, 0, 128), rgb(128, 0, 255));

}

100% {

background: linear-gradient(0deg, rgb(17, 51, 161), rgb(136, 34, 195));

}

}

#gameCanvas {

box-shadow: 10px 10px 20px black;

}

h1 {

text-align: center;

font-family: comic sans MS;

color: lightgray;

}

body {

display: flex;

justify-content: center;

align-items: center;

flex-direction: column;

height: 100vh;

margin: 0;

overflow: hidden;

animation: backgroundAnimation 10s infinite alternate;

}

#gameContainer {

display: flex;

flex-direction: column;

align-items: center;

margin-top: 20px;

}

.footer-text-container {

margin-top: 20px;

padding: 15px;

border-radius: 10px;

text-align: right;

font-family: Arial, sans-serif;

font-size: 14px;

background: linear-gradient(90deg, rgba(255, 0, 128, 1), rgba(0, 204, 255, 1));

color: white;

box-shadow: 5px 5px 15px rgba(0, 0, 0, 0.3);

}

button {

background-color: green;

position: absolute;

top: 490px;

right: 730px;

padding: 10px 20px;

font-size: 16px;

cursor: pointer;

}

</style>

</head>

<body>

<div>

<h1>Pac-Man</h1>

<canvas id="gameCanvas"></canvas>

</div>

<div id="gameContainer">

<br><br>

<h1>Enjoy Game</h1>

<div class="footer-text-container">

This game was developed by group 21 <br>

1. Ansuman Mahapatra <br>

2. Biswajeet Patra <br>

3. Tapan Kumar Sahoo

</div>

</div>

<script src="src holder/Game.js" type="module"></script>

<button onclick="restartPage()">Restart</button>

<script>

function restartPage() {

location.reload(); // This will reload the current page

}

</script>

</body>

</html**>**

We divided the java script code into 5 sub parts . They are :-

1.Enemy.js

2.Game.js

3.MovingDirection.js

4.Pacman.js

5.TileMap.js

We do such that thing because if an error occure in the game then we go to that file and easily solve it and it consumes our time .

***1.Enemy.js***

import MovingDirection from "./MovingDirection.js";

export default class Enemy {

  constructor(x, y, tileSize, velocity, tileMap) {

    this.x = x;

    this.y = y;

    this.tileSize = tileSize;

    this.velocity = velocity;

    this.tileMap = tileMap;

    this.#loadImages();

    this.movingDirection = Math.floor(

      Math.random() \* Object.keys(MovingDirection).length

    );

    this.directionTimerDefault = this.#random(10, 25);

    this.directionTimer = this.directionTimerDefault;

    this.scaredAboutToExpireTimerDefault = 10;

    this.scaredAboutToExpireTimer = this.scaredAboutToExpireTimerDefault;

  }

  draw(ctx, pause, pacman) {

    if (!pause) {

      this.#move();

      this.#changeDirection();

    }

    this.#setImage(ctx, pacman);

  }

  collideWith(pacman) {

    const size = this.tileSize / 2;

    if (

      this.x < pacman.x + size &&

      this.x + size > pacman.x &&

      this.y < pacman.y + size &&

      this.y + size > pacman.y

    ) {

      return true;

    } else {

      return false;

    }

  }

  #setImage(ctx, pacman) {

    if (pacman.powerDotActive) {

      this.#setImageWhenPowerDotIsActive(pacman);

    } else {

      this.image = this.normalGhost;

    }

    ctx.drawImage(this.image, this.x, this.y, this.tileSize, this.tileSize);

  }

  #setImageWhenPowerDotIsActive(pacman) {

    if (pacman.powerDotAboutToExpire) {

      this.scaredAboutToExpireTimer--;

      if (this.scaredAboutToExpireTimer === 0) {

        this.scaredAboutToExpireTimer = this.scaredAboutToExpireTimerDefault;

        if (this.image === this.scaredGhost) {

          this.image = this.scaredGhost2;

        } else {

          this.image = this.scaredGhost;

        }

      }

    } else {

      this.image = this.scaredGhost;

    }

  }

  #changeDirection() {

    this.directionTimer--;

    let newMoveDirection = null;

    if (this.directionTimer == 0) {

      this.directionTimer = this.directionTimerDefault;

      newMoveDirection = Math.floor(

        Math.random() \* Object.keys(MovingDirection).length

      );

    }

    if (newMoveDirection != null && this.movingDirection != newMoveDirection) {

      if (

        Number.isInteger(this.x / this.tileSize) &&

        Number.isInteger(this.y / this.tileSize)

      ) {

        if (

          !this.tileMap.didCollideWithEnvironment(

            this.x,

            this.y,

            newMoveDirection

          )

        ) {

          this.movingDirection = newMoveDirection;

        }

      }

    }

  }

  #move() {

    if (

      !this.tileMap.didCollideWithEnvironment(

        this.x,

        this.y,

        this.movingDirection

      )

    ) {

      switch (this.movingDirection) {

        case MovingDirection.up:

          this.y -= this.velocity;

          break;

        case MovingDirection.down:

          this.y += this.velocity;

          break;

        case MovingDirection.left:

          this.x -= this.velocity;

          break;

        case MovingDirection.right:

          this.x += this.velocity;

          break;

      }

    }

  }

  #random(min, max) {

    return Math.floor(Math.random() \* (max - min + 1)) + min;

  }

  #loadImages() {

    this.normalGhost = new Image();

    this.normalGhost.src = "images/ghost.png";

    this.scaredGhost = new Image();

    this.scaredGhost.src = "images/scaredGhost.png";

    this.scaredGhost2 = new Image();

    this.scaredGhost2.src = "images/scaredGhost2.png";

    this.image = this.normalGhost;

  }

}

***2.Game.js***

import TileMap from "./TileMap.js";

const tileSize = 32;

const velocity = 2;

const canvas = document.getElementById("gameCanvas");

const ctx = canvas.getContext("2d");

const tileMap = new TileMap(tileSize);

const pacman = tileMap.getPacman(velocity);

const enemies = tileMap.getEnemies(velocity);

let gameOver = false;

let gameWin = false;

const gameOverSound = new Audio("sounds/gameOver.wav");

const gameWinSound = new Audio("sounds/gameWin.wav");

function gameLoop() {

  tileMap.draw(ctx);

  drawGameEnd();

  pacman.draw(ctx, pause(), enemies);

  enemies.forEach((enemy) => enemy.draw(ctx, pause(), pacman));

  checkGameOver();

  checkGameWin();

}

function checkGameWin() {

  if (!gameWin) {

    gameWin = tileMap.didWin();

    if (gameWin) {

      gameWinSound.play();

    }

  }

}

function checkGameOver() {

  if (!gameOver) {

    gameOver = isGameOver();

    if (gameOver) {

      gameOverSound.play();

    }

  }

}

function isGameOver() {

  return enemies.some(

    (enemy) => !pacman.powerDotActive && enemy.collideWith(pacman)

  );

}

function pause() {

  return !pacman.madeFirstMove || gameOver || gameWin;

}

function drawGameEnd() {

  if (gameOver || gameWin) {

    let text = " You Win!";

    if (gameOver) {

      text = "Game Over";

    }

    ctx.fillStyle = "black";

    ctx.fillRect(0, canvas.height / 3.2, canvas.width, 80);

    ctx.font = "75px comic sans";

    const gradient = ctx.createLinearGradient(0, 0, canvas.width, 0);

    gradient.addColorStop("0", "magenta");

    gradient.addColorStop("0.5", "blue");

    gradient.addColorStop("1.0", "red");

    ctx.fillStyle = gradient;

    ctx.fillText(text, 10, canvas.height / 2);

  }

}

tileMap.setCanvasSize(canvas);

setInterval(gameLoop, 1000 / 75);

***3.MovingDirection.js***

const MovingDirection = {

  up: 0,

  down: 1,

  left: 2,

  right: 3,

};

export default MovingDirection;

***4.Pacman.js***

import MovingDirection from "./MovingDirection.js";

export default class Pacman {

  constructor(x, y, tileSize, velocity, tileMap) {

    this.x = x;

    this.y = y;

    this.tileSize = tileSize;

    this.velocity = velocity;

    this.tileMap = tileMap;

    this.currentMovingDirection = null;

    this.requestedMovingDirection = null;

    this.pacmanAnimationTimerDefault = 10;

    this.pacmanAnimationTimer = null;

    this.pacmanRotation = this.Rotation.right;

    this.wakaSound = new Audio("sounds/waka.wav");

    this.powerDotSound = new Audio("sounds/power\_dot.wav");

    this.powerDotActive = false;

    this.powerDotAboutToExpire = false;

    this.timers = [];

    this.eatGhostSound = new Audio("sounds/eat\_ghost.wav");

    this.madeFirstMove = false;

    document.addEventListener("keydown", this.#keydown);

    this.#loadPacmanImages();

  }

  Rotation = {

    right: 0,

    down: 1,

    left: 2,

    up: 3,

  };

  draw(ctx, pause, enemies) {

    if (!pause) {

      this.#move();

      this.#animate();

    }

    this.#eatDot();

    this.#eatPowerDot();

    this.#eatGhost(enemies);

    const size = this.tileSize / 2;

    ctx.save();

    ctx.translate(this.x + size, this.y + size);

    ctx.rotate((this.pacmanRotation \* 90 \* Math.PI) / 180);

    ctx.drawImage(

      this.pacmanImages[this.pacmanImageIndex],

      -size,

      -size,

      this.tileSize,

      this.tileSize

    );

    ctx.restore();

    // ctx.drawImage(

    //   this.pacmanImages[this.pacmanImageIndex],

    //   this.x,

    //   this.y,

    //   this.tileSize,

    //   this.tileSize

    // );

  }

  #loadPacmanImages() {

    const pacmanImage1 = new Image();

    pacmanImage1.src = "images/pac0.png";

    const pacmanImage2 = new Image();

    pacmanImage2.src = "images/pac1.png";

    const pacmanImage3 = new Image();

    pacmanImage3.src = "images/pac2.png";

    const pacmanImage4 = new Image();

    pacmanImage4.src = "images/pac1.png";

    this.pacmanImages = [

      pacmanImage1,

      pacmanImage2,

      pacmanImage3,

      pacmanImage4,

    ];

    this.pacmanImageIndex = 0;

  }

  #keydown = (event) => {

    //up

    if (event.keyCode == 38) {

      if (this.currentMovingDirection == MovingDirection.down)

        this.currentMovingDirection = MovingDirection.up;

      this.requestedMovingDirection = MovingDirection.up;

      this.madeFirstMove = true;

    }

    //down

    if (event.keyCode == 40) {

      if (this.currentMovingDirection == MovingDirection.up)

        this.currentMovingDirection = MovingDirection.down;

      this.requestedMovingDirection = MovingDirection.down;

      this.madeFirstMove = true;

    }

    //left

    if (event.keyCode == 37) {

      if (this.currentMovingDirection == MovingDirection.right)

        this.currentMovingDirection = MovingDirection.left;

      this.requestedMovingDirection = MovingDirection.left;

      this.madeFirstMove = true;

    }

    //right

    if (event.keyCode == 39) {

      if (this.currentMovingDirection == MovingDirection.left)

        this.currentMovingDirection = MovingDirection.right;

      this.requestedMovingDirection = MovingDirection.right;

      this.madeFirstMove = true;

    }

  };

  #move() {

    if (this.currentMovingDirection !== this.requestedMovingDirection) {

      if (

        Number.isInteger(this.x / this.tileSize) &&

        Number.isInteger(this.y / this.tileSize)

      ) {

        if (

          !this.tileMap.didCollideWithEnvironment(

            this.x,

            this.y,

            this.requestedMovingDirection

          )

        )

          this.currentMovingDirection = this.requestedMovingDirection;

      }

    }

    if (

      this.tileMap.didCollideWithEnvironment(

        this.x,

        this.y,

        this.currentMovingDirection

      )

    ) {

      this.pacmanAnimationTimer = null;

      this.pacmanImageIndex = 1;

      return;

    } else if (

      this.currentMovingDirection != null &&

      this.pacmanAnimationTimer == null

    ) {

      this.pacmanAnimationTimer = this.pacmanAnimationTimerDefault;

    }

    switch (this.currentMovingDirection) {

      case MovingDirection.up:

        this.y -= this.velocity;

        this.pacmanRotation = this.Rotation.up;

        break;

      case MovingDirection.down:

        this.y += this.velocity;

        this.pacmanRotation = this.Rotation.down;

        break;

      case MovingDirection.left:

        this.x -= this.velocity;

        this.pacmanRotation = this.Rotation.left;

        break;

      case MovingDirection.right:

        this.x += this.velocity;

        this.pacmanRotation = this.Rotation.right;

        break;

    }

  }

  #animate() {

    if (this.pacmanAnimationTimer == null) {

      return;

    }

    this.pacmanAnimationTimer--;

    if (this.pacmanAnimationTimer == 0) {

      this.pacmanAnimationTimer = this.pacmanAnimationTimerDefault;

      this.pacmanImageIndex++;

      if (this.pacmanImageIndex == this.pacmanImages.length)

        this.pacmanImageIndex = 0;

    }

  }

  #eatDot() {

    if (this.tileMap.eatDot(this.x, this.y) && this.madeFirstMove) {

      this.wakaSound.play();

    }

  }

  #eatPowerDot() {

    if (this.tileMap.eatPowerDot(this.x, this.y)) {

      this.powerDotSound.play();

      this.powerDotActive = true;

      this.powerDotAboutToExpire = false;

      this.timers.forEach((timer) => clearTimeout(timer));

      this.timers = [];

      let powerDotTimer = setTimeout(() => {

        this.powerDotActive = false;

        this.powerDotAboutToExpire = false;

      }, 1000 \* 6);

      this.timers.push(powerDotTimer);

      let powerDotAboutToExpireTimer = setTimeout(() => {

        this.powerDotAboutToExpire = true;

      }, 1000 \* 3);

      this.timers.push(powerDotAboutToExpireTimer);

    }

  }

  #eatGhost(enemies) {

    if (this.powerDotActive) {

      const collideEnemies = enemies.filter((enemy) => enemy.collideWith(this));

      collideEnemies.forEach((enemy) => {

        enemies.splice(enemies.indexOf(enemy), 1);

        this.eatGhostSound.play();

      });

    }

  }

}

***6.Tilemap.js***

import Pacman from "./Pacman.js";

import Enemy from "./Enemy.js";

import MovingDirection from "./MovingDirection.js";

export default class TileMap {

constructor(tileSize) {

this.tileSize = tileSize;

this.yellowDot = new Image();

this.yellowDot.src = "images/yellowDot.png";

this.pinkDot = new Image();

this.pinkDot.src = "images/pinkDot.png";

this.wall = new Image();

this.wall.src = "images/wall.png";

this.powerDot = this.pinkDot;

this.powerDotAnmationTimerDefault = 30;

this.powerDotAnmationTimer = this.powerDotAnmationTimerDefault;

}

map = [

[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],

[1, 7, 0, 0, 4, 0, 0, 0, 0, 0, 0, 7, 1],

[1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1],

[1, 0, 1, 6, 0, 0, 0, 0, 0, 0, 1, 0, 1],

[1, 0, 1, 7, 1, 1, 1, 0, 1, 0, 1, 0, 1],

[1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1],

[1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],

[1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],

[1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1],

[1, 6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6, 1],

[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],

];

draw(ctx) {

for (let row = 0; row < this.map.length; row++) {

for (let column = 0; column < this.map[row].length; column++) {

let tile = this.map[row][column];

if (tile === 1) {

this.#drawWall(ctx, column, row, this.tileSize);

} else if (tile === 0) {

this.#drawDot(ctx, column, row, this.tileSize);

} else if (tile == 7) {

this.#drawPowerDot(ctx, column, row, this.tileSize);

} else {

this.#drawBlank(ctx, column, row, this.tileSize);

}

);

}

}

}

#drawDot(ctx, column, row, size) {

ctx.drawImage(

this.yellowDot,

column \* this.tileSize,

row \* this.tileSize,

size,

size

);

}

#drawPowerDot(ctx, column, row, size) {

this.powerDotAnmationTimer--;

if (this.powerDotAnmationTimer === 0) {

this.powerDotAnmationTimer = this.powerDotAnmationTimerDefault;

if (this.powerDot == this.pinkDot) {

this.powerDot = this.yellowDot;

} else {

this.powerDot = this.pinkDot;

}

}

ctx.drawImage(this.powerDot, column \* size, row \* size, size, size);

}

#drawWall(ctx, column, row, size) {

ctx.drawImage(

this.wall,

column \* this.tileSize,

row \* this.tileSize,

size,

size

);

}

#drawBlank(ctx, column, row, size) {

ctx.fillStyle = "black";

ctx.fillRect(column \* this.tileSize, row \* this.tileSize, size, size);

}

getPacman(velocity) {

for (let row = 0; row < this.map.length; row++) {

for (let column = 0; column < this.map[row].length; column++) {

let tile = this.map[row][column];

if (tile === 4) {

this.map[row][column] = 0;

return new Pacman(

column \* this.tileSize,

row \* this.tileSize,

this.tileSize,

velocity,

this

);

}

}

}

}

getEnemies(velocity) {

const enemies = [];

for (let row = 0; row < this.map.length; row++) {

for (let column = 0; column < this.map[row].length; column++) {

const tile = this.map[row][column];

if (tile == 6) {

this.map[row][column] = 0;

enemies.push(

new Enemy(

column \* this.tileSize,

row \* this.tileSize,

this.tileSize,

velocity,

this

)

);

}

}

}

return enemies;

}

setCanvasSize(canvas) {

canvas.width = this.map[0].length \* this.tileSize;

canvas.height = this.map.length \* this.tileSize;

}

didCollideWithEnvironment(x, y, direction) {

if (direction == null) {

return;

}

if (

Number.isInteger(x / this.tileSize) &&

Number.isInteger(y / this.tileSize)

) {

let column = 0;

let row = 0;

let nextColumn = 0;

let nextRow = 0;

switch (direction) {

case MovingDirection.right:

nextColumn = x + this.tileSize;

column = nextColumn / this.tileSize;

row = y / this.tileSize;

break;

case MovingDirection.left:

nextColumn = x - this.tileSize;

column = nextColumn / this.tileSize;

row = y / this.tileSize;

break;

case MovingDirection.up:

nextRow = y - this.tileSize;

row = nextRow / this.tileSize;

column = x / this.tileSize;

break;

case MovingDirection.down:

nextRow = y + this.tileSize;

row = nextRow / this.tileSize;

column = x / this.tileSize;

break;

}

const tile = this.map[row][column];

if (tile === 1) {

return true;

}

}

return false;

}

didWin() {

return this.#dotsLeft() === 0;

}

#dotsLeft() {

return this.map.flat().filter((tile) => tile === 0).length;

}

eatDot(x, y) {

const row = y / this.tileSize;

const column = x / this.tileSize;

if (Number.isInteger(row) && Number.isInteger(column)) {

if (this.map[row][column] === 0) {

this.map[row][column] = 5;

return true;

}

}

return false;

}

eatPowerDot(x, y) {

const row = y / this.tileSize;

const column = x / this.tileSize;

if (Number.isInteger(row) && Number.isInteger(column)) {

const tile = this.map[row][column];

if (tile === 7) {

this.map[row][column] = 5;

return true;

}

}

return false;

}

}