Race Car Game

Introduction

Welcome to the documentation for our engaging race car game project implemented in C++. This semester-long project focuses on creating an immersive and interactive gaming experience. The game leverages various data structures, including graphs, linked lists, queues, and heaps, to efficiently manage game elements. Below, we provide an overview of the key components and implementation details.

Implementation

Graph Structure
Our game world is represented as a well-structured n x n graph, implemented using an adjacency list. Each vertex in the graph maintains references to its neighbors, which are stored in linked lists. Linked lists are also employed to manage in-game items such as coins and bonuses. During automatic mode, vertex highlighting is achieved through a stack. Obstacles dynamically appear in gameplay using a queue. Additionally, a heap data structure is employed to manage the leaderboard, prioritizing player achievements.

Classes and Methods

Car Class

Car::Car: Constructor for the Car class.
Car::checkWin: Check if the player has won the game.
Car::coordinates: Get the current coordinates of the car.
Car::moveDown, Car::moveLeft, Car::moveRight, Car::moveUp: Move the car in different directions.
Car::printPlayer: Display the player's details.
Car::RecordScore: Record the player's score.
Car::SearchName: Search for a player by name.
Car::showWin: Display the winning message.
Car::updateScore: Update the player's score.
Member variables: name, score, x, y.

Graph Class

Graph::~Graph: Destructor for the Graph class.
Graph::addEdge: Add an edge between two vertices.
Graph::buildMaze: Build the maze for the game.
Graph::ifDownExists, Graph::ifLeftExists, Graph::ifRightExists, Graph::ifUpExists: Check if a neighboring vertex exists in a given direction.
Graph::adjacencyList: Maintain the adjacency list.
Graph::gridSize: Get the size of the game grid.

LinkedList Class

LinkedList::addNeighbor: Add a neighbor to a vertex.
LinkedList::LinkedList: Constructor for the LinkedList class.
Overloaded LinkedList::operator[]: Access elements in the linked list.
Member variables: car, finish, head, numNeb, obstacle, powerup, weight, x, y.

PriorityQueue Class

PriorityQueue::~PriorityQueue: Destructor for the PriorityQueue class.
PriorityQueue::dequeue, PriorityQueue::enqueue: Perform enqueue and dequeue operations.
PriorityQueue::frontD, PriorityQueue::frontX, PriorityQueue::frontY: Get the front element's details.
PriorityQueue::isEmpty: Check if the priority queue is empty.
PriorityQueue::distance: Calculate the distance between two points.
PriorityQueue::front, PriorityQueue::rear, PriorityQueue::size, PriorityQueue::x, PriorityQueue::y: Member variables.

Queue Class

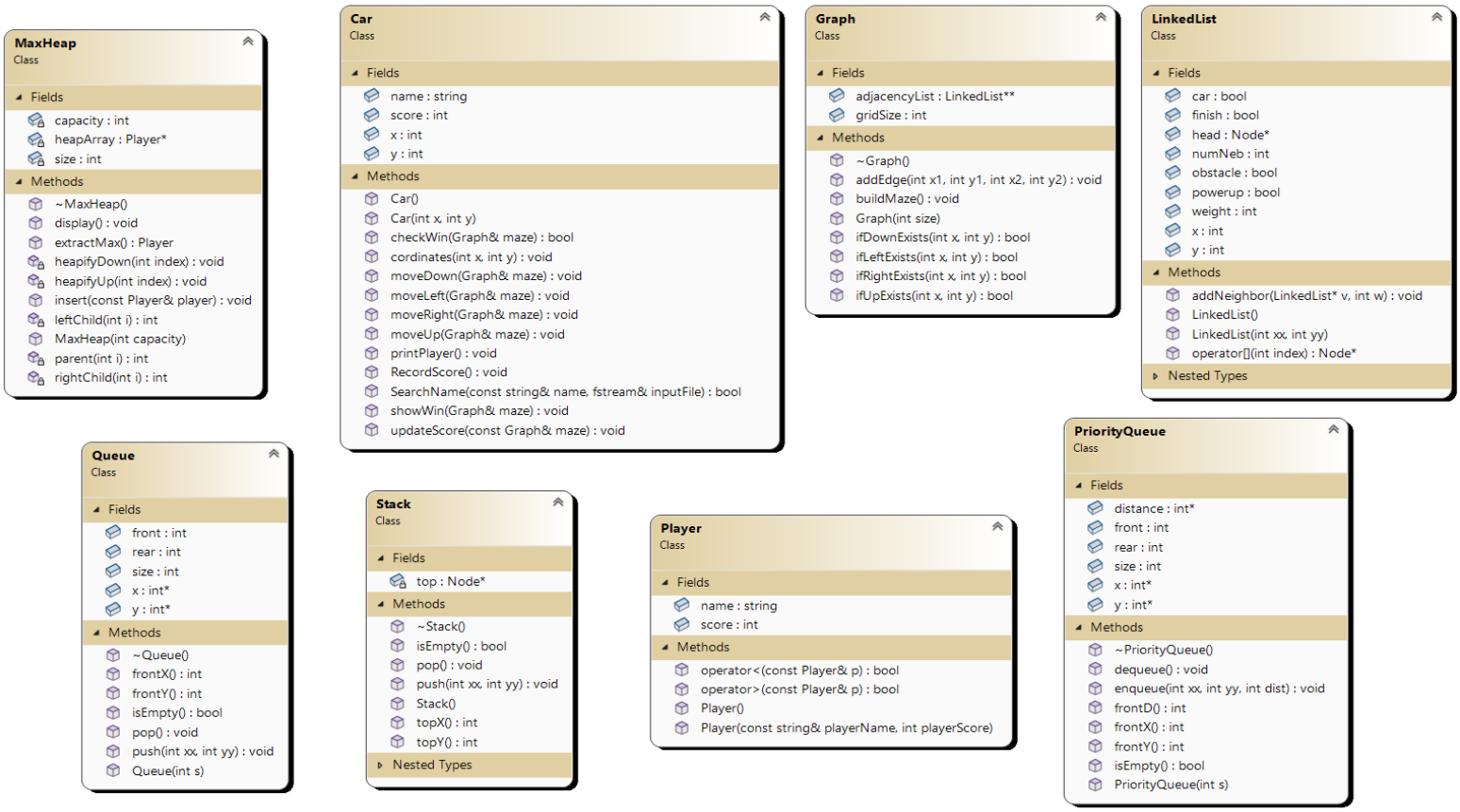
Queue::~Queue: Destructor for the Queue class.
Queue::frontX, Queue::frontY: Get the front element's coordinates.
Queue::isEmpty: Check if the queue is empty.
Queue::pop, Queue::push: Perform pop and push operations.
Queue::front, Queue::rear, Queue::size, Queue::x, Queue::y: Member variables.

Stack Class

Stack::~Stack: Destructor for the Stack class.
Stack::isEmpty: Check if the stack is empty.
Stack::pop, Stack::push: Perform pop and push operations.
Stack::topX, Stack::topY: Get the top element's coordinates.

Stack::top: Get the top element.

Class Diagram



Dijkstra's Algorithm in Auto Mode

In the auto mode of our race car game, we implemented Dijkstra's algorithm to determine the optimal path for the car to navigate through the maze. The graph structure, represented by an adjacency list, serves as the basis for this algorithm. Dijkstra's algorithm efficiently calculates the shortest path from the starting point to the finish line, considering the weights associated with each edge. As the car moves automatically, the algorithm dynamically updates the path, ensuring the car takes the most efficient route, avoiding obstacles and collecting power-ups along the way. The priority queue plays a crucial role in selecting the next vertex with the minimum distance, facilitating smooth and intelligent movement during auto mode.

Scoring System

The scoring system in our race car game adds a competitive and rewarding dimension to the player experience. Players earn points based on their performance, considering factors such as time taken to complete the race, the number of coins collected, and the successful navigation through challenging sections of the maze. Each completed race contributes to the player's overall score, motivating them to improve and compete with others on the leaderboard. Additionally, bonus points are awarded for achieving specific milestones, such as reaching the finish line without hitting obstacles or collecting all power-ups. The scoring system not only reflects the player's skills in navigating the maze but also encourages strategic decision-making and efficient gameplay.

Conclusion

This project demonstrates the practical application of various data structures in creating an engaging race car game. By utilizing graphs, linked lists, queues, and heaps, we've achieved efficient management of game elements, providing an enjoyable gaming experience. The implementation details of each class and method outlined above should serve as a comprehensive guide for understanding the project's structure and functionality.