Data Structures (CS-2001)

Project Report

Submitted by:

Faizan Rasheed (22i-2734)

Zaid Masood (22i-8793)

Submitted to:

Ms. Maheen Arshad

Table of Contents

1. Purpose

The project aims to provide a 2D car maze game for the users that is entertaining and easy to play. The goal is to provide this gaming experience within the console environment that involves maneuvering a player-controlled car that navigates through the maze, avoiding obstacles, collecting trophies and power-ups, and collecting coins to reach the endpoint with the highest scores the player can get. The implementation is written in C++ and the title chosen for the game is **"Text Overdrive".**

2. Project Structure

Header.h:

This contains all the class definitions used throughout the program. It only has the function declarations contained within these classes and the implementation is done separately in the other file.

Source.cpp

This contains all the function definitions for all the declared classes of the header files. It also contains the game loop and enables user interaction via the user input received from the main() function.

Built-in header files:

#include<iostream>:

- Used for basic input (cin) and output (cout) operations.

#include<string>:

- Used for storing alphabetic data such as names and to call some built-in string functions such as stoi() for file handling.

#include<Windows.h>:

- Used for console-related functions such as SetConsoleTextAttribute() and system("cls").

#include<conio.h>:

- Used for the function _getch() to get a single key input from user without the need to press the Enter key.

#include<iomanip>:

- Used for the function setw() to adjust the output character to fill up the screen.

#include<chrono>:

Used for utilizing the internal clock of the system for managing time points, and performing time related operaions e.g. std::chrono::steady_clock::now().

#include<fstream>:

- Used for reading from (ofstream) and writing to files (ifstream).

#include<sstream>:

- Used for converting between strings and other data types via getline() which reads a line from an input stream to a string .

```
#include <iostream>
#include <string>
#include <Windows.h>
#include <conio.h>
#include <iomanip>
#include <chrono>
#include "Header.h"
#include <fstream>
#include <sstream>
```

Figure 1 Built-in header files

Helping classes:

Node:

- Represents a node in the graph and/or a linked list.
- Contains a char data member named “data” to store character data and a pointer of type node* to the next node named “next”.

```
class node {
public:
char data;
node* next;
node();
node(char x);

};
```

Figure 2 Class node

Vertex:

- Represents a vertex in the graph.
- Contains a pointer to the head named “head” of type *node.

```
class vertex {
public:
    node* head;
    vertex();
};
```

Figure 3 Class vertex

StackNode:

- Represents a node in a stack.
- Contains a char data member named “data” to store character data and a pointer of type stacknode* named “next” that points to the next stacknode.

```
class stacknode {
public:
    char data;
    stacknode* next;
   stacknode();
   stacknode(char x);
};
```

Figure 4 Class stacknode

QueueNode:

- Represents a node in a queue.
- Contains a char data member named “data” to store character data and a pointer of type queuenode* named “next” that points to the next queuenode.

```
class queuenode {
public:
    char data;
    queuenode* next;
    queuenode();
    queuenode(char x);
};
```

Figure 5 Class queuenode

FileNode:

- Represents a node in a linked list for storing file information.
- Contains a string data member named “name” to store player name, int data member named “score” to store player score, and a pointer of type filenode* named “next” that to the next filenode.

```
class filenode {
public:
    string name;
    int score;
    filenode* next;
    filenode();
    filenode(string x,int y);
};
```

Figure 6 Class filenode

FileList:

- Represents a linked list for storing file information.
- Contains a pointer of type filenode* named “head” that points to the next filenode in the filelist.

```
class filelist {
public:
    filenode* head;
    filelist();
    void add(string x,int y);
    void sort();
};
```

Figure 7 Class filelist

TreeNode:

- Represents a node in a binary search tree (BST) for storing high scores.
- Contains a string data member named “name” to store the player’s name, int data member named “score”, and pointers of type treenode* named “left” and “right” that points to the left and right of the binary search tree.

```
class treenode {
    public:
    string name;
    int score;
    treenode* left;
    treenode* right;
    treenode();
    treenode(string n,int x);
};
```

Figure 8 Class treenode

3. Data Structures used

Graph:

Purpose:

- For map representation of the grid and navigation on the map grid enabling us to implement shortest path finding algorithms i.e. Dijkstra’s algorithm.

Usage:

- Contains data members such as score of type int to store “score” in real-time, “size” of type int to track the size of the grid, and 2D pointer of type vertex named “maze” enabling us to navigate the maze vertices.
- Contains member functions such as graph(), graph(int x) as constructors as well as functions such as createMaze(), createpath(), and others to help build the game logic and representation.

```
class graph {
public:
int score;
int size;
vertex** maze;
graph();
graph(int x);
void createMaze();
void printMaze();
int  createpath(int n,int x,int y);
void addEdge(int x, int y,int x2,int y2,char z);
void removeEdge(int x, int y, int x2, int y2);
queue movecar(queue q,stack *s,stack *h,int le,string nm,int pa,int level);
queue movecarresume(queue q, stack* s, stack* h, int le, string nm,int xr,int yr,int pa,int levl);
queue addobstacle(int x,int y,queue q,char p,int l);
void addcoin(int x, int y,char p);
void addcoinauto(int x, int y, char p);
void stats(stack s,string name,int sc);
void displayandsavetopscorer();
void addpathauto();
queue addobstaclelevelauto(int x, int y, queue q);
queue autosolve(queue q, stack* s, stack* h);

};
```

Figure 9 Class graph

Queue:

Purpose:

- For managing obstacle generation where obstacles are enqueued when the program is initialized and dequeued when the game starts.

Usage:

- Contains data members such as a pointer of type queuenode* named front to store the front of the queue, a pointer of type queuenode* named rear to store the rear of the queue.
- Contains member functions typically found in a queue data structure.

```
class queue {
public:
    queuenode* front;
    queuenode* rear;
    char frontelement();
    queue();
    void enqueue(char x);
    char dequeue();

    bool isEmpty();

};
```

Figure 10 Class queue

Stack:

Purpose:

- For storing and maintaining user scores during the runtime of the game as it increases/decreases.

Usage:

- Contains a single data member named "top" as pointer of type stacknode*.
- Contains member functions typically found in a stack data structure along with additional ones such as search(), remove(), emptystack(), print(), calc().

```
class stack {
public:
    stacknode* top;
    stack();
   void push(char x);
    char pop();
    void remove(char x);
    int calculate();
    bool search(char x);
    bool isEmpty();
    void emptystack();
    void print();
    void calc();


};
```

Figure 11 Class stack

Binary Search Tree:

Purpose:

- For storing high scores achieved during the game that are visible via the high scores option in the game menu.

Usage:

- Contains a single data member of named "root" which is a pointer of type treenode* that stores the root node of the BST.
- Contains member functions for inserting the name and score (insert(string n,int x)) and traversing the tree in order (inorder()).

```
class BSTtree {

    public:
        BSTtree();
        treenode* root;
        void insert(string n,int x);
        void inorder(treenode* root);
        void inorder();


};
```

Figure 12 class BSTtree

4. Game Logic

Game Loop:

The game loop refers to the continuous execution the game until the user voluntarily quits the game, updating the game state to update the user actions, rendering the frames after the user has performed their respective actions and display the results, and capturing user input via functions such as _getch() to commence actions against them.

User Interface:

The user interface provides a clear and understandable display for the users to navigate and play the game.

Main menu:

The options provided in the main menu help facilitate the user in navigating to different functions of the game. The screenshot below shows us various options such as "high scores", "controls" etc.

Figure 13 Main menu

Gameplay:

The gameplay is affected by the selection of the choices after the "play" option has been chosen. The user is presented with three options, the first option is to select whether the game should be played in manual mode or automatic mode along with the option to resume a previously saved game session, second is the option to choose the preferred difficulty ("easy", "medium", "hard")

of the game which affects the obstacle generation of the game, step limits, and items spawn rate, and the third is the level selection (3 levels) where each level has a different grid map representation with a unique path and trophies to collect.

Figure 14 Game mode

Figure 15 Difficulty mode

Figure 16 Level selection

Before the game begins, the user is prompted to enter their name which is used in case the user may get a high score:

Figure 17 Enter your name

After entering the game, user gets to see the map along with indicators such as “score”, “steps”, “your health”, “your collections”:

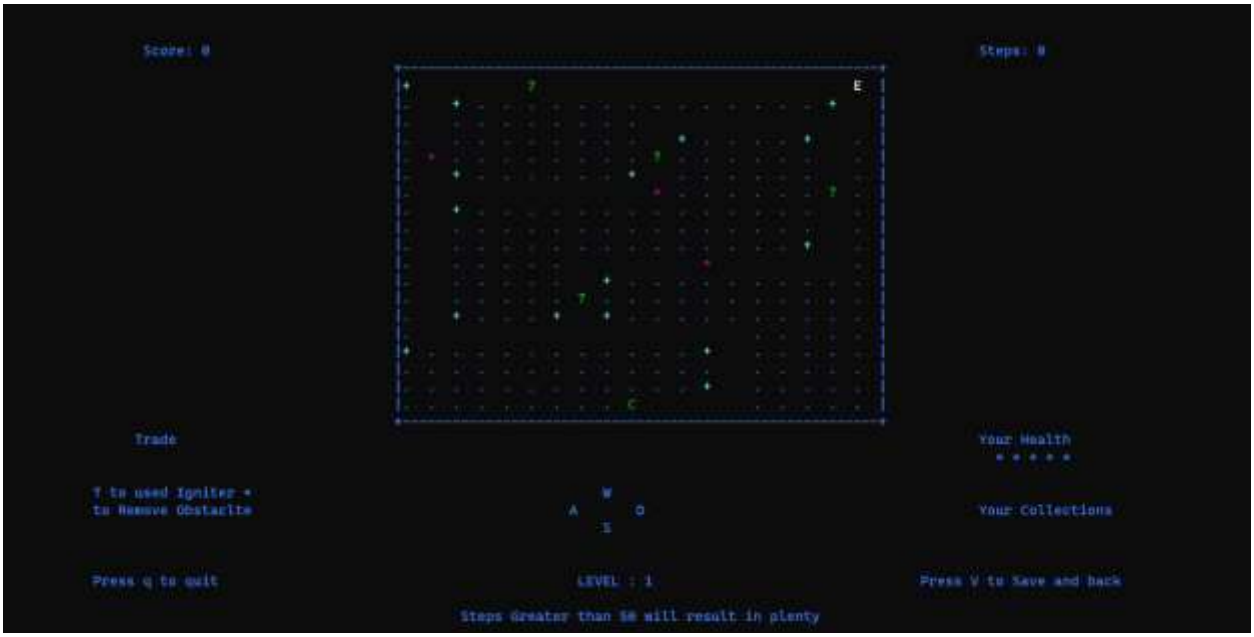

Figure 18 Gameplay screen (manual)

Scoring system:

The scoring system is dependent on the user's collection of items and collisions with obstacles:

$- represents a coin that will increment the score by 1 point.

@ and #-represent obstacles that will decrease the score by 1 point.

?-represents a trophy that will increment the score by 5 points.

Furthermore, if the user exceeds the stated number of steps then the score will start to decrease by 1 with each step taken. The score count doesn't affect the final state of the game, hence, it is entirely possible to finish the game with a negative score count.

Collision system:

The collision system is affected by the obstacles present on the path represented by symbols such as @ and # which will count as a collision. The collison will lead to deduction the user's "health" that is represented by 5 points. When all 5 points have depleted, the game will be considered as "over".

Figure 19 Game over when health is depleted

5. Usage

Manual:

The manual mode is a user-controlled mode that will have the player character (represented by "C") navigated through the maze via the WASD keys (W=up, A=left, S=down, D=right) or the arrow keys (changeable via the "controls" option in the menu). The user must use their understanding of the path to take the best route and get to the endpoint without losing all health points and achieving the highest score possible. The difficulty levels modify how the game will play out:

"Easy"- More coins, less obstacles, and more number of steps

"Medium"- Medium coins, medium obstacles, and medium number of steps

"Hard"- Less coins, more obstacles, and less number of steps.

The obstacle and coin generation will be randomized meaning the user can be forced to change their path depending how the obstacles and coins are generated.

Automatic:

The automatic mode is an automated mode where the "C" will move through the path which it will calculate to be the shortest. The car will then move this path and can change it's direction depending upon whether it detects an obstacles or not.

_-represents the highlighted path that the car has taken

The obstacle and coin generation will stay similar to the manual mode as they will pop up randomly throughout the map.

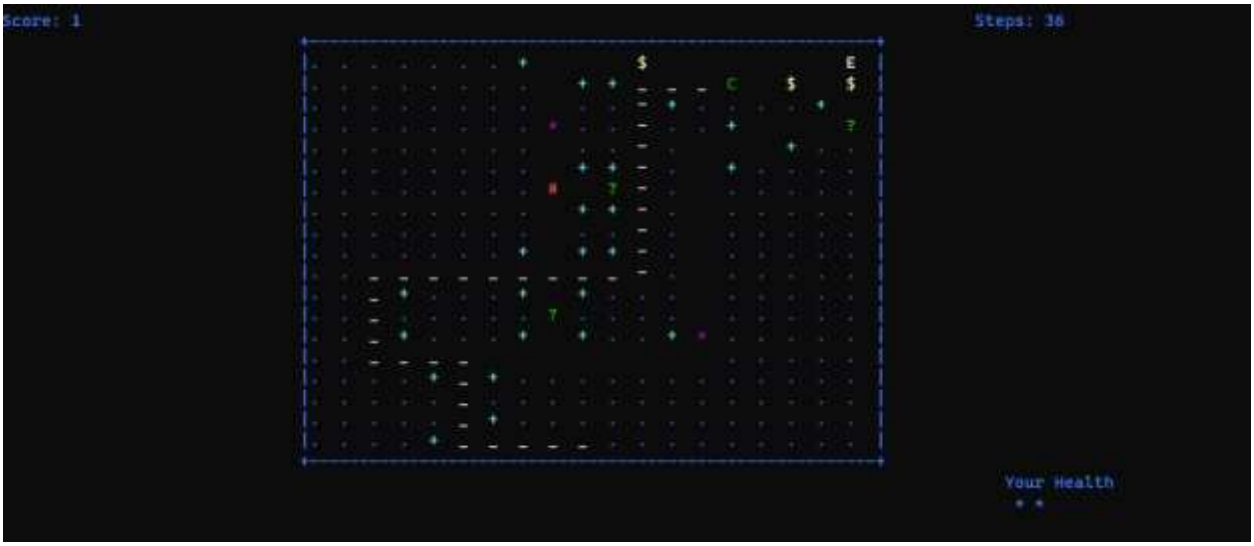

Figure 20 Automatic mode

←----------End of report----------→