#include <SFML/Graphics.hpp>

#include <iostream>

#include<string.h>

#include<cstdlib>

#include <fstream>

using namespace std;

using namespace sf;

const int gridSize = 8;

const int cellSize = 64; // each cell in size is 64 units

const int scoreFontSize = 20; // Font size for the score

const int timerFontSize = 20; // Font size for time

const int MAX\_PLAYERS = 5;

const string SCORES\_FILE = "scores.txt";

const string FONT\_FILE = "arial.ttf";

const string MAIN\_BACKGROUND = ".\\image\\main\_background.png";

const string INSTRUCT\_BACKGROUND = ".\\image\\instruct\_background.jpg";

const string GAME\_BACKGROUND = ".\\image\\background.jpg";

const string GEM\_IMAGES\_PATH = ".\\image";

void displayMatrix(int gemMatrix[gridSize][gridSize]);

void generateGem(int gemMatrix[gridSize][gridSize]);

void checkForThreeMatches(int gemMatrix[gridSize][gridSize], int& score);

void checkForFourFlameGemMatches(int gemMatrix[gridSize][gridSize], int& score);

void checkForFourMatches(int gemMatrix[gridSize][gridSize], int& score, int selectedX, int selectedY);

void checkForFiveMatches(int gemMatrix[gridSize][gridSize], int& score);

void fillEmptySpaces(int gemMatrix[gridSize][gridSize]);

void checkAndDestroyMatches(int gemMatrix[gridSize][gridSize], int& score, int selectedX, int selectedY);

void displayScore(sf::RenderWindow& window, sf::Font& font, int score);

void displayTimer(RenderWindow& window, sf::Font& font, int time);

void displayRemarks(RenderWindow& window, Font& font, string remarks);

void drawGemMatrixWindow(RenderWindow& window, Font& font, int gemMatrix[gridSize][gridSize],

Texture gemTextures[],

int space,

int selectedX, int selectedY);

bool checkIfHorizontallySwapPossible(int gemMatrix[gridSize][gridSize]);

bool checkIfVerticallySwapPossible(int gemMatrix[gridSize][gridSize]);

bool isValidSwap(int gemMatrix[gridSize][gridSize], int cursorX, int cursorY, int newCursorY, int newCursorX);

void displayGameEnd(Font& font, int& score);

bool intializeGame(RenderWindow& window, Sprite& backgroundSprite, Texture& backgroundTextrue, Texture gemTexture[], int gemMatrix[gridSize][gridSize]);

void handleGameEvents(RenderWindow& window, int gemMatrix[gridSize][gridSize], int& cursorX, int& cursorY, bool& enterPressed);

void playGame(Font& font);

void showInstructions(Font& font);

void showCreditsWindow(Font& font);

void displayMainWindow(Font& font);

string getInputName(Font& font);

void saveScore(const string& name, int score);

void loadScores(string names[MAX\_PLAYERS], int scores[MAX\_PLAYERS]);

void sortAccordingToScore(string names[MAX\_PLAYERS], int scores[MAX\_PLAYERS]);

void displayTopScores(Font& font);

int main() {

Font font;

if (!font.loadFromFile(FONT\_FILE)) {

cout << "Failed to load font file." << endl;

return false;

}

displayMainWindow(font);

return 0;

}

// Function to display main window

void displayMainWindow(Font& font) {

// Create a window for the main menu with dimensions 800x600 and title "Main Menu"

RenderWindow window(VideoMode(800, 600), "Main Menu");

// Load background texture for the main menu

Texture backgroundTexture;

if (!backgroundTexture.loadFromFile(MAIN\_BACKGROUND)) {

return; // Return if there's an error loading the background image

}

// Create a sprite for the background image and adjust its scale to fit the window

Sprite backgroundSprite(backgroundTexture);

backgroundSprite.setScale(static\_cast<float>(window.getSize().x) / backgroundTexture.getSize().x,

static\_cast<float>(window.getSize().y) / backgroundTexture.getSize().y);

// Button properties

const int buttonWidth = 200;

const int buttonHeight = 50;

const Color buttonBackgroundColor = Color(200, 200, 200);

const Color buttonBorderColor = Color::Red;

const Color textColor = Color::Black;

// Play button

RectangleShape playButton(Vector2f(buttonWidth, buttonHeight));

playButton.setFillColor(buttonBackgroundColor);

playButton.setOutlineColor(buttonBorderColor);

playButton.setOutlineThickness(2); // Border thickness

playButton.setPosition((window.getSize().x - buttonWidth) / 2, 100);

Text playText("Play", font, 30);

playText.setFillColor(textColor);

playText.setPosition((window.getSize().x - playText.getLocalBounds().width) / 2,

playButton.getPosition().y + (buttonHeight - playText.getLocalBounds().height) / 2);

// Credits button

RectangleShape creditsButton(Vector2f(buttonWidth, buttonHeight));

creditsButton.setFillColor(buttonBackgroundColor);

creditsButton.setOutlineColor(buttonBorderColor);

creditsButton.setOutlineThickness(2); // Border thickness

creditsButton.setPosition((window.getSize().x - buttonWidth) / 2, 200);

Text creditsText("Credits", font, 30);

creditsText.setFillColor(textColor);

creditsText.setPosition((window.getSize().x - creditsText.getLocalBounds().width) / 2,

creditsButton.getPosition().y + (buttonHeight - creditsText.getLocalBounds().height) / 2);

// Instructions button

RectangleShape instructionsButton(Vector2f(buttonWidth, buttonHeight));

instructionsButton.setFillColor(buttonBackgroundColor);

instructionsButton.setOutlineColor(buttonBorderColor);

instructionsButton.setOutlineThickness(2); // Border thickness

instructionsButton.setPosition((window.getSize().x - buttonWidth) / 2, 300);

Text instructionsText("Instructions", font, 30);

instructionsText.setFillColor(textColor);

instructionsText.setPosition((window.getSize().x - instructionsText.getLocalBounds().width) / 2,

instructionsButton.getPosition().y + (buttonHeight - instructionsText.getLocalBounds().height) / 2);

RectangleShape topScoreButton(Vector2f(buttonWidth, buttonHeight));

topScoreButton.setFillColor(buttonBackgroundColor);

topScoreButton.setOutlineColor(buttonBorderColor);

topScoreButton.setOutlineThickness(2); // Border thickness

topScoreButton.setPosition((window.getSize().x - buttonWidth) / 2, 400);

// Top score button

Text topScoresText("Top Scores", font, 30);

topScoresText.setFillColor(textColor);

topScoresText.setPosition((window.getSize().x - topScoresText.getLocalBounds().width) / 2,

topScoreButton.getPosition().y + (buttonHeight - topScoresText.getLocalBounds().height) / 2);

// Exit button

RectangleShape exitButton(Vector2f(buttonWidth, buttonHeight));

exitButton.setFillColor(buttonBackgroundColor);

exitButton.setOutlineColor(buttonBorderColor);

exitButton.setOutlineThickness(2); // Border thickness

exitButton.setPosition((window.getSize().x - buttonWidth) / 2, 500);

Text exitText("Exit", font, 30);

exitText.setFillColor(textColor);

exitText.setPosition((window.getSize().x - exitText.getLocalBounds().width) / 2,

exitButton.getPosition().y + (buttonHeight - exitText.getLocalBounds().height) / 2);

// Event handling loop for the main menu window

while (window.isOpen()) {

Event event;

while (window.pollEvent(event)) {

// Event handling for window closure

// Event handling for button clicks

if (event.type == Event::Closed) {

window.close();

}

// Check for button clicks

if (event.type == Event::MouseButtonPressed) {

if (event.mouseButton.button == Mouse::Left) {

Vector2i mousePos = Mouse::getPosition(window);

// Check if Play button is clicked

if (playButton.getGlobalBounds().contains(static\_cast<float>(mousePos.x), static\_cast<float>(mousePos.y))) {

window.clear();

window.close();

playGame(font);

break;

}

// Check if Credits button is clicked

else if (creditsButton.getGlobalBounds().contains(static\_cast<float>(mousePos.x), static\_cast<float>(mousePos.y))) {

window.clear();

window.close();

showCreditsWindow(font);

break;

// Add logic for credits here

}

// Check if Instructions button is clicked

else if (instructionsButton.getGlobalBounds().contains(static\_cast<float>(mousePos.x), static\_cast<float>(mousePos.y))) {

window.clear();

window.close();

showInstructions(font);

break;

}

// Check if Top scores button is clicked

else if (topScoreButton.getGlobalBounds().contains(static\_cast<float>(mousePos.x), static\_cast<float>(mousePos.y))) {

window.clear();

window.close();

displayTopScores(font);

break;

}

// Check if Exit button is clicked

if (exitButton.getGlobalBounds().contains(static\_cast<float>(mousePos.x), static\_cast<float>(mousePos.y))) {

window.close();

}

}

}

}

// Clear the window

window.clear();

window.draw(backgroundSprite);

// Draw buttons

window.draw(playButton);

window.draw(playText);

window.draw(creditsButton);

window.draw(creditsText);

window.draw(instructionsButton);

window.draw(instructionsText);

window.draw(topScoreButton);

window.draw(topScoresText);

window.draw(exitButton);

window.draw(exitText);

// Display the contents of the window

window.display();

}

}

// Function to get user input for name through a window

string getInputName(Font& font) {

// Create a window for user input with dimensions 300x300 and title "User Input"

RenderWindow window(VideoMode(300, 300), "User Input");

// Text objects for prompting user and displaying input text

Text promptText;

// Setting properties for promptText

promptText.setFont(font);

promptText.setCharacterSize(30);

promptText.setFillColor(sf::Color::White);

promptText.setPosition(10, 10);

promptText.setString("Enter your name:");

Text inputText;

// Setting properties for inputText

inputText.setFont(font);

inputText.setCharacterSize(30);

inputText.setFillColor(sf::Color::White);

inputText.setPosition(10, 50);

string playerName; // String to store user input

// Event handling loop for the user input window

while (window.isOpen()) {

sf::Event event;

while (window.pollEvent(event)) {

// Event handling for window closure

if (event.type == Event::Closed) {

window.close();

}

else if (sf::Keyboard::isKeyPressed(sf::Keyboard::Enter)) {

cout << "Enter pressed" << endl;

window.clear();

window.close();

break;

}

// Event handling for keyboard input

else if (event.type == Event::TextEntered) {

if (event.text.unicode < 128) {

char inputChar = static\_cast<char>(event.text.unicode);

if (inputChar == '\n') {

cout << "entere pressed";

// If Enter key is pressed, break out of loop

break;

}

else {

// Append entered character to playerName and update inputText

playerName += inputChar;

inputText.setString(playerName);

}

}

}

}

// Clear the window and draw promptText and inputText

window.clear();

window.draw(promptText);

window.draw(inputText);

window.display();

}

window.close(); // Close the window when input is complete

return playerName; // Return the user-entered name

}

// Function to save the player's score along with their name to a file

void saveScore(const string& name, int score) {

string names[MAX\_PLAYERS];

int scores[MAX\_PLAYERS];

loadScores(names, scores);

sortAccordingToScore(names, scores);

string tempNames[MAX\_PLAYERS];

int tempScores[MAX\_PLAYERS];

ofstream scoresFile(SCORES\_FILE, ios::out);

if (scoresFile.is\_open()) {

int i = 0;

int j = 0;

// Copy existing scores to temp arrays until reaching the end or finding a higher score

while (i < MAX\_PLAYERS && (scores[j] >= 0 && scores[j] >= score)) {

tempNames[i] = names[j];

tempScores[i] = scores[j];

i++;

j++;

}

// Insert the new score

if (i < MAX\_PLAYERS) {

tempNames[i] = name;

tempScores[i] = score;

i++;

}

// Copy remaining scores

while (i < MAX\_PLAYERS) {

tempNames[i] = names[j];

tempScores[i] = scores[j];

i++;

j++;

}

// Write temp arrays to the file

for (int i = 0; i < MAX\_PLAYERS; i++) {

scoresFile << tempNames[i] << " " << tempScores[i] << endl;

}

scoresFile.close();

}

else {

cout << "Failed to open scores file for writing." << endl; // Display an error message if the file failed to open

}

}

// Function to load scores and names from a file

void loadScores(string names[MAX\_PLAYERS], int scores[MAX\_PLAYERS]) {

// Open the scores file for reading

ifstream scoresFile(SCORES\_FILE);

// Check if the file is successfully opened

if (scoresFile.is\_open()) {

// Read player names and scores from the file

for (int i = 0; i < MAX\_PLAYERS; ++i) {

// Read name and score pairs into arrays names[] and scores[]

if (scoresFile >> names[i] >> scores[i]) {

// Continue reading if data is successfully read

}

else {

break; // Break loop if there's no more data to read

}

}

scoresFile.close(); // Close the file after reading

}

else {

// Display an error message if the file failed to open

cout << "Failed to open scores file for reading." << endl;

}

}

// Function to sort player names and scores in descending order based on scores

void sortAccordingToScore(string names[MAX\_PLAYERS], int scores[MAX\_PLAYERS]) {

// Bubble sort algorithm to sort scores and corresponding names

for (int i = 0; i < MAX\_PLAYERS - 1; ++i) {

for (int j = 0; j < MAX\_PLAYERS - i - 1; ++j) {

if (scores[j] < scores[j + 1]) {

swap(scores[j], scores[j + 1]);

swap(names[j], names[j + 1]);

}

}

}

}

// Function to display the top scores in a window

void displayTopScores(Font& font) {

// Create a window for displaying highest scores

RenderWindow window(VideoMode(800, 600), "Highest Score");

// Arrays to store player names and scores

string names[MAX\_PLAYERS];

int scores[MAX\_PLAYERS];

// Load scores and names from a file

loadScores(names, scores);

sortAccordingToScore(names, scores);

// Prepare a string containing names and scores

string nameWithScores = "";

for (int i = 0; i < MAX\_PLAYERS; ++i) {

if (scores[i] > 0) {

nameWithScores += to\_string(i + 1) + ". " + names[i] + ": " + to\_string(scores[i]) + "\n";

}

}

// Load background texture for the window

Texture backgroundTexture;

if (!backgroundTexture.loadFromFile(INSTRUCT\_BACKGROUND)) {

return;

}

// Create a sprite for the background image and adjust its scale to fit the window

Sprite backgroundSprite(backgroundTexture);

backgroundSprite.setScale(static\_cast<float>(window.getSize().x) / backgroundTexture.getSize().x,

static\_cast<float>(window.getSize().y) / backgroundTexture.getSize().y);

// Instructions text with line wrapping

Text topScoresText;

topScoresText.setFont(font);

topScoresText.setCharacterSize(20);

topScoresText.setFillColor(Color::White);

topScoresText.setPosition(20, 20);

// Set line spacing and wrap text

topScoresText.setLineSpacing(1.5);

topScoresText.setString(

"\n\n\n\n\Hghest Scores\n"

+ nameWithScores

);

// Back button properties

const int buttonWidth = 200;

const int buttonHeight = 50;

RectangleShape backButton(Vector2f(buttonWidth, buttonHeight));

backButton.setFillColor(Color::Cyan);

backButton.setOutlineThickness(2);

backButton.setPosition(250, 400);

Text backText("Back", font, 30);

backText.setFillColor(Color::Black);

backText.setPosition((window.getSize().x - backText.getLocalBounds().width) / 2,

backButton.getPosition().y + (buttonHeight - backText.getLocalBounds().height) / 2);

// Main loop for the instructions window

while (window.isOpen()) {

Event event;

while (window.pollEvent(event)) {

if (event.type == Event::Closed) {

window.close();

}

if (event.type == Event::MouseButtonPressed) {

if (event.mouseButton.button == Mouse::Left) {

Vector2i mousePos = Mouse::getPosition(window);

// Check if Play button is clicked

if (backButton.getGlobalBounds().contains(static\_cast<float>(mousePos.x), static\_cast<float>(mousePos.y))) {

cout << "Play button clicked! Game starts..." << endl;

window.clear();

window.close();

break;

}

}

}

}

// Clear the instructions window

window.clear();

// Draw instructions text

window.draw(backgroundSprite);

window.draw(topScoresText);

window.draw(backButton);

window.draw(backText);

// Display the contents of the instructions window

window.display();

}

displayMainWindow(font);

}

// Function to display the top scores in a window

void playGame(Font& font) {

// Create a window for displaying highest scores

RenderWindow window(VideoMode(gridSize \* cellSize + 200, gridSize \* cellSize), "Bejeweled Blitz");

// Get player name through a user input window

string playerName = getInputName(font);

// Declare necessary variables for game elements

Sprite backgroundSprite;

Texture backgroundTexture;

Texture gemTextures[7];

Clock clock;

int gemMatrix[gridSize][gridSize]; // Matrix to store gem information

int score = 1; // Initalize Score

float timeRemaining = 120; // Set initial time remaining

bool enterPressed = false; // Track Enter key press

int cursorX = 1; // X-coordinate of cursor

int cursorY = 1; // Y-coordinate of cursor

// Initialize the game elements and check if successful

if (intializeGame(window, backgroundSprite, backgroundTexture, gemTextures, gemMatrix)) {

// Check for initial matches and update score

checkAndDestroyMatches(gemMatrix, score, cursorX, cursorY);

score = 0; // Reset score

// Main game loop running while the window is open

while (window.isOpen()) {

// Handle game events (e.g., mouse clicks)

handleGameEvents(window, gemMatrix, cursorX, cursorY, enterPressed);

// Update time remaining based on elapsed time

Time elapsed = clock.restart();

timeRemaining -= (elapsed.asSeconds());

// Check if time is up, then save the score and display game end

if (timeRemaining <= 0) {

window.clear();

window.close();

saveScore(playerName, score); // Save player score

displayGameEnd(font, score); // Display game end screen

break; // Break out of the game loop

}

else {

int prevScore = score;

// Check for gem matches and update score if necessary

checkAndDestroyMatches(gemMatrix, score, cursorX, cursorY);

if (prevScore != score) {

sleep(milliseconds(500));

}

// Clear the window and draw game elements

window.clear();

window.draw(backgroundSprite);

const int space = 5; // Space between gem images

drawGemMatrixWindow(window, font, gemMatrix, gemTextures, space, cursorY, cursorX); // Draw gem matrix

displayScore(window, font, score); // Display current score

displayTimer(window, font, timeRemaining); // Display remaining time

}

window.display(); // Display updated window contents

}

}

else {

cout << "could not load game due to error\n"; // Display error if game initialization fails

}

}

// Function to initialize the game by setting up the game background, gem textures, and matrix

bool intializeGame(RenderWindow& window, Sprite& backgroundSprite, Texture& backgroundTexture, Texture gemTextures[], int gemMatrix[gridSize][gridSize]) {

// Load game background texture

if (!backgroundTexture.loadFromFile(GAME\_BACKGROUND)) {

// Return false if failed to load the background texture

return false;

}

// Set the game background sprite and adjust its scale to fit the window

backgroundSprite.setTexture(backgroundTexture);

backgroundSprite.setScale(static\_cast<float>(window.getSize().x) / backgroundTexture.getSize().x,

static\_cast<float>(window.getSize().y) / backgroundTexture.getSize().y);

// Load gem textures from files

for (int i = 0; i < 7; ++i) {

// Display error message if failed to load a gem texture

if (!gemTextures[i].loadFromFile(GEM\_IMAGES\_PATH + "\\gem" + to\_string(i) + ".png")) {

cout << "Failed to load gem" << i << ".png" << endl;

return false;

}

}

// Generate initial gem matrix for the game

generateGem(gemMatrix);

return true; // Return true to indicate successful initialization of the game

}

// Function to generate initial gem matrix for the game

void generateGem(int gemMatrix[gridSize][gridSize]) {

srand(42); // Seed the random number generator with a fixed value (42 in this case)

for (int i = 0; i < gridSize; ++i) {

for (int j = 0; j < gridSize; ++j) {

gemMatrix[i][j] = rand() % 7; // You can replace this with your logic for generating gem types

}

}

}

// Function to handle game events such as keyboard input for gem swapping and window closure

void handleGameEvents(RenderWindow& window, int gemMatrix[gridSize][gridSize], int& cursorX, int& cursorY, bool& enterPressed) {

Event event;

while (window.pollEvent(event)) {

if (event.type == Event::Closed) {

window.close(); // Close the window if the close event is triggered

}

// Handle arrow key input for gem swapping

else

if (event.type == Event::KeyPressed) {

int newCursorX = cursorX;

int newCursorY = cursorY;

if (event.key.code == Keyboard::Enter) { // Set enterPressed flag when the Enter key is pressed

enterPressed = true;

}

else if (event.key.code == Keyboard::Up && cursorY > 0) {

newCursorY = cursorY - 1; // Move cursor up if the Up arrow key is pressed and cursorY is greater than 0

}

else if (event.key.code == Keyboard::Down && cursorY < gridSize - 1) {

newCursorY = cursorY + 1; // Move cursor down if the Down arrow key is pressed and cursorY is less than gridSize - 1

}

else if (event.key.code == Keyboard::Left && cursorX > 0) {

newCursorX = cursorX - 1; // Move cursor left if the Left arrow key is pressed and cursorX is greater than 0

}

else if (event.key.code == Keyboard::Right && cursorX < gridSize - 1) {

newCursorX = cursorX + 1; // Move cursor right if the Right arrow key is pressed and cursorX is less than gridSize - 1

}

// Perform gem swap if cursor movement is valid and Enter key is pressed

if (newCursorX != cursorX || newCursorY != cursorY) {

if (enterPressed) {

if (isValidSwap(gemMatrix, cursorX, cursorY, newCursorY, newCursorX)) {

// Swap gems in the matrix if the swap is valid

swap(gemMatrix[cursorY][cursorX], gemMatrix[newCursorY][newCursorX]);

}

enterPressed = false; // Reset enterPressed flag after swapping

}

cursorX = newCursorX; // Update cursorX with new position

cursorY = newCursorY; // Update cursorY with new position

}

}

}

}

// Function to draw the gem matrix on the game window with highlighted selection

void drawGemMatrixWindow(RenderWindow& window, Font& font, int gemMatrix[gridSize][gridSize],

Texture gemTextures[],

int space,

int selectedX, int selectedY) {

for (int i = 0; i < gridSize; ++i) {

for (int j = 0; j < gridSize; ++j) {

if (gemMatrix[i][j] >= 7) {

RectangleShape background;

background.setSize(Vector2f(cellSize - 2 \* space, cellSize - 2 \* space));

background.setFillColor(Color::Black);

background.setPosition(j \* cellSize + space, i \* cellSize + space);

window.draw(background);

}

// Create a sprite for each gem and position it accordingly

Sprite gemSprite(gemTextures[(gemMatrix[i][j] % 7)]);

gemSprite.setPosition(j \* cellSize + space, i \* cellSize + space);

float scale = static\_cast<float>(cellSize - 2 \* space) / gemTextures[(gemMatrix[i][j] % 7)].getSize().x;

gemSprite.setScale(scale, scale);

window.draw(gemSprite); // Draw the gem sprite on the window

// Highlight the selected gem with a bounding box

if (i == selectedX && j == selectedY) {

RectangleShape boundingBox;

boundingBox.setSize(Vector2f(cellSize - 2 \* space, cellSize - 2 \* space));

boundingBox.setOutlineThickness(4);

boundingBox.setOutlineColor(Color(255, 215, 0)); // Set the outline color for the bounding box

boundingBox.setFillColor(Color(0, 0, 0, 0)); // Set transparent fill color

boundingBox.setPosition(j \* cellSize + space, i \* cellSize + space);

window.draw(boundingBox); // Draw the bounding box around the selected gem

}

}

}

}

// Function to check for gem matches and perform necessary actions

void checkAndDestroyMatches(int gemMatrix[gridSize][gridSize], int& score, int selectedX, int selectedY) {

checkForFourFlameGemMatches(gemMatrix, score); // Check for matches of four gems and draw create a flame gem and update the score

checkForFiveMatches(gemMatrix, score); // Check for matches of five gems including the selected gem and update the score

checkForFourMatches(gemMatrix, score, selectedX, selectedY); // Check for matches of four or more gems and update the score

checkForThreeMatches(gemMatrix, score); // Check for matches of three gems and update the score

fillEmptySpaces(gemMatrix); // Fill empty spaces in the gem matrix after gem destruction

}

// Function to display the gem matrix in the console

void displayMatrix(int gemMatrix[gridSize][gridSize]) {

for (int i = 0; i < gridSize; i++) {

for (int j = 0; j < gridSize; j++) {

cout << gemMatrix[i][j] << " "; // Print each element of the gem matrix followed by a space

}

cout << "\n"; // Move to the next row in the console output

}

}

// Check for matches of three and update the score

void checkForThreeMatches(int gemMatrix[gridSize][gridSize], int& score) {

for (int i = 0; i < gridSize; ++i) {

int count = 1; // Count of consecutive matching gems

for (int j = 1; j < gridSize; ++j) {

if (gemMatrix[i][j] == gemMatrix[i][j - 1] && gemMatrix[i][j] != -1) {

// Increment the count for consecutive matching gems

count++;

}

else {

// Reset the count if the sequence is broken

count = 1;

}

// Check if we have three or more consecutive matching gems

if (count == 3) {

// Destroy the matching gems

for (int k = j - count + 1; k <= j; ++k) {

gemMatrix[i][k] = -1;

}

score += 30; // Increase score (adjust accordingly)

}

}

}

// Check vertically

for (int j = 0; j < gridSize; ++j) {

int count = 1; // Count of consecutive matching gems

for (int i = 1; i < gridSize; ++i) {

if (gemMatrix[i][j] == gemMatrix[i - 1][j] && gemMatrix[i][j] != -1) {

// Increment the count for consecutive matching gems

count++;

}

else {

// Reset the count if the sequence is broken

count = 1;

}

// Check if we have three or more consecutive matching gems

if (count >= 3) {

// Destroy the matching gems

for (int k = i - count + 1; k <= i; k++) {

gemMatrix[k][j] = -1;

}

score += 30; // Increase score (adjust accordingly)

}

}

}

}

// This function checks for matches of four or more consecutive identical gems in a grid and updates the score.

void checkForFourFlameGemMatches(int gemMatrix[gridSize][gridSize], int& score) {

// Check horizontally for matches

for (int i = 0; i < gridSize; i++) {

int count = 1; // Count of consecutive matching gems

for (int j = 0; j < gridSize; j++) {

// Check for matches in different configurations (left-right)

if (gemMatrix[i][j] >= 7) {

if ((j <= 5 && gemMatrix[i][j] % 7 == gemMatrix[i][j + 1] && gemMatrix[i][j] % 7 == gemMatrix[i][j + 2])

||

(j >= 2 && gemMatrix[i][j] % 7 == gemMatrix[i][j - 1] && gemMatrix[i][j] % 7 == gemMatrix[i][j - 2])

||

(j != 0 && j != 7 && gemMatrix[i][j] % 7 == gemMatrix[i][j - 1] && gemMatrix[i][j] % 7 == gemMatrix[i][j + 1])

) {

// If a match is found, set those elements to -1 (representing removal)

for (int k = 0; k < gridSize; k++) {

gemMatrix[i][k] = -1;

}

// Update score for the match and return

for (int k = 0; k < gridSize; k++) {

gemMatrix[k][j] = -1;

}

score += 100;

return;

}

}

}

}

// Check vertically

for (int j = 0; j < gridSize; j++) {

int count = 1; // Count of consecutive matching gems

for (int i = 0; i < gridSize; i++) {

if (gemMatrix[i][j] >= 7) {

// Checking for various configurations of vertical gem matches (up-down)

if ((i <= 5 && gemMatrix[i][j] % 7 == gemMatrix[i + 1][j] && gemMatrix[i][j] % 7 == gemMatrix[i + 2][j])

||

(i >= 2 && gemMatrix[i][j] % 7 == gemMatrix[i - 1][j] && gemMatrix[i][j] % 7 == gemMatrix[i - 2][j])

||

(i != 0 && i != 7 && gemMatrix[i][j] % 7 == gemMatrix[i - 1][j] && gemMatrix[i][j] % 7 == gemMatrix[i + 1][j])

) {

// If a match is found, set those elements to -1 (representing removal)

for (int k = 0; k < gridSize; k++) {

gemMatrix[i][k] = -1;

}

for (int k = 0; k < gridSize; k++) {

gemMatrix[k][j] = -1;

}

// Update the score for the match and return from the function

score += 100;

return;

}

}

}

}

}

// Check for matches of four gems and update the score

void checkForFourMatches(int gemMatrix[gridSize][gridSize], int& score, int selectedX, int selectedY) {

for (int i = 0; i < gridSize; ++i) {

int count = 1; // Count of consecutive matching gems

for (int j = 1; j < gridSize; ++j) {

if (gemMatrix[i][j] == gemMatrix[i][j - 1] && gemMatrix[i][j] != -1) {

count++; // Increment the count for consecutive matching gems

}

else { // Reset the count if the sequence is broken

count = 1;

}

if (count == 4) {

score += 100;

gemMatrix[i][j] = gemMatrix[i][j] + 7;

for (int k = j - 1; k > j - 4; k--) {

gemMatrix[i][k] = -1;

}

}

}

}

// Check vertically

for (int j = 0; j < gridSize; ++j) {

int count = 1; // Count of consecutive matching gems

for (int i = 1; i < gridSize; ++i) {

if (gemMatrix[i][j] == gemMatrix[i - 1][j] && gemMatrix[i][j] != -1) {

// Increment the count for consecutive matching gems

count++;

}

else {

// Reset the count if the sequence is broken

count = 1;

}

// Check if we have three or more consecutive matching gems

if (count == 4) {

score += 100;

gemMatrix[i][j] = gemMatrix[i][j] + 7;

for (int k = j - 1; k > j - 4; k--) {

gemMatrix[i][k] = -1;

}

}

}

}

}

void checkForFiveMatches(int gemMatrix[gridSize][gridSize], int& score) {

for (int i = 0; i < gridSize; ++i) {

int count = 1; // Count of consecutive matching gems

for (int j = 1; j < gridSize; ++j) {

if (gemMatrix[i][j] == gemMatrix[i][j - 1] && gemMatrix[i][j] != -1) {

// Increment the count for consecutive matching gems

count++;

}

else {

// Reset the count if the sequence is broken

count = 1;

}

// Check if we have three or more consecutive matching gems

if (count == 5) {

generateGem(gemMatrix);

score += 300; // Increase score (adjust accordingly)

return;

}

}

}

// Check vertically

for (int j = 0; j < gridSize; ++j) {

int count = 1; // Count of consecutive matching gems

for (int i = 1; i < gridSize; ++i) {

if (gemMatrix[i][j] == gemMatrix[i - 1][j] && gemMatrix[i][j] != -1) {

// Increment the count for consecutive matching gems

count++;

}

else {

// Reset the count if the sequence is broken

count = 1;

}

// Check if we have three or more consecutive matching gems

if (count == 5) {

// Destroy the matching gems

generateGem(gemMatrix);

score += 300; // Increase score (adjust accordingly)

return;

}

}

}

}

//Function for generating new gems at empty spaces

void fillEmptySpaces(int gemMatrix[gridSize][gridSize]) {

for (int j = 0; j < gridSize; ++j) {

int k = gridSize - 1;

// Shift non-empty gems downward in the column

for (int i = gridSize - 1; i >= 0; --i) {

if (gemMatrix[i][j] != -1) {

gemMatrix[k--][j] = gemMatrix[i][j];

}

}

// Fill the remaining empty spaces with new gems

for (int i = k; i >= 0; --i) {

gemMatrix[i][j] = rand() % 7; // You can replace this with your logic for generating gem types

}

}

}

void displayScore(sf::RenderWindow& window, sf::Font& font, int score) {

// Creating a text object to display the score

Text scoreText;

scoreText.setFont(font); // Use default font

scoreText.setCharacterSize(20); // Set the character size to 20

scoreText.setFillColor(sf::Color::Red); // Set text color to red

scoreText.setPosition(gridSize \* cellSize + 10, 30); // Set the position on the window

scoreText.setString("Score => " + to\_string(score)); // Set the string to display the score

// Draw the score text onto the window

window.draw(scoreText);

}

void displayTimer(RenderWindow& window, sf::Font& font, int time) {

// Creating a text object to display the timer

Text timerText;

timerText.setFont(font); // Set the font to the provided font

timerText.setCharacterSize(20); // Set the character size to 20

timerText.setFillColor(Color::Blue); // Set text color to blue

timerText.setPosition(gridSize \* cellSize + 10, 130); // Set the position on the window

timerText.setString("Timer => " + to\_string(time) + "s"); // Set the string to display the time

// Draw the timer text onto the window

window.draw(timerText);

}

void displayRemarks(RenderWindow& window, Font& font, string remarks) {

// Creating a text object to display remarks

Text timerText;

timerText.setFont(font); // Set the font to the provided font

timerText.setCharacterSize(timerFontSize); // Set the character size to the specified timerFontSize

timerText.setFillColor(Color::White); // Set text color to white

timerText.setPosition(gridSize \* cellSize + 10, 500); // Set the position on the window

timerText.setString("Remarks: \n" + remarks); // Set the string to display remarks

// Draw the remarks text onto the window

window.draw(timerText);

}

// This function checks if there are possible horizontally swappable gems in the gemMatrix.

bool checkIfHorizontallySwapPossible(int gemMatrix[gridSize][gridSize]) {

int count = 1;

for (int i = 0; i < gridSize; i++) {

for (int j = 0; j < gridSize - 1; j++) {

if (gemMatrix[i][j] % 7 == gemMatrix[i][j + 1] % 7) {

count += 1;

if (count >= 3) {

return true;

}

}

else {

count = 1;

}

}

}

return false;

}

// This function checks if there are possible vertically swappable gems in the gemMatrix.

bool checkIfVerticallySwapPossible(int gemMatrix[gridSize][gridSize]) {

int count = 1;

for (int i = 0; i < gridSize; i++) {

for (int j = 0; j < gridSize - 1; j++) {

if (gemMatrix[j][i] % 7 == gemMatrix[j + 1][i] % 7) {

count += 1;

if (count >= 3) {

return true;

}

}

else {

count = 1;

}

}

}

return false;

}

// This function checks if a swap between two gems at specific coordinates in the gemMatrix is valid.

bool isValidSwap(int gemMatrix[gridSize][gridSize], int cursorX, int cursorY, int newCursorY, int newCursorX) {

// Swap the gems at the specified coordinates to check if a swap is possible

swap(gemMatrix[cursorY][cursorX], gemMatrix[newCursorY][newCursorX]);

// Variables to check if horizontal or vertical swaps are possible

bool isSwapPossible = false;

bool horizontalSwapPossible = checkIfHorizontallySwapPossible(gemMatrix);

bool verticalSwapPossible = checkIfVerticallySwapPossible(gemMatrix);

// Check if either horizontal or vertical swaps are possible

if (horizontalSwapPossible || verticalSwapPossible) {

isSwapPossible = true;

}

// Swap the gems back to their original positions

swap(gemMatrix[newCursorY][newCursorX], gemMatrix[cursorY][cursorX]);

// Return if a valid swap is possible

return isSwapPossible;

}

// This function displays the game end screen with the final score and a 'Quit' button.

void displayGameEnd(Font& font, int& score) {

// Create a window for the game end screen

RenderWindow window(VideoMode(gridSize \* cellSize + 200, gridSize \* cellSize), "Game Over!!");

// Text displaying 'Game Over'

Text gameEnd;

gameEnd.setFont(font);

gameEnd.setCharacterSize(50);

gameEnd.setFillColor(Color::Red);

gameEnd.setPosition(250, 180);

gameEnd.setString("Game Over");

// Text displaying the final score

Text scoreText;

scoreText.setFont(font);

scoreText.setCharacterSize(20);

scoreText.setFillColor(Color::Green);

scoreText.setPosition(350, 230);

scoreText.setString("Score : " + to\_string(score));

// Define dimensions and properties for the 'Quit' button

const int buttonWidth = 200;

const int buttonHeight = 50;

RectangleShape quitButton(Vector2f(buttonWidth, buttonHeight));

quitButton.setFillColor(Color::Cyan);

quitButton.setOutlineThickness(2);

quitButton.setPosition(250, 400);

// Text for the 'Quit' button

Text quitText("Quit", font, 30);

quitText.setFillColor(Color::Black);

quitText.setPosition((window.getSize().x - quitText.getLocalBounds().width) / 2,

quitButton.getPosition().y + (buttonHeight - quitText.getLocalBounds().height) / 2);

// Event loop for handling window events and button clicks

while (window.isOpen()) {

Event event;

while (window.pollEvent(event)) {

if (event.type == Event::Closed) {

window.close();

}

// Check for button clicks

if (event.type == Event::MouseButtonPressed) {

if (event.mouseButton.button == Mouse::Left) {

Vector2i mousePos = Mouse::getPosition(window);

// Check if Play button is clicked

if (quitButton.getGlobalBounds().contains(static\_cast<float>(mousePos.x), static\_cast<float>(mousePos.y))) {

// Quit button clicked

cout << "Play button clicked! Game starts..." << endl;

window.clear();

window.close();

break;

}

}

}

}

// Drawing elements on the window

window.clear();

window.draw(gameEnd);

window.draw(scoreText);

window.draw(quitButton);

window.draw(quitText);

window.display();

}

// Once the window is closed, display the main window again

displayMainWindow(font);

}

void showInstructions(Font& font) {

// Create a new window for instructions

RenderWindow window(VideoMode(800, 600), "Instructions");

Texture backgroundTexture;

if (!backgroundTexture.loadFromFile(INSTRUCT\_BACKGROUND)) {

// Handle error loading the background image

return;

}

// Create a sprite for the background image

Sprite backgroundSprite(backgroundTexture);

backgroundSprite.setScale(static\_cast<float>(window.getSize().x) / backgroundTexture.getSize().x,

static\_cast<float>(window.getSize().y) / backgroundTexture.getSize().y);

// Font for buttons

// Instructions text with line wrapping

Text instructionsText;

instructionsText.setFont(font);

instructionsText.setCharacterSize(20);

instructionsText.setFillColor(Color::White);

instructionsText.setPosition(20, 20);

// Set line spacing and wrap text

instructionsText.setLineSpacing(1.5);

instructionsText.setString(

"\n\n\n\n\nInstructions:\n"

"- Swap gems by clicking on two adjacent gems to match three or more of the same\n"

" type horizontally or vertically.\n"

"- Matching gems will be destroyed, and new gems will fall from above.\n"

"- Try to make matches to score points within the given time limit.\n"

"- Have fun playing the game!"

);

const int buttonWidth = 200;

const int buttonHeight = 50;

RectangleShape backButton(Vector2f(buttonWidth, buttonHeight));

backButton.setFillColor(Color::Cyan);

backButton.setOutlineThickness(2);

backButton.setPosition(250, 400);

Text backText("Back", font, 30);

backText.setFillColor(Color::Black);

backText.setPosition((window.getSize().x - backText.getLocalBounds().width) / 2,

backButton.getPosition().y + (buttonHeight - backText.getLocalBounds().height) / 2);

// Main loop for the instructions window

while (window.isOpen()) {

Event event;

while (window.pollEvent(event)) {

if (event.type == Event::Closed) {

window.close();

}

if (event.type == Event::MouseButtonPressed) {

if (event.mouseButton.button == Mouse::Left) {

Vector2i mousePos = Mouse::getPosition(window);

// Check if Play button is clicked

if (backButton.getGlobalBounds().contains(static\_cast<float>(mousePos.x), static\_cast<float>(mousePos.y))) {

cout << "Play button clicked! Game starts..." << endl;

window.clear();

window.close();

break;

}

}

}

}

// Clear the instructions window

window.clear();

// Draw instructions text

window.draw(backgroundSprite);

window.draw(instructionsText);

window.draw(backButton);

window.draw(backText);

// Display the contents of the instructions window

window.display();

}

displayMainWindow(font);

}

void showCreditsWindow(Font& font) {

// Create a new window for instructions

RenderWindow window(VideoMode(800, 600), "Credits");

Texture backgroundTexture;

if (!backgroundTexture.loadFromFile(INSTRUCT\_BACKGROUND)) {

// Handle error loading the background image

return;

}

// Create a sprite for the background image

Sprite backgroundSprite(backgroundTexture);

backgroundSprite.setScale(static\_cast<float>(window.getSize().x) / backgroundTexture.getSize().x,

static\_cast<float>(window.getSize().y) / backgroundTexture.getSize().y);

// Font for buttons

// Instructions text with line wrapping

Text creditsText;

creditsText.setFont(font);

creditsText.setCharacterSize(20);

creditsText.setFillColor(Color::White);

creditsText.setPosition(20, 20);

// Set line spacing and wrap text

creditsText.setLineSpacing(1.5);

creditsText.setString(

"\n\n\n\n\nCredits:\n"

"- Name: Uma e Rubab\n"

" Roll No: 23L-0928.\n"

"- Name: Ayesha Khalid.\n"

" Roll No: 23L-0667.\n"

);

const int buttonWidth = 200;

const int buttonHeight = 50;

RectangleShape backButton(Vector2f(buttonWidth, buttonHeight));

backButton.setFillColor(Color::Cyan);

backButton.setOutlineThickness(2);

backButton.setPosition(250, 400);

Text backButtonText("Back", font, 30);

backButtonText.setFillColor(Color::Black);

backButtonText.setPosition((window.getSize().x - backButtonText.getLocalBounds().width) / 2,

backButton.getPosition().y + (buttonHeight - backButtonText.getLocalBounds().height) / 2);

// Main loop for the instructions window

while (window.isOpen()) {

Event event;

while (window.pollEvent(event)) {

if (event.type == Event::Closed) {

window.close();

}

if (event.type == Event::MouseButtonPressed) {

if (event.mouseButton.button == Mouse::Left) {

Vector2i mousePos = Mouse::getPosition(window);

// Check if Play button is clicked

if (backButton.getGlobalBounds().contains(static\_cast<float>(mousePos.x), static\_cast<float>(mousePos.y))) {

cout << "Play button clicked! Game starts..." << endl;

window.clear();

window.close();

break;

}

}

}

}

// Clear the instructions window

window.clear();

// Draw instructions text

window.draw(backgroundSprite);

window.draw(creditsText);

window.draw(backButton);

window.draw(backButtonText);

// Display the contents of the instructions window

window.display();

}

displayMainWindow(font);

}