**Beschreibung der GameBoard-Klasse (Maze- und Spiellogik)**

Die Klasse GameBoard bildet eine zentrale Komponente der Spielarchitektur. Sie verwaltet das Spielfeld, das Labyrinth-Layout, die Verteilung von Münzen und – optional – Sonderobjekte wie Früchte. Dabei stellt sie sowohl die zugrunde liegende Datenstruktur des Spielfelds bereit als auch Funktionen zur Initialisierung, Manipulation und Verwaltung des Spielfeldzustands.

Das Spielfeld selbst wird durch ein zweidimensionales Vektor-Array field vom Typ Cell repräsentiert. Jede Zelle entspricht einem logischen Zustand (z. B. Wand, Münze), wobei die Cell-Typen zentral in der Datei Constants.h definiert sind. Diese Trennung zwischen Logik und Darstellung gewährleistet eine saubere Entkopplung und erleichtert die Wartung des Codes.

Eine der zentralen Methoden ist generateRandomMap(), die mithilfe eines rekursiven Tiefensuchalgorithmus (DFS, Depth-First Search) ein prozedural generiertes Labyrinth erstellt. Dieser Algorithmus operiert auf einem temporären Gitter aus MazeCell-Objekten und erzeugt ein sogenanntes *perfektes Maze* – ein Labyrinth, in dem es genau einen eindeutigen Pfad zwischen zwei Punkten gibt und keine isolierten Bereiche entstehen. Der DFS-Algorithmus beginnt bei einer Startzelle und erkundet rekursiv benachbarte Zellen, wobei Wände entfernt und Wege geschaffen werden.

Für die Variation der Labyrinthstruktur wird in jeder Rekursion die Richtungsreihenfolge mithilfe der Methode shuffleDirections() zufällig neu gemischt. Dadurch entstehen bei jedem Spielstart neue, einzigartige Labyrinthe, was die Wiederspielbarkeit deutlich erhöht.

Nach der Maze-Erstellung wird das eigentliche Spielfeld (field) mit Wänden und Münzen initialisiert. Dabei gelten klare Regeln: Begehbare Felder erhalten automatisch Münzen, während Wände bestehen bleiben. Zur Verbesserung der Spielbarkeit und zur Reduktion unnötiger Sackgassen werden im Nachgang gezielt zusätzliche Verbindungen – sogenannte *extra connections* – eingefügt. Diese entstehen durch das Entfernen einzelner Wände, sofern dadurch sinnvolle neue Verbindungen geschaffen werden (z. B. zwischen zwei offenen Horizontalfeldern).

Darüber hinaus verwaltet die GameBoard-Klasse weitere spielrelevante Informationen wie die Anzahl verbleibender Münzen (coinsLeft) sowie die Position und Existenz eines optionalen Früchteobjekts (fruitPresent, fruitX, fruitY). Dadurch übernimmt sie eine zentrale Rolle bei der Spiellogik und dem Spielziel (alle Münzen einsammeln).

Die Methode getGhostStartPositions() liefert eine Liste fester Koordinaten zurück, an denen gegnerische Spielfiguren (Geister) zu Spielbeginn erscheinen. Diese Positionen werden mithilfe eines static-Vektors einmalig initialisiert und effizient wiederverwendet.

Aus objektorientierter Sicht ist die GameBoard-Klasse gut gekapselt: Die Spiellogik ist klar von der Darstellung getrennt, interne Methoden wie dfs() und shuffleDirections() sind privat deklariert, wodurch eine saubere Schnittstelle gewährleistet wird. Die Verwendung separater Datentypen (MazeCell, Cell) fördert zudem Wartbarkeit und Erweiterbarkeit.

Die Architektur dieser Klasse wurde gezielt auf Modularität, Testbarkeit und Flexibilität ausgelegt. Änderungen an der Maze-Logik oder dem Spielfeldaufbau können einfach vorgenommen werden, ohne andere Teile des Codes zu beeinflussen. Die Kombination aus objektorientierter Struktur und prozeduraler Erzeugung ergibt eine performante und robuste Spiellogik, die auch bei häufig wechselnden Karten zuverlässig funktioniert.

Verbesserungen für die Architektur:

* Komposition statt Monolith: Zur besseren Strukturierung sollten verschiedene Spielfeldelemente wie Power-Ups, Hindernisse oder Spezialfelder als eigenständige Klassen modelliert und über Komposition in das GameBoard integriert werden. So können neue Elementtypen einfach ergänzt werden, ohne die GameBoard-Klasse selbst stark zu verändern.
* CellFactory und CellManager: Die Erzeugung von Zellen kann in eine separate *Factory*-Klasse ausgelagert werden, die verschiedene Zelltypen erzeugt. Zusätzlich empfiehlt sich ein *CellManager*, der das interne 2D-Array verwaltet und kontrollierten Zugriff über Methoden wie at() oder über Iteratoren bietet. Dadurch wird die interne Struktur gekapselt und besser wartbar.
* Designmuster für saubere Struktur: Der Einsatz von Iteratoren oder das *Visitor-Pattern* erlaubt es, Operationen wie Rendering, Kollisionsprüfung oder Updates modular auf die Zellen anzuwenden – ohne die interne Spielfeldstruktur offenzulegen. Das erhöht Wiederverwendbarkeit und Testbarkeit einzelner Komponenten.
* Maze-Erzeugung entkoppeln: Auch die Tiefensuche zur Maze-Generierung sollte nicht direkt auf das field zugreifen, sondern den CellManager verwenden, um Zellen zu markieren und Wände zu entfernen. Dadurch bleibt die Spielfeldstruktur flexibel und modular.

**Beschreibung der Leaderboard-Klasse (Highscore-Verwaltung)**

Die Leaderboard-Klasse ist für die Verwaltung und dauerhafte Speicherung der Highscores im Spiel zuständig. Sie speichert eine Liste der besten Spielergebnisse samt Spielernamen und stellt Methoden bereit, um diese zu laden, zu speichern und bei Bedarf zu aktualisieren. Dadurch bildet sie die Schnittstelle zwischen der Spiellogik und einer externen Textdatei, in der die Highscores persistent abgelegt werden.

Zentrales Element der Klasse ist ein Vektor von Paaren, bestehend aus einem int-Wert für den Punktestand und einem std::string für den Namen des Spielers. Dieser Vektor – entries – enthält die aktuellen Top-Ergebnisse. Beim Erstellen eines Leaderboard-Objekts wird eine Datei übergeben, die als Ziel für die Lese- und Schreibvorgänge dient. Die Methode load() wird direkt im Konstruktor aufgerufen und lädt beim Programmstart alle vorhandenen Highscores aus dieser Datei.

Die Methode tryUpdateHighscore() übernimmt die Aufgabe, ein neues Spielergebnis in die bestehende Liste einzupflegen. Dabei wird zunächst geprüft, ob der Name bereits existiert. Ist das der Fall, wird der vorhandene Eintrag nur dann überschrieben, wenn der neue Punktestand höher ist als der bisherige. Ist der Name hingegen neu, wird das Ergebnis einfach ergänzt. Im Anschluss wird die gesamte Liste absteigend nach Punktzahl sortiert und auf die zehn besten Einträge gekürzt, um Übersichtlichkeit zu wahren und ein unkontrolliertes Anwachsen der Datei zu verhindern.

Nach jeder erfolgreichen Aktualisierung wird die neue Highscore-Liste über die Methode save() zurück in die Datei geschrieben. Die Einträge werden dabei zeilenweise gespeichert, wobei jede Zeile aus einem Spielernamen und dem zugehörigen Punktestand besteht. Dieses einfache Format ermöglicht eine schnelle und unkomplizierte Dateiverarbeitung.

Die Methoden getHighscore() und getHighscoreName() erlauben den direkten Zugriff auf das aktuell beste Spielergebnis bzw. den führenden Spielernamen. Ist die Tabelle leer, geben beide Methoden sinnvolle Standardwerte zurück, beispielsweise "---" für den Namen.

Die Leaderboard-Klasse ist bewusst kompakt und modular aufgebaut. Durch die klare Trennung von Lade-, Speicher- und Verwaltungslogik bleibt der Code gut wartbar und lässt sich bei Bedarf leicht erweitern. Auch ein Wechsel des Speicherformats – etwa auf JSON oder eine Datenbankanbindung – kann vorgenommen werden, ohne dass die Spiellogik oder Benutzeroberfläche davon betroffen wäre.

Für eine noch sauberere Modularisierung wäre es sinnvoll, die Ein- und Ausgabe strikt voneinander zu trennen. So könnte man beispielsweise separate Klassen für das Lesen und Schreiben der Highscore-Daten sowie für deren Darstellung in der Konsole oder in einer GUI einführen. Dadurch ließen sich die Verantwortlichkeiten klar aufteilen: Eine Klasse wäre ausschließlich für die Datenhaltung zuständig, während eine andere sich um die Benutzerinteraktion kümmert. Das erleichtert nicht nur die Wartung, sondern auch zukünftige Erweiterungen – etwa die Unterstützung eines neuen Dateiformats oder einer webbasierten Oberfläche.

Darüber hinaus empfiehlt sich die Einführung einer eigenen Klasse für einzelne Highscore-Einträge, anstatt mit einem std::pair oder einer einfachen struct zu arbeiten. Eine solche LeaderboardEntry-Klasse könnte neben den Datenfeldern auch Methoden zur Validierung, zum Vergleich oder zur Formatierung enthalten. Dadurch wird die Struktur objektorientierter, klarer und besser testbar, da relevante Logik direkt an den zugehörigen Datenobjekten gekapselt wird.

Um die Flexibilität weiter zu erhöhen, bietet sich ein Entwurf nach dem Prinzip des Model-View-Controller (MVC) an. Die Leaderboard-Klasse würde dann ausschließlich die Daten verwalten – also Einfügen, Aktualisieren und Speichern übernehmen. Für die Darstellung wäre eine separate View-Komponente zuständig. Zudem ließe sich für die Speicherlogik ein Strategie-Muster verwenden, das alternative Speicherarten erlaubt, ohne die Hauptlogik zu verändern. So könnten Highscores künftig nicht nur in Textdateien, sondern auch in Datenbanken oder über das Netzwerk gespeichert werden.

**Beschreibung der Renderer-Klasse (Darstellung und Grafik-Handling)**

Die Renderer-Klasse ist für die vollständige visuelle Darstellung sämtlicher Spielelemente verantwortlich. Dazu gehören das Spielfeld, die Spielfiguren (Pac-Man und die Geister), die Benutzeroberfläche sowie sämtliche Menüs wie Startbildschirm, Pausemenü, Game-Over- und Highscore-Anzeige. Hier wird stark auf Methoden der Raylib Bibliothek für eine optisch ansprechendere Darstellung zurückgegriffen.

Die Entscheidung für eine rein statische Klasse erfolgte bewusst: Da die Rendering-Funktionalität keinen internen Zustand benötigt, der sich zwischen Instanzen unterscheidet, sind statische Methoden ideal geeignet. Sie ermöglichen einen globalen, einfachen Zugriff auf alle Darstellungsfunktionen, ohne dass Objekte instanziiert werden müssen. Diese Architektur unterstützt eine klare Trennung von Spiellogik und visueller Darstellung und vereinfacht den Einsatz des Renderers im gesamten Projekt.

Ein zentrales Prinzip des Designs ist die strikte Trennung von Darstellung und Spielzustand. Der Renderer erhält die erforderlichen Informationen über konstante Referenzen oder Kopien von Objekten wie GameBoard, Player, Ghost oder Leaderboard, greift jedoch nicht in deren Zustand ein.

Zu Beginn des Programms lädt die Methode init() notwendige Ressourcen – etwa Texturen für Pac-Man – in den Speicher, um Ladeverzögerungen zur Laufzeit zu vermeiden. Beim Beenden des Spiels oder Szenenwechsel sorgt die Methode unload() für die korrekte Freigabe dieser Ressourcen, um Speicherlecks zu verhindern.

Die zentrale Methode drawGame() übernimmt die Darstellung des laufenden Spiels. Sie beginnt mit dem Zeichnen des Spielfelds, wobei alle Zellen des GameBoard durchlaufen und in Abhängigkeit ihres Typs (Wand, Münze, Frucht) entsprechende Grafiken gerendert werden. Die Positionierung ist dynamisch und orientiert sich an der aktuellen Fenstergröße. Zellgrößen und Offsets werden automatisch berechnet, um das Spielfeld stets zentriert anzuzeigen – unabhängig von der Bildschirmauflösung.

Anschließend werden die Geister – dargestellt als lila Kreise – sowie Pac-Man – mithilfe einer vordefinierten Textur und DrawTexturePro() – in korrekter Position und Skalierung gezeichnet. Die Benutzeroberfläche (UI) wird ebenfalls innerhalb von drawGame() gerendert und informiert den Spieler über Punktestand, verbleibende Münzen sowie den aktuellen Highscore mit Spielernamen. Diese Anzeigen sind fix positioniert und bleiben unabhängig vom Spielgeschehen sichtbar.

Weitere Bildschirminhalte wie Startmenü, Pausemenü, Game-Over-Szene und Highscore-Anzeige werden über dedizierte Methoden wie drawStartMenu(), drawPauseMenu(), drawGameOver() und drawLeaderboard() umgesetzt. Jede dieser Methoden beginnt mit BeginDrawing() und ClearBackground() und visualisiert die jeweiligen Inhalte in abgestimmtem Layout und Design. Texte und Überschriften sind dabei zentriert ausgerichtet und folgen einem einheitlichen Farbschema zur klaren Benutzerführung.

Die Methode drawLeaderboard(), welche zur Laufzeit Highscore-Daten aus einer Datei einliest und die zehn besten Einträge anzeigt. Sollte die Datei nicht verfügbar sein, wird eine entsprechende Fehlermeldung ausgegeben – ein einfacher, aber effektiver Mechanismus zur Erhöhung der Robustheit.

Die Umsetzung folgt durchgehend objektorientierten Prinzipien. Die Renderer-Klasse ist ausschließlich für die Darstellung zuständig, während Spiellogik, Datenhaltung und Zustandsverwaltung klar in anderen Komponenten organisiert sind. Die statische Struktur vermeidet unerwünschte Seiteneffekte und sorgt für Übersichtlichkeit im Code.

Dennoch ist die aktuelle Implementierung stark monolithisch aufgebaut. Die Darstellungslogik für Spielfeld, Spielfiguren, Benutzeroberfläche und Menüs ist zentral in einer einzigen Klasse gebündelt. Dadurch entstehen umfangreiche, wenig modulare Methoden, was Wartung und Erweiterung – etwa durch neue Spielobjekte oder alternative Darstellungsformen – erschwert.

Ein möglicher nächster Entwicklungsschritt wäre die Aufteilung der Renderer-Logik in spezialisierte Komponenten: etwa einen BoardRenderer für das Spielfeld, einen EntityRenderer für Spieler und Gegner sowie einen UIRenderer für Menüs und Statusanzeigen. Ergänzend könnte ein Renderable-Interface eingeführt werden, das von allen darstellbaren Objekten implementiert wird. Diese Objekte kapseln ihre eigene Darstellungslogik, während ein zentraler Renderer lediglich koordinierend agiert.

Auch das Ressourcenmanagement, insbesondere für Texturen, sollte ausgelagert werden – idealerweise in eine dedizierte TextureManager-Klasse. Dadurch ließe sich die Renderer-Klasse weiter entlasten und die Verwaltung von Assets zentralisieren. Darüber hinaus empfiehlt es sich, die Steuerung von BeginDrawing() und EndDrawing() in eine zentrale Koordinationsinstanz auszulagern. Einzelne Komponenten sollten ausschließlich Zeichnungsbefehle absetzen, nicht jedoch den Rendering-Zyklus selbst steuern.

Diese Modularisierung würde zu einer klareren, besser wartbaren und testbaren Codebasis führen. Sie schafft die Grundlage für zukünftige Erweiterungen – etwa neue Spielfunktionen, alternative Render-Backends oder visuelle Optimierungen – ohne die bestehende Struktur zu belasten.

**main.cpp – Einstiegspunkt des Pacman-Spiels**

Die Datei main.cpp bildet den Einstiegspunkt der Anwendung und ist für die Initialisierung des Spiels verantwortlich. Sie übernimmt ausschließlich die Aufgaben des Setups und des Starts der Hauptspielschleife, wodurch sie bewusst minimal gehalten ist, und die eigentliche Spiellogik auslagert.

Zu Beginn der main()-Funktion wird der Zufallsgenerator mit srand() anhand der aktuellen Systemzeit initialisiert. Dies stellt sicher, dass spätere zufällige Elemente im Spiel – wie etwa die zufällige Labyrinthgenerierung oder das Spawnverhalten der Geister – bei jedem Spielstart unterschiedlich ausfallen.

Im Anschluss wird eine Instanz der zentralen Game-Klasse erzeugt. Diese Klasse kapselt den vollständigen Spielablauf, inklusive Zustandsverwaltung, Spiellogik und Steuerung der Darstellung. Durch den Aufruf von game.run() wird die Hauptspielschleife gestartet. Diese Schleife übernimmt zyklisch:

* die Verarbeitung von Benutzereingaben,
* die Aktualisierung des Spielzustands,
* und das Rendering aller visuellen Komponenten.

Dank dieser klaren Aufgabentrennung bleibt main.cpp übersichtlich und wartungsfreundlich. Die Datei konzentriert sich ausschließlich auf den Lebenszyklus der Anwendung – vom Initialisieren bis zum Starten des Spiels. Die Spiellogik bleibt vollständig in der Game-Klasse gekapselt.

**MazeCell.h – Definition der Zelle im Labyrinth**

Die Struktur MazeCell repräsentiert eine einzelne Zelle des Spielfeld-Labyrinths. Jede Zelle besitzt zwei boolesche Attribute:

* visited zeigt an, ob die Zelle bereits bei der Labyrinthgenerierung besucht wurde,
* wall gibt an, ob die Zelle als Wand definiert ist.

Der Standardkonstruktor initialisiert eine Zelle so, dass sie zunächst als Wand (wall = true) gilt und noch nicht besucht wurde (visited = false). Diese einfache Datenstruktur bildet das Labyrinth als Raster aus Zellen ab und ermöglicht eine effiziente Steuerung sowohl der Generierung als auch der Darstellung.

Durch die Verwendung einer simplen struct bleiben Speicherverbrauch und Zugriffsgeschwindigkeit minimal, was gerade bei der dynamischen Verarbeitung großer Spielfeldbereiche von Vorteil ist.

Aktuell ist MazeCell jedoch sehr einfach gehalten und beschränkt sich auf boolesche Zustände. Dies kann problematisch werden, wenn neue Zellinhalte wie verschiedene Fruchttypen, Power-Ups oder andere Spielobjekte ergänzt werden sollen. Jedes weitere Feature würde zusätzliche Flags oder Anpassungen an der MazeCell-Struktur erfordern, was den Code mit der Zeit unübersichtlich und schwer wartbar macht.

Eine zukunftsfähigere Lösung besteht darin, das State-Pattern oder Kompositionsansätze zu verwenden. Beispielsweise könnte jede Zelle statt einfacher Flags einen Zeiger oder eine Referenz auf ein abstraktes CellContent-Objekt halten. CellContent wäre dabei eine Basisklasse, die durch konkrete Subklassen wie Wall, Dot, Fruit, PowerUp etc. erweitert wird. So lassen sich neue Zelltypen modular hinzufügen, ohne die MazeCell selbst zu verändern. Dies steigert die Flexibilität, unterstützt Wiederverwendbarkeit und verbessert die Wartbarkeit des Codes erheblich.