**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.

Verbesserungsvorschläge 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, statt lediglich 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.

In der aktuellen Implementierung basiert die Speicherung noch auf einem std::vector<std::pair<int, std::string>>, was jedoch durch eine eigene Entry-Klasse ersetzt werden sollte, um Struktur und Erweiterbarkeit zu verbessern. Die Sortierung erfolgt korrekt nach Score in absteigender Reihenfolge, und die maximale Eintragsanzahl ist auf zehn begrenzt. Auch wenn die Dateioperationen derzeit direkt erfolgen, ließen sie sich durch eine abstrahierte Speicherstrategie problemlos austauschen.

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

Die Renderer-Klasse übernimmt in meiner Softwarearchitektur die vollständige Verantwortung für die visuelle Darstellung sämtlicher Elemente des Spiels. Dazu zählen das Spielfeld, die Spielfiguren (Pac-Man und die Geister), die Benutzeroberfläche sowie sämtliche Menüs wie Start-, Pause-, Game-Over- und Highscore-Bildschirm.

Ich habe mich bewusst für eine statische Klasse entschieden, da die Rendering-Funktionalität keinen internen Zustand erfordert, der sich zwischen Instanzen unterscheiden würde. Die Nutzung statischer Methoden ermöglicht einen globalen Zugriff auf die Darstellungsfunktionen, ohne dass ein Objekt erzeugt werden muss. Diese Designentscheidung vereinfacht den Einsatz des Renderers und führt zu einer klaren Trennung zwischen Spiellogik und Grafikdarstellung.

Ein zentrales Prinzip der Architektur ist die strikte Trennung zwischen Darstellung und Spielzustand. Der Renderer erhält alle notwendigen Informationen über Referenzen oder Kopien von Spielobjekten wie GameBoard, Player, Ghost oder Leaderboard, verändert diese jedoch nicht. Dieses Vorgehen entspricht dem Single-Responsibility-Prinzip und trägt entscheidend zur Wartbarkeit und Erweiterbarkeit des Codes bei.

Beim Programmstart wird in der Methode init() die Textur für Pac-Man aus den Assets geladen, um Ladeverzögerungen während des Spiels zu vermeiden. Die Methode unload() sorgt dafür, dass diese Textur beim Beenden des Programms oder beim Szenenwechsel korrekt freigegeben wird, was Speicherlecks verhindert.

Die Hauptmethode drawGame() ist für die Darstellung des laufenden Spiels zuständig. Sie zeichnet zunächst das Spielfeld, indem sie alle Zellen des GameBoard durchläuft. Je nach Zelltyp (Wand, Münze, Frucht) wird eine entsprechende Grafik verwendet: Wände erscheinen als blaue Rechtecke, Münzen und Früchte als farbige Kreise. Die Positionierung ist dynamisch berechnet und richtet sich nach der aktuellen Fenstergröße. Zellgröße und Offsets werden so bestimmt, dass das Spielfeld stets zentriert angezeigt wird – unabhängig von der Auflösung.

Anschließend folgen die Geister, die als lila Kreise dargestellt werden, ebenfalls relativ zur Zellgröße positioniert. Pac-Man selbst wird mithilfe einer zuvor geladenen Textur mit DrawTexturePro() gezeichnet, um eine exakte Platzierung und Skalierung zu gewährleisten.

Auch die Benutzeroberfläche wird in drawGame() mitgerendert. Sie zeigt aktuelle Informationen wie Punktestand, verbleibende Münzen sowie den Highscore mitsamt Spielernamen an. Diese Informationen erscheinen als Text an festen Positionen und bleiben somit stets sichtbar.

Für die übrigen Bildschirme stellt der Renderer ebenfalls spezialisierte Methoden bereit: drawStartMenu(), drawPauseMenu(), drawLeaderboard() und drawGameOver(). Jede dieser Methoden beginnt mit BeginDrawing() und ClearBackground(), gefolgt von der Ausgabe der jeweiligen Inhalte. Überschriften, Texte und Listen sind zentriert ausgerichtet und farblich abgestimmt, um eine klare Benutzerführung zu ermöglichen.

Besonders hervorzuheben ist drawLeaderboard(), das die Highscore-Daten zur Laufzeit aus einer Datei liest und die zehn besten Einträge anzeigt. Sollte die Datei nicht verfügbar sein, wird eine Fehlermeldung ausgegeben – ein einfacher, aber effektiver Mechanismus zur Erhöhung der Robustheit.

Die Trennung der Zuständigkeiten innerhalb des Programms folgt konsequent objektorientierten Prinzipien. Der Renderer kümmert sich ausschließlich um die grafische Darstellung, während andere Klassen Spiellogik, Datenhaltung und Zustandsverwaltung übernehmen. Die statische Umsetzung der Klasse vermeidet dabei unerwünschte Nebeneffekte durch Zustandsänderungen und erhöht die Übersichtlichkeit.

Trotz der funktionalen Umsetzung ist die aktuelle Struktur der Renderer-Klasse noch stark monolithisch. Alle Darstellungsaufgaben für Spielfeld, Spieler, Gegner, UI und Menüs sind in einer einzigen Klasse gebündelt, meist in großen, wenig granularen Methoden. Das erschwert nicht nur die Wartung, sondern auch die Erweiterung – etwa beim Hinzufügen neuer Spielobjekte oder Darstellungsarten.

Eine sinnvolle Weiterentwicklung wäre die Aufteilung des Renderers in spezialisierte Komponenten. So könnte etwa ein BoardRenderer für das Spielfeld, ein EntityRenderer für Spieler und Gegner sowie ein UIRenderer für Menü- und Statusanzeigen zuständig sein. Ergänzend ließe sich ein Renderable-Interface einführen, das von allen darstellbaren Objekten implementiert wird. Diese kapseln dann ihre eigene Darstellungslogik, während der zentrale Renderer lediglich die entsprechenden Methoden aufruft.

Auch das Ressourcenmanagement – insbesondere für Texturen – sollte künftig von einer eigenen TextureManager-Klasse übernommen werden. Damit würde der Renderer entlastet und Ressourcen könnten zentral verwaltet werden. Zudem empfiehlt es sich, den Aufruf von BeginDrawing() und EndDrawing() auf eine zentrale Steuerung zu beschränken. Einzelne Komponenten sollten nur Zeichnungsbefehle absetzen, nicht aber selbst den Zeichenprozess starten oder beenden.

Durch diese Modularisierung entsteht ein übersichtlicherer, leichter wart- und testbarer Code, der sich flexibel erweitern lässt – sei es um neue Spielfeatures, alternative Render-Backends oder optimierte Darstellungsformen.

**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, was eine gute Modularität und Nachvollziehbarkeit der Programmstruktur fördert.

**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.