Projekt-Dokumentation: Der Ritter der verlorenen Seelen

Projekttitel: Der Ritter der verlorenen Seelen

Plattform: Java (Swing/AWT)

Abgabetermin: 15.12.2025

Modul: Programmieren 1 – Prof. Dr. Jan Rexilius

Gruppe: Marvin Sandmeier, Marc Mittelstedt, Ralf Schmidt, Johann Issa

Diese Dokumentation hält den inkrementellen Aufbau des Spiels fest und kommentiert die wichtigsten architektonischen und technischen Entscheidungen für die Bereiche **(a) GUI & Leveldesign** und **(b) Spiellogik** des Aufgabenblattes.

Teil 1: GUI und Leveldesign (Aufgabe a)

Dieser Abschnitt beschreibt die Implementierung der grafischen Benutzeroberfläche, der visuellen Rückmeldungen und der Levelstruktur.

A.1. Grafische Darstellung und Zustandsverwaltung

Die Darstellung der Szene erfolgt kontinuierlich, wobei klare visuelle Rückmeldungen über den Spielzustand gegeben werden.

> **Verwendeter Prompt:** "Definiere in Java Swing eine Hauptklasse `GamePanel` als `JPanel` mit einem `Timer` für eine konstante Game-Loop (ca. 60 FPS). Nutze die Trennung in `paintComponent()` (Zeichnen) und `actionPerformed()` (Logik-Update) und implementiere ein Zustandssystem (`GameState`) für Menü und Spiel."

- **Designentscheidung (GUI):** Die zentrale Klasse `GamePanel` sorgt mit ihrem `Timer` (16 ms Verzögerung) für die **kontinuierlich aktualisierte Szene**. Die Zeichnung (`paintComponent`) ist strikt von der Logik (`actionPerformed`) getrennt, um eine stabile Darstellung zu gewährleisten.
- **Technische Notiz (Zustände):** Das Spiel verwendet die `GameState`-Enumeration zur Steuerung der **visuellen Zustände** ("Menu Screen", "Spiel startet", "Level Complete", "Game Over").
- **Technische Notiz (HUD):** Die Anzeige (`drawHUD`) im `GamePanel` zeigt kontinuierlich die **Punkteanzeige** und die verbleibenden Leben. Spielfigur und Gegner werden als grafische Objekte animiert und aktualisiert.

A.2. Leveldesign und Kameraführung

Die Levelstruktur unterstützt das Gameplay durch Abwechslung und eine an den Spieler angepasste Kamera.

> **Verwendeter Prompt:** "Implementiere eine Methode in `Level.java` zur Generierung von Leveln. Füge eine Logik für eine Kamera hinzu, die dem Spieler sanft folgt, aber an den Level-Grenzen stoppt."

- **Designentscheidung (Level-Anforderung):** Das Spiel verfügt über **8 Levels** (Index 0 bis 7). Die Levels unterscheiden sich im Layout (Plattformen, Rampen) und durch neue Objekte wie bewegliche Plattformen und unterschiedliche Gegnertypen.
- **Designentscheidung (Levelwechsel):** Der **Levelwechsel** erfolgt entweder **automatisch** durch das Erreichen der Endflagge oder **manuell** über die Levelauswahl im Menü.
- **Technische Notiz (Kamera-System):** Die Kamera-Verschiebung (`camX`) im `GamePanel` zentriert den Spieler horizontal, wird jedoch an den Levelgrenzen begrenzt. Dadurch bleibt der Levelanfang bzw. das Levelende sichtbar und es entstehen keine leeren Bereiche.
- **Technische Notiz (Parallax-Scrolling):** Ein Parallax-Effekt (`parallaxShift = camX / 2`) in `Level.draw()` erzeugt visuelle Tiefe, indem sich der Hintergrund langsamer bewegt als die Vordergrund-Elemente.
- **Designentscheidung (Manuelles Leveldesign):** Alle Levels wurden **vollständig manuell entworfen und iterativ getestet**, da automatisch generierte oder KI-vorgeschlagene Level häufig unlösbar, zu simpel oder spielmechanisch unausgewogen waren.

Teil 2: Spiellogik (Aufgabe b)

Dieser Abschnitt dokumentiert die Implementierung der Spielmechanik, der Steuerung und der objektorientierten Prinzipien.

B.1. Steuerung und Interaktion

Die Benutzerinteraktion ist vollständig tastaturbasiert umgesetzt.

> **Verwendeter Prompt:** "Erstelle einen `MenuManager`, der auf `GameState` reagiert und dynamische Buttons für Level-Auswahl generiert. Implementiere die Steuerung über WASD/Pfeiltasten mittels `InputMap` und `ActionMap`."

- **Designentscheidung (Steuerung):** Die Spielfigur wird über **WASD oder Pfeiltasten** gesteuert, der Sprung erfolgt über die **Leertaste**. Jede Eingabe führt zu einer klaren und unmittelbaren Wirkung im Spiel.
- **Technische Notiz (Input Handling):** Die Nutzung von **`InputMap` und `ActionMap`** ermöglicht eine saubere, konfliktfreie Verarbeitung von Tasteneingaben unabhängig vom Fokus einzelner GUI-Komponenten.
- **Designentscheidung (Spielziel & Ablauf):** Das Spiel ist über das Hauptmenü eigenständig startbar. Ziel ist das **Sammeln von Punkten**, das Besiegen von Gegnern und das Erreichen der Endflagge eines Levels.

B.2. Objektorientierte Prinzipien und Ereignisverarbeitung

Zur Einhaltung der OOP-Anforderungen werden klare Klassenstrukturen und Verantwortlichkeiten verwendet.

> **Verwendeter Prompt:** "Implementiere die Gegner-Kollisionslogik im `GamePanel`, wobei der Spieler nur dann Punkte erhält und abprallt (`player.bounceAfterStomp()`), wenn er fällt (`player.isFalling()`) und seine Y-Position über der des Gegners liegt."

- **Designentscheidung (Klassenhierarchie):** Die Basisklasse `Tile` repräsentiert alle Level-Elemente. Spezialisierungen wie `MovingPlatform` erben davon. Durch **Kapselung** (z. B. `getRect()`) bleibt die Kollisionslogik von der Darstellung getrennt.
- **Designentscheidung (Ereignisverarbeitung):** Die Methode `handleEnemyCollision` verarbeitet das Ereignis **Kollision** zentral.
  - **Logik-Priorisierung:** Vertikale Kollisionen (Stomp → Punkte, Gegner entfernen) haben Vorrang vor horizontalen Kollisionen (Schaden → Leben verlieren).
- **Designentscheidung (Spielzustände):** Startbedingung ist der Klick auf „Start“ im Menü. Endbedingungen sind entweder das Erreichen der Flagge („Level Complete“) oder der Verlust aller Leben („Game Over“).

B.3. Erweiterte Logik und Persistenz

Zusätzliche Komplexität entsteht durch physikbasierte Objekte und ein Speichersystem.

> **Verwendeter Prompt:** "Schreibe eine Java-Klasse `MovingPlatform`, die sich entlang eines Vektors (Start-X/Y zu End-X/Y) bewegt. Nutze Vektorprojektion, um die Richtung bei Erreichen der Grenzen umzukehren. Implementiere eine Klasse `Storage` zum Speichern von Highscores für mehrere Levels."

- **Designentscheidung (Bewegliche Plattformen):** Die Klasse `MovingPlatform` verändert in ihrer `update`-Methode die geerbten Koordinaten (`super.x`, `super.y`) anhand von Fließkommawerten. Dadurch bleibt die Kollisionsbox stets synchron zur Bewegung.
- **Technische Notiz (Physik):** Die Bewegungsumkehr erfolgt mithilfe von **Vektorprojektion**:

$$proj = \frac{(\vec{P} - \vec{A}) \cdot (\vec{B} - \vec{A})}{|\vec{B} - \vec{A}|^2}$$

- **Designentscheidung (Persistenz):** Die Klasse `Storage` speichert Highscores pro Level sowie einen aggregierten `totalScore` und unterstützt damit das Punktesystem als zentrales Spielziel.

- **Technische Notiz (Stabilität):** In der Game-Loop (`actionPerformed`) werden Kopien von Listen (`new ArrayList<>(enemies)`) verwendet, um gleichzeitige Iteration und Modifikation zu vermeiden.

Ergänzende Hinweise zur Eigenleistung und Iteration

- **Grafiken & Assets:** Sämtliche grafischen Inhalte des Spiels (Spielfigur, Gegner, Tiles, Plattformen, Hintergründe, HUD-Elemente) wurden **vollständig selbst gepixelt**. Es wurden **keine externen Asset-Pakete oder Vorlagen** verwendet.
- **Iterative Entwicklung & Bugfixing:** Die Entwicklung erfolgte strikt **inkrementell**. Nach nahezu jedem größeren Implementierungsschritt traten kleinere oder größere Fehler auf, die analysiert und behoben wurden (z. B. Kollisionen, Kamerabegrenzung, Plattformverhalten, Zustandswechsel). Diese kontinuierlichen Bugfixes führten zu einer stabilen und konsistenten Spiellogik.
- **Eigenständiges Leveldesign:** Alle Levels wurden **vollständig eigenständig entworfen, getestet und balanciert**. Automatisch erzeugte oder KI-basierte Vorschläge wurden verworfen, da sie häufig unlösbar, unausgewogen oder spielmechanisch ungeeignet waren.